Handout 3 (Finite Automata)

Every formal language and compiler 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, Ruby and Java are so slow with certain regular expressions.

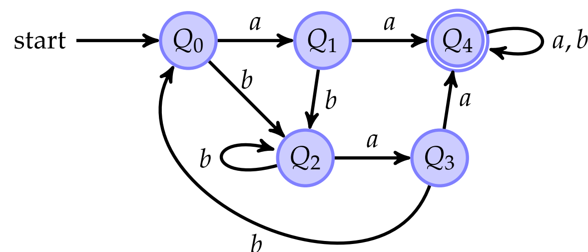
Deterministic Finite Automata

The central definition is:

A *deterministic finite automaton* (DFA), say A, is given by a five-tuple written $\mathcal{A}(\Sigma, Qs, Q_0, F, \delta)$ where

- Σ is an alphabet,
- Qs is a finite set of states,
- $Q_0 \in Qs$ is the start state,
- $F \subseteq Qs$ 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". That means actually we have *partial* functions as transitions—see the Scala implementation of DFAs later on. A typical example of a DFA is



© Christian Urban, King's College London, 2014, 2015, 2016, 2017

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 also 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:

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

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 state, 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. This also means that if along the way we hit the case where the transition function δ is not defined, we need to raise an error. In our implementation we will deal with this case elegantly by using Scala's `Try`. So a string s is in the *language accepted by the automaton* $\mathcal{A}(\Sigma, Q, Q_0, F, \delta)$ iff

$$\widehat{\delta}(Q_0, s) \in F$$

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

My take of a simple Scala implementation for DFAs is given in Figure 1. As you can see, there are many features of the mathematical definition that are quite closely reflected in the code. In the DFA-class, there is a starting state, called `start`, with the polymorphic type `A`. There is a partial function `delta` for specifying the transitions—these partial functions take a state (of polymorphic type `A`) and an input (of polymorphic type `C`) and produce a new state (of type `A`). For the moment it is OK to assume that `A` is some arbitrary type for states and the input is just characters. (The reason for having polymorphic types for the states and the input of DFAs will become clearer later on.)

The most important point in this implemnetation is that I use Scala's partial functions for representing the transitions; alternatives would have been `Maps`

```scala
// DFAs in Scala based on partial functions
import scala.util.Try

// type abbreviation for partial functions
type :=>[A, B] = PartialFunction[A, B]

case class DFA[A, C](start: A,               // starting state
                     delta: (A, C) :=> A,    // transition (partial fun)
                     fins:  A => Boolean) {  // final states

  def deltas(q: A, s: List[C]) : A = s match {
    case Nil => q
    case c::cs => deltas(delta(q, c), cs)
  }

  def accepts(s: List[C]) : Boolean =
    Try(fins(deltas(start, s))) getOrElse false
}

// the example shown earlier in the handout
abstract class State
case object Q0 extends State
case object Q1 extends State
case object Q2 extends State
case object Q3 extends State
case object Q4 extends State

val delta : (State, Char) :=> State =
  { case (Q0, 'a') => Q1
    case (Q0, 'b') => Q2
    case (Q1, 'a') => Q4
    case (Q1, 'b') => Q2
    case (Q2, 'a') => Q3
    case (Q2, 'b') => Q2
    case (Q3, 'a') => Q4
    case (Q3, 'b') => Q0
    case (Q4, 'a') => Q4
    case (Q4, 'b') => Q4 }

val dfa = DFA(Q0, delta, Set[State](Q4))

dfa.accepts("bbabaab".toList)   // true
dfa.accepts("baba".toList)      // false
```

Figure 1: A Scala implementation of DFAs using partial functions. Notice some subtleties: `deltas` implements the delta-hat construction by lifting the transition (partial) function to lists of characters. Since `delta` is given as a partial function, it can obviously go "wrong" in which case the `Try` in `accepts` catches the error and returns `false`—that means the string is not accepted. The example `delta` implements the DFA example shown earlier in the handout.

or even `Lists`. One of the main advantages of using partial functions is that transitions can be quite nicely defined by a series of `case` statements (see Lines 28 – 38 for an example). If you need to represent an automaton with a sink state (catch-all-state), you can use Scala's pattern matching and write something like

```
abstract class State
...
case object Sink extends State

val delta : (State, Char) :=> State =
  { case (S0, 'a') => S1
    case (S1, 'a') => S2
    case _ => Sink
  }
```

I let you think what this DFA looks like in the graphical notation.

The DFA-class has also an argument for specifying final states. In the implementation it not a set of states, as in the matemathical definition, but a function from states to booleans (this function is supposed to return true whenever a state is final; false otherwise). While this boolean function is different from the sets of states, Scala allows to use sets for such functions (see Line 40 where the DFA is initialised). Again it will become clear later on why I use functions for final states, rather than sets.

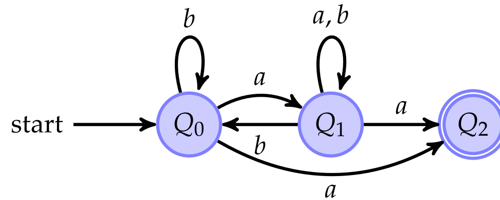
I let you ponder whether this is a good implementation of DFAs. In doing so I hope you notice that the Σ and Qs components (the alphabet and the set of finite states, respectively) are missing from the class definition. This means that the implementation allows you to do some fishy things you are not meant to do with DFAs. Which fishy things could that be?

Non-Deterministic Finite Automata

While with DFAs it is always be clear that given a state and a character what the next state is (potentially none), it will be useful to relax this restriction. That means we allow states to have several potential successor states. We even allow more than one starting state. The resulting construction is called a *Non-Deterministic Finite Automaton* (NFA) given also as a five-tuple $\mathcal{A}(\Sigma, Qs, Q_{0s}, F, \rho)$ where

- Σ is an alphabet,
- Qs is a finite set of states
- Q_{0s} is a set of start states ($Q_{0s} \subseteq Qs$)
- F are some accepting states with $F \subseteq Qs$, and
- ρ is a transition relation.

A typical example of a NFA is



This NFA happens to have only one starting state, but in general there could be more. Notice that in state Q_0 we might go to state Q_1 *or* to state Q_2 when receiving an a. Similarly in state Q_1 and receiving an a, we can stay in state Q_1 *or* go to Q_2. This kind of choice is not allowed with DFAs. The downside of this choice is that when it comes to deciding whether a string is accepted by a NFA we potentially have to explore all possibilities. I let you think which kind of strings the above NFA accepts.

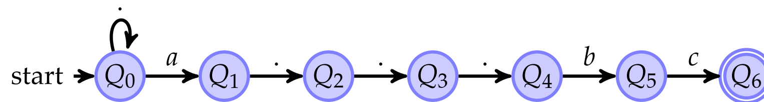
There are a number of additional points you should note with NFAs. Every DFA is a NFA, but not vice versa. The ρ in NFAs is a transition *relation* (DFAs have a transition function). The difference between a function and a relation is that a function has always a single output, while a relation gives, roughly speaking, several outputs. Look again at the NFA above: if you are currently in the state Q_1 and you read a character b, then you can transition to either Q_0 *or* Q_2. Which route, or output, you take is not determined. This non-determinism can be represented by a relation.

My implementation of NFAs in Scala is shown in Figure 2. Perhaps interestingly, I do not actually use relations for my NFAs, and I also do not use transition functions that return sets of states (another popular choice for implementing NFAs). For reasons that become clear in a moment, I use sets of partial functions instead—see Line 7 in Figure 2. DFAs have only one such partial function; my NFAs have a set. Another parameter, `starts`, is in NFAs a set of states; `fins` is again a function from states to booleans. The `next` function calculates the set of next states reachable from a single state `q` by a character `c`—this is calculated by going through all the partial functions in the `delta`-set and apply `q` and `c` (see Line 13). This gives a set of `Some`s (in case the application succeeded) and possibly some `None`s (in case the partial function is not defined or produces an error). The `None`s are filtered out by the `flatMap`, leaving the values inside the `Some`s. The function `nexts` just lifts this function to sets of states. `Deltas` and `accept` are similar to the DFA definitions.

The reason for using sets of partial functions for specifying the transitions in NFAs has to do with pattern matching. Consider the following example: a popular benchmark regular expression is $(.)^* \cdot a \cdot (.)^{\{n\}} \cdot b \cdot c$. The important point to note is that it uses . in order to represent the regular expression that accepts any character. A NFA that accepts the same strings as this regular expression (for $n = 3$) is as follows:

```scala
// NFAs in Scala based on sets of partial functions

// type abbreviation for partial functions
type :=>[A, B] = PartialFunction[A, B]

case class NFA[A, C](starts: Set[A],           // starting states
                     delta: Set[(A, C) :=> A],  // transitions
                     fins:  A => Boolean) {     // final states

  // given a state and a character, what is the set of next states?
  // if there is none => empty set
  def next(q: A, c: C) : Set[A] =
    delta.flatMap(d => Try(d(q, c)).toOption)

  def nexts(qs: Set[A], c: C) : Set[A] =
    qs.flatMap(next(_, c))

  def deltas(qs: Set[A], s: List[C]) : Set[A] = s match {
    case Nil => qs
    case c::cs => deltas(nexts(qs, c), cs)
  }

  def accepts(s: List[C]) : Boolean =
    deltas(starts, s).exists(fins)
}
```

Figure 2: A Scala implementation of NFAs using sets of partial functions. Notice some subtleties: Since `delta` is given as a set of partial functions, each of them can obviously go "wrong" in which case the `Try`. The function `accepts` implements the acceptance of a string in a breath-first fashion. This can be costly way of deciding whether a string is accepted in practical contexts.



Also here the . stands for accepting any single character: for example if we are in Q_0 and read an a we can either stay in Q_0 (since any character will do for this) or advance to Q_1 (but only if it is an a). Why this is a good benchmark regular expression is irrelevant here. The point is that this NFA can be conveniently represented by the code:

```
val delta = Set[(State, Char) :=> State](
  { case (Q0, 'a') => Q1 },
  { case (Q0, _)   => Q0 },
  { case (Q1, _)   => Q2 },
  { case (Q2, _)   => Q3 },
  { case (Q3, _)   => Q4 },
  { case (Q4, 'b') => Q5 },
  { case (Q5, 'c') => Q6 }
)

NFA(Set[State](Q0), delta, Set[State](Q6))
```

where the .-transitions translate into a underscore-pattern-matching. Recall that in Q_0 if we read an a we can go to Q_1 (by the first partial function in the set) and also stay in Q_0 (by the second partial function). Representing such transitions in any other way in Scala seems to be somehow awkward; the set of partial function representation makes them easy to implement.

Look very careful again at the `accepts` and `deltas` functions in NFAs and remember that when accepting a string by an NFA we might have to explore all possible transitions (recall which state to go to is not unique anymore with NFAs). The implementation achieves this exploration in a *breadth-first search* manner. This is fine for very small NFAs, but can lead to problems when the NFAs are bigger. Take for example the regular expression $(.)^* \cdot a \cdot (.)^{\{n\}} \cdot b \cdot c$ from above. If n is large, say 100 or 1000, then the corresponding NFA will have 104, respectively 1004, nodes. The problem is that with certain strings this can lead to 1000 "active" nodes in the breadth-first search, all of which we need to analyse when determining the next states. This can be a real memory strain in practical applications. As result, some regular expression matching engines resort to a *depth-first search* with *backtracking* in unsuccessful cases. In our implementation we can implement a depth-first version of `accepts` using Scala's `exists` as follows:

```
def search(q: A, s: List[C]) : Boolean = s match {
  case Nil => fins(q)
  case c::cs =>
    delta.exists(d => Try(search(d(q, c), cs)) getOrElse false)
}

def accepts(s: List[C]) : Boolean =
  starts.exists(search(_, s))
```

This depth-first way of exploration seems to work efficiently in many examples and is much less of strain on memory. The problem is that the backtracking can get "catastrophic" in some examples—remember the catastrophic backtracking from earlier lectures. This depth-first search with backtracking is the reason for the abysmal performance of some regular expression macthings in Java, Ruby and Python. I like to show you this next.

Thompson Construction

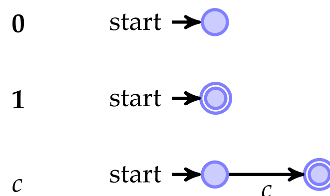
In order to get an idea what calculations are done in Java & friends, we need a method for translating regular expressions into automata. The simplest and most well-known method is called *Thompson Construction*, after the Turing Award winner Ken Thompson who implemented this method in early versions of grep????

The reason for introducing NFAs is that there is a relatively simple (recursive) translation of regular expressions into NFAs. Consider the simple regular expressions **0**, **1** and c. They can be translated as follows:

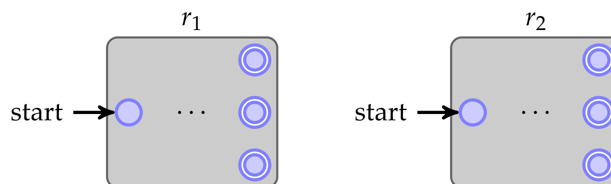
0

1

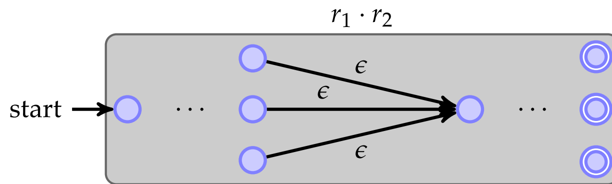
c



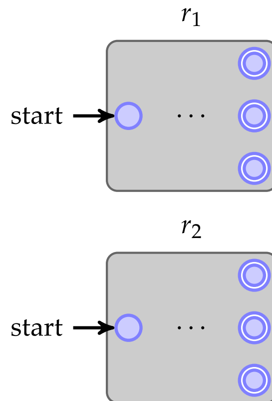
The case for the sequence regular expression $r_1 \cdot r_2$ is as follows: We are given by recursion two automata representing r_1 and r_2 respectively.



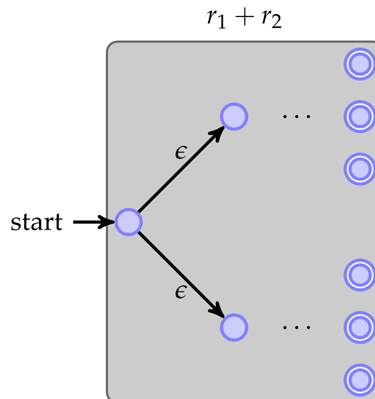
The first automaton has some accepting states. We obtain an automaton for $r_1 \cdot r_2$ by connecting these accepting states with ϵ-transitions to the starting state of the second automaton. By doing so we make them non-accepting like so:



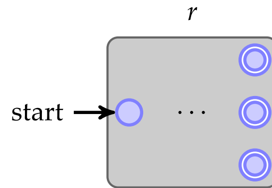
The case for the choice regular expression $r_1 + r_2$ is slightly different: We are given by recursion two automata representing r_1 and r_2 respectively.



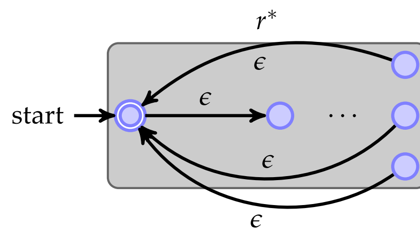
Each automaton has a single start state and potentially several accepting states. We obtain a NFA for the regular expression $r_1 + r_2$ by introducing a new starting state and connecting it with an ϵ-transition to the two starting states above, like so



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



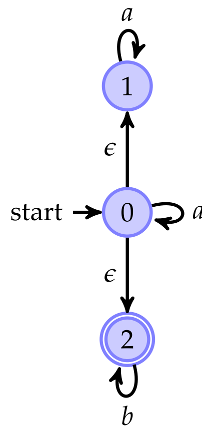
and connect its accepting states to a new starting state via ϵ-transitions. This new starting state is also an accepting state, because r^* can recognise the empty string. This gives the following automaton for r^*:



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

Subset Construction

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

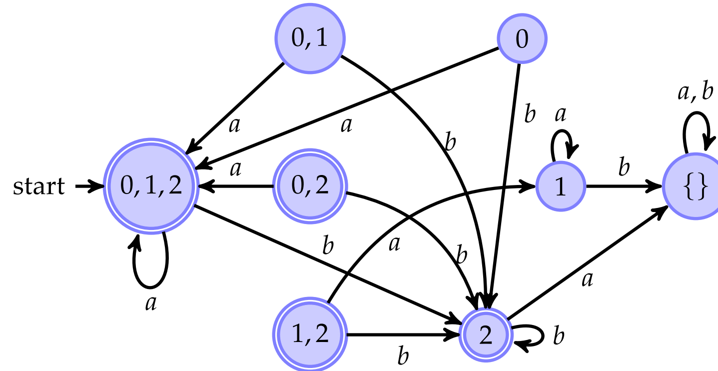


nodes	a	b
$\{\}$	$\{\}$	$\{\}$
$\{0\}$	$\{0,1,2\}$	$\{2\}$
$\{1\}$	$\{1\}$	$\{\}$
$\{2\}\star$	$\{\}$	$\{2\}$
$\{0,1\}$	$\{0,1,2\}$	$\{2\}$
$\{0,2\}\star$	$\{0,1,2\}$	$\{2\}$
$\{1,2\}\star$	$\{1\}$	$\{2\}$
s: $\{0,1,2\}\star$	$\{0,1,2\}$	$\{2\}$

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 $\{\}$ is the sink node which has transitions for a and b to itself. The next three lines are calculated as follows:

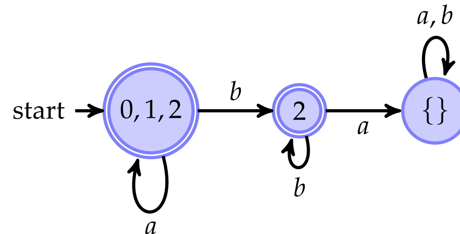
- suppose you calculate the entry for the transition for a and the node $\{0\}$
- start from the node 0 in the NFA
- do as many ϵ-transition as you can obtaining a set of nodes, in this case $\{0, 1, 2\}$
- filter out all notes that do not allow an a-transition from this set, this excludes 2 which does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

So the transition from the state $\{0\}$ reading an a goes to the state $\{0, 1, 2\}$. 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. The 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. So the corresponding DFA to the NFA from above is



There are two points to note: One is that very often the resulting DFA contains a number of "dead" nodes that are never reachable from the starting state. For example there is no way to reach node $\{0, 2\}$ from the starting state $\{0, 1, 2\}$. I let you find the other dead states. In effect the 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, however, 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).

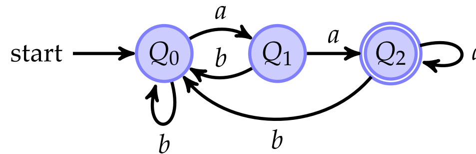
Removing all the dead states in the automaton above, gives a much more legible automaton, namely



Now the big question is whether this DFA can recognise the same language as the NFA we started with. I let you ponder about this question.

Brzozowski's Method

As said before, we can also go into the other direction—from DFAs to regular expressions. Brzozowski's method calculates a regular expression using familiar transformations for solving equational systems. Consider the DFA:



for which we can set up the following equational system

$$Q_0 = \mathbf{1} + Q_0\,b + Q_1\,b + Q_2\,b \quad (1)$$
$$Q_1 = Q_0\,a \quad (2)$$
$$Q_2 = Q_1\,a + Q_2\,a \quad (3)$$

There is an equation for each node in the DFA. Let us have a look how the right-hand sides of the equations are constructed. First have a look at the second equation: the left-hand side is Q_1 and the right-hand side $Q_0\,a$. The right-hand side is essentially all possible ways how to end up in node Q_1. There is only one incoming edge from Q_0 consuming an a. Therefore the right hand side is this state followed by character—in this case $Q_0\,a$. Now lets have a look at the third equation: there are two incoming edges for Q_2. Therefore we have two terms, namely $Q_1\,a$ and $Q_2\,a$. These terms are separated by $+$. The first states that if in state Q_1 consuming an a will bring you to Q_2, and the secont that being in Q_2 and consuming an a will make you stay in Q_2. The right-hand side of the first equation is constructed similarly: there are three incoming edges, therefore there are three terms. There is one exception in that we also "add" $\mathbf{1}$ to the first equation, because it corresponds to the starting state in the DFA.

Having constructed the equational system, the question is how to solve it? Remarkably the rules are very similar to solving usual linear equational systems. For example the second equation does not contain the variable Q_1 on the right-hand side of the equation. We can therefore eliminate Q_1 from the system by just substituting this equation into the other two. This gives

$$Q_0 = \mathbf{1} + Q_0\, b + Q_0\, a\, b + Q_2\, b \tag{4}$$
$$Q_2 = Q_0\, a\, a + Q_2\, a \tag{5}$$

where in Equation (4) we have two occurences of Q_0. Like the laws about $+$ and $\cdot$, we can simplify Equation (4) to obtain the following two equations:

$$Q_0 = \mathbf{1} + Q_0\, (b + a\, b) + Q_2\, b \tag{6}$$
$$Q_2 = Q_0\, a\, a + Q_2\, a \tag{7}$$

Unfortunately we cannot make any more progress with substituting equations, because both (6) and (7) contain the variable on the left-hand side also on the right-hand side. Here we need to now use a law that is different from the usual laws about linear equations. It is called *Arden's rule*. It states that if an equation is of the form $q = q\, r + s$ then it can be transformed to $q = s\, r^*$. Since we can assume $+$ is symmetric, Equation (7) is of that form: s is $Q_0\, a\, a$ and r is a. That means we can transform (7) to obtain the two new equations

$$Q_0 = \mathbf{1} + Q_0\, (b + a\, b) + Q_2\, b \tag{8}$$
$$Q_2 = Q_0\, a\, a\, (a^*) \tag{9}$$

Now again we can substitute the second equation into the first in order to eliminate the variable Q_2.

$$Q_0 = \mathbf{1} + Q_0\, (b + a\, b) + Q_0\, a\, a\, (a^*)\, b \tag{10}$$

Pulling Q_0 out as a single factor gives:

$$Q_0 = \mathbf{1} + Q_0\, (b + a\, b + a\, a\, (a^*)\, b) \tag{11}$$

This equation is again of the form so that we can apply Arden's rule (r is $b + a\, b + a\, a\, (a^*)\, b$ and s is $\mathbf{1}$). This gives as solution for Q_0 the following regular expression:

$$Q_0 = \mathbf{1}\, (b + a\, b + a\, a\, (a^*)\, b)^* \tag{12}$$

Since this is a regular expression, we can simplify away the $\mathbf{1}$ to obtain the slightly simpler regular expression

$$Q_0 = (b + a\,b + a\,a\,(a^*)\,b)^* \quad (13)$$

Now we can unwind this process and obtain the solutions for the other equations. This gives:

$$Q_0 = (b + a\,b + a\,a\,(a^*)\,b)^* \quad (14)$$
$$Q_1 = (b + a\,b + a\,a\,(a^*)\,b)^*\,a \quad (15)$$
$$Q_2 = (b + a\,b + a\,a\,(a^*)\,b)^*\,a\,a\,(a)^* \quad (16)$$

Finally, we only need to "add" up the equations which correspond to a terminal state. In our running example, this is just Q_2. Consequently, a regular expression that recognises the same language as the automaton is

$$(b + a\,b + a\,a\,(a^*)\,b)^*\,a\,a\,(a)^*$$

You can somewhat crosscheck your solution by taking a string the regular expression can match and and see whether it can be matched by the automaton. One string for example is *aaa* and *voila* this string is also matched by the automaton.

We should prove that Brzozowski's method really produces an equivalent regular expression for the automaton. But for the purposes of this module, we omit this.

Automata Minimization

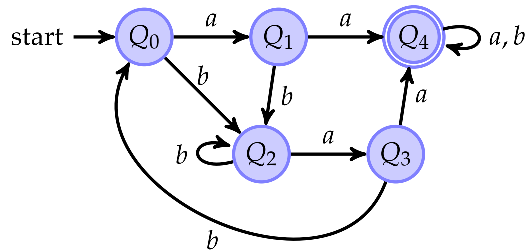
As seen in the subset construction, the translation of a NFA to a DFA can result in a rather "inefficient" DFA. Meaning there are states that are not needed. A DFA can be *minimised* by the following algorithm:

1. Take all pairs (q, p) with $q \neq p$
2. Mark all pairs that accepting and non-accepting states
3. For all unmarked pairs (q, p) and all characters c test whether

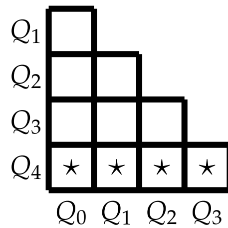
   $$(\delta(q,c), \delta(p,c))$$

   are marked. If there is one, then also mark (q, p).
4. Repeat last step until no change.
5. All unmarked pairs can be merged.

To illustrate this algorithm, consider the following DFA.



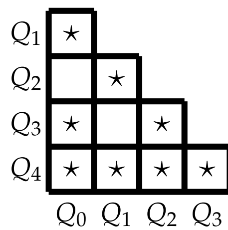
In Step 1 and 2 we consider essentially a triangle of the form



Q_1				
Q_2				
Q_3				
Q_4	$\star$	$\star$	$\star$	$\star$
	Q_0	Q_1	Q_2	Q_3

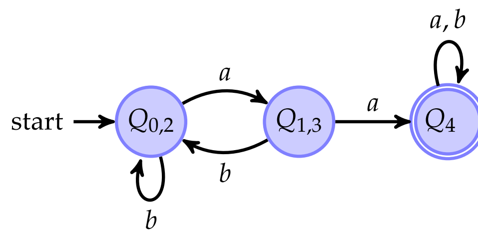
where the lower row is filled with stars, because in the corresponding pairs there is always one state that is accepting (Q_4) and a state that is non-accepting (the other states).

Now in Step 3 we need to fill in more stars according whether one of the next-state pairs are marked. We have to do this for every unmarked field until there is no change anymore. This gives the triangle



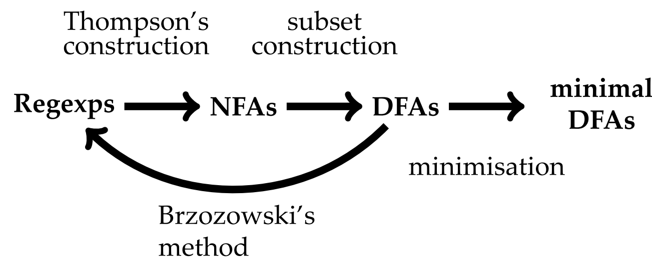
Q_1	$\star$			
Q_2		$\star$		
Q_3	$\star$		$\star$	
Q_4	$\star$	$\star$	$\star$	$\star$
	Q_0	Q_1	Q_2	Q_3

which means states Q_0 and Q_2, as well as Q_1 and Q_3 can be merged. This gives the following minimal DFA



Regular Languages

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



By going from regular expressions over NFAs to DFAs, we can always ensure that for every regular expression there exists a NFA and a 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 or not 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 a 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. An alternative to the potentially slow depth-first search is to explore the search space in a breadth-first fashion, but this might incur a big 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 remember the subset construction in the worst case explodes the number of states by 2^n. Effectively also the translation to DFAs can incur a big runtime penalty.

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 then a DFA that both recognise the same language. Once you have the DFA it is very easy to construct the automaton for the language not recognised by a DFA. If the DFA 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 and that will be the regular expression that can match all strings the original regular expression could *not* match.

It is also interesting that not all languages are regular. The most well-known example of a language that is not regular consists of all the strings of the form

$$a^n\,b^n$$

meaning strings that have the same number of as and bs. You can try, but you cannot find a regular expression for this language and also not an automaton. One can actually prove that there is no regular expression nor automaton for this language, but again that would lead us too far afield for what we want to do in this module.

Further Reading

Compare what a "human expert" would create as an automaton for the regular expression $a(b + c)^*$ and what the Thomson algorithm generates.