# 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:

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

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 $s$.