LECTURE NOTES

CPSC 351 — Winter 2025

Theoretical Foundations of Computer Science II

Philipp Woelfel

Chapter 3: Regular Languages and Expressions
Chapter 4: Finite-State Automata

Contents

3 Regular Languages and Expressions

Computer scientists are interested in the complexity and solvability of computational problems. A computational problem is defined by an input-output relation, which assigns the input to an algorithm to the correct answers. Often, the correct output to an input is exactly one of “yes” or “no”. In this case the computational problem is a *decision problem.* The study of complexity and solvability often focuses on decision problems: Such problems have clean and elegant specifications, and many results generalize to problems with more than two answers.

A *formal language* describes the input strings for which the correct output of a decision problem is “yes”. Thus, formal languages are used to define decision problems.

In this section we will first introduce basic concepts of formal language theory, such as alphabets, strings, languages, and their operations. We will then discuss *regular* languages, which have many applications in Computer Science, because they can be defined by regular expressions and finite automata.

3.1 Strings

Sequences of symbols are called *strings.* We are interested in strings whose symbols are taken from a fixed set, called the *alphabet.* We will use the symbol λ to describe the empty string, so that symbol must not appear in the alphabet.

Definition 3.1. *An* alphabet Σ *is a set that does not contain* λ*. The elements of* Σ *are called* symbols*.*

For example, $\Sigma_B = \{0, 1\}$ is an alphabet, and so is the set $\Sigma_E = \{a, b, \dots, z, A, B, \dots, Z\}$ of all English letters. In this course we will mostly consider finite alphabets, even though in general, alphabets can be infinite (or even uncountable).

Definition 3.2. *A* string *(or* word*) over an alphabet* Σ*, is a sequence of symbols from* Σ*. The* length *of a string* x *is the number of its symbols, and is denoted* $|x|$*. The* empty *string is denoted* λ*.*

For example, “*hello*” is a string over the alphabet Σ_E. In general, a string of length n over some alphabet Σ has the form $x_1x_2 \dots x_n$, where $x_i \in \Sigma$ for all $i \in \{1, \dots, n\}$.

3.1.1 Substrings

A *substring* of a string x is a contiguous sequence of symbols of x. In other words, we can obtain a substring by removing zero or more symbols from the beginning and from the end of x. (A generalization is a *subsequence*, which we obtain by removing zero or more symbols from anywhere.) Note that the empty string, λ, and x are also substrings of x. For example, substrings of "banana" are "nan", λ, and "banana". We say that y is a *proper* substring of x, if y is a substring of x and y is shorter than x (i.e., $|y| < |x|$).

If we only remove (zero or more) symbols from the end of string x, we obtain a *prefix* of x. Similarly, if we only remove symbols from the beginning, we obtain a *postfix*. For example, "ban" is a prefix of "banana", and "na" is a postfix. Again, a string is a prefix and postfix of itself, and so is λ. And a prefix (postfix) of string x is a *proper* prefix (postfix) of x, if it is shorter than x.

The opposite notion of substring is that of a *superstring*. Precisely, if y is a substring of x, then x is a superstring of y. For example, "anana" is a superstring of "nan". Note that x is substring and superstring of x, and any string is a superstring of λ.

3.1.2 String Concatenation

If $x = x_1 \dots x_n$ and $y = y_1 \dots y_m$ are two strings, then their concatenation is

$$x \circ y = x_1 \dots x_n y_1 \dots y_m.$$

For example, if $x = 01011$ and $y = 101$, then

$$x \circ y = \underbrace{01011}_{x}\underbrace{101}_{y}.$$

If it is clear from the context that x and y are strings, we may also omit the concatenation operator, $\circ$, and simply write xy.

The result of concatenating a string k times with itself is denoted

$$x^k = \underbrace{x \circ x \circ \cdots \circ x}_{k \text{ times}}.$$

For example, if $x = ab$, then $x^3 = ababab$.

Observe that the length of the concatenation of two strings x and y is $|x \circ y| = |x| + |y|$, and $|x^k| = k \cdot |x|$.

Note that concatenating the empty string, λ with any string x does not change the string. In particular, $\lambda x = x = x\lambda$ and $\lambda^k = \lambda$. Thus, λ is the *neutral element* in the set of strings w.r.t. concatenation. Also, $x^1 = x$, and concatenating any string x 0 times leaves us with the empty string: $x^0 = \lambda$.

3.2 Formal Languages

A *formal language* (or short, language) is simply a set of strings. As we mentioned earlier, a language can be used to describe the inputs to a decision problem, for which the answer is “yes”. Another application is that of searching for strings, e.g., in a text document. A language can be used to specify the strings we are searching for.

Definition 3.3. *A* (formal) language *over an alphabet Σ is a set of strings from Σ.*

For example, let $\Sigma_E = \{a, \ldots, z, A, \ldots, Z\}$ be the alphabet of all English capital and lower case letters. Then the set $\{Alice, Bob, Chris, Dana\}$ is a language over Σ_E. The set $\{\lambda, 0, 00, 000, 0000, 00000, \ldots\}$ is the (infinite) language of all strings overs $\{0, 1\}$ that contain only 0s. The *empty language* is denoted $\emptyset$ or $\{\}$.

Concatenation of Languages. We can concatenate two languages L_1 and L_2. This means that we just create all strings that can be obtained by concatenating a string from L_1 with a string from L_2. More precisely, the concatenation of L_1 and L_2 is

$$L_1 \circ L_2 = \{xy \mid x \in L_1 \wedge y \in L_2\}.$$

For example, suppose $\Sigma = \{0, 1\}$, $L_0 = \{\lambda, 0, 00, 000, 0000, \ldots\}$ is the language of all all-0 strings, and $L_1 = \{\lambda, 1, 11, 111, 1111, \ldots\}$ is the language of all all-1 strings. Then $L_0 \circ L_1$ is the language containing all strings of 0s followed by 1s. For example, λ, 01, 0011111, 000001, 000, 11, are all words in $L_0 \circ L_1$, but 010 is not.

If we concatenate a language L with itself, we obtain $L^2 = L \circ L$. If we concatenate a language k times with itself, we obtain

$$L^k = \underbrace{L \circ L \circ \cdots \circ L}_{k \text{ times}}.$$

According to this definition, $L^1 = L$.

An interesting case is $L^0 = \{\lambda\}$. L^0 is defined this way, so that it is the *neutral element* w.r.t. concatenation. This means that if we concatenate L^0 with any other language, we get that other language:

$$L^0 \circ L' = \{\lambda\} \circ L' = L' = L' \circ L^0.$$

Even if $L = \emptyset$, we have $L^0 = \{\lambda\}$.

Here are some examples: Suppose $L_1 = \{01, 10\}$ and $L_2 = \{00, 11\}$. Then

$$L_1 \circ L_2 = \{0100, 0111, 1000, 1011\}.$$

For $L_3 = L_2 \cup \{\lambda\} = \{00, 11, \lambda\}$, we get

$$L_1 \circ L_3 = \{0100, 0111, 1000, 1011, 01, 10\}.$$

Now, let $L_4 = \{\lambda, 0, 00, 000, \ldots\} = \left\{0^k \;\middle|\; k \in \mathbb{N}_0\right\}$, and $L_5 = \{\lambda, 1, 11, 111, \ldots\} = \left\{1^k \;\middle|\; k \in \mathbb{N}_0\right\}$. Thus, L_4 is the language of all all-0 strings (including the empty string), and L_5 is the language of all all-1 strings. Then

$$L_4 \circ L_5 = \left\{0^k 1^\ell \;\middle|\; k \in \mathbb{N}_0, \ell \in \mathbb{N}_0\right\}$$

is the language of all strings of zero or more 0s followed by zero or more 1s. This language contains, for example, 000011, 0000, 1111 and λ. But it does not contain 010 or 11110000.

Definition 3.4. *The* positive closure *of a language L, denoted L^+, contains all words that can be obtained by concatenating one or more words from L:*

$$L^+ = \bigcup_{k=1}^{\infty} L^k = L^1 \cup L^2 \cup L^3 \cup \ldots.$$

The Kleene closure *(also called* star closure*), contains all words that can be obtained by concatenating zero or more words from L:*

$$L^* = \bigcup_{k=0}^{\infty} L^k = \{\lambda\} \cup L^1 \cup L^2 \cup L^3 \cup \cdots = \{\lambda\} \cup L^+$$

Thus, the Kleene closure is the same as the positive closure, except that is also contains λ:

$$L^* = L^+ \cup \{\lambda\}.$$

For example, if $L = \{1\}$, then L^+ is the language of all all-1 strings of length one or larger, whereas L^* contains all all-1 strings of length zero or larger.

3.3 Regular Languages

An important class of formal languages consists of the so-called *regular languages*. We will show that regular languages can be defined through regular expressions, which are a commonly used for searching documents. They also correspond to the decision problems that can be solved with finite automata, which are a model for hardware.

Definition 3.5. *A language R over an alphabet Σ is* regular*, if it is one of the following:*

- *$R = \emptyset$;*
- *$R = \{\lambda\}$ or $R = \{a\}$ for $a \in \Sigma$; or*
- *R is the concatenation or union of two regular languages L_1 and L_2 (i.e., $L = L_1 \circ L_2$ or $L = L_1 \cup L_2$), or*
- *R is the Kleene closure or positive closure of a regular language L (i.e., $R = L^*$ or $R = L^+$).*

Here are some examples of regular languages over the alphabet $\Sigma = \{0, 1\}$.

- $R_0 = \{\lambda\}$.
- $R_1 = \{0\}$.
- $R_2 = \{1\}$.
- $R_3 = \{01\} = R_1 \circ R_2$.
- $R_4 = \{10\} = R_2 \circ R_1$.
- $R_5 = \{\lambda, 0, 1, 01, 10\} = R_0 \cup R_1 \cup R_2 \cup R_3 \cup R_4$.
- $R_6 = \{0, 1, 01, 10, 11, 00, 01110, 1111000\}$. (Union and multiple concatenations of R_1 and R_2.)
- $R_7 = \{0\} \circ \{0, 1\}^*$ (Contains all bit-strings starting with a 0. Does not contain the empty string.)
- $R_8 = \{0, 1\}^* \circ \{010\} \circ \{0, 1\}^*$ (Contains all bit-strings that have 010 as a sub-string.)
- $R_9 = \{0, 1\}^*\{\lambda, 1\}$ (Contains all bit-strings.)
- $\{\lambda, 1\}^+$ (Contains all all-1 strings and λ.)

We observe that any *finite* language R is regular: For each word $x = x_1 \dots x_k \in R$, language $\{x\}$ is regular, because it is the concatenation $\{x_1\} \circ \{x_2\} \circ \cdots \circ \{x_k\}$. Since R is the union of all $\{x\}$ for words $x \in R$, R is regular.

Observation 3.6. *If R is a finite language, then R is regular.*

Example 3.7. Let $\Sigma = \{a, b, !\}$. The *sheep language*, S, contains all words starting with a "b", followed by one or more a's, and ending with a single "!". Is the sheep language regular?

The answer is also yes, because we can obtain it from finite languages (which are regular by Observation 3.6) through the "allowed" operations, concatenation, union, positive closure, and Kleene closure:

$$S = \{b\} \circ \{a\}^+\{!\}.$$ ◀

Example 3.8. Let $\Sigma = \{a, b\}$. Let L be the language that contains "*abab*" as a substring. Is L regular?

The answer is yes, because we can obtain L from finite languages through the allowed operations:

$$L = \{a, b\}^* \circ \{abab\} \circ \{a, b\}^*. \qquad \blacktriangleleft$$

3.4 Regular Expressions

See also Section 13.4.1 in the textbook.

Definition 3.9. *The* regular expressions *over an alphabet* Σ *are defined as follows:*

- $\emptyset$ *is a regular expression;*
- a *is a regular expression for each* $a \in \Sigma \cup \{\lambda\}$.
- *If* r *and* s *are regular expressions, then so are*
  - rs,
  - $r + s$,
  - r^*, *and*
  - r^+.

Each regular expression x corresponds to a language $L(x)$, which is defined as follows.

- $L(\emptyset) = \emptyset = \{\}$ (empty language).
- $L(a) = \{a\}$ for each $a \in \Sigma \cup \{\lambda\}$.

And for regular expressions r and s:

- $L(rs) = L(r) \circ L(s)$ (concatenation)
- $L(r + s) = L(r) \cup L(s)$ (union).
- $L(r^*) = L(r)^*$ (Kleene closure).
- $L(r^+) = L(r)^+$ (positive closure).

Example 3.10. For each regular expression r among the following, describe how any string $x \in L(r)$ looks like:

- 01^*: a 0 followed by zero or more 1s.

- $(01)^*$: zero or more copies of 01.
- $0 + 01$: either 0 or 01.
- $(0 + 01)^+$: any non-empty string, where each 1 follows a 0.
- $(0+1)^*\big((00) + (11)\big)(0+1)^*$: any string that contains a repeated symbol (the same symbol in two consecutive positions).
- 0^*10^*: any string containing exactly one 1. ◀

It is not always obvious to see that the description matches the regular expression, and we may need to prove it.

Example 3.11. What are the strings in $L = L((0^*1)^*)$? We claim that this language contains all strings that do not end with a 0.

Note that

$$L = L((0^*1)^*) = L(0^*1)^* = \bigcup_{k\in\mathbb{N}_0} L(0^*1)^k. \tag{3.1}$$

Thus, for each string $x \in L$, there is a non-negative integer k, such that x is the concatenation of k substrings $x_1, \ldots, x_k \in L(0^*1)$. Each such substring consists of zero or more 0s followed by a 1.

If $k = 0$, then x is the empty string, and so it does not end with a 0. If $k \geq 1$, then clearly x ends with the same symbol as x_k, which is a 1. This proves that any string in L does not end with a 0.

It remains to prove that if a string x does not end with a 0, then it is in L. If x is the empty string, then it is in $L(01^*)^0$. Otherwise, x ends with a 1. Suppose x contains $k \geq 1$ 1s. Then we can split x into k substrings $x_1, x_2 \ldots x_k$, such that the i-th substring starts after the $(i-1)$-th 1 and ends with the i-th 1. Thus, $x \in L(0^*1)^k$, and thus it is in L by eq. (3.1). ◀

Example 3.12. Give a regular expression r for all non-empty strings over $\{0, 1\}$ with alternating symbols.

The starting symbol (0 or 1) and the length uniquely determine the string. For example, if the string starts with 0 and has length 5, then it must be 01010. We can generate all strings of even length starting with 0 using the following regular expression:

$$(01)^+.$$

In order to also allow odd length strings, we have several options. One is to simply take the union of strings of even length and of odd length (starting with 0):

$$\underbrace{(01)^+}_{\text{even length}} + \underbrace{0(10)^*}_{\text{odd length}}.$$

To get all alternating strings, the ones starting with 0 and the ones starting with 1, we use

$$\underbrace{(01)^+ + 0(10)^*}_{\text{starting with 0}} + \underbrace{(10)^+ + 1(01)^*}_{\text{starting with 1}}.$$

We observe that $0(10)^* + (10)^+$ can be simplified to $(0+10)(10)^*$. Similarly, $(01)^+ + 1(01)^*$ can be simplified to $(1+01)(01)^*$. Thus, an equivalent regular expression for all alternating and non-empty strings over $\{0,1\}$ is

$$r = (0+10)(10)^* + (1+01)(01)^*. \tag{3.2}$$

◀

Example 3.13. For each of the following description of strings over $\{0,1,2\}$, give a regular expression r, such that $L(r)$ is exactly the language of described strings.

- All strings that contain three or more 1s: $(0+1+2)^*1(0+1+2)^*1(0+1+2)^*1(0+1+2)^*$.
- All strings that contain exactly three 1s: $(0+2)^*1(0+2)^*1(0+2)^*1(0+2)^*$.
- All strings that start with a 0 and contain at most two 0s: $0(1+2)^*(0+\lambda)(1+2)^*$.
- All strings, where every 2 is immediately preceded by a 1: $(0+1+12)^*$.
- All strings with an even number of 1s: $(0+2)^* \underbrace{\left(1(0+2)^*1(0+2)^*\right)^*}_{\text{even number of 1s, starting with 1}}$. ◀

Example 3.14. Let L denote the language of all strings over $\{0,1,2\}$ that contain no repeated symbols (i.e., no symbol appears in two consecutive positions). Give a regular expression for L.

The idea is the following: We split any string x in L into multiple parts, separated by 2s. I.e., we write x as

$$x_1 2 x_2 2 x_3 2 \ldots 2 x_k,$$

such that $x_1, \ldots, x_k$ do not contain any 2s. Then each string x_i must alternate 0s and 1s. This is similar to the strings considered in Example 3.12. But note that x_1 and x_k may be equal to λ, but none of $x_2, \ldots, x_{k-1}$ can be the empty string, as otherwise we would have two consecutive 2s.

Hence, each x_i for $i \in \{2, \ldots, k-1\}$, can be generated by the regular expression from Example 3.12 in eq. (3.2).

$$r = (0+10)(10)^* + (1+01)(01)^*.$$

To generate x_1 or x_k we need to also allow λ, so we can use

$$\lambda + r,$$

where r is the regular expression defined above. Overall we get:

$$\underbrace{(\lambda + r)\,2}_{x_1 2}\;\underbrace{(r\,2)^*}_{x_2 2 \ldots x_{k-1} 2}\;\underbrace{(\lambda + r)}_{x_k}.$$

The only problem with that is, that it requires at least one 2. I.e., it assumes that the string is of the form $x_1 2 x_2 2 \ldots 2 x_k$ for $k \geq 2$, and not for $k = 1$. But this is easy to fix: Using the "+" operator, we combine the above expression with $\lambda + r$, in order to also allow all alternating 0/1-strings. This way, we obtain

$$\underbrace{\lambda + r}_{\text{no 2}} + \underbrace{(\lambda + r)\,2\,(r\,2)^*(\lambda + r)}_{\text{at least one 2}}. \qquad ◀$$

3.5 Exercises

3.1 Let $A = \{1, 00\}$ and $B = \{11, 10, \lambda\}$.

language operations

(a) Give an alphabet Σ, such that A and B are languages over Σ.

(b) Describe each of the following languages: AB, BA, B^2, $(AB)^*$, $(A \cup B)^*$.

3.2 For each of the following statements, decide if it is true for every language A, or if there is a language A for which it is not true. Justify your answer.

language operations

(a) $A^* = (A^*)^*$

(b) $A^+ = (A^+)^+$

(c) $A^+ = (A^+)^*$

3.3 For each of the following statements, decide if it is true or false, and justify your answer.

language operations

(a) $\{0,1\}^* = \big(\{0\}\{1\}\big)^*$

(b) $\{0,1\}^* = \big(\{0\}^*\{1\}^*\big)^*$

(c) $\{0,1\}^* = \big(\{0\}^*\{1\}^*\{0\}^*\big)^*$

3.4 For each of the language operations concatenation, union, and intersection, give the neutral element with respect to that operations, if it exists.

language operations

3.5 Give all languages L, such that $L^* = L$. Do the same for the plus closure.

language operations

3.6 Let A and B be two languages over an alphabet Σ. Prove each of the following statements, or give a counter example.

language operations

(a) $|AB| = |A| \cdot |B|$

(b) $|AB| \leq |A| \cdot |B|$

(c) If $A \subseteq B$, then $A^* \subseteq B^*$.

(d) If $A \subsetneq B$, then $A^* \subsetneq B^*$.

3.7 For each of the following languages, decide if it contains the string $abba$.

language operations

(a) $\{a\}^*\{b\}^*$

(b) $\{a\}^* \cup \{b\}^*$

(c) $(\{a\} \cup \{b\})^*$

(d) $\{ab\}^* \cup \{ba\}^*$

(e) $\{a\}^*\{b\}^*\{a\}^*$

(f) $\{b\}^*\{a\}^*\{b\}^*$

(g) $(\{b\}^*\{a\}^*)^+$

3.8 Let Σ be a finite alphabet, and $a \in \Sigma \cup \{\lambda\}$. Prove that $\overline{\{a\}}$ is a regular language over Σ.

regular languages

3.9 Give regular expressions for each of the following languages over the alphabet $\Sigma = \{a, b, c\}$, which contains the given strings.

regular expressions

1. a, b, ab
2. all strings except a, b, ab
3. all strings of even length
4. all strings that begin with a
5. all strings that do not begin with a
6. all strings that begin with a and end with c
7. all strings that do not begin with a or do not end with c
8. all strings that do not begin with a and do not end with c
9. all strings that contain ab as a substring
10. all strings that do not contain ab as a substring
11. all strings, where no c precedes a or b, and no b precedes a
12. all strings that contain exactly 3 a's
13. all strings, where the total number of a's is a multiple of 3

3.10 For each of the following regular expressions, describe the language it defines.

regular expressions

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

(b) $(a+b)^*bab(a+b)^*$

(c) $(a+b)\big((a+b)(a+b)\big)^*$

(d) $(aa)^*(ab + bb + b)^*$.

3.11 For each of the following statements, decide if it is true or false. Justify your answer.

regular expressions

(a) $bbaab$ is in the language defined by the regular expression $(a+b)^*a(a+b)^*$.

(b) $aabb$ is in the language defined by the regular expression $(ab + ba)^*$.

3.12 Two regular expressions r_1 and r_2 are equivalent, if they define the same language, i.e., if $L(r_1) = L(r_2)$. For each of the following pairs of regular expressions, decide if they are equivalent, and justify your answer.

regular expressions

(a) $\lambda + ab(ab)^*$ and $(ab)^*$

(b) $(ab + bb)^*$ and $(a + b)b^*$

(c) $a(ba)^*$ and $(ab)^*a$

3.6 Selected Solutions

Exercise 3.6

(a) The claim is not true. For example, let $\Sigma = \{0, 1\}$, $A = \{\lambda, 0, 1\}$ and $B = \{\lambda, 0, 1\}$. Then $|A| = |B| = 3$. On the other hand, $AB = \{\lambda, 0, 1, 00, 01, 10, 11\}$, and so $|AB| = 7$. But $|A| \cdot |B| = 3 \cdot 3 = 9 \neq |AB|$.

(b) The claim is true. Note that $|A| \cdot |B|$ counts the number of pairs $(a, b) \in A \times B$, i.e., $|A| \cdot |B| = |A \times B|$. The set AB contains all concatenations ab, where $(a, b) \in A \times B$. Clearly, there can be at most $|A \times B|$ such concatenations (there may be less, because not all concatenations are distinct). Hence, $|AB| \leq |A \times B| = |A| \cdot |B|$.

Exercise 3.8 Observe that $\overline{\{a\}} = \Sigma^* \setminus \{a\}$, so we will show that $\Sigma^* \setminus \{a\}$ is regular.

First assume that $a = \lambda$. In this case, $\Sigma^* \setminus \{a\} = \Sigma^+$, which is the plus closure of the regular language Σ, so it is regular.

Now assume that $a \in \Sigma$. Let $S = \Sigma \setminus \{a\}$. Then S is regular, because it is the union over all languages $\{s\}$, where $s \in \Sigma$ is a symbol other than a. (Alternatively, we can argue with Observation 3.6, because S is finite.)

Now let $L = \{\lambda\} \cup S \cup (\Sigma \circ \Sigma^+)$. Observe that L is regular, because it is the union of the regular languages $\{\lambda\}$, S, and $\Sigma \circ \Sigma^+$.

Thus, it suffices to show that $L = \Sigma^* \setminus a$. Consider any string in Σ^*. We will show that $x \in L$ if and only if $x \neq a$.

So, first assume $x \neq a$. If $|x| = 0$, then $x = \lambda \in L$. If $|x| = 1$, then $x \in \Sigma \setminus a \subseteq L$. Finally, if $|x| > 1$, then $x \in \Sigma \circ \Sigma^+ \subseteq L$.

Now assume $x = a$. Then $x \neq \lambda$, $x \notin S$, and $x \notin \Sigma \circ \Sigma^+$, so $x \notin L$.

Exercise 3.12

(a) The regular expressions are equivalent:

$$\begin{aligned}
L(ab)^* = \{ab\}^* &= \bigcup_{k=0}^{\infty} \{ab\}^k \\
&= \{ab\}^0 \cup \left(\bigcup_{k=1}^{\infty} \{ab\}^k \right) \\
&= \{\lambda\} \cup \left(\bigcup_{k=1}^{\infty} \{ab\}^k \right) \\
&= \{\lambda\} \cup \left(\bigcup_{k=0}^{\infty} \{ab\} \circ \{ab\}^k \right) \\
&= \{\lambda\} \cup \{ab\} \circ \left(\bigcup_{k=0}^{\infty} \{ab\}^k \right) \\
&= \{\lambda\} \cup \{ab\} \circ \{ab\}^* \\
&= L(\lambda + ab(ab)^*).
\end{aligned}$$

(b) The regular expressions are not equivalent. For example, $abab \in L((ab+bb)^2) \subseteq L((ab+bb)^*)$, because we just repeat twice the string ab, which is in $L(ab+bb)$. But $abab$ is not in $L((a+b)b^*)$: Each string in that language is a concatenation of a or b with a string in $\{b\}^*$. Thus, such a string can only contain one a, but $abab$ contains two a's.

(c) The statement is correct. Consider a word w in A^*. Then by definition of the star-closure, $w \in A^k$ for some $k \in \mathbb{N}_0$. If $k = 0$ then $w = \lambda \in B^*$. Now suppose $k > 0$. Then $w = w_1 w_2 \dots w_k$, where each $w_1, w_2, \dots, w_k \in A$. Since $A \subseteq B$, $w_1, w_2, \dots, w_k \in B$, and thus $w \in B^k \subseteq B^*$.

(d) The statement is not true: For example, if $A = \{00\}$ and $B = \{0, 1\}$, then A^* contains all strings of an even number of 0s, and B^* contains all bit strings. Therefore, $A^* \subseteq B^*$.

4 Finite-State Automata

A finite-state automaton (also sometimes called finite-state machine) is a simple abstract model for a computer with finite amount of memory. I.e., it is a hardware model.

4.1 Deterministic Finite-State Automata

Let Σ be some alphabet. The deterministic finite-state automaton (DFA) has a fixed number of states. It reads an input string $x = x_1 \dots x_n \in \Sigma^*$ symbol by symbol from left to right. At any point the DFA is in a given state q. Whenever it reads a new symbol x_i, it transitions from its current state into a new state. That state transition is defined by a transition function δ, which maps the current state, and the symbol read to a new state. For example, if the DFA is in state q and reads symbol x_i, then it transitions into the new state $q' = \delta(q, x_i)$. Once the DFA has processed the entire input this way, it reaches some final state q^*, and stops.

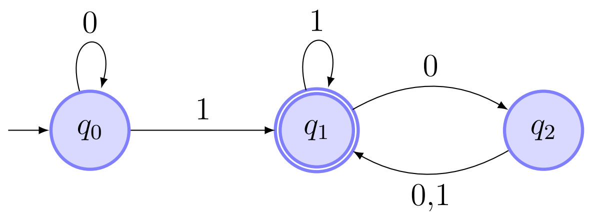
In order to compute something, we will have some special states that are called *accepting* states. Suppose upon processing input x, the DFA reaches the final state q^*. If q^* is an accepting state, then we say that the DFA *accepts* input x. Otherwise, it *rejects* the input. This way, each input $x \in \Sigma^*$ will either get accepted or rejected by the DFA. Thus, a DFA is suitable for solving *decision problems*, which have for each input either a "yes" or a "no" output, where accepting and rejecting and input can be thought of outputting "yes" and "no", respectively.

Formally, a deterministic finite-state automaton (DFA) is defined by the following five components:

- a finite set Q of *states*;
- a finite set Σ of input symbols (the *input alphabet*),
- a *transition function* $\delta : Q \times \Sigma \to Q$;
- an *initial state* $q_0 \in Q$; and
- a set $A \subseteq Q$ of *accepting states.*

Thus, a DFA is defined by a quintuple listing each of its components.

Definition 4.1. *A* deterministic finite automaton *(DFA) is a quintuple* $(Q, \Sigma, \delta, q_0, A)$*, where* Q *is the finite set of states,* Σ *is a finite set of input symbols, called input alphabet,* $\delta : Q \times \Sigma \to Q$ *is the transition function,* $q_0 \in Q$ *is the initial state, and* $A \subseteq Q$ *is the set of accepting states.*



(a) DFA Diagram

δ	0	1
q_0	q_0	q_1
q_1	q_2	q_1
q_2	q_1	q_1

(b) Transition Function δ

Figure 4.1: A Simple DFA.

We can depict DFAs using circles for states, and directed edges (arrows) with labels for state transitions. The initial state is depicted by an incoming arrow that does not leave any other state. Accepting states are indicated by double circles.

Example 4.2. The DFA depicted in Figure 4.1 (a) is the quintuple $(Q, \Sigma, \delta, q_0, A)$, where

- $Q = \{q_0, q_1, q_2\}$ (is the set of states),
- $\Sigma = \{0, 1\}$ (is the input alphabet),
- $\delta : Q \times \Sigma \to Q$ (the transition function) is defined as follows:

$$\begin{aligned} \delta(q_0, 0) &= q_0 & \qquad \delta(a, 1) &= q_1 \\ \delta(q_1, 0) &= q_2 & \qquad \delta(b, 1) &= q_1 \\ \delta(q_2, 0) &= q_1 & \qquad \delta(c, 1) &= q_1. \end{aligned}$$

- $q_0 = a$ (is the initial state), and
- $A = \{b\}$ (is the set of accepting states).

It is often easier to describe the transition function δ with a table. For example, the transition function for the automaton in Figure 4.1 (a) is given in Figure 4.1 (b). Each row corresponds to a state from Q, and each column to a symbol from Σ. The table entry in row x and column y is $\delta(x, y)$. ◀

As we discussed above, a DFA with input alphabet Σ either accepts or rejects an input string $x \in \Sigma^*$. To determine if it accepts the string or not, we follow the path from the starting state along the edges labelled with the symbols from the input string. I.e., if the input is $x = x_1 x_2 \ldots x_n$, then the i-th edge on the path should go along the edge labelled with x_i. We say the DFA accepts the input, if in the end an accepting state is reached, and otherwise it rejects the input.

To illustrate this, consider the DFA from Example 4.2, and suppose the input is 0011011. The path on the DFA corresponding to that input is as follows:

$$q_0 \xrightarrow{0} q_0 \xrightarrow{0} q_0 \xrightarrow{1} q_1 \xrightarrow{1} q_1 \xrightarrow{0} q_2 \xrightarrow{1} q_1 \xrightarrow{1} q_1.$$

Since b is an accepting state, the DFA accepts the input 0011011.

Definition 4.3. *Given a DFA M with input alphabet Σ, the* language recognized by M *is the set of all strings $x \in \Sigma^*$, such that M accepts x.*

Example 4.4. What languages do the DFAs M_1 and M_2 in Figure 4.2 accept?

(a) Consider the DFA M_1. Observe, that only the initial state, q_0, is an accepting state. As long as M_1 reads only a's, it remains in that state. But as soon as it reads a b, it transitions into state q_1, and remains there forever, because all edges leaving q_1 are self-loops going back to q_1.

Hence, the DFA accepts a string only, if it consists entirely of a's. Thus, it recognizes the language $\{a\}^*$.

(b) Now consider the DFA M_2. Its only accepting state is q_1. That state will be reached for the first time, when M_2 reads a 1. After that, state q_1 is reached again as long as the automaton reads only 1s, or if it reads a 00 or 01. In other words, the automaton ends up in an accepting state, for any string that contains a 1 and an even number of 0s after the last 1.

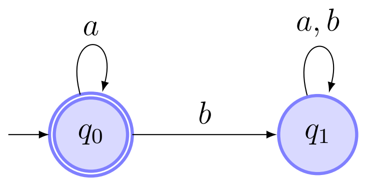
Thus, we make the following claim:

> *The automaton accepts a bit-string if and only if it contains at least one 1 and an even number of 0s after the last 1.*

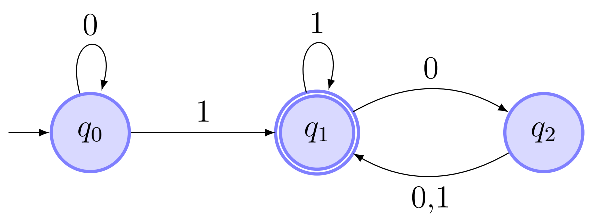
Observe that 0 is an even number, so a string that ends with a 1 (e.g., 101) also matches the above description.

To prove this, consider an input $x \in \{0, 1\}$. First assume that x is in the desired format, i.e., it contains a 1 and an even number of 0s after the last 1. We observe that each edge of M_2 labelled with 1 points to state q_1. Hence, after reading the last 1 in x, the automaton is in state q_1. Since M_2 only reads 0s afterwards, it alternates between states q_2 and q_1. As it reads an even number of 0s after the last 1, the automaton ends up in state q_1, which is an accepting state.

Now suppose that x is not in the desired format. Then either x contains no 1, or it contains an odd number of 0s after the last 1. If x contains no 1, then M_2 can never leave the starting state, q_0, which has a self-loop with label 0. As q_0 is not an accepting state, the DFA does not accept x. Now suppose x contains a 1 and an odd number of 0s after the last 1. As we argued before, after reading the last 1, M_2 is in state q_1, and then it



(a) DFA M_1



(b) DFA M_2

Figure 4.2: DFAs for Example 4.4.

alternates between q_2 and q_1 for each 0 it reads. Since it reads an odd number of 0s after the last 1, M_2 ends up in state q_2 and rejects. ◀

Example 4.5. Construct a DFA that recognizes any string over the alphabet $\Sigma = \{0, 1, 2\}$ that contains an even number of 0s (and any number of 1s and 2s).

To design such an automaton, we define states that can be used to "memorize" the important property of the string read up to the point when the DFA enters each state. For this specific language, we will use two states, q_{even} and q_{odd}. We design the automaton in such a way that after reading an even number of 0s, it will be in state q_{even}, and after reading an odd number of 0s it will be in state q_{odd}. Thus, the state "memorizes" the parity of the number of 0s read.

The corresponding automaton is depicted in Figure 4.3. It is easy to see that it has the desired property. In fact, we can prove this by induction on the length of the input. (This may seem like overkill, but it's a good opportunity to practice the skills learned in CPSC 251.)

Consider the following predicate $P(k)$: For any input $x \in \Sigma^k$ (i.e., the input has length k), after reading x the automaton is in state q_{even} if and only if x contains an even number of 0s.

Base Case: The empty string, λ, contains an even number of 0s. And after reading λ, the automaton is in the initial state, which is q_{even}. Hence, $P(0)$ is true.

Now assume that $P(k)$ is true for some $k \in \mathbb{N}_0$ (inductive hypothesis). We will show that $P(k+1)$ is also true. Hence, let $x = x_1 \dots x_k x_{k+1} \in \Sigma^{k+1}$ be a string of length $k+1$. Let q be the state the automaton reaches after reading the prefix $x_1 \dots x_k$ of length k. By the inductive hypothesis, $q = q_{\text{even}}$ if and only if $x_1 \dots x_k$ contains an even number of 0s. If $x_{k+1} \in \{1, 2\}$, then reading x_{k+1} makes the DFA remain in state q due to the self-loops labelled 1 and 2. In that case, x contains as many 0s as $x_1 \dots x_k$, and so the DFA ends up in state q_{even} if and only if x has an even number of 0s. Now assume that $x_{k+1} = 0$. Then upon reading x_{k+1}, the DFA transitions into the state that's not q. Thus, the DFA ends up in state q_{even} if and only if $x_1 \dots x_k$ contains an odd number of 0s. But since x contains one more 0 than $x_1 \dots x_k$, the DFA ends up in q_{even} if and only if x contains an even number of 0s. ◀

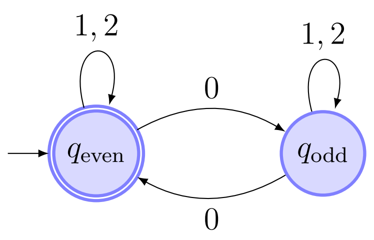


Figure 4.3: A DFA that accepts strings with an even number of 0s.

Example 4.6. Construct a DFA that recognize the language $L(r)$, where $r = 0(0+1)^*0$.

We claim that the DFA in Figure 4.4 does the job. State q_3 is a trap-state: Any input that starts with a 1 will force the DFA to become trapped in that state because of the self-loop labelled 0 and 1. Since q_3 is not an accepting state, the DFA will reject all inputs starting with 1. In other words, it can only accept inputs starting with 0.

For all inputs starting with a 0, the DFA will end up in states q_1 or q_2. State q_2 is used to memorize that the last symbol read (after the first 0) is a 0 (because all 0-arrows leaving q_1 or q_2 point to q_2), and q_1 is used to memorize that the last symbol is a 1. State q_2 is the only accepting state.

We will now prove that the DFA does indeed recognize $L(r)$. To that end, we will prove for each input $x \in L(r)$ that the DFA accepts x, and for each input $x \notin L(r)$ that the DFA will reject x.

First consider a string $x \in L(r)$. Then x begins and ends with a 0. I.e., $x = 0y0$, where $y \in \{0,1\}^*$. After reading the first 0, the DFA enters q_1. Since every state transition from one of q_1 and q_2 leads back to one of q_1 and q_2, the DFA remains in one of these two states. Finally, since all arrows with label 0 leaving one of these two states enter q_2, after reading a 0 the DFA will be in state q_2. Hence, after reading the final 0 of input x, the DFA is in q_2, which is an accepting state.

Now consider a string $x \notin L(r)$. Then either x does not begin with a 0 or it does not end with a 0. First assume x does not begin with a 0. Then when reading the first 1 of the input, the DFA transitions from the starting state, q_0 into state q_3. Due to the self-loops of q_3, the DFA remains in that state forever, and in the end rejects.

Now suppose that x does begin with a 0, but it ends with a 1. After reading the first 0, the DFA is in state q_1. As argued earlier, it will remain in states q_1 and q_2 forever. Since all 1-edges leaving any of these two states point to q_1, the DFA will be in state q_1 whenever it read a 1. Hence, when it reads the last 1 of input x it ends up in state q_1, and rejects. ◀

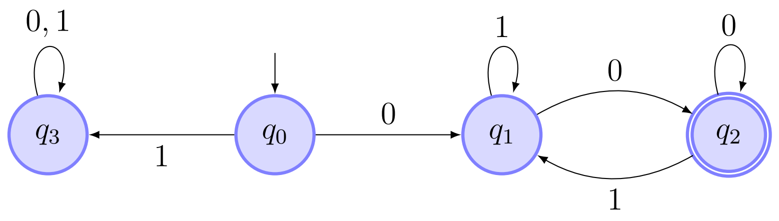


Figure 4.4: A DFA for $L\big(0(0+1)^*0\big)$.

4.2 Nondeterministic Finite Automata

We will now extend the power of our automata, by allowing them to make nondeterministic choices. This will lead to the definition of *nondeterministic finite automata*, or short NFAs. For each state q and each symbol a, an NFA may now have multiple outgoing arrows labelled with a, instead of just one. I.e., the automaton may need to choose one state transition among multiple valid options. In addition, we also allow that for some symbols in the input alphabet there are no arrows leaving q that are labelled with those symbols.

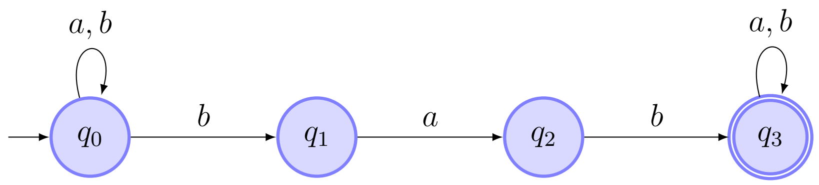
Consider the NFA depicted in Figure 4.5 (a). If in state q_0 the automaton reads the input b, then it may remain in state q_0, or it may transition into state q_1. Moreover, state q_1 has no outgoing edge labelled b, and state q_2 has no outgoing edge labelled a. Thus, if the automaton ever reaches state q_1, and the next input symbol to process is b, then it is "stuck".

Recall that for a DFA the transition function maps a pair $(q,x) \in Q \times \Sigma$ to a new state $\delta(q,x)$. This is the unique state the DFA would transition into upon reading symbol x in state q. But for an NFA that reads symbol x in state q, there may be multiple or no states it can transition into. Thus, we need to map (q,x) to a *set* of all possible new states the automaton can choose. We model this by letting δ be a mapping from $Q \times \Sigma$ to $\mathcal{P}(Q)$, which is the power set of Q. (Recall that the power set of a set S is the set of all subsets of S, and is denoted $\mathcal{P}(S)$.) Thus, $\delta(q,x)$ is a *set* of states instead of a single state. If q has no outgoing arrow labelled x, then $\delta(q,x) = \emptyset$.

For example, the automaton depicted in Figure 4.5 (a) has a transition function δ, where $\delta(q_0,b) = \{q_0,q_1\}$ and $\delta(q_0,a) = \{q_0\}$: If in state q_0 the automaton reads symbol b, then it can transition into one of q_0 and q_1, but if it reads a, it can only remain in q_0. Similarly, $\delta(q_1,b) = \emptyset$, because q_1 has no outgoing arrow labelled b. The full transition function of the automaton in Figure 4.5(a) is defined by the function table in Figure 4.5(b).

NFAs may have multiple choices for state transitions when processing an input. And only if the right choices are being made, an accepting state may be reached. For some inputs, however, no accepting state may be reached, no matter what choices an NFA makes.

Let N be an NFA with input alphabet Σ. Moreover, let $k \in \mathbb{N}_0$ and let $x = x_1 \ldots x_k \in \Sigma^k$ be an input. A *valid path* on N for input x is a path of length $k+1$ that begins in the initial state



(a) A Nondeterministic Automaton

δ	a	b
q_0	$\{q_0\}$	$\{q_0, q_1\}$
q_1	$\{q_2\}$	$\emptyset$
q_2	$\emptyset$	$\{q_3\}$
q_3	$\{q_3\}$	$\{q_3\}$

(b) Transition Function

Figure 4.5: A Nondeterministic Finite Automaton

of N, and where the i-th edge on the path is labelled x_i. Hence, a path $(v_0, \dots, v_k)$ on N is valid for input $x = x_1 \dots x_k$, if v_0 is the initial state, and for each $i \in \{0, \dots, k-1\}$ the NFA has an edge from v_i to v_{i+1} that is labelled with x_i.

Consider, for example the NFA in Figure 4.5, and the input *abbaba*. One valid path for that input is $(q_0, q_0, q_0, q_0, q_0, q_1, q_2)$:

$$q_0 \xrightarrow{a} q_0 \xrightarrow{b} q_0 \xrightarrow{b} q_0 \xrightarrow{a} q_0 \xrightarrow{b} q_1 \xrightarrow{a} q_2.$$

Another one is $(q_0, q_0, q_0, q_1, q_2, q_3, q_3)$:

$$q_0 \xrightarrow{a} q_0 \xrightarrow{b} q_0 \xrightarrow{b} q_1 \xrightarrow{a} q_2 \xrightarrow{b} q_3 \xrightarrow{a} q_3.$$

Observe that the first valid path ends up in a non-accepting state, and the second valid path ends up in an accepting state. We say the NFA *can accept* an input x, if there *exists* a valid path for that input, which ends up in an accepting state.

Definition 4.7. *Let N be an NFA with input alphabet Σ. The language* recognized *by N is*

$$L(N)\{x \in \Sigma^* \mid N \textit{ can accept } x\}.$$

We can think of NFA's as randomized machines, where whenever the NFA has multiple choices, it chooses one of them at random. Thus, each valid path corresponds to one particular computation, based on which random choices the NFA makes. But the language $L(N)$ recognized by N is not defined in terms of a particular computation. I.e., it is not about whether for a given input the NFA ends up in an accepting state for one particular computation the NFA performs. Instead, we ask if there *exists* a possible computation that leads the NFA into an accepting state. In other words, an input x is in $L(n)$, if and only if the NFA *can* make the choices that lead to an accepting state (once the input has been processed). This is equivalent to saying that if the NFA always chooses uniformly at random among all its options, then the probability that it ends up in an accepting state is not 0.

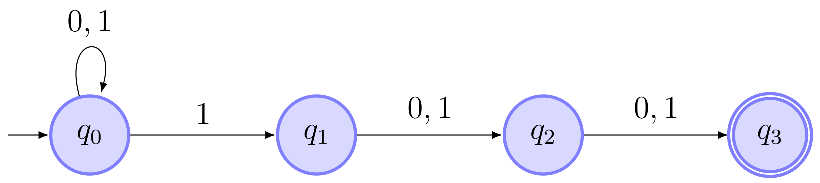


Figure 4.6: NFA for Example 4.8

In a sense, this is similar to a labyrinth, where the question is not to *find* a path out of the labyrinth, but rather if there *exists* a path out.

Note that if we consider one particular computation by an NFA, it may happen that before the input has been fully processed, the NFA ends up in a state such that no transition is possible anymore. For example, for the NFA in Figure 4.5 and the input bb, if the NFA decides at random to transition from q_0 to q_1 when processing the first input symbol, b, then it ends up in q_1 and now needs to process the second b. But q_1 has no outgoing edge labelled b. In this case, we assume that the NFA simply stops processing the input (obviously without reaching an accepting state).

Example 4.8. What language does the NFA in Figure 4.6 recognize?

If the NFA is in state q_0, and it reads a 0, then it must remain in state q_0. If it reads a 1, it can choose to remain in state q_0, or it can choose to transition into state q_1. But once it is in q_1, it must read exactly two more input symbols, in order to reach an accepting state q_3. If it reads fewer than two more input symbols, then it will end up in q_1 or q_2, both of which are non-accepting states. If it reads more than two input symbols, then it will be “stuck” in q_3, unable to process all input symbols.

Thus, we claim that the NFA recognizes the language

$$L = L\big((0+1)^*1(0+1)(0+1)\big).$$

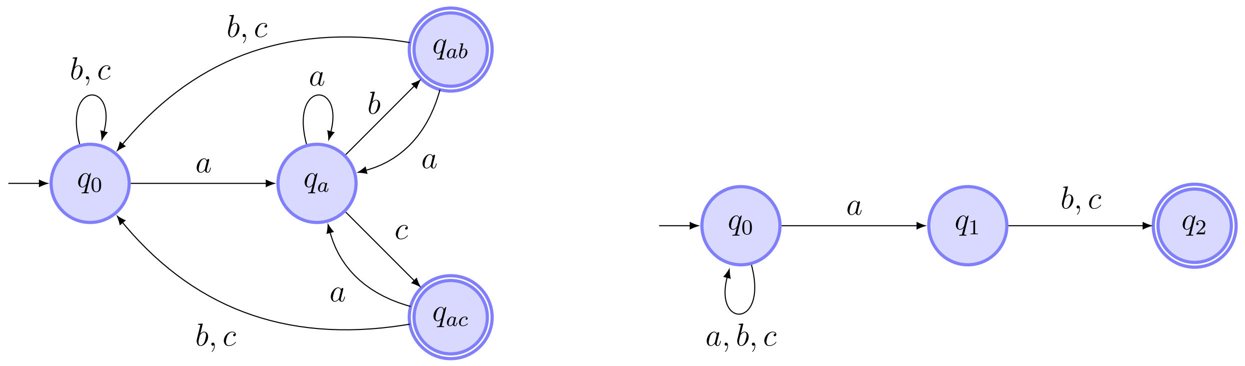
This language contains all strings that have a 1 in the third last position.

To prove that the NFA recognizes L, first consider an input $x \in L$. Then $x = x_1 \ldots x_k 1 x_{k+2} x_{k+3} \in \{0,1\}^{k+3}$. There is a valid path on the NFA that leads to the accepting state q_3:

$$q_0 \overset{x_1}{\to} q_0 \overset{x_2}{\to} q_0 \overset{x_3}{\to} \ldots \overset{x_k}{\to} q_0 \overset{1}{\to} q_1 \overset{x_{k+2}}{\to} q_2 \overset{x_{k+3}}{\to} q_3.$$

Hence, the NFA can accept x, and thus x is in the language recognized by the NFA.

Now consider an input $x = x_1 \ldots x_n$ that the NFA can accept. Then there is a valid path $(v_0, v_1, \ldots, v_n)$, where $v_0 = q_0$ and v_n is the only accepting state, q_3. The last three edges on that path must go from q_0 to q_1, from q_1 to q_2, and from q_2 to q_3. These edges are labelled with the last 3 symbols of x, i.e, with x_{n-2}, x_{n-1}, and x_n, respectively. Since the edge from q_0 to q_1



(a) DFA (b) NFA

Figure 4.7: Automata recognizing the language of all strings ending in ab and ac.

is labelled 1, we conclude that $x_{n-2} = 1$. It follows that the third last symbol of x is 1, and thus $x \in L$. ◀

Example 4.9. Suppose we want to construct a DFA that recognizes the language of all strings over $\Sigma = \{a, b, c\}$ that end with ab or ac. We need separate states to memorize if the last symbol processes was an a, as well as if the last two symbols processed were ab or ac.

The DFA in Figure 4.7 (a) recognizes that language. It is in state q_a whenever it has read a prefix of the input that ends with a. If it has processed a prefix ending in ab, then the DFA is in state q_{ab}, and if the prefix ends in ac, then the DFA is in state q_{ac}. These properties are not hard to prove, but we will leave that as an exercise.

It is, however much easier to construct an NFA for that language: An NFA can "guess", when there are only 2 symbols left to read of an input x.

Consider the NFA in Figure 4.7 (b). It starts in state q_0, and remains in state q_0 as long as it reads only b's or c's, and it may remain in state q_0 for each a that it reads. However, when it reads an a and guesses that this is the second last symbol of the input, it transitions into state q_1. After that, it must read exactly one more symbol b or c in order to accept.

It is not hard to see that the NFA can accept an input if and only if that input ends with ab or bc. ◀

4.2.1 λ-Transitions

We will now extend NFAs even further, by allowing transitions without processing any inputs. These are called λ-*transitions*. Each arrow pointing from one state to another can now be

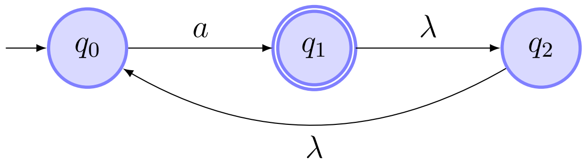


Figure 4.8: An NFA with λ-transitions.

marked with λ instead of a symbol from the input alphabet. Thus, the transition function is now

$$\delta : Q \times (\Sigma \cup \{\lambda\}) \to \mathcal{P}(Q).$$

I.e., it still maps a pair (q, x) to a set $\delta(q, x) \subseteq Q$ of new states, but now it is also defined for $x = \lambda$.

Definition 4.10. *A* nondeterministic finite automaton *(NFA) is a quintuple* $(Q, \Sigma, \delta, q_0, A)$, *where* Q *is the finite set of states,* Σ *is a finite set of input symbols, called input alphabet,* $\delta : Q \times (\Sigma \cup \{\lambda\}) \to \mathcal{P}(Q)$ *is the transition function,* $q_0 \in Q$ *is the initial state, and* $A \subseteq Q$ *is the set of accepting states.*

Whenever the NFA reaches a state q, then it may transition from q into any state in $\delta(q, \lambda)$ without reading an input symbol. Thus, we have to adjust our definition of a valid path: A valid path for an input $x = x_1 \dots x_n \in \Sigma^*$ of length n begins in the starting state, has exactly n non-λ edges, and the i-th non-λ edge is labelled with x_i.

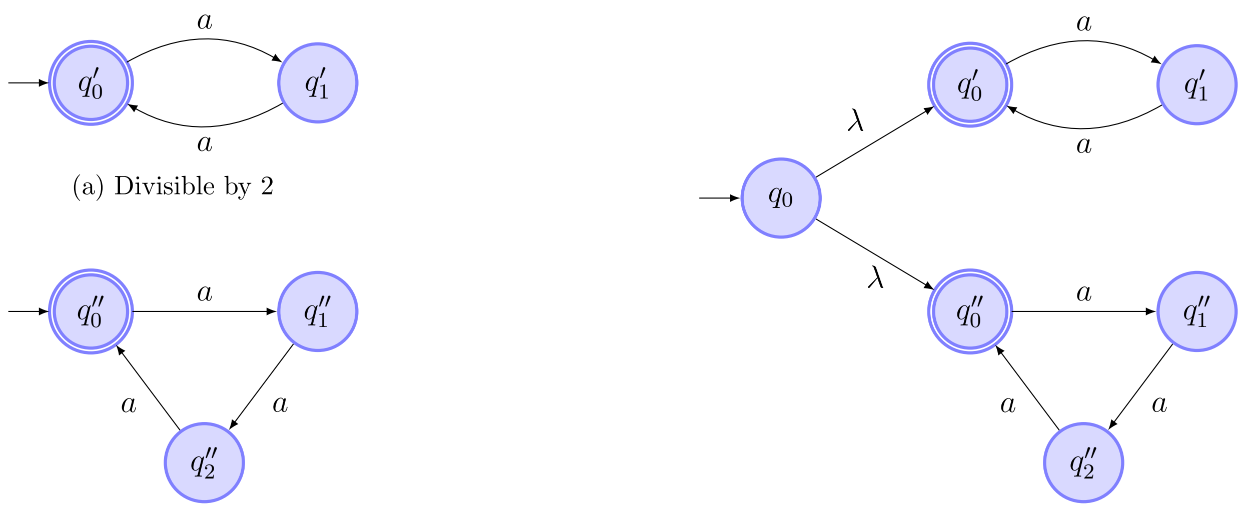
As with NFAs without λ-transitions, we say that the NFA *can accept* an input x, if there exists a valid path for x that ends up in an accepting state.

The analogy to a randomized automaton also still works: We can assume that whenever the NFA has multiple options, including taking λ-transitions, it chooses one of the options at random. Thus, the NFA N can accept an input x if and only if the probability that N reaches an accepting state at the end of processing x is not zero.

Example 4.11. Consider the NFA in Figure 4.8. This NFA recognizes the language $\{a\}^+$, i.e., all strings of one or more a's.

First, consider an input string in $\{a\}^+$. Thus, the input is a^k for some $k \geq 1$. The cycle (q_0, q_1, q_2, q_0) "consumes" exactly one a. Thus, the NFA can read $k-1$ a's by going around that cycle $k-1$ times. It then is in state q_0. From there it can transition into the accepting state q_1, processing the k-th and final a:

$$\underbrace{q_0 \xrightarrow{a} q_1 \xrightarrow{\lambda} q_2 \xrightarrow{\lambda}}_{\text{cycle 1}} \underbrace{q_0 \xrightarrow{a} q_1 \xrightarrow{\lambda} q_2 \xrightarrow{\lambda}}_{\text{cycle 2}} \dots \xrightarrow{\lambda} \underbrace{q_0 \xrightarrow{a} q_1 \xrightarrow{\lambda} q_2 \xrightarrow{\lambda}}_{\text{cycle } k-1} q_0 \xrightarrow{a} q_1.$$



(a) Divisible by 2

(b) Divisible by 3

(c) Divisible by 2 or 3

Figure 4.9: Finite automata recognizing strings of lengths divisible by 2, 3, and 2 or 3.

Now consider a string x that the NFA can accept. Thus, there is a valid path for x that ends in q_1, which is the only accepting state. Such a path must consist of $\ell \geq 0$ repetitions of the cycle (q_0, q_1, q_2, q_0), followed by the edge (q_0, q_1). The non-λ symbols on the edge labels of such a path form the string $a^{\ell+1}$. Hence, $x \in \{a\}^+$. ◀

Example 4.12. Let $\Sigma = \{a\}$. Construct an NFA that recognizes the language of all strings of length n, where n is divisible by 2 or by 3.

It is easy to construct a DFA that recognizes the language of all strings of length divisible by 2, and similarly also for the language of strings of length divisible by 3—see Figure 4.9 (a) and (b). But to construct a DFA that accepts both, strings of even length and strings of length divisible by 3, complicates things considerably. An NFA can use an initial λ-transition in order to "guess" if the string is going to have length divisible by 2 or by 3, and then only check one of the two conditions.

The NFA in Figure 4.9 (c) does exactly that: From the initial state, q_0, we have two λ-transitions. One leads to q_0', which corresponds to the initial state of a DFA (with states q_0' and q_1'), which accepts strings of even length. The other λ-transitions leads to q_0'', which corresponds to the initial state of a DFA that accepts all strings of length divisible by 3.

Clearly, if an input has length divisible by 2 or 3, then the NFA *can* accept that input, by first performing the appropriate λ-transition, and then processing the entire input.

On the other hand, if the NFA accepts an input of length n, then it must first perform one of the two λ-transitions, and then read exactly n symbols, where n is divisible by 2 (if it transitioned into q_0') or by 3 (if it transitioned into q_0''). ◀

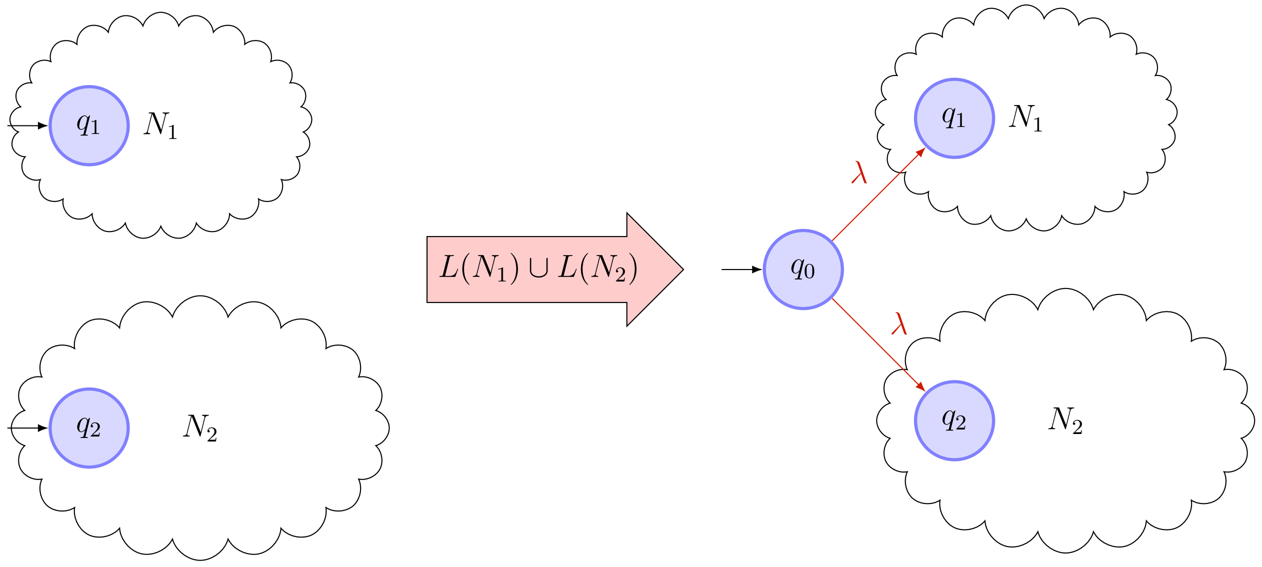


Figure 4.10: Constructing an NFA that recognizes $L(N_1) \cup L(N_2)$.

4.2.2 NFA Closure of Union

Consider two NFAs N_1 and N_2. Example 4.12 shows that we can easily construct an NFA N that recognizes $L(N_1) \cup L(N_2)$, i.e., N can accept an input if and only if either N_1 can accept it or N_2 can accept it. (In Example 4.12, $L(N_1)$ contains all strings of even length, and $L(N_2)$ all strings of length divisible by 3.)

The following theorem formalizes this observation.

Theorem 4.13. *Let N_1 and N_2 be NFAs with the same* [1] *input alphabet Σ. Then there is an NFA N that recognizes $L(N_1) \cup L(N_2)$.*

Proof. For each $i \in \{1, 2\}$, let $N_i = (Q_i, \Sigma, \delta_i, q_i, A_i)$. We may assume w.l.o.g. that Q_1 and Q_2 are disjoint (if not, we can just rename states of one of the two automata to unique ones).

Let q_0 be a state that is not in $Q_1 \cup Q_2$. Our new NFA uses q_0 as its initial state, and $Q_1 \cup Q_2 \cup \{q_0\}$ as its set of states. It then has a λ-transition from q_0 to q_1 and from q_0 to q_2. All transitions between states in Q_1 and Q_2 remain the same, and all accepting states of N_1 or N_2 are also accepting states of N. See Figure 4.10 for an illustration of the construction.

Thus, $N = (Q, \Sigma, \delta, q_0, A)$, where

- $Q = Q_1 \cup Q_2 \cup \{q_0\}$ for a new state $q_0 \notin Q_1 \cup Q_2$,

[1] The same construction given in the proof of this theorem also works for NFAs with different input alphabets, Σ_1 and Σ_2. The resulting NFA N then has the input alphabet $\Sigma = \Sigma_1 \cup \Sigma_2$.

- $\delta : Q \times (\Sigma \cup \{\lambda\}) \to \mathcal{P}(Q)$ is defined as follows:
  - $\delta(q_0, \lambda) = \{q_1, q_2\}$;
  - $\delta(q_0, a) = \emptyset$ for each $a \in \Sigma$;
  - $\delta(q, a) = \delta_i(q, a)$ for each $i \in \{1, 2\}$, $q \in Q_i$ and $a \in \Sigma_i$.
- $A = A_1 \cup A_2$.

It is easy to see that $L(N) = L(N_1) \cup L(N_2)$: Suppose $x \in L(N)$. Then for x there is a valid path in N that leads to an accepting state. That path must lead from q_0 over one of the two λ-transitions to a state q_i, where $i \in \{1, 2\}$. Thus, there is a valid path for x in N_i that leads to a state in A_i (the set of N_i's accepting states). Hence, N_i can accept x, and so $x \in L(N_i)$. Thus, $x \in L(N_1) \cup L(N_2)$.

Now assume that $x \in L(N_1) \cup L(N_2)$. Then $x \in L(N_i)$ for some $i \in \{1, 2\}$, and thus, there is a valid path $(q_1, \ldots, q_k)$ for x in N_i that leads to an accepting state of N_i, i.e., $q_k \in A_i$. Then the path $(q_0, q_1, \ldots, q_k)$ is a valid path for x on N, because it first uses the λ-transition from q_0 to q_1 and then exactly the same edges as the valid path in N_i. Since N's accepting states form the set $A_1 \cup A_2$, q_k is an accepting state of N. Hence, N can accept x, and thus $x \in L(N)$. $\square$

4.2.3 NFA Closure of Concatenation

Example 4.14. Consider the language L over $\Sigma = \{a, b, c\}^*$, which consists of all strings $x' \circ x''$, where $x', x'' \in \Sigma^*$, and x' contains an even number of a's and x'' an even number of b's.

The NFA in Figure 4.11 recognizes this language: States q_0 and q_1 form a DFA, which upon reading an input with an even number of a's ends up in state q_0, and otherwise in state q_1. Similarly, states q_0' and q_1' form a DFA that accepts any string with an even number of b's, assuming that q_0' is the initial state.

Thus, the NFA can accept any string $x = x'x''$, if x' contains an even number of L's and x'' and even number of b's: It first processes x' using states q_0 and q_1, and ends up in state q_0. Then it uses the λ-transition and enters state q_0'. Finally, it processes x'', using states q_0' and q_1', and since y contains an even number of b's it ends up in the accepting state q_0'.

Now suppose the NFA can accept a string x. Then on the valid path leading to q_0', the NFA must use the λ-transition from q_0 to q_0'. Let x' be the prefix of x that the NFA reads before taking the λ-transition, and x'' the postfix that the NFA reads after taking the λ-transition. Then x' contains an even number of a's, because the valid path leads from q_0 back to q_0. Similarly, x'' contains an even number of b's, because after the λ-transition the valid path leads from q_0' back to the q_0', which is the only accepting state. Hence, $x = x'x''$, where x' contains an even number of a's and x'' an even number of b's. Thus, $x \in L$. ◀

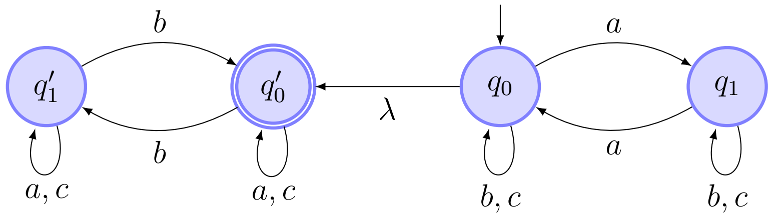


Figure 4.11: NFA for Example 4.14.

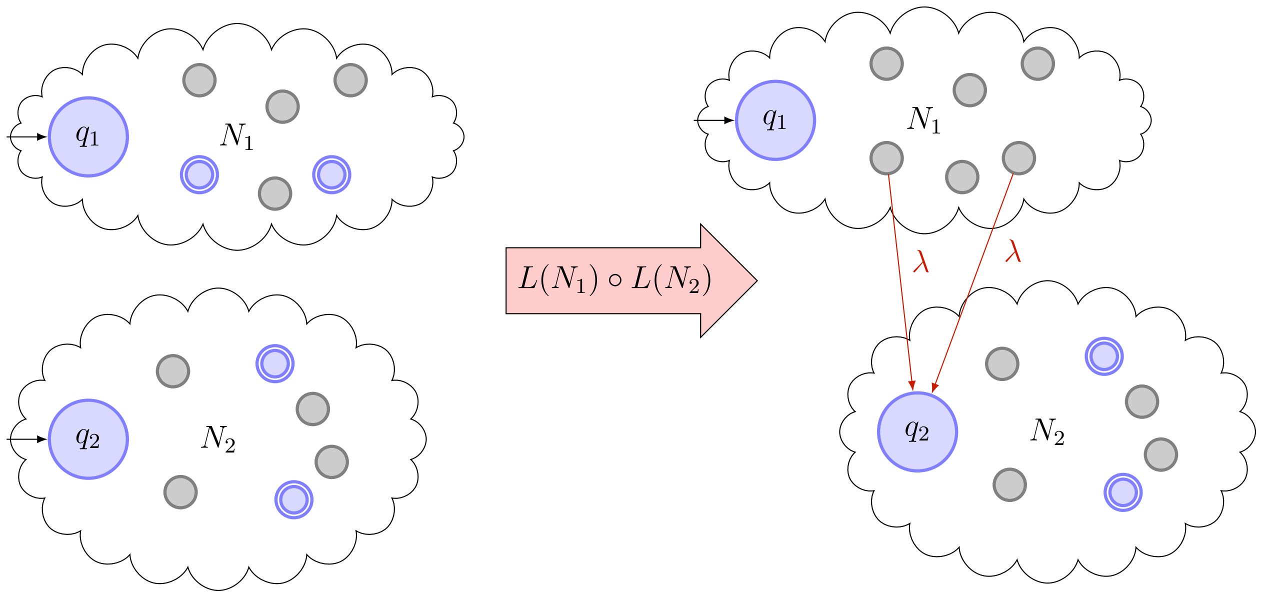


Figure 4.12: Constructing an NFA that recognizes $L(N_1) \circ L(N_2)$.

Theorem 4.15. *Let N_1 and N_2 be NFAs with the same*[2] *input alphabet Σ. Then there exists an NFA N that recognizes $L(N_1) \circ L(N_2)$.*

Proof. For each $i \in \{1, 2\}$ let $N_i = (Q_i, \Sigma, \delta_i, q_i, A_i)$. We may assume w.l.o.g. that Q_1 and Q_2 are disjoint (otherwise we can just rename the states of one of the two NFAs).

Our new NFA N copies all states of N_1 and N_2, with q_1 as the new starting states. All existing edges between states of N_1 and N_2 remain unchanged. Moreover, we add a λ-transition from each accepting state of N_1 to q_2, the initial state of N_2. Finally, we replace all the accepting states of N_1 with non-accepting ones. I.e., the accepting states of the new NFA, N, are the same as that of N_2.

Formally, $N = (Q, \Sigma, \delta, q_1, A)$, where

[2]The same construction given in the proof of this theorem also works for NFAs with different input alphabets, Σ_1 and Σ_2. The resulting NFA N then has the input alphabet $\Sigma = \Sigma_1 \cup \Sigma_2$.

- $Q = Q_1 \cup Q_2$;
- $\delta : Q \times (\Sigma \cup \{\lambda\}) \to \mathcal{P}(Q)$, where for each $q \in Q$ and $a \in \Sigma \cup \{\lambda\}$:
  - if $q \in A_1$ and $a = \lambda$, then $\delta(q, a) = \delta_1(q, \lambda) \cup \{q_2\}$ (this adds the λ-transitions from the accepting states of N_1 to q_2);
  - otherwise $\delta(q, a) = \delta_i(q, a)$, where i is the index such that $q \in Q_i$.
- $A = A_2$.

Consider an input $x \in L(N_1) \circ L(N_2)$. Then $x = x'x''$, where $x' \in L(N_1)$ and $x'' \in L(N_2)$. Thus, in N_1 there is a valid path $(v_1, \ldots, v_k)$ for x' from q_1 to an accepting state $v_k \in A_1$, and in N_2 there is valid path $(u_1, \ldots, u_\ell)$ for x'' from q_2 to an accepting state $u_\ell \in A_2$. Then the path $(v_1, \ldots, v_k, u_1, \ldots, u_\ell)$ is a valid path for x (which uses the λ-transition from v_k to u_1). Since u_ℓ is an accepting state of N, $x \in L(N)$.

We leave it as an exercise to show that if $x \in L(N)$, then $x \in L(N_1) \circ L(N_2)$. $\square$

Theorem 4.16. *Let N be an NFA. Then there is an NFA N_1 that recognizes $L(N)^+$ and an NFA N_2 that recognizes $L(N)^*$.*

Proof Sketch. To construct N_1, we simply need to add a λ-transition from each accepting state of N back to N's initial state. It is easy to see that N_1 recognizes $L(N)^+$.

We can obtain N_2 from N_1 by simply adding another accepting state q^*, and a λ-transition q^* to N_1's initial state, q_0. Then N_2 recognizes the language $L(N)^+ \cup \{\lambda\} = L(N)^*$.

A full proof is left as an exercise. $\square$

4.2.4 NFAs and Regular Expressions

Corollary 4.17. *Let r be a regular expression over the alphabet Σ. Then there exists and NFA N with input alphabet Σ, which recognizes $L(r)$.*

Proof. We can prove this by strong induction on the number k of operations ($\circ$, $+$, Kleene closure, and positive closure) the expression uses.

Base Case: $k = 0$. Let r be a regular expression that uses no operations. By Definition 3.9, r is one of $\emptyset$, λ, and a, where $a \in \Sigma$. NFAs for the languages defined by these regular expressions are depicted in Figure 4.14. If $r = \emptyset$, then we can simply use the NFA that consists of a single initial state, which is not an accepting state, and has no edges. If $r = \lambda$, then we use the NFA with a single initial state, which is also an accepting state, and which has no edges. Finally, if $r = a$ for $a \in \Sigma$, then we can use a two-state NFA as depicted in Figure 4.14 (c).

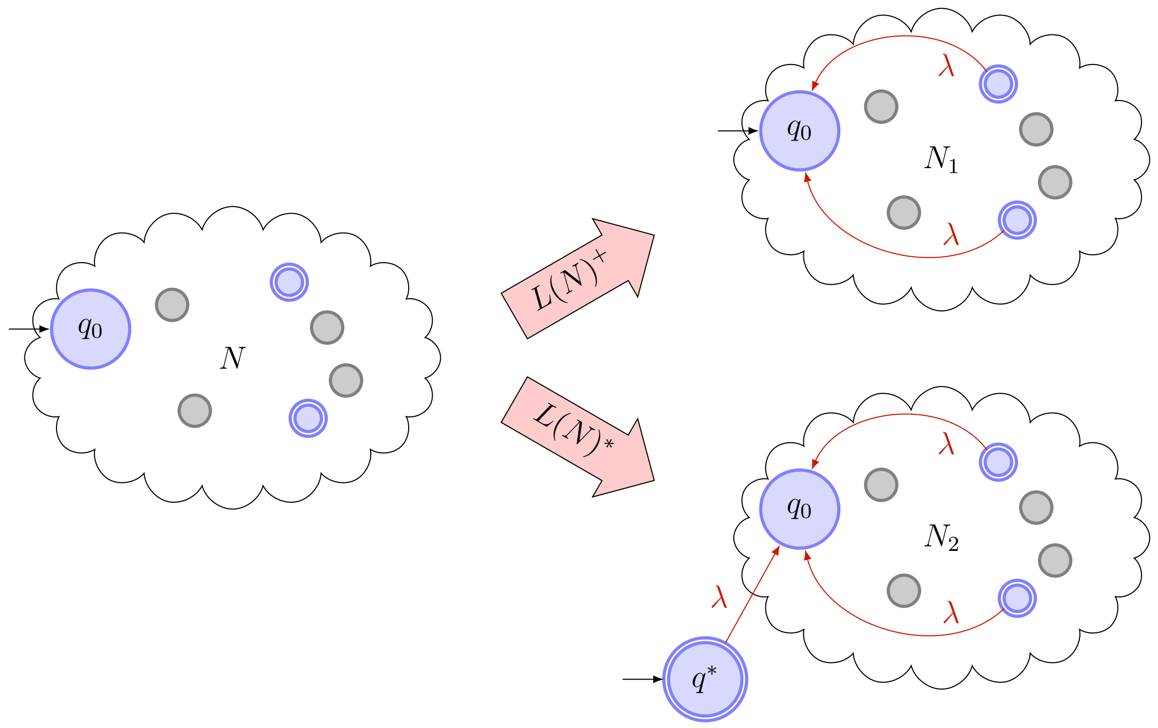


Figure 4.13: Constructing NFAs for $L(N)^+$ and $L(N)^*$.

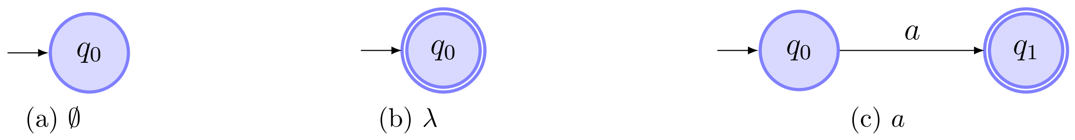


Figure 4.14: NFAs for languages defined by the regular expressions $\emptyset$, λ, and a, where $a \in \Sigma$.

Inductive Step: Assume that the claim is true for all regular expressions that use at most k operations. Let r be a regular expression using $k+1$ operations. Then r is none of $\emptyset$, λ, and a for $a \in \Sigma$. By Definition 3.9, r is one of $r_1 \circ r_2$, $r_1 + r_2$, r_1^*, or r_1^+, where r_1 (and r_2 if applicable) are regular expressions that use at most k operations. By the inductive hypothesis, we can construct NFAs N_1 and N_2 that recognize $L(r_1)$ and $L(r_2)$, respectively. It then follows from Theorems 4.13, 4.15 and 4.16 that we can construct an NFA N from N_1 and N_2 that recognizes $L(r)$. □

Theorem 4.18. *Let N be an NFA. Then there is a regular expression r such that $L(r) = L(N)$.*

Proof Sketch. We again use strong induction, this time on the total number edges, k.

Base Case: $k = 0$. Then N has no edges. Thus, the only valid path for any input is the path of length 0, which starts and ends at q_0. Then $L(N) = \{\lambda\}$, if q_0 is an accepting state, and otherwise $L(N) = \emptyset$. Thus, $L(N) = L(r)$, where r is one of the regular expressions λ and $\emptyset$.

Inductive Step: Now assume that N has $k \geq 1$ edges. The inductive hypothesis is that every NFA with fewer than k edges has a regular expression that defines the language the NFA recognizes. Let q_0 be the initial state of N.

Case 1: First assume that q_0 has no incoming edges. If it also has no outgoing edges, either, then N is equivalent to an NFA with a single state and no edges, and so the claim follows from the base case. Hence, assume that q_0 has at least one outgoing edge labelled s, going to a state $q_1 \neq q_0$, where $s \in \Sigma \cup \{\lambda\}$.

We obtain N_1 from N by removing the edge from q_0 to q_1. For N_2 we also remove that edge, but in addition we make q_1 the initial state. Since N_1 and N_2 each have one fewer edge than N, there are regular expressions r_1 and r_2, such that $L(r_1) = L(N_1)$ and $L(r_2) = L(N_2)$. The construction of N_1 and N_2 is depicted in Figure 4.15.

Observe that N can accept an input $x \in \Sigma^*$ in exactly two ways:

1. It first reads s while transitioning from q_0 to q_1, and then finds a valid path to an accepting state in N_2. In that case, $x = s \circ x'$ for $x' \in \Sigma^*$, and thus N_2 can accept x'. (For this to be true it is important that q_0 has no incoming edges. Otherwise, there could be a valid path in N, first cycling back to q_0, and then using the edge from q_0 to q_1, before continuing to an accepting state. And that valid path would not translate to a valid path in N_2.)
2. It does not transition from q_0 to q_1. In that case, N_1 can accept x.

It is also not hard to see that if N_2 accepts a string x', then $s \circ x'$ is in $L(N)$, and if N_1 accepts a string x, then $x \in L(N)$.

It follows that $L(N) = L(r_1 + s \circ r_2)$. Hence, $r = r_1 + s \circ r_2$ is the desired regular expression. □

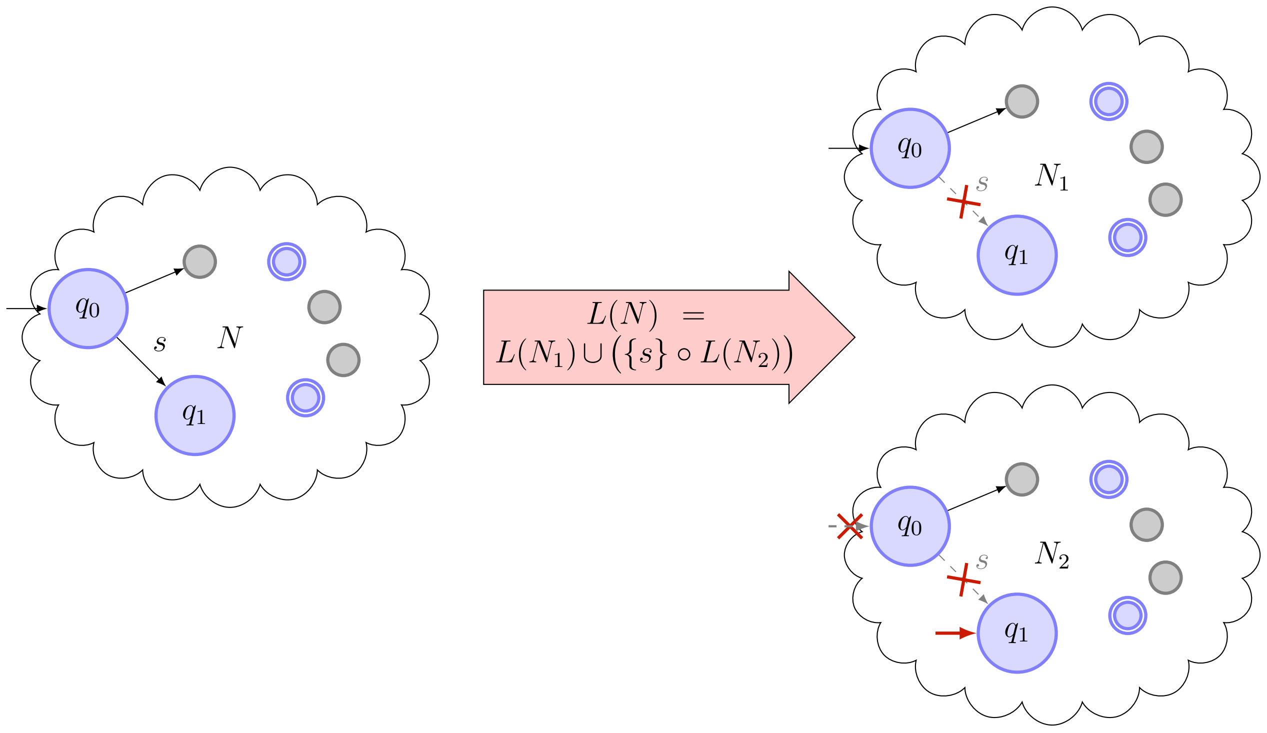


Figure 4.15: Decomposition of an NFA N whose initial state has no incoming edges.

Case 2: Now assume that q_0 has at least one incoming edge. Suppose one such edge is from a state q_1 to q_0, and it is labelled $s \in \Sigma \cup \{\lambda\}$. Again, we remove that edge to obtain an NFA N_1. Then we create another NFA N_2 from N_1 by making only q_1 an accepting state and making all other states non-accepting. See Figure 4.15 for an illustration of the construction.

Since N_1 and N_2 are smaller than N, there are regular expressions r_1 and r_2 such that $L(N_1) = L(r_1)$ and $L(N_2) = L(r_2)$. We let $r = (r_2 \circ s)^* \circ r_1$. (If $s = \lambda$, then this simplifies to $r = (r_2)^* r_1$.)

We prove that $L(r) = L(N)$. First consider an input $x \in L(r)$. We will show that $x \in L(N)$. Since $x \in L((r_2)^* r_1)$, it can be written as $x'x''$, where $x' \in L((r_2 \circ s)^*)$ and $x'' \in L(r_1)$. Then $x' = x_1 s x_2 s \dots x_\ell s$ for some $\ell \in \mathbb{N}_0$, where $x_i \in L(r_2)$ for each $i \in \{1, \dots, \ell\}$. Since q_1 is the only accepting state of N_2, there is a valid path from q_0 to q_1 for each x_i. Therefore, in N, there is a valid path for $x_i s$ that goes from q_0 to q_1 and then back to q_0. Thus, here is a valid path for $x' = x_1 s x_2 s \dots x_\ell s$ in N_1 that goes from q_0 back to q_0 (cycling ℓ times). Since x'' is in $L(r_2)$, there is a valid path in N_2 from q_0 to an accepting state q^* of N_2. That path also exists in N. Hence, $x'x''$ has a valid path from q_0 to q_0 (processing x') and then continuing to q^* (while processing x''). Since q^* is also an accepting state of N, it follows that $x \in L(N)$.

Now assume $x \in L(N)$. Then in N there is a valid path for x to an accepting state q^*. Suppose that path passes through the s-edge from q_1 to q_0 exactly ℓ times. Then x can be written as $x_1 s x_2 s \dots x_\ell s x_{\ell+1}$, such that there is a valid path from q_0 to q_1 for each x_i, $i \in \{1, \dots, \ell\}$,

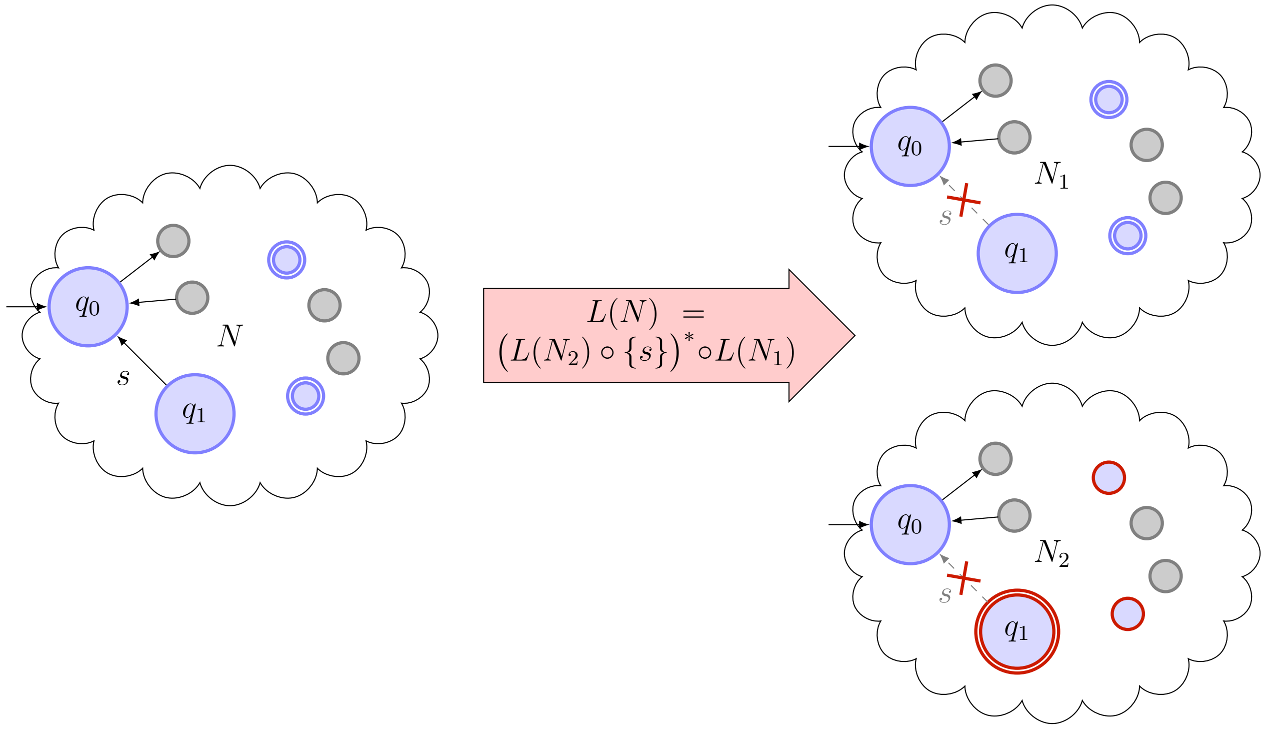


Figure 4.16: Decomposition of an NFA N whose initial state has at least one incoming edge.

which does not use the s-edge from q_1 to q_0. Moreover, there is a valid path from q_0 to q^* for $x_{\ell+1}$, which also does not pass through the s-edge from q_1 to q_0. It follows that N_1 can accept $x_{\ell+1}$ and N_2 can accept x_i for each $i \in \{1, \dots, \ell\}$. Hence, $x_{\ell+1} \in L(r_1)$, $x_i \in L(r_2)$, and thus $x = x_1 s x_2 s \dots x_\ell s x_{\ell+1} \in L((r_2 s)^* r_1) = L(r)$. □

4.2.5 Converting NFAs to DFAs: The Power Set Construction

We say two automata are *equivalent*, if they recognize the same language. We will now show how to convert an NFA N to an equivalent DFA D, i.e., $L(N) = L(D)$. The algorithm we use is called *power set construction*, for a reason that will become clear shortly.

Suppose N has the state set Q. Contrary to a DFA, an NFA can reach multiple possible states for a given input. A DFA can only reach exactly one state. The idea is now that while processing the input, a DFA D must keep track of (or memorize), the *set of all possible states* that N can have reached if it has read the same input symbols as D.

We choose D's set of states as the power set of Q, $\mathcal{P}(Q)$. (Hence, the name power set construction.) I.e., each of D's states corresponds to a *set* of N's states. Suppose that after processing input x DFA D reaches a state $\{q_1, \dots, q_k\}$ (where $q_1, \dots, q_k$ are states of N). Then we want this to mean that N *can* reach any of the states $q_1, \dots, q_k$ for input x, but no other states.

The initial state of D contains the initial state of N, as well as all states that N can reach through λ-transitions from N's initial state. We call this the λ-*closure* of N's initial state. More generally, the λ-closure of a set S of states of an NFA, is the set of all states that can be reached from any state $q \in S$ using zero or more λ-transitions. The λ-closure of set S is denoted S^λ.

Consider for example the NFA in Figure 4.17 (a). Then $\{q_0\}^\lambda = \{q_0, q_2\}$, and $\{q_0, q_1\}^\lambda = \{q_0, q_1, q_2\}$.

Hence, if N's initial state is q_0, then D's initial state is $\{q_0\}^\lambda$. This reflects the fact that if the input is an empty string, then N can reach any state in $\{q_0\}^\lambda$.

We can now describe how to add the state transitions for D. Consider a state $S \in \mathcal{P}(Q)$ of D (i.e., $S \subseteq Q$ is a set of N's states). Recall that we want D to reach state S for input x if and only if N can reach any state $q \in S$ for that input. Suppose this is true for a given state S of D (as it is for $x = \lambda$ and the initial state of D). Consider a symbol $a \in \Sigma$. Then, after processing input $x \circ a$, N can reach any state q', such that there is a state $q \in S$ and an a-edge from q to q' in N. In addition, N can reach any state that it can reach from q' through λ-transitions. Hence, for input $x \circ a$, N can reach any state in S_a^λ, where

$$S_a = \{q' \in Q \mid \exists q \in S : q' \in \delta(q, a)\}. \tag{4.1}$$

Thus, in D we add an a-edge from S to S_a. If we do this for all states S of D, we obtain the following:

> *For any $x \in \Sigma^*$, if D reaches state $S \in \mathcal{P}(Q)$ for input x, then and only then for each $q \in S$, N can reach state q for input x.*

It is not hard to prove this property formally, e.g., by induction.

Our DFA D must accept an input if and only if N can reach an accepting state for that input. Thus, D's accepting states comprise all sets of states of N that contain *at least one* of N's accepting states. I.e., S is an accepting state of D, if $S \cap A = \emptyset$, where A is N's set of accepting state.

To summarize: Given an NFA $N = (Q, \Sigma, \delta, q_0, A)$, we construct a DFA $D = (Q', \Sigma, \delta', q_0', A')$, where

- $Q' = \mathcal{P}(Q)$.
- For each $S \in \mathcal{P}(Q)$ and $a \in \Sigma$: $\delta'(S, a) = S_a^\lambda$ for S_a as defined in eq. (4.1).
- $q_0' = \{q_0\}^\lambda$.
- $A' = \{S \subseteq Q \mid S \cap A \neq \emptyset\}$.

The DFA D as defined above, may have unreachable state, i.e., states that cannot be reached from its starting state. In that case, it is not necessary to even construct those states. Thus, instead of creating all states of $\mathcal{P}(Q)$, we can perform the construction in a "lazy" manner using the following algorithm:

1. Construct the starting state $\{q_0\}^\lambda$ of D.
2. As long as D has a state S without an outgoing a-edge for some symbol $a \in \Sigma$:
   - Compute S_a^λ as described above (see eq. (4.1)).
   - If D does not yet contain state S_a^λ, add S_a^λ.
   - Add an edge with label a from S to S_a^λ.

Note that as a result of this algorithm, D may have a state $\emptyset$ (because the set S_a^λ defined in eq. (4.1) may be empty). In this case, all outgoing edges of that state are self-loops, which also follows immediately from eq. (4.1).

It is not hard to prove that the resulting DFA D recognizes the same language as N (but a full proof is omitted, here).

Theorem 4.19. *For each NFA N there exists a DFA D such that $L(N) = L(D)$.*

Example 4.20. Consider the NFA N in Figure 4.17 (a). We construct an equivalent DFA D.

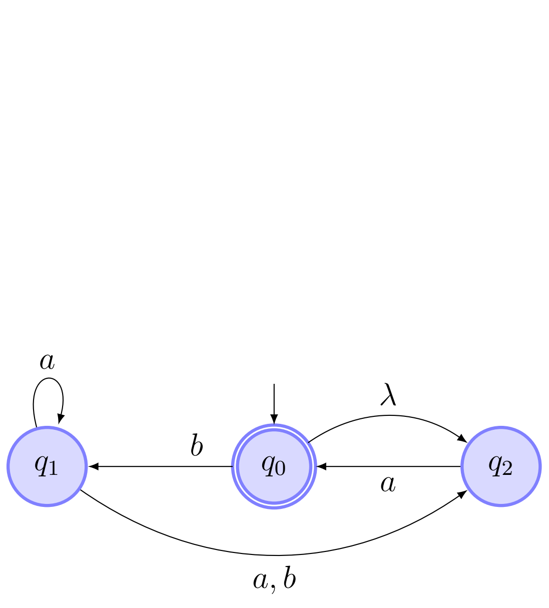
From the initial state of N, q_0, we can reach state q_2 through a λ-transition. Thus, we make $\{q_0\}^\lambda = \{q_0, q_2\}$ the initial state of D. Now we compute the states reachable from $\{q_0, q_2\}$ and the corresponding transitions:

First we construct the outgoing a-edge. In N q_0 has no outgoing a-edge, but q_2 has an outgoing a-edge to q_0. Thus, the states reachable from $\{q_0, q_2\}$ are again in $\{q_0\}^\lambda$. Hence, in D we create a self-loop labelled a from $\{q_0, q_2\}$ to itself.

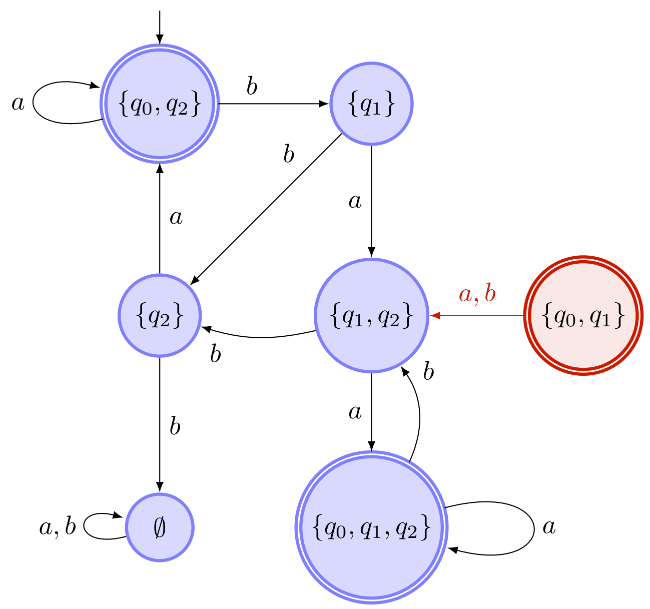
The b-edge from q_0 of N leads to q_1. State q_2 has no outgoing b-edge. Since q_1 also has no outgoing λ-transitions, we have $\{q_1\}^\lambda = \{q_1\}$. Thus, in D we create a new state $\{q_1\}$ and add a b-edge from $\{q_0, q_2\}$ to $\{q_1\}$.

Now D has a new state, $\{q_1\}$ that does not yet have any outgoing edges. We add an a-edge to a new state $\{q_1, q_2\}$ of D, because if N reads an a in state q_1, it can either follow the self loop and remain in q_1, or it can reach q_2. We also add a b-edge from $\{q_1\}$ to a new state $\{q_2\}$ of D, because q_2 is the only state N can reach if it reads a b in state q_1.

Now D has two states without any outgoing edges, $\{q_1, q_2\}$ and $\{q_2\}$. We consider state $\{q_2\}$. If N reads a in state q_2, it can reach q_0, and by following a λ-transition also q_2. Hence, we add an edge from q_2 to $\{q_0, q_2\}$. If N reads b in state q_2, it can reach no state, as q_2 has no outgoing b-edges. Thus, to D we add a new state $\emptyset$, as well as a b-edge from $\{q_2\}$ to $\emptyset$.



(a) NFA N



(b) DFA D

Figure 4.17: An NFA N and the equivalent DFA D obtained by the power set construction. The red state $\{q_0, q_1\}$ of C is not reachable, and thus does not need to be constructed.

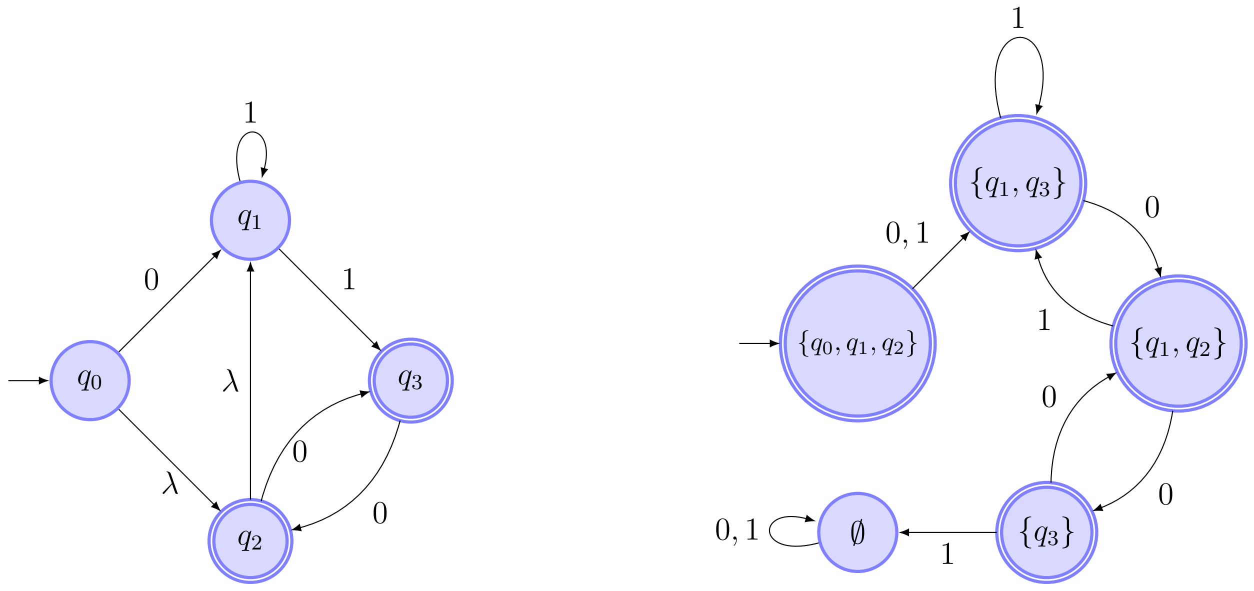
Now, we can add self-loops labelled a and b from state $\emptyset$ to itself. The reason is that D reaches state $\emptyset$ for an input x, for which N cannot reach any state. Hence, for input xa or xb N can still not reach any state. So we want D to also reach state $\emptyset$ for xa and xb.

Continuing this construction yields the blue states and transitions between them depicted in Figure 4.17 (b). At that point, we have added an outgoing a- and b-edge to each state of D.

This construction does not add the states $\{q_0\}$ and $\{q_0, q_1\}$, because they are not reachable. To illustrate this fact, state $\{q_0, q_1\}$ and its outgoing edges are shown in red. Note that it does not hurt adding that state and its outgoing edges, but it does not help, either. Another unreachable state that we could (but didn't) add is $\{q_0\}$.

Finally, we need to choose the accepting states of D: The accepting state of N is q_0, so all states S of D that contain q_0 must be accepting states of D. These are $\{q_0, q_2\}$ and $\{q_0, q_1, q_2\}$. ◀

Example 4.21. Consider the NFA in Figure 4.18 (a). The DFA resulting from the power set construction is shown in Figure 4.18 (b). For an explanation, we refer to the Wikipedia article "Powerset construction" (see https://en.wikipedia.org/wiki/Powerset_construction), which uses an equivalent NFA N as an example. ◀



(a) NFA N (b) DFA D

Figure 4.18: An NFA N and the equivalent DFA D. These automata are equivalent to the ones in an example in the Wikipedia article "Powerset construction".

4.3 Regular Languages and Their Properties

A language L is *regular*, if there is a regular expression r, such that $L(r) = L$. We have seen that for each regular language, we can construct an NFA (see Corollary 4.17) that recognizes it, and conversely that if N is an NFA, then the language is regular (see Theorem 4.18). Moreover, if D is a DFA, then obviously there exists and NFA that recognizes $L(D)$ (simply replace each state q of D with $\{q\}$, leaving existing edges intact). Hence, $L(D)$ is also regular. On the other hand, the power set construction allows us to create from each NFA a DFA that recognizes the same language (see Theorem 4.19). Thus, we obtain the following result:

Corollary 4.22. *Let L be a language. The following three statements are equivalent:*

- *L is regular.*
- *There exists an NFA that recognizes L.*
- *There exists a DFA that recognizes L.*

In particular, if we want to prove that a language is regular, we can either construct a regular expression, or an NFA, or a DFA.

4.3.1 Closure Properties of Regular Languages

A set S is *closed* under an operation, if applying that operation to elements in S always yields again an element in S. For example, the set of positive integers, $\mathbb{N}$, is closed under addition and multiplication, because the sum or the product of two positive integers is also a positive integer. But it is not closed under division or subtraction, because, for example, $1/2$ and $1-2$ are not positive integers, even though 1 and 2 are.

The set of regular languages is closed under all important operations, such as union, concatenation, intersection, Kleene closure, positive closure, and complement. The fact that we can represent regular languages using NFAs, DFAs, and regular expressions (according to Corollary 4.22), makes it easy to prove that.

Theorem 4.23. *Let L_1 and L_2 be regular languages over the same alphabet Σ. Then $L_1 \cup L_2$, $L_1 \cap L_2$, $\overline{L_1}$, $L_1 \circ L_2$, L_1^*, and L_1^+ are also regular.*

Proof. Since L_1 and L_2 are regular, there are regular expressions r_1 and r_2 such that $L(r_1) = L_1$ and $L(r_2) = L_2$. Each of $r_1 + r_2$, $r_1 \circ r_2$, r_1^*, and r_1^+ is a regular expression. Thus, it follows immediately from the definition of regular expressions and the languages they define that $L_1 \cup L_2$, $L_1 \circ L_2$, L_1^*, and L_1^+ are regular.

We will now show that $\overline{L_1}$ is regular. Since L_1 is regular, by Corollary 4.22 there is a DFA D that recognizes L_1. We construct a DFA D' by making each accepting state in D non-accepting, and each non-accepting state accepting. Let $x \in \Sigma^*$ be an input. If $x \in L$, then D reaches an accepting state for that input. Hence, D' reaches a non-accepting state for x, and does not accept x. Similarly, if $x \notin L$, then D reaches a non-accepting state for that input, while D' reaches an accepting state. It follows that D' recognizes $\overline{L}$, and thus $\overline{L}$ is regular by Corollary 4.22.

Finally, we will show that $L_1 \cap L_2$ is regular. By De Morgan's rules,

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}.$$

Since we have already proved that the set of regular languages is closed under union and complement, it follows that $L_1 \cap L_2$ is regular. □

4.3.2 Proving Nonregularity

Perhaps all languages are regular? This seems unlikely, because intuitively, some decision problems cannot be solved with a finite amount of memory. Consider for example, the language $L = \{0^n 1^n \mid n \in \mathbb{N}_0\}$. To determine that an input is in the language, a DFA would need to read the input, and count how many 0s it has read. But since there can be any number of 0s, this does not seem to be possible with a finite amount of memory. In particular, eventually the DFA would run out of states to keep track of the number of 0s read. This just gives us an intuition, and not a proof. The following example will show how to actually prove this.

Example 4.24. Prove that $L = \{0^n 1^n \mid n \in \mathbb{N}_0\}$ is not regular.

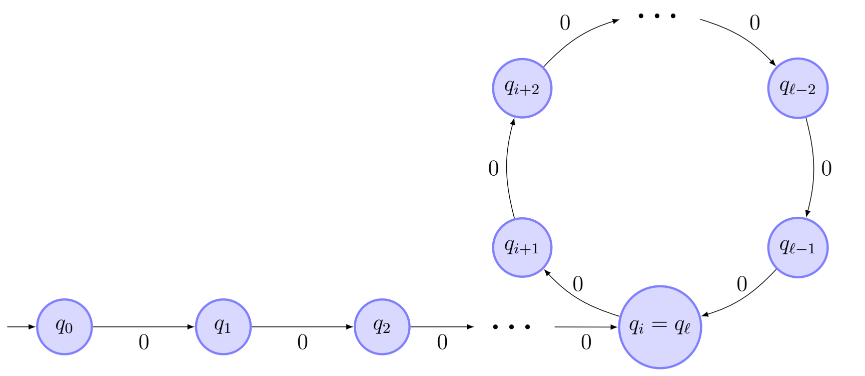
According to Corollary 4.22, it suffices to prove that there is no DFA that recognizes L. We will perform a proof by contradiction. To that end, suppose there is a DFA D that recognizes L.

Let k be the number of states of DFA D. Since the number of states is finite, k is some number in $\mathbb{N}_0$. Consider the input 0^k. For that input D performs k state transitions, and thus visits $k+1$ states, $q_0, q_1, \ldots, q_k$ (in this order). Since the number of states it visits is larger than the number of states it has, it must visit at least one state twice. I.e., there are states q_i and q_ℓ, $0 \leq i < \ell \leq k$, such that D visits state $q_i = q_\ell$ twice.

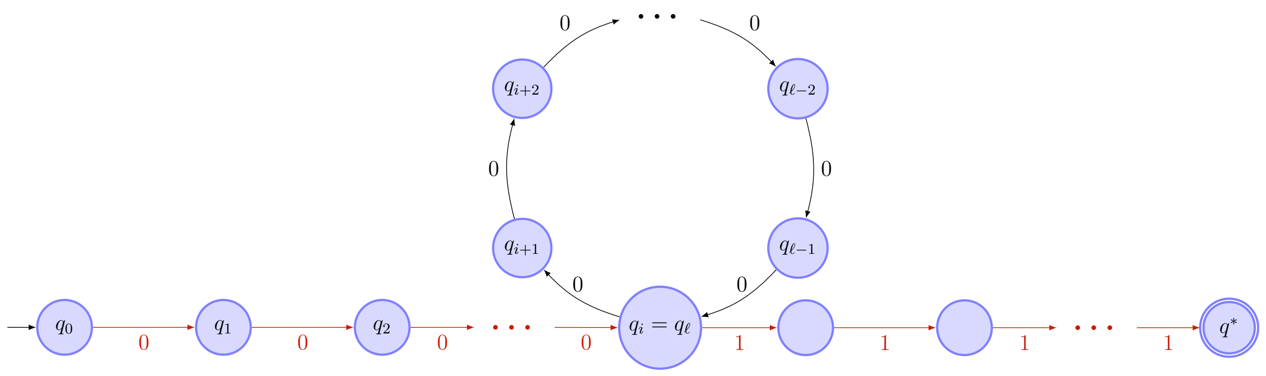
In other words, on its computation path from state q_i to q_ℓ DFA D closes a cycle (i.e, $(q_i, q_{i+1}, \ldots, q_\ell)$ forms a cycle of 0-edges). Note that D reaches state q_i after having read i 0s and again after having read a total of ℓ 0s. See Figure 4.19 (a) for an illustration of that fact.

Now consider the input $0^i 1^i$. Since that input is in L, D must accept it. Hence, if D is in state q_i and then reads exactly i 1s, it will reach an accepting state q^*. The corresponding path is illustrated in Figure 4.19 (b).

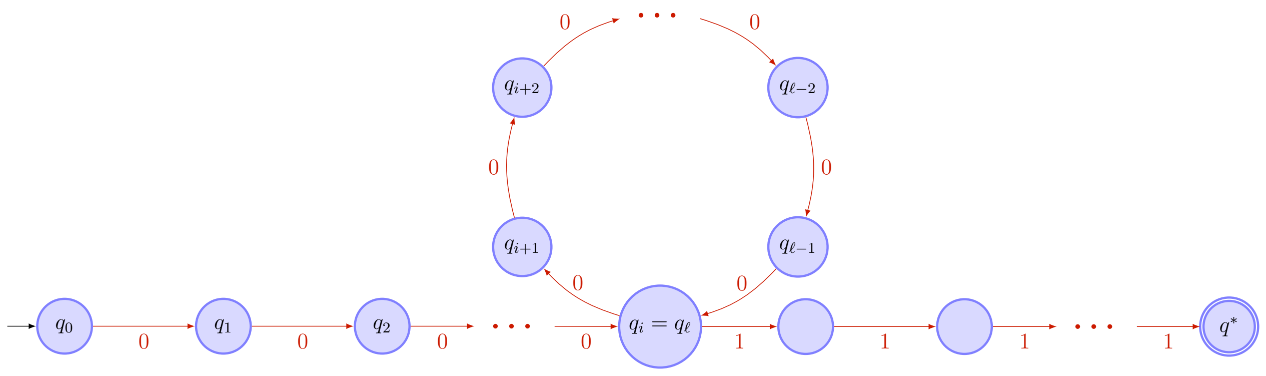
But now recall that D also reaches state q_i for the input 0^ℓ. Hence, for the input $0^\ell 1^i$, DFA D reaches state q_i after reading the first ℓ 0s, and then it reaches the accepting state q^* after continuing from there, reading i 1s—see Figure 4.19 (c). It follows that D also accepts $0^i 1^\ell$. But since $i < \ell$, this input is not in L. Thus, D accepts at least one input which is not in L. This contradicts our assumption that D recognizes L. ◀



(a) For input 0^k, where k is the number of states of the DFA, the same state is reached at least twice (after reading i and ℓ 0s, respectively).



(b) For the input 0^i1^i an accepting state q^* must be reached.



(c) For the input $0^\ell 1^i$ state q^* must also be reached.

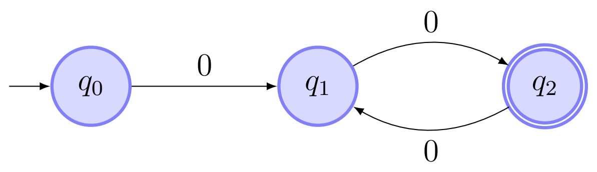
Figure 4.19: Proof that no DFA can recognize $\{0^n1^n \mid n \in \mathbb{N}_0\}$.

4.4 Exercises

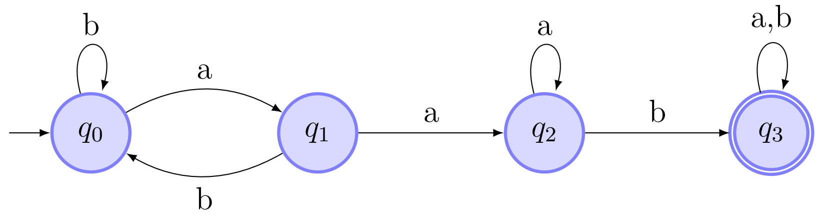
4.1 Determine the language that each of the following DFAs recognizes. Justify your answers.

DFAs

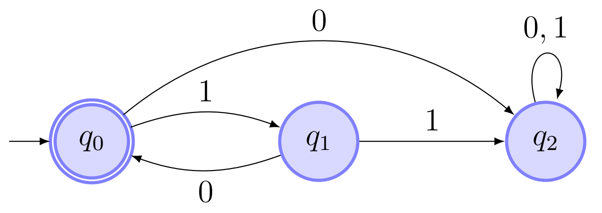
(a)



(b)



(c)



4.2 Construct DFAs recognizing the following languages over Σ. Justify your answers.

DFAs

(a) The language of all strings over $\Sigma = \{0, 1\}$ beginning with 00.

(b) The language of all strings over $\Sigma = \{0, 1\}$ ending with 10.

(c) The language of all strings over $\Sigma = \{0, 1\}$, whose length is a multiple of 3.

(d) The language of all strings over $\Sigma = \{0, 1, \ldots, 9\}$, which represent in decimal notation an integer that is divisible by 3. (Recall that an integer is divisible by 3 if and only if the sum of its digits in decimal notation is divisible by 3.)

(e) The language of all strings over $\Sigma = \{e, o\}$ which contain an even number of e's and an odd number of o's.

(f) The language of all strings over $\Sigma = \{0,1\}$ that do not contain any consecutive 0s.

(g) The language of all strings over $\Sigma = \{0,1,2\}$ that do not contain any two identical consecutive symbols.

(h) The language $\bigcup_{k=0}^{\infty} \left(\{a\}\{a,b\}^{2k} \cup \{b\}\{a,b\}^{2k+1} \right)$ over $\Sigma = \{a,b\}$.

(i) The language $\{w \in \{a,b\}^* \mid w \text{ does not contain the substring } bba\}$.

4.3 Let $M = \{Q, \Sigma, \delta, q_0, A\}$ be a DFA that recognizes a language $L = L(M)$ over $\Sigma = \{0,1\}$. For each of the following languages, first describe informally how to obtain a DFA recognizing that language. Then describe the new quintuple that specifies that DFA. DFAs

(a) $\overline{L}$

(b) $\{0\} \circ L$

(c) $L \circ \{1\}$ (challenging)

4.4 Let M_1 and M_2 be DFAs with input alphabets Σ_1 and Σ_2, respectively. For each of the following languages, describe how to construct a DFA M from M_1 and M_2, such that M recognizes that language. DFAs, closure properties

(a) $L(M_1) \circ L(M_2)$, assuming that Σ_1 and Σ_2 are disjoint.

(b) $L(M_1) \cup L(M_2)$, assuming that Σ_1 and Σ_2 are disjoint.

(c) For a challenge: $L(M_1) \cup L(M_2)$, assuming that $\Sigma_1 = \Sigma_2$.

4.5 For each of the following languages over $\{0,1\}$ depict an NFA recognizing the language, and give the corresponding quintuple. Try to use at most as many states as indicated. NFAs

(a) $L_1 = \{w \in \{0,1\}^* \mid w \text{ ends in } 00\}$ (3 states)

(b) $L_2 = \{w \in \{0,1\}^* \mid w \text{ contains the substring } 0101\}$ (5 states)

(c) $L_3 = \{0\}^*$ (1 state)

4.6 Let N be an NFA with input alphabet $\Sigma = \{a,b\}$. Show how to construct an NFA for the language $L(N) \cup \{aaaa\}$. NFAs

4.7 Let $N = (Q, \Sigma, \delta, q_0, A)$ be an NFA, and let $N' = (Q, \Sigma, \delta, q_0, Q \setminus A)$. Is it true that $L(N') = \overline{L(N)}$? Justify your answer. NFAs

4.8 Prove that every finite language can be recognized by some NFA.

NFAs

4.9 Let L be any regular language over the alphabet Σ. Prove that the following languages are also regular.

NFAs, DFAs, regular languages

(a) L^R, which contains all strings of L in reverse. Formally, $L^R = \{x_1 \dots x_k \in \Sigma^k \mid x_k \dots x_1 \in L\}$.

(b) L^{even}, which contains for each string $w \in L$ the string comprising every symbol in an even position of w. For example, if $w = w_1 w_2 w_3 w_4 \dots \in L$, then $w_2 w_4 \dots$ is in L^{even}.

(c) $L^{postfix-2}$, which contains for each string w in L the postfix of w of length $|w| - 2$, i.e., the string with the first two symbols of w removed.

(d) $L^{prefix-2}$, which contains for each string w in L the prefix of w of length $|w| - 2$, i.e., the string with the last two symbols of w removed.

4.10 Let $N = (Q, \Sigma, \delta, q_0, A)$ be an NFA, and let q' be a new state, i.e., $q' \notin Q$. Further, let $Q' = Q \cup \{q'\}$ and $N' = (Q', \Sigma, \delta', q_0, A)$, where $\delta' : (Q' \times (\Sigma \cup \{\lambda\}) \to \mathcal{P}(Q')$ is defined as follows: For each $q \in Q'$ and $a \in \Sigma \cup \{\lambda\}$:

- if $q = q_0$ and $a = \lambda$, then $\delta'(q, a) = \delta(q, a) \cup \{q'\}$, and
- otherwise $\delta'(q, a) = \delta(q, a)$.

Depict N', and decide if $L(N') = L(N) \cup \{\lambda\}$. Justify your answer.

4.11 Two automata are *equivalent* if they recognize the same language.

DFAs, NFAs

(a) Prove that for any NFA N there exists an equivalent NFA N' that has at most one accepting state.

(b) Is it true that for any DFA D there exists an equivalent DFA D' that has at most one accepting state? Justify your answer.

4.12 Complete the proof of Theorem 4.16. I.e., show that the NFAs N_1 and N_2 constructed there recognize the languages $L(N^+)$ and $L(N^*)$, respectively.

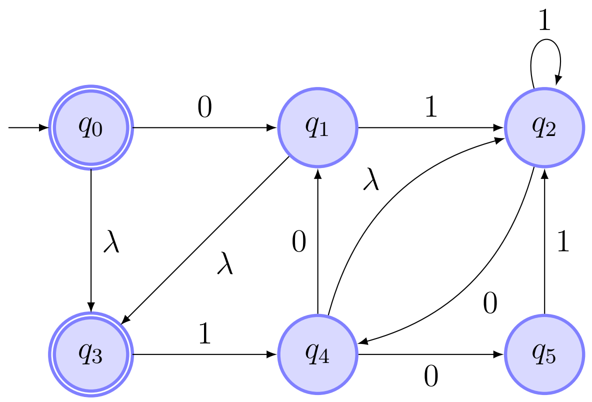
NFAs, closure properties, proofs

4.13 Convert each of the NFAs constructed in Exercise 4.5 to a DFA.

power set construction

4.14 Convert the following NFA to a DFA:

power set construction



4.15 Prove that each of the following languages over $\Sigma = \{a, b, c\}$ is not regular. (If $x = x_1 \dots x_k \in \Sigma^k$ for some $k \in \mathbb{N}_0$, then $x^R = x_k x_{k-1} \dots x_1$ is the reversal of x.)

nonregularity

(a) $\{a^{2n}b^n \mid n \in \mathbb{N}_0\}$

(b) $\{x \in \Sigma^* \mid x$ contains equally many a's as b's$\}$

(c) $\left\{x \circ x^R \mid x \in \Sigma^*\right\}$

(d) $\left\{x \in \Sigma^* \mid x = x^R\right\}$

(e) $\left\{a^{n^2} \mid n \in \mathbb{N}_0\right\}$

4.16 Let $\Sigma = \{\#\}$ and $L = \left\{\#^k \mid k \text{ is a multiple of 2 or 3}\right\}$.

impossibility proof, accepting states, NFA, DFA

(a) Give a DFA that recognizes L.

(b) Show that any DFA that recognizes L must have at least two accepting states.

(c) Give an NFA that has exactly one accepting state and recognizes L.

(d) Prove that any regular language has an NFA with exactly one accepting state.

4.5 Selected Solutions

Section 4.4

(a) The DFA recognizes the set of non-empty strings that contain an even number of 0's. State q_0 is only reached for the empty input. State q_1 is reached for the input 0, then state q_2 for the input 00, state q_1 for 000, state q_2 for 0000, and so on. Specifically, state q_1 is reached for all inputs of odd length, and q_2 for all non-empty inputs of even length. (This can be proved by a simple induction.) As q_2 is the only accepting state, the claim follows.

(b) The DFA recognizes the language of all strings that contain the substring aab. Informal explanation: State q_0 is reached for the empty string, and any string that does not contain aab but ends with b. State q_1 is reached for strings that do not contain aab and end with a but not aa. State q_2 is reached for inputs that do not contain aab and end with aa. Finally, state q_3 is reached for the first time when aab has been observed, and after that the DFA remains in that state.

Exercise 4.3

(a) See the proof of Theorem 4.23.

(b) We create a new initial state q_0', as well as a trap state q_t. We add a 1-edge from q_0' to q_t and a 0-edge from q_0' to the initial state of M. Thus, the new DFA is $M' = (Q', \Sigma, \delta', q_0', A)$, where $Q' = Q \cup \{q_0', q_t\}$, and $\delta'(q, a) = \delta(q, a)$ for all $q \in Q$ and $a \in \{0, 1\}$, and $\delta'(q_0', 0) = q_t$, $\delta'(q_0', 1) = q_0$, and $\delta'(q_t, 0) = \delta'(q_t, 1) = q_t$.

We now argue that this DFA M' recognizes $\{0\} \circ L$: Consider an input $0x$, where $x \in L$. Then M accepts x. Moreover, M' traverses to q_0, the starting state of M by processing the leading 0. After that, M' behaves like M, and so for x it reaches the accepting state of M. Thus, M' accepts $0x$.

Now assume M' accepts some string y. Then M' must first go along the 0-edge from q_0' to q_0, and then for the remaining input reach an accepting state of M. I.e., $y = 0y'$, where y' is accepted by M. Thus, $y \in \{0\} \circ L$.

Exercise 4.8 Consider a single string $w = w_1 \dots w_k$. There is a simple NFA that can accept only this word: The NFA comprises the states $q_0, \dots, q_k$, where q_0 is the initial state, and for each $i \in \{1, \dots, k\}$ there is a w_i-transition from q_{i-1} to q_i. (I.e., the NFA is a path from q_0 to q_k, where the i-th edge is labeled w_i.) It is obvious that this NFA recognizes only the language $\{w\}$.

Now consider a finite language $L = \{x_1, \dots, x_\ell\}$, where each x_i is a string. For each string x_i we can construct an NFA N_i as described above. Let $q_{i,0}$ be the initial state of N_i. Now, we create a new initial state q and add a λ-transition from q to each state $q_{i,0}$. Then the resulting NFA recognizes exactly the language $L(N_1) \cup \dots \cup L(N_\ell) = \{x_1, \dots, x_\ell\}$.

Exercise 4.9 (b) If L is regular, then there exists a DFA D that recognizes L. Let $D = (Q, \Sigma, \delta, q_0, A)$. We will construct an NFA $N = (Q', \Sigma, \delta', q'_0, A')$ that recognizes L^{even}.

The idea is as follows: Consider an input $x_2 x_4 \dots x_{2k} \in \Sigma^*$. Our NFA has to "guess" (or randomly choose) the symbols at odd positions, $x_1, x_3, \dots, x_{2k-1}$, and possibly x_{2k+1}. At the same time it must check if D would accept $x_1 x_2 \dots x_{2k}$ or $x_1 x_2 \dots x_{2k+1}$. To do that, N memorizes the following for each random computation:

- The state that D would have reached for the symbols processed so far (including the randomly chosen symbols); and
- If an even or odd number of symbols have been read.

To memorize this information, N needs for each state $q \in Q$ two states $(q, 0), (q, 1) \in Q'$. The first component of each pair indicates which state D would have reached so far, and the second component indicates the parity of the number of symbols D would have read. Consider the following predicate:

> $P(k)$: Let $x = x_2 x_4 \dots x_{2k} \in \Sigma^*$ and let $q \in Q$. If NFA N can reach state $(q, 0)$ for input x, then and only then there exist $x_1, x_3, \dots, x_{2k-1} \in \Sigma^*$ such that for input $x_1 x_2 \dots x_{2k}$ DFA D reaches state q. And if NFA N can reach state $(q, 1)$ for input x, then and only then there exist $x_1, x_3, \dots, x_{2k+1} \in \Sigma^*$ such that for input $x_1 x_2 \dots x_{2k+1}$ DFA D reaches state q.

We will construct the starting state of N and the transition function in such a way that $P(k)$ is true for every non-negative integer k. The initial state of N is $(q_0, 0)$.

For each edge labelled a from a state q to a state q' in D, N contains a λ-edge form $(q, 0)$ to $(q', 1)$, and an a-edge from state $(q, 1)$ to $(q', 0)$. Formally, the transition function of N, $\delta' : Q' \times (\Sigma \cup \{\lambda\}) \to \mathcal{P}(Q')$ can be defined as follows: For each $q \in Q$ and each $a \in \Sigma$

- $\delta'((q, 0), \lambda) = \{(q', 1) \mid \exists b \in \Sigma : \delta(q, b) = q'\}$.
  (For each a-edge (q, q') in D create a λ-edge from $(q, 0)$ to $(q', 1)$ in N.)

- $\delta'((q,0),a) = \emptyset$.
  (N contains no a-edges leaving node $(q,0)$, because they each such edge is replaced with λ-edges to a state $(\cdot,1)$.)
- $\delta'((q,1),\lambda) = \emptyset$.
  (N' contains no λ-edges leaving a node $(q,1)$.)
- $\delta'((q,1),a)) = \{\delta(q,a)\}$.
  (For each a-edge from q to q' in D, add an a-edge from $(q,1)$ to $(q',0)$.)

Recall that the initial state of N is $(q,0)$. The accepting states of N are all states (q,i), $i \in \{0,1\}$, where q is an accepting state of D. It follows immediately from predicate $P(k)$, $k \in \mathbb{N}_0$, that N accepts the language L^{even}: If $x = x_2x_4 \dots x_{2k} \in L^{even}$, then and only then N has a valid path from $(q_0,0)$ to an accepting state (q,i), $i \in \{0,1\}$, of N. By $P(k)$, then and only then there exist $x_1, x_3, \dots, x_{2k+1}$, such that D reaches state q either for the input $x_1 \dots x_{2k}$ (if $i = 0$) or for the input $x_1 \dots x_{2k+1}$ ($i = 1$). Thus, if and only if N can accept x, D accepts $x_1x_2 \dots x_{2k}$ or $x_1 \dots x_{2k+1}$ for some symbols $x_1, x_3, \dots, x_{2k+1}$.

Thus, it remains to prove that $P(k)$ is true for all $k \in \mathbb{N}_0$.

Base Case: The base case is for $k = 0$. Thus, consider the empty string, $x = \lambda$, as an input to N, and consider any state $q \in Q$. Suppose D reaches state q for the input λ, or there exists a symbol $x_1 \in \Sigma$ such that D reaches state q for the input x_1. In the first case, $q = q_0$, and clearly N can reach state $(q,0)$ for input λ. In the second case, N contains an x_1-edge from q_0 to q, and thus N contains a λ transition between $(q_0,0)$ and $(q,1)$. Hence, N can reach $(q,1)$ for the input λ.

Similarly, suppose N can reach a state $(q,0)$ from its initial state, $(q_0,0)$, for the input λ. Then N can use only λ-transitions. By construction, $(q,0)$ has no ingoing λ-edges. Thus, $(q,0) = (q_0,0)$, and D reaches q_0 for input λ as well. Now suppose N can reach a state $(q,1)$ from its initial state, $(q_0,0)$, using only λ-transitions. Since any state of the form $(\cdot,1)$ has no outgoing λ-transition, there is no λ-transition from $(q_0,1)$ to $(q,1)$. Hence, there must be a λ-transition from $(q_0,0)$ to $(q,1)$. Hence, by construction, D has an x_1-edge from q_0 to q for some symbol $x_1 \in \Sigma$. Thus, D can reach state q for the input x_1.

This proves $P(0)$.

Inductive Step: Now let $k > 0$, and assume that $P(\ell)$ is true for each $\ell < k$. Let $x = x_2 \dots x_{2k}$ and $q \in Q$.

First assume there exists $x_1, x_3, \dots, x_{2k-1} \in \Sigma$ such that D reaches state q for input $x_1 \dots x_{2k}$, and let q' be the preceding state on the path from q_0 to q. Thus, D reaches q' for the input $x_1 \dots x_{2k-1}$. Then D contains an edge labelled x_{2k} from q' to q. By the inductive hypothesis, N can reach $(q',1)$ for the input $x_2x_4 \dots x_{2k-2}$. Moreover, it contains an edge labelled x_{2k} from $(q',1)$ to $(q,0)$. Hence, N can reach $(q,0)$ for the input $x_2x_4 \dots x_{2k-2}x_{2k}$.

Now assume there exist $x_1, x_3, \ldots, x_{2k-1}, x_{2k+1} \in \Sigma$ such that D reaches state q for input $x_1 \ldots x_{2k} x_{2k+1}$. Let p' be the preceding state on the path from q_0 to q. Thus, D reaches q' for the input $x_1 \ldots x_{2k}$. As we have just proved above, N can reach state $(q', 0)$ for the input $x_2 \ldots x_{2k}$. Moreover, D contains an x_{2k+1}-edge from q' to q, and thus by construction N contains a λ-edge from $(q', 0)$ to $(q, 1)$. Thus, N can reach $(q, 1)$ for the input $x_2 \ldots x_{2k}$.

Now suppose that N can reach a state $(q, 0)$ for the input $x_2 x_4 \ldots x_{2k}$. By construction all incoming edges of $(q, 0)$ come from a state $(q', 1)$ (i.e., the second component is flipped). Then N has an x_{2k}-edge from $(q', 1)$ to $(q, 0)$ (because all edges leaving a state $(\cdot, 1)$ are labelled with a symbol, and not with λ). Thus, D has an x_{2k}-edge from q' to q. By the inductive hypothesis, there exist $x_1, x_3, \ldots, x_{2k-1}$ such that D reaches state q' for the input $x_1 \ldots x_{2k-1}$. Thus, given the x_{2k}-edge from q' to q, means that D reaches state q for the input $x_1 \ldots x_{2k}$.

Finally, suppose that N can reach a state $(q, 1)$ for the input $x_2 x_4 \ldots x_{2k}$. By construction, each incoming edge of $(q, 1)$ is a λ-edge coming from some vertex $(q', 0)$. Thus, N can reach such a state $(q', 0)$ for the same input $x_2 x_4 \ldots x_{2k}$, because it must visit such a state before reaching $(q, 1)$. By construction, D has an a-edge from q' to q for some symbol a. Moreover, by the inductive hypothesis, there exist $x_1, x_3, \ldots, x_{2k-3}$ such that D reaches state q' for the input $x_1 x_2 \ldots x_{2k-2}$. Thus, using $x_{2k-1} = a$, it follows that D reaches state q for the input $x_1 x_2 \ldots x_{2k-1}$.

This completes the inductive step, and proves that $P(k)$ is true for all $k \in \mathbb{N}_0$. $\square$

(a) Given an NFA N, we can create a new state q_A, and add from each accepting state of N a λ-transition to q_A. Moreover, we make each accepting state of N non-accepting. Let N' be the resulting NFA.

Clearly, N' has exactly one accepting state, namely q_A.

Suppose N can accept some string w. Then clearly, N' can accept the same string, as after processing w it can end in a state that is accepting in N, and then take the λ-transition to q_A. Similarly, if N' can accept some string w, then it must be able to reach an accepting state of N by processing w, and then take a λ-transition to q_A, so w can be accepted by N.

(b) The statement is not true. Specifically, consider the language $L = \{\lambda, a\}$ over the alphabet $\Sigma = \{a\}$. Assume there is a DFA for L that has only one accepting state. Then the initial state, q_0 must be the accepting state, because the DFA accepts λ. Moreover, as there is no other accepting state and the DFA must accept the input a, there must be an a-transition from q_0 to q_0. But then the DFA also accepts aa (and in fact any string in $\{a\}^*$). This contradicts the assumption that the DFA recognizes L.