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. On the way we will also see what are the limitations of regular expressions. Unfortunately, they cannot be used for *everything*.

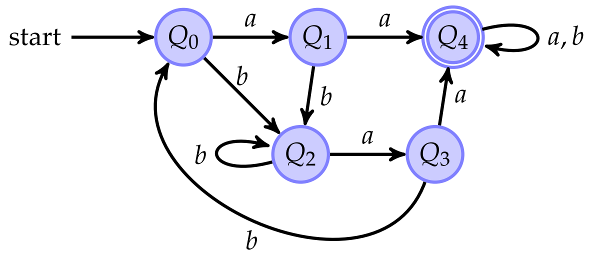
Deterministic Finite Automata

Lets start…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.

I am sure you have seen this defininition already before. 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



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 state 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`. Summing up: 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 a determinsitic finite automaton.

My take on an implementation of DFAs in Scala 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 not having concrete types, but polymorphic types for the states and the input of DFAs will become clearer later on.)

```
1  // DFAs in Scala using partial functions
2  import scala.util.Try
3
4  // type abbreviation for partial functions
5  type :=>[A, B] = PartialFunction[A, B]
6
7  case class DFA[A, C](start: A,               // starting state
8                       delta: (A, C) :=> A,    // transition (partial fun)
9                       fins:  A => Boolean) { // final states
10
11   def deltas(q: A, s: List[C]) : A = s match {
12     case Nil => q
13     case c::cs => deltas(delta(q, c), cs)
14   }
15
16   def accepts(s: List[C]) : Boolean =
17     Try(fins(deltas(start, s))) getOrElse false
18 }
19
20 // the example shown earlier in the handout
21 abstract class State
22 case object Q0 extends State
23 case object Q1 extends State
24 case object Q2 extends State
25 case object Q3 extends State
26 case object Q4 extends State
27
28 val delta : (State, Char) :=> State =
29   { case (Q0, 'a') => Q1
30     case (Q0, 'b') => Q2
31     case (Q1, 'a') => Q4
32     case (Q1, 'b') => Q2
33     case (Q2, 'a') => Q3
34     case (Q2, 'b') => Q2
35     case (Q3, 'a') => Q4
36     case (Q3, 'b') => Q0
37     case (Q4, 'a') => Q4
38     case (Q4, 'b') => Q4 }
39
40 val dfa = DFA(Q0, delta, Set[State](Q4))
41
42 dfa.accepts("bbabaab".toList)   // true
43 dfa.accepts("baba".toList)      // false
```

Figure 1: A Scala implementation of DFAs using partial functions. Note some subtleties: `deltas` implements the delta-hat construction by lifting the (partial) transition 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` in Line 28–38 implements the DFA example shown earlier in the handout.

The DFA-class has also an argument for specifying final states. In the implementation it is not a set of states, as in the mathematical 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.

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

```scala
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 the corresponding DFA looks like in the graph notation.

Finally, I let you ponder whether this is a good implementation of DFAs or not. 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?

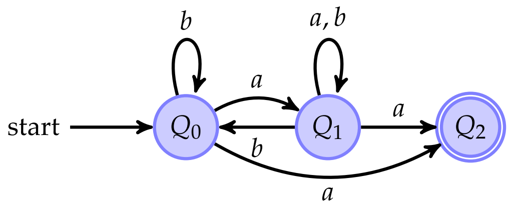
Non-Deterministic Finite Automata

Remember we want to find out what the relation is between regular expressions and automata. To do this with DFAs is a bit unwieldy. While with DFAs it is always clear that given a state and a character what the next state is (potentially none), it will be convenient 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


$$\begin{tikzcd}
\text{start} \arrow[r] & Q_0 \arrow[loop above, "b"] \arrow[r, bend left, "a"] \arrow[rr, bend right, "a"'] & Q_1 \arrow[loop above, "{a,b}"] \arrow[l, "b"] \arrow[r, "a"] & Q_2
\end{tikzcd}$$

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 in NFAs 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 strings the above NFA accepts.

There are a number of additional points you should note about 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, but use transition functions that return sets of states. DFAs have partial transition functions that return a single state; my NFAs return a set of states. I let you think about this representation for NFA-transitions and how it corresponds to the relations used in the mathematical definition of NFAs. An example of a transition function in Scala for the NFA shown above is

```scala
val nfa_delta : (State, Char) :=> Set[State] =
  { case (Q0, 'a') => Set(Q1, Q2)
    case (Q0, 'b') => Set(Q0)
    case (Q1, 'a') => Set(Q1, Q2)
    case (Q1, 'b') => Set(Q0, Q1) }
```

Like in the mathematical definition, `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`. In case there is no such state—the partial transition function is undefined—the empty set is returned (see function `applyOrElse` in Lines 9 and 10). The function `nexts` just lifts this function to sets of states.

Look very careful at the `accepts` and `deltas` functions in NFAs and remember that when accepting a string by a NFA we might have to explore all possible

```scala
// NFAs in Scala using partial functions (returning
// sets of states)
import scala.util.Try

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

// return an empty set when not defined
def applyOrElse[A, B](f: A :=> Set[B], x: A) : Set[B] =
  Try(f(x)) getOrElse Set[B]()


// NFAs
case class NFA[A, C](starts: Set[A],           // starting states
                     delta: (A, C) :=> Set[A], // transition function
                     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] =
    applyOrElse(delta, (q, c))

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

  // given some states and a string, what is the set
  // of next states?
  def deltas(qs: Set[A], s: List[C]) : Set[A] = s match {
    case Nil => qs
    case c::cs => deltas(nexts(qs, c), cs)
  }

  // is a string accepted by an NFA?
  def accepts(s: List[C]) : Boolean =
    deltas(starts, s).exists(fins)
}
```

Figure 2: A Scala implementation of NFAs using partial functions. Notice that the function `accepts` implements the acceptance of a string in a breath-first search fashion. This can be a costly way of deciding whether a string is accepted or not in applications that need to handle large NFAs and large inputs.

transitions (recall which state to go to is not unique anymore with NFAs...we need to explore potentially all next states). The implementation achieves this exploration through a *breadth-first search*. This is fine for small NFAs, but can lead to real memory problems when the NFAs are bigger and larger strings need to be processed. 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`-function as follows:

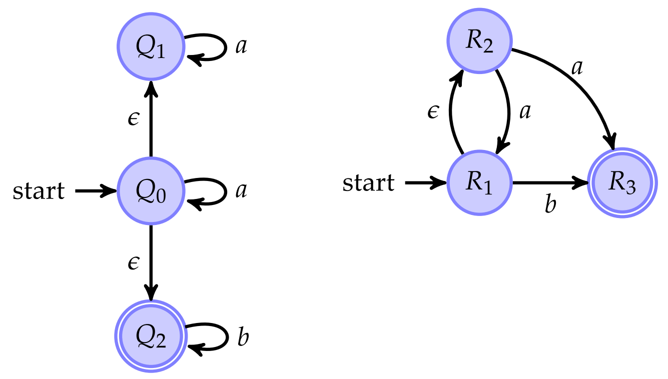
```scala
def search(q: A, s: List[C]) : Boolean = s match {
  case Nil => fins(q)
  case c::cs => next(q, c).exists(search(_, cs))
}

def accepts2(s: List[C]) : Boolean =
  starts.exists(search(_, s))
```

This depth-first way of exploration seems to work quite efficiently in many examples and is much less of a 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 matchings in Java, Ruby and Python. I like to show you this in the next two sections.

Epsilon NFAs

In order to get an idea what calculations are performed by Java & friends, we need a method for transforming a regular expression into an automaton. This automaton should accept exactly those strings that are accepted by the regular expression. The simplest and most well-known method for this is called *Thompson Construction*, after the Turing Award winner Ken Thompson. This method is by recursion over regular expressions and depends on the non-determinism in NFAs described in the previous section. You will see shortly why this construction works well with NFAs, but is not so straightforward with DFAs.

Unfortunately we are still one step away from our intended target though—because this construction uses a version of NFAs that allows "silent transitions". The idea behind silent transitions is that they allow us to go from one state to the next without having to consume a character. We label such silent transition with the letter ϵ and call the automata ϵNFAs. Two typical examples of ϵNFAs are:

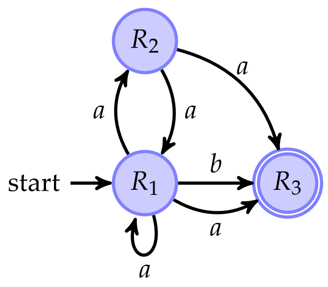


Consider the ϵNFA on the left-hand side: the ϵ-transitions mean you do not have to "consume" any part of the input string, but "silently" change to a different state. In this example, if you are in the starting state Q_0, you can silently move either to Q_1 or Q_2. You can see that once you are in Q_1, respectively Q_2, you cannot "go back" to the other states. So it seems allowing ϵ-transitions is a rather substancial extension to NFAs. On first appearances, ϵ-transitions might even look rather strange, or even silly. To start with, silent transitions make the decision whether a string is accepted by an automaton even harder: with ϵNFAs we have to look whether we can do first some ϵ-transitions and then do a "proper"-transition; and after any "proper"-transition we again have to check whether we can do again some silent transitions. Even worse, if there is a silent transition pointing back to the same state, then we have to be careful our decision procedure for strings does not loop (remember the depth-first search for exploring all states).

The obvious question is: Do we get anything in return for this hassle with silent transitions? Well, we still have to work for it…unfortunately. If we were to follow the many textbooks on the subject, we would now start with defining what ϵNFAs are—that would require extending the transition relation of NFAs. Next, we woudl show that the ϵNFAs are equivalent to NFAs and so on. Once we have done all this on paper, we would need to implement ϵNFAs. Lets try to take a shortcut instead. We are not really interested in ϵNFAs; they are only a convenient tool for translating regular expressions into automata. So we are not going to implementing them explicitly, but translate them immediately into NFAs (in a sense ϵNFAs are just a convenient API for lazy people ;o). How does the translation work? Well we have to find all transitions of the form

$$q \xrightarrow{\epsilon} \ldots \xrightarrow{\epsilon} \xrightarrow{a} \xrightarrow{\epsilon} \ldots \xrightarrow{\epsilon} q'$$

and replace them with $q \xrightarrow{a} q'$. Doing this to the ϵNFA on the right-hand side above gives the NFA



where the single ϵ-transition is replaced by three additional a-transitions. Please do the calculations yourself and verify that I did not forget any transition.

So in what follows, whenever we are given an ϵNFA we will replace it by an equivalent NFA. The Scala code for this translation is given in Figure 3. The main workhorse in this code is a function that calculates a fixpoint of function (Lines 5–10). This function is used for "discovering" which states are reachable by ϵ-transitions. Once no new state is discovered, a fixpoint is reached. This is used for example when calculating the starting states of an equivalent NFA (see Line 36): we start with all starting states of the ϵNFA and then look for all additional states that can be reached by ϵ-transitions. We keep on doing this until no new state can be reached. This is what the ϵ-closure, named in the code `ecl`, calculates. Similarly, an accepting state of the NFA is when we can reach an accepting state of the ϵNFA by ϵ-transitions.

Also look carefully how the transitions of ϵNFAs are implemented. The additional possibility of performing silent transitions is encoded by using `Option[C]` as the type for the "input". The `Some`s stand for "propper" transitions where a character is consumed; `None` stands for ϵ-transitions. The transition functions for the two ϵNFAs from the beginning of this section can be defined as

```scala
val enfa_trans1 : (State, Option[Char]) :=> Set[State] =
  { case (Q0, Some('a')) => Set(Q0)
    case (Q0, None) => Set(Q1, Q2)
    case (Q1, Some('a')) => Set(Q1)
    case (Q2, Some('b')) => Set(Q2) }

val enfa_trans2 : (State, Option[Char]) :=> Set[State] =
  { case (R1, Some('b')) => Set(R3)
    case (R1, None) => Set(R2)
    case (R2, Some('a')) => Set(R1, R3) }
```

I hope you agree now with my earlier statement that the ϵNFAs are just an API for NFAs.

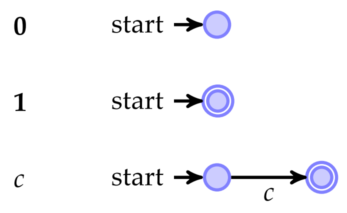
Thompson Construction

Having the translation of ϵNFAs to NFAs in place, we can finally return to the problem of translating regular expressions into equivalent NFAs. Recall that by

```
 1  // epsilon NFAs...immediately translated into NFAs
 2  // (needs nfa.scala)
 3  
 4  // fixpoint construction
 5  import scala.annotation.tailrec
 6  @tailrec
 7  def fixpT[A](f: A => A, x: A): A = {
 8    val fx = f(x)
 9    if (fx == x) x else fixpT(f, fx)
10  }
11  
12  // translates eNFAs directly into NFAs
13  def eNFA[A, C](starts: Set[A],                     // starting states
14                 delta: (A, Option[C]) :=> Set[A],   // epsilon-transitions
15                 fins: A => Boolean) : NFA[A, C] = { // final states
16  
17    // epsilon transitions
18    def enext(q: A) : Set[A] =
19      applyOrElse(delta, (q, None))
20  
21    def enexts(qs: Set[A]) : Set[A] =
22      qs | qs.flatMap(enext(_))     // | is the set-union in Scala
23  
24    // epsilon closure
25    def ecl(qs: Set[A]) : Set[A] =
26      fixpT(enexts, qs)
27  
28    // "normal" transitions
29    def next(q: A, c: C) : Set[A] =
30      applyOrElse(delta, (q, Some(c)))
31  
32    def nexts(qs: Set[A], c: C) : Set[A] =
33      ecl(ecl(qs).flatMap(next(_, c)))
34  
35    // result NFA
36    NFA(ecl(starts),
37        { case (q, c) => nexts(Set(q), c) },
38        q => ecl(Set(q)) exists fins)
39  }
```

Figure 3: A Scala function that translates ϵNFA into NFAs. The transtion function of ϵNFA takes as input an `Option[C]`. `None` stands for an ϵ-transition; `Some(c)` for a "proper" transition consuming a character. The functions in Lines 18–26 calculate all states reachable by one or more ϵ-transition for a given set of states. The NFA is constructed in Lines 36–38.

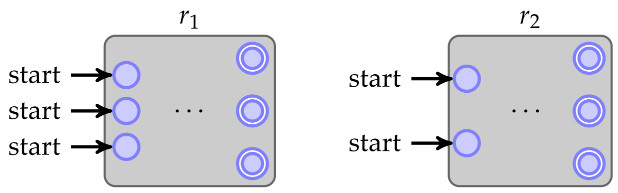
equivalent we mean that the NFAs recognise the same language. Consider the simple regular expressions **0**, **1** and c. They can be translated into equivalent NFAs as follows:



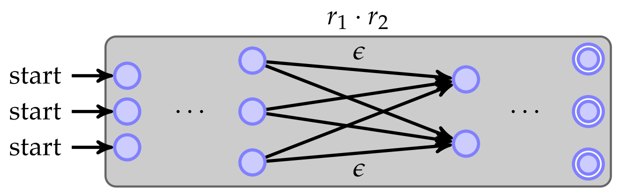
(1)

I let you think whether the NFAs can match exactly those strings the regular expressions can match. To do this translation in code we need a way to construct states programatically...and as an additional constrain Scala needs to recognise that these states are being distinct. For this I implemented in Figure 4 a class `TState` that includes a counter and a companion object that increases this counter whenever a new state is created.[1]

The case for the sequence regular expression $r_1 \cdot r_2$ is a bit more complicated: Say, we are given by recursion two NFAs representing the regular expressions r_1 and r_2 respectively.



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



The idea behind this construction is that the start of any string is first recognised by the first NFA, then we silently change to the second NFA; the ending of the string is recognised by the second NFA...just like matching of a string by the regular expression $r_1 \cdot r_2$. The Scala code for this constrction is given in Figure 5 in Lines 16–23. The starting states of the ϵNFA are the starting states of the first

[1]You might have to read up what *companion objects* do in Scala.

```
 1  // Thompson Construction (Part 1)
 2  // (needs  :load nfa.scala
 3  //         :load enfa.scala)
 4
 5
 6  // states for Thompson construction
 7  case class TState(i: Int) extends State
 8
 9  object TState {
10    var counter = 0
11
12    def apply() : TState = {
13      counter += 1;
14      new TState(counter - 1)
15    }
16  }
17
18
19  // a type abbreviation
20  type NFAt = NFA[TState, Char]
21
22
23  // a NFA that does not accept any string
24  def NFA_ZERO(): NFAt = {
25    val Q = TState()
26    NFA(Set(Q), { case _ => Set() }, Set())
27  }
28
29  // a NFA that accepts the empty string
30  def NFA_ONE() : NFAt = {
31    val Q = TState()
32    NFA(Set(Q), { case _ => Set() }, Set(Q))
33  }
34
35  // a NFA that accepts the string "c"
36  def NFA_CHAR(c: Char) : NFAt = {
37    val Q1 = TState()
38    val Q2 = TState()
39    NFA(Set(Q1), { case (Q1, d) if (c == d) => Set(Q2) }, Set(Q2))
40  }
```

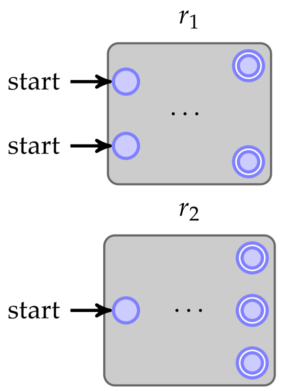
Figure 4: The first part of the Thompson Construction. Lines 7–16 implement a way of how to create new states that are all distinct by virtue of a counter. This counter is increased in the companion object of `TState` whenever a new state is created. The code in Lines 24–40 constructs NFAs for the simple regular expressions **0**, **1** and c. Compare the pictures given in (1).

```
// Thompson Construction (Part 2)

// some more type abbreviations
type NFAtrans = (TState, Char) :=> Set[TState]
type eNFAtrans = (TState, Option[Char]) :=> Set[TState]


// for composing an eNFA transition with a NFA transition
implicit class RichPF(val f: eNFAtrans) extends AnyVal {
  def +++(g: NFAtrans) : eNFAtrans =
  { case (q, None) =>  applyOrElse(f, (q, None))
    case (q, Some(c)) =>
      applyOrElse(f, (q, Some(c))) | applyOrElse(g, (q, c))  }
}

// sequence of two NFAs
def NFA_SEQ(enfa1: NFAt, enfa2: NFAt) : NFAt = {
  val new_delta : eNFAtrans =
    { case (q, None) if enfa1.fins(q) => enfa2.starts }

  eNFA(enfa1.starts, new_delta +++ enfa1.delta +++ enfa2.delta,
       enfa2.fins)
}

// alternative of two NFAs
def NFA_ALT(enfa1: NFAt, enfa2: NFAt) : NFAt = {
  val new_delta : NFAtrans = {
    case (q, c) => applyOrElse(enfa1.delta, (q, c)) |
                   applyOrElse(enfa2.delta, (q, c)) }
  val new_fins = (q: TState) => enfa1.fins(q) || enfa2.fins(q)

  NFA(enfa1.starts | enfa2.starts, new_delta, new_fins)
}

// star of a NFA
def NFA_STAR(enfa: NFAt) : NFAt = {
  val Q = TState()
  val new_delta : eNFAtrans =
    { case (Q, None) => enfa.starts
      case (q, None) if enfa.fins(q) => Set(Q) }

  eNFA(Set(Q), new_delta +++ enfa.delta, Set(Q))
}
```

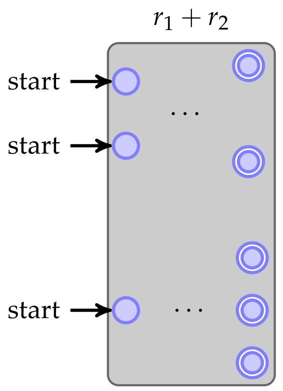
Figure 5: The second part of the Thompson Construction implementing the composition of NFAs according to $\cdot$, $+$ and $\_^*$. The implicit class about rich partial functions implements the infix operation +++ which combines an ϵNFA transition with a NFA transition (both given as partial functions).

NFA (corresponding to r_1); the accepting function is the accepting function of the second NFA (corresponding to r_2). The new transition function is all the "old" transitions plus the ϵ-transitions connecting the accepting states of the first NFA to the starting states of the first NFA (Lines 18 and 19). The ϵNFA is then immedately translated in a NFA.

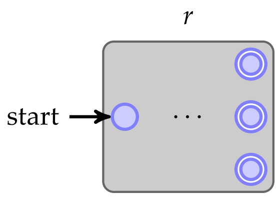
The case for the choice regular expression $r_1 + r_2$ is slightly different: We are given by recursion two NFAs representing r_1 and r_2 respectively.



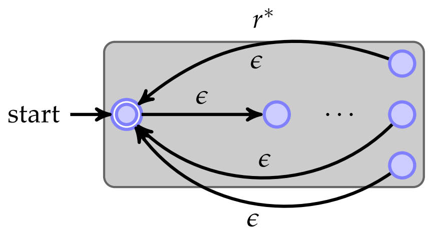
Each NFA has some starting states and some accepting states. We obtain a NFA for the regular expression $r_1 + r_2$ by composing the transition functions (this crucially depends on knowing that the states of each component NFA are distinct); and also combine the starting states and accepting functions:



The code for this construction is in Figure 5 in Lines 25–33. Finally for the $*$-case we have a NFA 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 ϵNFA for r^* (the corresponding code is in Figure 5 in Lines 35–43:

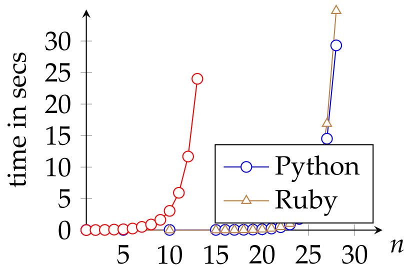


To sum ap, you can see in the sequence and star cases the need of having silent ϵ-transitions. Similarly the alternative case shows the need of the NFA-nondeterminsim. It seems awkward to form the 'alternative' composition of two DFAs, because DFA do not allow several starting and successor states. All these constructions can now be put together in the following recursive function:

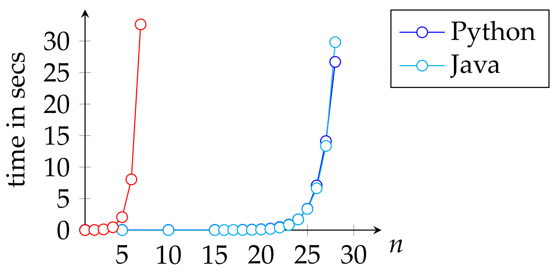
```
def thompson (r: Rexp) : NFAt = r match {
  case ZERO => NFA_ZERO()
  case ONE => NFA_ONE()
  case CHAR(c) => NFA_CHAR(c)
  case ALT(r1, r2) => NFA_ALT(thompson(r1), thompson(r2))
  case SEQ(r1, r2) => NFA_SEQ(thompson(r1), thompson(r2))
  case STAR(r1) => NFA_STAR(thompson(r1))
}
```

It calculates a NFA from a regular expressions. At last we can run a NFA for the our evil regular expression examples.

Graph: `a?{n} a{n}` and strings $\underbrace{\texttt{a}\ldots\texttt{a}}_{n}$



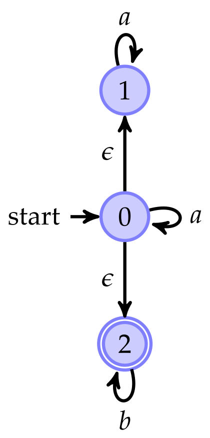
Graph: `(a*)* b` and strings $\underbrace{\texttt{a}\ldots\texttt{a}}_{n}$



Subset Construction

Remember that we did not bother with defining and implementing ϵNFA; we immediately translated them into equivalent NFAs. Equivalent in the sense of accepting the same language (though we only claimed this and did not prove it rigorously). Remember also that NFAs have a non-deterministic transitions, given as a relation. This non-determinism makes it harder to decide when a string is accepted or not; such a decision is rather straightforward with DFAs (remember their transition function).

What is interesting is that for every NFA we can find a DFA that also recognises the same language. This might sound like a bit paradoxical, but I litke to show you this next. There are a number of ways of transforming a NFA into an equivalent DFA, but the most famous is *subset construction*. Consider again the NFA below on the left, consisting of nodes labelled, say, with 0, 1 and 2.



$$\begin{tikzcd}
 & 1 \arrow[loop above, "a"] & \\
\text{start} \arrow[r] & 0 \arrow[u, "\epsilon"] \arrow[d, "\epsilon"] \arrow[loop right, "a"] & \\
 & 2 \arrow[loop below, "b"] &
\end{tikzcd}$$

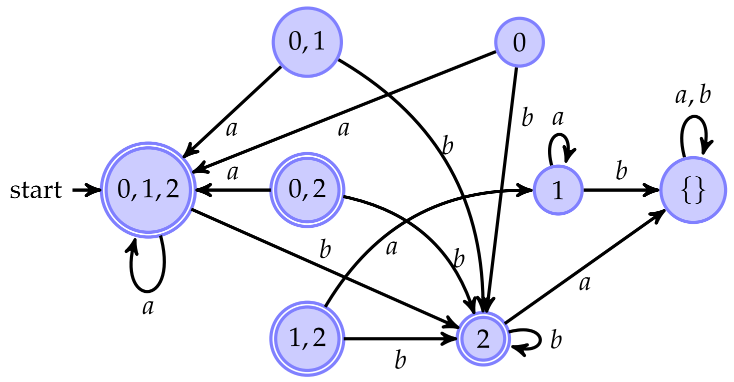
nodes	a	b
$\{\}$	$\{\}$	$\{\}$
$\{0\}$	$\{0,1,2\}$	$\{2\}$
$\{1\}$	$\{1\}$	$\{\}$
$\{2\}\star$	$\{\}$	$\{2\}$
$\{0,1\}$	$\{0,1,2\}$	$\{2\}$
$\{0,2\}\star$	$\{0,1,2\}$	$\{2\}$
$\{1,2\}\star$	$\{1\}$	$\{2\}$
s: $\{0,1,2\}\star$	$\{0,1,2\}$	$\{2\}$

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

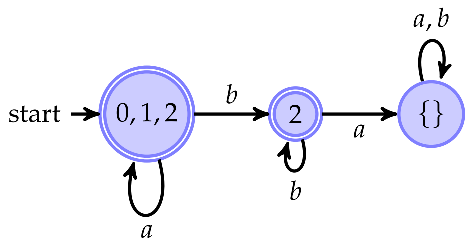
So the transition from the state $\{0\}$ reading an a goes to the state $\{0,1,2\}$. Similarly for the other entries in the rows for $\{0\}$, $\{1\}$ and $\{2\}$. The other rows are calculated by just taking the union of the single node entries. For example for a

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).

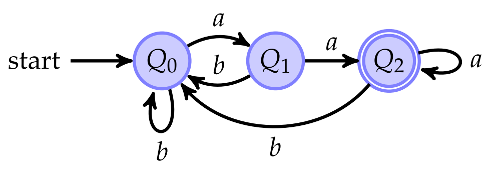
Removing all the dead states in the automaton above, gives a much more legible automaton, namely



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

$$\begin{aligned} Q_0 &= \mathbf{1} + Q_0\, b + Q_1\, b + Q_2\, b && (2)\\ Q_1 &= Q_0\, a && (3)\\ Q_2 &= Q_1\, a + Q_2\, a && (4) \end{aligned}$$

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 second 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" **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

$$\begin{aligned} Q_0 &= \mathbf{1} + Q_0\, b + Q_0\, a\, b + Q_2\, b && (5)\\ Q_2 &= Q_0\, a\, a + Q_2\, a && (6) \end{aligned}$$

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

$$\begin{aligned} Q_0 &= \mathbf{1} + Q_0\, (b + a\, b) + Q_2\, b && (7)\\ Q_2 &= Q_0\, a\, a + Q_2\, a && (8) \end{aligned}$$

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

$$\begin{aligned} Q_0 &= \mathbf{1} + Q_0\,(b + a\,b) + Q_2\,b && (9)\\ Q_2 &= Q_0\,a\,a\,(a^*) && (10)\end{aligned}$$

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

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

Pulling Q_0 out as a single factor gives:

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

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:

$$Q_0 = \mathbf{1}\,(b + a\,b + a\,a\,(a^*)\,b)^* \qquad (13)$$

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

$$Q_0 = (b + a\,b + a\,a\,(a^*)\,b)^* \qquad (14)$$

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

$$\begin{aligned} Q_0 &= (b + a\,b + a\,a\,(a^*)\,b)^* && (15)\\ Q_1 &= (b + a\,b + a\,a\,(a^*)\,b)^*\,a && (16)\\ Q_2 &= (b + a\,b + a\,a\,(a^*)\,b)^*\,a\,a\,(a)^* && (17)\end{aligned}$$

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

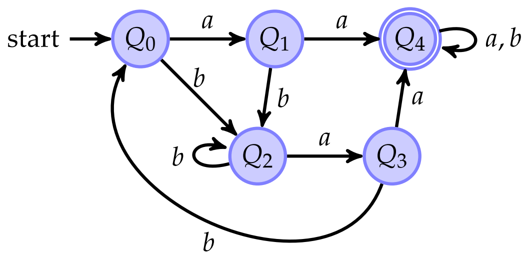
As seen in the subset construction, the translation of a NFA to a DFA can result in a rather "inefficient" DFA. Meaning there are states that are not needed. A DFA can be *minimised* by the following algorithm:

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

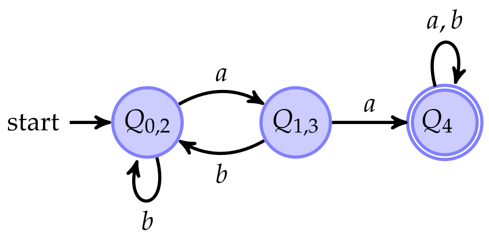
	Q_0	Q_1	Q_2	Q_3
Q_1				
Q_2				
Q_3				
Q_4	⋆	⋆	⋆	⋆

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_0	Q_1	Q_2	Q_3
Q_1	⋆			
Q_2		⋆		
Q_3	⋆		⋆	
Q_4	⋆	⋆	⋆	⋆

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:

$$\begin{tikzcd}
\textbf{Regexps} \arrow[r, "\text{Thompson's construction}"] & \textbf{NFAs} \arrow[r, "\text{subset construction}"] & \textbf{DFAs} \arrow[r, "\text{minimisation}"'] \arrow[ll, bend left, "\text{Brzozowski's method}"] & \textbf{minimal DFAs}
\end{tikzcd}$$

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.