# Handout 2 (Regular Expression Matching)

This lecture is about implementing a more efficient regular expression matcher (the plots on the right below)—more efficient than the matchers from regular expression libraries in Ruby, Python, JavaScript, Swift and Java (the plots on the left).[1]

To start with let us look more closely at the experimental data: The first pair of plots shows the running time for the regular expression $(a^*)^* \cdot b$ and strings composed of n as, like

$$\underbrace{\texttt{a...a}}_{n}$$

This means the regular expression actually does not match the strings. The second pair of plots shows the running time for the regular expressions of the form $a^{?\{n\}} \cdot a^{\{n\}}$ and corresponding strings composed of n as (this time the regular expressions match the strings). To see the substantial differences in the left and right plots below, note the different scales of the x-axis.

Graphs: $(a^*)^* \cdot b$ and strings $\underbrace{a \ldots a}_{n}$
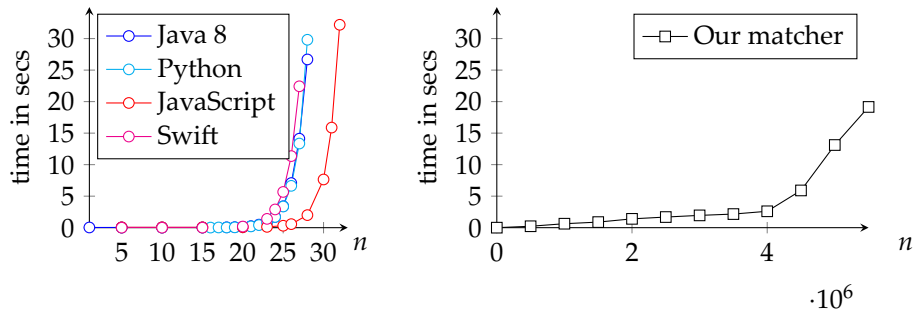
Graphs: $a^{?\{n\}} \cdot a^{\{n\}}$ and strings $\underbrace{a \ldots a}_{n}$
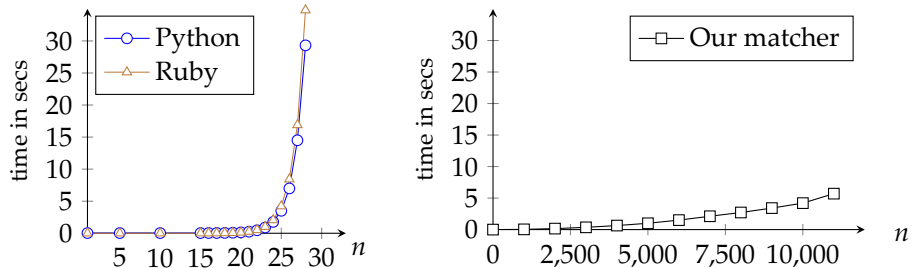
[1]Have a look at KEATS: students last year contributed also data for the Dart language.

In what follows we will use these regular expressions and strings as running examples. There will be several versions (V1, V2, V3,…) of our matcher.[2]

Having specified in the previous lecture what problem our regular expression matcher is supposed to solve, namely for any given regular expression r and string s answer *true* if and only if

$$s \in L(r)$$

we can look for an algorithm to solve this problem. Clearly we cannot use the function L directly for this, because in general the set of strings L returns is infinite (recall what $L(a^*)$ is). In such cases there is no way we can implement an exhaustive test for whether a string is member of this set or not. In contrast our matching algorithm will operate on the regular expression r and string s, only, which are both finite objects. Before we explain the matching algorithm, let us have a closer look at what it means when two regular expressions are equivalent.

Regular Expression Equivalences

We already defined in Handout 1 what it means for two regular expressions to be equivalent, namely whether their *meaning* is the same language:

$$r_1 \equiv r_2 \overset{\text{def}}{=} L(r_1) = L(r_2)$$

It is relatively easy to verify that some concrete equivalences hold, for example

$$
\begin{aligned}
(a + b) + c &\equiv & a + (b + c) \\
a + a &\equiv & a \\
a + b &\equiv & b + a \\
(a \cdot b) \cdot c &\equiv & a \cdot (b \cdot c) \\
c \cdot (a + b) &\equiv & (c \cdot a) + (c \cdot b)
\end{aligned}
$$

but also easy to verify that the following regular expressions are *not* equivalent

$$
\begin{aligned}
a \cdot a &\not\equiv & a \\
a + (b \cdot c) &\not\equiv & (a + b) \cdot (a + c)
\end{aligned}
$$

I leave it to you to verify these equivalences and non-equivalences. It is also interesting to look at some corner cases involving **1** and **0**:

$$
\begin{aligned}
a \cdot \mathbf{0} &\not\equiv & a \\
a + \mathbf{1} &\not\equiv & a \\
\mathbf{1} &\equiv & \mathbf{0}^* \\
\mathbf{1}^* &\equiv & \mathbf{1} \\
\mathbf{0}^* &\not\equiv & \mathbf{0}
\end{aligned}
$$

---

[2]The corresponding files are `re1.sc`, `re2.sc` and so on. As usual, you can find the code on KEATS.

Again I leave it to you to make sure you agree with these equivalences and non-equivalences.

For our matching algorithm however the following seven equivalences will play an important role:

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

They always hold no matter what the regular expression r looks like. The first two are easy to verify since $L(\mathbf{0})$ is the empty set. The next two are also easy to verify since $L(\mathbf{1}) = \{[]\}$ and appending the empty string to every string of another set, leaves the set unchanged. Be careful to fully comprehend the fifth and sixth equivalence: if you concatenate two sets of strings and one is the empty set, then the concatenation will also be the empty set. To see this, check the definition of _@_ for sets. The last equivalence is again trivial.

What will be critical later on is that we can orient these equivalences and read them from left to right. In this way we can view them as *simplification rules*. Consider for example the regular expression

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

If we can find an equivalent regular expression that is simpler (that usually means smaller), then this might potentially make our matching algorithm run faster. We can look for such a simpler, but equivalent, regular expression r' because whether a string s is in $L(r)$ or in $L(r')$ does not matter as long as $r \equiv r'$. Yes? [3]

In the example above you will see that the regular expression in (1) is equivalent to just r_1. You can verify this by iteratively applying the simplification rules from above:

$$
\begin{aligned}
&(r_1 + \mathbf{0}) \cdot \mathbf{1} + ((\mathbf{1} + r_2) + r_3) \cdot (\underline{r_4 \cdot \mathbf{0}}) \\
\equiv\ &(r_1 + \mathbf{0}) \cdot \mathbf{1} + \underline{((\mathbf{1} + r_2) + r_3) \cdot \mathbf{0}} \\
\equiv\ &\underline{(r_1 + \mathbf{0}) \cdot \mathbf{1}} + \mathbf{0} \\
\equiv\ &\underline{(r_1 + \mathbf{0}) + \mathbf{0}} \\
\equiv\ &\underline{r_1 + \mathbf{0}} \\
\equiv\ &r_1
\end{aligned}
$$

In each step, I underlined where a simplification rule is applied. Our matching algorithm in the next section will often generate such "useless" $\mathbf{1}$s and $\mathbf{0}$s, therefore simplifying them away will make the algorithm quite a bit faster.

---

[3]You have checked this for yourself? Your friendly lecturer might talk rubbish…one never knows.

Finally here are three equivalences between regular expressions which are not so obvious:

$$
\begin{array}{rcl}
r^* & \equiv & \mathbf{1} + r \cdot r^* \\
(r_1 + r_2)^* & \equiv & r_1^* \cdot (r_2 \cdot r_1^*)^* \\
(r_1 \cdot r_2)^* & \equiv & \mathbf{1} + r_1 \cdot (r_2 \cdot r_1)^* \cdot r_2
\end{array}
$$

We will not use them in our algorithm, but feel free to convince yourself that they actually hold. As an aside, there has been a lot of research on questions like: Can one always decide whether two regular expressions are equivalent or not? What does an algorithm look like to decide this efficiently? Surprisingly, many of such questions turn out to be non-trivial problems.

The Matching Algorithm

The regular expression matching algorithm we will introduce below consists of two parts: One is the function *nullable* which takes a regular expression as an argument and decides whether it can match the empty string (this means it returns a boolean in Scala). This can be easily defined recursively as follows:

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

The idea behind this function is that the following property holds:

$$nullable(r) \ \text{ if and only if } \ [] \in L(r)$$

Note on the left-hand side of the if-and-only-if we have a function we can implement, for example in Scala; on the right we have its specification (which we cannot implement in a programming language).

The other function of our matching algorithm calculates a *derivative* of a regular expression. This is a function which will take a regular expression, say r, and a character, say c, as arguments and returns a new regular expression. Be mindful that the intuition behind this function is not so easy to grasp on first reading. Essentially this function solves the following problem: if r can match a string of the form $c :: s$, what does a regular expression look like that can match just s? The definition of this function is as follows:

4

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

The first two clauses can be rationalised as follows: recall that *der* should calculate a regular expression so that provided the "input" regular expression can match a string of the form $c :: s$, we want a regular expression for s. Since neither $\mathbf{0}$ nor $\mathbf{1}$ can match a string of the form $c :: s$, we return $\mathbf{0}$. In the third case we have to make a case-distinction: In case the regular expression is c, then clearly it can recognise a string of the form $c :: s$, just that s is the empty string. Therefore we return the $\mathbf{1}$-regular expression in this case, as it can match the empty string. In the other case we again return $\mathbf{0}$ since no string of the $c :: s$ can be matched. Next come the recursive cases, which are a bit more involved. Fortunately, the $+$-case is still relatively straightforward: all strings of the form $c :: s$ are either matched by the regular expression r_1 or r_2. So we just have to recursively call *der* with these two regular expressions and compose the results again with $+$. Makes sense?

The $\cdot$-case is more complicated: if $r_1 \cdot r_2$ matches a string of the form $c :: s$, then the first part must be matched by r_1. Consequently, it makes sense to construct the regular expression for s by calling *der* with r_1 and "appending" r_2. There is however one exception to this simple rule: if r_1 can match the empty string, then all of $c :: s$ is matched by r_2. Therefore in case r_1 is nullable (that is can match the empty string) we have to allow the choice $der\ c\ r_2$ for calculating the regular expression that can match s. This means we have to add the regular expression $der\ c\ r_2$ in the result. The $*$-case is again simple: if r^* matches a string of the form $c :: s$, then the first part must be "matched" by a single copy of r. Therefore we call recursively $der\ c\ r$ and "append" r^* in order to match the rest of s. Still makes sense?

If all this did not make sense yet, here is another way to explain the definition of *der* by considering the following operation on sets:

$$Der\ c\ A \overset{\text{def}}{=} \{s \mid c :: s \in A\} \tag{2}$$

This operation essentially transforms a set of strings A by filtering out all strings that do not start with c and then strips off the c from all the remaining strings. For example suppose $A = \{foo, bar, frak\}$ then

$$Der\ f\ A = \{oo, rak\} \quad , \quad Der\ b\ A = \{ar\} \quad \text{and} \quad Der\ a\ A = \{\}$$

Note that in the last case *Der* is empty, because no string in A starts with a. With this operation we can state the following property about *der*:

$$L(der\,c\,r) = Der\,c\,(L(r))$$

This property clarifies what regular expression *der* calculates, namely take the set of strings that *r* can match (that is $L(r)$), filter out all strings not starting with *c* and strip off the *c* from the remaining strings—this is exactly the language that *der c r* can match.

If we want to find out whether the string *abc* is matched by the regular expression r_1 then we can iteratively apply *der* as follows

Input: r_1, *abc*

Step 1:	build derivative of *a* and r_1	$(r_2 = der\,a\,r_1)$
Step 2:	build derivative of *b* and r_2	$(r_3 = der\,b\,r_2)$
Step 3:	build derivative of *c* and r_3	$(r_4 = der\,c\,r_3)$
Step 4:	the string is exhausted: test whether r_4 can recognise the empty string	$(nullable(r_4))$
Output:	result of this test $\Rightarrow$ *true* or *false*	

Again the operation *Der* might help to rationalise this algorithm. We want to know whether $abc \in L(r_1)$. We do not know yet—but let us assume it is. Then $Der\,a\,L(r_1)$ builds the set where all the strings not starting with *a* are filtered out. Of the remaining strings, the *a* is stripped off. So we should still have *bc* in the set. Then we continue with filtering out all strings not starting with *b* and stripping off the *b* from the remaining strings, that means we build $Der\,b\,(Der\,a\,(L(r_1)))$. Finally we filter out all strings not starting with *c* and strip off *c* from the remaining string. This is $Der\,c\,(Der\,b\,(Der\,a\,(L(r_1))))$. Now if *abc* was in the original set ($L(r_1)$), then $Der\,c\,(Der\,b\,(Der\,a\,(L(r_1))))$ must contain the empty string. If not, then *abc* was not in the language we started with.

Our matching algorithm using *der* and *nullable* works similarly, just using regular expressions instead of sets. In order to define our algorithm we need to extend the notion of derivatives from single characters to strings. This can be done using the following function, taking a string and a regular expression as input and a regular expression as output.

$$ders\,[]\,r \quad \overset{\text{def}}{=}\quad r$$
$$ders\,(c::s)\,r \quad \overset{\text{def}}{=}\quad ders\,s\,(der\,c\,r)$$

This function iterates *der* taking one character at the time from the original string until the string is exhausted. Having *ders* in place, we can finally define our matching algorithm:

$$matcher\,r\,s \overset{\text{def}}{=} nullable(ders\,s\,r)$$

and we can claim that

$$\textit{matcher } r \, s \quad \text{if and only if} \quad s \in L(r)$$

holds, which means our algorithm satisfies the specification. Of course we can claim many things…whether the claim holds any water is a different question.

This algorithm was introduced by Janusz Brzozowski in 1964, but is more widely known only in the last 10 or so years. Its main attractions are simplicity and being fast, as well as being easily extendible for other regular expressions such as $r^{\{n\}}$, $r^?$, $\sim r$ and so on (this is subject of Coursework 1).

The Matching Algorithm in Scala

Another attraction of the algorithm is that it can be easily implemented in a functional programming language, like Scala. Given the implementation of regular expressions in Scala shown in the first lecture and handout, the functions and subfunctions for `matcher` are shown in Figure 1.

For running the algorithm with our first example, the evil regular expression $a^{?\{n\}} \cdot a^{\{n\}}$, we need to implement the optional regular expression and the 'exactly n-times regular expression'. This can be done with the translations

```scala
def OPT(r: Rexp) = ALT(r, ONE)

def NTIMES(r: Rexp, n: Int) : Rexp = n match {
  case 0 => ONE
  case 1 => r
  case n => SEQ(r, NTIMES(r, n - 1))
}
```

Running the matcher with this example, we find it is slightly worse then the matcher in Ruby and Python. Ooops…

Graph: $a^{?\{n\}} \cdot a^{\{n\}}$ and strings $\underbrace{a \ldots a}_{n}$
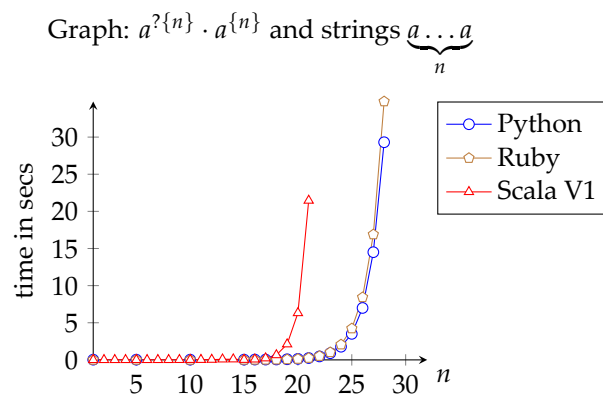
Analysing this failure we notice that for $a^{\{n\}}$, for example, we generate quite big regular expressions:

```scala
def nullable(r: Rexp) : Boolean = r match {
  case ZERO => false
  case ONE => true
  case CHAR(_) => false
  case ALT(r1, r2) => nullable(r1) || nullable(r2)
  case SEQ(r1, r2) => nullable(r1) && nullable(r2)
  case STAR(_) => true
}

def der(c: Char, r: Rexp) : Rexp = r match {
  case ZERO => ZERO
  case ONE => ZERO
  case CHAR(d) => if (c == d) ONE else ZERO
  case ALT(r1, r2) => ALT(der(c, r1), der(c, r2))
  case SEQ(r1, r2) =>
    if (nullable(r1)) ALT(SEQ(der(c, r1), r2), der(c, r2))
    else SEQ(der(c, r1), r2)
  case STAR(r) => SEQ(der(c, r), STAR(r))
}

def ders(s: List[Char], r: Rexp) : Rexp = s match {
  case Nil => r
  case c::s => ders(s, der(c, r))
}

def matcher(r: Rexp, s: String) : Boolean =
  nullable(ders(s.toList, r))
```

Figure 1: A Scala implementation of the *nullable* and the derivative function. These functions are easy to implement in functional programming languages. This is because pattern matching and recursion allow us to mimic the mathematical definitions very closely. Nearly all functional programming languages support pattern matching and recursion out of the box.

$$
\begin{aligned}
1: &\quad a \\
2: &\quad a \cdot a \\
3: &\quad a \cdot a \cdot a \\
&\quad \dots \\
13: &\quad a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \\
&\quad \dots
\end{aligned}
$$

Our algorithm traverses such regular expressions at least once every time a derivative is calculated. So having large regular expressions will cause problems. This problem is aggravated by $a^?$ being represented as $a + \mathbf{1}$.

We can however fix this easily by having an explicit constructor for $r^{\{n\}}$. In Scala we would introduce a constructor like

```scala
case class NTIMES(r: Rexp, n: Int) extends Rexp
```

With this fix we have a constant "size" regular expression for our running example no matter how large n is (see the `size` section in the implementations). This means we have to also add cases for NTIMES in the functions *nullable* and *der*. Does the change have any effect?

Graph: $a^{?\{n\}} \cdot a^{\{n\}}$ and strings $\underbrace{a \dots a}_{n}$
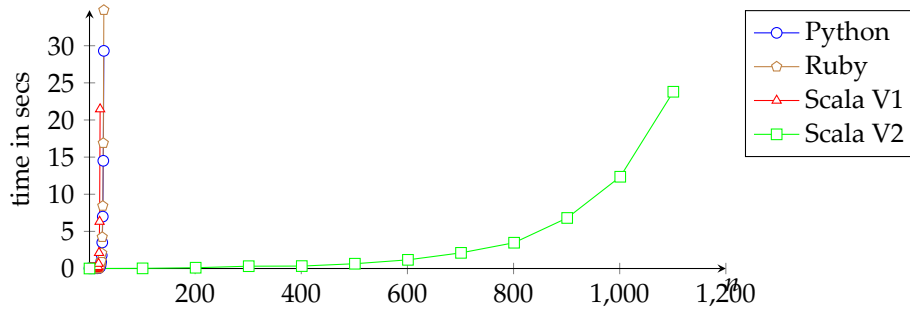
Now we are talking business! The modified matcher can within 25 seconds handle regular expressions up to $n = 1{,}100$ before a StackOverflow is raised. Recall that Python and Ruby (and our first version, Scala V1) could only handle $n = 27$ or so in 30 seconds. We have not tried our algorithm on the second example $(a^*)^* \cdot b$—I leave this to you.

The moral is that our algorithm is rather sensitive to the size of regular expressions it needs to handle. This is of course obvious because both *nullable* and *der* frequently need to traverse the whole regular expression. There seems, however, one more issue for making the algorithm run faster. The derivative function often produces "useless" $\mathbf{0}$s and $\mathbf{1}$s. To see this, consider $r = ((a \cdot b) + b)^*$ and the following three derivatives

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

If we simplify them according to the simplification rules from the beginning, we can replace the right-hand sides by the smaller equivalent regular expressions

$$der\ a\ r \equiv b \cdot r$$
$$der\ b\ r \equiv r$$
$$der\ c\ r \equiv \mathbf{0}$$

I leave it to you to contemplate whether such a simplification can have any impact on the correctness of our algorithm (will it change any answers?). Figure 2 gives a simplification function that recursively traverses a regular expression and simplifies it according to the rules given at the beginning. There are only rules for $+$ and $\cdot$. There is no simplification rule for a star, because empirical data and also a little thought showed that simplifying under a star is a waste of computation time. The simplification function will be called after every derivation. This additional step removes all the "junk" the derivative function introduced. Does this improve the speed? You bet!!

Graph: $a^{?\{n\}} \cdot a^{\{n\}}$ and strings $\underbrace{a \ldots a}_{n}$
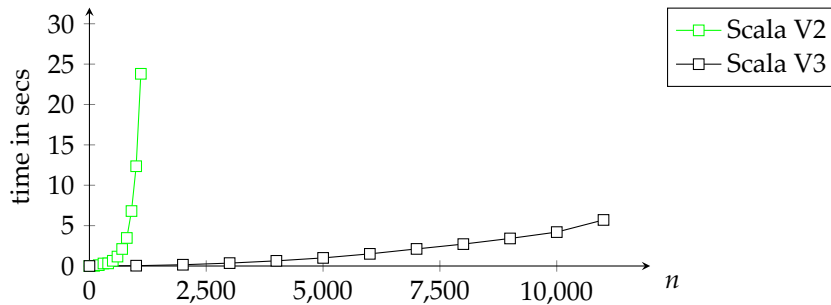
To recap, Python and Ruby needed approximately 30 seconds to match a string of 28 as and the regular expression $a^{?\{n\}} \cdot a^{\{n\}}$. We need a third of this time to do the same with strings up to 11,000 as. Similarly, Java 8 and Python needed 30 seconds to find out the regular expression $(a^*)^* \cdot b$ does not match the string of 28 as. In Java 9 and later this has been cranked up to 39,000 as, but we can do the same in the same amount of time for strings composed of nearly 6,000,000 as. This is shown in the following plot.

```
1   def simp(r: Rexp) : Rexp = r match {
2     case ALT(r1, r2) => {
3       (simp(r1), simp(r2)) match {
4         case (ZERO, r2s) => r2s
5         case (r1s, ZERO) => r1s
6         case (r1s, r2s) =>
7           if (r1s == r2s) r1s else ALT(r1s, r2s)
8       }
9     }
10    case SEQ(r1, r2) => {
11      (simp(r1), simp(r2)) match {
12        case (ZERO, _) => ZERO
13        case (_, ZERO) => ZERO
14        case (ONE, r2s) => r2s
15        case (r1s, ONE) => r1s
16        case (r1s, r2s) => SEQ(r1s, r2s)
17      }
18    }
19    case r => r
20  }
21
22  def ders(s: List[Char], r: Rexp) : Rexp = s match {
23    case Nil => r
24    case c::s => ders(s, simp(der(c, r)))
25  }
```

Figure 2: The simplification function and modified `ders`-function; this function now calls `der` first, but then simplifies the resulting derivative regular expressions before building the next derivative, see Line 24.

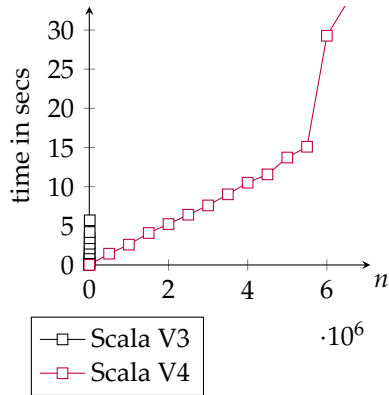Graph: $(a^*)^* \cdot b$ and strings $\underbrace{a \ldots a}_{n}$
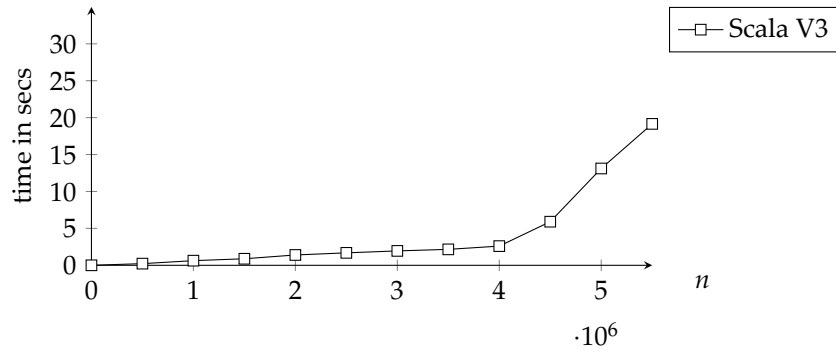
Epilogue

(23/Aug/2016) I found another place where this algorithm can be sped up (this idea is not integrated with what is coming next, but I present it nonetheless). The idea is to not define `ders` so that it iterates the derivative character-by-character, but in bigger chunks. The resulting code for `ders2` looks as follows:

```scala
def ders2(s: List[Char], r: Rexp) : Rexp = (s, r) match {
  case (Nil, r) => r
  case (s, ZERO) => ZERO
  case (s, ONE) => if (s == Nil) ONE else ZERO
  case (s, CHAR(c)) => if (s == List(c)) ONE else
                       if (s == Nil) CHAR(c) else ZERO
  case (s, ALT(r1, r2)) => ALT(ders2(s, r2), ders2(s, r2))
  case (c::s, r) => ders2(s, simp(der(c, r)))
}
```

I have not fully understood why this version is much faster, but it seems it is a combination of the clauses for `ALT` and `SEQ`. In the latter case we call `der` with a single character and this potentially produces an alternative. The derivative of such an alternative can then be more efficiently calculated by `ders2` since it pushes a whole string under an `ALT`. The numbers are that in the second case $(a^*)^* \cdot b$ both versions are pretty much the same, but in the first case $a^{?\{n\}} \cdot a^{\{n\}}$ the improvement gives another factor of 100 speedup. Nice!

Graph: $a^{?\{n\}} \cdot a^{\{n\}}$ and strings $\underbrace{a \ldots a}_{n}$

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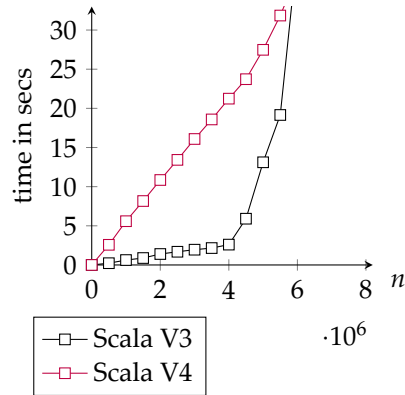
Proofs

You might not like doing proofs. But they serve a very important purpose in Computer Science: How can we be sure that our algorithm matches its specification? We can try to test the algorithm, but that often overlooks corner cases and an exhaustive testing is impossible (since there are infinitely many inputs). Proofs allow us to ensure that an algorithm really meets its specification.

For the programs we look at in this module, the proofs will mostly by some form of induction. Remember that regular expressions are defined as

$$
\begin{array}{lll}
r ::= & \mathbf{0} & \text{nothing} \\
& | \ \mathbf{1} & \text{empty string / "" / []} \\
& | \ c & \text{single character} \\
& | \ r_1 + r_2 & \text{alternative / choice} \\
& | \ r_1 \cdot r_2 & \text{sequence} \\
& | \ r^* & \text{star (zero or more)}
\end{array}
$$

If you want to show a property $P(r)$ for *all* regular expressions r, then you have to follow essentially the recipe:

- P has to hold for $\mathbf{0}$, $\mathbf{1}$ and c (these are the base cases).

- P has to hold for $r_1 + r_2$ under the assumption that P already holds for r_1 and r_2.

- P has to hold for $r_1 \cdot r_2$ under the assumption that P already holds for r_1 and r_2.

- P has to hold for r^* under the assumption that P already holds for r.

A simple proof is for example showing the following property:

$$nullable(r) \text{ if and only if } [] \in L(r) \tag{3}$$

Let us say that this property is $P(r)$, then the first case we need to check is whether $P(\mathbf{0})$ (see recipe above). So we have to show that

$$nullable(\mathbf{0}) \text{ if and only if } [] \in L(\mathbf{0})$$

whereby $nullable(\mathbf{0})$ is by definition of the function $nullable$ always $false$. We also have that $L(\mathbf{0})$ is by definition $\{\}$. It is impossible that the empty string $[]$ is in the empty set. Therefore also the right-hand side is false. Consequently we verified this case: both sides are false. We would still need to do this for $P(\mathbf{1})$ and $P(c)$. I leave this to you to verify.

Next we need to check the inductive cases, for example $P(r_1 + r_2)$, which is

$$nullable(r_1 + r_2) \text{ if and only if } [] \in L(r_1 + r_2) \tag{4}$$

The difference to the base cases is that in the inductive cases we can already assume we proved P for the components, that is we can assume.

$$nullable(r_1) \text{ if and only if } [] \in L(r_1) \text{ and}$$
$$nullable(r_2) \text{ if and only if } [] \in L(r_2)$$

These are called the induction hypotheses. To check this case, we can start from $nullable(r_1 + r_2)$, which by definition of $nullable$ is

$$nullable(r_1) \lor nullable(r_2)$$

Using the two induction hypotheses from above, we can transform this into

$$[] \in L(r_1) \lor [] \in (r_2)$$

We just replaced the $nullable(\ldots)$ parts by the equivalent $[] \in L(\ldots)$ from the induction hypotheses. A bit of thinking convinces you that if $[] \in L(r_1) \lor [] \in L(r_2)$ then the empty string must be in the union $L(r_1) \cup L(r_2)$, that is

$$[] \in L(r_1) \cup L(r_2)$$

but this is by definition of L exactly $[] \in L(r_1 + r_2)$, which we needed to establish according to statement in (4). What we have shown is that starting from $nullable(r_1 + r_2)$ we have done equivalent transformations to end up with $[] \in L(r_1 + r_2)$. Consequently we have established that $P(r_1 + r_2)$ holds.

In order to complete the proof we would now need to look at the cases $P(r_1 \cdot r_2)$ and $P(r^*)$. Again I let you check the details.

You might also have to do induction proofs over strings. That means you want to establish a property $P(s)$ for all strings s. For this remember strings are lists of characters. These lists can be either the empty list or a list of the form $c :: s$. If you want to perform an induction proof for strings you need to consider the cases

- P has to hold for $[]$ (this is the base case).

- P has to hold for $c :: s$ under the assumption that P already holds for s.

Given this recipe, I let you show

$$Ders\, s\, (L(r)) = L(ders\, s\, r) \qquad (5)$$

by induction on s. Recall Der is defined for character—see (2); $Ders$ is similar, but for strings:

$$Ders\, s\, A \;\overset{\text{def}}{=}\; \{s' \,|\, s@s' \in A\}$$

In this proof you can assume the following property for der and Der has already been proved, that is you can assume

$$L(der\, c\, r) = Der\, c\, (L(r))$$

holds (this would be of course another property that needs to be proved in a side-lemma by induction on r). This is a bit more challenging, but not impossible.

To sum up, using reasoning like the one shown above allows us to show the correctness of our algorithm. To see this, start from the specification

$$s \in L(r)$$

That is the problem we want to solve. Thinking a little, you will see that this problem is equivalent to the following problem

$$[] \in Ders\, s\, (L(r)) \qquad (6)$$

You agree? But we have shown above in (5), that the $Ders$ can be replaced by $L(ders \ldots)$. That means (6) is equivalent to

$$[] \in L(ders\, s\, r) \qquad (7)$$

We have also shown that testing whether the empty string is in a language is equivalent to the *nullable* function; see (3). That means (7) is equivalent with

$$nullable(ders\, s\, r)$$

But this is just the definition of *matcher*

$$matcher\, s\, r \;\overset{\text{def}}{=}\; nullable(ders\, s\, r)$$

In effect we have shown

$$matcher\, s\, r \;\; \text{if and only if} \;\; s \in L(r)$$

which is the property we set out to prove: our algorithm meets its specification. To have done so, requires a few induction proofs about strings and regular

expressions. Following the *induction recipes* is already a big step in actually performing these proofs. If you do not believe it, proofs have helped me to make sure my code is correct and in several instances prevented me of letting slip embarrassing mistakes into the 'wild'. In fact I have found a number of mistakes in the brilliant work by Sulzmann and Lu, which we are going to have a look at in Lecture 4. But in Lecture 3 we should first find out why on earth are existing regular expression matchers so abysmally slow. Are the people in Python, Ruby, Swift, JavaScript, Java and also in Rust[4] just idiots? For example could it possibly be that what we have implemented here in Scala is faster than the regex engine that has been implemented in Rust? See you next week…

---

[4]Interestingly the regex engine in Rust says it guarantees a linear behaviour when deciding when a regular expression matches a string. ☝