Organization of Programming Languages — Midterm (T. I/18–19)

Directions: This examination is divided into 2 periods. Your midterm grade will be calculated as

$$\frac{x + y/5}{T},$$

where x is your score from the first period, y is your score from the second period, and T is our target total. We're aiming to use $T = 42$. Anything above 100% is extra credit.

- ▷ To upload your work, zip up the files and upload it to Canvas.
- ▷ Your Scala code must strictly follow the functional style. No loops, no mutable variables (no `var`), no mutable collections.
- ▷ Unless stated otherwise, use built-in functions when possible. The goal is to be short, concise, and efficient.

Period I: Hand in before 1:59pm.

- ▻ Work on this examination individually—no consultation with other people is permitted.
- ▻ **Hand in only 3 problems**; pick your best 3. ..or else, we'll pick the worst 3 for you.
- ▻ You can use anything from class and may access it online. You can also use the Internet to look up scala doc or look at basic tutorials. Just don't copy/look at someone else's solutions.
- ▻ Feel free to copy/reuse your own code from this term (e.g., lecture exercises and/or assignments).

Period II: Hand in before 11:59pm.

- ▻ Work in groups of 3–4 people (we're preassigning the group for you).
- ▻ You may *not* discuss it with other groups.
- ▻ **Hand in *all* 5 problems.**
- ▻ Each person hands in the solutions resulting from our coll
- ▻ You'll individually hand in solutions compiled from the collective wisdom of your group. Your solution doesn't have to be identical to your peers'.

(Q1) Iterator flatMap in Python. [10 points] *Save your work in* `iterflatmap.py`. The flatMap function is a mainstay of functional programming. This problem will ask you to reinvent it for Python. Here's a typical signature of the function, rendered in Scala style:

```
def flatMap[A, B](xs: List[A], f: A => List[B]): List[B]
```

When called with `flatMap(xs, f)`, it applies `f` to each element of `xs` (in that order) and returns the concatenation of their results.

In this problem, however, there are several nontraditional aspects:

- You're going to write flatMap in Python;
- `xs` won't be a list; it is an iterator;
- `f` will return an iterator; and
- your function is expected to return an iterator.

To help navigate this, we're providing a `mypy` stub:

```
def flat_map(xs: Iterator[A], f: Callable[[A], Iterator[B]]) -> Iterator[B]
```

Your goal is to implement this function. Your implementation *must not* convert the output of `f` into a list nor store/buffer the output from `f`. As an iterator, your function must return the entries as soon as they're available.

Note I: The input iterator `xs` can be extremely long or possibly infinite; your code must work in this case.

Note II: Your implementation must pass `mypy` with no errors or warnings.

Note III: You must *not* use anything from `itertools` or `collections`.

(Q2) Extreme Tic-Tac-Toe. [10 points] *Save your work in* `IteratorTTT.java`. Halloween is a good excuse to be curious. For example, one could wonder what all the endgame positions look like in the game of Tic-Tac-Toe on an n-by$n$ board. Luckily, we have friends on the Autonomouse Cat Machine (ACM) team, who has graciously provided us with a piece of code that performs this computation.

Check out their creation in the file `TicTacToe.java`. In it, you'll (unsurprisingly) find a public class `TicTacToe`. Please look around.

You'll find that the coding style stinks, but fortunately, you don't need to understand how it works. The only things you need to know are the following:

- The class's constructor takes a single number, the size of the board n. One can use this code by writing `new TicTacToe(n)`, where n is the desired board size.
- You can actually run it by calling `.run()`.
- Whenever this code finds a new endgame position, it calls the method `reportFound`. This method currently prints out that endgame configuration to screen using `System.out.printf`.

Your Task: Inside `IteratorTTT.java`, you will come up with an implementation that wraps the `TicTacToe` class into a static method

```
public static Iterable<String> allEndgame(int n)
```

which returns an iterable of all endgame configurations for an n-by-n board as produced by the `TicTacToe` class. You must *not* modify the original implementation nor could you replicate their logic. Here's how you might approach it (the code below is the starter code):

- Instead of printing to screen, modify the behavior of `TicTacToe` by inheriting it and overriding the `reportFound` method. Because you can override this, you can make it do whatever you so choose. We've already started this for you in the inner class `EnhancedTTT`.
- Remember what you did in Assignment 1 to report all the numbers in a (deeply) nested list?
- You may wish to use Threads.
- You're allowed to write more (internal) classes as you see fit.

Importantly, as an Iterable, your implementation must provide the entries as soon as they're available.

```
public class IteratorTTT {

    static class EnhancedTTT extends TicTacToe {
        // TODO: feel free to change the constructor
        EnhancedTTT(int n) {
            super(n);
        }

        @Override
        void reportFound(String sol) {
            // TODO: Do the right thing
        }
    }


    public static Iterable<String> allEndgame(int n) {
        // TODO: implement me using the set up above
        EnhancedTTT enhancedTTT = new EnhancedTTT(n);

        return null;
    }
}
```

(Q3) Your Own flatMap in Scala. [10 points] *Save your work in* `Reinvent.scala`. You will write a familiar list function and use it to solve another task. All your implementations for this problem will go inside `object Reinvent`. You do not need to extend App but may do so if you wish.

a) **(8 points)** Implement a function `flatMap` with the following signature:

```
def flatMap[A, B](xs: List[A], f: A => List[B]): List[B]
```

The function takes a list `xs` of type `List[A]`, and a function `f` that accepts an element of type `A` and returns a list `List[B]` (notice `B` may be different from `A`). `flatMap` returns a new list resulting from applying the function `f` to each element of `xs` and concatenating the results, retaining the input's ordering. As an example:

```
val foo = (x: Int) => if (x < 0) Nil else List(1, x, x*x)
// foo has type Int => List[Int]
val xs = List(3,-4,-5,2)
val ys = flatMap(xs, foo) // ys: List(1, 3, 9, 1, 2, 4)
```

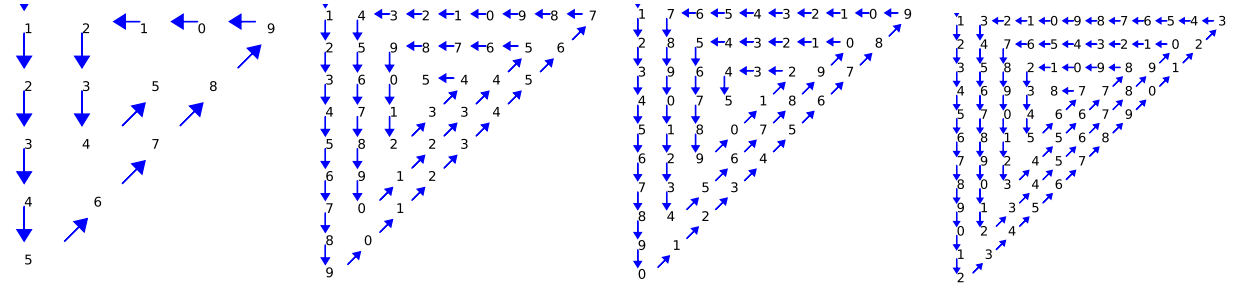
SPECIAL REQUIREMENTS: Do *not* use the built-in `flatMap`, `map`, or `flatten`. Your implementation must be tail recursive.

b) **(2 points)** Write a function `def posNeg(n: Int): List[Int]` that takes an integer $n \geqslant 1$ and returns the list `List(1, -1, 2, -2,..., n, -n)`. *Hint:* Use your `flatMap` function above. If your `flatMap` doesn't work properly, use the built-in `flatMap` for this part.

c) **(Bonus)** Assume that the function `f` runs in time proportional to the the length of the sequence it returns. Upgrade your `flatMap` so that it runs in $O(m)$ time, where m is the length of the output list.
(*Hint:* Only form a list by using `Nil` or the cons-ing operator (`::`). That is, do *not* use ++, ++:, :::.)

(Q4) Magic Triangle. [10 points] *Save your work in* `MagicTriangle.scala`. All your implementations for this problem will go inside `object MagicTriangle`. You do not need to extend App but may do so if you wish.

In celebration of Halloween this year, we're planning to build magic triangles, which are believed to ward off bad spirits. A size-N magic triangle is constructed on an N-by-N array where only half of it is filled, so it has the shape of a right triangle. Each leg has length N. Example gadgets of sizes $N = 5$, $N = 9$, $N = 10$, and $N = 12$ are shown below.



The numbers in the right triangle are filled counterclockwise. Starting by filling the vertical leg in the first column, we then move up diagonally to the right until we hit the top row—and move leftward until we hit our starting point. At this point, we make our way down again. We repeatedly fill the triangle in this circular motion until the area is full.

The area is actually filled with numbers. We start counting from 1. When the number reaches 9, the next number is reset to 0. (It helps to look at the examples below.) As a result of this process, all the numbers are between 0 and 9 (inclusive).

You Task: Implement a function `def triangle(n: Int): List[List[Int]]` which returns a list of lists representing the rows of the magic triangle of size n. Some examples are given below:

```
triangle(5) == List(
  List(1,2,1,0,9),
  List(2,3,5,8),
  List(3,4,7),
  List(4,6),
  List(5))
```

```
triangle(9) == List(
  List(1,4,3,2,1,0,9,8,7),
  List(2,5,9,8,7,6,5,6),
  List(3,6,0,5,4,4,5),
  List(4,7,1,3,3,4),
  List(5,8,2,2,3),
  List(6,9,1,2),
  List(7,0,1),
  List(8,0),
  List(9))
```

```
triangle(10) == List(
  List(1,7,6,5,4,3,2,1,0,9),
  List(2,8,5,4,3,2,1,0,8),
  List(3,9,6,4,3,2,9,7),
  List(4,0,7,5,1,8,6),
  List(5,1,8,0,7,5),
  List(6,2,9,6,4),
  List(7,3,5,3),
  List(8,4,2),
  List(9,1),
  List(0))
```

NOTE: Your function must not be excessively slow. Importantly, it must adhere to the functional style.

(*Hint:* How can you construct a size n triangle from a (variant of) size $n - 1$ triangle?)

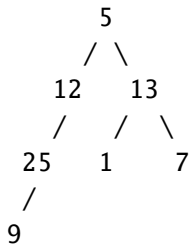
(Q5) Layers Of A Tree. [10 points] *Save your work in* `Layers.scala`. All your implementations for this problem will go inside `object Layers`. Whether you extend App or not is up to you.

You'll work with binary trees, which can be easily defined in Scala. The following code defines a "sum" type corresponding to the notion that a binary tree either is empty or contains a key and two children (`l` for left and `r` for right). For simplicity, the keys are assumed to be integers (`Long`).

```
sealed trait Tree
case object Empty extends Tree
case class Node(key: Long, l: Tree = Empty, r: Tree = Empty) extends Tree
```

As was done in class, this definition allows us to define, for example, the tree on the right using the code below:

```
val tr =
  Node(5,
    Node(12,
      Node(25,
        Node(9),
        Empty),
      Empty),
    Node(13,
      Node(1),
      Node(7)))
```



We say that the "root" of the tree is at layer 0. The immediate children of the root are at layer 1. The immediate children of tree nodes on layer 1 are at layer 2. In the example above, we have:

Layer: 0	1	2	3
5	12, 13	25, 1, 7	9

Your Task: Implement a function `def getLayer(tr: Tree, k: Int): List[Long]` which returns the list of numbers on layer `k` when read from left to right. Using `tr` from the example above, we get

```
getLayer(tr, 1) == List(12, 13)
getLayer(tr, 2) == List(25, 1, 7)
getLayer(tr, 3) == List(9)
getLayer(tr, 4) == List()
```

You're free to write helper functions.

(Bonus) Write `getLayer` in the continuation-passing style (CPS).