Formal Languages and Automata Theory

D. Goswami and K. V. Krishna

November 5, 2010

Contents

Chapter 1

Mathematical Preliminaries

Chapter 2

Formal Languages

A language can be seen as a system suitable for expression of certain ideas, facts and concepts. For formalizing the notion of a language one must cover all the varieties of languages such as natural (human) languages and programming languages. Let us look at some common features across the languages. One may broadly see that a language is a collection of sentences; a sentence is a sequence of words; and a word is a combination of syllables. If one considers a language that has a script, then it can be observed that a word is a sequence of symbols of its underlying alphabet. It is observed that a formal learning of a language has the following three steps.

1. Learning its alphabet - the symbols that are used in the language.
2. Its words - as various sequences of symbols of its alphabet.
3. Formation of sentences - sequence of various words that follow certain rules of the language.

In this learning, step 3 is the most difficult part. Let us postpone to discuss construction of sentences and concentrate on steps 1 and 2. For the time being instead of completely ignoring about sentences one may look at the common features of a word and a sentence to agree upon both are just sequence of some symbols of the underlying alphabet. For example, the English sentence

```
"The English articles - a, an and the - are
      categorized into two types:  indefinite and definite."
```

may be treated as a sequence of symbols from the Roman alphabet along with enough punctuation marks such as comma, full-stop, colon and further one more special symbol, namely *blank-space* which is used to separate two words. Thus, abstractly, a sentence or a word may be interchangeably used

for a sequence of symbols from an alphabet. With this discussion we start with the basic definitions of alphabets and strings and then we introduce the notion of language formally.

Further, in this chapter, we introduce some of the operations on languages and discuss algebraic properties of languages with respect to those operations. We end the chapter with an introduction to finite representation of languages via regular expressions.

2.1 Strings

We formally define an *alphabet* as a non-empty finite set. We normally use the symbols $a, b, c, \ldots$ with or without subscripts or $0, 1, 2, \ldots$, etc. for the elements of an alphabet.

A *string* over an alphabet Σ is a finite sequence of symbols of Σ. Although one writes a sequence as $(a_1, a_2, \ldots, a_n)$, in the present context, we prefer to write it as $a_1a_2\cdots a_n$, i.e. by juxtaposing the symbols in that order. Thus, a string is also known as a *word* or a *sentence*. Normally, we use lower case letters towards the end of English alphabet, namely z, y, x, w, etc., to denote strings.

Example 2.1.1. Let $\Sigma = \{a, b\}$ be an alphabet; then $aa, ab, bba, baaba, \ldots$ are some examples of strings over Σ.

Since the empty sequence is a finite sequence, it is also a string. Which is ( ) in earlier notation; but with the notation adapted for the present context we require a special symbol. We use ε, to denote the empty string.

The set of all strings over an alphabet Σ is denoted by Σ^*. For example, if $\Sigma = \{0, 1\}$, then

$$\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, \ldots\}.$$

Although the set Σ^* is infinite, it is a countable set. In fact, Σ^* is countably infinite for any alphabet Σ. In order to understand some such fundamental facts we introduce some string operations, which in turn are useful to manipulate and generate strings.

One of the most fundamental operations used for string manipulation is concatenation. Let $x = a_1a_2\cdots a_n$ and $y = b_1b_2\cdots b_m$ be two strings. The *concatenation* of the pair x, y denoted by xy is the string

$$a_1a_2\cdots a_nb_1b_2\cdots b_m.$$

Clearly, the binary operation concatenation on Σ^* is associative, i.e., for all $x, y, z \in \Sigma^*$,

$$x(yz) = (xy)z.$$

Thus, $x(yz)$ may simply be written as xyz. Also, since ε is the empty string, it satisfies the property

$$\varepsilon x = x\varepsilon = x$$

for any sting $x \in \Sigma^*$. Hence, Σ^* is a monoid with respect to concatenation. The operation concatenation is not commutative on Σ^*.

For a string x and an integer $n \geq 0$, we write

$$x^{n+1} = x^n x \text{ with the base condition } x^0 = \varepsilon.$$

That is, x^n is obtained by concatenating n copies of x. Also, whenever $n = 0$, the string $x_1 \cdots x_n$ represents the empty string ε.

Let x be a string over an alphabet Σ. For $a \in \Sigma$, the number of occurrences of a in x shall be denoted by $|x|_a$. The *length* of a string x denoted by $|x|$ is defined as

$$|x| = \sum_{a \in \Sigma} |x|_a.$$

Essentially, the length of a string is obtained by counting the number of symbols in the string. For example, $|aab| = 3$, $|a| = 1$. Note that $|\varepsilon| = 0$.

If we denote A_n to be the set of all strings of length n over Σ, then one can easily ascertain that

$$\Sigma^* = \bigcup_{n \geq 0} A_n.$$

And hence, being A_n a finite set, Σ^* is a countably infinite set.

We say that x is a *substring* of y if x occurs in y, that is $y = uxv$ for some strings u and v. The substring x is said to be a *prefix* of y if $u = \varepsilon$. Similarly, x is a *suffix* of y if $v = \varepsilon$.

Generalizing the notation used for number of occurrences of symbol a in a string x, we adopt the notation $|y|_x$ as the number of occurrences of a string x in y.

2.2 Languages

We have got acquainted with the formal notion of strings that are basic elements of a language. In order to define the notion of a language in a broad spectrum, it is felt that it can be any collection of strings over an alphabet.

Thus we define a *language* over an alphabet Σ as a subset of Σ^*.

Example 2.2.1.

1. The emptyset $\varnothing$ is a language over any alphabet. Similarly, $\{\varepsilon\}$ is also a language over any alphabet.

2. The set of all strings over $\{0, 1\}$ that start with 0.

3. The set of all strings over $\{a, b, c\}$ having ac as a substring.

Remark 2.2.2. Note that $\varnothing \neq \{\varepsilon\}$, because the language $\varnothing$ does not contain any string but $\{\varepsilon\}$ contains a string, namely ε. Also it is evident that $|\varnothing| = 0$; whereas, $|\{\varepsilon\}| = 1$.

Since languages are sets, we can apply various well known set operations such as union, intersection, complement, difference on languages. The notion of concatenation of strings can be extended to languages as follows.

The concatenation of a pair of languages L_1, L_2 is

$$L_1L_2 = \{xy \mid x \in L_1 \ \wedge \ y \in L_2\}.$$

Example 2.2.3.

1. If $L_1 = \{0, 1, 01\}$ and $L_2 = \{1, 00\}$, then
   $L_1L_2 = \{01, 11, 011, 000, 100, 0100\}$.

2. For $L_1 = \{b, ba, bab\}$ and $L_2 = \{\varepsilon, b, bb, abb\}$, we have
   $L_1L_2 = \{b, ba, bb, bab, bbb, babb, baabb, babbb, bababb\}$.

Remark 2.2.4.

1. Since concatenation of strings is associative, so is the concatenation of languages. That is, for all languages L_1, L_2 and L_3,

   $$(L_1L_2)L_3 = L_1(L_2L_3).$$

   Hence, $(L_1L_2)L_3$ may simply be written as $L_1L_2L_3$.

2. The number of strings in L_1L_2 is always less than or equal to the product of individual numbers, i.e.

   $$|L_1L_2| \leq |L_1||L_2|.$$

3. $L_1 \subseteq L_1L_2$ if and only if $\varepsilon \in L_2$.

Proof. The "if part" is straightforward; for instance, if $\varepsilon \in L_2$, then for any $x \in L_1$, we have $x = x\varepsilon \in L_1L_2$. On the other hand, suppose $\varepsilon \notin L_2$. Now, note that a string $x \in L_1$ of shortest length in L_1 cannot be in L_1L_2. This is because, if $x = yz$ for some $y \in L_1$ and a nonempty string $z \in L_2$, then $|y| < |x|$. A contradiction to our assumption that x is of shortest length in L_1. Hence $L_1 \not\subseteq L_1L_2$. □

4. Similarly, $\varepsilon \in L_1$ if and only if $L_2 \subseteq L_1L_2$.

We write L^n to denote the language which is obtained by concatenating n copies of L. More formally,
$L^0 = \{\varepsilon\}$ and
$L^n = L^{n-1}L$, for $n \geq 1$.

In the context of formal languages, another important operation is Kleene star. *Kleene star* or *Kleene closure* of a language L, denoted by L^*, is defined as

$$L^* = \bigcup_{n \geq 0} L^n.$$

Example 2.2.5.

1. Kleene star of the language $\{01\}$ is
   $\{\varepsilon, 01, 0101, 010101, \ldots\} = \{(01)^n \mid n \geq 0\}$.

2. If $L = \{0, 10\}$, then $L^* = \{\varepsilon, 0, 10, 00, 010, 100, 1010, 000, \ldots\}$

Since an arbitrary string in L^n is of the form $x_1x_2 \cdots x_n$, for $x_i \in L$ and $L^* = \bigcup_{n \geq 0} L^n$, one can easily observe that

$$L^* = \{x_1x_2 \cdots x_n \mid n \geq 0 \text{ and } x_i \in L, \text{ for } 1 \leq i \leq n\}$$

Thus, a typical string in L^* is a concatenation of finitely many strings of L.

Remark 2.2.6. Note that, the Kleene star of the language $L = \{0, 1\}$ over the alphabet $\Sigma = \{0, 1\}$ is

$$\begin{aligned} L^* &= L^0 \cup L \cup L^2 \cup \cdots \\ &= \{\varepsilon\} \cup \{0, 1\} \cup \{00, 01, 10, 11\} \cup \cdots \\ &= \{\varepsilon, 0, 1, 00, 01, 10, 11, \cdots\} \\ &= \text{the set of all strings over } \Sigma. \end{aligned}$$

Thus, the earlier introduced notation Σ^* is consistent with the notation of Kleene star by considering Σ as a language over Σ.

The *positive closure* of a language L is denoted by L^+ is defined as

$$L^+ = \bigcup_{n \geq 1} L^n.$$

Thus, $L^* = L^+ \cup \{\varepsilon\}$.

We often can easily describe various formal languages in English by stating the property that is to be satisfied by the strings in the respective languages. It is not only for elegant representation but also to understand the properties of languages better, describing the languages in set builder form is desired.

Consider the set of all strings over $\{0, 1\}$ that start with 0. Note that each such string can be seen as $0x$ for some $x \in \{0, 1\}^*$. Thus the language can be represented by

$$\{0x \mid x \in \{0, 1\}^*\}.$$

Examples

1. The set of all strings over $\{a, b, c\}$ that have ac as substring can be written as

   $$\{xacy \mid x, y \in \{a, b, c\}^*\}.$$

   This can also be written as

   $$\{x \in \{a, b, c\}^* \mid |x|_{ac} \geq 1\},$$

   stating that the set of all strings over $\{a, b, c\}$ in which the number of occurrences of substring ac is at least 1.

2. The set of all strings over some alphabet Σ with even number of a's is

   $$\{x \in \Sigma^* \mid |x|_a = 2n, \text{ for some } n \in \mathbb{N}\}.$$

   Equivalently,

   $$\{x \in \Sigma^* \mid |x|_a \equiv 0 \mod 2\}.$$

3. The set of all strings over some alphabet Σ with equal number of a's and b's can be written as

   $$\{x \in \Sigma^* \mid |x|_a = |x|_b\}.$$

4. The set of all palindromes over an alphabet Σ can be written as

   $$\{x \in \Sigma^* \mid x = x^R\},$$

   where x^R is the string obtained by reversing x.

5. The set of all strings over some alphabet Σ that have an a in the 5th position from the right can be written as

$$\{xay \mid x, y \in \Sigma^* \text{ and } |y| = 4\}.$$

6. The set of all strings over some alphabet Σ with no consecutive a's can be written as

$$\{x \in \Sigma^* \mid |x|_{aa} = 0\}.$$

7. The set of all strings over $\{a, b\}$ in which every occurrence of b is not before an occurrence of a can be written as

$$\{a^m b^n \mid m, n \geq 0\}.$$

   Note that, this is the set of all strings over $\{a, b\}$ which do not contain ba as a substring.

2.3 Properties

The usual set theoretic properties with respect to union, intersection, complement, difference, etc. hold even in the context of languages. Now we observe certain properties of languages with respect to the newly introduced operations concatenation, Kleene closure, and positive closure. In what follows, L, L_1, L_2, L_3 and L_4 are languages.

P1 Recall that concatenation of languages is associative.

P2 Since concatenation of strings is not commutative, we have $L_1L_2 \neq L_2L_1$, in general.

P3 $L\{\varepsilon\} = \{\varepsilon\}L = L$.

P4 $L\varnothing = \varnothing L = \varnothing$.

Proof. Let $x \in L\varnothing$; then $x = x_1x_2$ for some $x_1 \in L$ and $x_2 \in \varnothing$. But $\varnothing$ being emptyset cannot hold any element. Hence there cannot be any element $x \in L\varnothing$ so that $L\varnothing = \varnothing$. Similarly, $\varnothing L = \varnothing$ as well. □

P5 Distributive Properties:

1. $(L_1 \cup L_2)L_3 = L_1L_3 \cup L_2L_3$.

Proof. Suppose $x \in (L_1 \cup L_2)L_3$

$$\begin{aligned}
&\implies x = x_1x_2, \text{ for some } x_1 \in L_1 \cup L_2), \text{ and some } x_2 \in L_3 \\
&\implies x = x_1x_2, \text{ for some } x_1 \in L_1 \text{ or } x_1 \in L_2, \text{ and } x_2 \in L_3 \\
&\implies x = x_1x_2, \text{ for some } x_1 \in L_1 \text{ and } x_2 \in L_3, \\
&\qquad\qquad\qquad\qquad \text{or } x_1 \in L_2 \text{ and } x_2 \in L_3 \\
&\implies x \in L_1L_3 \text{ or } x \in L_2L_3 \\
&\implies x \in L_1L_3 \cup L_2L_3.
\end{aligned}$$

Conversely, suppose $x \in L_1L_3 \cup L_2L_3 \implies x \in L_1L_3$ or $x \in L_2L_3$. Without loos of generality, assume $x \notin L_1L_3$. Then $x \in L_2L_3$.

$$\begin{aligned}
&\implies x = x_3x_4, \text{ for some } x_3 \in L_2 \text{ and } x_4 \in L_3 \\
&\implies x = x_3x_4, \text{ for some } x_3 \in L_1 \cup L_2, \text{ and some } x_4 \in L_3 \\
&\implies x \in (L_1 \cup L_2)L_3.
\end{aligned}$$

Hence, $(L_1 \cup L_2)L_3 = L_1L_3 \cup L_2L_3$. □

2. $L_1(L_2 \cup L_3) = L_1L_2 \cup L_1L_3$.

   Proof. Similar to the above. □

From these properties it is clear that the concatenation is distributive over finite unions. Moreover, we can observe that concatenation is also distributive over countably infinite unions. That is,

$$\begin{aligned}
L\left(\bigcup_{i\geq 1} L_i\right) &= \bigcup_{i\geq 1} LL_i \quad \text{and} \\
\left(\bigcup_{i\geq 1} L_i\right) L &= \bigcup_{i\geq 1} L_iL
\end{aligned}$$

P6 If $L_1 \subseteq L_2$ and $L_3 \subseteq L_4$, then $L_1L_3 \subseteq L_2L_4$.

P7 $\varnothing^* = \{\varepsilon\}$.

P8 $\{\varepsilon\}^* = \{\varepsilon\}$.

P9 If $\varepsilon \in L$, then $L^* = L^+$.

P10 $L^*L = LL^* = L^+$.

Proof. Suppose $x \in L^*L$. Then $x = yz$ for some $y \in L^*$ and $z \in L$. But $y \in L^*$ implies $y = y_1 \cdots y_n$ with $y_i \in L$ for all i. Hence,

$$x = yz = (y_1 \cdots y_n)z = y_1(y_2 \cdots y_n z) \in LL^*.$$

Converse is similar. Hence, $L^*L = LL^*$.

Further, when $x \in L^*L$, as above, we have $x = y_1 \cdots y_n z$ is clearly in L^+. On the other hand, $x \in L^+$ implies $x = x_1 \cdots x_m$ with $m \geq 1$ and $x_i \in L$ for all i. Now write $x' = x_1 \cdots x_{m-1}$ so that $x = x'x_m$. Here, note that $x' \in L^*$; particularly, when $m = 1$ then $x' = \varepsilon$. Thus, $x \in L^*L$. Hence, $L^+ = L^*L$. □

P11 $(L^*)^* = L^*$.

P12 $L^*L^* = L^*$.

P13 $(L_1L_2)^*L_1 = L_1(L_2L_1)^*$.

Proof. Let $x \in (L_1L_2)^*L_1$. Then $x = yz$, where $z \in L_1$ and $y = y_1 \cdots y_n \in (L_1L_2)^*$ with $y_i \in L_1L_2$. Now each $y_i = u_iv_i$, for $u_i \in L_1$ and $v_i \in L_2$. Note that $v_iu_{i+1} \in L_2L_1$, for all i with $1 \leq i \leq n-1$. Hence, $x = yz = (y_1 \cdots y_n)z = (u_1v_1 \cdots u_nv_n)z = u_1(v_1u_2 \cdots v_{n-1}u_nv_nz) \in L_1(L_2L_1)^*$. Converse is similar. Hence, $(L_1L_2)^*L_1 = L_1(L_2L_1)^*$. □

P14 $(L_1 \cup L_2)^* = (L_1^*L_2^*)^*$.

Proof. Observe that $L_1 \subseteq L_1^*$ and $\{\varepsilon\} \subseteq L_2^*$. Hence, by properties P3 and P6, we have $L_1 = L_1\{\varepsilon\} \subseteq L_1^*L_2^*$. Similarly, $L_2 \subseteq L_1^*L_2^*$. Hence, $L_1 \cup L_2 \subseteq L_1^*L_2^*$. Consequently, $(L_1 \cup L_2)^* \subseteq (L_1^*L_2^*)^*$.

For converse, observe that $L_1^* \subseteq (L_1 \cup L_2)^*$. Similarly, $L_2^* \subseteq (L_1 \cup L_2)^*$. Thus,

$$L_1^*L_2^* \subseteq (L_1 \cup L_2)^*(L_1 \cup L_2)^*.$$

But, by property P12, we have $(L_1 \cup L_2)^*(L_1 \cup L_2)^* = (L_1 \cup L_2)^*$ so that $L_1^*L_2^* \subseteq (L_1 \cup L_2)^*$. Hence,

$$(L_1^*L_2^*)^* \subseteq ((L_1 \cup L_2)^*)^* = (L_1 \cup L_2)^*.$$

□

2.4 Finite Representation

Proficiency in a language does not expect one to know all the sentences of the language; rather with some limited information one should be able to come up with all possible sentences of the language. Even in case of programming languages, a compiler validates a program - a sentence in the programming language - with a finite set of instructions incorporated in it. Thus, we are interested in a finite representation of a language - that is, by giving a finite amount of information, all the strings of a language shall be enumerated/validated.

Now, we look at the languages for which finite representation is possible. Given an alphabet Σ, to start with, the languages with single string $\{x\}$ and $\varnothing$ can have finite representation, say x and $\varnothing$, respectively. In this way, finite languages can also be given a finite representation; say, by enumerating all the strings. Thus, giving finite representation for infinite languages is a nontrivial interesting problem. In this context, the operations on languages may be helpful.

For example, the infinite language $\{\varepsilon, ab, abab, ababab, \ldots\}$ can be considered as the Kleene star of the language $\{ab\}$, that is $\{ab\}^*$. Thus, using Kleene star operation we can have finite representation for some infinite languages.

While operations are under consideration, to give finite representation for languages one may first look at the indivisible languages, namely $\varnothing, \{\varepsilon\}$, and $\{a\}$, for all $a \in \Sigma$, as basis elements.

To construct $\{x\}$, for $x \in \Sigma^*$, we can use the operation concatenation over the basis elements. For example, if $x = aba$ then choose $\{a\}$ and $\{b\}$; and concatenate $\{a\}\{b\}\{a\}$ to get $\{aba\}$. Any finite language over Σ, say $\{x_1, \ldots, x_n\}$ can be obtained by considering the union $\{x_1\} \cup \cdots \cup \{x_n\}$.

In this section, we look at the aspects of considering operations over basis elements to represent a language. This is one aspect of representing a language. There are many other aspects to give finite representations; some such aspects will be considered in the later chapters.

2.4.1 Regular Expressions

We now consider the class of languages obtained by applying union, concatenation, and Kleene star for finitely many times on the basis elements. These languages are known as regular languages and the corresponding finite representations are known as regular expressions.

Definition 2.4.1 (Regular Expression)**.** We define a *regular expression* over an alphabet Σ recursively as follows.

1. $\varnothing, \varepsilon$, and a, for each $a \in \Sigma$, are regular expressions representing the languages $\varnothing, \{\varepsilon\}$, and $\{a\}$, respectively.

2. If r and s are regular expressions representing the languages R and S, respectively, then so are

   (a) $(r + s)$ representing the language $R \cup S$,

   (b) (rs) representing the language RS, and

   (c) (r^*) representing the language R^*.

In a regular expression we keep a minimum number of parenthesis which are required to avoid ambiguity in the expression. For example, we may simply write $r + st$ in case of $(r + (st))$. Similarly, $r + s + t$ for $((r + s) + t)$.

Definition 2.4.2. If r is a regular expression, then the language represented by r is denoted by $L(r)$. Further, a language L is said to be *regular* if there is a regular expression r such that $L = L(r)$.

Remark 2.4.3.

1. A regular language over an alphabet Σ is the one that can be obtained from the emptyset, $\{\varepsilon\}$, and $\{a\}$, for $a \in \Sigma$, by finitely many applications of union, concatenation and Kleene star.

2. The smallest class of languages over an alphabet Σ which contains $\varnothing, \{\varepsilon\}$, and $\{a\}$ and is closed with respect to union, concatenation, and Kleene star is the class of all regular languages over Σ.

Example 2.4.4. As we observed earlier that the languages $\varnothing, \{\varepsilon\}$, $\{a\}$, and all finite sets are regular.

Example 2.4.5. $\{a^n \mid n \geq 0\}$ is regular as it can be represented by the expression a^*.

Example 2.4.6. Σ^*, the set of all strings over an alphabet Σ, is regular. For instance, if $\Sigma = \{a_1, a_2, \ldots, a_n\}$, then Σ^* can be represented as $(a_1 + a_2 + \cdots + a_n)^*$.

Example 2.4.7. The set of all strings over $\{a, b\}$ which contain ab as a substring is regular. For instance, the set can be written as

$$\begin{aligned} & \{x \in \{a,b\}^* \mid ab \text{ is a substring of } x\} \\ = \; & \{yabz \mid y, z \in \{a,b\}^*\} \\ = \; & \{a,b\}^*\{ab\}\{a,b\}^* \end{aligned}$$

Hence, the corresponding regular expression is $(a + b)^*ab(a + b)^*$.

Example 2.4.8. The language L over $\{0,1\}$ that contains 01 or 10 as substring is regular.

$$\begin{aligned} L &= \{x \mid 01 \text{ is a substring of } x\} \cup \{x \mid 10 \text{ is a substring of } x\} \\ &= \{y01z \mid y, z \in \Sigma^*\} \cup \{u10v \mid u, v \in \Sigma^*\} \\ &= \Sigma^*\{01\}\Sigma^* \cup \Sigma^*\{10\}\Sigma^* \end{aligned}$$

Since Σ^*, $\{01\}$, and $\{10\}$ are regular we have L to be regular. In fact, at this point, one can easily notice that

$$(0+1)^*01(0+1)^* + (0+1)^*10(0+1)^*$$

is a regular expression representing L.

Example 2.4.9. The set of all strings over $\{a,b\}$ which do not contain ab as a substring. By analyzing the language one can observe that precisely the language is as follows.

$$\{b^n a^m \mid m, n \geq 0\}$$

Thus, a regular expression of the language is b^*a^* and hence the language is regular.

Example 2.4.10. The set of strings over $\{a,b\}$ which contain odd number of a's is regular. Although the set can be represented in set builder form as

$$\{x \in \{a,b\}^* \mid |x|_a = 2n+1, \text{ for some } n\},$$

writing a regular expression for the language is little tricky job. Hence, we postpone the argument to Chapter 3 (see Example 3.3.6), where we construct a regular grammar for the language. Regular grammar is a tool to generate regular languages.

Example 2.4.11. The set of strings over $\{a,b\}$ which contain odd number of a's and even number of b's is regular. As above, a set builder form of the set is:

$$\{x \in \{a,b\}^* \mid |x|_a = 2n+1, \text{ for some } n \text{ and } |x|_b = 2m, \text{ for some } m\}.$$

Writing a regular expression for the language is even more trickier than the earlier example. This will be handled in Chapter 4 using finite automata, yet another tool to represent regular languages.

Definition 2.4.12. Two regular expressions r_1 and r_2 are said to be *equivalent* if they represent the same language; in which case, we write $r_1 \approx r_2$.

Example 2.4.13. The regular expressions $(10+1)^*$ and $((10)^*1^*)^*$ are equivalent.

Since $L((10)^*) = \{(10)^n \mid n \geq 0\}$ and $L(1^*) = \{1^m \mid m \geq 0\}$, we have $L((10)^*1^*) = \{(10)^n 1^m \mid m, n \geq 0\}$. This implies

$$\begin{aligned} L(((10)^*1^*)^*) &= \{(10)^{n_1}1^{m_1}(10)^{n_2}1^{m_2}\cdots(10)^{n_l}1^{m_l} \mid m_i, n_i \geq 0 \text{ and } 0 \leq i \leq l\} \\ &= \{x_1x_2\cdots x_k \mid x_i = 10 \text{ or } 1\}, \text{ where } k = \sum_{i=0}^{l}(m_i + n_i) \\ &\subseteq L((10+1)^*). \end{aligned}$$

Conversely, suppose $x \in L((10+1)^*)$. Then,

$$\begin{aligned} & x = x_1x_2\cdots x_p \text{ where } x_i = 10 \text{ or } 1 \\ \implies & x = (10)^{p_1}1^{q_1}(10)^{p_2}1^{q_2}\cdots(10)^{p_r}1^{q_r} \text{ for } p_i, q_j \geq 0 \\ \implies & x \in L(((10)^*1^*)^*). \end{aligned}$$

Hence, $L((10+1)^*) = L(((10)^*1^*)^*)$ and consequently, $(10+1)^* \approx ((10)^*1^*)^*$.

From property P14, by choosing $L_1 = \{10\}$ and $L_2 = \{1\}$, one may notice that

$$(\{10\} \cup \{1\})^* = (\{10\}^*\{1\}^*)^*.$$

Since 10 and 1 represent the regular languages $\{10\}$ and $\{1\}$, respectively, from the above equation we get

$$(10+1)^* \approx ((10)^*1^*)^*.$$

Since those properties hold good for all languages, by specializing those properties to regular languages and in turn replacing by the corresponding regular expressions we get the following identities for regular expressions.

Let r, r_1, r_2, and r_3 be any regular expressions

1. $r\varepsilon \approx \varepsilon r \approx r$.
2. $r_1r_2 \not\approx r_2r_1$, in general.
3. $r_1(r_2r_3) \approx (r_1r_2)r_3$.
4. $r\varnothing \approx \varnothing r \approx \varnothing$.
5. $\varnothing^* \approx \varepsilon$.
6. $\varepsilon^* \approx \varepsilon$.

7. If $\varepsilon \in L(r)$, then $r^* \approx r^+$.
8. $rr^* \approx r^*r \approx r^+$.
9. $(r_1 + r_2)r_3 \approx r_1r_3 + r_2r_3$.
10. $r_1(r_2 + r_3) \approx r_1r_2 + r_1r_3$.
11. $(r^*)^* \approx r^*$.
12. $(r_1r_2)^*r_1 \approx r_1(r_2r_1)^*$.
13. $(r_1 + r_2)^* \approx (r_1^*r_2^*)^*$.

Example 2.4.14.

1. $b^+(a^*b^* + \varepsilon)b \approx b(b^*a^* + \varepsilon)b^+ \approx b^+a^*b^+$.

   Proof.

   $$\begin{aligned} b^+(a^*b^* + \varepsilon)b &\approx (b^+a^*b^* + b^+\varepsilon)b \\ &\approx b^+a^*b^*b + b^+b \\ &\approx b^+a^*b^+ + b^+b \\ &\approx b^+a^*b^+, \text{ since } L(b^+b) \subseteq L(b^+a^*b^+). \end{aligned}$$

   Similarly, one can observe that $b(b^*a^* + \varepsilon)b^+ \approx b^+a^*b^+$. □

2. $(0^+(01)^*0 + 0^*(10)^*)^* \approx (0 + 10)^*$.

   Proof.

   $$\begin{aligned} (0^+(01)^*0 + 0^*(10)^*)^* &\approx (0^+0(10)^* + 0^*(10)^*)^* \\ &\approx ((0^+0 + 0^*)(10)^*)^* \\ &\approx (0^*(10)^*)^*, \text{ since } L(0^+0) \subseteq L(0^*) \\ &\approx (0 + 10)^*. \end{aligned}$$

   □

Notation 2.4.15. If L is represented by a regular expression r, i.e. $L(r) = L$, then we may simply use r instead of $L(r)$ to indicated the language L. As a consequence, for two regular expressions r and r', $r \approx r'$ and $r = r'$ are equivalent.

Chapter 3

Grammars

In this chapter, we introduce the notion of grammar called context-free grammar (CFG) as a language generator. The notion of derivation is instrumental in understanding how the strings are generated in a grammar. We explain the various properties of derivations using a graphical representation called derivation trees. A special case of CFG, viz. regular grammar, is discussed as tool to generate to regular languages. A more general notion of grammars is presented in Chapter 7.

In the context of natural languages, the grammar of a language is a set of rules which are used to construct/validate sentences of the language. It has been pointed out, in the introduction of Chapter 2, that this is the third step in a formal learning of a language. Now we draw the attention of a reader to look into the general features of the grammars (of natural languages) to formalize the notion in the present context which facilitate for better understanding of formal languages. Consider the English sentence

`The students study automata theory.`

In order to observe that the sentence is grammatically correct, one may attribute certain rules of the English grammar to the sentence and validate it. For instance, the *Article* `the` followed by the *Noun* `students` form a *Noun-phrase* and similarly the *Noun* `automata theory` form a *Noun-phrase*. Further, `study` is a *Verb*. Now, choose the *Sentential form* "*Subject Verb Object*" of the English grammar. As *Subject* or *Object* can be a *Noun-phrase* by plugging in the above words one may conclude that the given sentence is a grammatically correct English sentence. This verification/derivation is depicted in Figure 3.1. The derivation can also be represented by a tree structure as shown in Figure 3.2.

Sentence ⇒ *Subject Verb Object*
⇒ *Noun-phrase Verb Object*
⇒ *Article Noun Verb Object*
⇒ `The` *Noun Verb Object*
⇒ `The students` *Verb Object*
⇒ `The students study` *Object*
⇒ `The students study` *Noun-phrase*
⇒ `The students study` *Noun*
⇒ `The students study automata theory`

Figure 3.1: Derivation of an English Sentence

In this process, we observe that two types of words are in the discussion.

1. The words like `the`, `study`, `students`.
2. The words like *Article*, *Noun*, *Verb*.

The main difference is, if you arrive at a stage where type (1) words are appearing, then you need not say anything more about them. In case you arrive at a stage where you find a word of type (2), then you are assumed to say some more about the word. For example, if the word *Article* comes, then one should say which article need to be chosen among `a`, `an` and `the`. Let us call the type (1) and type (2) words as terminals and nonterminals, respectively, as per their features.

Thus, a grammar should include terminals and nonterminals along with a set of rules which attribute some information regarding nonterminal symbols.

3.1 Context-Free Grammars

We now understand that a grammar should have the following components.

- A set of nonterminal symbols.
- A set of terminal symbols.
- A set of rules.
- As the grammar is to construct/validate sentences of a language, we distinguish a symbol in the set of nonterminals to represent a sentence – from which various sentences of the language can be generated/validated.

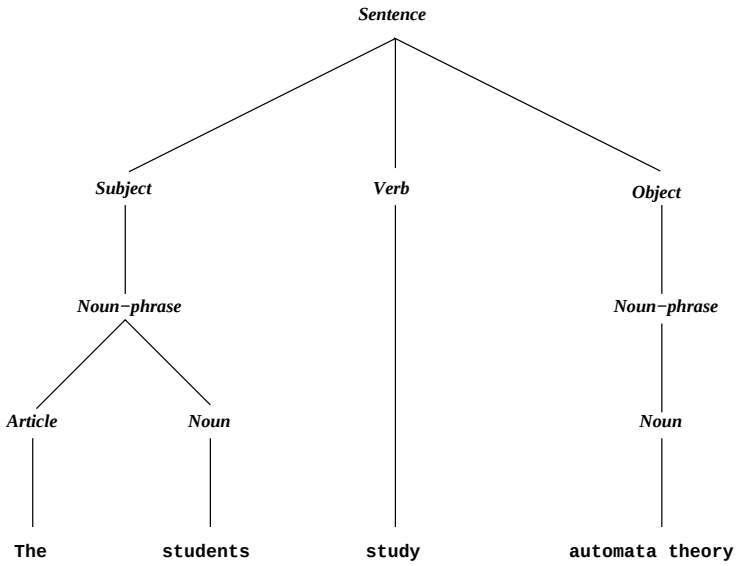


Figure 3.2: Derivation Tree of an English Sentence

With this, we formally define the notion of grammar as below.

Definition 3.1.1. A *grammar* is a quadruple

$$\mathcal{G} = (N, \Sigma, P, S)$$

where

1. N is a finite set of nonterminals,
2. Σ is a finite set of terminals,
3. $S \in N$ is the start symbol, and
4. P is a finite subset of $N \times V^*$ called the set of production rules. Here, $V = N \cup \Sigma$.

It is convenient to write $A \to \alpha$, for the production rule $(A, \alpha) \in P$.

To define a formal notion of validating or deriving a sentence using a grammar, we require the following concepts.

Definition 3.1.2. Let $\mathcal{G} = (N, \Sigma, P, S)$ be a grammar with $V = N \cup \Sigma$.

1. We define a binary relation $\underset{\mathcal{G}}{\Rightarrow}$ on V^* by

   $$\alpha \underset{\mathcal{G}}{\Rightarrow} \beta \text{ if and only if } \alpha = \alpha_1 A \alpha_2, \beta = \alpha_1 \gamma \alpha_2 \text{ and } A \to \gamma \in P,$$

   for all $\alpha, \beta \in V^*$.

2. The relation $\overset{}{\underset{\mathcal{G}}{\Rightarrow}}$ is called *one step* relation on $\mathcal{G}$. If $\alpha \overset{}{\underset{\mathcal{G}}{\Rightarrow}} \beta$, then we call α *yields* β *in one step* in $\mathcal{G}$.

3. The reflexive-transitive closure of $\overset{}{\underset{\mathcal{G}}{\Rightarrow}}$ is denoted by $\overset{*}{\underset{\mathcal{G}}{\Rightarrow}}$. That is, for $\alpha, \beta \in V^*$,

$$\alpha \overset{*}{\underset{\mathcal{G}}{\Rightarrow}} \beta \quad \text{if and only if} \quad \begin{cases} \exists n \geq 0 \text{ and } \alpha_0, \alpha_1, \ldots, \alpha_n \in V^* \text{ such that} \\ \alpha = \alpha_0 \overset{}{\underset{\mathcal{G}}{\Rightarrow}} \alpha_1 \overset{}{\underset{\mathcal{G}}{\Rightarrow}} \cdots \overset{}{\underset{\mathcal{G}}{\Rightarrow}} \alpha_{n-1} \overset{}{\underset{\mathcal{G}}{\Rightarrow}} \alpha_n = \beta. \end{cases}$$

4. For $\alpha, \beta \in V^*$, if $\alpha \overset{*}{\underset{\mathcal{G}}{\Rightarrow}} \beta$, then we say β is *derived* from α or α *derives* β. Further, $\alpha \overset{*}{\underset{\mathcal{G}}{\Rightarrow}} \beta$ is called as a *derivation* in $\mathcal{G}$.

5. If $\alpha = \alpha_0 \overset{}{\underset{\mathcal{G}}{\Rightarrow}} \alpha_1 \overset{}{\underset{\mathcal{G}}{\Rightarrow}} \cdots \overset{}{\underset{\mathcal{G}}{\Rightarrow}} \alpha_{n-1} \overset{}{\underset{\mathcal{G}}{\Rightarrow}} \alpha_n = \beta$ is a derivation, then the *length of the derivation* is n and it may be written as $\alpha \overset{n}{\underset{\mathcal{G}}{\Rightarrow}} \beta$.

6. In a given context, if we deal with only one grammar $\mathcal{G}$, then we may simply write $\Rightarrow$, in stead of $\overset{}{\underset{\mathcal{G}}{\Rightarrow}}$.

7. If $\alpha \overset{*}{\Rightarrow} \beta$ is a derivation, then we say β is the *yield* of the derivation.

8. A string $\alpha \in V^*$ is said to be a *sentential form* in $\mathcal{G}$, if α can be derived from the start symbol S of $\mathcal{G}$. That is, $S \overset{*}{\Rightarrow} \alpha$.

9. In particular, if $\alpha \in \Sigma^*$, then the sentential form α is known as a *sentence*. In which case, we say α is *generated* by $\mathcal{G}$.

10. The *language generated* by $\mathcal{G}$, denoted by $L(\mathcal{G})$, is the set of all sentences generated by $\mathcal{G}$. That is,

$$L(\mathcal{G}) = \{x \in \Sigma^* \mid S \overset{*}{\Rightarrow} x\}.$$

Note that a production rule of a grammar is of the form $A \to \alpha$, where A is a nonterminal symbol. If the nonterminal A appears in a sentential form $X_1 \cdots X_k A X_{k+1} \cdots X_n$, then the sentential form $X_1 \cdots X_k \alpha X_{k+1} \cdots X_n$ can be obtained in one step by replacing A with α. This replacement is independent of the neighboring symbols of A in $X_1 \cdots X_k A X_{k+1} \cdots X_n$. That is, X_i's will not play any role in the replacement. One may call A is within the context of X_i's and hence the rules $A \to \alpha$ are said to be of context-free type. Thus, the type of grammar that is defined here is known as *context-free grammar*, simply CFG. In the later chapters, we relax the constraint and discuss more general types of grammars.

Example 3.1.3. Let $P = \{S \to ab, S \to bb, S \to aba, S \to aab\}$ with $\Sigma = \{a, b\}$ and $N = \{S\}$. Then $\mathcal{G} = (N, \Sigma, P, S)$ is a context-free grammar. Since left hand side of each production rule is the start symbol S and their right hand sides are terminal strings, every derivation in $\mathcal{G}$ is of length one. In fact, we precisely have the following derivation in $\mathcal{G}$.

1. $S \Rightarrow ab$
2. $S \Rightarrow bb$
3. $S \Rightarrow aba$
4. $S \Rightarrow aab$

Hence, the language generated by $\mathcal{G}$,

$$L(\mathcal{G}) = \{ab, bb, aba, aab\}.$$

Notation 3.1.4.

1. $A \to \alpha_1$, $A \to \alpha_2$ can be written as $A \to \alpha_1 \mid \alpha_2$.
2. Normally we use S as the start symbol of a grammar, unless otherwise specified.
3. To give a grammar $\mathcal{G} = (N, \Sigma, P, S)$, it is sufficient to give the production rules only since one may easily find the other components N and Σ of the grammar $\mathcal{G}$ by looking at the rules.

Example 3.1.5. Suppose L is a finite language over an alphabet Σ, say $L = \{x_1, x_2, \ldots, x_n\}$. Then consider the finite set $P = \{S \to x_1 \mid x_2 \mid \cdots \mid x_n\}$. Now, as discussed in Example 3.1.3, one can easily observe that the CFG $(\{S\}, \Sigma, P, S)$ generates the language L.

Example 3.1.6. Let $\Sigma = \{0, 1\}$ and $N = \{S\}$. Consider the CFG $\mathcal{G} = (N, \Sigma, P, S)$, where P has precisely the following production rules.

1. $S \to 0S$
2. $S \to 1S$
3. $S \to \varepsilon$

We now observe that the CFG $\mathcal{G}$ generates Σ^*. As every string generated by $\mathcal{G}$ is a sequence of 0′s and 1′s, clearly we have $L(\mathcal{G}) \subseteq \Sigma^*$. On the other hand, let $x \in \Sigma^*$. If $x = \varepsilon$, then x can be generated in one step using the rule $S \to \varepsilon$. Otherwise, let $x = a_1 a_2 \cdots a_n$, where $a_i = 0$ or 1 for all i. Now, as shown below, x can be derived in $\mathcal{G}$.

$$\begin{array}{rcll} S & \Rightarrow & a_1 S & \text{(if } a_1 = 0\text{, then by rule 1; else by rule 2)} \\ & \Rightarrow & a_1 a_2 S & \text{(as above)} \\ & \vdots & & \\ & \Rightarrow & a_1 a_2 \cdots a_n S & \text{(as above)} \\ & \Rightarrow & a_1 a_2 \cdots a_n \varepsilon & \text{(by rule 3)} \\ & = & x. & \end{array}$$

Hence, $x \in L(\mathcal{G})$. Thus, $L(\mathcal{G}) = \Sigma^*$.

Example 3.1.7. The language $\varnothing$ (over any alphabet Σ) can be generated by a CFG.

Method-I. If the set of productions P is empty, then clearly the CFG $\mathcal{G} = (\{S\}, \Sigma, P, S)$ does not generate any string and hence $L(\mathcal{G}) = \varnothing$.

Method-II. Consider a CFG $\mathcal{G}$ in which each production rule has some non-terminal symbol on its right hand side. Clearly, no terminal string can be generated in $\mathcal{G}$ so that $L(\mathcal{G}) = \varnothing$.

Example 3.1.8. Consider $\mathcal{G} = (\{S\}, \{a\}, P, S)$ where P has the following rules

1. $S \to aS$
2. $S \to \varepsilon$

Let us look at the strings that can be generated by $\mathcal{G}$. Clearly, ε can be generated in one step by using the rule $S \to \varepsilon$. Further, if we choose rule (1) and then rule (2) we get the string a in two steps as follows:

$$S \Rightarrow aS \Rightarrow a\varepsilon = a$$

If we use the rule (1), then one may notice that there will always be the nonterminal S in the resulting sentential form. A derivation can only be terminated by using rule (2). Thus, for any derivation of length k, that

derives some string x, we would have used rule (1) for $k-1$ times and rule (2) once at the end. Precisely, the derivation will be of the form

$$\begin{aligned} S &\Rightarrow aS \\ &\Rightarrow aaS \\ &\vdots \\ &\Rightarrow \overbrace{aa\cdots a}^{k-1} S \\ &\Rightarrow \overbrace{aa\cdots a}^{k-1} \varepsilon = a^{k-1} \end{aligned}$$

Hence, it is clear that $L(\mathcal{G}) = \{a^k \mid k \geq 0\}$

In the following, we give some more examples of typical CFGs.

Example 3.1.9. Consider the grammar having the following production rules:

$$S \rightarrow aSb \mid \varepsilon$$

One may notice that the rule $S \rightarrow aSb$ should be used to derive strings other than ε, and the derivation shall always be terminated by $S \rightarrow \varepsilon$. Thus, a typical derivation is of the form

$$\begin{aligned} S &\Rightarrow aSb \\ &\Rightarrow aaSbb \\ &\vdots \\ &\Rightarrow a^nSb^n \\ &\Rightarrow a^n\varepsilon b^n = a^nb^n \end{aligned}$$

Hence, $L(\mathcal{G}) = \{a^nb^n \mid n \geq 0\}$.

Example 3.1.10. The grammar

$$S \rightarrow aSa \mid bSb \mid a \mid b \mid \varepsilon$$

generates the set of all palindromes over $\{a, b\}$. For instance, the rules $S \rightarrow aSa$ and $S \rightarrow bSb$ will produce same terminals at the same positions towards left and right sides. While terminating the derivation the rules $S \rightarrow a \mid b$ or $S \rightarrow \varepsilon$ will produce odd or even length palindromes, respectively. For example, the palindrome $abbabba$ can be derived as follows.

$$\begin{aligned} S &\Rightarrow aSa \\ &\Rightarrow abSba \\ &\Rightarrow abbSbba \\ &\Rightarrow abbabba \end{aligned}$$

Example 3.1.11. Consider the language $L = \{a^m b^n c^{m+n} \mid m, n \geq 0\}$. We now give production rules of a CFG which generates L. As given in Example 3.1.9, the production rule

$$S \to aSc$$

can be used to produce equal number of a's and c's, respectively, in left and right extremes of a string. In case, there is no b in the string, the production rule

$$S \to \varepsilon$$

can be used to terminate the derivation and produce a string of the form $a^m c^m$. Note that, b's in a string may have leading a's; but, there will not be any a after a b. Thus, a nonterminal symbol that may be used to produce b's must be different from S. Hence, we choose a new nonterminal symbol, say A, and define the rule

$$S \to A$$

to handover the job of producing b's to A. Again, since the number of b's and c's are to be equal, choose the rule

$$A \to bAc$$

to produce b's and c's on either sides. Eventually, the rule

$$A \to \varepsilon$$

can be introduced, which terminate the derivation. Thus, we have the following production rules of a CFG, say $\mathcal{G}$.

$$S \to aSc \mid A \mid \varepsilon$$

$$A \to bAc \mid \varepsilon$$

Now, one can easily observe that $L(\mathcal{G}) = L$.

Example 3.1.12. For the language $\{a^m b^{m+n} c^n \mid m, n \geq 0\}$, one may think in the similar lines of Example 3.1.11 and produce a CFG as given below.

$$\begin{aligned} S &\to AB \\ A &\to aAb \mid \varepsilon \\ B &\to bBc \mid \varepsilon \end{aligned}$$

$$
\begin{array}{lcl}
S & \Rightarrow & S * S \\
 & \Rightarrow & S * a \\
 & \Rightarrow & S + S * a \\
 & \Rightarrow & a + S * a \\
 & \Rightarrow & a + b * a
\end{array}
\qquad
\begin{array}{lcl}
S & \Rightarrow & S * S \\
 & \Rightarrow & S + S * S \\
 & \Rightarrow & a + S * S \\
 & \Rightarrow & a + b * S \\
 & \Rightarrow & a + b * a
\end{array}
\qquad
\begin{array}{lcl}
S & \Rightarrow & S + S \\
 & \Rightarrow & a + S \\
 & \Rightarrow & a + S * S \\
 & \Rightarrow & a + b * S \\
 & \Rightarrow & a + b * a
\end{array}
$$

(1) (2) (3)

Figure 3.3: Derivations for the string $a + b * a$

3.2 Derivation Trees

Let us consider the CFG whose production rules are:

$$S \rightarrow S * S \mid S + S \mid (S) \mid a \mid b$$

Figure 3.3 gives three derivations for the string $a + b * a$. Note that the three derivations are different, because of application of different sequences of rules. Nevertheless, the derivations (1) and (2) share the following feature. A nonterminal that appears at a particular common position in both the derivations derives the same substring of $a + b * a$ in both the derivations. In contrast to that, the derivations (2) and (3) are not sharing such feature. For example, the second S in step 2 of derivations (1) and (2) derives the substring a; whereas, the second S in step 2 of derivation (3) derives the substring $b * a$. In order to distinguish this feature between the derivations of a string, we introduce a graphical representation of a derivation called derivation tree, which will be a useful tool for several other purposes also.

Definition 3.2.1. Let $\mathcal{G} = (N, \Sigma, P, S)$ be a CFG and $V = N \cup \Sigma$. For $A \in N$ and $\alpha \in V^*$, suppose

$$A \stackrel{*}{\Rightarrow} \alpha$$

is a derivation in $\mathcal{G}$. A *derivation tree* or a *parse tree* of the derivation is iteratively defined as follows.

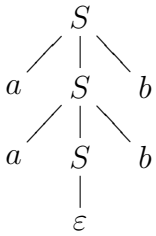
1. A is the root of the tree.
2. If the rule $B \rightarrow X_1 X_2 \cdots X_k$ is applied in the derivation, then new nodes with labels $X_1, X_2, \ldots, X_k$ are created and made children to the node B from left to right in that order.

The construction of the derivation tree of a derivation is illustrated with the following examples.

Example 3.2.2. Consider the following derivation in the CFG given in Example 3.1.9.

$$\begin{aligned} S &\Rightarrow aSb \\ &\Rightarrow aaSbb \\ &\Rightarrow aa\varepsilon bb \\ &= aabb \end{aligned}$$

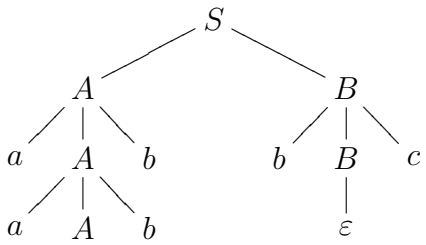
The derivation tree of the above derivation $S \overset{*}{\Rightarrow} aabb$ is shown below.



Example 3.2.3. Consider the following derivation in the CFG given in Example 3.1.12.

$$\begin{aligned} S &\Rightarrow AB \\ &\Rightarrow aAbB \\ &\Rightarrow aaAbbB \\ &\Rightarrow aaAbbB \\ &\Rightarrow aaAbbbBc \\ &\Rightarrow aaAbbb\varepsilon c \\ &= aaAbbbc \end{aligned}$$

The derivation tree of the above derivation $S \overset{*}{\Rightarrow} aaAbbbc$ is shown below.



Note that the yield α of the derivation $A \overset{*}{\Rightarrow} \alpha$ can be identified in the derivation tree by juxtaposing the labels of the leaf nodes from left to right.

Example 3.2.4. Recall the derivations given in Figure 3.3. The derivation trees corresponding to these derivations are in Figure 3.4.

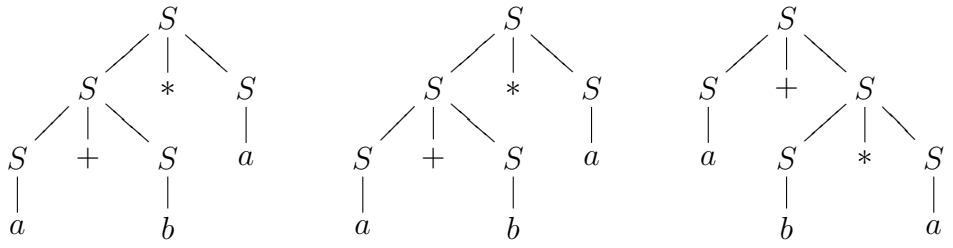


Figure 3.4: Derivation Trees for Figure 3.3

Now we are in a position to formalize the notion of equivalence between derivations, which precisely captures the feature proposed in the beginning of the section. Two derivations are said to be *equivalent* if their derivation trees are same. Thus, equivalent derivations are precisely differed by the order of application of same production rules. That is, the application of production rules in a derivation can be permuted to get an equivalent derivation. For example, as derivation trees (1) and (2) of Figure 3.4 are same, the derivations (1) and (2) of Figure 3.3 are equivalent. Thus, a derivation tree may represent several equivalent derivations. However, for a given derivation tree, whose yield is a terminal string, there is a unique special type of derivation, viz. leftmost derivation (or rightmost derivation).

Definition 3.2.5. For $A \in N$ and $x \in \Sigma^*$, the derivation $A \stackrel{*}{\Rightarrow} x$ is said to be a *leftmost derivation* if the production rule applied at each step is on the leftmost nonterminal symbol of the sentential form. In which case, the derivation is denoted by $A \stackrel{*}{\underset{L}{\Rightarrow}} x$.

Similarly, for $A \in N$ and $x \in \Sigma^*$, the derivation $A \stackrel{*}{\Rightarrow} x$ is said to be a *rightmost derivation* if the production rule applied at each step is on the rightmost nonterminal symbol of the sentential form. In which case, the derivation is denoted by $A \stackrel{*}{\underset{R}{\Rightarrow}} x$.

Because of the similarity between leftmost derivation and rightmost derivation, the properties of the leftmost derivations can be imitated to get similar properties for rightmost derivations. Now, we establish some properties of leftmost derivations.

Theorem 3.2.6. *For $A \in N$ and $x \in \Sigma^*$, $A \stackrel{*}{\Rightarrow} x$ if and only if $A \stackrel{*}{\underset{L}{\Rightarrow}} x$.*

Proof. "Only if" part is straightforward, as every leftmost derivation is, anyway, a derivation.

For "if" part, let

$$A = \alpha_0 \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_{n-1} \Rightarrow \alpha_n = x$$

be a derivation. If $\alpha_i \Rightarrow_L \alpha_{i+1}$, for all $0 \leq i < n$, then we are through. Otherwise, there is an i such that $\alpha_i \not\Rightarrow_L \alpha_{i+1}$. Let k be the least such that $\alpha_k \not\Rightarrow_L \alpha_{k+1}$. Then, we have $\alpha_i \Rightarrow_L \alpha_{i+1}$, for all $i < k$, i.e. we have leftmost substitutions in the first k steps. We now demonstrate how to extend the leftmost substitution to $(k+1)$th step. That is, we show how to convert the derivation

$$A = \alpha_0 \Rightarrow_L \alpha_1 \Rightarrow_L \cdots \Rightarrow_L \alpha_k \Rightarrow \alpha_{k+1} \Rightarrow \cdots \Rightarrow \alpha_{n-1} \Rightarrow \alpha_n = x$$

in to a derivation

$$A = \alpha_0 \Rightarrow_L \alpha_1 \Rightarrow_L \cdots \Rightarrow_L \alpha_k \Rightarrow_L \alpha'_{k+1} \Rightarrow \alpha'_{k+2} \Rightarrow \cdots \Rightarrow \alpha'_{n-1} \Rightarrow \alpha'_n = x$$

in which there are leftmost substitutions in the first $(k+1)$ steps and the derivation is of same length of the original. Hence, by induction one can extend the given derivation to a leftmost derivation $A \overset{*}{\underset{L}{\Rightarrow}} x$.

Since $\alpha_k \Rightarrow \alpha_{k+1}$ but $\alpha_k \not\Rightarrow_L \alpha_{k+1}$, we have

$$\alpha_k = yA_1\beta_1A_2\beta_2,$$

for some $y \in \Sigma^*$, $A_1, A_2 \in N$ and $\beta_1, \beta_2 \in V^*$, and $A_2 \to \gamma_2 \in P$ such that

$$\alpha_{k+1} = yA_1\beta_1\gamma_2\beta_2.$$

But, since the derivation eventually yields the terminal string x, at a later step, say pth step (for $p > k$), A_1 would have been substituted by some string $\gamma_1 \in V^*$ using the rule $A_1 \to \gamma_1 \in P$. Thus the original derivation looks as follows.

$$\begin{aligned}
A = \alpha_0 \Rightarrow_L \alpha_1 \Rightarrow_L \cdots \Rightarrow_L \alpha_k &= yA_1\beta_1A_2\beta_2 \\
&\Rightarrow \alpha_{k+1} = yA_1\beta_1\gamma_2\beta_2 \\
&\qquad = yA_1\xi_1, \text{ with } \xi_1 = \beta_1\gamma_2\beta_2 \\
&\Rightarrow \alpha_{k+2} = yA_1\xi_2, (\text{ here } \xi_1 \Rightarrow \xi_2) \\
&\vdots \\
&\Rightarrow \alpha_{p-1} = yA_1\xi_{p-k-1} \\
&\Rightarrow \alpha_p = y\gamma_1\xi_{p-k-1} \\
&\Rightarrow \alpha_{p+1} \\
&\vdots \\
&\Rightarrow \alpha_n = x
\end{aligned}$$

We now demonstrate a mechanism of using the rule $A_1 \to \gamma_1$ in $(k+1)$th step so that we get a desired derivation. Set

$$\begin{aligned}
&\alpha'_{k+1} = y\gamma_1\beta_1 A_2\beta_2\\
&\alpha'_{k+2} = y\gamma_1\beta_1\gamma_2\beta_2 = y\gamma_1\xi_1\\
&\alpha'_{k+3} = y\gamma_1\xi_2\\
&\qquad\vdots\\
&\alpha'_{p-1} = y\gamma_1\xi_{p-k-2}\\
&\alpha'_{p} = y\gamma_1\xi_{p-k-1} = \alpha_p\\
&\alpha'_{p+1} = \alpha_{p+1}\\
&\qquad\vdots\\
&\alpha'_{n} = \alpha_n
\end{aligned}$$

Now we have the following derivation in which $(k+1)$th step has leftmost substitution, as desired.

$$\begin{aligned}
A = \alpha_0 \Rightarrow_L \alpha_1 \Rightarrow_L \cdots \Rightarrow_L \alpha_k \quad &= \quad yA_1\beta_1 A_2\beta_2\\
&\Rightarrow_L \quad \alpha'_{k+1} = y\gamma_1\beta_1 A_2\beta_2\\
&\Rightarrow \quad \alpha'_{k+2} = y\gamma_1\beta_1\gamma_2\beta_2 = y\gamma_1\xi_1\\
&\Rightarrow \quad \alpha'_{k+3} = y\gamma_1\xi_2\\
&\quad\vdots\\
&\Rightarrow \quad \alpha'_{p-1} = y\gamma_1\xi_{p-k-2}\\
&\Rightarrow \quad \alpha'_{p} = y\gamma_1\xi_{p-k-1} = \alpha_p\\
&\Rightarrow \quad \alpha'_{p+1} = \alpha_{p+1}\\
&\quad\vdots\\
&\Rightarrow \quad \alpha'_{n} = \alpha_n = x
\end{aligned}$$

As stated earlier, we have the theorem by induction. □

Proposition 3.2.7. *Two equivalent leftmost derivations are identical.*

Proof. Let T be the derivation tree of two leftmost derivations D_1 and D_2. Note that the production rules applied at each nonterminal symbol is precisely represented by its children in the derivation tree. Since the derivation tree is same for D_1 and D_2, the production rules applied in both the derivations are same. Moreover, as D_1 and D_2 are leftmost derivations, the order of application of production rules are also same; that is, each production is applied to the leftmost nonterminal symbol. Hence, the derivations D_1 and D_2 are identical. □

Now we are ready to establish the correspondence between derivation trees and leftmost derivations.

Theorem 3.2.8. *Every derivation tree, whose yield is a terminal string, represents a unique leftmost derivation.*

Proof. For $A \in N$ and $x \in \Sigma^*$, suppose T is the derivation tree of a derivation $A \overset{*}{\Rightarrow} x$. By Theorem 3.2.6, we can find an equivalent leftmost derivation $A \overset{*}{\underset{L}{\Rightarrow}} x$. Hence, by Proposition 3.2.7, we have a unique leftmost derivation that is represented by T. □

Theorem 3.2.9. *Let $\mathcal{G} = (N, \Sigma, P, S)$ be a CFG and*

$$\kappa = \max\{|\alpha| \mid A \to \alpha \in P\}.$$

For $x \in L(\mathcal{G})$, if T is a derivation tree for x, then

$$|x| \leq \kappa^h$$

where h is the height of the tree T.

Proof. Since κ is the maximum length of righthand side of each production rule of $\mathcal{G}$, each internal node of T is a parent of at most κ number of children. Hence, T is a κ-ary tree. Now, in the similar lines of Theorem 1.2.3 that is given for binary trees, one can easily prove that T has at most κ^h leaf nodes. Hence, $|x| \leq \kappa^h$. □

3.2.1 Ambiguity

Let $\mathcal{G}$ be a context-free grammar. It may be a case that the CFG $\mathcal{G}$ gives two or more inequivalent derivations for a string $x \in L(\mathcal{G})$. This can be identified by their different derivation trees. While deriving the string, if there are multiple possibilities of application of production rules on the same symbol, one may have a difficulty in choosing a correct rule. In the context of compiler which is constructed based on a grammar, this difficulty will lead to an ambiguity in parsing. Thus, a grammar with such a property is said to be ambiguous.

Definition 3.2.10. Formally, a CFG $\mathcal{G}$ is said to be *ambiguous*, if $\mathcal{G}$ has two different leftmost derivations for some string in $L(\mathcal{G})$. Otherwise, the grammar is said to be *unambiguous*.

Remark 3.2.11. One can equivalently say that a CFG $\mathcal{G}$ is *ambiguous*, if $\mathcal{G}$ has two different rightmost derivations or derivation trees for a string in $L(\mathcal{G})$.

Example 3.2.12. Recall the CFG which generates arithmetic expressions with the production rules

$$S \to S * S \mid S + S \mid (S) \mid a \mid b$$

As shown in Figure 3.3, the arithmetic expression $a + b * a$ has two different leftmost derivations, viz. (2) and (3). Hence, the grammar is ambiguous.

Example 3.2.13. An unambiguous grammar which generates the arithmetic expressions (the language generated by the CFG given in Example 3.2.12) is as follows.

$$\begin{aligned} S &\to S + T \mid T \\ T &\to T * R \mid R \\ R &\to (S) \mid a \mid b \end{aligned}$$

In the context of constructing a compiler or similar such applications, it is desirable that the underlying grammar be unambiguous. Unfortunately, there are some CFLs for which there is no CFG which is unambiguous. Such a CFL is known as *inherently ambiguous* language.

Example 3.2.14. The context-free language

$$\{a^m b^m c^n d^n \mid m, n \geq 1\} \cup \{a^m b^n c^n d^m \mid m, n \geq 1\}$$

is inherently ambiguous. For proof, one may refer to the Hopcroft and Ullman [1979].

3.3 Regular Grammars

Definition 3.3.1. A CFG $\mathcal{G} = (N, \Sigma, P, S)$ is said to be *linear* if every production rule of $\mathcal{G}$ has at most one nonterminal symbol in its righthand side. That is,

$$A \to \alpha \in P \implies \alpha \in \Sigma^* \text{ or } \alpha = xBy, \text{ for some } x, y \in \Sigma^* \text{ and } B \in N.$$

Example 3.3.2. The CFGs given in Examples 3.1.8, 3.1.9 and 3.1.11 are clearly linear. Whereas, the CFG given in Example 3.1.12 is not linear.

Remark 3.3.3. If $\mathcal{G}$ is a linear grammar, then every derivation in $\mathcal{G}$ is a leftmost derivation as well as rightmost derivation. This is because there is exactly one nonterminal symbol in the sentential form of each internal step of the derivation.

Definition 3.3.4. A linear grammar $\mathcal{G} = (N, \Sigma, P, S)$ is said to be *right linear* if the nonterminal symbol in the righthand side of each production rule, if any, occurs at the right end. That is, righthand side of each production rule is of the form – a terminal string followed by at most one nonterminal symbol – as shown below.

$$A \to x \quad \text{or} \quad A \to xB$$

for some $x \in \Sigma^*$ and $B \in N$.

Similarly, a *left linear grammar* is defined by considering the production rules in the following form.

$$A \to x \quad \text{or} \quad A \to Bx$$

for some $x \in \Sigma^*$ and $B \in N$.

Because of the similarity in the definitions of left linear grammar and right linear grammar, every result which is true for one can be imitated to obtain a parallel result for the other. In fact, the notion of left linear grammar is equivalent right linear grammar (see Exercise **??**). Here, by equivalence we mean, if L is generated by a right linear grammar, then there exists a left linear grammar that generates L; and vice-versa.

Example 3.3.5. The CFG given in Example 3.1.8 is a right linear grammar.

For the language $\{a^k \mid k \geq 0\}$, an equivalent left linear grammar is given by the following production rules.

$$S \to Sa \mid \varepsilon$$

Example 3.3.6. We give a right linear grammar for the language

$$L = \{x \in \{a, b\}^* \mid |x|_a = 2n + 1, \quad \text{for some } n\}.$$

We understand that the nonterminal symbols in a grammar maintain the properties of the strings that they generate or whenever they appear in a derivation they determine the properties of the partial string that the derivation so far generated.

Since we are looking for a right linear grammar for L, each production rule will have at most one nonterminal symbol on its righthand side and hence at each internal step of a desired derivation, we will have exactly one nonterminal symbol. While, we are generating a sequence of a's and b's for the language, we need not keep track of the number of b's that are generated, so far. Whereas, we need to keep track the number of a's; here, it need not be

the actual number, rather the parity (even or odd) of the number of a's that are generated so far. So to keep track of the parity information, we require two nonterminal symbols, say O and E, respectively for odd and even. In the beginning of any derivation, we would have generated zero number of symbols – in general, even number of a's. Thus, it is expected that the nonterminal symbol E to be the start symbol of the desired grammar. While E generates the terminal symbol b, the derivation can continue to be with nonterminal symbol E. Whereas, if one a is generated then we change to the symbol O, as the number of a's generated so far will be odd from even. Similarly, we switch to E on generating an a with the symbol O and continue to be with O on generating any number of b's. Precisely, we have obtained the following production rules.

$$E \to bE \mid aO$$

$$O \to bO \mid aE$$

Since, our criteria is to generate a string with odd number of a's, we can terminate a derivation while continuing in O. That is, we introduce the production rule

$$O \to \varepsilon$$

Hence, we have the right linear grammar $\mathcal{G} = (\{E, O\}, \{a, b\}, P, E)$, where P has the above defined three productions. Now, one can easily observe that $L(\mathcal{G}) = L$.

Recall that the language under discussion is stated as a regular language in Chapter 1 (refer Example 2.4.10). Whereas, we did not supply any proof in support. Here, we could identify a right linear grammar that generates the language. In fact, we have the following result.

Theorem 3.3.7. *The language generated by a right linear grammar is regular. Moreover, for every regular language L, there exists a right linear grammar that generates L.*

An elegant proof of this theorem would require some more concepts and hence postponed to later chapters. For proof of the theorem, one may refer to Chapter 4. In view of the theorem, we have the following definition.

Definition 3.3.8. Right linear grammars are also called as *regular grammars*.

Remark 3.3.9. Since left linear grammars and right linear grammars are equivalent, left linear grammars also precisely generate regular languages. Hence, left linear grammars are also called as regular grammars.

Example 3.3.10. The language $\{x \in \{a,b\}^* \mid ab \text{ is a substring of } x\}$ can be generated by the following regular grammar.

$$S \to aS \mid bS \mid abA$$

$$A \to aA \mid bA \mid \varepsilon$$

Example 3.3.11. Consider the regular grammar with the following production rules.

$$S \to aA \mid bS \mid \varepsilon$$

$$A \to aS \mid bA$$

Note that the grammar generates the set of all strings over {a, b} having even number of a's.

Example 3.3.12. Consider the language represented by the regular expression a^*b^+ over $\{a,b\}$, i.e.

$$\{a^m b^n \in \{a,b\}^* \mid m \geq 0, n \geq 1\}$$

It can be easily observe that the following regular grammar generates the language.

$$\begin{aligned} S &\to aS \mid B \\ B &\to bB \mid b \end{aligned}$$

Example 3.3.13. The grammar with the following production rules is clearly regular.

$$\begin{aligned} S &\to bS \mid aA \\ A &\to bA \mid aB \\ B &\to bB \mid aS \mid \varepsilon \end{aligned}$$

It can be observed that the language $\{x \in \{a,b\}^* \mid |x|_a \equiv 2 \mod 3\}$ is generated by the grammar.

Example 3.3.14. Consider the language

$$L = \Big\{ x \in (0+1)^* \;\Big|\; |x|_0 \text{ is even } \Leftrightarrow |x|_1 \text{ is odd} \Big\}.$$

It is little tedious to construct a regular grammar to show that L is regular. However, using a better tool, we show that L is regular later (See Example 5.1.3).

3.4 Digraph Representation

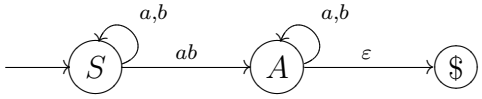
We could achieve in writing a right linear grammar for some languages. However, we face difficulties in constructing a right linear grammar for some languages that are known to be regular. We are now going to represent a right linear grammar by a digraph which shall give a better approach in writing/constructing right linear grammar for languages. In fact, this digraph representation motivates one to think about the notion of finite automaton which will be shown, in the next chapter, as an ultimate tool in understanding regular languages.

Definition 3.4.1. Given a right linear grammar $G = (N, T, P, S)$, define the digraph (V, E), where the vertex set $V = N \cup \{\$\}$ with a new symbol \$ and the edge set E is formed as follows.

1. $(A, B) \in E \iff A \to xB \in P$, for some $x \in T^*$
2. $(A, \$) \in E \iff A \to x \in P$, for some $x \in T^*$

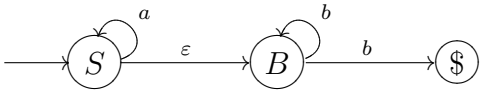
In which case, the arc from A to B is labeled by x.

Example 3.4.2. The digraph for the grammar presented in Example 3.3.10 is as follows.



Remark 3.4.3. From a digraph one can easily give the corresponding right linear grammar.

Example 3.4.4. The digraph for the grammar presented in Example 3.3.12 is as follows.



Remark 3.4.5. A derivation in a right linear grammar can be represented, in its digraph, by a path from the starting node S to the special node \$; and conversely. We illustrate this through the following.

Consider the following derivation for the string aab in the grammar given in Example 3.3.12.

$$S \Rightarrow aS \Rightarrow aaS \Rightarrow aaB \Rightarrow aab$$

The derivation can be traced by a path from S to \$ in the corresponding digraph (refer to Example 3.4.4) as shown below.

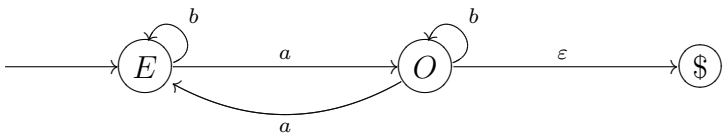
$$S \xrightarrow{a} S \xrightarrow{a} S \xrightarrow{\varepsilon} B \xrightarrow{b} \$$$

One may notice that the concatenation of the labels on the path, called *label of the path*, gives the desired string. Conversely, it is easy to see that the label of any path from S to \$ can be derived in G.

Chapter 4

Finite Automata

Regular grammars, as language generating devices, are intended to generate regular languages - the class of languages that are represented by regular expressions. Finite automata, as language accepting devices, are important tools to understand the regular languages better. Let us consider the regular language - the set of all strings over $\{a, b\}$ having odd number of a's. Recall the grammar for this language as given in Example 3.3.6. A digraph representation of the grammar is given below:



Let us traverse the digraph via a sequence of a's and b's starting at the node E. We notice that, at a given point of time, if we are at the node E, then so far we have encountered even number of a's. Whereas, if we are at the node O, then so far we have traversed through odd number of a's. Of course, being at the node \$ has the same effect as that of node O, regarding number of a's; rather once we reach to \$, then we will not have any further move.

Thus, in a digraph that models a system which understands a language, nodes holds some information about the traversal. As each node is holding some information it can be considered as a state of the system and hence a state can be considered as a memory creating unit. As we are interested in the languages having finite representation, we restrict ourselves to those systems with finite number of states only. In such a system we have transitions between the states on symbols of the alphabet. Thus, we may call them as finite state transition systems. As the transitions are predefined in a finite state transition system, it automatically changes states based on the symbols given as input. Thus a finite state transition system can also be called as a

finite state automaton or simply a finite automaton – a device that works automatically. The plural form of automaton is automata.

In this chapter, we introduce the notion of finite automata and show that they model the class of regular languages. In fact, we observe that finite automata, regular grammars and regular expressions are equivalent; each of them are to represent regular languages.

4.1 Deterministic Finite Automata

Deterministic finite automaton is a type of finite automaton in which the transitions are deterministic, in the sense that there will be exactly one transition from a state on an input symbol. Formally,

a *deterministic finite automaton* (DFA) is a quintuple $\mathscr{A} = (Q, \Sigma, \delta, q_0, F)$, where

Q is a finite set called the *set of states*,

Σ is a finite set called the *input alphabet*,

$q_0 \in Q$, called the *initial/start state*,

$F \subseteq Q$, called the set of *final/accept states*, and

$\delta : Q \times \Sigma \longrightarrow Q$ is a function called the *transition function* or *next-state function*.

Note that, for every state and an input symbol, the transition function δ assigns a unique next state.

Example 4.1.1. Let $Q = \{p, q, r\}$, $\Sigma = \{a, b\}$, $F = \{r\}$ and δ is given by the following table:

δ	a	b
p	q	p
q	r	p
r	r	r

Clearly, $\mathscr{A} = (Q, \Sigma, \delta, p, F)$ is a DFA.

We normally use symbols $p, q, r, \ldots$ with or without subscripts to denote states of a DFA.

Transition Table

Instead of explicitly giving all the components of the quintuple of a DFA, we may simply point out the initial state and the final states of the DFA in the table of transition function, called *transition table*. For instance, we use an arrow to point the initial state and we encircle all the final states. Thus, we can have an alternative representation of a DFA, as all the components of

the DFA now can be interpreted from this representation. For example, the DFA in Example 4.1.1 can be denoted by the following transition table.

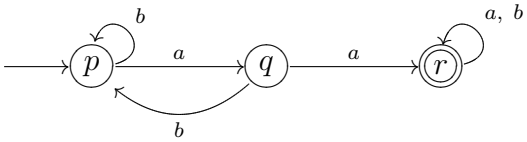
δ	a	b
$\rightarrow p$	q	p
q	r	p
$\textcircled{r}$	r	r

Transition Diagram

Normally, we associate some graphical representation to understand abstract concepts better. In the present context also we have a digraph representation for a DFA, $(Q, \Sigma, \delta, q_0, F)$, called a *state transition diagram* or simply a *transition diagram* which can be constructed as follows:

1. Every state in Q is represented by a node.
2. If $\delta(p, a) = q$, then there is an arc from p to q labeled a.
3. If there are multiple arcs from labeled $a_1, \ldots a_{k-1}$, and a_k, one state to another state, then we simply put only one arc labeled $a_1, \ldots, a_{k-1}, a_k$.
4. There is an arrow with no source into the initial state q_0.
5. Final states are indicated by double circle.

The transition diagram for the DFA given in Example 4.1.1 is as below:



Note that there are two transitions from the state r to itself on symbols a and b. As indicated in the point 3 above, these are indicated by a single arc from r to r labeled a, b.

Extended Transition Function

Recall that the transition function δ assigns a state for each state and an input symbol. This naturally can be extended to all strings in Σ^*, i.e. assigning a state for each state and an input string.

The extended transition function $\hat{\delta} : Q \times \Sigma^* \longrightarrow Q$ is defined recursively as follows: For all $q \in Q$, $x \in \Sigma^*$ and $a \in \Sigma$,

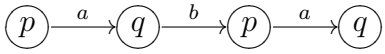
$$\begin{aligned} \hat{\delta}(q, \varepsilon) &= q \text{ and} \\ \hat{\delta}(q, xa) &= \delta(\hat{\delta}(q, x), a). \end{aligned}$$

For example, in the DFA given in Example 4.1.1, $\hat{\delta}(p, aba)$ is q because

$$\begin{aligned} \hat{\delta}(p, aba) &= \delta(\hat{\delta}(p, ab), a) \\ &= \delta(\delta(\hat{\delta}(p, a), b), a) \\ &= \delta(\delta(\delta(\hat{\delta}(p, \varepsilon), a), b), a) \\ &= \delta(\delta(\delta(p, a), b), a) \\ &= \delta(\delta(q, b), a) \\ &= \delta(p, a) = q \end{aligned}$$

Given $p \in Q$ and $x = a_1 a_2 \cdots a_k \in \Sigma^*$, $\hat{\delta}(p, x)$ can be evaluated easily using the transition diagram by identifying the state that can be reached by traversing from p via the sequence of arcs labeled $a_1, a_2, \ldots, a_k$.

For instance, the above case can easily be seen by the traversing through the path labeled aba from p to reach to q as shown below:



Language of a DFA

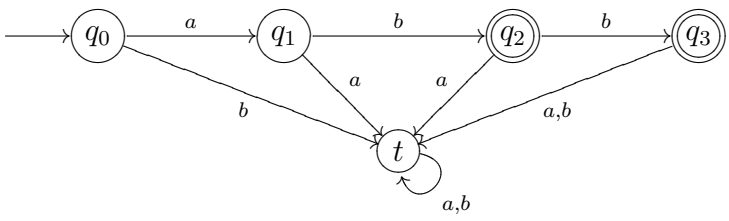
Now, we are in a position to define the notion of acceptance of a string, and consequently acceptance of a language, by a DFA.

A string $x \in \Sigma^*$ is said to be *accepted by a DFA* $\mathscr{A} = (Q, \Sigma, \delta, q_0, F)$ if $\hat{\delta}(q_0, x) \in F$. That is, when you apply the string x in the initial state the DFA will reach to a final state.

The set of all strings accepted by the DFA $\mathscr{A}$ is said to be the language accepted by $\mathscr{A}$ and is denoted by $L(\mathscr{A})$. That is,

$$L(\mathscr{A}) = \{x \in \Sigma^* \mid \hat{\delta}(q_0, x) \in F\}.$$

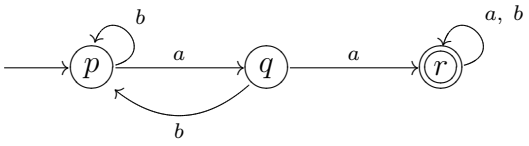
Example 4.1.2. Consider the following DFA



The only way to reach from the initial state q_0 to the final state q_2 is through the string ab and it is through abb to reach another final state q_3. Thus, the language accepted by the DFA is

$$\{ab, abb\}.$$

Example 4.1.3. As shown below, let us recall the transition diagram of the DFA given in Example 4.1.1.



1. One may notice that if there is aa in the input, then the DFA leads us from the initial state p to the final state r. Since r is also a trap state, after reaching to r we continue to be at r on any subsequent input.

2. If the input does not contain aa, then we will be shuffling between p and q but never reach the final state r.

Hence, the language accepted by the DFA is the set of all strings over $\{a, b\}$ having aa as substring, i.e.

$$\{xaay \mid x, y \in \{a, b\}^*\}.$$

Description of a DFA

Note that a DFA is an abstract (computing) device. The depiction given in Figure 4.1 shall facilitate one to understand its behavior. As shown in the figure, there are mainly three components namely input tape, reading head, and finite control. It is assumed that a DFA has a left-justified infinite tape to accommodate an input of any length. The input tape is divided into cells such that each cell accommodate a single input symbol. The reading head is connected to the input tape from finite control, which can read one symbol at a time. The finite control has the states and the information of the transition function along with a pointer that points to exactly one state.

At a given point of time, the DFA will be in some internal state, say p, called the current state, pointed by the pointer and the reading head will be reading a symbol, say a, from the input tape called the current symbol. If $\delta(p, a) = q$, then at the next point of time the DFA will change its internal state from p to q (now the pointer will point to q) and the reading head will move one cell to the right.

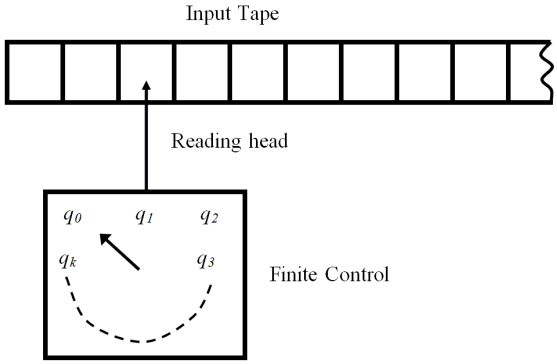


Figure 4.1: Depiction of Finite Automaton

Initializing a DFA with an input string $x \in \Sigma^*$ we mean x be placed on the input tape from the left most (first) cell of the tape with the reading head placed on the first cell and by setting the initial state as the current state. By the time the input is exhausted, if the current state of the DFA is a final state, then the input x is accepted by the DFA. Otherwise, x is rejected by the DFA.

Configurations

A configuration or an instantaneous description of a DFA gives the information about the current state and the portion of the input string that is right from and on to the reading head, i.e. the portion yet to be read. Formally, a *configuration* is an element of $Q \times \Sigma^*$.

Observe that for a given input string x the initial configuration is (q_0, x) and a final configuration of a DFA is of the form (p, ε).

The notion of computation in a DFA $\mathscr{A}$ can be described through configurations. For which, first we define one step relation as follows.

Definition 4.1.4. Let $C = (p, x)$ and $C' = (q, y)$ be two configurations. If $\delta(p, a) = q$ and $x = ay$, then we say that the DFA $\mathscr{A}$ moves from C to C' in *one step* and is denoted as $C \vdash_{\mathscr{A}} C'$.

Clearly, $\vdash_{\mathscr{A}}$ is a binary relation on the set of configurations of $\mathscr{A}$. In a given context, if there is only one DFA under discussion, then we may simply use $\vdash$ instead of $\vdash_{\mathscr{A}}$. The reflexive transitive closure of $\vdash$ may be denoted by $\overset{*}{\vdash}$. That is, $C \overset{*}{\vdash} C'$ if and only if there exist configurations

$C_0, C_1, \ldots, C_n$ such that

$$C = C_0 \vdash C_1 \vdash C_2 \vdash \cdots \vdash C_n = C'$$

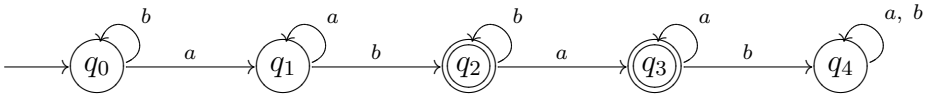
Definition 4.1.5. A the *computation* of $\mathscr{A}$ on the input x is of the form

$$C \vdash^* C'$$

where $C = (q_0, x)$ and $C' = (p, \varepsilon)$, for some p.

Remark 4.1.6. Given a DFA $\mathscr{A} = (Q, \Sigma, \delta, q_0, F)$, $x \in L(\mathscr{A})$ if and only if $(q_0, x) \vdash^* (p, \varepsilon)$ for some $p \in F$.

Example 4.1.7. Consider the following DFA



As states of a DFA are memory creating units, we demonstrate the language of the DFA under consideration by explaining the roles of each of its states.

1. It is clear that, if the input contains only b's, then the DFA remains in the initial state q_0 only.

2. On the other hand, if the input has an a, then the DFA transits from q_0 to q_1. On any subsequent a's, it remains in q_1 only. Thus, the role of q_1 in the DFA is to understand that the input has at least one a.

3. Further, the DFA goes from q_1 to q_2 via a b and remains at q_2 on subsequent b's. Thus, q_2 recognizes that the input has an occurrence of an ab.

   Since q_2 is a final state, the DFA accepts all those strings which have one occurrence of ab.

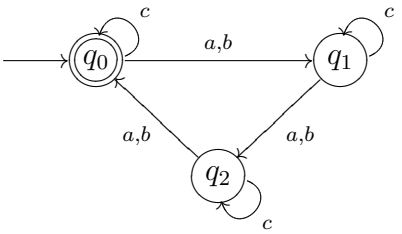
4. Subsequently, if we have a number of a's, the DFA will reach to q_3, which is a final state, and remains there; so that all such strings will also be accepted.

5. But, from then, via b the DFA goes to the trap state q_4 and since it is not a final state, all those strings will not be accepted. Here, note that role of q_4 is to remember the second occurrence of ab in the input.

Thus, the language accepted by the DFA is the set of all strings over $\{a, b\}$ which have exactly one occurrence of ab. That is,

$$\Big\{ x \in \{a, b\}^* \;\Big|\; |x|_{ab} = 1 \Big\}.$$

Example 4.1.8. Consider the following DFA



1. First, observe that the number of occurrences of c's in the input at any state is simply ignored by the state by remaining in the same state.

2. Whereas, on input a or b, every state will lead the DFA to its next state as q_0 to q_1, q_1 to q_2 and q_2 to q_0.

3. It can be clearly visualized that, if the total number of a's and b's in the input is a multiple of 3, then the DFA will be brought back to the initial state q_0. Since q_0 is the final state those strings will be accepted.

4. On the other hand, if any string violates the above stated condition, then the DFA will either be in q_1 or be in q_2 and hence they will not be accepted. More precisely, if the total number of a's and b's in the input leaves the remainder 1 or 2, when it is divided by 3, then the DFA will be in the state q_1 or q_2, respectively.

Hence the language of the DFA is

$$\Big\{x \in \{a,b,c\}^* \;\Big|\; |x|_a + |x|_b \equiv 0 \mod 3\Big\}.$$

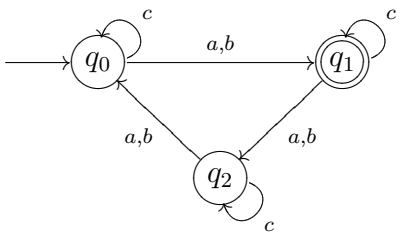
From the above discussion, further we ascertain the following:

1. Instead of q_0, if q_1 is only the final state (as shown in (i), below), then the language will be

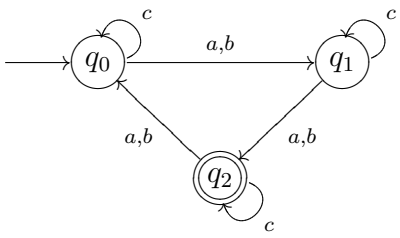
$$\Big\{x \in \{a,b,c\}^* \;\Big|\; |x|_a + |x|_b \equiv 1 \mod 3\Big\}$$

2. Similarly, instead of q_0, if q_2 is only the final state (as shown in (ii), below), then the language will be

$$\Big\{x \in \{a,b,c\}^* \;\Big|\; |x|_a + |x|_b \equiv 2 \mod 3\Big\}$$



(i)

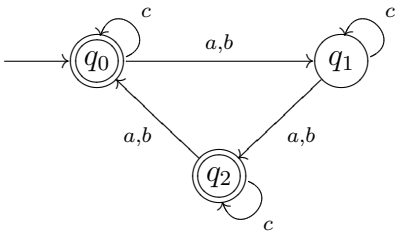


(ii)

3. In the similar lines, one may observe that the language

$$\begin{aligned} & \left\{ x \in \{a,b,c\}^* \;\middle|\; |x|_a + |x|_b \not\equiv 1 \mod 3 \right\} \\ = \; & \left\{ x \in \{a,b,c\}^* \;\middle|\; |x|_a + |x|_b \equiv \; 0 \text{ or } 2 \mod 3 \right\} \end{aligned}$$

can be accepted by the DFA shown below, by making both q_0 and q_2 final states.



4. Likewise, other combinations of the states, viz. q_0 and q_1, or q_1 and q_2, can be made final states and the languages be observed appropriately.

Example 4.1.9. Consider the language L over $\{a, b\}$ which contains the strings whose lengths are from the arithmetic progression

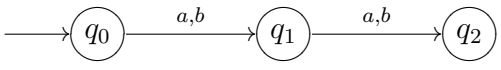
$$P = \{2, 5, 8, 11, \ldots\} = \{2 + 3n \mid n \geq 0\}.$$

That is,

$$L = \left\{ x \in \{a,b\}^* \;\middle|\; |x| \in P \right\}.$$

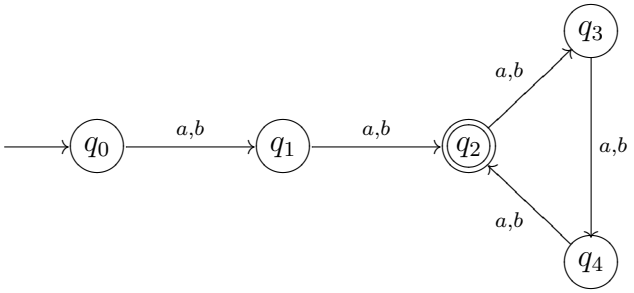
We construct a DFA accepting L. Here, we need to consider states whose role is to count the number of symbols of the input.

1. First, we need to recognize the length 2. That is, first two symbols; for which we may consider the following state transitions.



Clearly, on any input over $\{a, b\}$, we are in the state q_2 means that, so far, we have read the portion of length 2.

2. Then, the length of rest of the string need to be a multiple of 3. Here, we borrow the idea from the Example 4.1.8 and consider the following depicted state transitions, further.

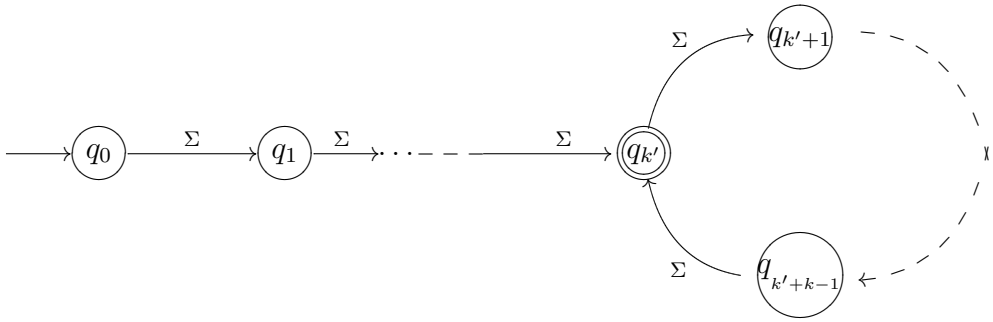


Clearly, this DFA accepts L.

In general, for any arithmetic progression $P = \{k' + kn \mid n \geq 0\}$ with $k, k' \in \mathbb{N}$, if we consider the language

$$L = \left\{ x \in \Sigma^* \;\middle|\; |x| \in P \right\}$$

over an alphabet Σ, then the following DFA can be suggested for L.



Example 4.1.10. Consider the language over $\{a, b\}$ consisting of those strings that end with a. That is,

$$\{xa \mid x \in \{a, b\}^*\}.$$

We observe the following to construct a DFA for the language.

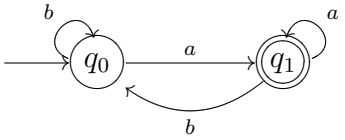
1. If we assume a state q_0 as the initial state, then an occurrence of a in the input need to be distinguished. This can be done by changing the state to some other, say q_1. Whereas, we continue to be in q_0 on b's.

2. On subsequent a's at q_1 let us remain at q_1. If a string ends on reaching q_1, clearly it is ending with an a. By making q_1 a final state, all such strings can be accepted.

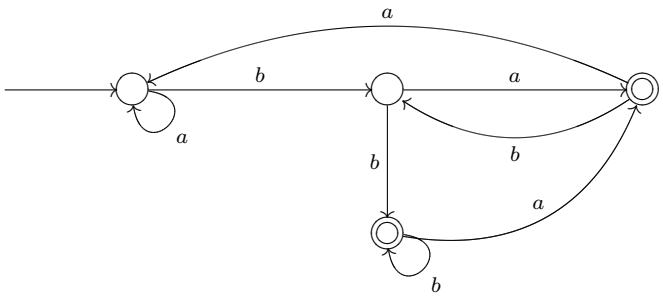
3. If we encounter a b at q_1, then we should go out of q_1, as it is a final state.

4. Since again the possibilities of further occurrences of a's need to cross-checked and q_0 is already taking care of that, we may assign the transition out of q_1 on b to q_0.

Thus, we have the following DFA which accepts the given language.

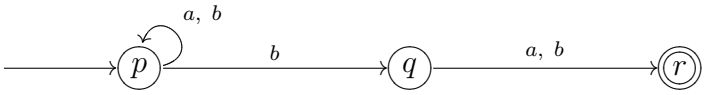


Example 4.1.11. Let us consider the following DFA

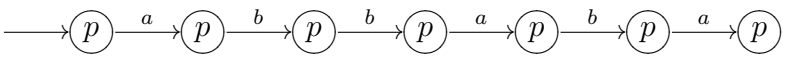


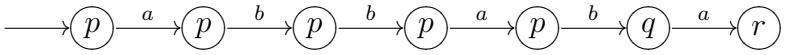
Unlike the above examples, it is little tricky to ascertain the language accepted by the DFA. By spending some amount of time, one may possibly report that the language is the set of all strings over $\{a, b\}$ with last but one symbol as b.

But for this language, if we consider the following type of finite automaton one can easily be convinced (with an appropriate notion of language acceptance) that the language is so.



Note that, in this type of finite automaton we are considering multiple (possibly zero) number of transitions for an input symbol in a state. Thus, if a string is given as input, then one may observe that there can be multiple next states for the string. For example, in the above finite automaton, if *abbaba* is given as input then the following two traces can be identified.





Clearly, p and r are the next states after processing the string *abbaba*. Since it is reaching to a final state, viz. r, in one of the possibilities, we may say that the string is accepted. So, by considering this notion of language acceptance, the language accepted by the finite automaton can be quickly reported as the set of all strings with the last but one symbol as b, i.e.

$$\Big\{xb(a+b) \;\Big|\; x \in (a+b)^*\Big\}.$$

Thus, the corresponding regular expression is $(a+b)^*b(a+b)$.

This type of automaton with some additional features is known as nondeterministic finite automaton. This concept will formally be introduced in the following section.

4.2 Nondeterministic Finite Automata

In contrast to a DFA, where we have a unique next state for a transition from a state on an input symbol, now we consider a finite automaton with nondeterministic transitions. A transition is nondeterministic if there are several (possibly zero) next states from a state on an input symbol or without any input. A transition without input is called as *ε-transition*. A nondeterministic finite automaton is defined in the similar lines of a DFA in which transitions may be nondeterministic.

Formally, a *nondeterministic finite automaton* (NFA) is a quintuple $\mathscr{N} = (Q, \Sigma, \delta, q_0, F)$, where Q, Σ, q_0 and F are as in a DFA; whereas, the transition function δ is as below:

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \longrightarrow \wp(Q)$$

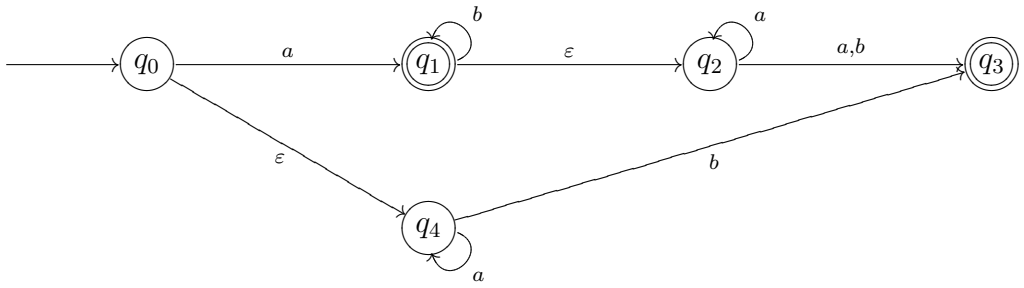
is a function so that, for a given state and an input symbol (possibly ε), δ assigns a set of next states, possibly empty set.

Remark 4.2.1. Clearly, every DFA can be treated as an NFA.

Example 4.2.2. Let $Q = \{q_0, q_1, q_2, q_3, q_4\}$, $\Sigma = \{a, b\}$, $F = \{q_1, q_3\}$ and δ be given by the following transition table.

δ	a	b	ε
q_0	$\{q_1\}$	$\varnothing$	$\{q_4\}$
q_1	$\varnothing$	$\{q_1\}$	$\{q_2\}$
q_2	$\{q_2, q_3\}$	$\{q_3\}$	$\varnothing$
q_3	$\varnothing$	$\varnothing$	$\varnothing$
q_4	$\{q_4\}$	$\{q_3\}$	$\varnothing$

The quintuple $\mathscr{N} = (Q, \Sigma, \delta, q_0, F)$ is an NFA. In the similar lines of a DFA, an NFA can be represented by a state transition diagram. For instance, the present NFA can be represented as follows:



Note the following few nondeterministic transitions in this NFA.

1. There is no transition from q_0 on input symbol b.
2. There are multiple (two) transitions from q_2 on input symbol a.
3. There is a transition from q_0 to q_4 without any input, i.e. ε-transition.

Consider the traces for the string ab from the state q_0. Clearly, the following four are the possible traces.

(i) $q_0 \xrightarrow{a} q_1 \xrightarrow{b} q_1$

(ii) $q_0 \xrightarrow{\varepsilon} q_4 \xrightarrow{a} q_4 \xrightarrow{b} q_3$

(iii) $q_0 \xrightarrow{a} q_1 \xrightarrow{\varepsilon} q_2 \xrightarrow{b} q_3$

(iv) $q_0 \xrightarrow{a} q_1 \xrightarrow{b} q_1 \xrightarrow{\varepsilon} q_2$

Note that three distinct states, viz. q_1, q_2 and q_3 are reachable from q_0 via the string ab. That means, while tracing a path from q_0 for ab we consider possible insertion of ε in ab, wherever ε-transitions are defined. For example, in trace (ii) we have included an ε-transition from q_0 to q_4, considering ab as εab, as it is defined. Whereas, in trace (iii) we consider ab as $a\varepsilon b$. It is clear that, if we process the input string ab at the state q_0, then the set of next states is $\{q_1, q_2, q_3\}$.

Definition 4.2.3. Let $\mathscr{N} = (Q, \Sigma, \delta, q_0, F)$ be an NFA. Given an input string $x = a_1 a_2 \cdots a_k$ and a state p of $\mathscr{N}$, the set of next states $\hat{\delta}(p, x)$ can be easily computed using a tree structure, called a *computation tree* of $\hat{\delta}(p, x)$, which is defined in the following way:

1. p is the root node

2. Children of the root are precisely those nodes which are having transitions from p via ε or a_1.

3. For any node, whose branch (from the root) is labeled $a_1a_2\cdots a_i$ (as a resultant string by possible insertions of ε), its children are precisely those nodes having transitions via ε or a_{i+1}.

4. If there is a final state whose branch from the root is labeled x (as a resultant string), then mark the node by a tick mark ✓.

5. If the label of the branch of a leaf node is not the full string x, i.e. some proper prefix of x, then it is marked by a cross ✗ – indicating that the branch has reached to a dead-end before completely processing the string x.

Example 4.2.4. The computation tree of $\hat{\delta}(q_0, ab)$ in the NFA given in Example 4.2.2 is shown in Figure 4.2

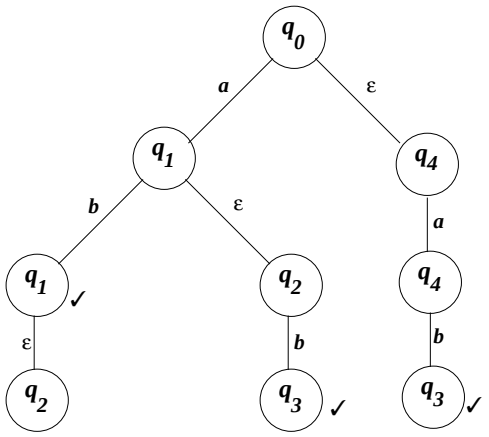


Figure 4.2: Computation Tree of $\hat{\delta}(q_0, ab)$

Example 4.2.5. The computation tree of $\hat{\delta}(q_0, abb)$ in the NFA given in Example 4.2.2 is shown in Figure 4.3. In which, notice that the branch $q_0 - q_4 - q_4 - q_3$ has the label ab, as a resultant string, and as there are no further transitions at q_3, the branch has got terminated without completely processing the string abb. Thus, it is indicated by marking a cross ✗ at the end of the branch.

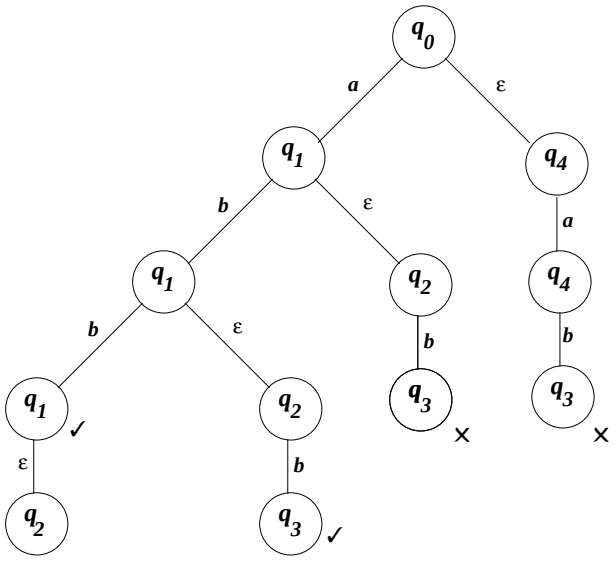


Figure 4.3: Computation Tree of $\hat{\delta}(q_0, abb)$

As the automaton given in Example 4.2.2 is nondeterministic, if a string is processed at a state, then there may be multiple next states (unlike DFA), possibly empty. For example, if we apply the string bba at the state q_0, then the only possible way to process the first b is going via ε-transition from q_0 to q_4 and then from q_4 to q_3 via b. As there are no transitions from q_3, the string cannot be processed further. Hence, the set of next states for the string bba at q_0 is empty.

Thus, given a string $x = a_1a_2\cdots a_n$ and a state p, by treating x as $\varepsilon a_1\varepsilon a_2\varepsilon\cdots\varepsilon a_n\varepsilon$ and by looking at the possible complete branches starting at p, we find the set of next states for p via x. To introduce the notion of $\hat{\delta}$ in an NFA, we first introduce the notion called ε-closure of a state.

Definition 4.2.6. An *ε-closure* of a state p, denoted by $E(p)$ is defined as the set of all states that are reachable from p via zero or more ε-transitions.

Example 4.2.7. In the following we enlist the ε-closures of all the states of the NFA given in Example 4.2.2.
$E(q_0) = \{q_0, q_4\}; E(q_1) = \{q_1, q_2\}; E(q_2) = \{q_2\}; E(q_3) = \{q_3\}; E(q_4) = \{q_4\}$.

Further, for a set A of states, ε-closure of A, denoted by $E(A)$ is defined as

$$E(A) = \bigcup_{p\in A} E(p).$$

Now we are ready to formally define the set of next states for a state via a string.

Definition 4.2.8. Define the function

$$\hat{\delta} : Q \times \Sigma^* \longrightarrow \wp(Q)$$

by

1. $\hat{\delta}(q, \varepsilon) = E(q)$ and
2. $\hat{\delta}(q, xa) = E\Big(\bigcup_{p \in \hat{\delta}(q,x)} \delta(p, a) \Big)$

Definition 4.2.9. A string $x \in \Sigma^*$ is said to be accepted by an NFA $\mathscr{N} = (Q, \Sigma, \delta, q_0, F)$, if

$$\hat{\delta}(q_0, x) \cap F \neq \varnothing.$$

That is, in the computation tree of (q_0, x) there should be final state among the nodes marked with ✓. Thus, the language accepted by $\mathscr{N}$ is

$$L(\mathscr{N}) = \Big\{ x \in \Sigma^* \Big| \ \hat{\delta}(q_0, x) \cap F \neq \varnothing \Big\}.$$

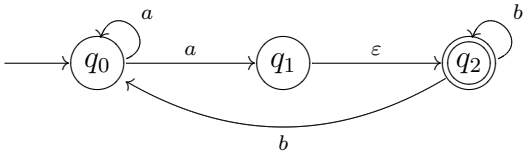
Example 4.2.10. Note that q_1 and q_3 are the final states for the NFA given in Example 4.2.2.

1. The possible ways of reaching q_1 from the initial state q_0 is via the set of strings represented by the regular expression ab^*.
2. The are two possible ways of reaching q_3 from q_0 as discussed below.
   (a) Via the state q_1: With the strings of ab^* we can clearly reach q_1 from q_0, then using the ε-transition we can reach q_2. Then the strings of $a^*(a + b)$ will lead us from state q_2 to q_3. Thus, the set of string in this case can be represented by $ab^*a^*(a + b)$.
   (b) Via the state q_4: Initially, we use ε-transition to reach q_4 from q_0, then clearly the strings of a^*b will precisely be useful to reach from q_4 to q_3. And hence ab^* itself represent the set of strings in this case.

   Thus, the language accepted by the NFA can be represented by the following regular expression:

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

Example 4.2.11. Consider the following NFA.



1. From the initial state q_0 one can reach back to q_0 via strings from a^* or from $a\varepsilon b^*b$, i.e. ab^+, or via a string which is a mixture of strings from the above two sets. That is, the strings of $(a + ab^+)^*$ will lead us from q_0 to q_0.

2. Also, note that the strings of ab^* will lead us from the initial state q_0 to the final state q_2.

3. Thus, any string accepted by the NFA can be of the form – a string from the set $(a + ab^+)^*$ followed by a string from the set ab^*.

Hence, the language accepted by the NFA can be represented by

$$(a + ab^+)^*ab^*$$

4.3 Equivalence of NFA and DFA

Two finite automata $\mathscr{A}$ and $\mathscr{A}'$ are said to be equivalent if they accept the same language, i.e. $L(\mathscr{A}) = L(\mathscr{A}')$. In the present context, although NFA appears more general with a lot of flexibility, we prove that NFA and DFA accept the same class of languages.

Since every DFA can be treated as an NFA, one side is obvious. The converse, given an NFA there exists an equivalent DFA, is being proved through the following two lemmas.

Lemma 4.3.1. *Given an NFA in which there are some ε-transitions, there exists an equivalent NFA without ε-transitions.*

Proof. Let $\mathscr{N} = (Q, \Sigma, \delta, q_0, F)$ be a finite automaton in which there are some ε-transitions. Set $\mathscr{N}' = (Q, \Sigma, \delta', q_0, F')$, where $\delta' : Q \times \Sigma \longrightarrow \wp(Q)$ is given by

$$\delta'(q, a) = \hat{\delta}(q, a) \quad \text{for all } a \in \Sigma,\ q \in Q$$

and $F' = F \cup \{q \in Q \mid E(q) \cap F \neq \varnothing\}$. It is clear that $\mathscr{N}'$ is a finite automaton without ε-transitions.

We claim that $L(\mathscr{N}) = L(\mathscr{N}')$. First note that

$$\begin{aligned}
\varepsilon \in L(\mathscr{N}) &\iff \hat{\delta}(q_0, \varepsilon) \cap F \neq \varnothing \\
&\iff E(q_0) \cap F \neq \varnothing \\
&\iff q_0 \in F' \\
&\iff \varepsilon \in L(\mathscr{N}').
\end{aligned}$$

Now, for any $x \in \Sigma^+$, we prove that $\hat{\delta}'(q_0, x) = \hat{\delta}(q_0, x)$ so that $L(\mathscr{N}) = L(\mathscr{N}')$. This is because, if $\hat{\delta}'(q_0, x) = \hat{\delta}(q_0, x)$, then

$$\begin{aligned}
x \in L(\mathscr{N}) &\iff \hat{\delta}(q_0, x) \cap F \neq \varnothing \\
&\iff \hat{\delta}'(q_0, x) \cap F \neq \varnothing \\
&\implies \hat{\delta}'(q_0, x) \cap F' \neq \varnothing \\
&\iff x \in L(\mathscr{N}').
\end{aligned}$$

Thus, $L(\mathscr{N}) \subseteq L(\mathscr{N}')$.

Conversely, let $x \in L(\mathscr{N}')$, i.e. $\hat{\delta}'(q_0, x) \cap F' \neq \varnothing$. If $\hat{\delta}'(q_0, x) \cap F \neq \varnothing$ then we are through. Otherwise, there exists $q \in \hat{\delta}'(q_0, x)$ such that $E(q) \cap F \neq \varnothing$. This implies that $q \in \hat{\delta}(q_0, x)$ and $E(q) \cap F \neq \varnothing$. Which in turn implies that $\hat{\delta}(q_0, x) \cap F \neq \varnothing$. That is, $x \in L(\mathscr{N})$. Hence $L(\mathscr{N}) = L(\mathscr{N}')$.

So, it is enough to prove that $\hat{\delta}'(q_0, x) = \hat{\delta}(q_0, x)$ for each $x \in \Sigma^+$. We prove this by induction on $|x|$. If $x \in \Sigma$, then by definition of δ' we have

$$\hat{\delta}'(q_0, x) = \delta'(q_0, x) = \hat{\delta}(q_0, x).$$

Assume the result for all strings of length less than or equal to m. For $a \in \Sigma$, let $|xa| = m + 1$. Since $|x| = m$, by inductive hypothesis,

$$\hat{\delta}'(q_0, x) = \hat{\delta}(q_0, x).$$

Now

$$\begin{aligned}
\hat{\delta}'(q_0, xa) &= \delta'(\hat{\delta}'(q_0, x), a) \\
&= \bigcup_{p \in \hat{\delta}'(q_0, x)} \delta'(p, a) \\
&= \bigcup_{p \in \hat{\delta}(q_0, x)} \hat{\delta}(p, a) \\
&= \hat{\delta}(q_0, xa).
\end{aligned}$$

Hence by induction we have $\delta'(q_0, x) = \hat{\delta}(q_0, x) \quad \forall x \in \Sigma^+$. This completes the proof. □

Lemma 4.3.2. *For every NFA $\mathscr{N}'$ without ε-transitions, there exists a DFA $\mathscr{A}$ such that $L(\mathscr{N}') = L(\mathscr{A})$.*

Proof. Let $\mathscr{N}' = (Q, \Sigma, \delta', q_0, F')$ be an NFA without ε-transitions. Construct $\mathscr{A} = (P, \Sigma, \mu, p_0, E)$, where

$$P = \Big\{ p_{\{i_1,\ldots,i_k\}} \mid \{q_{i_1}, \ldots, q_{i_k}\} \subseteq Q \Big\},$$

$$p_0 = p_{\{0\}},$$

$$E = \Big\{ p_{\{i_1,\ldots,i_k\}} \in P \mid \{q_{i_1}, \ldots, q_{i_k}\} \cap F' \neq \varnothing \Big\}, \text{ and}$$

$\mu : P \times \Sigma \longrightarrow P$ defined by

$$\mu(p_{\{i_1,\ldots,i_k\}}, a) = p_{\{j_1,\ldots,j_m\}} \text{ if and only if } \bigcup_{i_l \in \{i_1,\ldots,i_k\}} \delta'(q_{i_l}, a) = \{q_{j_1}, \ldots, q_{j_m}\}.$$

Clearly, $\mathscr{A}$ is a DFA. To show $L(\mathscr{N}') = L(\mathscr{A})$, we prove that, for all $x \in \Sigma^*$,

$$\hat{\mu}(p_0, x) = p_{\{i_1,\ldots,i_k\}} \quad \text{if and only if} \quad \hat{\delta}'(q_0, x) = \{q_{i_1}, \ldots, q_{i_k}\}. \qquad (\star)$$

This suffices the result, because

$$\begin{aligned} x \in L(\mathscr{N}') &\iff \hat{\delta}'(q_0, x) \cap F' \neq \varnothing \\ &\iff \hat{\mu}(p_0, x) \in E \\ &\iff x \in L(\mathscr{A}). \end{aligned}$$

Now, we prove the statement in $(\star)$ by induction on the length of x.

If $|x| = 0$ then $\hat{\delta}'(q_0, x) = \{q_0\}$, also $\hat{\mu}(p_0, x) = p_0 = p_{\{0\}}$ so that the statement in $(\star)$ holds in this case. (In case $x = 1$ also, one may notice that the statement directly follows from the definition of μ.)

Assume that the statement is true for the strings whose length is less than or equal to n. Let $x \in \Sigma^*$ with $|x| = n$ and $a \in \Sigma$. Then

$$\hat{\mu}(p_0, xa) = \mu(\hat{\mu}(p_0, x), a).$$

By inductive hypothesis,

$$\hat{\mu}'(p_0, x) = p_{\{i_1,\ldots,i_k\}} \quad \text{if and only if} \quad \hat{\delta}'(q_0, x) = \{q_{i_1}, \ldots, q_{i_k}\}.$$

Now by definition of μ,

$$\mu(p_{\{i_1,\ldots,i_k\}}, a) = p_{\{j_1,\ldots,j_m\}} \text{ if and only if } \bigcup_{i_l \in \{i_1,\ldots,i_k\}} \delta'(q_{i_l}, a) = \{q_{j_1}, \ldots, q_{j_m}\}.$$

Thus,

$$\hat{\mu}(p_0, xa) = p_{\{j_1,\ldots,j_m\}} \quad \text{if and only if} \quad \hat{\delta}'(q_0, xa) = \{q_{j_1}, \ldots, q_{j_m}\}.$$

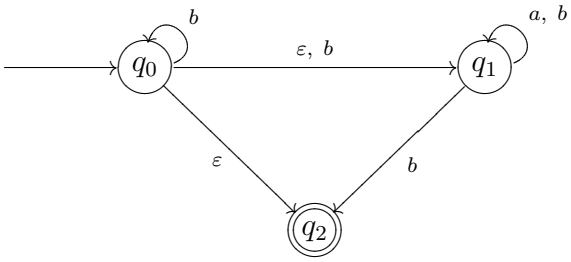
Hence, by induction, the statement in $(\star)$ is true. □

Thus, we have the following theorem.

Theorem 4.3.3. *For every NFA there exists an equivalent DFA.*

We illustrate the constructions for Lemma 4.3.1 and Lemma 4.3.2 through the following example.

Example 4.3.4. Consider the following NFA



That is, the transition function, say δ, can be displayed as the following table

δ	a	b	ε
q_0	$\varnothing$	$\{q_0, q_1\}$	$\{q_1, q_2\}$
q_1	$\{q_1\}$	$\{q_1, q_2\}$	$\varnothing$
q_2	$\varnothing$	$\varnothing$	$\varnothing$

In order to remove ε-transitions and obtain an equivalent finite automaton, we need to calculate $\hat{\delta}(q, a)$, for all states q and for all input symbols a. That is given in the following table.

$\delta' = \hat{\delta}$	a	b
q_0	$\{q_1\}$	$\{q_0, q_1, q_2\}$
q_1	$\{q_1\}$	$\{q_1, q_2\}$
q_2	$\varnothing$	$\varnothing$

Thus, $(Q = \{q_0, q_1, q_2\}, \{a, b\}, \delta', q_0, \{q_0, q_2\})$ is an equivalent NFA in which there are no ε-transitions. Now, to convert this to an equivalent DFA, we consider each subset of Q as a state in the DFA and its transition function μ assigns the union of respective transitions at each state of the above NFA

which are in the subset under consideration. That is, for any input symbol a and a subset X of $\{0, 1, 2\}$ (the index set of the states)

$$\mu(p_X, a) = \bigcup_{i \in X} \delta'(q_i, a).$$

Thus the resultant DFA is

$$(P, \{a, b\}, \mu, p_{\{0\}}, E)$$

where
$P = \left\{ p_X \;\middle|\; X \subseteq \{0, 1, 2\} \right\}$,
$E = \left\{ p_X \middle|\; X \cap \{0, 2\} \neq \varnothing \right\}$; in fact there are six states in E, and the transition map μ is given in the following table:

μ	a	b
$p_\varnothing$	$p_\varnothing$	$p_\varnothing$
$p_{\{0\}}$	$p_{\{1\}}$	$p_{\{0,1,2\}}$
$p_{\{1\}}$	$p_{\{1\}}$	$p_{\{1,2\}}$
$p_{\{2\}}$	$p_\varnothing$	$p_\varnothing$
$p_{\{0,1\}}$	$p_{\{1\}}$	$p_{\{0,1,2\}}$
$p_{\{1,2\}}$	$p_{\{1\}}$	$p_{\{1,2\}}$
$p_{\{0,2\}}$	$p_{\{1\}}$	$p_{\{0,1,2\}}$
$p_{\{0,1,2\}}$	$p_{\{1\}}$	$p_{\{0,1,2\}}$

It is observed that while converting an NFA $\mathscr{N}$ to an equivalent DFA $\mathscr{A}$, the number of states of $\mathscr{A}$ has an exponential growth compared to that of $\mathscr{N}$. In fact, if $\mathscr{N}$ has n states, $\mathscr{A}$ will have 2^n states. It may also be noted that, among these 2^n states, some of the states are dead – the states which are either inaccessible from the initial state or no final state is accessible from them. If there is a procedure to remove some of the dead states which do not alter the language, then we may get a DFA with lesser number of states.

In the following we provide certain heuristics to convert a given NFA with n states to an equivalent DFA with number of states less than 2^n.

4.3.1 Heuristics to Convert NFA to DFA

We demonstrate certain heuristics to convert an NFA without ε-transitions to its equivalent DFA. Nondeterminism in a finite automaton without ε-transitions is clearly because of multiple transitions at a state for an input symbol. That is, for a state q and an input symbol a, if $\delta(q, a) = P$ with $|P| = 0$ or $|P| > 1$, then such a situation need to be handled to avoid

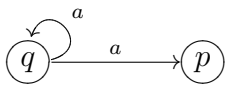
nondeterminism at q. To do this, we propose the following techniques and illustrate them through examples.

Case 1: $|P| = 0$. In this case, there is no transition from q on a. We create a new (trap) state t and give transition from q to t via a. For all input symbols, the transitions from t will be assigned to itself. For all such nondeterministic transitions in the finite automaton, creating only one new trap state will be sufficient.

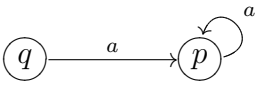
Case 2: $|P| > 1$. Let $P = \{p_1, p_2, \ldots, p_k\}$.

If $q \notin P$, then choose a new state p and assign a transition from q to p via a. Now all the transitions from $p_1, p_2, \ldots, p_k$ are assigned to p. This avoids nondeterminism at q; but, there may be an occurrence of nondeterminism on the new state p. One may successively apply this heuristic to avoid nondeterminism further.

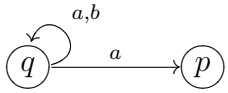
If $q \in P$, then the heuristic mentioned above can be applied for all the transitions except for the one reaching to q, i.e. first remove q from P and work as mentioned above. When there is no other loop at q, the resultant scenario with the transition from q to q is depicted as under:



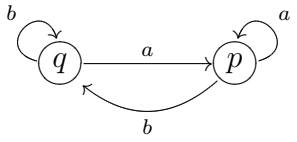
This may be replaced by the following type of transition:



In case there is another loop at q, say with the input symbol b, then scenario will be as under:

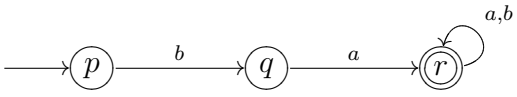


And in which case, to avoid the nondeterminism at q, we suggest the equivalent transitions as shown below:

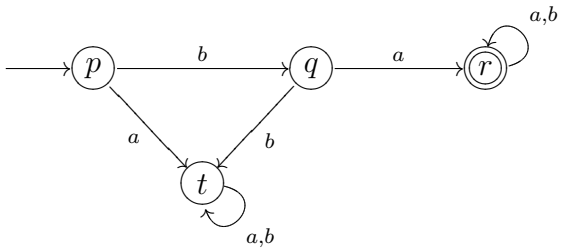


We illustrate these heuristics in the following examples.

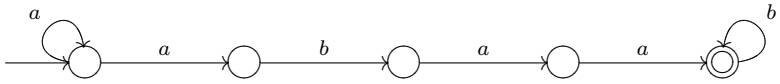
Example 4.3.5. For the language over $\{a, b\}$ which contains all those strings starting with ba, it is easy to construct an NFA as given below.



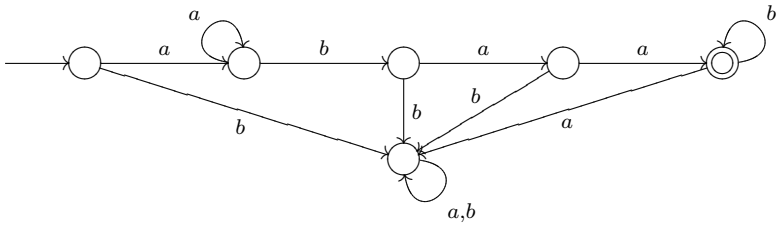
Clearly, the language accepted by the above NFA is $\{bax \mid x \in a, b^*\}$. Now, by applying the heuristics one may propose the following DFA for the same language.



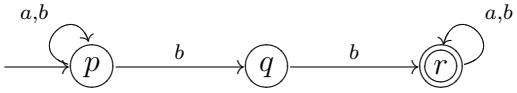
Example 4.3.6. One can construct the following NFA for the language $\{a^n x b^m \mid x = baa, n \geq 1 \text{ and } m \geq 0\}$.



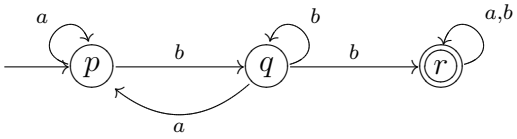
By applying the heuristics, the following DFA can be constructed easily.



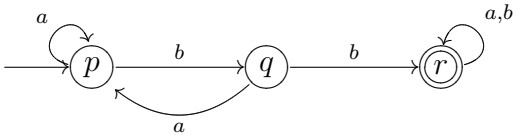
Example 4.3.7. We consider the following NFA, which accepts the language $\{x \in \{a, b\}^* \mid bb \text{ is a substring of } x\}$.



By applying a heuristic discussed in Case 2, as shown in the following finite automaton, we can remove nondeterminism at p, whereas we get nondeterminism at q, as a result.



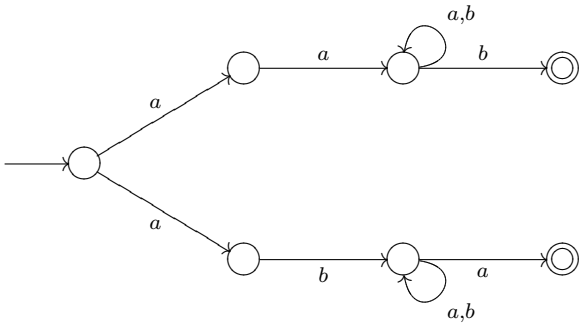
To eliminate the nondeterminism at q, we further apply the same technique and get the following DFA.



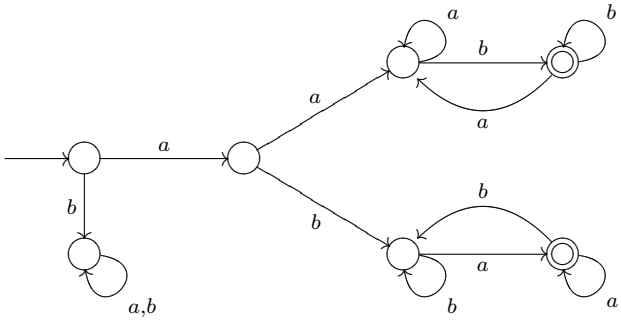
Example 4.3.8. Consider the language L over $\{a, b\}$ containing those strings x with the property that either x starts with aa and ends with b, or x starts with ab and ends with a. That is,

$$L = \{aaxb \mid x \in \{a, b\}^*\} \cup \{abya \mid y \in \{a, b\}^*\}.$$

An NFA shown below can easily be constructed for the language L.



By applying the heuristics on this NFA, the following DFA can be obtained, which accepts L.



4.4 Minimization of DFA

In this section we provide an algorithmic procedure to convert a given DFA to an equivalent DFA with minimum number of states. In fact, this assertion is obtained through well-known Myhill-Nerode theorem which gives a characterization of the languages accepted by DFA.

4.4.1 Myhill-Nerode Theorem

We start with the following basic concepts.

Definition 4.4.1. An equivalence relation $\sim$ on Σ^* is said to be *right invariant* if, for $x, y \in \Sigma^*$,

$$x \sim y \Longrightarrow \forall z(xz \sim yz).$$

Example 4.4.2. Let L be a language over Σ. Define the relation $\sim_L$ on Σ^* by

$$x \sim_L y \text{ if and only if } \forall z(xz \in L \Longleftrightarrow yz \in L).$$

It is straightforward to observe that $\sim_L$ is an equivalence relation on Σ^*. For $x, y \in \Sigma^*$ assume $x \sim_L y$. Let $z \in \Sigma^*$ be arbitrary. Claim: $xz \sim_L yz$. That is, we have to show that $(\forall w)\Big(xzw \in L \Longleftrightarrow yzw \in L\Big)$. For $w \in \Sigma^*$, write $u = zw$. Now, since $x \sim_L y$, we have $xu \in L \Longleftrightarrow yu \in L$, i.e. $xzw \in L \Longleftrightarrow yzw \in L$. Hence $\sim_L$ is a right invariant equivalence on Σ^*.

Example 4.4.3. Let $\mathscr{A} = (Q, \Sigma, \delta, q_0, F)$ be a DFA. Define the relation $\sim_{\mathscr{A}}$ on Σ^* by

$$x \sim_{\mathscr{A}} y \text{ if and only if } \hat{\delta}(q_0, x) = \hat{\delta}(q_0, y).$$

Clearly, $\sim_{\mathscr{A}}$ is an equivalence relation on Σ^*. Suppose $x \sim_{\mathscr{A}} y$, i.e. $\hat{\delta}(q_0, x) = \hat{\delta}(q_0, y)$. Now, for any $z \in \Sigma^*$,

$$\begin{aligned} \hat{\delta}(q_0, xz) &= \hat{\delta}(\hat{\delta}(q_0, x), z) \\ &= \hat{\delta}(\hat{\delta}(q_0, y), z) \\ &= \hat{\delta}(q_0, yz) \end{aligned}$$

so that $xz \sim_{\mathscr{A}} yz$. Hence, $\sim_{\mathscr{A}}$ is a right invariant equivalence relation.

Theorem 4.4.4 (Myhill-Nerode Theorem)**.** *Let L be a language over Σ. The following three statements regarding L are equivalent:*

1. *L is accepted by a DFA.*
2. *There exists a right invariant equivalence relation $\sim$ of finite index on Σ^* such that L is the union of some the equivalence classes of $\sim$.*
3. *The equivalence relation $\sim_L$ is of finite index.*

Proof. (1) $\Rightarrow$ (2): Assume L is accepted by the DFA $\mathscr{A} = (Q, \Sigma, \delta, q_0, F)$. First, we observe that the right invariant equivalence relation $\sim_{\mathscr{A}}$ is of finite index. For $x \in \Sigma^*$, if $\hat{\delta}(q_0, x) = p$, then the equivalence class containing x

$$[x]_{\sim_{\mathscr{A}}} = \{y \in \Sigma^* \mid \hat{\delta}(q_0, y) = p\}.$$

That is, given $q \in Q$, the set

$$C_q = \{x \in \Sigma^* \mid \hat{\delta}(q_0, x) = q\}$$

is an equivalence class (possibly empty, whenever q is not reachable from q_0) of $\sim_{\mathscr{A}}$. Thus, the equivalence classes of $\sim_{\mathscr{A}}$ are completely determined by the states of $\mathscr{A}$; moreover, the number of equivalence classes of $\sim_{\mathscr{A}}$ is less than or equal to the number of states of $\mathscr{A}$. Hence, $\sim_{\mathscr{A}}$ is of finite index. Now,

$$\begin{aligned} L &= \{x \in \Sigma^* \mid \hat{\delta}(q_0, x) \in F\} \\ &= \bigcup_{p \in F} \{x \in \Sigma^* \mid \hat{\delta}(q_0, x) = p\} \\ &= \bigcup_{p \in F} C_p. \end{aligned}$$

as desired.

(2) $\Rightarrow$ (3): Suppose $\sim$ is an equivalence with the criterion given in (2). We show that $\sim$ is a refinement of $\sim_L$ so that the number of equivalence classes of $\sim_L$ is less than or equal to the number of equivalence classes of $\sim$. For $x, y \in \Sigma^*$, suppose $x \sim y$. To show $x \sim_L y$, we have to show that $\forall z(xz \in L \iff yz \in L)$. Since $\sim$ is right invariant, we have $xz \sim yz$, for all z, i.e. xz and yz are in same equivalence class of $\sim$. As L is the union of some of the equivalence classes of $\sim$, we have

$$\forall z(xz \in L \iff yz \in L).$$

Thus, $\sim_L$ is of finite index.

(3) $\Rightarrow$ (1): Assume $\sim_L$ is of finite index. Construct

$$\mathscr{A}_L = (Q, \Sigma, \delta, q_0, F)$$

by setting

$Q = \Sigma^*/_{\sim_L} = \left\{[x] \;\middle|\; x \in \Sigma^*\right\}$, the set of all equivalence classes of Σ^* with respect to $\sim_L$,

$q_0 = [\varepsilon]$,

$F = \left\{[x] \in Q \;\middle|\; x \in L\right\}$ and

$\delta : Q \times \Sigma \longrightarrow Q$ is defined by $\delta([x], a) = [xa]$, for all $[x] \in Q$ and $a \in \Sigma$.

Since $\sim_L$ is of finite index, Q is a finite set; further, for $[x], [y] \in Q$ and $a \in \Sigma$,

$$[x] = [y] \Longrightarrow x \sim_L y \Longrightarrow xa \sim_L ya \Longrightarrow [xa] = [ya]$$

so that δ is well-defined. Hence $\mathscr{A}_L$ is a DFA. We claim that $L(\mathscr{A}_L) = L$. In fact, we show that

$$\hat{\delta}(q_0, w) = [w], \quad \forall w \in \Sigma^*;$$

this serves the purpose because $w \in L \iff [w] \in F$. We prove this by induction on $|w|$. Induction basis is clear, because $\hat{\delta}(q_0, \varepsilon) = q_0 = [\varepsilon]$. Also, by definition of δ, we have $\hat{\delta}(q_0, a) = \delta([\varepsilon], a) = [\varepsilon a] = [a]$, for all $a \in \Sigma$.

For inductive step, consider $x \in \Sigma^*$ and $a \in \Sigma$. Now

$$\begin{aligned} \hat{\delta}(q_0, xa) &= \delta(\hat{\delta}(q_0, x), a) \\ &= \delta([x], a) \text{ (by inductive hypothesis)} \\ &= [xa]. \end{aligned}$$

This completes the proof. □

Remark 4.4.5. Note that, the proof of (2) ⇒ (3) shows that the number of states of any DFA accepting L is greater than or equal to the index of $\sim_L$ and the proof of (3) ⇒ (1) provides us the DFA $\mathscr{A}_L$ with number of states equal to the index of $\sim_L$. Hence, $\mathscr{A}_L$ is a minimum state DFA accepting L.

Example 4.4.6. Consider the language $L = \{x \in \{a, b\}^* \mid ab \text{ is a substring of } x\}$. We calculate the equivalence classes of $\sim_L$. First observe that the strings ε, a and ab are not equivalent to each other, because of the following.

1. The string b distinguishes the pair ε and a: $\varepsilon b = b \notin L$, whereas $ab \in L$.
2. Any string which does not contain ab distinguishes ε and ab. For instance, $\varepsilon b = b \notin L$, whereas $abb \in L$.
3. Also, a and ab are not equivalent because, $aa \notin L$, whereas $aba \in L$.

Thus, the strings ε, a and ab will be in three different equivalence classes, say $[\varepsilon], [a]$ and $[ab]$, respectively. Now, we show that any other $x \in \{a, b\}^*$ will be in one of these equivalence classes.

- In case ab is a substring of x, clearly x will be in $[ab]$.
- If ab is not a substring of x, then we discuss in the following subcases.
  - For $n \geq 1$, $x = a^n$: In this case, x will be in $[a]$.
  - In case $x = b^n$, for some $n \geq 1$, x will be in $[\varepsilon]$.

– If x has some a's and some b's, then x must be of the form $b^n a^m$, for $n, m \geq 1$. In this case, x will be in $[a]$.

Thus, $\sim_L$ has exactly three equivalence classes and hence it is of finite index. By Myhill-Nerode theorem, there exists a DFA accepting L.

Example 4.4.7. Consider the language $L = \{a^n b^n \mid n \geq 1\}$ over the alphabet $\{a, b\}$. We show that the index of $\sim_L$ is not finite. Hence, by Myhill-Nerode theorem, there exists no DFA accepting L. For instance, consider $a^n, a^m \in \{a, b\}^*$, for $m \neq n$. They are not equivalent with respect to $\sim_L$, because, for $b^n \in \{a, b\}^*$, we have

$$a^n b^n \in L, \quad \text{whereas} \quad a^m b^n \notin L.$$

Thus, for each n, there has to be one equivalence classes to accommodate a^n. Hence, the index of $\sim_L$ is not finite.

4.4.2 Algorithmic Procedure for Minimization

Given a DFA, there may be certain states which are redundant in the sense that their roles in the language acceptance are same. Here, two states are said to have same role, if it will lead us to either both final states or both non-final states for every input string; so that they contribute in same manner in language acceptance. Among such group of states, whose roles are same, only one state can be considered and others can be discarded to reduce the number states without affecting the language. Now, we formulate this idea and present an algorithmic procedure to minimize the number of states of a DFA. In fact, we obtain an equivalent DFA whose number of states is minimum.

Definition 4.4.8. Two states p and q of a DFA $\mathscr{A}$ are said to be equivalent, denoted by $p \equiv q$, if for all $x \in \Sigma^*$ both the states $\delta(p, x)$ and $\delta(q, x)$ are either final states or non-final states.

Clearly, $\equiv$ is an equivalence relation on the set of states of $\mathscr{A}$. Given two states p and q, to test whether $p \equiv q$ we need to check the condition for all strings in Σ^*. This is practically difficult, since Σ^* is an infinite set. So, in the following, we introduce a notion called k-equivalence and build up a technique to test the equivalence of states via k-equivalence.

Definition 4.4.9. For $k \geq 0$, two states p and q of a DFA $\mathscr{A}$ are said to be k-equivalent, denoted by $p \overset{k}{\equiv} q$, if for all $x \in \Sigma^*$ with $|x| \leq k$ both the states $\delta(p, x)$ and $\delta(q, x)$ are either final states or non-final states.

Clearly, $\overset{k}{\equiv}$ is also an equivalence relation on the set of states of $\mathscr{A}$. Since there are only finitely man strings of length up to k over an alphabet, one may easily test the k-equivalence between the states. Let us denote the partition of Q under the relation $\equiv$ by $\prod$, whereas it is $\prod_k$ under the relation $\overset{k}{\equiv}$.

Remark 4.4.10. For any $p, q \in Q$,

$$p \equiv q \text{ if and only if } p \overset{k}{\equiv} q \text{ for all } k \geq 0.$$

Also, for any $k \geq 1$ and $p, q \in Q$, if $p \overset{k}{\equiv} q$, then $p \overset{k-1}{\equiv} q$.

Given a k-equivalence relation over the set of states, the following theorem provides us a criterion to calculate the $(k+1)$-equivalence relation.

Theorem 4.4.11. *For any $k \geq 0$ and $p, q \in Q$,*

$$p \overset{k+1}{\equiv} q \text{ if and only if } p \overset{0}{\equiv} q \text{ and } \delta(p,a) \overset{k}{\equiv} \delta(q,a)\ \forall a \in \Sigma.$$

Proof. If $p \overset{k+1}{\equiv} q$ then $p \overset{k}{\equiv} q$ holds (cf. Remark 4.4.10). Let $x \in \Sigma^*$ with $|x| \leq k$ and $a \in \Sigma$ be arbitrary. Then since $p \overset{k+1}{\equiv} q$, $\delta(\delta(p,a),x)$ and $\delta(\delta(q,a),x)$ both are either final states or non-final states, so that $\delta(p,a) \overset{k}{\equiv} \delta(q,a)$.

Conversely, for $k \geq 0$, suppose $p \overset{0}{\equiv} q$ and $\delta(p,a) \overset{k}{\equiv} \delta(q,a)\ \forall a \in \Sigma$. Note that for $x \in \Sigma^*$ with $|x| \leq k$ and $a \in \Sigma$, $\delta(\delta(p,a),x)$ and $\delta(\delta(q,a),x)$ both are either final states or non-final states, i.e. for all $y \in \Sigma^*$ and $1 \leq |y| \leq k+1$, both the states $\delta(p,y)$ and $\delta(q,y)$ are final or non-final states. But since $p \overset{0}{\equiv} q$ we have $p \overset{k+1}{\equiv} q$. □

Remark 4.4.12. Two k-equivalent states p and q will further be $(k+1)$-equivalent if $\delta(p,a) \overset{k}{\equiv} \delta(q,a)\ \forall a \in \Sigma$, i.e. from $\prod_k$ we can obtain $\prod_{k+1}$.

Theorem 4.4.13. *If $\prod_k = \prod_{k+1}$, for some $k \geq 0$, then $\prod_k = \prod$.*

Proof. Suppose $\prod_k = \prod_{k+1}$. To prove that, for $p, q \in Q$,

$$p \overset{k+1}{\equiv} q \implies p \overset{n}{\equiv} q \ \ \forall n \geq 0$$

it is enough to prove that

$$p \overset{k+1}{\equiv} q \implies p \overset{k+2}{\equiv} q.$$

Now,

$$\begin{aligned} p \overset{k+1}{\equiv} q &\implies \delta(p,a) \overset{k}{\equiv} \delta(q,a)\ \forall a \in \Sigma \\ &\implies \delta(p,a) \overset{k+1}{\equiv} \delta(q,a)\ \forall a \in \Sigma \\ &\implies p \overset{k+2}{\equiv} q \end{aligned}$$

Hence the result follows by induction. □

Using the above results, we illustrate calculating the partition $\prod$ for the following DFA.

Example 4.4.14. Consider the DFA given in following transition table.

δ	a	b
$\rightarrow q_0$	q_3	q_2
q_1	q_6	q_2
q_2	q_8	q_5
(q_3)	q_0	q_1
(q_4)	q_2	q_5
q_5	q_4	q_3
(q_6)	q_1	q_0
q_7	q_4	q_6
(q_8)	q_2	q_7
q_9	q_7	q_{10}
q_{10}	q_5	q_9

From the definition, two states p and q are 0-equivalent if both $\hat{\delta}(p,x)$ and $\hat{\delta}(q,x)$ are either final states or non-final states, for all $|x| = 0$. That is, both $\hat{\delta}(p,\varepsilon)$ and $\hat{\delta}(q,\varepsilon)$ are either final states or non-final states. That is, both p and q are either final states or non-final states.

Thus, under the equivalence relation $\overset{0}{\equiv}$, all final states are equivalent and all non-final states are equivalent. Hence, there are precisely two equivalence classes in the partition $\prod_0$ as given below:

$$\prod\nolimits_0 = \Big\{\{q_0, q_1, q_2, q_5, q_7, q_9, q_{10}\}, \{q_3, q_4, q_6, q_8\}\Big\}$$

From the Theorem 4.4.11, we know that any two 0-equivalent states, p and q, will further be 1-equivalent if

$$\delta(p,a) \overset{0}{\equiv} \delta(q,a)\ \forall a \in \Sigma.$$

Using this condition we can evaluate $\prod_1$ by checking every two 0-equivalent states whether they are further 1-equivalent or not. If they are 1-equivalent they continue to be in the same equivalence class. Otherwise, they will be put in different equivalence classes. The following shall illustrate the computation of $\prod_1$:

1. Consider the 0-equivalent states q_0 and q_1 and observe that

   (i) $\delta(q_0, a) = q_3$ and $\delta(q_1, a) = q_6$ are in the same equivalence class of $\prod_0$; and also

   (ii) $\delta(q_0, b) = q_2$ and $\delta(q_1, b) = q_2$ are in the same equivalence class of $\prod_0$.

   Thus, $q_0 \overset{1}{\equiv} q_1$ so that they will continue to be in same equivalence class in $\prod_1$ also.

2. In contrast to the above, consider the 0-equivalent states q_2 and q_5 and observe that

   (i) $\delta(q_2, a)$ and $\delta(q_5, a)$ are, respectively, q_8 and q_4; which are in the same equivalence class of $\prod_0$.

   (ii) Whereas, $\delta(q_2, b) = q_5$ and $\delta(q_5, b) = q_3$ are in different equivalences classes of $\prod_0$.

   Hence, q_2 and q_5 are not 1-equivalent. So, they will be in different equivalence classes of $\prod_1$.

3. Further, as illustrated above, one may verify whether are not $q_2 \overset{1}{\equiv} q_7$ and realize that q_2 and q_7 are not 1-equivalent. While putting q_7 in a different class that of q_2, we check for $q_5 \overset{1}{\equiv} q_7$ to decide to put q_5 and q_7 in the same equivalence class or not. As q_5 and q_7 are 1-equivalent, they will be put in the same equivalence of $\prod_1$.

Since, any two states which are not k-equivalent cannot be $(k+1)$-equivalent, we check for those pairs belonging to same equivalence of $\prod_0$ whether they are further 1-equivalent. Thus, we obtain

$$\prod_1 = \Big\{\{q_0, q_1, q_2\}, \{q_5, q_7\}, \{q_9, q_{10}\}, \{q_3, q_4, q_6, q_8\}\Big\}$$

Similarly, we continue to compute $\prod_2$, $\prod_3$, etc.

$$\prod_2 = \Big\{\{q_0, q_1, q_2\}, \{q_5, q_7\}, \{q_9, q_{10}\}, \{q_3, q_6\}, \{q_4, q_8\}\Big\}$$

$$\prod_3 = \Big\{\{q_0, q_1\}, \{q_2\}, \{q_5, q_7\}, \{q_9, q_{10}\}, \{q_3, q_6\}, \{q_4, q_8\}\Big\}$$
$$\prod_4 = \Big\{\{q_0, q_1\}, \{q_2\}, \{q_5, q_7\}, \{q_9, q_{10}\}, \{q_3, q_6\}, \{q_4, q_8\}\Big\}$$

Note that the process for a DFA will always terminate at a finite stage and get $\prod_k = \prod_{k+1}$, for some k. This is because, there are finite number of states and in the worst case equivalences may end up with singletons. Thereafter no further refinement is possible.

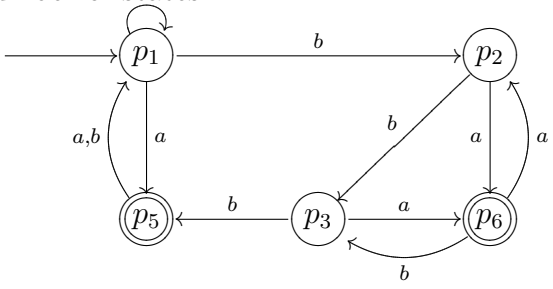
In the present context, $\prod_3 = \prod_4$. Thus it is partition $\prod$ corresponding to $\equiv$. Now we construct a DFA with these equivalence classes as states (by renaming them, for simplicity) and with the induced transitions. Thus we have an equivalent DFA with fewer number of states that of the given DFA.

The DFA with the equivalences classes as states is constructed below:

Let $P = \{p_1, \ldots, p_6\}$, where $p_1 = \{q_0, q_1\}$, $p_2 = \{q_2\}$, $p_3 = \{q_5, q_7\}$, $p_4 = \{q_9, q_{10}\}$, $p_5 = \{q_3, q_6\}$, $p_6 = \{q_4, q_8\}$. As p_1 contains the initial state q_0 of the given DFA, p_1 will be the initial state. Since p_5 and p_6 contain the final states of the given DFA, these two will form the set of final states. The induced transition function δ' is given in the following table.

δ'	a	b
$\rightarrow p_1$	p_5	p_2
p_2	p_6	p_3
p_3	p_6	p_5
p_4	p_3	p_4
(p_5)	p_1	p_1
(p_6)	p_2	p_3

Here, note that the state p_4 is inaccessible from the initial state p_1. This will also be removed and the following further simplified DFA can be produced with minimum number of states.



The following theorem confirms that the DFA obtained in this procedure, in fact, is having minimum number of states.

Theorem 4.4.15. *For every DFA $\mathscr{A}$, there is an equivalent minimum state DFA $\mathscr{A}'$.*

Proof. Let $\mathscr{A} = (Q, \Sigma, \delta, q_0, F)$ be a DFA and $\equiv$ be the state equivalence as defined in the Definition 4.4.8. Construct

$$\mathscr{A}' = (Q', \Sigma, \delta', q_0', F')$$

where

$Q' = \{[q] \mid q \text{ is accessible from } q_0\}$, the set of equivalence classes of Q with respect to $\equiv$ that contain the states accessible from q_0,

$q_0' = [q_0]$,

$F' = \{[q] \in Q' \mid q \in F\}$ and

$\delta' : Q' \times \Sigma \longrightarrow Q'$ is defined by $\delta'([q], a) = [\delta(q, a)]$.

For $[p], [q] \in Q'$, suppose $[p] = [q]$, i.e. $p \equiv q$. Now given $a \in \Sigma$, for each $x \in \Sigma^*$, both $\hat{\delta}(\delta(p, a), x) = \hat{\delta}(p, ax)$ and $\hat{\delta}(\delta(q, a), x) = \hat{\delta}(q, ax)$ are final or non-final states, as $p \equiv q$. Thus, $\delta(p, a)$ and $\delta(q, a)$ are equivalent. Hence, δ' is well-defined and $\mathscr{A}'$ is a DFA.

Claim 1: $L(\mathscr{A}) = L(\mathscr{A}')$.

Proof of Claim 1: In fact, we show that $\hat{\delta}'([q_0], x) = [\hat{\delta}(q_0, x)]$, for all $x \in \Sigma^*$. This suffices because

$$\hat{\delta}'([q_0], x) \in F' \Longleftrightarrow \hat{\delta}(q_0, x) \in F.$$

We prove our assertion by induction on $|x|$. Basis for induction is clear, because $\hat{\delta}'([q_0], \varepsilon) = [q_0] = [\hat{\delta}(q_0, \varepsilon)]$. For inductive step, consider $x \in \Sigma^*$ and $a \in \Sigma$. Now,

$$\begin{aligned}
\hat{\delta}'([q_0], xa) &= \delta'(\hat{\delta}'([q_0], x), a) \\
&= \delta'([\hat{\delta}(q_0, x)], a) \text{ (by inductive hypothesis)} \\
&= [\delta(\hat{\delta}(q_0, x), a)] \text{ (by definition of } \delta') \\
&= [\hat{\delta}(q_0, xa)]
\end{aligned}$$

as desired.

Claim 2: $\mathscr{A}'$ is a minimal state DFA accepting L.

Proof of Claim 2: We prove that the number of states of $\mathscr{A}'$ is equal to the number of states of $\mathscr{A}_L$, a minimal DFA accepting L. Since the states of $\mathscr{A}_L$ are the equivalence classes of $\sim_L$, it is sufficient to prove that

$$\text{the index of } \sim_L = \text{ the index of } \sim_{\mathscr{A}'}.$$

Recall the proof (2) $\Rightarrow$ (3) of Myhill Nerode theorem and observe that

$$\text{the index of } \sim_L \leq \text{ the index of } \sim_{\mathscr{A}'}.$$

On the other hand, suppose there are more number of equivalence classes for $\mathscr{A}'$ then that of $\sim_L$ does. That is, there exist $x, y \in \Sigma^*$ such that $x \sim_L y$, but not $x \sim_{\mathscr{A}'} y$.

That is, $\hat{\delta}'([q_0], x) \neq \hat{\delta}'([q_0], y)$.

That is, $[\hat{\delta}(q_0, x)] \neq [\hat{\delta}(q_0, y)]$.

That is, one among $\hat{\delta}(q_0, x)$ and $\hat{\delta}(q_0, y)$ is a final state and the other is a non-final state.

That is, $x \in L \iff y \notin L$. But this contradicts the hypothesis that $x \sim_L y$.

Hence, index of $\sim_L$ = index of $\sim_{\mathscr{A}'}$. □

Example 4.4.16. Consider the DFA given in following transition table.

δ	a	b
$\rightarrow \textcircled{q_0}$	q_1	q_0
q_1	q_0	q_3
$\textcircled{q_2}$	q_4	q_5
$\textcircled{q_3}$	q_4	q_1
$\textcircled{q_4}$	q_2	q_6
q_5	q_0	q_2
q_6	q_3	q_4

We compute the partition $\prod$ through the following:

$\prod_0 = \Big\{\{q_0, q_2, q_3, q_4\}, \{q_1, q_5, q_6\}\Big\}$

$\prod_1 = \Big\{\{q_0\}, \{q_2, q_3, q_4\}, \{q_1, q_5, q_6\}\Big\}$

$\prod_2 = \Big\{\{q_0\}, \{q_2, q_3, q_4\}, \{q_1, q_5\}, \{q_6\}\Big\}$

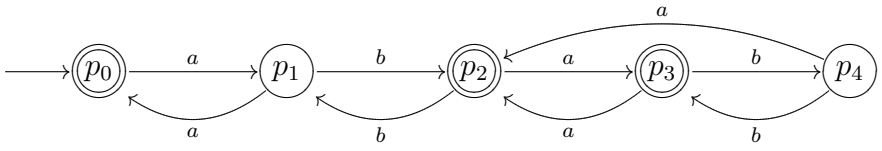
$\prod_3 = \Big\{\{q_0\}, \{q_2, q_3\}, \{q_4\}, \{q_1, q_5\}, \{q_6\}\Big\}$

$\prod_4 = \Big\{\{q_0\}, \{q_2, q_3\}, \{q_4\}, \{q_1, q_5\}, \{q_6\}\Big\}$

Since $\prod_3 = \prod_4$, we have $\prod_4 = \prod$. By renaming the equivalence classes as

$$p_0 = \{q_0\}; p_1 = \{q_1, q_5\}; p_2 = \{q_2, q_3\}; p_3 = \{q_4\}; p_4 = \{q_6\}$$

we construct an equivalent minimal DFA as shown in the following.



4.5 Regular Languages

Language represented by a regular expression is defined as a regular language. Now, we are in a position to provide alternative definitions for regular languages via finite automata (DFA or NFA) and also via regular grammars. That is the class of regular languages is precisely the class of languages accepted by finite automata and also it is the class of languages generated by regular grammars. These results are given in the following subsections. We also provide an algebraic method for converting a DFA to an equivalent regular expression.

4.5.1 Equivalence of Finite Automata and Regular Languages

A regular expressions r is said to be equivalent to a finite automaton $\mathscr{A}$, if the language represented by r is precisely accepted by the finite automaton $\mathscr{A}$, i.e. $L(r) = L(\mathscr{A})$. In order to establish the equivalence, we prove the following.

1. Given a regular expression, we construct an equivalent finite automaton.

2. Given a DFA $\mathscr{A}$, we show that $L(\mathscr{A})$ is regular.

To prove (1), we need the following. Let r_1 and r_2 be two regular expressions. Suppose there exist finite automata $\mathscr{A}_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ and $\mathscr{A}_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ which accept $L(r_1)$ and $L(r_2)$, respectively. Under this hypothesis, we prove the following three lemmas.

Lemma 4.5.1. *There exists a finite automaton accepting* $L(r_1 + r_2)$.

Proof. Construct $\mathscr{A} = (Q, \Sigma, \delta, q_0, F)$, where
$Q = Q_1 \cup Q_2 \cup \{q_0\}$ with a new state q_0,
$F = F_1 \cup F_2$ and
$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \longrightarrow \mathscr{P}(Q)$ defined by

$$\delta(q,a) = \begin{cases} \delta_1(q,a), & \text{if } q \in Q_1, a \in \Sigma \cup \{\varepsilon\}; \\ \delta_2(q,a), & \text{if } q \in Q_2, a \in \Sigma \cup \{\varepsilon\}; \\ \{q_1, q_2\}, & \text{if } q = q_0, a = \varepsilon. \end{cases}$$

Then clearly, $\mathscr{A}$ is a finite automaton as depicted in Figure 4.4.

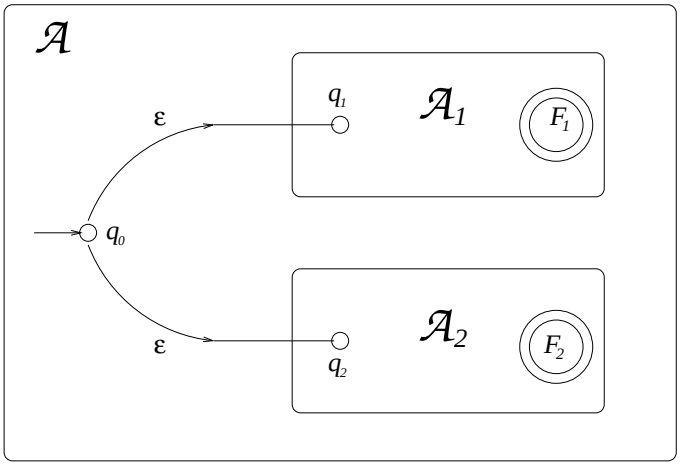


Figure 4.4: Parallel Composition of Finite Automata

Now, for $x \in \Sigma^*$,

$$\begin{aligned}
x \in L(\mathscr{A}) &\iff (\hat{\delta}(q_0, x) \cap F) \neq \varnothing \\
&\iff (\hat{\delta}(q_0, \varepsilon x) \cap F) \neq \varnothing \\
&\iff (\hat{\delta}(\hat{\delta}(q_0, \varepsilon), x) \cap F) \neq \varnothing \\
&\iff (\hat{\delta}(\{q_1, q_2\}, x) \cap F) \neq \varnothing \\
&\iff ((\hat{\delta}(q_1, x) \cup \hat{\delta}(q_2, x)) \cap F) \neq \varnothing \\
&\iff ((\hat{\delta}(q_1, x) \cap F) \cup (\hat{\delta}(q_2, x) \cap F)) \neq \varnothing \\
&\iff (\hat{\delta}(q_1, x) \cap F_1) \neq \varnothing \text{ or } (\hat{\delta}(q_2, x) \cap F_2) \neq \varnothing \\
&\iff x \in L(\mathscr{A}_1) \text{ or } x \in L(\mathscr{A}_2) \\
&\iff x \in L(\mathscr{A}_1) \cup L(\mathscr{A}_2).
\end{aligned}$$

Hence, $L(\mathscr{A}) = L(\mathscr{A}_1) \cup L(\mathscr{A}_2)$. □

Lemma 4.5.2. *There exists a finite automaton accepting $L(r_1 r_2)$.*

Proof. Construct

$$\mathscr{A} = (Q, \Sigma, \delta, q_1, F_2),$$

where $Q = Q_1 \cup Q_2$ and δ is defined by

$$\delta(q, a) = \begin{cases} \delta_1(q, a), & \text{if } q \in Q_1, a \in \Sigma \cup \{\varepsilon\}; \\ \delta_2(q, a), & \text{if } q \in Q_2, a \in \Sigma \cup \{\varepsilon\}; \\ \{q_2\}, & \text{if } q \in F_1, a = \varepsilon. \end{cases}$$

Then $\mathscr{A}$ is a finite automaton as shown in Figure 4.5.

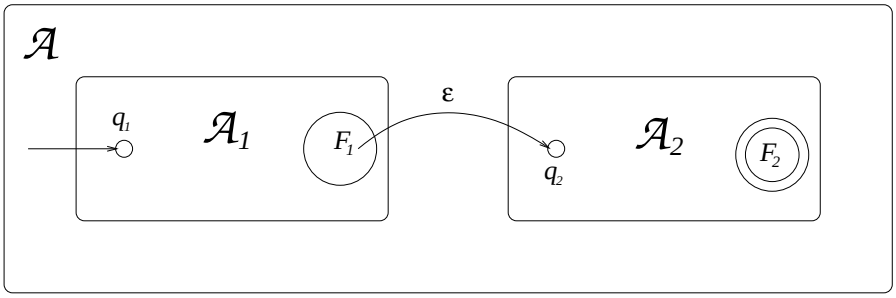


Figure 4.5: Series Composition of Finite Automata

We claim that $L(\mathscr{A}) = L(\mathscr{A}_1)L(\mathscr{A}_2)$. Suppose $x = a_1a_2 \ldots a_n \in L(\mathscr{A})$, i.e. $\hat{\delta}(q_1, x) \cap F_2 \neq \varnothing$. It is clear from the construction of $\mathscr{A}$ that the only way to reach from q_1 to any state of F_2 is via q_2. Further, we have only ε-transitions from the states of F_1 to q_2. Thus, while traversing through x from q_1 to some state of F_2, there exist $p \in F_1$ and a number $k \leq n$ such that

$$p \in \hat{\delta}(q_1, a_1a_2 \ldots a_k) \text{ and } \hat{\delta}(q_2, a_{k+1}a_{k+2} \ldots a_n) \cap F_2 \neq \varnothing.$$

Then

$$x_1 = a_1a_2 \ldots a_k \in L(\mathscr{A}_1) \text{ and } x_2 = a_{k+1}a_{k+2} \ldots a_n \in L(\mathscr{A}_2)$$

so that $x = x_1x_2 \in L(\mathscr{A}_1)L(\mathscr{A}_2)$.

Conversely, suppose $x \in L(\mathscr{A}_1)L(\mathscr{A}_2)$. Then $x = x_1x_2$, for some $x_1 \in L(\mathscr{A}_1)$ and some $x_2 \in L(\mathscr{A}_2)$. That is,

$$\hat{\delta}_1(q_1, x_1) \cap F_1 \neq \varnothing \text{ and } \hat{\delta}_2(q_2, x_2) \cap F_2 \neq \varnothing.$$

Now,

$$\begin{aligned}
(q_1, x) &= (q_1, x_1x_2) \\
&\vdash^* (p, x_2), \text{ for some } p \in F_1 \\
&= (p, \varepsilon x_2) \\
&\vdash (q_2, x_2), \text{ since } \delta(p, \varepsilon) = \{q_2\} \\
&\vdash^* (p', \varepsilon), \text{ for some } p' \in F_2
\end{aligned}$$

Hence, $\hat{\delta}(q_1, x) \cap F_2 \neq \varnothing$ so that $x \in L(\mathscr{A})$. □

Lemma 4.5.3. *There exists a finite automaton accepting $L(r_1)^*$.*

Proof. Construct

$$\mathscr{A} = (Q, \Sigma, \delta, q_0, F),$$

where $Q = Q_1 \cup \{q_0, p\}$ with new states q_0 and p, $F = \{p\}$ and define δ by

$$\delta(q,a) = \begin{cases} \{q_1, p\}, & \text{if } q \in F_1 \cup \{q_0\} \text{ and } a = \varepsilon \\ \delta_1(q,a), & \text{if } q \in Q_1 \text{ and } a \in \Sigma \cup \{\varepsilon\} \end{cases}$$

Then $\mathscr{A}$ is a finite automaton as given in Figure 4.6.

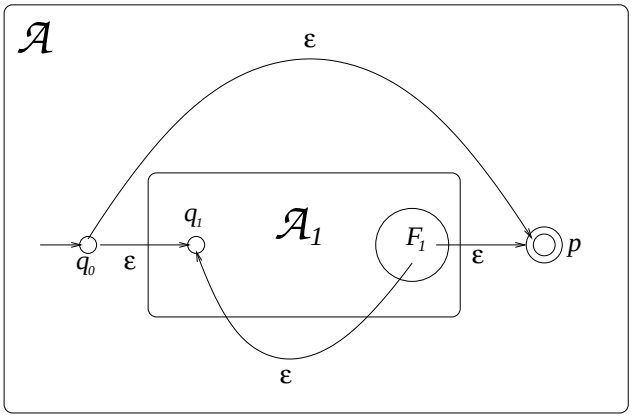


Figure 4.6: Kleene Star of a Finite Automaton

We prove that $L(\mathscr{A}) = L(\mathscr{A}_1)^*$. For $x \in \Sigma^*$,

$$\begin{aligned} x \in L(\mathscr{A}) &\implies \hat{\delta}(q_0, x) = \{p\} \\ &\implies \text{either } x = \varepsilon \text{ or } \hat{\delta}(q_0, x) \cap F_1 \neq \varnothing. \end{aligned}$$

If $x = \varepsilon$ then trivially $x \in L(\mathscr{A}_1)^*$. Otherwise, there exist

$$p_1, p_2, \ldots, p_k \in F_1 \text{ and } x_1, x_2, \ldots, x_k$$

such that

$$p_1 \in \hat{\delta}(q_1, x_1), p_2 \in \hat{\delta}(q_1, x_2), \ldots, p_k \in \hat{\delta}(q_1, x_k)$$

with $x = x_1 x_2 \ldots x_k$. Thus, for all $1 \leq i \leq k$, $x_i \in L(\mathscr{A}_1)$ so that $x \in L(\mathscr{A}_1)^*$.

Conversely, suppose $x \in L(\mathscr{A}_1)^*$. Then $x = x_1 x_2 \cdots x_l$ with $x_i \in L(\mathscr{A}_1)$ for all i and for some $l \geq 0$. If $l = 0$, then $x = \varepsilon$ and clearly, $x \in L(\mathscr{A})$. Otherwise, we have

$$\hat{\delta}_1(q_1, x_i) \cap F_1 \neq \varnothing \text{ for } 1 \leq i \leq l.$$

Now, consider the following computation of $\mathscr{A}$ on x:

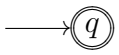
$$\begin{array}{rcl}
(q_0, x) & = & (q_0, \varepsilon x) \\
& \vdash & (q_1, x) \\
& = & (q_1, x_1 x_2 \cdots x_l) \\
& \vdash^* & (p_1', x_2 x_3 \cdots x_l), \quad \text{for some} \quad p_1' \in F_1 \\
& = & (p_1', \varepsilon x_2 x_3 \ldots x_l) \\
& \vdash & (q_1, x_2 x_3 \ldots x_l), \quad \text{since} \quad q_1 \in \delta(p_1', \varepsilon) \\
& \vdots & \\
& \vdash^* & (p_l', \varepsilon), \quad \text{for some} \quad p_l' \in F_1 \\
& \vdash & (p, \varepsilon), \quad \text{since} \quad p \in \delta(p_l', \varepsilon).
\end{array}$$

As $\hat{\delta}(q_0, x) \cap F \neq \varnothing$ we have $x \in L(\mathscr{A})$. □

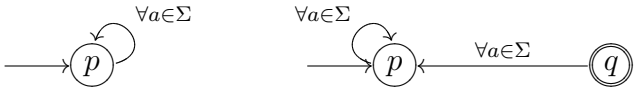
Theorem 4.5.4. *The language denoted by a regular expression can be accepted by a finite automaton.*

Proof. We prove the result by induction on the number of operators of a regular expression r. Suppose r has zero operators, then r must be $\varepsilon, \varnothing$ or a for some $a \in \Sigma$.

If $r = \varepsilon$, then the finite automaton as depicted below serves the purpose.

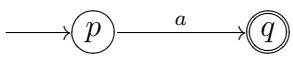


If $r = \varnothing$, then (i) any finite automaton with no final state will do; or (ii) one may consider a finite automaton in which final states are not accessible from the initial state. For instance, the following two automata are given for each one of the two types indicated above and serve the purpose.



(i) (ii)

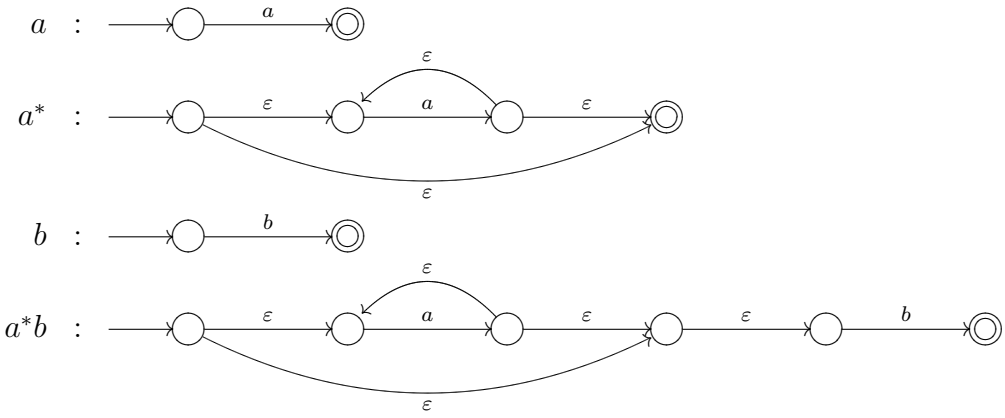
In case $r = a$, for some $a \in \Sigma$,



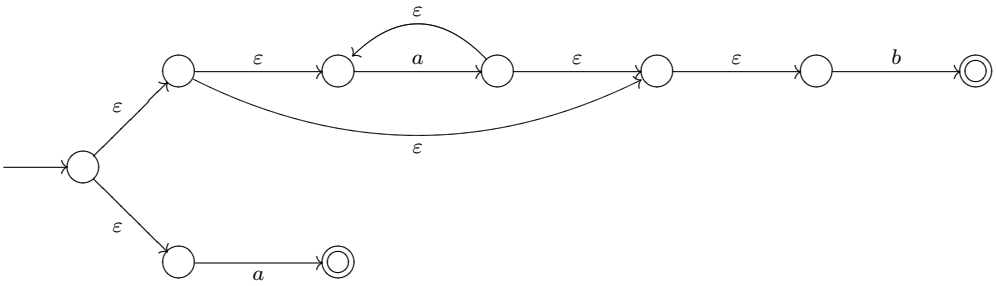
is a finite automaton which accepts r.

Suppose the result is true for regular expressions with k or fewer operators and assume r has $k+1$ operators. There are three cases according to the operators involved. (1) $r = r_1 + r_2$, (2) $r = r_1 r_2$, or (3) $r = r_1^*$, for some regular expressions r_1 and r_2. In any case, note that both the regular expressions r_1 and r_2 must have k or fewer operators. Thus by inductive hypothesis, there exist finite automata $\mathscr{A}_1$ and $\mathscr{A}_2$ which accept $L(r_1)$ and $L(r_2)$, respectively. Then, for each case we have a finite automaton accepting $L(r)$, by Lemmas 4.5.1, 4.5.2, or 4.5.3, case wise. □

Example 4.5.5. Here, we demonstrate the construction of an NFA for the regular expression $a^*b + a$. First, we list the corresponding NFA for each subexpression of $a^*b + a$.



Now, finally an NFA for $a^*b + a$ is:



Theorem 4.5.6. *If $\mathscr{A}$ is a DFA, then $L(\mathscr{A})$ is regular.*

Proof. We prove the result by induction on the number of states of DFA. For base case, let $\mathscr{A} = (Q, \Sigma, \delta, q_0, F)$ be a DFA with only one state. Then there are two possibilities for the set of final states F.

$F = \varnothing$: In this case $L(\mathscr{A}) = \varnothing$ and is regular.

$F = Q$: In this case $L(\mathscr{A}) = \Sigma^*$ which is already shown to be regular.

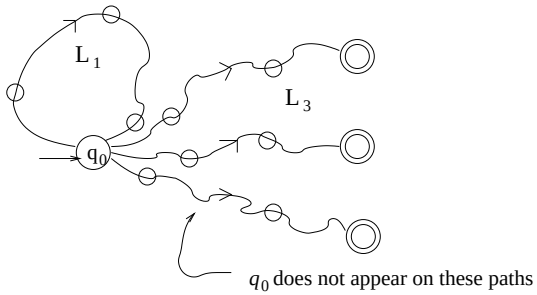


Figure 4.7: Depiction of the Language of a DFA

Assume that the result is true for all those DFA whose number of states is less than n. Now, let $\mathscr{A} = (Q, \Sigma, \delta, q_0, F)$ be a DFA with $|Q| = n$. First note that the language $L = L(\mathscr{A})$ can be written as

$$L = L_1^* L_2$$

where

> L_1 is the set of strings that start and end in the initial state q_0
>
> L_2 is the set of strings that start in q_0 and end in some final state. We include ε in L_2 if q_0 is also a final state. Further, we add a restriction that q_0 will not be revisited while traversing those strings. This is justified because, the portion of a string from the initial position q_0 till that revisits q_0 at the last time will be part of L_1 and the rest of the portion will be in L_2.

Using the inductive hypothesis, we prove that both L_1 and L_2 are regular. Since regular languages are closed with respect to Kleene star and concatenation it follows that L is regular.

The following notation shall be useful in defining the languages L_1 and L_2, formally, and show that they are regular. For $q \in Q$ and $x \in \Sigma^*$, let us denote the set of states on the path of x from q that come after q by $P_{(q,x)}$. That is, if $x = a_1 \cdots a_n$,

$$P_{(q,x)} = \bigcup_{i=1}^{n} \{\hat{\delta}(q, a_1 \cdots a_i)\}.$$

Define

$$\begin{aligned} L_1 &= \{x \in \Sigma^* \mid \hat{\delta}(q_0, x) = q_0\}; \text{ and} \\ L_2 &= \begin{cases} L_3, & \text{if } q_0 \notin F; \\ L_3 \cup \{\varepsilon\}, & \text{if } q_0 \in F, \end{cases} \end{aligned}$$

where $L_3 = \{x \in \Sigma^* \mid \hat{\delta}(q_0, x) \in F, q_0 \notin P_{(q_0,x)}\}$. Clearly, $L = L_1^* L_2$.

Claim 1: L_1 is regular.

Proof of Claim 1:

Consider the following set

$$A = \left\{ (a,b) \in \Sigma \times \Sigma \;\middle|\; \begin{array}{l} \hat{\delta}(q_0, axb) = q_0, \text{ for some } x \in \Sigma^*; \\ \delta(q_0, a) \neq q_0; \\ q_0 \notin P_{(q_a,x)}, \text{ where } q_a = \delta(q_0, a). \end{array} \right\}$$

and for $(a,b) \in A$, define

$$L_{(a,b)} = \{x \in \Sigma^* \mid \hat{\delta}(q_0, axb) = q_0 \text{ and } q_0 \notin P_{(q_a,x)}, \text{ where } q_a = \delta(q_0, a)\}.$$

Note that $L_{(a,b)}$ is the language accepted by the DFA

$$\mathscr{A}_{(a,b)} = (Q', \Sigma, \delta', q_a, F')$$

where $Q' = Q \setminus \{q_0\}$, $q_a = \delta(q_0, a)$, $F' = \{q \in Q \mid \delta(q, b) = q_0\} \setminus \{q_0\}$, and δ' is the restriction of δ to $Q' \times \Sigma$, i.e. $\delta' = \delta\Big|_{Q' \times \Sigma}$. For instance, if $x \in L(\mathscr{A}_{(a,b)})$ then, since $q_0 \notin Q'$, $q_0 \notin P_{(q_a,x)}$ and $\hat{\delta}'(q_a, x) \in F'$. This implies,

$$\begin{aligned} \hat{\delta}(q_0, axb) &= \hat{\delta}(\delta(q_0, a), xb) \\ &= \hat{\delta}(q_a, xb) \\ &= \delta(\hat{\delta}(q_a, x), b) \\ &= \delta(p, b), \quad \text{where } p \in F', \text{ as } \delta' = \delta\Big|_{Q' \times \Sigma} \\ &= q_0, \end{aligned}$$

so that $x \in L_{(a,b)}$. Converse is similar. Thus $L_{(a,b)} = L(\mathscr{A}_{(a,b)})$. Hence, as $|Q'| = n - 1$, by inductive hypothesis, $L_{(a,b)}$ is regular. Now if we write

$$B = \{a \in \Sigma \mid \delta(q_0, a) = q_0\} \cup \{\varepsilon\}$$

then clearly,

$$L_1 = B \cup \bigcup_{(a,b) \in A} a L_{(a,b)} b.$$

Hence L_1 is regular.

Claim 2: L_3 is regular.

Proof of Claim 2: Consider the following set

$$C = \{a \in \Sigma \mid \delta(q_0, a) \neq q_0\}$$

and for $a \in C$, define

$$L_a = \{x \in \Sigma^* \mid \hat{\delta}(q_0, ax) \in F \text{ and } q_0 \notin P_{(q_a,x)}, \text{ where } q_a = \delta(q_0, a)\}.$$

For $a \in C$, we set

$$\mathscr{A}_a = (Q', \Sigma, \delta', q_a, F'')$$

where $Q' = Q \setminus \{q_0\}$, $q_a = \delta(q_0, a)$, $F'' = F \setminus \{q_0\}$, and δ' is the restriction of δ to $Q' \times \Sigma$, i.e. $\delta' = \delta\Big|_{Q' \times \Sigma}$. It easy to observe that $L(\mathscr{A}_a) = L_a$. First note that q_0 does not appear in the context of L_a and $L(\mathscr{A}_a)$. Now,

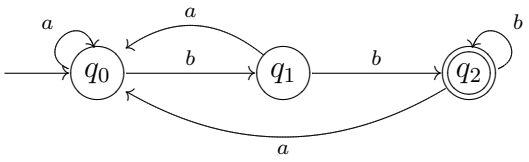
$$\begin{aligned}
x \in L(\mathscr{A}_a) &\iff \hat{\delta}'(q_a, x) \in F'' \text{ (as } q_0 \text{ does not appear)} \\
&\iff \hat{\delta}'(q_a, x) \in F \\
&\iff \hat{\delta}(q_a, x) \in F \\
&\iff \hat{\delta}(\delta(q_0, a), x) \in F \\
&\iff \hat{\delta}(q_0, ax) \in F \text{ (as } q_0 \text{ does not appear)} \\
&\iff ax \in L_a.
\end{aligned}$$

Again, since $|Q'| = n - 1$, by inductive hypothesis, we have L_a is regular. But, clearly

$$L_3 = \bigcup_{a \in C} aL_a$$

so that L_3 is regular. This completes the proof of the theorem. □

Example 4.5.7. Consider the following DFA



1. Note that the following strings bring the DFA from q_0 back to q_0.
   (a) a (via the path $\langle q_0, q_0 \rangle$)
   (b) ba (via the path $\langle q_0, q_1, q_0 \rangle$)
   (c) For $n \geq 0$, $bbb^n a$ (via the path $\langle q_0, q_1, q_2, q_2, \ldots, q_2, q_0 \rangle$)

   Thus $L_1 = \{a, ba, bbb^n a \mid n \geq 0\} = a + ba + bbb^* a$.

2. Again, since q_0 is not a final state, L_2 – the set of strings which take the DFA from q_0 to the final state q_2 – is

$$\{bbb^n \mid n \geq 0\} = bbb^*.$$

Thus, as per the construction of Theorem 4.5.6, the language accepted by the given DFA is $L_1^*L_2$ and it can be represented by the regular expression

$$(a + ba + bbb^*a)^*bbb^*$$

and hence it is regular.

Brzozowski Algebraic Method

Brzozowski algebraic method gives a conversion of DFA to its equivalent regular expression. Let $\mathscr{A} = (Q, \Sigma, \delta, q_0, F)$ be a DFA with $Q = \{q_0, q_2, \ldots, q_n\}$ and $F = \{q_{f_1}, \ldots, q_{f_k}\}$. For each $q_i \in Q$, write

$$R_i = \{x \in \Sigma^* \mid \hat{\delta}(q_0, x) = q_i\}$$

and note that

$$L(\mathscr{A}) = \bigcup_{i=1}^{k} R_{f_i}$$

In order to construct a regular expression for $L(\mathscr{A})$, here we propose an unknown expression for each R_i, say r_i. We are indented to observe that r_i, for all i, is a regular expression so that

$$r_{f_1} + r_{f_2} + \cdots + r_{f_k}$$

is a regular expression for $L(\mathscr{A})$, as desired.

Suppose $\Sigma_{(i,j)}$ is the set of those symbols of Σ which take $\mathscr{A}$ from q_i to q_j, i.e.

$$\Sigma_{(i,j)} = \{a \in \Sigma \mid \delta(q_i, a) = q_j\}.$$

Clearly, as it is a finite set, $\Sigma_{(i,j)}$ is regular with the regular expression as sum of its symbols. Let $s_{(i,j)}$ be the expression for $\Sigma_{(i,j)}$. Now, for $1 \leq j \leq n$, since the strings of R_j are precisely taking the DFA from q_0 to any state q_i then reaching q_j with the symbols of $\Sigma_{(i,j)}$, we have

$$R_j = R_0\Sigma_{(0,j)} \cup \cdots \cup R_i\Sigma_{(i,j)} \cup \cdots \cup R_n\Sigma_{(n,j)}.$$

And in case of R_0, it is

$$R_0 = R_0\Sigma_{(0,0)} \cup \cdots \cup R_i\Sigma_{(i,0)} \cup \cdots \cup R_n\Sigma_{(n,0)} \cup \{\varepsilon\},$$

as ε takes the DFA from q_0 to itself. Thus, for each j, we have an equation

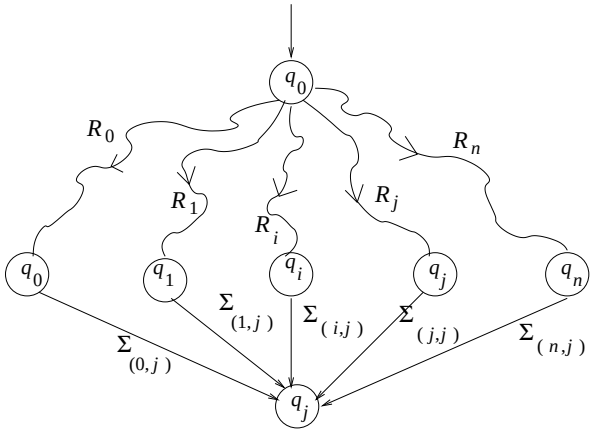


Figure 4.8: Depiction of the Set R_j

for r_j which is depending on all r_i's, called characteristic equation of r_j. The system of characteristic equations of $\mathscr{A}$ is:

$$\begin{aligned}
r_0 &= r_0 s_{(0,0)} + r_1 s_{(1,0)} + \cdots + r_i s_{(i,0)} + \cdots + r_n s_{(n,0)} + \varepsilon \\
r_1 &= r_0 s_{(0,1)} + r_1 s_{(1,1)} + \cdots + r_i s_{(i,1)} + \cdots + r_n s_{(n,1)} \\
&\vdots \\
r_j &= r_0 s_{(0,j)} + r_1 s_{(1,j)} + \cdots + r_i s_{(i,j)} + \cdots + r_n s_{(n,j)} \\
&\vdots \\
r_n &= r_0 s_{(0,n)} + r_1 s_{(1,n)} + \cdots + r_i s_{(i,n)} + \cdots + r_n s_{(n,n)}
\end{aligned}$$

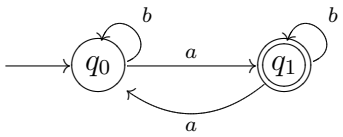
The system can be solved for r_{f_i}'s, expressions for final states, via straightforward substitution, except the same unknown appears on the both the left and right hand sides of a equation. This situation can be handled using Arden's principle (see Exercise **??**) which states that

> Let s and t be regular expressions and r is an unknown. A equation of the form $r = t + rs$, where $\varepsilon \notin L(s)$, has a unique solution given by $r = ts^*$.

By successive substitutions and application of Arden's principle we evaluate the expressions for final states purely in terms of symbols from Σ. Since the operations involved here are admissible for regular expression, we eventually obtain regular expressions for each r_{f_i}.

We demonstrate the method through the following examples.

Example 4.5.8. Consider the following DFA



The characteristic equations for the states q_0 and q_1, respectively, are:

$$\begin{aligned} r_0 &= r_0b + r_1a + \varepsilon \\ r_1 &= r_0a + r_1b \end{aligned}$$

Since q_1 is the final state, r_1 represents the language of the DFA. Hence, we need to solve for r_1 explicitly in terms of a's and b's. Now, by Arden's principle, r_0 yields

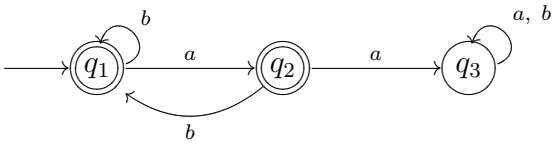
$$r_0 = (\varepsilon + r_1a)b^*.$$

Then, by substituting r_0 in r_1, we get

$$r_1 = (\varepsilon + r_1a)b^*a + r_1b = b^*a + r_1(ab^*a + b)$$

Then, by applying Arden's principle on r_1, we get $r_1 = b^*a(ab^*a + b)^*$ which is the desired regular expression representing the language of the given DFA.

Example 4.5.9. Consider the following DFA



The characteristic equations for the states q_1, q_2 and q_3, respectively, are:

$$\begin{aligned} r_1 &= r_1b + r_2b + \varepsilon \\ r_2 &= r_1a \\ r_3 &= r_2a + r_3(a + b) \end{aligned}$$

Since q_1 and q_2 are final states, $r_1 + r_2$ represents the language of the DFA. Hence, we need to solve for r_1 and r_2 explicitly in terms of a's and b's. Now, by substituting r_2 in r_1, we get

$$r_1 = r_1b + r_1ab + \varepsilon = r_1(b + ab) + \varepsilon.$$

Then, by Arden's principle, we have $r_1 = \varepsilon(b + ab)^* = (b + ab)^*$. Thus,

$$r_1 + r_2 = (b + ab)^* + (b + ab)^*a.$$

Hence, the regular expressions represented by the given DFA is

$$(b + ab)^*(\varepsilon + a).$$

4.5.2 Equivalence of Finite Automata and Regular Grammars

A finite automaton $\mathscr{A}$ is said to be equivalent to a regular grammar $\mathcal{G}$, if the language accepted by $\mathscr{A}$ is precisely generated by the grammar $\mathcal{G}$, i.e. $L(\mathscr{A}) = L(\mathcal{G})$. Now, we prove that finite automata and regular grammars are equivalent. In order to establish this equivalence, we first prove that given a DFA we can construct an equivalent regular grammar. Then for converse, given a regular grammar, we construct an equivalent generalized finite automaton (GFA), a notion which we introduce here as an equivalent notion for DFA.

Theorem 4.5.10. *If $\mathscr{A}$ is a DFA, then $L(\mathscr{A})$ can be generated by a regular grammar.*

Proof. Suppose $\mathscr{A} = (Q, \Sigma, \delta, q_0, F)$ is a DFA. Construct a grammar $\mathcal{G} = (\mathcal{N}, \Sigma, \mathcal{P}, S)$ by setting $\mathcal{N} = Q$ and $S = q_0$ and

$$\mathcal{P} = \{A \to aB \mid B \in \delta(A, a)\} \cup \{A \to a \mid \delta(A, a) \in F\}.$$

In addition, if the initial state $q_0 \in F$, then we include $S \to \varepsilon$ in $\mathcal{P}$. Clearly $\mathcal{G}$ is a regular grammar. We claim that $L(\mathcal{G}) = L(\mathscr{A})$.

From the construction of $\mathcal{G}$, it is clear that $\varepsilon \in L(\mathscr{A})$ if and only if $\varepsilon \in L(\mathcal{G})$. Now, for $n \geq 1$, let $x = a_1 a_2 \dots a_n \in L(\mathscr{A})$ be arbitrary. That is $\hat{\delta}(q_0, a_1 a_2 \dots a_n) \in F$. This implies, there exists a sequence of states

$$q_1, q_2, \dots, q_n$$

such that

$$\delta(q_{i-1}, a_i) = q_i, \quad \text{for} \quad 1 \leq i \leq n, \quad \text{and} \quad q_n \in F.$$

As per construction of $\mathcal{G}$, we have

$$q_{i-1} \to a_i q_i \in \mathcal{P}, \quad \text{for} \quad 1 \leq i \leq n-1, \quad \text{and} \quad q_n \to a_n \in \mathcal{P}.$$

Using these production rules we can derive x in $\mathcal{G}$ as follows:

$$\begin{aligned} S = q_0 &\Rightarrow a_1 q_1 \\ &\Rightarrow a_1 a_2 q_2 \\ &\vdots \\ &\Rightarrow a_1 a_2 \cdots a_{n-1} q_{n-1} \\ &\Rightarrow a_1 a_2 \cdots a_n = x. \end{aligned}$$

Thus $x \in L(\mathcal{G})$.

Conversely, suppose $y = b_1 \cdots b_m \in L(\mathcal{G})$, for $m \geq 1$, i.e. $S \overset{*}{\Rightarrow} y$ in $\mathcal{G}$. Since every production rule of $\mathcal{G}$ is form $A \to aB$ or $A \to a$, the derivation $S \overset{*}{\Rightarrow} y$ has exactly m steps and first $m-1$ steps are because of production rules of the type $A \to aB$ and the last mth step is because of the rule of the form $A \to a$. Thus, in every step of the deviation one b_i of y can be produced in the sequence. Precisely, the derivation can be written as

$$\begin{aligned} S &\Rightarrow b_1 B_1 \\ &\Rightarrow b_1 b_2 B_2 \\ &\vdots \\ &\Rightarrow b_1 b_2 \cdots b_{m-1} B_{m-1} \\ &\Rightarrow b_1 b_2 \cdots b_m = y. \end{aligned}$$

From the construction of $\mathcal{G}$, it can be observed that

$$\delta(B_{i-1}, b_i) = B_i, \quad \text{for} \quad 1 \leq i \leq m-1, \quad \text{and } B_0 = S$$

in $\mathscr{A}$. Moreover, $\delta(B_{m-1}, b_m) \in F$. Thus,

$$\begin{aligned} \hat{\delta}(q_0, y) = \hat{\delta}(S, b_1 \cdots b_m) &= \hat{\delta}(\delta(S, b_1), b_2 \cdots b_m) \\ &= \hat{\delta}(B_1, b_2 \cdots b_m) \\ &\vdots \\ &= \hat{\delta}(B_{m-1}, b_m) \\ &= \delta(B_{m-1}, b_m) \in F \end{aligned}$$

so that $y \in L(\mathscr{A})$. Hence $L(\mathscr{A}) = L(\mathcal{G})$. □

Example 4.5.11. Consider the DFA given in Example 4.5.8. Set $\mathcal{N} = \{q_0, q_1\}$, $\Sigma = \{a, b\}$, $S = q_0$ and $\mathcal{P}$ has the following production rules:

$$\begin{aligned} q_0 &\to aq_1 \mid bq_0 \mid a \\ q_1 &\to aq_0 \mid bq_1 \mid b \end{aligned}$$

Now $\mathcal{G} = (\mathcal{N}, \Sigma, \mathcal{P}, S)$ is a regular grammar that is equivalent to the given DFA.

Example 4.5.12. Consider the DFA given in Example 4.5.9. The regular grammar $\mathcal{G} = (\mathcal{N}, \Sigma, \mathcal{P}, S)$, where $\mathcal{N} = \{q_1, q_2, q_3\}$, $\Sigma = \{a, b\}$, $S = q_1$ and P has the following rules

$$\begin{aligned} q_1 &\to aq_2 \mid bq_1 \mid a \mid b \mid \varepsilon \\ q_2 &\to aq_3 \mid bq_1 \mid b \\ q_3 &\to aq_3 \mid bq_3 \end{aligned}$$

is equivalent to the given DFA. Here, note that q_3 is a trap state. So, the production rules in which q_3 is involved can safely be removed to get a simpler but equivalent regular grammar with the following production rules.

$$\begin{aligned} q_1 &\rightarrow aq_2 \mid bq_1 \mid a \mid b \mid \varepsilon \\ q_2 &\rightarrow bq_1 \mid b \end{aligned}$$

Definition 4.5.13. A *generalized finite automaton* (GFA) is nondeterministic finite automaton in which the transitions may be given via stings from a finite set, instead of just via symbols. That is, formally, GFA is a sextuple $(Q, \Sigma, X, \delta, q_0, F)$, where Q, Σ, q_0, F are as usual in an NFA. Whereas, the transition function

$$\delta : Q \times X \longrightarrow \wp(Q)$$

with a finite subset X of Σ^*.

One can easily prove that the GFA is no more powerful than an NFA. That is, the language accepted by a GFA is regular. This can be done by converting each transition of a GFA into a transition of an NFA. For instance, suppose there is a transition from a state p to a state q via s string $x = a_1 \cdots a_k$, for $k \geq 2$, in a GFA. Choose $k-1$ new state that not already there in the GFA, say $p_1, \ldots, p_{k-1}$ and replace the transition

$$p \xrightarrow{x} q$$

by the following new transitions via the symbols a_i's

$$p \xrightarrow{a_1} p_1, \;\; p_1 \xrightarrow{a_2} p_2, \;\; \cdots, \;\; p_{k-1} \xrightarrow{a_k} q$$

In a similar way, all the transitions via strings, of length at least 2, can be replaced in a GFA to convert that as an NFA without disturbing its language.

Theorem 4.5.14. *If L is generated by a regular grammar, then L is regular.*

Proof. Let L be generated by the regular grammar $\mathcal{G} = (\mathcal{N}, \Sigma, \mathcal{P}, S)$. Note that every production rule of $\mathcal{G}$ is either of the form $A \rightarrow xB$ or of the form $A \rightarrow x$, for some $x \in \Sigma^*$ and $A, B \in \mathcal{N}$. We construct a GFA that accepts L, so that L is regular.

Let X be the set of all terminal strings that are on the righthand side of productions of $\mathcal{G}$. That is,

$$X = \Big\{ x \;\Big|\; A \rightarrow xB \in \mathcal{P} \;\text{ or }\; A \rightarrow x \in \mathcal{P} \Big\}.$$

Since $\mathcal{P}$ is a finite set, we have X is a finite subset of Σ^*. Now, construct a GFA

$$\mathscr{A} = (Q, \Sigma, X, \delta, q_0, F)$$

by setting $Q = \mathcal{N} \cup \{\$\}$, where $\$$ is a new symbol, $q_0 = S$, $F = \{\$\}$ and the transition function δ is defied by

$$B \in \delta(A, x) \Longleftrightarrow A \to xB \in \mathcal{P}$$

and

$$\$ \in \delta(A, x) \Longleftrightarrow A \to x \in \mathcal{P}$$

We claim that $L(\mathscr{A}) = L$.

Let $w \in L$, i.e. there is a derivation for w in $\mathcal{G}$. Assume the derivation has k steps, which is obtained by the following $k-1$ production rules in the first $k-1$ steps

$$A_{i-1} \to x_i A_i, \text{ for } 1 \leq i \leq k-1, \text{ with } S = A_0,$$

and at the end, in the kth step, the production rule

$$A_{k-1} \to x_k.$$

Thus, $w = x_1 x_2 \cdots x_k$ and the derivation is as shown below:

$$\begin{aligned} S = A_0 &\Rightarrow x_1 A_1 \\ &\Rightarrow x_1 x_2 A_2 \\ &\vdots \\ &\Rightarrow x_1 x_2 \cdots x_{k-1} A_{k-1} \\ &\Rightarrow x_1 x_2 \cdots x_k = w. \end{aligned}$$

From the construction of $\mathscr{A}$, it is clear that $\mathscr{A}$ has the following transitions:

$$(A_{i-1}, x_i) \vdash (A_i, \varepsilon) \text{ for } 1 \leq i \leq k,$$

where $A_0 = S$ and $A_k = \$$. Using these transitions, it can be observed that $w \in L(\mathscr{A})$. For instance,

$$\begin{aligned} (q_0, w) = (S, x_1 \cdots x_k) &= (A_0, x_1 x_2 \cdots x_k) \\ &\vdash (A_1, x_2 \cdots x_k) \\ &\vdots \\ &\vdash (A_{k-1}, x_k) \\ &\vdash (\$, \varepsilon). \end{aligned}$$

Thus, $L \subseteq L(\mathscr{A})$. Converse can be shown using a similar argument as in the previous theorem. Hence $L(\mathscr{A}) = L$. □

Example 4.5.15. Consider the regular grammar given in the Example 3.3.10. Let $Q = \{S, A, \$\}$, $\Sigma = \{a, b\}$, $X = \{\varepsilon, a, b, ab\}$ and $F = \{\$\}$. Set $\mathscr{A} = (Q, \Sigma, X, \delta, S, F)$, where $\delta : Q \times X \longrightarrow \wp(Q)$ is defined by the following table.

δ	ε	a	b	ab
S	$\varnothing$	$\{S\}$	$\{S\}$	$\{A\}$
A	$\{\$\}$	$\{A\}$	$\{A\}$	$\varnothing$
$\$$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$

Clearly, $\mathscr{A}$ is a GFA. Now, we convert the GFA $\mathscr{A}$ to an equivalent NFA. Consider a new symbol B and split the production rule

$$S \to abA$$

of the grammar into the following two production rules

$$S \to aB \text{ and } B \to bA$$

and replace them in place of the earlier one. Note that, in the resultant grammar, the terminal strings that is occurring in the righthand sides of production rules are of lengths at most one. Hence, in a straightforward manner, we have the following NFA $\mathscr{A}'$ that is equivalent to the above GFA and also equivalent to the given regular grammar. $\mathscr{A}' = (Q', \Sigma, \delta', S, F)$, where $Q' = \{S, A, B, \$\}$ and $\delta' : Q' \times \Sigma \longrightarrow \wp(Q')$ is defined by the following table.

δ	ε	a	b
S	$\varnothing$	$\{S, B\}$	$\{S\}$
A	$\{\$\}$	$\{A\}$	$\{A\}$
B	$\varnothing$	$\varnothing$	$\{A\}$
$\$$	$\varnothing$	$\varnothing$	$\varnothing$

Example 4.5.16. The following an equivalent NFA for the regular grammar given in Example 3.3.13.

δ	ε	a	b
S	$\varnothing$	$\{A\}$	$\{S\}$
A	$\varnothing$	$\{B\}$	$\{A\}$
B	$\{\$\}$	$\{S\}$	$\{B\}$
$\$$	$\varnothing$	$\varnothing$	$\varnothing$

4.6 Variants of Finite Automata

Finite state transducers, two-way finite automata and two-tape finite automata are some variants of finite automata. Finite state transducers are introduced as devices which give output for a given input; whereas, others are language acceptors. The well-known Moore and Mealy machines are examples of finite state transducers All these variants are no more powerful than DFA; in fact, they are equivalent to DFA. In the following subsections, we introduce two-way finite automata and Mealy machines. Some other variants are described in Exercises.

4.6.1 Two-way Finite Automaton

In contrast to a DFA, the reading head of a two-way finite automata is allowed to move both left and right directions on the input tape. In each transition, the reading head can move one cell to its right or one cell to its left. Formally, a two-way DFA is defined as follows.

Definition 4.6.1. A *two-way DFA* (2DFA) is a quintuple $\mathscr{A} = (Q, \Sigma, \delta, q_0, F)$, where Q, Σ, q_0 and F are as in DFA, but the transition function is

$$\delta : Q \times \Sigma \longrightarrow Q \times \{L, R\},$$

where L is indicating a left move and R is indicating a right move.

A *configuration* (or *ID*) of a 2DFA is an element of $Q \times \Sigma^* \times \Sigma \times \Sigma^*$ or $Q \times \Sigma^*$. A configuration $(q, x, a, y) \in Q \times \Sigma^* \times \Sigma \times \Sigma^*$ indicates that the current state of the 2DFA is q and while the input is xay, the reading head is scanning the symbol a. For convenience, we write $(q, x\underline{a}y)$ instead of (q, x, a, y) by denoting the position of reading head with an underscore. If the reading head goes beyond the input, i.e. it has just finished reading the input, then there will not be any indication of reading head in a configuration. In which case, a configuration is of the form just (q, x) which is an element of $Q \times \Sigma^*$; this type of configurations will be at the end of a computation of a 2DFA with input x.

As usual for an automaton, a *computation* of a 2DFA is also given by relating two configurations C and C' with $\overset{*}{\vdash\!\!\!-}$, i.e. a computation is of the form $C \overset{*}{\vdash\!\!\!-} C'$, where $\vdash\!\!\!-$ is the one-step relation in a 2DFA which is defined as follows:

If $\delta(p, a) = (q, X)$, then the configuration $C = (p, x\underline{a}y)$ gives the configuration C' in *one-step*, denoted by $C \vdash\!\!\!- C'$, where C' is given by

Case-I: $X = R$.

$$C' = \begin{cases} (q, xa), & \text{if } y = \varepsilon; \\ (q, xa\underline{b}y'), & \text{whenever } y = by', \text{ for some } b \in \Sigma. \end{cases}$$

Case-II: $X = L$.

$$C' = \begin{cases} \text{undefined}, & \text{if } x = \varepsilon; \\ (q, x'\underline{b}ay'), & \text{whenever } x = x'b, \text{ for some } b \in \Sigma. \end{cases}$$

A string $x = a_1a_2\cdots a_n \in \Sigma^*$ is said to be accepted by a 2DFA $\mathscr{A}$ if $(q_0, \underline{a_1}a_2\cdots a_n) \overset{*}{\vdash} (p, a_1a_2\cdots a_n)$, for some $p \in F$. Further, the language accepted by $\mathscr{A}$ is

$$L(\mathscr{A}) = \{x \in \Sigma^* \mid x \text{ is accepted by } \mathscr{A}\}.$$

Example 4.6.2. Consider the language L over $\{0, 1\}$ that contain all those strings in which every occurrence of 01 is preceded by a 0, i.e. before every 01 there should be a 0. For example, 1001100010010 is in L; whereas, 101100111 is not in L. The following 2DFA given in transition table representation accepts the language L.

δ	0	1
$\rightarrow \textcircled{q_0}$	(q_1, R)	(q_0, R)
$\textcircled{q_1}$	(q_1, R)	(q_2, L)
q_2	(q_3, L)	(q_3, L)
q_3	(q_4, R)	(q_5, R)
q_4	(q_6, R)	(q_6, R)
q_5	(q_5, R)	(q_5, R)
q_6	(q_0, R)	(q_0, R)

The following computation on 10011 shows that the string is accepted by the 2DFA.

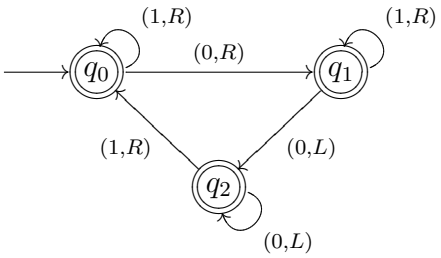
$$\begin{aligned} (q_0, \underline{1}0011) &\vdash (q_0, 1\underline{0}011) \vdash (q_1, 10\underline{0}11) \vdash (q_1, 100\underline{1}1) \\ &\vdash (q_2, 10\underline{0}11) \vdash (q_3, 1\underline{0}011) \vdash (q_4, 10\underline{0}11) \\ &\vdash (q_6, 100\underline{1}1) \vdash (q_0, 1001\underline{1}) \vdash (q_0, 10011). \end{aligned}$$

Given 1101 as input to the 2DFA, as the state component the final configuration of the computation

$$\begin{aligned} (q_0, \underline{1}101) &\vdash (q_0, 1\underline{1}01) \vdash (q_0, 11\underline{0}1) \vdash (q_1, 110\underline{1}) \\ &\vdash (q_2, 11\underline{0}1) \vdash (q_3, 1\underline{1}01) \vdash (q_5, 11\underline{0}1) \\ &\vdash (q_5, 11\underline{0}1) \vdash (q_5, 1101). \end{aligned}$$

is a non-final state, we observe that the string 1101 is not accepted by the 2DFA.

Example 4.6.3. Consider the language over $\{0, 1\}$ that contains all those strings with no consecutive 0′s. That is, any occurrence of two 1′s have to be separated by at least one 0. In the following we design a 2DFA which checks this parameter and accepts the desired language. We show the 2DFA using a transition diagram, where the left or right move of the reading head is indicated over the transitions.



4.6.2 Mealy Machines

As mentioned earlier, Mealy machine is a finite state transducer. In order to give output, in addition to the components of a DFA, we have a set of output symbols from which the transducer produces the output through an output function. In case of Mealy machine, the output is associated to each transition, i.e. given an input symbol in a state, while changing to the next state, the machine emits an output symbol. Thus, formally, a Mealy machine is defined as follows.

Definition 4.6.4. A *Mealy machine* is a sextuple $\mathscr{M} = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$, where Q, Σ, δ and q_0 are as in a DFA, whereas, Δ is a finite set called *output alphabet* and

$$\lambda : Q \times \Sigma \longrightarrow \Delta$$

is a function called *output function.*

In the depiction of a DFA, in addition to its components, viz. input tape, reading head and finite control, a Mealy machine has a writing head and an output tape. This is shown in the Figure 4.9.

Note that the output function λ assigns an output symbol for each state transition on an input symbol. Through the natural extension of λ to strings, we determine the output string corresponding to each input string. The extension can be formally given by the function

$$\hat{\lambda} : Q \times \Sigma^* \longrightarrow \Delta^*$$

that is defined by

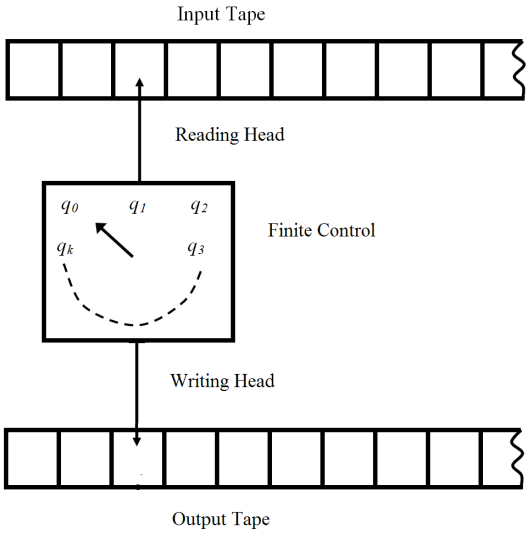


Figure 4.9: Depiction of Mealy Machine

1. $\hat{\lambda}(q, \varepsilon) = \varepsilon$, and
2. $\hat{\lambda}(q, xa) = \hat{\lambda}(q, x)\lambda(\hat{\delta}(q, x), a)$

for all $q \in Q$, $x \in \Sigma^*$ and $a \in \Sigma$. That is, if the input string $x = a_1 a_2 \cdots a_n$ is applied in a state q of a Mealy machine, then the output sequence

$$\hat{\lambda}(q, x) = \lambda(q_1, a_1)\lambda(q_2, a_2) \cdots \lambda(q_n, a_n)$$

where $q_1 = q$ and $q_{i+1} = \delta(q_i, a_i)$, for $1 \leq i < n$. Clearly, $|x| = |\hat{\lambda}(q, x)|$.

Definition 4.6.5. Let $\mathscr{M} = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$ be a Mealy machine. We say $\hat{\lambda}(q_0, x)$ is the *output* of $\mathscr{M}$ for an input string $x \in \Sigma^*$.

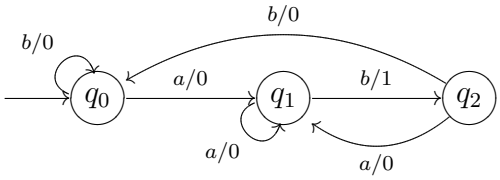
Example 4.6.6. Let $Q = \{q_0, q_1, q_2\}$, $\Sigma = \{a, b\}$, $\Delta = \{0, 1\}$ and define the transition function δ and the output function λ through the following tables.

δ	a	b
q_0	q_1	q_0
q_1	q_1	q_2
q_2	q_1	q_0

λ	a	b
q_0	0	0
q_1	0	1
q_2	0	0

Clearly, $\mathscr{M} = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$ is a Mealy machine. By incorporating the information of output, following the representation of DFA, $\mathscr{M}$ can be rep-

resented by a digraph as shown below.



For instance, output of $\mathscr{M}$ for the input string *baababa* is 0001010. In fact, this Mealy machine prints 1 for each occurrence of *ab* in the input; otherwise, it prints 0.

Example 4.6.7. In the following we construct a Mealy machine that performs binary addition. Given two binary numbers $a_1 \cdots a_n$ and $b_1 \cdots b_n$ (if they are different length then we put some leading $0'$s to the shorter one), the input sequence will be considered as we consider in the manual addition, as shown below.

$$\begin{pmatrix} a_n \\ b_n \end{pmatrix} \begin{pmatrix} a_{n-1} \\ b_{n-1} \end{pmatrix} \cdots \begin{pmatrix} a_1 \\ b_1 \end{pmatrix}$$

Here, we reserve $a_1 = b_1 = 0$ so as to accommodate the extra bit, if any, during addition. The expected output

$$c_n c_{n-1} \cdots c_1$$

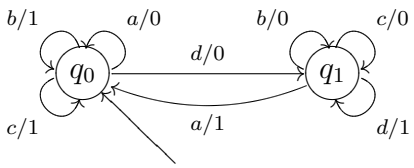
is such that

$$a_1 \cdots a_n + b_1 \cdots b_n = c_1 \cdots c_n.$$

Note that there are four input symbols, viz.

$$\begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \end{pmatrix} \text{ and } \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

For notational convenience, let us denote the above symbols by a, b, c and d, respectively. Now, the desired Mealy machine, while it is in the initial state, say q_0, if the input is d, i.e. while adding $1 + 1$, it emits the output 0 and remembers the carry 1 through a new state, say q_1. For other input symbols, viz. a, b and c, as there is no carry, it will continue in q_0 and performs the addition. Similarly, while the machine continues in q_1, for the input a, i.e. while adding $0 + 0$, it changes to the state q_0, indicating that the carry is 0 and emits 1 as output. Following this mechanism, the following Mealy machine is designed to perform binary addition.



Chapter 5

Properties of Regular Languages

In the previous chapters we have introduced various tools, viz. grammars, automata, to understand regular languages. Also, we have noted that the class of regular languages is closed with respect to certain operations like union, concatenation, Kleene closure. Now, with this information, can we determine whether a given language is regular or not? If a given language is regular, then to prove the same we need to use regular expression, regular grammar, finite automata or Myhill-Nerode theorem. Is there any other way to prove that a language is regular? The answer is "Yes". If a given language can be obtained from some known regular languages by applying those operations which preserve regularity, then one can ascertain that the given language is regular. If a language is not regular, although we have Myhill-Nerode theorem, a better and more practical tool viz. pumping lemma will be introduced to ascertain that the language is not regular. If we were somehow know that some languages are not regular, then again closure properties might be helpful to establish some more languages that are not regular. Thus, closure properties play important role not only in proving certain languages are regular, but also in establishing non-regularity of languages. Hence, we are indented to explore further closure properties of regular languages.

5.1 Closure Properties

5.1.1 Set Theoretic Properties

Theorem 5.1.1. *The class of regular languages is closed with respect to complement.*

Proof. Let L be a regular language accepted by a DFA $\mathscr{A} = (Q, \Sigma, \delta, q_0, F)$. Construct the DFA $\mathscr{A}' = (Q, \Sigma, \delta, q_0, Q - F)$, that is, by interchanging the roles of final and nonfinal states of $\mathscr{A}$. We claim that $L(\mathscr{A}') = L^c$ so that L^c is regular. For $x \in \Sigma^*$,

$$\begin{aligned} x \in L^c &\iff x \notin L \\ &\iff \hat{\delta}(q_0, x) \notin F \\ &\iff \hat{\delta}(q_0, x) \in Q - F \\ &\iff x \in L(\mathscr{A}'). \end{aligned}$$

□

Corollary 5.1.2. *The class of regular languages is closed with respect to intersection.*

Proof. If L_1 and L_2 are regular, then so are L_1^c and L_2^c. Then their union $L_1^c \cup L_2^c$ is also regular. Hence, $(L_1^c \cup L_2^c)^c$ is regular. But, by De Morgan's law

$$L_1 \cap L_2 = (L_1^c \cup L_2^c)^c$$

so that $L_1 \cap L_2$ is regular. □

Alternative Proof by Construction. For $i = 1, 2$, let $\mathscr{A}_i = (Q_i, \Sigma, \delta_i, q_i, F_i)$ be two DFA accepting L_i. That is, $L(\mathscr{A}_1) = L_1$ and $L(\mathscr{A}_2) = L_2$. Set the DFA

$$\mathscr{A} = (Q_1 \times Q_2, \Sigma, \delta, (q_1, q_2), F_1 \times F_2),$$

where δ is defined point-wise by

$$\delta((p, q), a) = (\delta_1(p, a), \delta_2(q, a)),$$

for all $(p, q) \in Q_1 \times Q_2$ and $a \in \Sigma$. We claim that $L(\mathscr{A}) = L_1 \cap L_2$. Using induction on $|x|$, first observe that $\hat{\delta}\Big((p, q), x\Big) = \Big(\hat{\delta}_1(p, x), \hat{\delta}_2(q, x)\Big)$, for all $x \in \Sigma^*$.

Now it clearly follows that

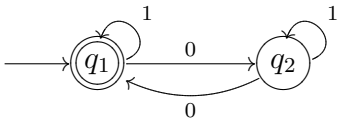
$$\begin{aligned} x \in L(\mathscr{A}) &\iff \hat{\delta}\Big((q_1, q_2), x\Big) \in F_1 \times F_2 \\ &\iff \Big(\hat{\delta}_1(q_1, x), \hat{\delta}_2(q_2, x)\Big) \in F_1 \times F_2 \\ &\iff \hat{\delta}_1(q_1, x) \in F_1 \text{ and } \hat{\delta}_2(q_2, x) \in F_2 \\ &\iff x \in L_1 \text{ and } x \in L_2 \\ &\iff x \in L_1 \cap L_2. \end{aligned}$$

□

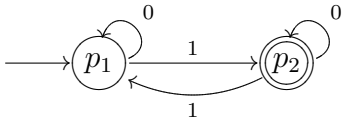
Example 5.1.3. Using the construction given in the above proof, we design a DFA that accepts the language

$$L = \{x \in (0+1)^* \mid |x|_0 \text{ is even and } |x|_1 \text{ is odd}\}$$

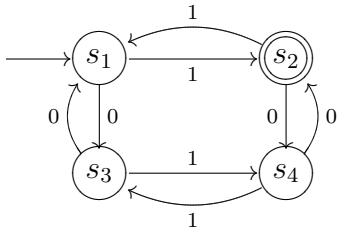
so that L is regular. Note that the following DFA accepts the language $L_1 = \{x \in (0+1)^* \mid |x|_0 \text{ is even}\}$.



Also, the following DFA accepts the language $L_2 = \{x \in (0+1)^* \mid |x|_1 \text{ is odd}\}$.



Now, let $s_1 = (q_1, p_1)$, $s_2 = (q_1, p_2)$, $s_3 = (q_2, p_1)$ and $s_4 = (q_2, p_2)$ and construct the automaton that accepts the intersection of L_1 and L_2 as shown below.



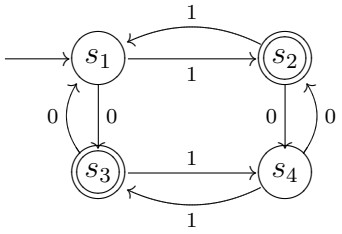
Further, we note the following regarding the automaton. If an input x takes the automaton (from the initial state) to the state

1. s_1, that means, x has even number of $0'$s and even number of $1'$s.
2. s_2, that means, x has even number of $0'$s and odd number of $1'$s (as desired in the current example).
3. s_3, that means, x has odd number of $0'$s and even number of $1'$s.
4. s_4, that means, x has odd number of $0'$s and odd number of $1'$s.

By choosing any combination of states among s_1, s_2, s_3 and s_4, appropriately, as final states we would get DFA which accept input with appropriate combination of $0's$ and $1'$s. For example, to show that the language

$$L' = \Big\{x \in (0+1)^* \;\Big|\; |x|_0 \text{ is even } \Leftrightarrow |x|_1 \text{ is odd}\Big\}.$$

is regular, we choose s_2 and s_3 as final states and obtain the following DFA which accepts L'.



Similarly, any other combination can be considered.

Corollary 5.1.4. *The class of regular languages is closed under set difference.*

Proof. Since $L_1 - L_2 = L_1 \cap L_2^c$, the result follows. □

Example 5.1.5. The language $L = \{a^n \mid n \geq 5\}$ is regular. We apply Corollary 5.1.4 with $L_1 = L(a^*)$ and $L_2 = \{\varepsilon, a, a^2, a^3, a^4\}$. Since L_1 and L_2 are regular, $L = L_1 - L_2$ is regular.

Remark 5.1.6. In general, one may conclude that the removal of finitely many strings from a regular language leaves a regular language.

5.1.2 Other Properties

Theorem 5.1.7. *If L is regular, then so is $L^R = \{x^R \mid x \in L\}$.*

To prove this we use the following lemma.

Lemma 5.1.8. *For every regular language L, there exists a finite automaton $\mathscr{A}$ with a single final state such that $L(\mathscr{A}) = L$.*

Proof. Let $\mathscr{A} = (Q, \Sigma, \delta, q_0, F)$ be a DFA accepting L. Construct $\mathscr{B} = (Q \cup \{p\}, \Sigma, \delta', q_0, \{p\})$, where $p \notin Q$ is a new state and δ' is given by

$$\delta'(q,a) = \begin{cases} \delta(q,a), & \text{if } q \in Q, a \in \Sigma \\ p, & \text{if } q \in F, a = \varepsilon. \end{cases}$$

Note that $\mathscr{B}$ is an NFA, which is obtained by adding a new state p to $\mathscr{A}$ that is connected from all the final states of $\mathscr{A}$ via ε-transitions. Here p is the only final state in $\mathscr{B}$ and all the final states of $\mathscr{A}$ are made nonfinal states. It is easy to prove that $L(\mathscr{A}) = L(\mathscr{B})$. □

Proof of the Theorem 5.1.7. Let $\mathscr{A}$ be a finite automaton with the initial state q_0 and single final state q_f that accepts L. Construct a finite automaton $\mathscr{A}^R$ by reversing the arcs in $\mathscr{A}$ with the same labels and by interchanging the roles of initial and final states. If $x \in \Sigma^*$ is accepted by $\mathscr{A}$, then there is a path q_0 to q_f labeled x in $\mathscr{A}$. Therefore, there will be a path from q_f to q_0 in $\mathscr{A}^R$ labeled x^R so that $x^R \in L(\mathscr{A}^R)$. Conversely, if x is accepted by $\mathscr{A}^R$, then using the similar argument one may notice that its reversal $x^R \in L(\mathscr{A})$. Thus, $L(\mathscr{A}^R) = L^R$ so that L^R is regular. □

Example 5.1.9. Consider the alphabet $\Sigma = \{a_0, a_1, \ldots, a_7\}$, where $a_i = \begin{pmatrix} b_i' \\ b_i'' \\ b_i''' \end{pmatrix}$ and $b_i' b_i'' b_i'''$ is the binary representation of decimal number i, for $0 \leq i \leq 7$. That is, $a_0 = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$, $a_1 = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$, $a_2 = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$, …, $a_7 = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$.

Now a string $x = a_{i_1} a_{i_2} \cdots a_{i_n}$ over Σ is said to represent correct binary addition if

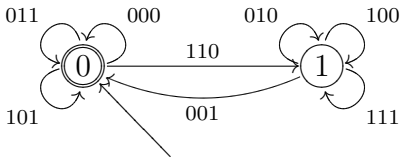
$$b'_{i_1} b'_{i_2} \cdots b'_{i_n} + b''_{i_1} b''_{i_2} \cdots b''_{i_n} = b'''_{i_1} b'''_{i_2} \cdots b'''_{i_n}.$$

For example, the string $a_5 a_1 a_6 a_5$ represents correct addition, because $1011 + 0010 = 1101$. Whereas, $a_5 a_0 a_6 a_5$ does not represent a correct addition, because $1011 + 0010 \neq 1001$.

We observe that the language L over Σ which contain all strings that represent correct addition, i.e.

$$L = \{a_{i_1} a_{i_2} \cdots a_{i_n} \in \Sigma^* \mid b'_{i_1} b'_{i_2} \cdots b'_{i_n} + b''_{i_1} b''_{i_2} \cdots b''_{i_n} = b'''_{i_1} b'''_{i_2} \cdots b'''_{i_n}\},$$

is regular. Consider the NFA shown in the following.



Note that the NFA accepts $L^R \cup \{\varepsilon\}$. Hence, by Remark 5.1.6, L^R is regular. Now, by Theorem 5.1.7, L is regular, as desired.

Definition 5.1.10. Let L_1 and L_2 be two languages over Σ. Right quotient of L_1 by L_2, denoted by L_1/L_2, is the language

$$\{x \in \Sigma^* \mid \exists y \in L_2 \text{ such that } \ xy \in L_1\}.$$

Example 5.1.11. 1. Let $L_1 = \{a, ab, bab, baba\}$ and $L_2 = \{a, ab\}$; then $L_1/L_2 = \{\varepsilon, bab, b\}$.

2. For $L_3 = 10^*1$ and $L_4 = 1$, we have $L_3/L_4 = 10^*$.

3. Let $L_5 = 0^*10^*$.

   (a) $L_5/0^* = L_5$.

   (b) $L_5/10^* = 0^*$.

   (c) $L_5/1 = 0^*$.

4. If $L_6 = a^*b^*$ and $L_7 = \{(a^nb^n)a^* \mid n \geq 0\}$, then $L_6/L_7 = a^*$.

Theorem 5.1.12. *If L is a regular language, then so is L/L', for any language L'.*

Proof. Let L be a regular language and L' be an arbitrary language. Suppose $\mathscr{A} = (Q, \Sigma, \delta, q_0, F)$ is a DFA which accepts L. Set $\mathscr{A}' = (Q, \Sigma, \delta, q_0, F')$, where

$$F' = \{q \in Q \mid \delta(q, x) \in F, \text{ for some } x \in L'\},$$

so that $\mathscr{A}'$ is a DFA. We claim that $L(\mathscr{A}') = L/L'$. For $w \in \Sigma^*$,

$$\begin{aligned} w \in L(\mathscr{A}') &\iff \hat{\delta}(q_0, w) \in F' \\ &\iff \hat{\delta}(q_0, wx) \in F, \text{ for some } x \in L' \\ &\iff w \in L/L'. \end{aligned}$$

Hence L/L' is regular. □

Note that Σ^* is a monoid with respect to the binary operation concatenation. Thus, for two alphabets Σ_1 and Σ_2, a mapping

$$h : \Sigma_1^* \longrightarrow \Sigma_2^*$$

is a homomorphism if, for all $x, y \in \Sigma_1^*$,

$$h(xy) = h(x)h(y).$$

One may notice that to give a homomorphism from Σ_1^* to Σ_2^*, it is enough to give images for the elements of Σ_1. This is because as we are looking for a homomorphism one can give the image of $h(x)$ for any $x = a_1a_2 \cdots a_n \in \Sigma_1^*$ by

$$h(a_1)h(a_2) \cdots h(a_n).$$

Therefore, a homomorphism from Σ_1^* to Σ_2^* is a mapping from Σ_1 to Σ_2^*.

Example 5.1.13. Let $\Sigma_1 = \{a, b\}$ and $\Sigma_2 = \{0, 1\}$. Define $h : \Sigma_1 \longrightarrow \Sigma_2^*$ by

$$h(a) = 10 \quad \text{and} \quad h(b) = 010.$$

Then, h is a homomorphism from Σ_1^* to Σ_2^*, which for example assigns the image 10010010 for the string *abb*.

We can generalize the concept of homomorphism by substituting a language instead of a string for symbols of the domain. Formally, a *substitution* is a mapping from Σ_1 to $\mathscr{P}(\Sigma_2^*)$.

Example 5.1.14. Let $\Sigma_1 = \{a, b\}$ and $\Sigma_2 = \{0, 1\}$. Define $h : \Sigma_1 \longrightarrow \mathscr{P}(\Sigma_2^*)$ by

$$\begin{aligned} h(a) &= \{0^n \mid n \geq 0\}, \text{ say } L_1; \\ h(b) &= \{1^n \mid n \geq 0\}, \text{ say } L_2. \end{aligned}$$

Then, h is a substitution. Now, for any string $a_1 a_2 \cdots a_n \in \Sigma_1^*$, its image under the above substitution h is

$$h(a_1 a_2 \cdots a_n) = h(a_1)h(a_2) \cdots h(a_n),$$

the concatenation of languages. For example, $h(ab)$ is the language

$$L_1 L_2 = \{0^m 1^n \mid m, n \geq 0\} = L(0^*1^*).$$

Given a substitution h from Σ_1 to Σ_2 one may naturally define $h(L)$ for a language L over Σ_1 by

$$h(L) = \bigcup_{x \in L} h(x).$$

Example 5.1.15. Consider the substitution h given in Example 5.1.14 and let $L = \{a^n b^n \mid n \geq 0\}$. For which,

$$\begin{aligned} h(L) &= \bigcup_{x \in L} h(x) \\ &= \bigcup_{n \geq 0} h(a^n b^n) \\ &= \bigcup_{n \geq 0} \overbrace{h(a) \cdots h(a)}^{n \text{ times}} \overbrace{h(b) \cdots h(b)}^{n \text{ times}} \\ &= \bigcup_{n \geq 0} \overbrace{0^* \cdots 0^*}^{n \text{ times}} \overbrace{1^* \cdots 1^*}^{n \text{ times}} \\ &= \{0^m 1^n \mid m, n \geq 0\} = 0^*1^*. \end{aligned}$$

Example 5.1.16. Define the substitution $h : \{a, b\} \longrightarrow \mathscr{P}(\{0, 1\}^*)$ by

$$\begin{aligned} h(a) &= \text{the set of strings over } \{0,\ 1\} \text{ ending with } 1; \\ h(b) &= \text{the set of strings over } \{0,\ 1\} \text{ starting with } 0. \end{aligned}$$

For the language $L = \{a^n b^m \mid n, m \geq 1\}$, we compute $h(L)$ through regular expressions described below.

Note that the regular expression for L is a^+b^+ and that are for $h(a)$ and $h(b)$ are $(0 + 1)^*1$ and $0(0 + 1)^*$. Now, write the regular expression that is obtained from the expression of L by replacing each occurrence of a by the expression of $h(a)$ and by replacing each occurrence of b by the expression of $h(b)$. That is, from a^+b^+, we obtain the regular expression

$$((0 + 1)^*1)^+(0(0 + 1)^*)^+.$$

This can be simplified as follows.

$$\begin{aligned} ((0 + 1)^*1)^+(0(0 + 1)^*)^+ &= ((0 + 1)^*1)^*((0 + 1)^*1)(0(0 + 1)^*)(0(0 + 1)^*)^* \\ &= ((0 + 1)^*1)^*(0 + 1)^*10(0 + 1)^*(0(0 + 1)^*)^* \\ &= (0 + 1)^*10(0 + 1)^*. \end{aligned}$$

The following Theorem 5.1.17 confirms that the expression obtained in this process represents $h(L)$. Thus, from the expression of $h(L)$, we can conclude that the language $h(L)$ is the set of all strings over $\{0, 1\}$ that have 10 as substring.

Theorem 5.1.17. *The class of regular languages is closed under substitutions by regular languages. That is, if h is a substitution on Σ such that $h(a)$ is regular for each $a \in \Sigma$, then $h(L)$ is regular for each regular language L over Σ.*

Proof. Let r be a regular expression for language L over Σ, i.e. $L(r) = L$, and, for each $a \in \Sigma$, let r_a be a regular expression for $h(a)$. Suppose r' is the expression obtained by replacing r_a for each occurrence of a (of Σ) in r so that r' is a regular expression. We claim that $L(r') = h(L(r))$ so that $h(L)$ is regular. We prove our claim by induction on the number of operations involved in r.

Assume that the number of operations involved in r is zero. Then there are three possibilities for r, viz. 1. $\varnothing$, 2. ε and 3. a for some $a \in \Sigma$.

1. If $r = \varnothing$, then $r' = \varnothing$ and $h(L(\varnothing)) = \varnothing$ so that the result is straightforward.

2. In case $r = \varepsilon$, $r' = \varepsilon$ and hence we have

$$h(L(r)) = h(\{\varepsilon\}) = \{\varepsilon\} = L(r').$$

3. For other case, let $r = a$, for some $a \in \Sigma$. Then, $r' = r_a$ so that

$$h(L(r)) = h(\{a\}) = L(r_a) = L(r').$$

Hence basis of the induction is satisfied.

For inductive hypothesis, assume $L(r') = h(L(r))$ for all those regular expressions r which have k or fewer operations. Now consider a regular expression r with $k+1$ operations. Then

$$r = r_1 + r_2 \quad \text{or} \quad r = r_1 r_2 \quad \text{or} \quad r = r_1^*$$

for some regular expressions r_1 and r_2. Note that both r_1 and r_2 have k or fewer operations. Hence, by inductive hypothesis, we have

$$L(r_1') = h(L(r_1)) \quad \text{and} \quad L(r_2') = h(L(r_2)),$$

where r_1' and r_2' are the regular expressions which are obtained from r_1 and r_2 by replacing r_a for each a in r_1 and r_2, respectively.

Consider the case where $r = r_1 + r_2$. The expression r' (that is obtained from r) is nothing else but replacing each r_a in the individual r_1 and r_2, we have

$$r' = r_1' + r_2'.$$

Hence,

$$\begin{aligned} L(r') &= L(r_1' + r_2') \\ &= L(r_1') \cup L(r_2') \\ &= h(L(r_1)) \cup h(L(r_2)) \\ &= h(L(r_1) \cup L(r_2)) \\ &= h(L(r_1 + r_2)) \\ &= h(L(r)) \end{aligned}$$

as desired, in this case. Similarly, other two cases, viz. $r = r_1 r_2$ and $r = r_1^*$, can be handled.

Hence, the class of regular languages is closed under substitutions by regular languages. □

Corollary 5.1.18. *The class of regular languages is closed under homomorphisms.*

Theorem 5.1.19. *Let $h : \Sigma_1 \longrightarrow \Sigma_2^*$ be a homomorphism and $L \subseteq \Sigma_2^*$ be regular. Then, inverse homomorphic image of L,*

$$h^{-1}(L) = \{x \in \Sigma_1^* \mid h(x) \in L\}$$

is regular.

Proof. Let $\mathscr{A} = (Q, \Sigma_2, \delta, q_0, F)$ be a DFA accepting L. We construct a DFA $\mathscr{A}'$ which accepts $h^{-1}(L)$. Note that, in $\mathscr{A}'$, we have to define transitions for the symbols of Σ_1 such that $\mathscr{A}'$ accepts $h^{-1}(L)$. We compose the homomorphism h with the transition map δ of $\mathscr{A}$ to define the transition map of $\mathscr{A}'$. Precisely, we set $\mathscr{A}' = (Q, \Sigma_1, \delta', q_0, F)$, where the transition map

$$\delta' : Q \times \Sigma_1 \longrightarrow Q$$

is defined by

$$\delta'(q, a) = \hat{\delta}(q, h(a))$$

for all $q \in Q$ and $a \in \Sigma_1$. Note that $\mathscr{A}'$ is a DFA. Now, for all $x \in \Sigma_1^*$, we prove that

$$\hat{\delta}'(q_0, x) = \hat{\delta}(q_0, h(x)).$$

This gives us $L(\mathscr{A}') = h^{-1}(L)$, because, for $x \in \Sigma_1^*$,

$$\begin{aligned} x \in h^{-1}(L) &\iff h(x) \in L \\ &\iff \hat{\delta}(q_0, h(x)) \in F \\ &\iff \hat{\delta}'(q_0, x) \in F \\ &\iff x \in L(\mathscr{A}'). \end{aligned}$$

We prove our assertion by induction on $|x|$. For basis, suppose $|x| = 0$. That is $x = \varepsilon$. Then clearly,

$$\hat{\delta}'(q_0, x) = q_0 = \hat{\delta}(q_0, \varepsilon) = \hat{\delta}(q_0, h(x)).$$

Here $h(\varepsilon) = \varepsilon$, because h is a homomorphism. Further, by definition of δ', we have

$$\hat{\delta}'(q_0, a) = \delta'(q_0, a) = \hat{\delta}(q_0, h(a)),$$

for all $a \in \Sigma_1^*$, so that the assertion is true for $|x| = 1$ also. For inductive hypothesis, assume

$$\hat{\delta}'(q_0, x) = \hat{\delta}(q_0, h(x))$$

for all $x \in \Sigma_1^*$ with $|x| = k$. Let $x \in \Sigma_1^*$ with $|x| = k$ and $a \in \Sigma_1$. Now,

$$\begin{aligned}
\hat{\delta}'(q_0, xa) &= \hat{\delta}'(\hat{\delta}'(q_0, x), a) \\
&= \hat{\delta}'(\hat{\delta}(q_0, h(x)), a) && \text{(by inductive hypothesis)} \\
&= \delta'(\hat{\delta}(q_0, h(x)), a) \\
&= \hat{\delta}(\hat{\delta}(q_0, h(x)), h(a)) && \text{(by definition of } \delta') \\
&= \hat{\delta}(q_0, h(x)h(a)) \\
&= \hat{\delta}(q_0, h(xa)). && \text{(by the property of homomorphism)}
\end{aligned}$$

Hence the result. □

Example 5.1.20. Suppose $L \subseteq (0+1)^*$ is a regular language. Now we observe that the language

$$L' = \{a_1 b_1 \cdots a_n b_n \in (0+1)^* \mid a_1 \cdots a_n \in L \text{ and } a_i = 0 \text{ iff } b_i = 1\}$$

is regular. For instance, define

$$f : \{0, 1\} \longrightarrow \{0, 1\}^*$$

by $f(0) = 01$ and $f(1) = 10$ so that f is a homomorphism. Now note that

$$\begin{aligned}
f(L) &= \{f(x) \mid x \in L\} \\
&= \{f(a_1 \cdots a_n) \mid a_1 \cdots a_n \in L\} \\
&= \{f(a_1) \cdots f(a_n) \mid a_1 \cdots a_n \in L\} \\
&= \{a_1 b_1 \cdots a_n b_n \mid a_1 \cdots a_n \in L \text{ and } a_i = 0 \text{ iff } b_i = 1\} \\
&= L'
\end{aligned}$$

Being homomorphic image of a regular language, L' is regular.

5.2 Pumping Lemma

Theorem 5.2.1 (Pumping Lemma)**.** *If L is an infinite regular language, then there exists a number κ (associated to L) such that for all $x \in L$ with $|x| \geq \kappa$, x can be written as uvw satisfying the following:*

1. *$v \neq \varepsilon$, and*
2. *$uv^i w \in L$, for all $i \geq 0$.*

Proof. Let $\mathscr{A} = (Q, \Sigma, \delta, q_0, F)$ be a DFA accepting L. Let $|Q| = \kappa$. Since L is infinite, there exists a string $x = a_1 a_2 \cdots a_n$ in L with $n \geq \kappa$, where $a_i \in \Sigma$. Consider the accepting sequence of x, say

$$q_0, q_1, \ldots, q_n$$

where, for $0 \leq i \leq n-1$, $\delta(q_i, a_{i+1}) = q_{i+1}$ and $q_n \in F$. As there are only κ states in Q, by pigeon-hole principle, at least one state must be repeated in the accepting sequence of x. Let $q_r = q_s$, for $0 \leq r < s \leq n$. Write $u = a_1 \cdots a_r$, $v = a_{r+1} \cdots a_s$ and $w = a_{s+1} \cdots a_n$; so that $x = uvw$. Note that, as $r < s$, we have $v \neq \varepsilon$. Now we will prove that $uv^i w \in L$, for all $i \geq 0$.

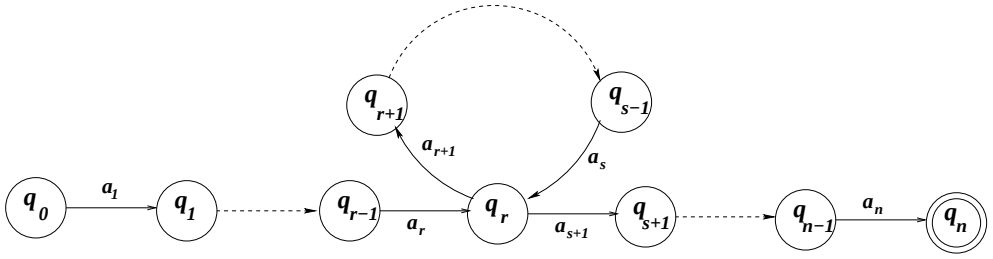


Figure 5.1: Pumping Sequence

For a given $i \geq 0$, $uv^i w = a_1 \cdots a_r (a_{r+1} \cdots a_s)^i a_{s+1} \cdots a_n$. Since $q_r = q_s$, we see that there is a computation for $uv^i w$ in $\mathscr{A}$, as given below:

$$\begin{aligned}
(q_0, uv^i w) &\vdash^* (q_r, v^i w) \\
&\vdash^* (q_s, v^{i-1w}) \\
&= (q_r, v^{i-1} w) \\
&\vdash^* (q_s, v^{i-2} w) \\
&\vdots \\
&\vdash^* (q_s, w) \\
&\vdash^* q_n
\end{aligned}$$

Thus, for $i \geq 0$, $uv^i w \in L$. □

Remark 5.2.2. If L is finite, then by choosing $\kappa = 1 + \max\{|x| \mid x \in L\}$ one may notice that L vacuously satisfies the pumping lemma, as there is no string of length greater than or equal to κ in L. Thus, the pumping lemma holds good for all regular languages.

A Logical Formula. If we write

$P(L)$: L satisfies pumping lemma and

$R(L)$: L is regular,

then the pumping lemma for regular languages is $R(L) \Longrightarrow P(L)$. The statement $P(L)$ can be elaborated by the following logical formula:

$$(\forall L)(\exists \kappa)(\forall x)\Big[x \in L \text{ and } |x| \geq \kappa \Longrightarrow (\exists u, v, w)\Big(x = uvw, v \neq \varepsilon \Longrightarrow (\forall i)(uv^i w \in L)\Big)\Big].$$

A Tool to Ascertain Non-regularity. If a language fails to satisfy the pumping lemma, then it cannot be regular. That is, $\neg P(L) \Longrightarrow \neg R(L)$ – the contrapositive form of the pumping lemma. The $\neg P(L)$ can be formulated as below:

$$(\exists L)(\forall \kappa)(\exists x)\Big[x \in L \text{ and } |x| \geq \kappa \text{ and } (\forall u, v, w)\Big(x = uvw, v \neq \varepsilon \text{ and } (\exists i)(uv^i w \notin L)\Big)\Big].$$

This statement can be better explained via the following adversarial game. Given a language L, if we want to show that L is not regular, then we play as given in the following steps.

1. An opponent will give us an arbitrary number κ.
2. Given κ, we pickup a string $x \in L$ with $|x| \geq \kappa$.
3. Opponent will divide x into u, v and w, arbitrarily, with $v \neq \varepsilon$. (Here $x = uvw$.)
4. We pickup an i such that $uv^i w \notin L$.

Example 5.2.3. We wish to show that $L = \{a^n b^n \mid n \geq 0\}$ is not regular. An opponent will give us a number κ. Then, we choose a string $x = a^m b^m \in L$ with $|x| \geq \kappa$. The opponent will give us u, v and w, where $x = uvw$ and $v \neq \varepsilon$. Now, there are three possibilities for v, viz. (1) a^p, for $p \geq 1$ (2) b^q, for $q \geq 1$ (3) $a^p b^q$, for $p, q \geq 1$. In each case, we give an i such that $uv^i w \notin L$, as shown below.

Case-1 ($v = a^p$, $p \geq 1$). Choose $i = 2$. Then $uv^i w = a^{m+k} b^m$ which is clearly not in L.

(In fact, except for $i = 1$, for all i, $uv^i w \notin L$)

Case-2 ($v = b^q$, $q \geq 1$). Similar to the Case-1, one may easily observe that except for $i = 1$, for all i, $uv^iw \notin L$. As above, say $i = 2$, then $uv^2w \notin L$.

Case-3 ($v = a^pb^q$, $p, q \geq 1$). Now, again choose $i = 2$. Note that

$$uv^iw = a^{m-p}(a^pb^q)^2b^{m-q} = a^mb^qa^pb^m.$$

Since $p, q \geq 1$, we observe that there is an a after a b in the resultant string. Hence $uv^2w \notin L$.

Hence, we conclude that L is not regular.

Example 5.2.4. We prove that $L = \{ww \mid w \in (0+1)^*\}$ is not regular. On contrary, assume that L is regular. Let κ be the pumping lemma constant associated to L. Now, choose $x = 0^n1^n0^n1^n$ from L with $|x| \geq \kappa$. If x is written as uvw with $v \neq \varepsilon$, then there are ten possibilities for v. In each case, we observe that pumping v will result a string that is not in L. This contradicts the pumping lemma so that L is not regular.

Case-1 (For $p \geq 1$, $v = 0^p$ with $x = 0^{k_1}v0^{k_2}1^n0^n1^n$).
Through the following points, in this case, we demonstrate a contradiction to pumping lemma.

1. In this case, v is in the first block of $0'$s and $p < n$.
2. Suppose v is pumped for $i = 2$.
3. If the resultant string uv^iw is of odd length, then clearly it is not in L.
4. Otherwise, suppose $uv^iw = yz$ with $|y| = |z|$.
5. Then, clearly, $|y| = |z| = \frac{4n+p}{2} = 2n + \frac{p}{2}$.
6. Since z is the suffix of the resultant string and $|z| > 2n$, we have $z = 1^{\frac{p}{2}}0^n1^n$.
7. Hence, clearly, $y = 0^{n+p}1^{n-\frac{p}{2}} \neq z$ so that $yz \notin L$.

Using a similar argument as given in Case-1 and the arguments shown in Example 5.2.3, one can demonstrate contradictions to pumping lemma in each of the following remaining cases.

Case-2 (For $q \geq 1$, $v = 1^q$ with $x = 0^n1^{k_1}v1^{k_2}0^n1^n$). That is, v is in the first block of $1'$s.

Case-3 (For $p \geq 1$, $v = 0^p$ with $x = 0^n1^n0^{k_1}v0^{k_2}1^n$). That is, v is in the second block of $0'$s.

Case-4 (For $q \geq 1$, $v = 1^q$ with $x = 0^n1^n0^n1^{k_1}v1^{k_2}$). That is, v is in the second block of $1'$s.

Case-5 (For $p, q \geq 1$, $v = 0^p1^q$ with $x = 0^{k_1}v1^{k_2}0^n1^n$). That is, v is in the first block of 0^n1^n.

Case-6 (For $p, q \geq 1$, $v = 1^p0^q$ with $x = 0^n1^{k_1}v0^{k_2}1^n$). That is, v is in the block of 1^n0^n.

Case-7 (For $p, q \geq 1$, $v = 0^p1^q$ with $x = 0^n1^n0^{k_1}v1^{k_2}$). That is, v is in the second block of 0^n1^n.

Case-8 (For $p, q \geq 1$, $v = 0^p1^n0^q$ with $x = 0^{k_1}v0^{k_2}1^n$). That is, v is in the block of $0^n1^n0^n$.

Case-9 (For $p, q \geq 1$, $v = 1^p0^n1^q$ with $x = 0^n1^{k_1}v1^{k_2}$). That is, v is in the block of $1^n0^n1^n$.

Case-10 (For $p, q \geq 1$, $v = 0^p1^n0^n1^q$ with $x = 0^{k_1}v1^{k_2}$). That is, v extended over all the blocks of 0's and $1'$s.

Hence, L is not regular.

Remark 5.2.5. Although it is sufficient to choose a particular string to counter the pumping lemma, it is often observed that depending on the string chosen there can be several possibilities of partitions as uvw that are to be considered as we have to check for all possibilities. For instance, in Example 5.2.3 we have discussed three cases. On the other hand, in Example 5.2.4 instead of choosing a typical string we have chosen a string which reduces the number of possibilities to discuss. Even then, there are ten possibilities to discuss.

In the following, we show how the number of possibilities, to be considered, can be reduced further. In fact, we observe that it is sufficient to consider the occurrence of v within the first κ symbols of the string under consideration. More precisely, we state the assertion through the following theorem, a restricted version of pumping lemma.

Theorem 5.2.6 (Pumping Lemma – A Restricted Version)**.** *If L is an infinite regular language, then there exists a number κ (associated to L) such that for all $x \in L$ with $|x| \geq \kappa$, x can be written as uvw satisfying the following:*

1. *$v \neq \varepsilon$,*
2. *$|uv| \leq \kappa$, and*
3. *$uv^iw \in L$, for all $i \geq 0$.*

Proof. The proof of the theorem is exactly same as that of Theorem 5.2.1, except that, when the pigeon-hole principle is applied to find the repetition of states in the accepting sequence, we find the repetition of states within the first $\kappa + 1$ states of the sequence. As $|x| \geq \kappa$, there will be at least $\kappa + 1$ states in the sequence and since $|Q| = \kappa$, there will be repetition in the first $\kappa + 1$ states. Hence, we have the desired extra condition $|uv| \leq \kappa$. □

Example 5.2.7. Consider that language $L = \{ww \mid w \in (0 + 1)^*\}$ given in Example 5.2.4. Using Theorem 5.2.6, we supply an elegant proof for L is not regular. If L is regular, then let κ be the pumping lemma constant associated to L. Now choose the string $x = 0^\kappa 1^\kappa 0^\kappa 1^\kappa$ from L. Using the Theorem 5.2.6, if x is written as uvw with $|uv| \leq \kappa$ and $v \neq \varepsilon$ then there is only one possibility for v in x, that is a substring of first block of $0'$s. Hence, clearly, by pumping v we get a string that is not in the language so that we can conclude L is not regular.

Example 5.2.8. We show that the language $L = \{x \in (a + b)^* \mid x = x^R\}$ is not regular. Consider the string $x = 0^\kappa 1^\kappa 1^\kappa 0^\kappa$ from L, where κ is pumping lemma constant associated to L. If x is written as uvw with $|uv| \leq \kappa$ and $v \neq \varepsilon$ then there is only one possibility for v in x, as in the previous example. Now, pumping v will result a string that is not a palindrome so that L is not regular.

Example 5.2.9. We show that the language

$$L = \{xcy \mid x, y \in \{a, b\}^* \text{ and } |x| = |y|\}$$

is not regular. On contrary, assume that L is regular. Then, since regular languages are closed with respect to intersection, $L' = L \cap a^*cb^*$ is regular. But, note that

$$L' = \{a^n cb^n \mid n \geq 0\},$$

because $xcy \in a^*cb^*$ and $|x| = |y|$ implies $xcy = a^n cb^n$. Now, using the homomorphism h that is defined by

$$h(a) = 0,\ h(b) = 1,\ \text{and } h(c) = \varepsilon$$

we have

$$h(L') = \{0^n 1^n \mid n \geq 0\}.$$

Since regular languages are closed under homomorphisms, $h(L')$ is also regular. But we know that $\{0^n 1^n \mid n \geq 0\}$ is not regular. Thus, we arrived at a contradiction. Hence, L is not regular.