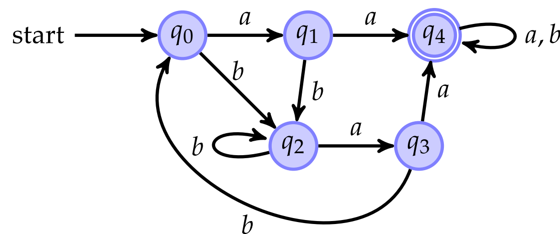
Handout 3 (Automata)

Every formal language course I know of bombards you first with automata and then to a much, much smaller extend with regular expressions. As you can see, this course is turned upside down: regular expressions come first. The reason is that regular expressions are easier to reason about and the notion of derivatives, although already quite old, only became more widely known rather recently. Still let us in this lecture have a closer look at automata and their relation to regular expressions. This will help us with understanding why the regular expression matchers in Python and Ruby are so slow with certain regular expressions. The central definition is:

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

- Q is a finite 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 transition 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 "failure exception". A typical example of a DFA is



In this graphical notation, the accepting state q_4 is indicated with double circles. Note that there can be more than one accepting state. It is also 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:

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

This lifted transition function is often called "delta-hat". Given a string, 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 by the automaton. Otherwise it is not accepted. So s is in the *language accepted by the automaton* $A(Q, q_0, F, \delta)$ iff

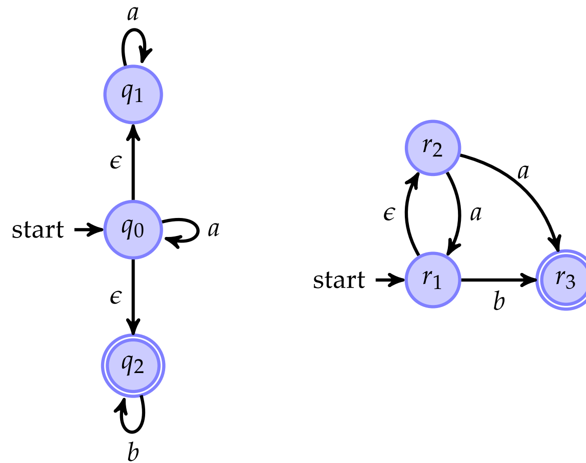
$$\hat{\delta}(q_0, s) \in F$$

I let you think about a definition that describes the set of strings accepted by an automaton.

While with DFAs it will always clear that given a character what the next state is (potentially none), it will be useful to relax this restriction. That means we have several potential successor states. We even allow "silent transitions", also called epsilon-transitions. They allow us to go from one state to the next without having a character consumed. We label such silent transition with the letter ϵ. The resulting construction is called a *non-deterministic finite automaton* (NFA) given also 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.

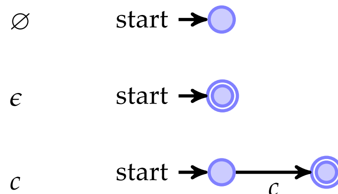
Two typical examples of NFAs are



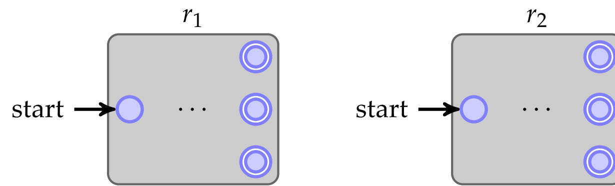
There are, however, a number of points you should note. Every DFA is a NFA, but not vice versa. The ρ in NFAs is a transition *relation* (DFAs have a transition function). The difference between a function and a relation is that a function has always a single output, while a relation gives, roughly speaking, 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 either 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 the special silent transition in NFAs. As mentioned already this transition means you do not have to "consume" any part of the input string, but "silently" change to a different state. In the left picture, for example, if you are in the starting state, you can silently move either to q_1 or q_2.

Thompson Construction

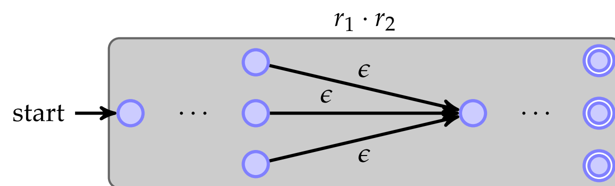
The reason for introducing NFAs is that there is a relatively simple (recursive) translation of regular expressions into NFAs. Consider the simple regular expressions $\varnothing$, ϵ and c. They can be translated as follows:



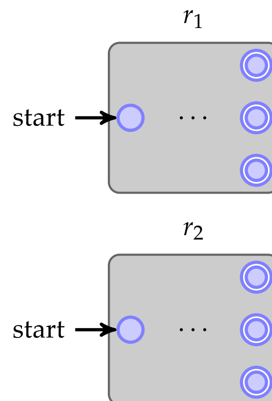
The case for the sequence regular expression $r_1 \cdot r_2$ is as follows: We are given by recursion two automata representing r_1 and r_2 respectively.



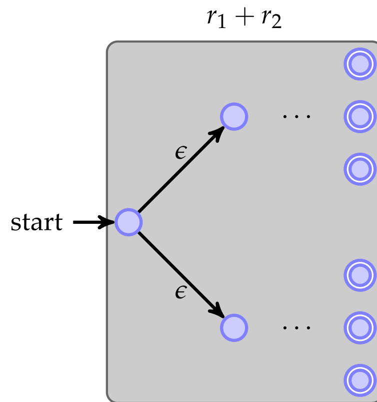
The first automaton has some accepting states. We obtain an automaton for $r_1 \cdot r_2$ by connecting these accepting states with ϵ-transitions to the starting state of the second automaton. By doing so we make them non-accepting like so:



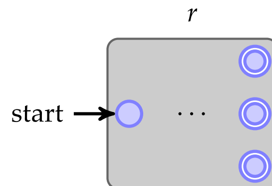
The case for the choice regular expression $r_1 + r_2$ is slightly different: We are given by recursion two automata representing r_1 and r_2 respectively.



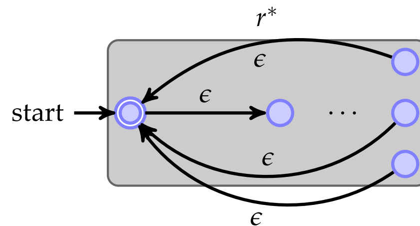
Each automaton has a single start state and potentially several accepting states. We obtain a NFA for the regular expression $r_1 + r_2$ by introducing a new starting state and connecting it with an ϵ-transition to the two starting states above, like so



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



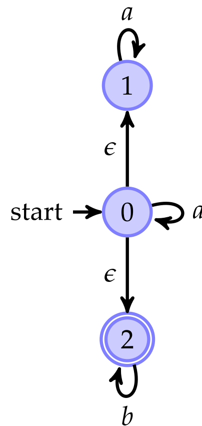
and connect its accepting states to a new starting state via ϵ-transitions. This new starting state is also an accepting state, because r^* can recognise the empty string. This gives the following automaton for r^*:



This construction of a NFA from a regular expression was invented by Ken Thompson in 1968.

Subset Construction

What is interesting that for every NFA we can find a DFA which recognises the same language. This can be done by the *subset construction*. Consider again the NFA on the left, consisting of nodes labeled 0, 1 and 2.



nodes	a	b
$\varnothing$	$\varnothing$	$\varnothing$
$\{0\}$	$\{0,1,2\}$	$\{2\}$
$\{1\}$	$\{1\}$	$\varnothing$
$\{2\}\star$	$\varnothing$	$\{2\}$
$\{0,1\}$	$\{0,1,2\}$	$\{2\}$
$\{0,2\}\star$	$\{0,1,2\}$	$\{2\}$
$\{1,2\}\star$	$\{1\}$	$\{2\}$
s: $\{0,1,2\}\star$	$\{0,1,2\}$	$\{2\}$

The nodes of the DFA are given by calculating all subsets of the set of nodes of the NFA (seen in the nodes column on the right). The table shows the transition function for the DFA. The first row states that $\varnothing$ is the sink node which has transitions for a and b to itself. The next three lines are calculated as follows:

- suppose you calculate the entry for the transition for a and the node $\{0\}$
- start from the node 0 in the NFA
- do as many ϵ-transition as you can obtaining a set of nodes, in this case $\{0,1,2\}$
- filter out all notes that do not allow an a-transition from this set, this excludes 2 which does not permit a a-transition
- from the remaining set, do as many ϵ-transition as you can, this yields $\{0,1,2\}$
- the resulting set specifies the transition from $\{0\}$ when given an a

Similarly for the other entries in the rows for $\{0\}$, $\{1\}$ and $\{2\}$. The other rows are calculated by just taking the union of the single node entries. For example for a and $\{0,1\}$ we need to take the union of $\{0,1,2\}$ (for 0) and $\{1\}$ (for 1). The starting state of the DFA can be calculated from the starting state of the NFA, that is 0, and then do as many ϵ-transitions as possible. This gives $\{0,1,2\}$ which is the starting state of the DFA. One terminal states in the DFA are given by all sets that contain a 2, which is the terminal state of the NFA. This completes the subset construction.

There are two points to note: One is that the resulting DFA contains a number of "dead" nodes that are never reachable from the starting state (that is that the calculated DFA in this example is not a minimal DFA). Such dead nodes can be safely removed without changing the language that is recognised by the DFA. Another point is that in some cases the subset construction produces a DFA that does *not* contain any dead nodes…that means it calculates a minimal

DFA. Which in turn means that in some cases the number of nodes by going from NFAs to DFAs exponentially increases, namely by 2^n (which is the number of subsets you can form for n nodes).

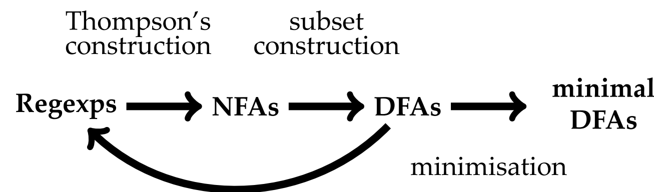
Brzozowski's Method

DFA -> NFA

Automata Minimization

Regular Languages

Given the constructions in the previous sections we obtain the following picture:



By going from regular expressions over NFAs to DFAs, we can always ensure that for every regular expression there exists a NFA and DFA that can recognise the same language. Although we did not prove this fact. Similarly by going from DFAs to regular expressions, we can make sure for every DFA there exists a regular expression that can recognise the same language. Again we did not prove this fact.

The interesting conclusion is that automata and regular expressions can recognise the same set of languages:

> A language is *regular* iff there exists a regular expression that recognises all its strings.

or equivalently

> A language is *regular* iff there exists an automaton that recognises all its strings.

So for deciding whether a string is recognised by a regular expression, we could use our algorithm based on derivatives or NFAs or DFAs. But let us quickly look at what the differences mean in computational terms. Translating a regular expression into a NFA gives us an automaton that has $O(n)$ nodes—that means the size of the NFA grows linearly with the size of the regular expression. The problem with NFAs is that the problem of deciding whether a string

is accepted is computationally not cheap. Remember with NFAs we have potentially many next states even for the same input and also have the silent ϵ-transitions. If we want to find a path from the starting state of an NFA to an accepting state, we need to consider all possibilities. In Ruby and Python this is done by a depth-first search, which in turn means that if a "wrong" choice is made, the algorithm has to backtrack and thus explore all potential candidates. This is exactly the reason why Ruby and Python are so slow for evil regular expressions. The alternative is to explore the search space in a breadth-first fashion, but this might incur a memory penalty.

To avoid the problems with NFAs, we can translate them into DFAs. With DFAs the problem of deciding whether a string is recognised or not is much simpler, because in each state it is completely determined what the next state will be for a given input. So no search is needed. The problem with this is that the translation to DFAs can explode exponentially the number of states. Therefore when this route is taken, we definitely need to minimise the resulting DFAs in order to have an acceptable memory and runtime behaviour.

But this does not mean that everything is bad with automata. Recall the problem of finding a regular expressions for the language that is *not* recognised by a regular expression. In our implementation we added explicitly such a regular expressions because they are useful for recognising comments. But in principle we did not need to. The argument for this is as follows: take a regular expression, translate it into a NFA and DFA that recognise the same language. Once you have the DFA it is very easy to construct the automaton for the language not recognised by an DFA. If the DAF is completed (this is important!), then you just need to exchange the accepting and non-accepting states. You can then translate this DFA back into a regular expression.