Functional & Parallel Programming — Test I (T. I/23–24)

Directions:

- ▷ This examination consists of *four* problems. You have *almost* one day to complete it. The examination is due back on Canvas by 10am on Friday Oct 20, 2023. We don't expect you to use the entire 20+ hours, though. The test should be do-able in 3–4 hours. In the same zip file as this handout, you will also find 4 starter files, one per problem.
- ▷ This is a take-home exam with an open-everything policy; you can look at your notes, previous assignments, search the Internet, etc. However, you must cite your sources. Also, you are *not* allowed to talk to/collaborate with another person (or AI). This automatically disallows ChatGPT, co-pilot, and similar services. Asking questions on or using answers from online study-help sites (e.g., Chegg) is considered cheating. But using stackoverflow and similiar sites in *read-only mode* is okay.
- ▷ To upload your work, zip up the files and upload it to Canvas. There are only 4 files to include in the zip (one per problem). Don't include any other files.
- ▷ Your Scala code must strictly follow the functional style, just like in your assignments so far. 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. Most subtasks should not need more than 5–10 lines of code.
- ▷ Use `#funpar-text` for questions/clarification/etc. For private questions, direct message or email me.
- ▷ Good luck!

(Q1) Standard Techniques [100 points] *Save your work in* `BasicsApp.scala`. You will write a few Scala methods. There may be special requirements. All your implementations will go inside `object BasicsApp`, which already extends `App`.

(a) **(25 points)** Implement a function `flatMap` with the following signature:

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

In words, the function is curried. The first argument is a function `f` that accepts an element of type `A` and yields a list `List[B]`. The argument after that takes a list `xs: List[A]`. Notice `B` may be different from `A`. Like the built-in `flatMap`, our `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.

Examples:

```
val foo = (x: Int) => if x < 0 then Nil else List(1, x, x*x)
// foo has type Int => List[Int]
val xs = List(3,-4,-5,2)
val ys = flatMap(foo)(xs) // ys: List(1, 3, 9, 1, 2, 4)
val bar = (x: String) => List(x.length/2.0, 0.5 + x.length)
// bar has type String => List[Double]
val zs = List("hi", "hello")
val ts = flatMap(bar)(zs) // ts: List(1.0, 2.5, 2.5, 5.5)
```

SPECIAL REQUIREMENTS: Do *not* use the built-in `flatMap`, `map`, or `flatten`. Your implementation must be tail recursive. You can only use pattern matching, cons-ing `::`, and concatenation operators.

(b) **(Extra-Credit: +10 points)** While satisfying the special requirements above, upgrade your `flatMap` to run in time that is linear in the resulting list. That is to say, if `flatMap` returns a list of length m, it must run within $O(m)$ time. (*Hint:* Avoid concatenation.)

(c) **(25 points)** Write a function `def negNegPos(n: Int): List[Int]` that takes an integer `n` $\geqslant 1$ and returns the list `List(-1, -1, 1, -2, -2, 2, -3, -3, 3, ..., -n, -n, n)`. (*Hint:* Use your `flatMap` function above. If your `flatMap` isn't working properly, use the built-in `flatMap` for this part.)

(d) **(25 points)** Write a function

```
def winnerOf(xs: Vector[String]): (String, Int)
```

that takes in a vector of names ("votes") and returns a pair of `String` and `Int`, indicating the most-popular name and its frequency (i.e., the winner by popularity). For example:

```
val votes = Vector("Dan", "Alice", "Bob", "Bob", "Alice", "Bob", "Dylan")
val winner = winnerOf(votes) // => ("Bob", 3)
```

We promise that your code will only be tested on inputs where there is only one unique winner.

(e) **(25 points)** Write a function

```
def nonTrailingZeros(n: BigInt): Int
```

that takes as input a number `n` and returns the number of non-trailing zeros (i.e., zeros that aren't those appearing consecutively at the end). Examples below will be useful.

```
nonTrailingZeros(20300) // ==> 1
nonTrailingZeros(4040005010L) // ==> 5
nonTrailingZeros(123400000) // => 0
nonTrailingZeros(BigInt("10000100001010000000000")) // => 9
```

SPECIAL REQUIREMENTS: Your solution must *not* be recursive. For this task, your code has to work directly on the string and can use only `foldLeft` and/or `foldRight` on it (see starter code). You can't use other functions (e.g., `.length`, `.count`).

(*Hint:* Complete the code in the starter pack.)

(Q2) Flat Polynomials [5×20 points] *Save your work in* `FlatPolyApp.scala`. All your implementations will go inside `object FlatPolyApp`, which already extends `App`.

Basic Facts. Any single-variable polynomial, when fully expanded, is simply a sum of terms. For instance, the polynomial $p(x) := 3x^2 + 5x + 100x^3$, as shown, is a sum of three terms: $3 \times x^2$, $5 \times x$, and $100 \times x^3$. Furthermore, such a term can be easily expressed in terms of a *coefficient* and an *exponent* (abstracting out the variable name). For example, the term $3x^2$ has a coefficient of 3 and an exponent of 2 (the power of x).

With this knowledge, we define a case class `Term` as `sealed case class Term(coeff: Int, expo: Int)` to represent a term—and define the following class for a fully-expanded polynomial (aka. flat polynomial):

```
class FlatPoly(val terms: List[Term])
```

The `val` keyword here makes `terms` a member variable. The class stores a list of terms that makes up the polynomial. In this view, the example $p(x)$ from above is `List(Term(3, 2), Term(5, 1), Term(100, 3))`.

To help you get started, we have implemented a few things in the starter code:

- a `.toString` method for `FlatPoly` so that the polynomial can be nicely displayed.
- a `unary_-` method so that if `q` is a `FlatPoly` representing a polynomial $q(x)$, then the expression `-q` yields a `FlatPoly` representing $-q(x)$. In more detail, the operator negates all the coefficients of $q(x)$.

You are to extend the `FlatPoly` class to support the features below. Each requires adding a new method to `FlatPoly` and should be just a few lines of code. You can and are encouraged to use suitable built-in methods. Our running examples involve

```
val a = FlatPoly(List(Term(2, 3), Term(3, 2), Term(5, 3))) // 2x^3 + 3x^2 + 5x^3
val b = FlatPoly(List(Term(1, 1), Term(1, 2), Term(2, 0))) // x + x^2 + 2
```

(a) Write a method `def eval(x: Double): Double`. If `q` is a `FlatPoly` representing a polynomial $q(x)$, then `q.eval(x)` yields the value one would obtained from plugging in `x` into $q(x)$. For example:

```
a.eval(1) // => 10.0
a.eval(2.5) // => 128.125
```

(b) Write a method `def diff: FlatPoly`. If `q` is a `FlatPoly` representing a polynomial $q(x)$, then `q.diff` yields a polynomial that is equal to the derivative $\frac{d}{dx}q(x)$. Remember that $\frac{d}{dx}x^n = nx^{n-1}$.As an example:

```
a.diff // => 6x^2 + 6x + 15x^2
b.diff // => 2x + 1
```

(Notice that differentiating a constant term yields 0 and should be dropped from the resulting expression.)

(c) Write a method `def normalize: FlatPoly` that returns a simplified polynomial in the following sense:

- For each power, there is only one term with that particular power. That is, collect and combine terms with the same power. For example, $2x^3$ and $5x^3$ will be combined into $7x^3$.
- The terms are listed in increasing power.
- There are no terms with coefficient 0.

Examples:

```
a.normalize // => 3x^2 + 7x^3
b.normalize // => 2 + x + x^2
```

(d) Make `FlatPoly` support addition. Specifically, if `p` and `q` are `FlatPoly` representing polynomials $p(x)$ and $q(x)$, respectively, then writing `p + q` should result in a `FlatPoly` that is equal to $p(x) + q(x)$. Note: In your method's output, as long as all the terms are there, they can appear in any order. As an example:

```
val r = a + b // => 2x^3 + 3x^2 + 5x^3 + x + x^2 + 2
```

(e) Make `FlatPoly` support multiplication. Specifically, if `p` and `q` are `FlatPoly` representing polynomials $p(x)$ and $q(x)$, respectively, then writing `p * q` should result in a `FlatPoly` that is equal to $p(x) \times q(x)$. Note: In your output, as long as all the terms are there, they can appear in any order. As an example:

```
val r = a * b // => 6x^2 + 17x^3 + 10x^4 + 7x^5
```

(We have collected and combined similar terms in this example, but your code isn't required to.)

(Q3) **Streams/Lazy Lists [100 points]** *Save your work in* `StreamApp.scala`. All your implementations will go inside `object StreamApp`, which already extends `App`.

You will write a few methods involving constructing/manipulating lazy-lists (aka. streams).

(a) **(25 points)** A number n is *odd-odds* if n has an odd number of odd digits. For example, 1 has one odd digit, which meets the criteria. But 123 has two odd digits, so it is not odd-odds. In fact, between 1 and 150, the only numbers that meet this condition are

1, 3, 5, 7, 9, 10, 12, 14, 16, 18, 21, 23, 25, 27, 29, 30, 32, 34, 36, 38, 41, 43, 45, 47, 49, 50, 52, 54, 56, 58, 61, 63, 65, 67, 69, 70, 72, 74, 76, 78, 81, 83, 85, 87, 89, 90, 92, 94, 96, 98, 100, 102, 104, 106, 108, 111, 113, 115, 117, 119, 120, 122, 124, 126, 128, 131, 133, 135, 137, 139, 140, 142, 144, 146, 148

Write a method

```
def onlyOddOdds(s: LazyList[Long]): LazyList[Long]
```

that takes a `LazyList[Long]` called `s` and returns a lazy list such that only numbers from `s` that are odd-odds are retained. In other words, the function filters out numbers that do *not* meet the condition. We promise that `s` will only contain non-negative integers.

(b) **(25 points)** Define a value `val allOddOdds: LazyList[Long]` so that this is a lazy list of all positive odd-odds. Use the list of all odd-odds up to 150 above for your testing. (*Hint:* Create a lazy list of all positive integers of right type—and use the previous part. Otherwise, do you see any pattern in the list of numbers above?)

(c) **(25 points)** You will write code that produces a lazy list from any given linear recurrence. As is standard, let $a_1, a_2, \ldots, a_k$ and $b_1, b_2, \ldots, b_k$ be integers. The $b_1, \ldots, b_k$ values are the first k terms, and the $a_1, \ldots, a_k$ values are the coefficients for generating the next term in the recurrence. They are taken in as a list of pairs to make sure there are as many a's as b's. We'll use these numbers to generate an (infinite) sequence as follows:

$$g_1 = b_1, \quad g_2 = b_2, \quad \ldots, \quad g_k = b_k$$
$$g_n = a_1 g_{n-1} + a_2 g_{n-2} + a_3 g_{n-3} + \ldots a_k g_{n-k} \qquad \text{for} \quad n > k$$

As an example, using $a_1 = a_2 = b_1 = b_2 = 1$, we have the Fibonacci recurrence: $f_1 = f_2 = 1$ and $f_n = f_{n-1} + f_{n-2}$. As another example, using $a_1 = 2, a_2 = 3, b_1 = 2, b_2 = 1$, we have the recurrence $g_1 = 2, g_2 = 1$, and $g_n = 2g_{n-1} + 3g_{n-2}$, so the first few terms are $2, 1, 8, 19, 62, 181, 548, 1639, 4922, 14761, \ldots$. You will write a method

```
def recurrence(r: Vector[(Int, Int)]): LazyList[Long]
```

that takes as input a vector `r` storing $[(a_1, b_1), (a_2, b_2), \ldots, (a_k, b_k)]$ and returns a lazy list that is the sequence $g_1, g_2, g_3, \ldots$.
(*Hint:* Some useful bits appear in the starter code. You're free to change how `next` looks.)

(d) **(Extra-Credit: +10 points)** In your `recurrence` solution, implement your "next" function that uses no additional bookkeeping. That is to say, the signature for `next` is simply

```
def next(s: LazyList[Long]): LazyList[Long]
```

(e) **(25 points)** Define a value `val fibOddOdds: LazyList[Long]` so that this is a lazy list of all odd-odds Fibonacci numbers. For reference, the first 20 numbers in this list are

1, 1, 3, 5, 21, 34, 89, 144, 377, 610, 2584, 17711, 46368, 75025, 121393, 196418, 317811, 514229, 832040, 1346269

(*Hint:* Use the previous parts or write a function to generate a lazy list of Fibonacci directly.)

(Q4) Tree x Heap [100 points] *Save your work in* `TreapApp.scala`. All your implementations will go inside `object TreapApp`, which already extends `App`.

Background. A *Treap* (Tree + Heap) is (yet) another balanced binary search tree (BST). The main idea is to make the tree heap-ordered on a "random" set of numbers, in addition to being a BST on the user-provided keys. This constrains the tree to be "bushy" most of the time.

We keep a Treap like a normal BST. It's either empty or is a node storing a left subtree, a key, an associated value, and a right subtree, like so:

```
sealed trait Treap[+K, +V]
case object Empty extends Treap[Nothing, Nothing]
case class Node[K, V](
    left: Treap[K, V], key: K, value: V, right: Treap[K, V]
) extends Treap[K, V]
```

The *priority* of a node is given by a "hash" function `h` applied to the node's key. A simple `h`, which you'll use in this question, is provided in the starter code.

The Treap satisfies the following invariants:

(BST Ordering) The keys in the nodes are BST-ordered. This is the standard BST ordering invariant.

(Heap Ordering) The priorities satisfy max-heap[1] ordering. That is to say, the priority of a node is at least the priorities of its children (standard numerical comparison).

You will complete and extend our partial implementation of Treap in the following ways:

(a) **(25 points)** Use the exception jumping trick to write a method

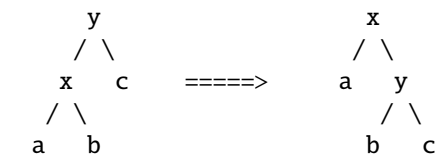
```
def getByValue[K, V](rt: Treap[K, V], v: V): Option[(K, V)]
```

that finds the smallest key whose value matches the given `v`. The function returns an `Option`—`None` if no matching values are found, and `Some` of a pair of key, value if a match is found. In other words, it performs a look up by value. (This subtask assumes only BST ordering and nothing about being a Treap)

SPECIAL REQUIREMENTS: Use exception jumping so it can return right away when a match is found.

(b) **(25 points)** Write a method `def avgDepth[K, V](rt: Treap[K, V]): Double` that computes the average depth of the leaf nodes. The depth of a leaf is the number of edges on the path from the root to that leaf, including both the root and the leaf themselves. Therefore, a tree with one node by itself has depth 1; the empty tree has depth 0. The goal here is to find the number of leaf nodes and the sum of their depths, so a simple average $(\sum d_i)/n$ can be computed. (This subtask assumes only that `rt` is a binary tree and nothing about BST/Treap.)

(c) **(25 points)** Rewrite the `rebalance` method. The starter code contains an `insert` function that guarantees BST ordering and already calls `rebalance` in places where rebalancing may be needed. However, the supplied `rebalance` does nothing. Rewrite it to ensure the heap ordering invariant. (Don't modify `insert`.) The figure below illustrates what rebalancing has to do. If on the left tree, `h(y) < h(x)`, then transforming it to the right tree fixes the problem, without breaking BST ordering. (*Hint:* the other bad case is symmetrical, so `rebalance` should be like 3–4 lines of match-case code and runs in constant time.)



(d) **(25 points)** Write a method `def satisfiesMHIV[K, V](rt: Treap[K, V]): Boolean` that determines whether or not `rt` satisfies the (max) heap-ordering invariant. Once you've written this method, use it to test your `rebalance` method. We aren't crazy about running time for this one, but it shouldn't be excessively slow. There is a simple linear-time implementation.

[1] This means the root has the largest priority value.