Name:_____

CSE341 Autumn 2019, Midterm Examination
October 28, 2019

Please do not turn the page until 8:30.

Rules:

- The exam is closed-book, closed-note, etc. except for *two* sides of one 8.5x11in piece of paper.
- **Please stop promptly at 9:20.**
- There are **100 points**, distributed **unevenly** among **5** 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.
- We will be scanning the exam to grade it. If you put an answer to a question in a non-intuitive spot, leave us a note. When in doubt, labeling where your answer is will never hurt.
- 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. (**27** points) This problem uses this datatype binding for rivers, where a river is either a source, or a fork where two other rivers join to make one larger river.

```
datatype river = Source of string | Fork of river * river
```

While different cultures have different river naming algorithms, usually the river formed at a fork has the name of the longer river of the two which come together. However, sometimes there's a tie. For this problem, we will consider the name of a river to be the name of the longer fork, except in the cases of a tie: in that case we will consider the name of the river to be the concatenation the names of both other rivers. Let the length of a `Source` be one, and let the length of a `Fork` be one more than the longest of the two rivers that join.

It turns out that this happens: in Eastern Virginia, the Mat and the Ta river join to form the Matta river. Likewise, the Po and the Ni river join to form the Poni river. Finally, the Matta and Poni river join to form the Mattaponi river.

(a) Write the `river_name` function of type `river -> string`, which takes in a river, and gives back its name (according to the specification above). You will likely want a helper function, but you don't have to have one. Recall that the infix operator `^` concatenates two strings together, and that `@` appends two lists.

(b) Given an example `river` datatype, with exactly 5 sources named 'Mat', 'Ta', 'Po', 'Ni', and 'York', whose name is 'MatTaPoNi' as calculated by `river_name`.

(c) Define `filter_river_sources`, of type `(string -> bool) -> river -> string list`, which takes a function and a river, and returns a list containing all of the strings from the Sources of that river, for which the given function returns true.

Name:_____

Solution:

```
fun name_helper r =
  case r of
      Source s => (s,0)
    | Fork (r1,r2) =>
      let
          val (n1,d1) = name_helper r1
          val (n2,d2) = name_helper r2
      in
          if d1 > d2 then (n1,d1+1)
          else if d2 > d1 then (n2,d2+1)
          else (n1^n2,d1+1)
      end

fun river_name r =
  let
      val (n,_) = name_helper r
  in
      n
  end


val mattaponi = (Fork
                    (Source "York",
                     (Fork
                          (Fork (Source "Mat",Source "Ta"),
                           (Fork (Source "Po",Source "Ni"))))))


fun filter_river_sources f r =
  case r of
      Source n => if f n then [n] else []
    | Fork(r1,r2) =>
      (filter_river_sources f r1)@(filter_river_sources f r2)
```

Name:_____

This page is extra space for your work.

Name:_____

2. (**16** points) This question uses the following code, which has been annotated with line numbers:

```
1 fun f l =
2   case l of
3      [] => 0
4    | (_,a,b)::(c,_,_)::r => a+b+c+(f r)
5    | x :: _ => 4
6    | _ :: (_,_,_) :: _ => 5
7    |  _ => 99
```

(a) Is the recursive call on line 4 a tail call?

(b) What will happen if I try to compile and run this program?

(c) Suppose the only tool I have is to remove some of lines 4,5,6, or 7. Give one subsetset of lines 4,5,6, and 7 which I could remove, which would fix the issue from above.

(d) Given your modification in part (c), write down the type of function `f`.

(e) Give another different answer to question (c).

Solution:

(a) No

(b) I'll get a "match redundant" compiler error. I'm looking for something more specific than just "compiler error", I want something about "unused match" or "redundant match" or something.

(c) Possible answers are: [6,7], [4,5,6], [5,6], [4,6,7], [4,5], or perhaps others as well

(d) TODO (depends on what's above) `'a list -> int`

(e) See above

Name:_____

3. (**26** points) This problem uses the following binding. In this problem `either` is polymorphic over two types, represented with the type variables `'a` and `'b`.

```
datatype ('a,'b) either = left of 'a | right of 'b
```

(a) Write `map_either` of type `(('a -> 'c) * ('b -> 'c)) -> ('a,'b) either list -> 'c list`, which will apply either the first or the second function to the contents of each contained element of each either in the third argument, and will return the results in the original order. Use no helper functions.

(b) Provide an equivalent definition `map_either2` which does the same thing, but uses `List.map`. In addition, take all arguments in curried form. Recall that the type of `List.map` is `('a -> 'b) -> 'a list -> 'b list`.

(c) Write `map_left_to_right` of type `('a -> 'b) -> ('a,'b) either list -> 'b list` which, for each element in the list (which is the second argument), will either convert the underlying element from type `'a` to type `'b` using the first argument, or will simply produce the underlying element of type `'b`. The result list will be returned in the same order as the original second element. Use any helper functions you would like, including any you've previously defined in this problem.

Solution:

```
fun map_either (f, g) l =
  case l of
      (left a) :: r => (f a) :: map_either (f,g) r
    | (right b) :: r => (g b) :: map_either (f,g) r
    |  [] => []

fun map_either2 f g l =
  let
      fun elem_fn elem =
        case elem of
            left a => f a
          | right b => g b
  in
      List.map elem_fn l
  end

fun map_left_to_right f l =
  case l of
      (left a) :: r => (f a) :: map_left_to_right f r
    | (right b) :: r => b :: map_left_to_right f r
    | [] => []

fun map_left_to_right2 f l =
  map_either (f,(fn x => x)) l
```

Name:_____
This is extra space for your work.

4. (**9** points) Give the type of the following SML bindings (q1, q2, and q3):

   (a)
   ```
   fun q1 f (x,y) = (f x,f y)
   ```

   (b)
   ```
   fun q2 x y z = q1 z (x,x)
   ```

   (c)
   ```
   fun q3 a r =
     case r of
        [] => a
      | c :: d => q3 (q2 a 0 c) d
   ```

   Solution:

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

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

   (c) `'a * 'a  -> ('a * 'a -> 'a) list -> 'a * 'a`

Name:_____

5. (**22** points) For this question, you will be dealing with the module system. You're given a signature which describes an optional string type. Fill in two structures which implement this signature, which must be indistinguishable: no user accessing `opt` or `opt2` through the interface defined in `OPTSIG` should be able to tell the difference between them. `valOf` must use the `NoVal` exception to indicate the lack of a value.

```
signature OPTSIG =
sig
    type  string_option
    exception NoVal
    val Some : string -> string_option
    val None : string_option
    val isSome : string_option -> bool
    val valOf : string_option -> string
end

structure opt :> OPTSIG =
struct
datatype string_option = Some of string | None









end

structure opt2 :> OPTSIG =
struct
type string_option = string list









end
```

Name:_____

Solution:

```
signature OPTSIG =
sig
    type  string_option
    exception NoVal
    val Some : string -> string_option
    val None : string_option
    val isSome : string_option -> bool
    val valOf : string_option -> string
end

structure opt :> OPTSIG =
struct
datatype string_option = Some of string | None
exception NoVal
fun isSome opt =
  case opt of
      Some _ => true
   |  None => false
fun valOf opt =
  case opt of
      Some a => a
    | None => raise NoVal
end

structure opt2 :> OPTSIG =
struct
type string_option = string list
exception NoVal
fun Some x = [x]
val None = []
fun isSome opt =
  case opt of
      [] => false
    | _ => true
fun valOf opt =
  case opt of
      [x] => x
    | _ => raise NoVal
end
```

Name:_____

There are no possible points on this page. Your exam score will not change in *any way* due to what you put on this page. Ignore this page until you're done with the rest of the exam.

6. (**0** points) **Creactivity**: Briefly describe the main characters and plot of a childrens book about functional programming.

7. (**0** points) **Reflection**: After using SML for about 5 weeks: If you got to write a program today in any language, what would that language be and why? There is no right answer.

8. (**0** points) **Bonus Question**: Suggest a problem for the next 341 midterm.