Coursework 8 (Scala, Regular Expressions

This coursework is worth 10%. It is about regular expressions and pattern matching. The first part is due on 30 November at 11pm; the second, more advanced part, is due on 7 December at 11pm. The second part is not yet included. For the first part 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 `ListBuffers`, 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.

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:

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

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.

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

[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:

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

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

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

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

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

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

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

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.

[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:

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

The second, 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 string $s_2 = c$ yields the string

$$ccbcabcaccc$$

[2 Mark]

Part 2 (4 Marks)

Coming soon.