

Coursework 8 (Regular Expressions and Brainf***)

This coursework is worth 10%. It is about regular expressions, pattern matching and an interpreter. The first part is due on 30 November at 11pm; the second, more advanced part, is due on 21 December at 11pm. In the first part, you are asked to implement a regular expression matcher based on derivatives of regular expressions. The reason is that regular expression matching in Java and Python can sometimes be extremely slow. The advanced part is about an interpreter for a very simple programming language.

Important:

- Make sure the files you submit can be processed by just calling `scala <<filename.scala>>` on the commandline. 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 use any mutable data structures in your submissions! They are not needed. This means you cannot create new `Arrays` or `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. 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 360 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.

Part 1 (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`:

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

Why? Knowing how to match regular expressions and strings 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 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

- <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`)

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:
0	\mapsto ZERO	
1	\mapsto ONE	
c	\mapsto CHAR(c)	
$r_1 + r_2$	\mapsto ALT(r_1 , r_2)	$r_1 \mid r_2$
$r_1 \cdot r_2$	\mapsto SEQ(r_1 , r_2)	$r_1 \sim r_2$
r^*	\mapsto STAR(r)	$r.\%$

- (1a) 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:

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

[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:

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

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

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

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

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

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

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

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 on the way simplifies every regular expression on the left (see below) to the regular expression on the right, except it does not simplify inside $*$ -regular expressions.

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

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 sometimes also called post-order traversal. Furthermore, remember numerical expressions from school times: there you had expressions like $u + \dots + (1 \cdot x) - \dots (z + (y \cdot 0)) \dots$ 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 rule should be applied next. [2 Marks]

- (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:

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

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]

- (1e) 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:

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

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]

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 (1b)) 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.

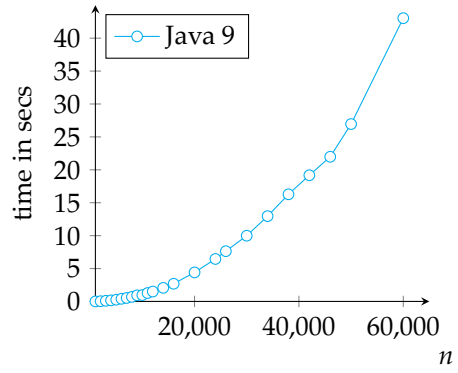
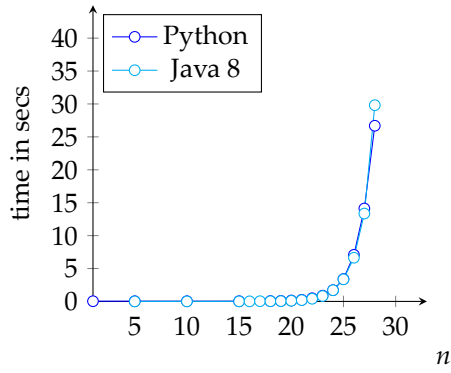
[https://en.wikipedia.org/wiki/Janusz_Brzozowski_\(computer_scientist\)](https://en.wikipedia.org/wiki/Janusz_Brzozowski_(computer_scientist))

If you want to see how badly the regular expression matchers do in Java¹ and in 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 `catastrophic.java` and `catastrophic.py` on KEATS). Compare this with the

¹Version 8 and below; Version 9 does not seem to be as catastrophic, but still worse than the regular expression matcher based on derivatives.

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 $\underbrace{a \dots a}_n$



safely reads the corresponding memory location. If the Map is not defined at the memory pointer, `sread` returns `0`.

Write another function `write`, which takes a memory, a memory pointer and an integer value as argument and updates the Map with the value at the given memory location. As usual the Map is not updated 'in-place' but a new map is created with the same data, except the value is stored at the given memory pointer. [1 Mark]

- (2b) Write two functions, `jumpRight` and `jumpLeft` that are needed to implement the loop constructs of `brainf***`. They take a program (a `String`) and a program counter (an `Int`) as argument and move right (respectively left) in the string in order to find the **matching** opening/closing bracket. For example, given the following program with the program counter indicated by an arrow:

```
--[. .+>--],>,++
      ↑
```

then the matching closing bracket is in 9th position (counting from 0) and `jumpRight` is supposed to return the position just after this

```
--[. .+>--],>,++
      ↑
```

meaning it jumps to after the loop. Similarly, if you are in 8th position then `jumpLeft` is supposed to jump to just after the opening bracket (that is jumping to the beginning of the loop):

```
--[. .+>--],>,++      jumpLeft →      --[. .+>--],>,++
      ↑                                  ↑
```

Unfortunately we have to take into account that there might be other opening and closing brackets on the 'way' to find the matching bracket. For example in the `brainf***` program

```
--[. .[+]--],>,++
      ↑
```

we do not want to return the index for the '-' in the 9th position, but the program counter for ',' in 12th position. The easiest to find out whether a bracket is matched is by using levels (which are the third argument in `jumpLeft` and `jumpLeft`). In case of `jumpRight` you increase the level by one whenever you find an opening bracket and decrease by one for a closing bracket. Then in `jumpRight` you are looking for the closing bracket on level 0. For `jumpLeft` you do the opposite. In this way you can find **matching** brackets in strings such as


```
--[. . [[-]+>[.] ]--],>,++  
      ↑
```

for which `jumpRight` should produce the position:

```
--[. . [[-]+>[.] ]--],>,++  
                        ↑
```

It is also possible that the position returned by `jumpRight` or `jumpLeft` is outside the string in cases where there are no matching brackets. For example

```
--[. . [[-]+>[.] ]--],>,++       $\xrightarrow{\text{jumpRight}}$   --[. . [[-]+>[.] ]-->,++  
      ↑                                                    ↑
```

[1 Mark]

- (2c) Write a recursive function `run` that executes a `brainf***` program. It takes a program, a program counter, a memory pointer and a memory as arguments. If the program counter is outside the program string, the execution stops and `run` returns the memory. If the program counter is inside the string, it reads the corresponding character and updates the program counter `pc`, memory pointer `mp` and memory `mem` according to the rules shown in Figure 1. It then calls recursively `run` with the updated data.

Write another function `start` that calls `run` with a given `brainfu**` program and memory, and the program counter and memory pointer set to 0. Like `run` it returns the memory after the execution of the program finishes. You can test your `brainf**k` interpreter with the Sierpinski triangle or the Hello world programs or have a look at

<https://esolangs.org/wiki/Brainfuck>

[2 Marks]

'>'	<ul style="list-style-type: none"> • $pc + 1$ • $mp + 1$ • mem unchanged
'<'	<ul style="list-style-type: none"> • $pc + 1$ • $mp - 1$ • mem unchanged
'+'	<ul style="list-style-type: none"> • $pc + 1$ • mp unchanged • mem updated with $mp \rightarrow mem(mp) + 1$
'-'	<ul style="list-style-type: none"> • $pc + 1$ • mp unchanged • mem updated with $mp \rightarrow mem(mp) - 1$
'.'	<ul style="list-style-type: none"> • $pc + 1$ • mp and mem unchanged • print out $mem(mp)$ as a character
','	<ul style="list-style-type: none"> • $pc + 1$ • mp unchanged • mem updated with $mp \rightarrow$ input <p>the input is given by <code>Console.in.read().toByte</code></p>
'['	<p>if $mem(mp) == 0$ then</p> <ul style="list-style-type: none"> • $pc = \text{jumpRight}(\text{prog}, pc + 1, 0)$ • mp and mem unchanged <p>otherwise if $mem(mp) != 0$ then</p> <ul style="list-style-type: none"> • $pc + 1$ • mp and mem unchanged
']'	<p>if $mem(mp) != 0$ then</p> <ul style="list-style-type: none"> • $pc = \text{jumpLeft}(\text{prog}, pc - 1, 0)$ • mp and mem unchanged <p>otherwise if $mem(mp) == 0$ then</p> <ul style="list-style-type: none"> • $pc + 1$ • mp and mem unchanged
any other char	<ul style="list-style-type: none"> • $pc + 1$ • mp and mem unchanged

Figure 1: The rules for how commands in the brainf^{***} language update the program counter pc , memory pointer mp and memory mem .