Main Part 3 (Scala, 6 Marks)

"Java is the most distressing thing to happen to computing since MS-DOS."
— Alan Kay, the inventor of object-oriented programming

This part is about a regular expression matcher described by Brzozowski in 1964. The background is that "out-of-the-box" regular expression matching in mainstream languages like Java, JavaScript and Python can sometimes be excruciatingly slow. You are supposed to implement a regular expression matcher that is much, much faster.

Important

- Make sure the files you submit can be processed by just calling `scala <<filename.scala>>` on the command line.[1] Use the template files provided and do not make any changes to arguments of functions or to any types. You are free to implement any auxiliary function you might need.
- **Do not leave any test cases running in your code because this might slow down your program!** Comment out test cases before submission, otherwise you might hit a time-out.
- Do not use any mutable data structures in your submissions! They are not needed. This means you cannot create new `Array`s or `ListBuffer`s, for example.
- Do not use `return` in your code! It has a different meaning in Scala than in Java. It changes the meaning of your program, and you should never use it.
- Do not use `var`! This declares a mutable variable. Only use `val`!
- Do not use any parallel collections! No `.par` therefore! Our testing and marking infrastructure is not set up for it.

Also note that the running time of each part will be restricted to a maximum of 30 seconds on my laptop.

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.

[1] All major OSes, including Windows, have a command line. So there is no good reason to not download Scala, install it and run it on your own computer. Just do it!

Reference Implementation

This Scala assignment comes with a reference implementation in form of a `jar`-file. This allows you to run any test cases on your own computer. For example you can call Scala on the command line with the option `-cp re.jar` and then query any function from the `re.scala` template file. As usual you have to prefix the calls with `M3` or import this object. Since some tasks are time sensitive, you can check the reference implementation as follows: if you want to know, for example, how long it takes to match strings of a's using the regular expression $(a^*)^* \cdot b$ you can query as follows:

```
$ scala -cp re.jar
scala> import M3._
scala> for (i <- 0 to 5000000 by 500000) {
  | println(f"$i: ${time_needed(2, matcher(EVIL, "a" * i))}%.5f secs.")
  | }
0: 0.00002 secs.
500000: 0.10608 secs.
1000000: 0.22286 secs.
1500000: 0.35982 secs.
2000000: 0.45828 secs.
2500000: 0.59558 secs.
3000000: 0.73191 secs.
3500000: 0.83499 secs.
4000000: 0.99149 secs.
4500000: 1.15395 secs.
5000000: 1.29659 secs.
```

Preliminaries

The task is to implement a regular expression matcher that is based on derivatives of regular expressions. Most of the functions are defined by recursion over regular expressions and can be elegantly implemented using Scala's pattern-matching. The implementation should 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 single 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 a string with zero or more copies of r

Why? Regular expressions are one of the simplest ways to match patterns in text, and are endlessly useful for searching, editing and analysing data in all sorts of places (for example analysing network traffic in order to detect security breaches). However, you need to be fast, otherwise you will stumble over problems such as recently reported at

- https://blog.cloudflare.com/details-of-the-cloudflare-outage-on-july-2-2019
- https://stackstatus.net/post/147710624694/outage-postmortem-july-20-2016
- https://vimeo.com/112065252
- https://davidvgalbraith.com/how-i-fixed-atom

Tasks (file re.scala)

The file `re.scala` has already a definition for regular expressions and also defines some handy shorthand notation for regular expressions. The notation in this document matches up with the code in the file as follows:

		code:	shorthand:
$\mathbf{0}$	$\mapsto$	`ZERO`	
$\mathbf{1}$	$\mapsto$	`ONE`	
c	$\mapsto$	`CHAR(c)`	
$\sum rs$	$\mapsto$	`ALTs(rs)`	
$r_1 + r_2$	$\mapsto$	`ALT(r1, r2)`	`r1 | r2`
$r_1 \cdot r_2$	$\mapsto$	`SEQ(r1, r2)`	`r1 ~ r2`
r^*	$\mapsto$	`STAR(r)`	`r.%`

The alternative regular expressions comes in two versions: one is binary (+ / `ALT`) and the other is n-ary ($\sum$ / `ALTs`). The latter takes a list of regular expressions as argument. In what follows we shall use *rs* to stand for lists of regular expressions. When the list is empty, we shall write $\sum[]$; if it is non-empty, we sometimes write $\sum[r_1, ..., r_n]$. The binary alternative can be seen as an abbreviation, that is $r_1 + r_2 \stackrel{\text{def}}{=} \sum[r_1, r_2]$. As a result we can ignore the binary version and only implement the n-ary alternative.

(1) Implement a function, called *nullable*, by recursion over regular expressions. This function tests whether a regular expression can match the empty string. This means given a regular expression it either returns true or false. The function *nullable* is defined as follows:

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

[0.5 Marks]

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

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

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

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

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

[1 Mark]

(3) We next want to simplify regular expressions: essentially we want to remove $\mathbf{0}$ in regular expressions like $r + \mathbf{0}$ and $\mathbf{0} + r$. However, our n-ary alternative takes a list of regular expressions as argument, we therefore need a more general "flatten" function, called `flts`. This function should analyse a list of regular expressions, say rs, as follows:

$$
\begin{array}{llcll}
1) & rs = [] & \stackrel{\text{def}}{=} & [] & \text{(empty list)} \\
2) & rs = \mathbf{0} :: rest & \stackrel{\text{def}}{=} & \text{flatten } rest & \\
3) & rs = (\sum rs_1) :: rest & \stackrel{\text{def}}{=} & rs_1 :::\ \text{flatten } rest & \\
4) & rs = r :: rest & \stackrel{\text{def}}{=} & r :: \text{flatten } rest & \text{(otherwise)}
\end{array}
$$

The first clause states that empty lists cannot be further flattened. The second removes the first **0** from the list and recurses. The third is when the first regular expression is an `ALTs`, then the content of this alternative should be spilled out and appended with the flattened rest of the list. The last case is for all other cases where the head of the list is not **0** and not an `ALTs`, then we just keep the head of the list and flatten the rest. [1 Mark]

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

$$
\begin{array}{lcl}
r \cdot \mathbf{0} & \mapsto & \mathbf{0} \\
\mathbf{0} \cdot r & \mapsto & \mathbf{0} \\
r \cdot \mathbf{1} & \mapsto & r \\
\mathbf{1} \cdot r & \mapsto & r \\
\sum [r_1, .., r_n] & \mapsto & \sum (\texttt{(flts + distinct)}[simp(r_1), .., simp(r_n)])
\end{array}
$$

The last case is as follows: first apply *simp* to all regular expressions $r_1, .., r_n$; then flatten the resulting list using `flts`; finally remove all duplicates in this list (this can be done in Scala using the function `_.distinct`). When you perform these operations, you end up with three cases, namely where the list is empty, contains a single element and "otherwise". These cases should be processed as follows

$$
\begin{array}{lcll}
\sum [] & \mapsto & \mathbf{0} & \\
\sum [r] & \mapsto & r & \\
\sum rs & \mapsto & \sum rs & \text{"otherwise"}
\end{array}
$$

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. **Hint:** Regular expressions can be seen as trees and there are several methods for traversing trees. One of them corresponds to the inside-out traversal, which is also sometimes called post-order traversal: you traverse inside the tree and on the way up you apply simplification rules. **Another Hint:** Remember numerical expressions from school times—there you had expressions 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 whether more rules are applicable. If you regard regular expressions as trees and if you organise the simplification in an inside-out fashion, it is always clear which simplification should be applied next. [1 Mark]

(5) 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:

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

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 for the regular expression $(a \cdot b) \cdot c$ and the string *abc*, but false if you give it the string *ab*. [1 Mark]

(6) 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:

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

You can use *size* in order to test how much the "evil" regular expression $(a^*)^* \cdot b$ grows when taking successive derivatives according the letter a without simplification and then compare it to taking the derivative, but simplify the result. The sizes are given in `re.scala`. [0.5 Marks]

(7) You do not have to implement anything specific under this task. The purpose here is that you will be marked for some "power" test cases. For example can your matcher decide within 30 seconds whether the regular expression $(a^*)^* \cdot b$ matches strings of the form $aaa \ldots aaaa$, for say 1 Million a's. And does simplification simplify the regular expression

```
SEQ(SEQ(SEQ(..., ONE | ONE) , ONE | ONE), ONE | ONE)
```

correctly to just `ONE`, where `SEQ` is nested 50 or more times?

[1 Mark]

Background

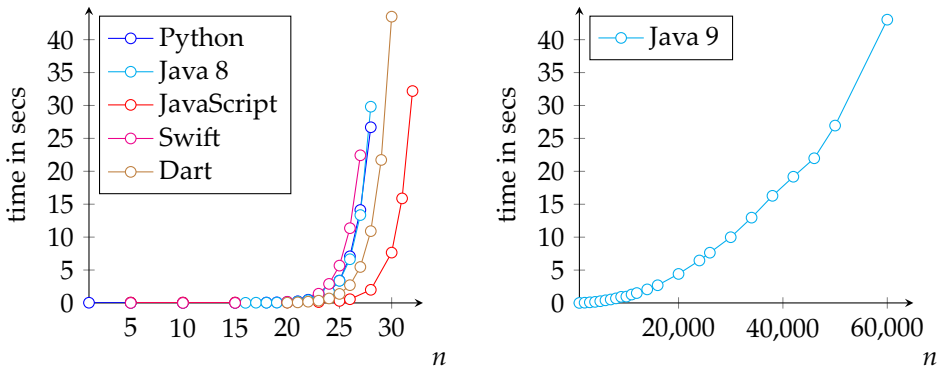
Although easily implementable in Scala (ok maybe the `simp` functions and `ALTs` needs a bit more thinking), the idea behind the derivative function might not so easy to be seen. To understand its purpose better, assume a regular expression r can match strings of the form $c :: cs$ (that means strings which start with a character c and have some rest, or tail, cs). If you take the derivative of r with respect to the character c, then you obtain a regular expression 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 whether this is indeed the case using the function nullable, which is what your matcher is doing.

The purpose of the *simp* function is to keep the regular expressions small. Normally the derivative function makes the regular expression bigger (see the SEQ case and the example in (2)) 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)

If you want to see how badly the regular expression matchers do in Java[2], JavaScript and Python with the "evil" regular expression $(a^*)^* \cdot b$, then have a look at the graphs below (you can try it out for yourself: have a look at the files `catastrophic9.java`, `catastrophic.js`, `catastrophic.py` etc on KEATS). Compare this with the matcher you have implemented. How long can a string of a's be in your matcher and still stay within the 30 seconds time limit? It should be muuuch better than your off-the-shelf matcher in your bog-standard language.

[2]Version 8 and below; Version 9 and above does not seem to be as catastrophic, but still much worse than the regular expression matcher based on derivatives.

Graph: $(a^*)^* \cdot b$ and strings $\underbrace{a \ldots a}_{n}$