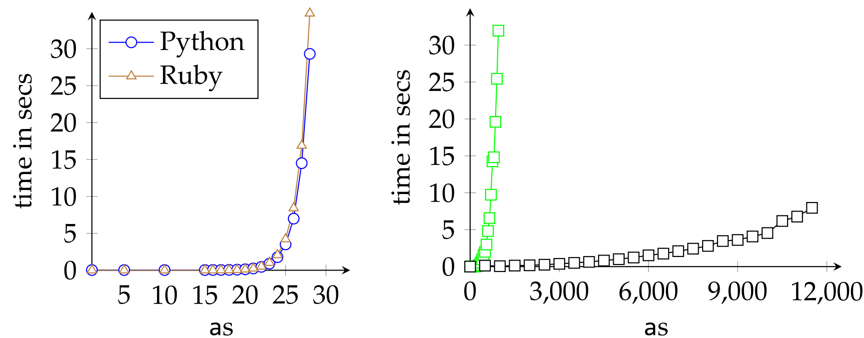
Handout 2 (Regular Expression Matching)

This lecture is about implementing a more efficient regular expression matcher (the plots on the right)—more efficient than the matchers from regular expression libraries in Ruby and Python (the plots on the left). These plots show the running time for the evil regular expression $a?^{\{n\}}a^{\{n\}}$ and string composed of n `as`. We will use this regular expression and strings as running example. To see the substantial differences in the two plots below, note the different scales of the x-axes.



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 at 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. Before we come to the matching algorithm, however, 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 if their meaning is the same language:

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

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

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

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

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

I leave it to you to verify these equivalences and non-equivalences. It is also interesting to look at some corner cases involving ϵ and $\varnothing$:

$$
\begin{array}{rcl}
a\cdot\varnothing & \not\equiv & a \\
a+\epsilon & \not\equiv & a \\
\epsilon & \equiv & \varnothing^* \\
\epsilon^* & \equiv & \epsilon \\
\varnothing^* & \not\equiv & \varnothing
\end{array}
$$

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:

$$
\begin{array}{rcl}
r+\varnothing & \equiv & r \\
\varnothing+r & \equiv & r \\
r\cdot\epsilon & \equiv & r \\
\epsilon\cdot r & \equiv & r \\
r\cdot\varnothing & \equiv & \varnothing \\
\varnothing\cdot r & \equiv & \varnothing \\
r+r & \equiv & r
\end{array}
$$

which always hold no matter what the regular expression r looks like. The first two are easy to verify since $L(\varnothing)$ is the empty set. The next two are also easy to verify since $L(\epsilon) = \{[]\}$ 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 `_ @ _`. The last equivalence is again trivial.

What will be important 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

$$
(r_1+\varnothing)\cdot\epsilon+((\epsilon+r_2)+r_3)\cdot(r_4\cdot\varnothing) \tag{1}
$$

If we can find an equivalent regular expression that is simpler (smaller for example), then this might potentially make our matching algorithm run faster.

The reason is that whether a string s is in $L(r)$ or in $L(r')$ with $r \equiv r'$ will always give the same answer. In the example above you will see that the regular expression is equivalent to r_1. You can verify this by iteratively applying the simplification rules from above:

$$\begin{array}{rl}
 & (r_1 + \varnothing) \cdot \epsilon + ((\epsilon + r_2) + r_3) \cdot (\underline{r_4 \cdot \varnothing}) \\
\equiv & (r_1 + \varnothing) \cdot \epsilon + \underline{((\epsilon + r_2) + r_3) \cdot \varnothing} \\
\equiv & \underline{(r_1 + \varnothing) \cdot \epsilon} + \varnothing \\
\equiv & \underline{(r_1 + \varnothing)} + \varnothing \\
\equiv & \underline{r_1 + \varnothing} \\
\equiv & r_1
\end{array}$$

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

The Matching Algorithm

The algorithm we will define below consists of two parts. One is the function *nullable* which takes a regular expression as 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:

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

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; 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 argument and return a new regular expression. Be careful 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 the regular expression look like that can match just s? The definition of this function is as follows:

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

The first two clauses can be rationalised as follows: recall that *der* should calculate a regular expression so that given the "input" regular expression can match a string of the form $c :: s$, we want a regular expression for s. Since neither $\varnothing$ nor ϵ can match a string of the form $c :: s$, we return $\varnothing$. 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 ϵ-regular expression. In the other case we again return $\varnothing$ 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 $+$. Yes, 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. So 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. Therefore 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.

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

$$Der\,c\,A \stackrel{\text{def}}{=} \{s \mid c :: s \in A\}$$

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

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

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	$(r_4 = der\,b\,r_3)$
Step 4:	the string is exhausted; test whether r_4 can recognise the empty string	$(nullable(r_4))$
Output:	result of the 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. 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 $Der\,c\,(Der\,b\,(Der\,a\,(L(r))))$. Now if *abc* was in the original set ($L(r_1)$), then in $Der\,c\,(Der\,b\,(Der\,a\,(L(r))))$ must be 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 expression instead of sets. For this 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 regular expression as input and a regular expression as output.

$$\begin{array}{lcl} ders\,[]\,r & \stackrel{\text{def}}{=} & r \\ ders\,(c::s)\,r & \stackrel{\text{def}}{=} & ders\,s\,(der\,c\,r) \end{array}$$

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

$$matches\,s\,r \stackrel{\text{def}}{=} nullable(ders\,s\,r)$$

and we can claim that

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

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, which for example is the point of the Strand-2 Coursework.

This algorithm was introduced by Janus Brzozowski in 1964. Its main attractions are simplicity and being fast, as well as being easily extendable for other regular expressions such as $r^{\{n\}}$, $r^?$, $\sim r$ and so on (this is subject of Strand-1 Coursework 1).

The Matching Algorithm in Scala

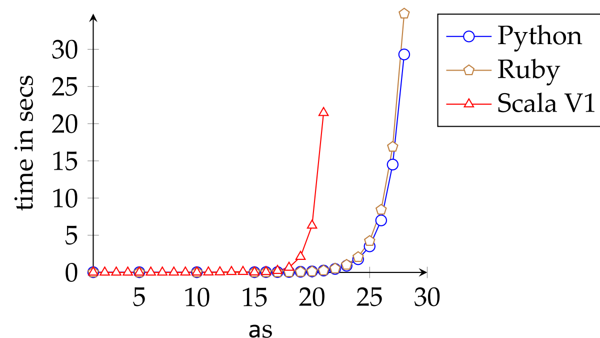
Another attraction of the algorithm is that it can be easily implemented in a functional programming language, like Scala. Given the implementation of regular expressions in Scala shown in the first lecture and handout, the functions and subfunctions for `matches` are shown in Figure 1.

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

```
def OPT(r: Rexp) = ALT(r, EMPTY)

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

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



Analysing this failure we notice that for $a^{\{n\}}$ we generate quite big regular expressions:

```scala
def nullable (r: Rexp) : Boolean = r match {
  case NULL => false
  case EMPTY => 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 NULL => NULL
  case EMPTY => NULL
  case CHAR(d) => if (c == d) EMPTY else NULL
  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 matches(r: Rexp, s: String) : Boolean =
  nullable(ders(s.toList, r))
```

Figure 1: Scala implementation of the nullable and derivatives functions. These functions are easy to implement in functional languages, because pattern matching and recursion allow us to mimic the mathematical definitions very closely.

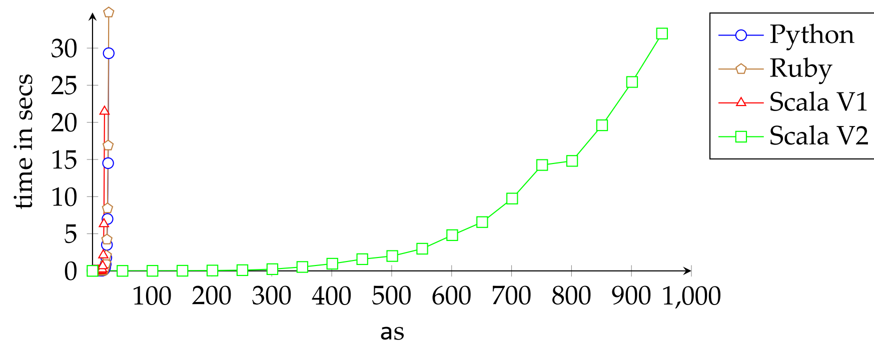
1:	a
2:	$a \cdot a$
3:	$a \cdot a \cdot a$
	$\ldots$
13:	$a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a$
	$\ldots$

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 $a + \epsilon$.

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

```
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. This means we have to also add cases for `NTIMES` in the functions *nullable* and *der*. Does the change have any effect?



Now we are talking business! The modified matcher can within 30 seconds handle regular expressions up to $n = 950$ before a StackOverflow is raised.

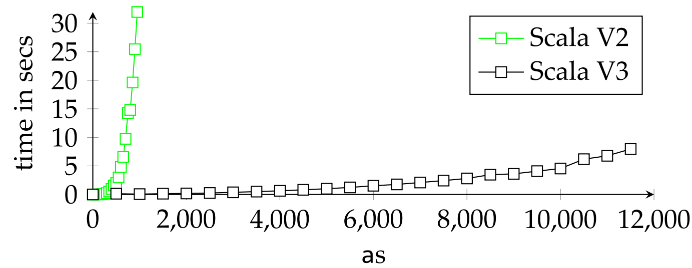
The moral is that our algorithm is rather sensitive to the size of regular expressions it needs to handle. This is of course obvious because both *nullable* and *der* frequently need to traverse the whole regular expression. There seems, however, one more issue for making the algorithm run faster. The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

$$\begin{aligned} der\, a\, r &= ((\epsilon \cdot b) + \varnothing) \cdot r \\ der\, b\, r &= ((\varnothing \cdot b) + \epsilon) \cdot r \\ der\, c\, r &= ((\varnothing \cdot b) + \varnothing) \cdot r \end{aligned}$$

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

$$der\, a\, r \equiv b \cdot r$$
$$der\, b\, r \equiv r$$
$$der\, c\, r \equiv \varnothing$$

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 $+$, $\cdot$ and n-times (the latter because we added it in the second version of our matcher). There is no 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!!



```scala
def simp(r: Rexp): Rexp = r match {
  case ALT(r1, r2) => {
    val r1s = simp(r1)
    val r2s = simp(r2)
    (r1s, r2s) match {
      case (NULL, _) => r2s
      case (_, NULL) => r1s
      case _ => if (r1s == r2s) r1s else ALT(r1s, r2s)
    }
  }
  case SEQ(r1, r2) => {
    val r1s = simp(r1)
    val r2s = simp(r2)
    (r1s, r2s) match {
      case (NULL, _) => NULL
      case (_, NULL) => NULL
      case (EMPTY, _) => r2s
      case (_, EMPTY) => r1s
      case _ => SEQ(r1s, r2s)
    }
  }
  case NTIMES(r, n) => NTIMES(simp(r), n)
  case r => r
}

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

Figure 2: The simplification function and modified `ders`-function; this function now calls `der` first, but then tries to simplify the resulting derivative regular expressions.