Exercise 1: Lambda Calculus

Church numerals are a representation of natural numbers using only functions. In this encoding, a number `n` is represented by a higher-order function that maps any function `f` to its `n`-fold composition. For examples, in Scheme–, `0`, `1`, `2` and `3` are represented as follows:

```
(val zero  (lambda (f x) x)
(val one   (lambda (f x) (f x))
(val two   (lambda (f x) (f (f x)))
(val three (lambda (f x) (f (f (f x))))

...

))))
```

Exercise 1.1

Implement a function that converts church numerals to primitive integers.

Exercise 1.2

Implement multiplication on church numerals.

Exercise 2: Lisp (from 2017's final)

After having written your Lisp interpreter, you decide that you want to implement some transformations over your Lisp programs. The first transformation you consider is partial derivation of mathematical formulas. Namely, you want terms of the shape `(∂ x expr)` to be transformed to some new term `expr2` that corresponds to the derived formula of `expr`. `expr2` does NOT need to be simplified.

`expr2` $= \frac{\partial}{\partial x}$ `expr`

For example, the term `(∂ x (+ y (+ 2 x)))` should be rewritten to `(+ 0 (+ 0 1))`.

You can assume the expr only contains numbers, symbols, sums `(+ e1 e2)` and multiplications `(* e1 e2)`.

Rules for partial derivation

$\frac{\partial}{\partial x} x = 1$

$\frac{\partial}{\partial x} n = 0$ **for** $n \in \mathbb{Z}$

$\frac{\partial}{\partial x} y = 0$

$\frac{\partial}{\partial x} \left(f(x) + g(x)\right) = \frac{\partial}{\partial x} f(x) + \frac{\partial}{\partial x} g(x)$

$\frac{\partial}{\partial x} \left(f(x) \cdot g(x)\right) = \left(\frac{\partial}{\partial x} f(x) \cdot g(x)\right) + \left(f(x) \cdot \frac{\partial}{\partial x} g(x)\right)$

Exercise 2.1: Scala implementation

In order to achieve the above task, complete the following function:

```
def derive(x: Symbol, expr: Any): Any = ???
```

Note that in the example above, you would call

```
derive('x, List('+, 'y, List('+, 2, 'x)))
```

and the expected result is

```
List('+, 0, List('+, 0, 1))
```

Remember that Lisp terms are of type `Any`. Composite terms have type `List[Any]`, variable names are instances of `Symbol` (you can use `==` between symbols) and numbers are of type `Int`. You may use any method in the Scala standard library (see appendix for most typical ones).

Hint: Remember that you can use a default case in a pattern match.

```
def derive(x: Symbol, expr: Any): Any =
```

Exercise 2.2: Translation into Lisp

You now want to implement the same functionality within the Lisp language itself. This means **your input now consists of a Lisp list**. Complete the following function definition (using the syntactic sugar proposed in the assignment):

```
(def (derive x expr) ??? rest)
```

The example above would this time correspond to

```
(derive 'x (list '+ 'y (list '+ 2 'x)))
```

and your function should output

```
(list '+ 0 (list '+ 0 1))
```

The available Lisp API can be found in the appendix. You may use the predicates `isCons?` and `isNil?` to determine whether a given term respectively corresponds to an instance of `cons` or `nil`. You are **strongly encouraged** to use `list` to create lists and `nth` to get the n^{th} element of a list (starting with `0`). If you wish to use a function that was not defined in the appendix, provide its definition as well.

Hint: Remember Lisp (non-)syntax of prefix notation, for example, comparisons are written as (= a b).

Note: Indent your code properly when answering this question, syntax is very important here.

```
(def (derive x expr)
  (

  )
  rest)
```

Exercise 3: Streams (from 2017's final)

We all know the common use case of replacing all occurrences of a substring `pattern` within a string `text` by another substring `replacement`. If we represent strings as `List[Char]`, we can write the signature of such a method as

```
def replaceAll(text: List[Char], pattern: List[Char],
    replacement: List[Char]): List[Char] = /* omitted */
```

(do *not* implement this method)

Even though this method seems very well defined on its own, there are several interpretations that we can give to partially overlapping patterns and replacements. For example, what should

```
replaceAll(List('a', 'a', 'a'), List('a', 'a'), List('b'))
```

return? It could return `List('b', 'a')` or `List('a', 'b')`.

In this exercise, we use the interpretation that patterns are tested *left-to-right*, so that the left-most match wins. Under that interpretation, `List('b', 'a')` must be returned.

Moreover, we choose a slightly unusual interpretation that, after replacing a substring, we consider the replacement itself, together with the rest of the input string, as candidate for further replacement. This means that

```
replaceAll(List('a', 'a', 'b', 'a', 'b', 'a'), List('a', 'b', 'a'), List('b', 'a'))
```

results in `List('a', 'b', 'b', 'a')`.

Indeed, we first find the substring `aba` at index 1. We immediately send the left part of the input (until index 1, i.e., `a`) to the output, so it will not be reconsidered for replacement. We then replace `aba` by `ba`, resulting in a new remaining input `baba`. In that substring, we identify a new occurrence of `aba` (after the first `b`, which is emitted to the output) which must be replaced. Note that this subsequence was not part of the original input. This results in the new remaining input `ba`. At this point, no new replacement is possible. The final result is therefore `a` followed by `b` followed by `ba`, i.e., `abba`.

In this exercise, the `replacement` is always guaranteed to be strictly shorter than the `pattern` (i.e., `replacement.size < pattern.size`).

We can generalize this problem to streams, by admitting `input: Stream[Char]` and returning an output `Stream[Char]`:

```
def replaceAll(input: Stream[Char], pattern: List[Char],
    replacement: List[Char]): Stream[Char] = ???
```

Note that the `pattern` and `replacement` remain `List`s. The resulting stream will be infinite if and only if `input` is infinite.

Exercise 3.1

Implement the helper function

```
def testStartsWith(input: Stream[Char], pattern: List[Char]): Option[Stream[Char]]
```

which tests whether `input` starts with `pattern`, and if yes, returns the remaining of the `input`.

Examples:

```
testStartsWith(Stream('a', 'b', 'c'), List('a', 'b')) == Some(Stream('c'))
testStartsWith(Stream('a', 'b', 'c'), List('b', 'c')) == None
testStartsWith('a' #:: 'b' #:: infiniteStream, List('a')) == Some('b' #:: infiniteStream)
```

When `replacement.size >= pattern.size`, your implementation can behave in an arbitrary way (e.g., it could throw or infinitely loop). For all other inputs, `testStartsWith` must terminate in finite time, whether `input` is finite or infinite.

To get full points in this subquestion, your implementation of `testStartsWith` should complete in `O(n)` time, where `n` is `pattern.size` (assuming the input stream can produce each new element that you request in constant time).

```
def testStartsWith(input: Stream[Char], pattern: List[Char]): Option[Stream[Char]] =
```

Exercise 3.2

Implement the function `replaceAll` for streams defined above.

Your implementation must be able to handle both finite and infinite input streams. This means that `replaceAll(input, pat, repl).take(n).toList` must complete in finite time for all `input`, `pat`, `repl` and `n`, such that `replacement.size < pattern.size`, even if `input` is infinite.

There are no constraints on the complexity of `replaceAll`: even an inefficient implementation can get full points (as long as it satisfies the previous paragraph).

```
def replaceAll(input: Stream[Char], pattern: List[Char],
    replacement: List[Char]): Stream[Char] =
```

Exercise 3.3

We now further generalize the problem to a list of pairs `(pattern, replacement)`:

```
def replaceAllMany(input: Stream[Char],
    patternsAndReplacements: List[(List[Char], List[Char])]): Stream[Char] = ???
```

which, at each step, tries all the patterns in `patternsAndReplacements` and applies the first one that matches, then starts over with the resulting new input.

For all pair `(pat, repl)` in `patternsAndReplacements`, it is guaranteed that `repl.size < pat.size`.

Examples:

```
replaceAll(Stream('a', 'a', 'b', 'b'), List(
    (List('a', 'b'), List('c')),
    (List('a', 'a'), List('d'))
)) == Stream('d', 'b', 'b')
```

Note that the second pattern won because it applies *earlier in the input string*.

```
replaceAll(Stream('d', 'a', 'a', 'a', 'b', 'b'), List(
    (List('e', 'b'), List('c')),
    (List('a', 'a', 'a'), List('d', 'e')),
    (List('d', 'd'), List('z'))
)) == Stream('d', 'd', 'c', 'b')
```

Here, note that after applying the second pattern at index 1, we obtained the new input `debb`, which subsequently matched the first pattern. The `dd` in the final string was not replaced by the third pattern, because the first `d` was sent to the output right from the start, given that the first match was found at index 1.

Implement `replaceAllMany`. Similarly to `replaceAll`, your implementation must be able to handle both finite and infinite input streams. There are otherwise no constraints on its complexity.

Please answer on the following page.

```
def replaceAllMany(input: Stream[Char],
    patternsAndReplacements: List[(List[Char], List[Char])]): Stream[Char] =
```

Exercise 4: The State Monad (from 2015's final)

It is the year-end holidays. Instead of revising your exams, you want to put your newly learned Scala skills to the test, by programming a little Mario-like game.

You start off with a fairly simple design. A game is defined by running the `runGame` function on a list of `GameAction`, where you execute every action using a `doAction` function, which returns a log message describing the action you undertook:

```
def runGame(ls: List[GameAction]): List[String] = {
  "game starts" :: (ls map doAction)
}
```

There are for now 4 actions in your game:

```
sealed abstract class GameAction
case object EatMushroom extends GameAction
case object JumpOnTortoise extends GameAction
case object SkidOnBanana extends GameAction
case object FallFromBridge extends GameAction

def doAction(ga: GameAction): String = ga match {
  case EatMushroom    => "ate a mushroom"
  case JumpOnTortoise => "jumped on tortoise"
  case SkidOnBanana   => "skid on a banana"
  case FallFromBridge => "fell from bridge"
}
```

Keeping Score

You now want to add the ability to count the score accumulated through the various actions. You come up with the following scoring scheme:

- eating a mushroom awards 5 points.
- jumping on a tortoise awards 10 points.
- skidding on a banana costs 5 points.
- falling from a bridge costs 10 points.

Keeping score with mutable variables (1 point)

Suppose, **for once**, that you can use mutable variables/imperative style code, and that the sequence of actions you have is:

```
val myActions = List(EatMushroom, JumpOnTortoise, SkidOnBanana)
```

We can then implement a *side effectful* function `doAction2`, that updates the variable `score`:

```
var score = 0
val gameRun = "game starts" :: (myActions map doAction2)
//score here equals 10
```

Implement `doAction2`, such that it updates the state variable `score`:

```
def doAction2(ga: GameAction): String = ???
```

The State Monad

But we want to be purely functional in our implementation. So rather than `runGame` returning `List[String]`, it is better to have it return some sort of game state, which contains information regarding the score. To this effect, you choose to use the state monad. We represent this monad as follows:

```
final class StateM[A] private (private val makeProgress: Int => (A, Int)) {
  def runState(initState: Int): (A, Int) = makeProgress(initState)
}
```

Essentially, the state monad is a function that takes an initial state (of type `Int`, for the score) and returns a value of type `A`, and a new state. In the above signature, the `private` keywords signify, respectively, that:

- we cannot use the constructor of `StateM` outside of the class and its companion object.
- we cannot refer to `makeProgress` outside the body of `StateM` and its companion object. If we want to execute it, we will instead use the `runState` method.

To allow access and to the state, the state monad comes with two helper functions, `getState` and `putState`:

```
object StateM {
  def getState: StateM[Int] = new StateM((s: Int) => (s, s))
  def putState(newState: Int): StateM[Unit] = new StateM((s: Int) => ((), newState))
}
```

Essentially, in a purely functional world, the state monad allows us to safely store and update state variables.

Coming back to our game, we can now convert `runGame` so that it returns a `StateM[List[String]]`: the main computation of this function is the log message, but it also carries some state that represents the score:

```
def runGame3(ls: List[GameAction]): StateM[List[String]] =
  ls.foldLeft[StateM[List[String]]](unit(List("game starts"))) {
    case (prevState, newAction) =>
      for {
        msg <- prevState
        msg2 <- doAction3(newAction)
      } yield {
        msg ++ List(msg2)
      }
  }
```

We run the initial game by giving it an initial state of `0` points:

```
runGame3(ls).runState(0)
```

Desugaring For Comprehensions (2 points)

As a warmup, desugar the above for expression in terms of `flatMap`, `map` and `withFilter`:

```
for {
  msg <- prevState
  msg2 <- doAction3(newAction)
} yield {
  msg ++ List(msg2)
}
```

The monadic operations: unit (2 points)

Recall that a monad must implement the `unit` and `flatMap` operations (that respect the monad laws). Implement `unit` for `StateM`:

```
// inside the StateM companion object
def unit[A](a: A): StateM[A] = ???
```

Calling `unit(a)` returns a function that takes some initial state `s`, and returns the element `a`, along with the initial state. It simply forwards the state through.

The monadic operations: flatMap (3 points)

Implement `flatMap`, or the monadic bind, for `StateM`:

```
// inside StateM[A], the class
def flatMap[B](f: A => StateM[B]): StateM[B] = ???
```

Calling `sm flatMap f` returns an instance of the state monad where, upon passing an initial state, the computation of `sm` is executed yielding a pair `(a, s2)`, of type `(A, Int)`. The value `a` is then passed to `f`, yielding an instance of `StateM[B]`. We finally pass `s2` to this instance.

Hint 1: here is the implementation of `map` for `StateM`:

```
// in StateM[A], the class
def map[B](f: A => B): StateM[B] = {
  val res = { (s: Int) =>
    val (a, s2) = makeProgress(s)
    (f(a), s2)
  }
  new StateM(res)
}
```

Hint 2: follow the types!

Actions that keep scores (2 points)

Finally, we need to implement `doAction3`. Convert the `doAction` function so that it now returns a `StateM[String]` that respects the above scoring function. Remember, you are only allowed to use the monad operations (`unit`, `map`, `flatMap`), `getState` and `putState`:

```
def doAction3(ga: GameAction): StateM[String] = ???
```