Part 9 (Scala)

"[Google's MapReduce] abstraction is inspired by the map and reduce primitives present in Lisp and many other functional language."

— Dean and Ghemawat, who designed this concept at Google

This part is about the shunting yard algorithm by Dijkstra and a regular expression matcher by Brzozowski. The preliminary part (4%) is due on 11 December at 4pm; the core, more advanced part, is due on 15 January at 4pm. The preliminary part is about the Shunting Yard Algorithm that transforms the usual infix notation of arithmetic expressions into the postfix notation, which is for example used in compilers. In the core part, you are asked to implement a regular expression matcher based on derivatives of regular expressions. 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 an 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 commandline.[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`!

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

- 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.

Reference Implementation

This Scala assignment comes with three reference implementations in form of `jar`-files you can download from KEATS. 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 `CW9a`, `CW9b` and `CW9c`. 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 CW9c._
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.
```

Preliminary Part (4 Marks)

The *Shunting Yard Algorithm* has been developed by Edsger Dijkstra, an influential computer scientist who developed many well-known algorithms. This algorithm transforms the usual infix notation of arithmetic expressions into the postfix notation, sometimes also called reverse Polish notation.

Why on Earth do people use the postfix notation? It is much more convenient to work with the usual infix notation for arithmetic expressions. Most modern calculators (as opposed to the ones used 20 years ago) understand infix notation. So why on Earth? …Well, many computers under the hood, even nowadays, use postfix notation extensively. For example if you give to the Java compiler the expression $1 + ((2 * 3) + (4 - 3))$, it will generate the Java Byte code

```
ldc 1
ldc 2
ldc 3
imul
ldc 4
ldc 3
isub
iadd
iadd
```

where the command `ldc` loads a constant onto the stack, and `imul`, `isub` and `iadd` are commands acting on the stack. Clearly this is the arithmetic expression in postfix notation.

The shunting yard algorithm processes an input token list using an operator stack and an output list. The input consists of numbers, operators ($+$, $-$, $*$, $/$) and parentheses, and for the purpose of the assignment we assume the input is always a well-formed expression in infix notation. The calculation in the shunting yard algorithm uses information about the precedences of the operators (given in the template file). The algorithm processes the input token list as follows:

- If there is a number as input token, then this token is transferred directly to the output list. Then the rest of the input is processed.
- If there is an operator as input token, then you need to check what is on top of the operator stack. If there are operators with a higher or equal precedence, these operators are first popped off from the stack and moved to the output list. Then the operator from the input is pushed onto the stack and the rest of the input is processed.
- If the input is a left-parenthesis, you push it on to the stack and continue processing the input.
- If the input is a right-parenthesis, then you pop off all operators from the stack to the output list until you reach the left-parenthesis. Then you discharge the (and) from the input and stack, and continue processing the input list.
- If the input is empty, then you move all remaining operators from the stack to the output list.

Tasks (file postfix.scala)

(1) Implement the shunting yard algorithm described above. The function, called `syard`, takes a list of tokens as first argument. The second and third arguments are the stack and output list represented as Scala lists. The most convenient way to implement this algorithm is to analyse what the input list, stack and output list look like in each step using pattern-matching. The algorithm transforms for example the input

```
List(3, +, 4, *, (, 2, -, 1, ))
```

into the postfix output

```
List(3, 4, 2, 1, -, *, +)
```

You can assume the input list is always a list representing a well-formed infix arithmetic expression. [1 Mark]

(2) Implement a compute function that takes a postfix expression as argument and evaluates it generating an integer as result. It uses a stack to evaluate the postfix expression. The operators $+$, $-$, $*$ are as usual; / is division on integers, for example $7/3 = 2$. [1 Mark]

Task (file postfix2.scala)

(3/4) Extend the code in (7) and (8) to include the power operator. This requires proper account of associativity of the operators. The power operator is right-associative, whereas the other operators are left-associative. Left-associative operators are popped off if the precedence is bigger or equal, while right-associative operators are only popped off if the precedence is bigger. The compute function in this task should use `Long`s, rather than `Int`s. [2 Marks]

Core Part (6 Marks)

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`:

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

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)`	
$r_1 + r_2$	$\mapsto$	`ALT(r1, r2)`	`r1 | r2`
$r_1 \cdot r_2$	$\mapsto$	`SEQ(r1, r2)`	`r1` $\sim$ `r2`
r^*	$\mapsto$	`STAR(r)`	`r.%`

(5) 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:

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

[1 Mark]

(6) 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:

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

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

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

Note, the last derivative can match the empty string, that is it is *nullable*.
[1 Mark]

(7) 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.

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

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. **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 organise the simplification in an inside-out fashion, it is always clear which simplification should be applied next. [1 Mark]

(8) 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]

(9) 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* 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`. [1 Mark]

(10) 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, 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.

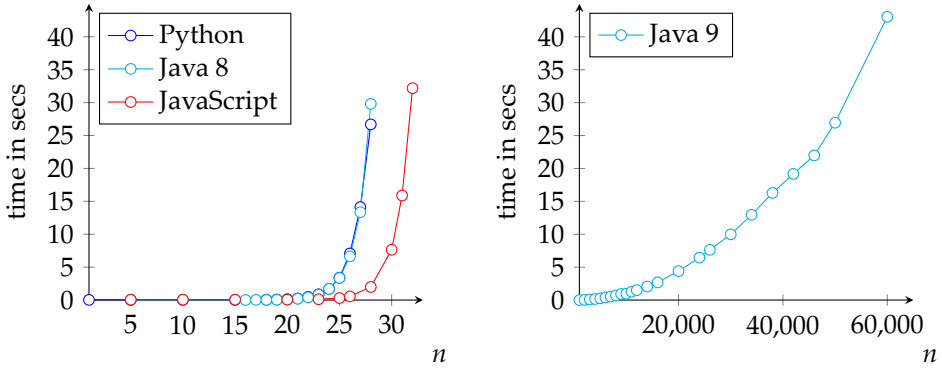
The purpose of the *simp* function is to keep the regular expressions small. Normally the derivative function makes the regular expression bigger (see the SEQ case and the example in (2)) and the algorithm would be slower and slower over time. The *simp* function counters this increase in size and the 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 file `catastrophic9.java`, `catastrophic.js` and `catastrophic.py` on KEATS). Compare this with the matcher you have implemented. How long can the string of a's be in your matcher and still stay within the 30 seconds time limit?

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



[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.