Handout 2

Having specified what problem our matching algorithm, $match$, is supposed to solve, namely for a given regular expression r and string s answer $true$ if and only if

$$s \in L(r)$$

Clearly we cannot use the function L directly in order to solve this problem, because in general the set of strings L returns is infinite (recall what $L(a^*)$ is). In such cases there is no algorithm then can test exhaustively, whether a string is member of this set.

The algorithm we 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). 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:

$$nullable(r) \;\; \text{if and only if} \;\; \text{""} \in L(r)$$

On the left-hand side we have a function we can implement; on the right we have its specification.

The other function is calculating 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. Beware 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, if the "input" regular expression can match a string of the form $c\!::\!s$. Since neither $\varnothing$ nor ϵ can match such a string 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. The $+$-case is 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 $+$. 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. 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.

Another way to rationalise the definition of der is to consider the following operation on sets:

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

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

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

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 ($L(r)$), 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.

For our matching algorithm we need to lift the notion of derivatives from 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.

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

Having $ders$ in place, we can finally define our matching algorithm:

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

We claim that

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

holds, which means our algorithm satisfies the specification. This algorithm can be easily extended for other regular expressions such as $r^{\{n\}}$, $r^?$, $\sim r$ and so on.

```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
}
```

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

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

Figure 1: Scala implementation of the nullable and derivatives functions.