Assignment 7 – Streams, types and lazyness

Advanced programming paradigms

Question 1 – *A covariant stack*

In this first question, you are given the following trait:

```scala
trait Stack[A] {
  def push(elem: A): ...
  def top: A
  def pop: Stack[A]
}
```

Starting with this code, you have to define a covariant definition of this stack with its implementation. For this, you will follow the idea we used in assignment 2 for the integer set.

(a) Begin by declaring two case classes, `EmptyStack` and `ElemStack`. The former is used to model an empty stack and the later is used to store elements. Do not forget we are only working with immutable data structures!
Hint: the constructor of the `ElemStack` requires to take an element as well as another `Stack`.

(b) Implement the methods in those classes.

(c) Considering that the same person implements everything, where is the best location to implement the `push` method? Why?

....

....

....

....

(d) When you are done, the following code should execute without failures

```scala
// Construction, pop and toString
val a = EmptyStack().push("hello").push("world").push("it's fun").pop
assert(a.toString() == "world,hello,EmptyStack()")

// Getting top
val b = EmptyStack().push(1).push(3)
assert(b.top == 3)

// Variance checks
class Foo
class Bar extends Foo
val c: Stack[Bar] = EmptyStack().push(new Bar()).push(new Bar())
assert(c.top.isInstanceOf[Bar] == true)
assert(c.top.isInstanceOf[Foo] == true)

// Variance check 2
val d: Stack[Foo] = EmptyStack().push(new Bar()).push(new Bar())
assert(d.top.isInstanceOf[Foo])
```

Question 2 – *Implicit conversions*

You will develop a class `TestConv` that enables easy temperature conversions. It uses a a common type for every temperature – `Temperature` – which should be declared `abstract` and `sealed` (all the subclasses declared in this source file are the only subclasses allowed). Two sub-types of this class are `Celsius` and `Kelvin`.

(a) Write the required *implicit conversions* and the required code to allow the following code to run:

```
val a: Celsius = 30
val b: Kelvin = 30
val c: Kelvin = Celsius(10)
val d: Celsius = c
val e: Temperature = d

println(a) // Should print "30° C"
println(b) // Should print "30° K"
```

(b) What will you get on the console if you try to print `e`?

(c) Why is it interesting to have the `Temperature` class? Explain!

Question 3 – *The Fibonacci sequence using infinite streams*[1]

To get used to infinite sequences, you will define the infinite stream of the Fibonacci sequence. In this sequence, the first two elements are 0 and 1. Each subsequent element is obtained by summing the two preceding elements.

To define this stream, start by writing the function `addStream`, which takes two integer streams in argument and returns a new stream whose elements are the sum of the elements of the two input streams. Its prototype is as follows:

```
def addStream(s1: Stream[int], s2: Stream[int]): Stream[int]
```

Using this function, the definition of the Fibonacci sequence takes a single line.

Question 4 – *Streams of prime numbers*

(a) Define an stream for integers which start at a given value and that continues up to infinity.

(b) Use this `Stream` to define the sequence of the prime numbers, using the techniques known as the *Sieves of Eratosthenes*. This very ancient technique works as follows:
Start with an infinite sequence of integers, starting with 2:

```
2 3 4 5 6 7 8 9 10 11 12 13 14 15...
```

We then take the head of the list (which is **always** a prime number) and we eliminate all the multiples of this number. We obtain a new infinite sequence:

```
2 3 5 7 9 11 13 15 17...
```

We recursively apply this function to the rest of the list (here 3 is the new head), obtaining:

```
3 5 7 11 13 17...
```