Coursework 8 (Scala, Regular Expressions

This coursework is worth 10%. It is about regular expressions, pattern matching and polymorphism. The first part is due on 30 November at 11pm; the second, more advanced part, is due on 7 December at 11pm. You are asked to implement a regular expression matcher. Make sure the files you submit can be processed by just calling `scala <<filename.scala>>`.

Important: Do not use any mutable data structures in your submission! They are not needed. This excludes the use of `ListBuffer`s, for example. Do not use `return` in your code! It has a different meaning in Scala, than in Java. Do not use `var`! This declares a mutable variable. Make sure the functions you submit are defined on the "top-level" of Scala, not inside a class or object. Also note that the running time of each part will be restricted to a maximum of 360 seconds.

Disclaimer!!!!!!!!

It should be understood that the work you submit represents your own effort! You have not copied from anyone else. An exception is the Scala code I showed during the lectures or uploaded to KEATS, which you can freely use.

Part 1 (6 Marks)

The task is to implement a regular expression matcher that is based on derivatives of regular expressions. The implementation can deal with the following regular expressions, which have been predefined in the file re.scala:

r	$::=$	$\mathbf{0}$	cannot match anything
	$\mid$	$\mathbf{1}$	can only match the empty string
	$\mid$	c	can match a character (in this case c)
	$\mid$	$r_1 + r_2$	can match a string either with r_1 or with r_2
	$\mid$	$r_1 \cdot r_2$	can match the first part of a string with r_1 and then the second part with r_2
	$\mid$	r^*	can match zero or more times r

Why? Knowing how to match regular expressions and strings fast will let you solve a lot of problems that vex other humans. Regular expressions are one of the fastest and simplest ways to match patterns in text, and are endlessly useful for searching, editing and analysing text in all sorts of places. However, you need to be fast, otherwise you will stumble over problems such as recently reported at

- http://stackstatus.net/post/147710624694/outage-postmortem-july-20-2016
- https://vimeo.com/112065252
- http://davidvgalbraith.com/how-i-fixed-atom/

Tasks (file re.scala)

(1a) Implement a function, called *nullable*, by recursion over regular expressions. This function tests whether a regular expression can match the empty string.

$$\begin{array}{lcl} nullable(\mathbf{0}) & \stackrel{\text{def}}{=} & false \\ nullable(\mathbf{1}) & \stackrel{\text{def}}{=} & true \\ nullable(c) & \stackrel{\text{def}}{=} & false \\ nullable(r_1 + r_2) & \stackrel{\text{def}}{=} & nullable(r_1) \vee nullable(r_2) \\ nullable(r_1 \cdot r_2) & \stackrel{\text{def}}{=} & nullable(r_1) \wedge nullable(r_2) \\ nullable(r^*) & \stackrel{\text{def}}{=} & true \end{array}$$

[1 Mark]

(1b) Implement a function, called *der*, by recursion over regular expressions. It takes a character and a regular expression as arguments and calculates the derivative regular expression according to the rules:

$$\begin{array}{lcl} der\ c\ (\mathbf{0}) & \stackrel{\text{def}}{=} & \mathbf{0} \\ der\ c\ (\mathbf{1}) & \stackrel{\text{def}}{=} & \mathbf{0} \\ der\ c\ (d) & \stackrel{\text{def}}{=} & \textit{if } c = d \textit{ then } \mathbf{1} \textit{ else } \mathbf{0} \\ der\ c\ (r_1 + r_2) & \stackrel{\text{def}}{=} & (der\ c\ r_1) + (der\ c\ r_2) \\ der\ c\ (r_1 \cdot r_2) & \stackrel{\text{def}}{=} & \textit{if } nullable(r_1) \\ & & \textit{then } ((der\ c\ r_1) \cdot r_2) + (der\ c\ r_2) \\ & & \textit{else } (der\ c\ r_1) \cdot r_2 \\ der\ c\ (r^*) & \stackrel{\text{def}}{=} & (der\ c\ r) \cdot (r^*) \end{array}$$

For example given the regular expression $r = (a \cdot b) \cdot c$, the derivatives w.r.t. the characters a, b and c are

$$\begin{array}{lcll} der\ a\ r & = & (\mathbf{1} \cdot b) \cdot c & (= r') \\ der\ b\ r & = & (\mathbf{0} \cdot b) \cdot c & \\ der\ c\ r & = & (\mathbf{0} \cdot b) \cdot c & \end{array}$$

Let r' stand for the first derivative, then taking the derivatives of r' w.r.t. the characters a, b and c gives

$$\begin{array}{lcll} der\ a\ r' & = & ((\mathbf{0} \cdot b) + \mathbf{0}) \cdot c & \\ der\ b\ r' & = & ((\mathbf{0} \cdot b) + \mathbf{1}) \cdot c & (= r'') \\ der\ c\ r' & = & ((\mathbf{0} \cdot b) + \mathbf{0}) \cdot c & \end{array}$$

One more example: Let r'' stand for the second derivative above, then taking the derivatives of r'' w.r.t. the characters a, b and c gives

$$\begin{array}{lcl} \mathit{der}\ a\ r'' & = & ((\mathbf{0} \cdot b) + \mathbf{0}) \cdot c + \mathbf{0} \\ \mathit{der}\ b\ r'' & = & ((\mathbf{0} \cdot b) + \mathbf{0}) \cdot c + \mathbf{0} \\ \mathit{der}\ c\ r'' & = & ((\mathbf{0} \cdot b) + \mathbf{0}) \cdot c + \mathbf{1} \end{array}$$

Note, the last derivative can match the empty string, that is it is *nullable*.

[1 Mark]

(1c) Implement the function *simp*, which recursively traverses a regular expression from the inside to the outside, and simplifies every sub-regular-expression on the left (see below) to the regular expression on the right, except it does not simplify inside *-regular expressions.

$$\begin{array}{lcl} r \cdot \mathbf{0} & \mapsto & \mathbf{0} \\ \mathbf{0} \cdot r & \mapsto & \mathbf{0} \\ r \cdot \mathbf{1} & \mapsto & r \\ \mathbf{1} \cdot r & \mapsto & r \\ r + \mathbf{0} & \mapsto & r \\ \mathbf{0} + r & \mapsto & r \\ r + r & \mapsto & r \end{array}$$

For example the regular expression

$$(r_1 + \mathbf{0}) \cdot \mathbf{1} + ((\mathbf{1} + r_2) + r_3) \cdot (r_4 \cdot \mathbf{0})$$

simplifies to just r_1. **Hints:** Regular expressions can be seen as trees and there are several methods for traversing trees. One of them corresponds to the inside-out traversal. Also remember numerical expressions from school: there you had exprssions like $u + \ldots + (1 \cdot x) * \ldots (z + (y \cdot 0)) \ldots$ and simplification rules that looked very similar to rules above. You would simplify such numerical expressions by replacing for example the $y \cdot 0$ by 0, or $1 \cdot x$ by x, and then look if more rules are applicable. If you organise this simplification in an inside-out fashion, it is always clear which rule should applied next.

[1 Mark]

(1d) Implement two functions: The first, called *ders*, takes a list of characters and a regular expression as arguments, and builds the derivative w.r.t. the list as follows:

$$\begin{array}{lcl} \mathit{ders}\ (\mathit{Nil})\ r & \stackrel{\text{def}}{=} & r \\ \mathit{ders}\ (c :: cs)\ r & \stackrel{\text{def}}{=} & \mathit{ders}\ cs\ (\mathit{simp}(\mathit{der}\ c\ r)) \end{array}$$

Note that this function is different from *der*, which only takes a single character.

The second function, called *matcher*, takes a string and a regular expression as arguments. It builds first the derivatives according to *ders* and after that tests whether the resulting derivative regular expression can match the empty string (using *nullable*). For example the *matcher* will produce true given the regular expression $(a \cdot b) \cdot c$ and the string *abc*.

[1 Mark]

(1e) Implement the function *replace* $r\ s_1\ s_2$: it searches (from the left to right) in the string s_1 all the non-empty substrings that match the regular expression r—these substrings are assumed to be the longest substrings matched by the regular expression and assumed to be non-overlapping. All these substrings in s_1 matched by r are replaced by s_2. For example given the regular expression

$$(a \cdot a)^* + (b \cdot b)$$

the string $s_1 = aabbbaaaaaaabaaaaabbaaaabb$ and replacement the string $s_2 = c$ yields the string

$$ccbcabcaccc$$

[2 Marks]

Part 2 (4 Marks)

You need to copy all the code from `re.scala` into `re2.scala` in order to complete Part 2. Parts (2a) and (2b) give you another method and datapoints for testing the *der* and *simp* functions from Part 1.

Tasks (file re2.scala)

(2a) Write a **polymorphic** function, called *iterT*, that is **tail-recursive**(!) and takes an integer n, a function f and an x as arguments. This function should iterate f n-times starting with the argument x, like

$$\underbrace{f(\ldots(f}_{n\text{-times}}(x)))$$

More formally that means *iterT* behaves as follows:

$$iterT(n, f, x) \stackrel{\text{def}}{=} \begin{cases} x & \text{if } n = 0 \\ f(iterT(n-1, f, x)) & \text{otherwise} \end{cases}$$

Make sure you write a **tail-recursive** version of *iterT*. If you add the annotation `@tailrec` (see below) your code should not produce an error message.

```
import scala.annotation.tailrec

@tailrec
def iterT[A](n: Int, f: A => A, x: A): A = ...
```

You can assume that *iterT* will only be called for positive integers $0 \leq n$. Given the type variable `A`, the type of f is `A => A` and the type of x is `A`. This means *iterT* can be used, for example, for functions from integers to integers, or strings to strings, or regular expressions to regular expressions.

[2 Marks]

(2b) Implement a function, called *size*, by recursion over regular expressions. If a regular expression is seen as a tree, then *size* should return the number of nodes in such a tree. Therefore this function is defined as follows:

$$\begin{array}{lcl} size(\mathbf{0}) & \stackrel{\text{def}}{=} & 1 \\ size(\mathbf{1}) & \stackrel{\text{def}}{=} & 1 \\ size(c) & \stackrel{\text{def}}{=} & 1 \\ size(r_1 + r_2) & \stackrel{\text{def}}{=} & 1 + size(r_1) + size(r_2) \\ size(r_1 \cdot r_2) & \stackrel{\text{def}}{=} & 1 + size(r_1) + size(r_2) \\ size(r^*) & \stackrel{\text{def}}{=} & 1 + size(r) \end{array}$$

You can use *size* and *iterT* in order to test how much the 'evil' regular expression $(a^*)^* \cdot b$ grows when taking successive derivatives according the letter a and then compare it to taking the derivative, but simlifying the derivative after each step. For example, the calls

```
size(iterT(20, (r: Rexp) => der('a', r), EVIL))
size(iterT(20, (r: Rexp) => simp(der('a', r)), EVIL))
```

produce without simplification a regular expression of size of 7340068 after 20 iterations, while the one with simplification gives just 8.

[1 Mark]

(2c) Write a **polymorphic** function, called *fixpT*, that takes a function f and an x as arguments. The purpose of *fixpT* is to calculate a fixpoint of the function f starting from the argument x. A fixpoint, say y, is when $f(y) = y$ holds. That means *fixpT* behaves as follows:

$$fixpT(f, x) \stackrel{\text{def}}{=} \begin{cases} x & \text{if } f(x) = x \\ fixpT(f, f(x)) & \text{otherwise} \end{cases}$$

Make sure you calculate in the code of *fixpT* the result of $f(x)$ only once. Given the type variable `A` in *fixpT*, the type of f is `A => A` and the type of x is `A`. The file `re2.scala` gives two example function where in one the fixpoint is 1 and in the other it is the string *a*.

[1 Mark]

Background

Although easily implementable in Scala, the idea behind the derivative function might not so easy to be seen. To understand its purpose better, assume a regular expression r can match strings of the form $c :: cs$ (that means strings which start with a character c and have some rest, or tail, cs). If you now take the derivative of r with respect to the character c, then you obtain a regular expressions that can match all the strings cs. In other words, the regular expression $der\ c\ r$ can match the same strings $c :: cs$ that can be matched by r, except that the c is chopped off.

Assume now r can match the string abc. If you take the derivative according to a then you obtain a regular expression that can match bc (it is abc where the a has been chopped off). If you now build the derivative $der\ b\ (der\ a\ r))$ you obtain a regular expression that can match the string c (it is bc where b is chopped off). If you finally build the derivative of this according c, that is $der\ c\ (der\ b\ (der\ a\ r)))$, you obtain a regular expression that can match the empty string. You can test this using the function nullable, which is what your matcher is doing.

The purpose of the simp function is to keep the regular expression small. Normally the derivative function makes the regular expression bigger (see the SEQ case and the example in (2b)) and the algorithm would be slower and slower over time. The simp function counters this increase in size and the result is that the algorithm is fast throughout. By the way, this algorithm is by Janusz Brzozowski who came up with the idea of derivatives in 1964 in his PhD thesis.

https://en.wikipedia.org/wiki/Janusz_Brzozowski_(computer_scientist)