CSC C24H3 S 2022 Final Examination
Duration — 3 hours
Aids allowed: none

Last Name: ______________________ **First Name:** ______________________

Student Number: ______________________ **UTORID:** ______________________

---

*Do **not** turn this page until you have received the signal to start.*
Good Luck!

This midterm consists of 7 questions on 11 pages (including this one). *When you receive the signal to start, please make sure that your copy is complete.*

- Legibly write your name, UTORID, and student number on this page.
- If you use any space for rough work, indicate clearly what you want marked.
- In all programming questions you may assume all input is valid.
- You do not need to write comments of any kind.

1: ______/20

2: ______/12

3: ______/12

4: ______/16

5: ______/10

6: ______/12

7: ______/ 8

TOTAL: ______/90

Question 1. [20 MARKS]

Recall the `Tree` data type from our exercise and the function `tfold` defined to work with it:

```
data Tree a = Leaf a | Node a (Tree a) (Tree a)
five = Node "one" (Node "two" (Leaf "three") (Leaf "four")) (Leaf "five")

tfold:: (a -> b) -> (a -> b -> b -> b) -> Tree a -> b
tfold f g (Leaf x) = f x
tfold f g (Node x left right) = g x (tfold f g left) (tfold f g right)
```

Your first task is to provide two implementations of the function `internal` such that `internal t` returns a list of values stored in the internal (non-leaf) nodes of `t`, in the preorder order. For example, `internal five` should return `["one","two"]`. **Also specify the type of** `internal`.

```
-- a recursive implementation
```

```
-- using a single call to tfold, no recursion, and no other higher order functions
```

Your second task is to provide two implementations of the function `tmap` such that `tmap f t` returns a tree obtained by applying `f` to every value stored in `t`. For example, `tmap length five` should return `Node 3 (Node 3 (Leaf 5) (Leaf 4)) (Leaf 4)`. **Also specify the type of** `tmap`.

```
-- a recursive implementation
```

```
-- using a single call to tfold, no recursion, and no other higher order functions
```

Question 2. [12 MARKS]

The Haskell standard library has an algebraic data type `Either` defined as:

```
data Either a b = Left a | Right b
```

A common use case is putting two element types in the same list. For example, although we can't directly put integers and strings in the same list, we can:

```
mixedList :: [Either Integer String]
mixedList = [Left 3, Right "hi", Left 1, Left 4, Right "bye"]
```

Your task is to provide two implementations of a function `sep` which separates an input "mixed" list into two lists, each containing elements of one of the two types. For example, the call `sep mixedList` should return `([3,1,4],["hi","bye"])`. **Also specify the type of** `sep`.

```
-- a recursive implementation
```

```
-- using a single call to foldr, no recursion, and no other higher order functions
```

Question 3. [12 MARKS]

The Haskell standard library has an algebraic data type `Maybe` defined as:

```
data Maybe a = Nothing | Just a
```

In this question we change our data type `Tree` so that it contains `Maybe`s, as follows:

```
data MTree a = MLeaf (Maybe a) |
               MNode (Maybe a) (MTree a) (MTree a)
tree = MNode (Just 1) (MLeaf Nothing) (MNode (Just 2) (MLeaf (Just 3)) (MLeaf Nothing))
```

Your first task is to implement a function `toList` such that `toList t` returns a list of `Just` values from `t` in a preorder order. For example, `toList tree` should return `[1,2,3]`. You may want to define a helper function. **Also specify the type of** `toList`.

Your second task is to implement a function `maybeTreeMap` such that `maybeTreeMap f t` returns an `MTree` that results from applying `f` to `Just` values in `t`. For example, `maybeTreeMap (\x->x+10) tree` should return the tree

```
MNode (Just 11) (MLeaf Nothing) (MNode (Just 12) (MLeaf (Just 13)) (MLeaf Nothing))
```

You may want to define a helper function. **Also specify the type of** `maybeTreeMap`.

Question 4. [16 MARKS]

Your task is to design and implement our Haskell "Maybe Tree"s in Java.

1. Define an interface `Maybe<T>` and two classes that implement it, `Nothing<T>` and `Just<T>`. The interface must declare these methods (and the classes must implement them):
   (a) `getValue` which returns the value of type `T` stored in the `Maybe` object, or `null` in case of a `Nothing` object;
   (b) `listify` which returns an empty `List` when called on a `Nothing` object and a `List` containing the value of type `T` when called on a `Just` object; and
   (c) `mapply` which takes a `Function<T,T2>` (a function that takes values of type `T` and returns values of type `T2`), and which returns a new `Maybe<T2>` object that contains the result of applying the function to the value stored in the `Maybe` object. Notice that applying any function to a `Nothing` object should return a `Nothing` object, and applying a function to a `Just<T>` object should return a corresponding `Just<T2>` object.

2. Define an interface `MaybeTree<T>` and two classes that implement in, `MaybeLeaf<T>` and `MaybeNode<T>`. The interface must declare these methods (and the classes must implement them). Note that they are just like our Haskell equivalents:
   (a) `toList` which returns a `List<T>` of non-`Nothing` values stored in the `MaybeTree<T>` object in a preorder order, and
   (b) `tmap` which takes a `Function<T,T2>`, and which returns a new `MaybeTree<T2>` — a tree that results from applying the function to every value stored in the `MaybeTree`.

Here is example usage of this design:

```
MaybeTree<Integer> tree = new MaybeNode<>(
  new Just<>(1),
  new MaybeLeaf<>(new Nothing<>()),
  new MaybeNode<>(new Just<>(2),
                  new MaybeLeaf<>(new Just<>(3)),
                  new MaybeLeaf<>(new Nothing<>())));
List<Integer> list = tree.toList();
MaybeTree<String> newTree = tree.tmap(x -> x.toString() + "!");
```

Output of printing `tree`, `list`, and `newTree` above (you do NOT need to provide the `toString` methods):

```
MaybeNode [
  Just 1,
  MaybeLeaf Nothing,
  MaybeNode [
    Just 2,
    MaybeLeaf Just 3,
    MaybeLeaf Nothing]]
[1, 2, 3]
MaybeNode [
  Just 1!,
  MaybeLeaf Nothing,
  MaybeNode [
    Just 2!,
    MaybeLeaf Just 3!,
    MaybeLeaf Nothing]]
```

You will find the following `Function<T,R>` method useful:

Modifier and Type	Method and Description
R	`apply(T t)` Applies this function to the given argument.

Question 4. (CONTINUED)

Question 4. (CONTINUED)

Question 5. [10 MARKS]

Recall our representation of natural numbers in Prolog from our lab.

```
zero, s(zero), s(s(zero)), ...
```

Your task is to implement two versions of subtraction predicates in Prolog.

Implement `subtract(+X, +Y, ?Z)` iff $Z = X - Y$ for natural numbers X, Y, Z represented as above.

In the second version we have `subtract(?X, ?Y, ?Z)`, i.e., X and Y are not always instantiated (inputs). Though, you may assume that at least two of the three variables are fully instantiated. Think carefully about making sure your solution terminates — both for the first answer and if the user asks for more answers.

If your solution in the previous part is also good for this part (unlikely!), you can just write "same as previous part".

Question 6. [12 MARKS]

Recall our work with trips in Prolog in the last lab.

```
plane(to, ny, 100).
plane(ny, london, 200).
plane(london, bombay, 500).
plane(london, oslo, 50).
plane(bombay, katmandu, 100).
boat(oslo, stockholm, 100).
boat(stockholm, bombay, 1000).
boat(bombay, maldives, 1000).
```

Your first task is to recall our solution for the `trip` predicate.

```
% trip(?X, ?Y, ?C) iff there is a trip from X to Y that costs C.
```

Your second task is to define the predicate `trip_via(?X, ?Y, ?Via)` which succeeds iff there is a trip from `X` to `Y` that passes through `Via`. Note that `Via` could also be the first or last place on the journey.

Finally, we want to add to our original solution and keep track of each leg of the trip. Define a predicate `trip(?X, ?Y, ?C, ?Trip)` which succeeds iff there is a trip `Trip` from `X` to `Y` that costs `C`, where `Trip` is a list of all places visited on the journey, in order. For example,

```
?- trip(to, bombay, C, Trip).
C = 800,
Trip = [to, ny, london, bombay] ;
C = 1450,
Trip = [to, ny, london, oslo, stockholm, bombay] ;
false.
```

Question 7. [8 marks]

Consider the following Prolog program I use to make daily decisions:

```
needToDo(monday, cscc24).
needToDo(wednesday, cscc24).
needToDo(monday, abcc23).
needToDo(tuesday, abcc23).
needToDo(wednesday, abcc23).

like(cscc24).
dontLike(abcc23).

needToWatchTV(wednesday).
needToWatchTV(thursday).
needToWatchTV(friday).

procrastinate(Day, Task) :- needToDo(Day, Task), dontLike(Task).
procrastinate(Day, Task) :- needToDo(Day, Task), needToWatchTV(Day).
procrastinate(Day, Task) :- needToDo(Day, Task), needToDo(Day, Task2),
                            Task \= Task2, like(Task2).
```

While this program gives correct solutions, it also produces duplicates. List all solutions, in order, to the following query:

```
?- procrastinate(When, abcc23).
```

Now rewrite the predicate `procrastinate` so it produces all solutions as before, but with no duplicates.

Extra page.