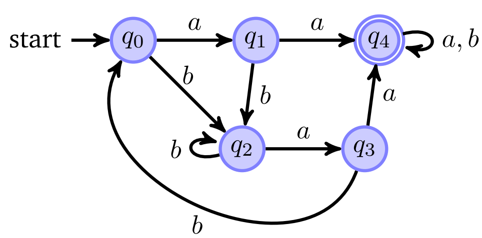
Handout 3

Let us have a closer look at automata and their relation to regular expressions. This will help us to understand why the regular expression matchers in Python and Ruby are so slow with certain regular expressions.

A *deterministic finite automaton* (DFA), say A, is defined by a four-tuple written $A(Q, q_0, F, \delta)$ where

- Q is a set of states,
- $q_0 \in Q$ is the start state,
- $F \subseteq Q$ are the accepting states, and
- δ is the transition function.

The transition function determines how to "transition" from one state to the next state with respect to a character. We have the assumption that these functions do not need to be defined everywhere: so it can be the case that given a character there is no next state, in which case we need to raise a kind of "raise an exception". A typical example of a DFA is



The accepting state q_4 is indicated with double circles. It is possible that a DFA has no accepting states at all, or that the starting state is also an accepting state. In the case above the transition function is defined everywhere and can be given as a table as follows:

$$
\begin{array}{lcl}
(q_0, a) & \rightarrow & q_1 \\
(q_0, b) & \rightarrow & q_2 \\
(q_1, a) & \rightarrow & q_4 \\
(q_1, b) & \rightarrow & q_2 \\
(q_2, a) & \rightarrow & q_3 \\
(q_2, b) & \rightarrow & q_2 \\
(q_3, a) & \rightarrow & q_4 \\
(q_3, b) & \rightarrow & q_0 \\
(q_4, a) & \rightarrow & q_4 \\
(q_4, b) & \rightarrow & q_4
\end{array}
$$

We need to define the notion of what language is accepted by an automaton. For this we lift the transition function δ from characters to strings as follows:

$$
\begin{array}{lcl}
\hat{\delta}(q, ``") & \stackrel{\text{def}}{=} & q \\
\hat{\delta}(q, c\!::\!s) & \stackrel{\text{def}}{=} & \hat{\delta}(\delta(q, c), s)
\end{array}
$$

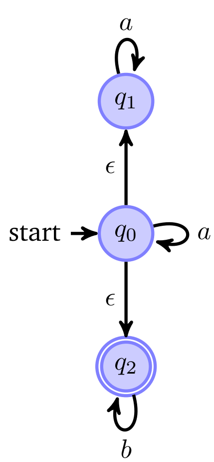
Given a string this means we start in the starting state and take the first character of the string, follow to the next sate, then take the second character and so on. Once the string is exhausted and we end up in an accepting state, then this string is accepted. Otherwise it is not accepted. So s in the *language accepted by the automaton* $A(Q, q_0, F, \delta)$ iff

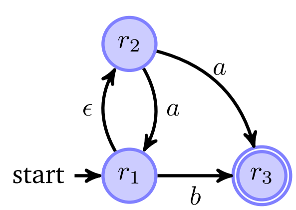
$$\hat{\delta}(q_0, s) \in F$$

While with DFA it will always clear that given a character what the next state is, it will be useful to relax this restriction. The resulting construction is called a *non-deterministic finite automaton* (NFA) given as a four-tuple $A(Q, q_0, F, \rho)$ where

- Q is a finite set of states
- q_0 is a start state
- F are some accepting states with $F \subseteq Q$, and
- ρ is a transition relation.

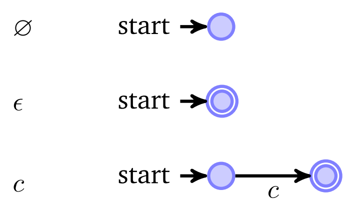
Two typical examples of NFAs are



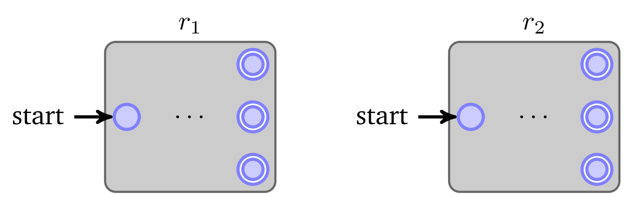


There are a number of points you should note. Every DFA is a NFA, but not vice versa. The ρ in NFAs is a transition *relation* (DFAs have a transition function). The difference between a function and a relation is that a function has always a single output, while a relation gives, roughly speaking, several outputs. Look at the NFA on the right-hand side above: if you are currently in the state r_2 and you read a character a, then you can transition to r_1 *or* r_3. Which route you take is not determined. This means if we need to decide whether a string is accepted by a NFA, we might have to explore all possibilities. Also there is a special transition in NFAs which is called *epsilon-transition* or *silent transition*. This transition means you do not have to "consume" no part of the input string, but "silently" change to a different state.

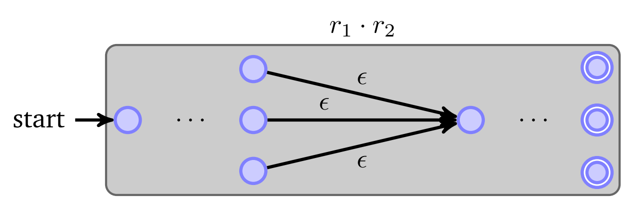
The reason for introducing NFAs is that there is a relatively simple (recursive) translation of regular expressions into NFAs. Consider the simple regular expressions $\varnothing$, ϵ and c. They can be translated as follows:



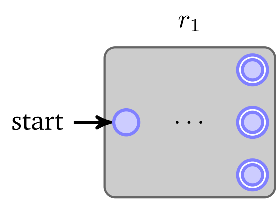
The case for the sequence regular expression $r_1 \cdot r_2$ is as follows: We are given by recursion two automata representing r_1 and r_2 respectively.

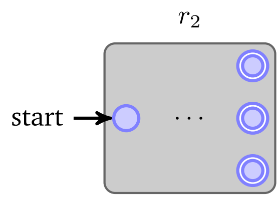


The first automaton has some accepting states. We obtain an automaton for $r_1 \cdot r_2$ by connecting these accepting states with ϵ-transitions to the starting state of the second automaton. By doing so we make them non-accepting like so:

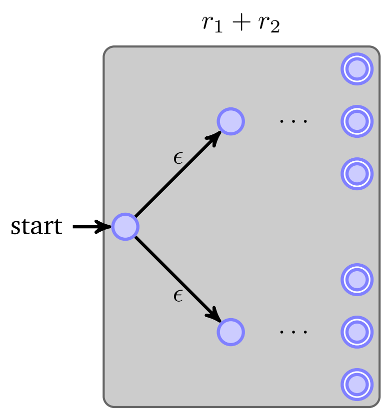


The case for the choice regular expression $r_1 + r_2$ is slightly different: We are given by recursion two automata representing r_1 and r_2 respectively.

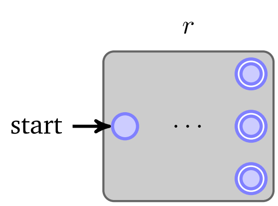




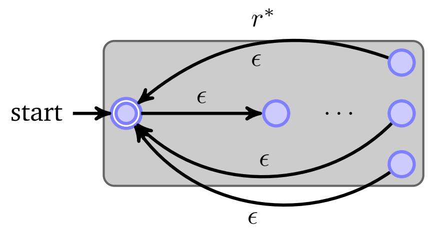
Each automaton has a single start state and potentially several accepting states. We obtain a NFA for the regular expression $r_1 + r_2$ by introducing a new starting state and connecting it with an ϵ-transition to the two starting states above, like so



Finally for the $*$-case we have an automaton for r



and connect its accepting states to a new starting state via ϵ-transitions. This new starting state is also an accepting state, because r^* can also recognise the empty string. This gives the following automaton for r^*:


r^*
start
ϵ
ϵ
$\cdots$
ϵ
ϵ