Chapter 9

Regular Expressions

So far, we have proven that DFA, NFAs without ϵ-transitions and NFAs with ϵ-transitions all accept exactly the same class of languages, which we called the class of *regular languages*. We have, for the most part, described these using natural language. For example, "The language of all strings over the alphabet $\{0, 1\}$ that end in 01", or "The language of all strings over $\{0, 1\}$ that have three consecutive zeroes". This was fine for our purposes, and we needed more formal notation, some of these languages were easy enough to express. For example, the language of all strings over $\{0, 1\}$ that end with 01 can more formally be denotes as the set

$$\{w|w \in \{0, 1\}^*, w = x01 \text{ for some } x \in \{0, 1\}^*\}.$$

But sometimes, natural language description is complicated and formal notation is quite cumbersome. Take, for example, the language "The language of all strings that start with zero or more 1's followed by zero or more repetitions of a pattern which consists of two successive 0's followed by one or more 1's". It takes some mental effort to comprehend this definition, and writing it in a formal way is less than trivial. Because of this, we are now going to introduce a very succinct notation for a class of languages. This notation is called *regular expressions*. We will also see that regular expressions describe exactly the *regular languages*, the languages accepted by DFAs and NFAs with or without ϵ-transitions.

9.1 Definition of a Regular Expressions

Regular expressions are expressions formed using three operations - *concatenation*, *Kleene star* and *union*. Each regular expression r defines the associated language $L(r)$, which is the language of all strings that "conform" to the regular expression (as we will see in the following discussion). However, it is important to understand that a regular expression is *not* a language - it is just a notation that describes a language. But sometimes, where no confusion could arise, we will use the term *regular expression* both for the notation itself and for the language it describes.

Regular expressions can be defined using the following recursive definition.

Definition 9.1: Regular Expressions

1. $\emptyset$ is a regular expression and $L(\emptyset) = \emptyset$.
2. ϵ is a regular expression and $L(\epsilon) = \{\epsilon\}$.
3. For any $a \in \Sigma$, $\mathbf{a}$ is a regular expression and $L(\mathbf{a}) = \{a\}$.
4. If r and s are regular expressions, then so are $r + s$, rs and r^*, where

$$L(r + s) = L(r) \cup L(s),$$

$$L(rs) = L(r)L(s),$$

$$L(r^*) = L(r)^*.$$

5. Nothing else is a regular expression

Following the definition, regular expressions are fully parenthesised. E.g. $r + s$ should really be written as $((r) + (s))$. However, this gets cumbersome very quickly, so we will adopt the precedence of operations used to form regular expressions:

1. $*$ has the highest priority.

2. Concatenation has the second highest priority.

3. $+$ has the lowest priority.

Thus, we can write, for example, $((((\mathbf{0})^*)\mathbf{1}) + \mathbf{0})$ as $\mathbf{0}^*\mathbf{1} + \mathbf{0}$. Furthermore, we will also abbreviate rr^* (concatenation of one or more r) as r^+.

Let us now see some examples of regular expressions.

Example 9.1: Example Regular Expressions

1. $\mathbf{0}$ describes the language that contains only the word 0.

2. $\mathbf{0} + \mathbf{1}$ describes the language that contains two words : 0 and 1.

3. $\mathbf{010} + \mathbf{11}$ describes the language that contains two words : 010 and 11.

4. $(\mathbf{0}+\mathbf{1})^*$ describes the language of all strings over $\{0, 1\}$. Recall that $\mathbf{0}+\mathbf{1}$ describes the language $\{0\} \cup \{1\} = \{0, 1\}$, thus $(\mathbf{0} + \mathbf{1})^*$ describes the language $\{0, 1\}^*$.

5. $(\mathbf{0} + \mathbf{1})^*\mathbf{01}$ describes the language of all words over $\{0, 1\}$ that end in 01.

6. $(\mathbf{0}+\mathbf{1})^*\mathbf{000}(\mathbf{0}+\mathbf{1})^*$ describes the language of all words over $\{0, 1\}$ that contain three successive 0's.

Let us see, for example, how we can obtain the expression $(\mathbf{0} + \mathbf{1})^*\mathbf{01}$ from the definition of a regular expression (Definition 9.1). Firstly, $\mathbf{0}$ and $\mathbf{1}$ are regular expressions due to rule 3, therefore so is $\mathbf{0} + \mathbf{1}$ due to rule 4. Note that this would technically be $((\mathbf{0}) + (\mathbf{1}))$, but we agreed to remove parentheses where unnecessary. Next, since $\mathbf{0} + \mathbf{1}$ is a regular expression, so is $(\mathbf{0} + \mathbf{1})^*$ due to rule 4. Note that we have to keep brackets here, as we agreed that $*$ has higher precedence than $+$, so if we write $\mathbf{0} + \mathbf{1}^*$, this is the regular expression that describes a completely different language - the language of strings which are either 0 or some sequence of $1's$ (including ϵ, which is a sequence of zero 1's). Proceeding on, $\mathbf{0}$ and $\mathbf{1}$ are regular expressions due to rule 3, therefore so is $\mathbf{01}$ due to rule 4. Finally, because $(\mathbf{0} + \mathbf{1})^*$ and $\mathbf{01}$ are regular expressions, so is $(\mathbf{0} + \mathbf{1})^*\mathbf{01}$.

Example 9.2: A Bit More Complicated Example Regular Expressions

1. $\mathbf{0}^*\mathbf{1}^*\mathbf{2}^*$ describes the language over $\{0, 1, 2\}$ that contains words with zero or more 0's, followed by zero or more 1's, followed by zero or more 2's.

2. $\mathbf{1}^*(\mathbf{001}^+)^*$ describes the language of strings that start with zero or more 1's followed by zero or more repetitions of a pattern which consists of two successive 0's followed by one or more 1's.

3. $\epsilon + \mathbf{1}(\mathbf{01})^*(\epsilon + \mathbf{0}) + \mathbf{0}(\mathbf{10})^*(\epsilon + \mathbf{1})$ describes the language of all strings over $\{0, 1\}$ that do not contain consecutive 0's or consecutive 1's.

4. $(\mathbf{1} + \epsilon)(\mathbf{01})^*(\mathbf{0} + \epsilon)$ describes the same language as above.

5. $(\mathbf{0} + \epsilon)(\mathbf{1} + \mathbf{10})^*$ describes the language of strings that do not contain two successive 0's.

9.2 Equivalence Between Regular Expressions and Finite Automata

Regular expressions are used in many systems. For example, almost all of the programming languages support regular-expression-based searching for patterns in text, as they make it much easier and much more concise to specify exactly what pattern you are looking for. But the question is how to write a piece of code that will recognise whether some pattern appears in some text, since regular expressions look pretty abstract and it is not at all obvious how to recognise all the strings of a language of some regular expression. However, it turns out that the class of languages that can be described using regular expressions is exactly the same as the class of languages accepted by finite automata (DFAs, NFAs and NFAs with ϵ-transitions). This is exactly why the languages accepted by DFAs (and NFAs and NFAs with ϵ-transitions) are called *regular languages*. This is a very important result and it is the first example of a relationship between a language and a model of a computer. In practical terms, for our example, this means that we can design an automaton that will recognise exactly the language of our regular expression. And the automata are very easy to implement in any decent programming language.

Theorem 9.1: Equivalence Between Regular Expressions and Finite Automata

The class of languages recognised by DFAs, NFAs and NFAs with ϵ-transitions and the class of languages that can be described using regular expressions are the same.

As usual, there are two parts of this theorem. The first is that if a language can be described with a regular expression, then there is a finite automata that accepts it. The second is the other way around - if the language is accepted by some form of finite automaton, then it can also be described with a regular expression. The first part is notably easier, therefore we will start with that.

9.2.1 From a Regular Expression to a Finite Automaton

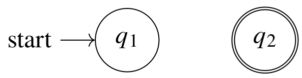
Lemma 9.2: Regular Expression to Finite Automata

Let r be a regular expression and let $L(r)$ be the language of that regular expression. Then there is a NFA with ϵ-transitions M' that accepts the language $L(r)$.

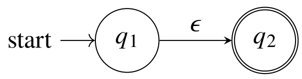
Proof. Of course, the statement of our lemma could have as well said "there exists a DFA M' that accepts the language $L(r)$", but we specified that it is an NFA with ϵ-transitions to make it more clear what kind of an automata we will build.

Since r is a regular expression, we know that r can only be obtained using the rules specified in Definition 9.1. If we show how to define an NFA with ϵ-transition for each of these rules (that is, to define an NFA with ϵ-transitions that will accept the language of a regular expression obtained using each of these rules), then we will be done. This is also done in a recursive way. In particular, we first show how to build NFAs with ϵ-transitions to accept the languages of each of the basic constructs for regular expressions (that is, the regular expressions that are just $\emptyset$, ϵ and **a**). Afterwards, we will show that if we know how to build NFAs with ϵ-transitions for regular expressions r and s, we also know how to build an NFA with ϵ-transitions for regular expressions $r + s$, rs and r^*. To make our life easier, we are going to construct NFAs with ϵ-transitions that have exactly one accepting state, and where the start state is distinct from the accepting state. This will make it easier to combine different NFAs with ϵ-transitions.

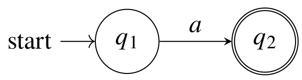
Base Cases. We know that $\emptyset$ is a regular expression that describes the language that is an empty set of strings $\emptyset$. One NFA that accepts the empty set of strings is



Next, ϵ is a regular expression that described the language ϵ. One NFA with ϵ-transitions that accepts this language is

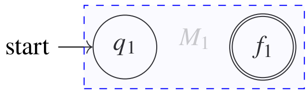


Finally, **a** is a regular expression that describes the language $\{a\}$. One NFA with ϵ-transitions that accepts this language is

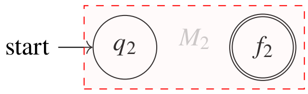


And with this, we have NFAs with ϵ-transitions for each of the basic regular expressions constructs.

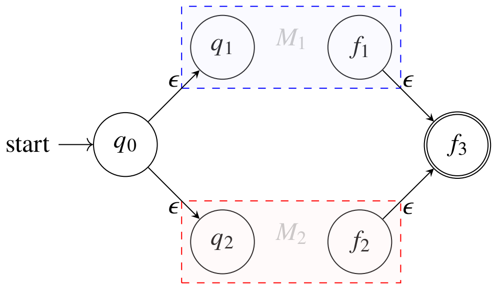
$*$, $+$ and Kleene Star operations. Let us now assume that we have an NFA with ϵ-transitions (with a single accepting state) M_1 that accepts the language described by the regular expression r and an NFA with ϵ-transitions M_2 (with a single accepting state) that accepts the regular expression s. Let us represent M_1 by



where q_1 is a start state of M_1 and f_2 is a single accpeting state of M_1. There are, obviously, states and transitions in between, but they are not relevant to how we combine two automatons, so we abstract them away. Similarly, let us represent M_2 by



Let us first construct an NFA with ϵ-transitions M_3 that accepts the language described by the regular expression $r + s$. Recall that the language described by $r + s$ is the union of languages described by r and by s. Therefore, M_3 is fairly easy to construct. Let M_3 be given by



How does this automaton operate on some input string w? Firstly, from its start state, it spontaneously transits to the starting states of the automata M_1 and M_2. Then it essentially runs these two automata in parallel. If either of them accepts the string w, M_3 will end up in the accepting state of that automaton (either f_1 or f_2), together with some other states possibly, and from there it will make a spontaneous transition to the accepting state f_3 of M_3. Therefore, any string w that is accepted by either M_1 or M_2 will also be accepted by M_3. Conversely, if a string w is accepted by M_3, then one of the states M_3 reaches after reading the whole string w is f_3. Since f_3 can be reached only from f_1 or f_2 using ϵ-transitions, M_3 must have ended up in the states f_1 or f_2 after reading the whole string w. Since M_3 transits from its initial state q_0 to both q_1 and q_2 without reading any input symbol, it follows that either M_1 or M_2 ends up in its accepting state if it starts reading w from its start state. In other words, w is accepted either by M_1 or M_2.

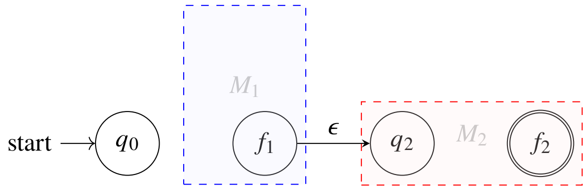
Remark 9.1: Formal Description of M_3

M_3 can formally be described in the following way. Let $M_1 = (Q_1, \Sigma, \delta_1, q_1, \{f_1\})$ be an NFA with ϵ-transitions that accepts the language described by the regular expression r and let $M_2 = (Q_2, \Sigma, \delta_2, q_2, \{f_2\})$ be an NFA with ϵ-transitions that accepts the language described by the regular expression r. Then the NFA with ϵ-transitions $M_3 = (Q_1 \cup Q_2 \cup \{q_0, f_3\}, \Sigma_1 \cup \Sigma_2, \delta_3, q_0, \{f_3\})$, where

$$\delta_3(q, a) = \begin{cases} \delta_1(q, a) \text{ if } q \in Q_1 \setminus \{f_1\} \text{ and } a \in \Sigma_1 \cup \{\epsilon\} \\ \delta_2(q, a) \text{ if } q \in Q_2 \setminus \{f_2\} \text{ and } a \in \Sigma_2 \cup \{\epsilon\} \\ \{f_3\} \text{ if } q \in \{f_1, f_2\} \text{ and } a = \epsilon \\ \{q_1, q_2\} \text{ if } q = q_0 \text{ and } a = \epsilon \end{cases}$$

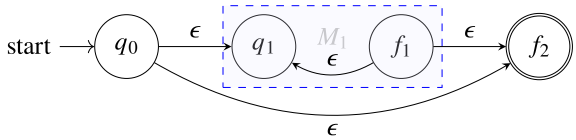
accepts the language described by the regular expression $r + s$,

Let us now design an automaton M_4 that accepts the language described by the regular expression rs. M_4 is given as



Recall that the language described by the regular expression rs is the language $L(r)L(s)$. That is, the language of concatenations of strings, where the first string belongs to the language described by r and the second string belongs to the language described by s. It is easy to see why M_4 accepts any string that is such a concatenation. On any input string w, M_4 will run M_1 on w, but also every time some prefix of w is accepted (that is, the accepting state f_1 of M_1 is reached), it will also try to run M_2 on the rest of the string. If running M_2 on the rest of the string results in reaching the accepting state of M_2, which is f_2, then the whole string is accepted, as f_2 is also an accepting state of M_3. Therefore, if $w = xy$, where x is some string accepted by M_1 and y is some string accepted by M_2, w will also be accepted by M_4. Conversely, it is also easy to see why any string that is accepted by M_4 has to be of the form xy, where x is accepted by M_1 and y is accepted by M_2. Which means that the language accepted by M_3 is exactly the set of all strings which are concatenations of a string accepted by M_1 and a string accepted by M_2. Which means that the language M_4 accepts is exactly the language described by the regular expression rs.

Finally, let us design an automaton M_5 that accepts the language described by the regular expression r^*. M_5 is given by



Here we introduced a new starting state q_0 and a new accepting state f_2 and we introduce an ϵ-transition from q_0 to the starting state of M_1, and also to the new accepting state f_2. Within M_1, we introduce an ϵ-transition from its accepting state to its starting state, allowing the automaton to loop. We also introduce an ϵ-transition from the accepting state of M_1 to the new accepting state f_2. The accepting state of M_1 is not an accepting state in our new automaton. It is easy to see why this automaton accepts the language $L(r)^*$ (the language of the regular expression r^*). Any string w from $L(r)^*$ can be written as a concatenation $w_1 w_2 \cdots w_n$ of strings w_i from $L(r)$, each of which is accepted by the automaton M_1. M_5 would simulate M_1 on w_1, reaching the state f_1 and then making an ϵ-transition back to the starting state q_1 of M_1. It would then simulate M_1 on w_2, eventually reaching f_1 and making another spontaneous transition to q_1. Eventually, simulating M_1 on w_n would result in reaching the state f_1, from which the accepting state f_2 would be reached via the ϵ-transition from f_1. So, the string $w = w_1 w_2 \ldots w_n$ would be accepted. The epsilon transition from q_0 to f_2 allows M_5 to also accept the empty string ϵ, which is always in $L(r)^*$ for any regular expression r. Therefore, every string from $L(r)^*$ will be accepted by M_3. It is equally easy to see that any string accepted by M_5 is of the form $w_1 w_2 \ldots w_n$, for some strings w_i from $L(r)$. Therefore, the language accepted by M_5 is exactly $L(r)^*$.

Remark 9.2: Formal Description of Our Automata

Similarly to how it was done in Remark 9.1, we can also easily formally describe the automata for rs and r^*. This, however, will be left as an exercise to the reader. We could also then, using this formal description, formally prove that the described automata really accept the languages described by regular expressions $r + s, rs$ and r^*, assuming that M_1 and M_2 accept the languages described by regular expressions r and s respectively. This is fairly easy, but somewhat tedious, so we will skip this part too.

With the last construction, we have shown that, if we know how to construct automata to accept the languages described by regular expressions r and s, then we also know how to construct automata to accept the languages described by regular expressions $r + s, rs$ and r^*. Together with showing how to construct automata to accept the languages described by the basic regular expressions $\emptyset$, ϵ and $\mathbf{a}$, we now have a method to construct an automata to accept the language described by any regular expression. This finishes the proof of our Lemma. □

What we have proven is that any language described by a regular expression is also accepted by some finite automaton. In the process of proving this result, we have given a method how to design such an automaton. We will now see how this works on an example.

Video 9.1: Regular Expression to Finite Automaton

Using the construction from the proof of Lemma 9.2, design a finite automaton that accepts the language designed by the regular expression $\mathbf{01}^* + \mathbf{1}$.