Formal Languages and Automata Theory

Géza Horváth, Benedek Nagy

Formal Languages and Automata Theory

Géza Horváth, Benedek Nagy

Tartalom

Az ábrák listája

A táblázatok listája

Formal Languages and Automata Theory

Géza Horváth, Benedek Nagy

University of Debrecen, 2014

Typotex Publishing, www.typotex.hu

ISBN: 978-963-279-344-3

Készült a TÁMOP-4.1.2.A/1-11/1-2011-0103 azonosítószámú „Gyires Béla Informatika Tárház” című projekt keretében.

Introduction

Formal Languages and Automata Theory are one of the most important base fields of (Theoretical) Computer Science. They are rooted in the middle of the last century, and these theories find important applications in other fields of Computer Science and Information Technology, such as, Compiler Technologies, at Operating Systems, ... Although most of the classical results are from the last century, there are some new developments connected to various fields.

The authors of this book have been teaching Formal Languages and Automata Theory for 20 years. This book gives an introduction to these fields. It contains the most essential parts of these theories with lots of examples and exercises. In the book, after discussing some of the most important basic definitions, we start from the smallest and simplest class of the Chomsky hierarchy. This class, the class of regular languages, is well known and has several applications; it is accepted by the class of finite automata. However there are some important languages that are not regular. Therefore we continue with the classes of linear and context-free languages. These classes have also a wide range of applications, and they are accepted by various families of pushdown automata. Finally, the largest classes of the hierarchy, the families of context-sensitive and recursively enumerable languages are presented. These classes are accepted by various families of Turing machines. At the end of the book we give some further literature for those who want to study these fields more deeply and/or interested to newer developments.

The comments of the lector and some other colleagues are gratefully acknowledged.

Debrecen, 2014.

Géza Horváth and Benedek Nagy

1. fejezet - Elements of Formal Languages

Summary of the chapter: *In this chapter, we discuss the basic expressions, notations, definitions and theorems of the scientific field of formal languages and automata theory. In the first part of this chapter, we introduce the alphabet, the word, the language and the operations over them. In the second part, we show general rewriting systems and a way to define algorithms by rewriting systems, namely Markov (normal) algorithms. In the third part, we describe a universal method to define a language by a generative grammar. Finally, in the last part of this chapter we show the Chomsky hierarchy: a classification of generative grammars are based on the form of their production rules, and the classification of formal languages are based on the classification of generative grammars generating them.*

1. 1.1. Basic Terms

An alphabet is a finite nonempty set of symbols. We can use the union, the intersection and the relative complement set operations over the alphabet. The absolute complement operation can be used if a universe set is defined.

Example 1. *Let the alphabet E be the English alphabet, the alphabet V contains the vowels, and the alphabet C is the set of the consonants. Then $V\cup C=E$, $V\cap C=\emptyset$ and* $E\setminus V=\overline{V}=C.$

A word is a finite sequence of symbols of the alphabet. The length of the word is the number of symbols it is composed of, with repetitions. Two words are equal if they have the same length and they have the same letter in each position. This might sound trivial, but let us see the following example:

Example 2. *Let the alphabet $V=\{1,2,+\}$ and the words $p=1+1$, $q=2$. In this case $1+1\neq 2$, i.e. $p\neq q$, because these two words have different lengths, and also different letters in the first position.*

There is a special operation on words called concatenation, this is the operation of joining two words end to end.

Example 3. *Let the alphabet E be the English alphabet. Let the word p=railway, and the word q=station over the alphabet E. Then, the concatenation of p and q is $p\cdot q$=railwaystation. The length of p is $|p|=7$ and the length of q is $|q|=7$, as well.*

If there is no danger of confusion we can omit the dot from between the parameters of the concatenation operation. It is obvious that the equation $|pq|=|p|+|q|$ holds for all words p and q. There is a special word called an empty word, whose length is 0 and denoted by λ. The empty word is the identity element of the concatenation operation, $\lambda p=p\lambda=p$ holds for each word p over any alphabet. The word u is a prefix of the word p if there exists a word w such that $p=uw$, and the word w is a suffix of the word p if there exists a word u such that $p=uw$. The word v is a subword of word p if there exists words u,w such that $p=uvw$. The word u is a proper prefix of the word p, if it is a prefix of p, and the following properties hold: $u\neq\lambda$ and $u\neq p$. The word w is a proper suffix of the word p, if it is a suffix of p, and $w\neq\lambda$, $w\neq p$. The word v is a proper subword of the word p, if it is a subword of p, and $v\neq\lambda$, $v\neq p$. As you can see, both the prefixes and suffixes are subwords, and both the proper prefixes and proper suffixes are proper subwords, as well.

Example 4. *Let the alphabet E be the English alphabet. Let the word p = railwaystation, and the words u=rail, v = way and w = station. In this example, the word u is a prefix, the word w is a suffix and the word v is a subword of the word p. However, the word uv is also a prefix of the word p, and it is a subword of the word p, as well. The word uvw is a suffix of the word p, but it is not a proper suffix.*

We can use the exponentiation operation on a word in a classical way, as well. $p^0=\lambda$ by definition, $p^1=p$, and $p^i=p^{i-1}p$, for each integer $i\geq 2$. The union of p^i for each integer $i\geq 0$ is denoted by p^*, and the union of p^i for each integer $i\geq 1$ is denoted by p^+. $\left(p^*=\bigcup_{i=1}^{\infty}p^i \text{ and } p^+=\bigcup_{i=1}^{\infty}p^i.\right)$ These operations are called Kleene star and Kleene plus operations. $p^*=p^+\cup\{\lambda\}$ holds for each word p, and if $p\neq\lambda$ then $p^+=p^*\setminus\{\lambda\}$. We can also use the Kleene star and Kleene plus operations on an alphabet. For an alphabet V we denote the set of all words over the alphabet by V^*, and the set of all words, but the empty word by V^+.

A language over an alphabet is not necessarily a finite set of words, and it is usually denoted by L. For a given alphabet V the language L over V is $L \subseteq V^*$. We have a set again, so we can use the classical set operations, union, intersection, and the complement operation, if the opera $\overline{L} = V^* \setminus L.$ fined over the same alphabet. For the absolute complement operation we use V^* for universe, so We can also use the concatenation operation. The concatenation of the languages L_1 and L_2 contains all words pq where $p \in L_1$ and $q \in L_2$. The exponentiation operation is defined in a classical way, as well. $L^0=\{\lambda\}$ by definition, $L^1=L$, and $L^i=L^{i-1}\cdot L$, for each integer $i \geq 2$. The language L^0 contains exactly one word, the empty word. The empty language does not $L^* = \bigcup_{i=0}^{\infty} L^i$, and $L^+ = \bigcup_{i=1}^{\infty} L^i$, l $L^0 \neq L_e$. We can also use the Kleene star and Kleene plus closure for languages.

so $L^*=L^+$ if and only if $\lambda \in L$.

The algebraic approach to formal languages can be useful for readers who prefer the classical mathematical models. The free monoid on an alphabet V is the monoid whose elements are from V^*. From this point of view both operations, concatenation, (also called product) - which is not a commutative operation in this case -, and the union operation (also called addition), create a free monoid on set V, because these operations are associative, and they both have an identity element.

1. Associative:

   - $(L_1 \cdot L_2) \cdot L_3 = L_1 \cdot (L_2 \cdot L_3)$,
   - $(L_1 \cup L_2) \cup L_3 = L_1 \cup (L_2 \cup L_3)$,

   where $L_1, L_2, L_3 \subseteq V^*$.

2. Identity element:

   - $L^0 \cdot L_1 = L_1 \cdot L^0 = L_1$,
   - $L_e \cup L_1 = L_1 \cup L_e = L_1$,

   where $L_1 \subseteq V^*$, $L^0 = \{\lambda\}$, $L_e = \emptyset$.

The equation $L_e \cdot L_1 = L_1 \cdot L_e = L_e$ also holds for each $L_1 \subseteq V^*$.

2. 1.2. Formal Systems

In this section, the definition of basic rewriting systems in general and a specific one, the Markov algorithm is presented.

Definition 1. *A formal system (or rewriting system) is a pair $W = (V, P)$, where V is an alphabet and P is a finite subset of the Cartesian product $V^* \times V^*$. The elements of P are the (rewriting) rules or productions. If $(p, q) \in P$, then we usually write it in the form $p \rightarrow q$. Let $r, s \in V^*$, and we say that s can be directly derived (or derived in one step) from r (formally: $r \Rightarrow s$) if there are words $p, p', p'', q \in V^*$ such that $r = p'pp''$, $s = p'qp''$ and $p \rightarrow q \in P$. The word s can be derived from r (formally: $r \Rightarrow^* s$) if there is a sequence of words $r_0, r_1, ..., r_k$ ($k \geq 1$) such that $r_0 = r$, $r_k = s$ and $r_i \Rightarrow r_{i+1}$ holds for every $0 \leq i < k$. Moreover, we consider the relation $r \Rightarrow^* r$ for every word $r \in V^*$. (Actually, the relation $\Rightarrow^*$ is the reflexive and transitive closure of the relation $\Rightarrow$.*

A rewriting (or derivation) step can be understood as follows: the word s can be obtained from the word r in such a way that in s the right hand side q of a production of P is written instead of the left hand side p of the same production in r.

Example 5. *Let $W = (V, P)$ with $V = \{ a, b, c, e, l, n, p, r, t \}$ and $P = \{ able \rightarrow can, r \rightarrow c, pp \rightarrow b \}$. Then applerat $\Rightarrow$ applecat $\Rightarrow$ ablecat $\Rightarrow$ cancat, and thus applerat $\Rightarrow^*$ cancat in the system W.*

Now, we are going to present a special version of the rewriting systems that is deterministic, and was given by the Russian mathematician A. A. Markov as *normal algorithms*.

Definition 2. *$M = (V, P, H)$ is a Markov (normal) algorithm where*

- *V is a finite alphabet,*

- *P is a finite ordered set (list) of productions of the form* $V^* \times V^*$,
- $H \subseteq P$ *is the set of halting productions.*

The execution of the algorithm on input $w \in V^*$ *is as follows:*

1. *Find the first production in P such that its left-hand-side p is a subword of w. Let this production be* $p \to q$. *If there is no such a production, then step 5 applies.*
2. *Find the first (leftmost) occurrence of p in w (such that w = p'pp'' and p'p contains the subword p exactly once: as a suffix).*
3. *Replace this occurrence of p in w by the right hand side q of the production.*
4. *If* $p \to q \in H$, *i.e., the applied production is a halting production, then the algorithm is terminated with the obtained word (string) as an output.*
5. *If there are no productions in P that can be applied (none of their left-hand-side is a subword of w), then the algorithm terminates and w is its output.*
6. *If a rewriting step is done with a non halting production, then the algorithm iterates (from step 1) for the newly obtained word as w.*

We note here that rules of type $\lambda \to q$ also can be applied to insert word q to the beginning of the current (input) word.

The Markov algorithm may be terminated with an output, or may work forever (infinite loop).

Example 6. *Let* $M = (\{1,2,3\},\{21 \to 12, 31 \to 13, 32 \to 23\},\{\})$ *be a Markov algorithm. Then, it is executed to the input* 1321 *as follows:*

$$1321 \Rightarrow 1312 \Rightarrow 1132 \Rightarrow 1123.$$

Since there are no more applicable productions, it is terminated. Actually, it is ordering the elements of the input word in a non-decreasing way.

Example 7. *Let*

$$W=(\{a,b,c\},\{aa \to bbb,\ ac \to bab,\ bc \to a\},\{\}).$$

Let us apply the algorithm W to the input

ababacbbaacaa.

As it can be seen in Animation 1 [3], the output is

ababbabbbbbabbb.

(For better understanding in the Animation the productions of the algorithm is numbered.)

Animation 1.

b,*c*},{1.*aa*→*bbb*,2.*ac*→*bab*,3.*bc*-

:

ababacbbaacaa

Exercise 1. *Execute the Markov algorithm*

$$M = (\{\, 1, + \,\}, \{\, 1+ \to +1, ++ \to +, + \to \lambda \,\}, \{\, + \to \lambda \,\})$$

on the following input words:

- 1 + 1 + 1 + 1,
- 11 + 11 + 11,
- 111,
- 111 + 1 + 11.

Exercise 2. *Execute the Markov algorithm* $M = (\{1, \times, a, b\},$

$\{\, \times 11 \to a \times 1, \times 1 \to a, 1a \to a1b, ba \to ab, b1 \to 1b, a1 \to a, ab \to b, b \to 1 \,\}, \{\,\})$

on the following input words:

- 1 × 1,
- 11 × 11,
- 111 × 1,
- 111 × 11.

The previous two exercises are examples for Markov algorithms that compute unary addition and unary multiplication, respectively.

It is important to know that this model of the algorithm is universal in the sense that every problem that can be solved in an algorithmic way can be solved by a Markov algorithm as well.

In the next section, other variations of the rewriting systems are shown: the generative grammars, which are particularly highlighted in this book. As we will see, the generative grammars use their productions in a non-deterministic way.

3. 1.3. Generative Grammars

The generative grammar is a universal method to define languages. It was introduced by Noam Chomsky in the 1950s. The formal definition of the generative grammar is the following:

Definition 3. *The generative grammar* (G) *is the following quadruple:*

$$G = (N, T, S, P)$$

where

- *N is the set of the nonterminal symbols, also called variables, (finite nonempty alphabet),*
- *T is the set of the terminal symbols or constants (finite nonempty alphabet),*
- *S is the start symbol, and*
- *P is the set of the production rules.*

The following properties hold: $N \cap T = \emptyset$ and $S \in N$.

Let us denote the union of the sets N and T by V ($V = N \cup T$). Then, the form of the production rules is $V^*N V^* \to V^*$.

Informally, we have two disjoint alphabets, the first alphabet contains the so called start symbol, and we also have a set of productions. The productions have two sides, both sides are words over the joint alphabet, and the word on the left hand side must contain a nonterminal symbol. We have a special notation for the production rules. Let the word p be the left hand side of the rule, and the word q be the right hand side of the rule, then we use the $p \to q \in P$ expression.

Example 8. *Let the generative grammar G be $G = (N, T, S, P)$, where*

```
        N = {S,A},
T = {a,b}, andP = {
  S → bAbS,
  bAb → bSab,
  A → λ,
  S → aa
}.
```

In order to understand the operation of the generative grammar, we have to describe how we use the production rules to generate words. First of all, we should give the definition of the derivation.

Definition 4. *Let $G = (N, T, S, P)$ be a generative grammar, and let p and q be words over the joint alphabet $V = N \cup T$. We say that the word q can be derived in one step from the word p, if p can be written in a form $p = uxw$, q can be written in a form $q = uyw$, and $x \to y \in P$. (Denoted by $p \Rightarrow q$.) The word q can be derived from the word p, if $q = p$ or there are words $r_1, r_2,..., r_n$ such that $r_1 = p$, $r_n = q$ and the word r_i can be derived in one step from the word r_{i-1}, for each $2 \leq i \leq n$. (Denoted by $p \Rightarrow^* q$.)*

Now, that we have a formal definition of the derivation, let us see an example for deeper understanding.

Example 9. *Let the generative grammar G be $G = (N, T, S, P)$, where*

```
        N = {S,A},
T = {0,1}, andP = {
  S → 1,
  S → 1A,
  A → AA,
  A → 0,
  A → 1
}.
```

Let the words p,t and q be $p = A0S0$, $t = A01A0$, and $q = A0110$. In this example, the word t can be derived from the word p in one step, $(p \to t)$, because p can be written in a form $p = uxw$, where $u = A0$, $x = S$, $w = 0$, and the word t can be written in a form $t = uyw$, where $y = 1A$, and $S \to 1A \in P$.

The word q can be derived from the word p, $(p \Rightarrow^* q)$, because there exist words r_1,r_2 and r_3 such that $r_1 = p$, $r_2 = t$, $r_3 = q$ and $r_1 \Rightarrow r_2$ and $r_2 \Rightarrow r_3$.

Now we have all the definitions to demonstrate how we can use the generative grammar to generate a language.

Definition 5. *Let $G = (N, T, S, P)$ be a generative grammar. The language generated by the grammar G is*

$$L(G)=\{p \mid p \in T^*, S \Rightarrow^* p\}.$$

The above definition claims that the language generated by the grammar G contains each word over the terminal alphabet which can be derived from the start symbol S.

Example 10. *Let the generative grammar G be $G = (N, T, S, P)$, where*

```
        N = {S,A},
T = {a,b}, andP = {
  S → bb,
  S → ASA,
  A → a
}.
```

In this example, the word bb can be derived from the start symbol S in one step, because $S \to bb \in P$, so the word bb is in the generated language, $bb \in L(G)$.

The word abba can be derived from the start symbol, because $S \Rightarrow ASA$, $ASA \Rightarrow aSA$, $aSA \Rightarrow abbA$, $abbA \to abba$, so the word abba is also in the generated language, $abba \in L(G)$.

The word bab can not be derived from the start symbol, so it is not in the generated language, $bab \notin L(G)$.

In this case, it is easy to determine the generated language, $L(G) = \{a^i bba^i \mid i \geq 0\}$.

Exercise 3. *Create a generative grammar G, which generates the language $L = \{a^* b^+ c^*\}$!*

Exercise 4. *Create a generative grammar G, which generates the language of all binary numbers!*

4. 1.4. Chomsky Hierarchy

The Chomsky hierarchy was described first by Noam Chomsky in 1956. It classifies the generative grammars based on the forms of their production rules. The Chomsky hierarchy also classifies languages, based on the classes of generative grammars generating them. The formal definition is provided below.

Definition 6. (Chomsky hierarchy) *Let $G = (N, T, S, P)$ be a generative grammar.*

- *Type 0 or unrestricted grammars. Each generative grammar is unrestricted.*
- *Type 1 or context-sensitive grammars. G is called context-sensitive, if all of its production rules have a form*

  $$p_1 A p_2 \to p_1 q p_2,$$

  or

  $$S \to \lambda,$$

where p_1, $p_2 \in V^*$, $A \in N$ and $q \in V^+$. If $S \to \lambda \in P$ then S does not appear in the right hand side word of any other rule.

- *Type 2 or context-free grammars. The grammar G is called context-free, if all of its productions have a form*

  $$A \to p,$$

  where $A \in N$ and $p \in V^*$.

- *Type 3 or regular grammars. G is called regular, if all of its productions have a form*

  $$A \to r,$$

  or

  $$A \to rB,$$

  where $A,B \in N$ and $r \in T^*$.

Example 11. *Let the generative grammar G_0 be*

```
        G
        0 = ({S,X},{x,y},S,P)
P = {
  S → SXSy,
  XS → y,
  X → SXS,
  S → yxx
  }.
```

This grammar is unrestricted, because the second rule is not a context-sensitive rule.

Example 12. *Let the generative grammar G_1 be*

```
        G
        1 = ({S,X},{x,y},S,P)
P = {
  S → SXSy,
  XSy → XyxXy,
  S → yXy,
  X → y
  }.
```

This grammar is context-sensitive, because each production rule has an appropriate form.

Example 13. *Let the generative grammar G_2 be*

```
        G
        2 = ({S,X},{x,y},S,P)
P = {
  S → SyS,
  S → XX,
  S → yxy,
  X → ySy,
  X → λ
}.
```

This grammar is context-free, because the left hand side of each production rule is a nonterminal symbol.

Example 14. *Let the generative grammar G be*

```
        G = ({S,X},{x,y},S,P)
P = {
  S → xyS,
  S → X,
  X → yxS,
  S → x,
  X → λ
}.
```

This grammar is regular, because the left hand side of each production rule is a nonterminal, and the right hand side contains at most one nonterminal symbol, in last position.

Exercise 5. *What is the type of the following grammars?*

```
      1. G = ({S,A},{0,1},S,P)
     P = {
     S → 0101A,
     S → λ,
     A → 1S,
     A → 000
     }.

  2. G = ({S,A,B},{0,1},S,P)
     P = {
     S → 0A01B,
     S → λ,
     0A01 → 01A101,
     A → 0BB1,
     B → 1A1B,
     B → 0011,
     A → 1
     }.

  3. G = ({S,A,B},{0,1},S,P)
     P = {
     S → 0ABS1,
     S → 10,
     0AB → 01SAB,
     1SA → 111,
     A → 0,
     B → 1
     }.

  4. G = ({S,A},{0,1},S,P)
     P = {
     S → 00A,
     S → λ,
     A → λ,
     A → S1S
     }.
```

Definition 7. *The language L is regular, if there exists a regular grammar G such that L = L (G).*

The same kind of definitions are given for the context-free, context-sensitive, and recursively enumerable languages:

Definition 8. *The language L is context-free, if there exists a context-free grammar G such that L = L (G).*

Definition 9. *The language L is context-sensitive, if there exists a context-sensitive grammar G such that L = L (G).*

Definition 10. *The language L is called recursively enumerable, if there exists an unrestricted grammar G such that L = L (G).*

Although these are the most often described classes of the Chomsky hierarchy, there are also a number of subclasses which are worth investigating. For example, in chapter 3 we introduce the linear languages. A grammar is linear, if it is context-free, and the right hand side of its rules contain maximum one nonterminal symbol. The class of linear languages is a proper subset of the class of context-free languages, and it is a proper superset of the regular language class. Example 15. [9] shows a typical linear language and a typical linear grammar.

Example 15. *Let L be the language of the palindromes over the alphabet T = {a,b}. (Palindromes are words that read the same backward or forward.) Language L is generated by the following linear grammar.*

```
        G = ({S},{a,b},S,P)
P = {
  S → aSa,
  S → bSb,
  S → a,
  S → b,
  S → λ
}.
```

Language L is linear, because it can be generated by the linear grammar G.

Example 16. *In Example 9 [5] we can see a context-free grammar, which is not linear, because the production $A \to AA$ is not a linear rule. However, this context-free grammar generates a regular language, because the same language can be generated by the following regular grammar.*

```
        G = ({S,A},{0,1},S,P, andP = {
  S → 1A,
  A → 1A,
  A → 0A,
 A → λ
}.
```

It is obvious that context-sensitive grammars are unrestricted as well, because each generative grammar is unrestricted. It is also obvious that regular grammars are context-free as well, because in regular grammars the left hand side of each rule is a nonterminal, which is the only condition to be satisfied for a grammar in order to be context-free.

Let us investigate the case of the context-free and context sensitive grammars.

Example 17. *Let the grammar G be*

```
        G = ({S,A},{x,y},S,P)
P = {
  S → AxA,
  A → SyS,
  A → λ,
  S → λ
}.
```

This grammar is context-free, because the left hand side of each rule contains one nonterminal. At the same time, this grammar is not context-sensitive, because

- *the rule $A \to \lambda$ is not allowed in context-sensitive grammars,*
- *if $S \to \lambda \in P$, then the rule $A \to SyS$ is not allowed.*

This example shows that there are context-free grammars which are not context-sensitive. Although this statement holds for grammars, we can show that in the case of languages the Chomsky hierarchy is a real hierarchy, because each context-free language is context-sensitive as well. To prove this statement, let us consider the following theorem.

Theorem 1. *For each context-free grammar G we can give context-free grammar G', which is context-sensitive as well, such that $L(G) = L(G')$.*

Proof. *We give a constructive proof of this theorem. We are going to show the necessary steps to receive an equivalent context-sensitive grammar G' for a context-free grammar $G = (N, T, S, P)$.*

1. First, we have to collect each nonterminal symbol from which the empty word can be derived. To do this, let the set $U(1)$ be $U(1) = \{A \mid A \in N, A \to \lambda \in P\}$. Then let $U(i) = U(i\text{-}1) \cup \{A \mid A \to B_1B_2 \ldots B_n \in P, B_1, B_2, \ldots,$

$B_n \in U(i\text{-}1)\}$. Finally, we have an integer i such that $U(i) = U(i\text{-}1) = U$ which contains all of the nonterminal symbols from which the empty word can be derived.

2. Second, we are going to create a new set of rules. The right hand side of these rules should not contain the empty word. Let $P' = (P \cup P_1)\backslash\{A\rightarrow \lambda | \ A \in N\}$. The set P_1 contains production rules as well. $B \rightarrow p \in P_1$ if $B \rightarrow q \in P$ and we get p from q by removing some letters contained in set U.

3. Finally, if $S \notin U$, then $G' = (N, T, S, P')$. If set U contains the start symbol, then the empty word can be derived from S, and $\lambda \in L(G)$. In this case, we have to add a new start symbol to the set of nonterminals, and we also have to add two new productions to the set P_1, and $G' = (N \cup \{S'\}, T, S', P' \cup \{S' \rightarrow \lambda, S' \rightarrow S\})$, so G' generates the empty word, and it is context sensitive.

QED.

Example 18. *Let the context-free grammar G be*

```
        G = ({S,A,B},{x,y},S,P)
P = {
  S → ASxB,
  S → AA,
  A → λ,
  B → SyA
}.
```

Now, we create a context-sensitive generative grammar G', such that L (G') = L (G).

```
        1. U (1) = {A},
     U (2) = {A,S} = U.

  2. P' = {
     S → ASxB, S → SxB, S → AxB, S → xB,
     S → AA, S → A,
     B → SyA, B → yA, B → Sy, B → y
     }.

  3. G' = ({S,A,B,S'},{x,y},S',P' ∪{S' → λ, S' → S}).
```

Exercise 6. *Create a context-sensitive generative grammar G', such that L (G') = L (G)!*

```
        G = ({S,A,B,C},{a,b},S,P)
P = {
  S → aAbB,
  S → aCCb,
  C → λ,
  A → C,
  B → ACC,
  A → aSb
}.
```

Exercise 7. *Create a context-sensitive generative grammar G', such that L (G') = L (G)!*

```
        G = ({S,X,Y},{0,1},S,P)
P = {
  S → λ,
  S → XXY,
  X → Y0Y1,
  Y → 1XS1,
  X → S00S
}.
```

In the following chapters we are going to investigate the language classes of the Chomsky hierarchy. We will show algorithms to decide if a word is in a language generated by a generative grammar. We will learn methods which help to specify the position of a language in the Chomsky hierarchy, and we are going to investigate the closure properties of the different language classes.

We will introduce numerous kinds of automata. Some of them accept languages, and we can use them as an alternative language definition tool, however, some of them calculate output words from the input word, and we can see them as programmable computational tools. First of all, in the next chapter, we are going to deal with the most simple class of the Chomsky hierarchy, the class of regular languages.

2. fejezet - Regular Languages and Finite Automata

Summary of the chapter: *In this chapter, we deal with the simplest languages of the Chomsky hierarchy, i.e., the regular languages. We show that they can be characterized by regular expressions. Another description of this class of languages can be given by finite automata: both the class of deterministic finite automata and the class of nondeterministic finite automata accept this class. The word problem (parsing) can be solved in real-time for this class by the deterministic finite automata. This class is closed under regular and under set-theoretical operations. This class also has a characterization in terms of analyzing the classes of possible continuations of the words (Myhill-Nerode theorem). We also present Mealy-type and Moore-type transducers: finite transducers are finite automata with output.*

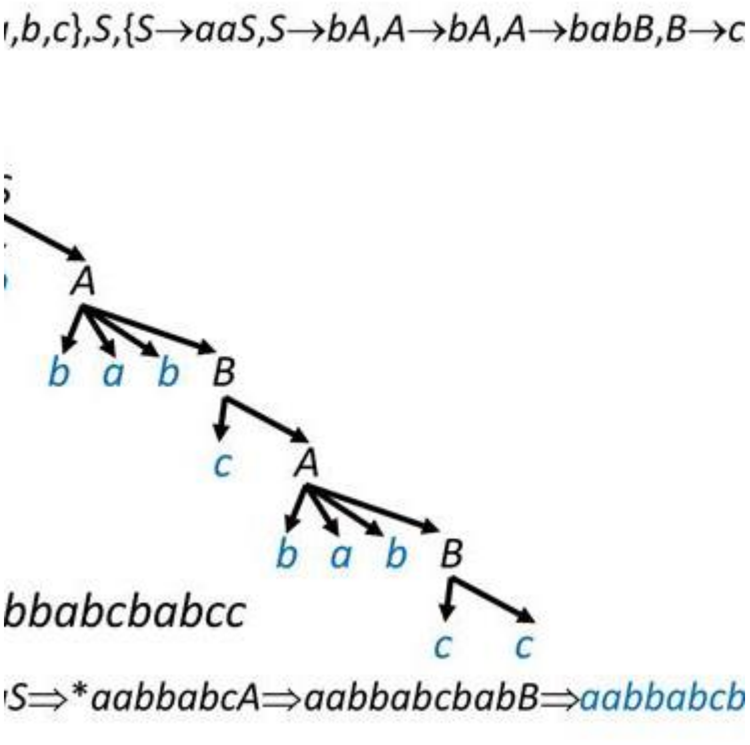
1. 2.1. Regular Grammars

In order to be comprehensive we present the definition of regular grammars here again.

Definition (Regular grammars). *A grammar G=(N,T,S,P) is regular if each of its productions has one of the following forms: $A \to u$, $A \to uB$, where $A,B \in N$, $u \in T^*$. The languages that can be generated by regular grammars are the regular languages (they are also called type 3 languages of the Chomsky hierarchy).*

Animation 2. [12] presents an example for a derivation in a regular grammar.

Animation 2.


,b,c},S,{S→aaS,S→bA,A→bA,A→babB,B→c
A
b a b B
c A
b a b B
c c
bbabcbabcc
S⇒*aabbabcA⇒aabbabcbabB⇒aabbabcb

We note here that the grammars and languages of the definition above are commonly referred to as right-linear grammars and languages, and regular grammars and languages are defined in a more restricted way:

Definition 12. (Alternative definition of regular grammars). *A grammar G = (N, T, S, P) is regular if each of its productions has one of the following forms: $A \to a$, $A \to aB$, $S \to \lambda$, where $A,B \in N$, $a \in T$. The languages that can be generated by these grammars are the regular languages.*

Now we show that the two definitions are equivalent in the sense that they define the same class of languages.

Theorem 2. *The language classes defined by our original definition and by the alternative definition coincide.*

Proof. The proof consists of two parts: we need to show that languages generated by grammars of one definition can also be generated by grammars of the other definition.

It is clear the grammars satisfying the alternative definition satisfy our original definition at the same time, therefore, every language that can be generated by the grammars of the alternative definition can also be generated by grammars of the original definition.

Let us consider the other direction. Let a grammar $G = (N, T, S, P)$ may contain rules of types $A \to u$ and $A \to uB$ ($A,B \in N$, $u \in T^*$). Then, we give a grammar $G' = (N', T, S', P')$ such that it may only contain rules of the forms $A \to a$, $A \to aB$, $S' \to \lambda$ (where $A,B \in N'$, $a \in T$) and it generates the same language as G, i.e., $L(G) = L(G')$.

First we obtain a grammar G'' such that $L(G) = L(G'')$ and $G'' = (N'', T, S, P'')$ may contain only rules of the following forms: $A \to a$, $A \to aB$, $A \to B$, $A \to \lambda$ (where $A,B \in N''$, $a \in T$). Let us check every rule in P: if it has one of the forms above, then we copy it to the set P'', else we will do the following:

- if the rule is of the form $A \to a_1 \ldots a_k$ for $k > 1$, where $a_i \in T$, $i \in \{1,\ldots,k\}$, then let the new nonterminals $X_1,\ldots, X_{k-1}$ be introduced (new set for every rule) and added to the set N'' and instead of the actual rule the next set of rules is added to P'': $A \to a_1X_1$, $X_1 \to a_2X_2$, ..., $X_{k-2} \to a_{k-1}X_{k-1}$, $X_{k-1} \to a_k$.

- if the rule is of the form $A \to a_1 \ldots a_kB$ for $k > 1$, where $a_i \in T$, $i \in \{1,\ldots,k\}$, then let the new nonterminals $X_1,\ldots, X_{k-1}$ be introduced (new set for every rule) and put to the set N'', and instead of the actual rule the next set of rules is added to P'': $A \to a_1X_1$, $X_1 \to a_2X_2$, ..., $X_{k-2} \to a_{k-1}X_{k-1}$, $X_{k-1} \to a_kB$.

When every rule is analyzed (and possibly replaced by a set of rules) we have grammar G''. It is clear that the set of productions P'' of G'' may contain only rules of the forms $A \to a$, $A \to aB$, $A \to a$, $A \to \lambda$ (where $A,B \in N''$, $a \in T$), since we have copied only rules from P that have these forms, and all the rules that are added to the set of productions P'' by replacing a rule are of the forms $A \to aB$, $A \to a$ (where $A,B \in N''$, $a \in T$). Moreover, exactly the same words can be derived in G'' and in G. The derivation graphs in the two grammars can be mapped in a bijective way. When a derivation step is applied in G with a rule $A \to a_1\ldots$ that is not in P'', then the rules must be used in G'' that are used to replace the rule $A \to a_1\ldots$: applying the first added rule first $A \to a_1X_1$, then there is only one rule that can be applied in G'', since there is only one rule added with X_1 in its left hand side... therefore, one needs to apply all the rules of the chain that was used to replace the original rule $A \to a_1\ldots$. Then, if there is a nonterminal B at the end of the rule $A \to a_1\ldots$, then it is located at the end of the last applied rule of the chain, and thus the derivation can be continued in the same way in both grammars. The other way around, if we use a newly introduced rule in the derivation in G'', then we must use all the rules of the chain, and also we can use the original rule that was replaced by this chain of rules in the grammar G. In this way, there is a one-to-one correspondence in the completed derivations of the two grammars. (See Figure 2.1. for an example of replacing a long rule by a sequence of shorter rules.)

2.1. ábra - In derivations the rules with long right hand side are replaced by chains of shorter rules, resulting binary derivation trees in the new grammar.

Now we may have some rules in P'' that do not satisfy the alternative definition. The form of these rules can only be $A \to B$ and $C \to \lambda$ (where $A,B,C \in N''$, $C \neq S$). The latter types of rules can be eliminated by the Empty-word lemma (see Theorem 1. [9]). Therefore, we can assume that we have a grammar $G''' = (N''', T, S', P''')$ such that $L(G''') = L(G)$ and P''' may contain only rules of the forms $A \to aB$, $A \to B$, $A \to a$, $S' \to \lambda$ (where $A,B \in N'''$, $a \in T$ and in case $S' \to \lambda \in P'''$ the start symbol S' does not occur on the right hand side of any of the rules of P'''). Let us define the following sets of nonterminals:

- let $U_1(A) = \{A\}$,

- let $U_{i+1}(A) = U_i(A) \cup \{B \in N''' | \exists\ C \in U_i(A)$ such that $C \to B \in P'''\}$, for $i > 1$.

Since N''' is finite there is a minimal index k such that $U_k(A) = U_{k+1}(A)$. Let $U(A)$ denote the set $U_k(A)$ with the above property. In this way, $U(A)$ contains exactly those nonterminals that can be derived from A by using rules only of the form $B \to C$ (where $B,C \in N'''$). We need to replace the parts $A \Rightarrow^* B \to r$ of the derivations in G''' by a direct derivation step in our new grammar, therefore, let $G' = (N''', T, S', P')$, where $P' = \{A \to r \mid \exists\ B \in N'''$ such that $B \to r \in P''', r \notin N'''$ and $B \in U(A)\}$. Then G' fulfills the alternative definition, moreover, it generates the same language as G''' and G.

QED.

Further we will call a grammar a *regular grammar in normal form* if it satisfies our alternative definitions. In these grammars the structures of the rules are more restricted: if we do not derive the empty word, then we can use only rules that have exactly one terminal in their right hand side.

Example 19. *Let*

$G = (\{S,A,B\},\{0,1,2\},S,$

```
{   S → 010B,
    A → B,
    A → 2021,
    B → 2A,
    B → S,
    B → λ
}).
```

Give a grammar that is equivalent to G and is in normal form.

Solution:

Let us exclude first the rules that contain more than one terminal symbols. Such rules are $S \to 010B$ *and* $A \to 2021$. *Let us substitute them by the sets*

$\{S \to 0X_1, X_1 \to 1X_2, X_2 \to 0B\}$

and

$\{A \to 2X_3, X_3 \to 0X_4, X_4 \to 2X_5, X_5 \to 1\}$

of rules, respectively, where the newly introduced nonterminals are $\{X_1, X_2\}$ *and* $\{X_3, X_4, X_5\}$, *respectively. The obtained grammar is*

$G'' = (\{S, A, B, X_1, X_2, X_3, X_4, X_5\},\{0,1,2\},S,$

```
{   S → 0X1,
    X1 → 1X2,
    X2 → 0B,
    A → B,
    A → 2X3,
    X3 → 0X4,
    X4 → 2X5,
    X5 → 1,
    B → 2A,
    B → S,
    B → λ
 ).
```

Now, by applying the Empty-word lemma, we can exclude rule $B \to \lambda$ *(the empty word can be derived from the nonterminals B and A in our example) and by doing so we obtain grammar*

$G''' = (\{S, A, B, X_1, X_2, X_3, X_4, X_5\}, \{0,1,2\}, S,$

```
{   S → 0X1,
    X1 → 1X2,
    X₂ → 0B,
    X₂ → 0,
    A → B,
    A → 2X₃,
    X₃ → 0X₄,
    X₄ → 2X₅,
    X₅ → 1,
    B → 2A,
    B → 2,
    B → S
}).
```

Now we are excluding the chain rules $A \longrightarrow B$, $B \longrightarrow S$. To do this step, first, we obtain the following sets:

$U_0(S) = \{S\}$		$U_0(A) = \{A\}$		$U_0(B) = \{B\}$
$U_1(S) = \{S\} = U(S)$		$U_1(A) = \{A,B\}$		$U_1(B) = \{B,S\}$
		$U_2(A) = \{A,B,S\}$		$U_2(B) = \{B,S\} = U(B)$
		$U_3(A) = \{A,B,S\} = U(A)$		

Actually for those nonterminals that are not appearing in chain rules these sets are the trivial sets, e.g., $U(X_1) = U_0(X_1) = \{X_1\}$. Thus, finally, we obtain grammar

$G' = (\{S, A, B, X_1, X_2, X_3, X_4, X_5\}, \{0,1,2\}, S,$

```
{   S → 0X₁,
    A → 0X₁,
    B → 0X₁,
    X₁ → 1X₂,
    X₂ → 0B,
    X₂ → 0,
    A → 2X₃,
    X₃ → 0X₄,
    X₄ → 2X₅,
    X₅ → 1,
    B → 2A,
    A → 2A,
    B → 2,
    A → 2
}).
```

Since our transformations preserve the generated language, every obtained grammar (G'', G''' and also G') is equivalent to G. Moreover, G' is in normal form. Thus the problem is solved.

Exercise 8. *Let*

$G = (\{S, A, B, C\},\{a,b\},S,$

```
{   S → abA,
    S → A,
    A → B,
    B → abab,
    B → aA,
    B → aaC,
    B → λ,
    C → aaS
}).
```

Give a grammar that is equivalent to G and is in normal form.

Exercise 9. *Let*

$G = (\{S, A, B, C\},\{a,b,c\},S,$

```
{   S → A,
    S → B,
    A → aaB,
    B → A,
    B → acS,
    B → C,
    C → c,
    C → λ
}).
```

Give a grammar that is equivalent to G and is in normal form.

Exercise 10. *Let*

$G = (\{S, A, B, C, D\},\{a,b,c\},S,$

```
{   S → aA,
    S → bB,
    A → B,
    B → A,
    B → ccccC,
    B → acbcB,
    C → caacA,
    C → cba
}).
```

Give a grammar that is equivalent to G and is in normal form.

Exercise 11. *Let*

$G = (\{S, A, B, C, D\},\{1,2,3,4\},S,$

```
{   S → 11A,
    S → 12B,
    A → B,
    B → C,
    B → 14,
    B → 4431,
    C → 3D,
    D → 233C
}).
```

Give a grammar that is equivalent to G and is in normal form.

2. 2.2. Regular Expressions

In this section, we will describe the regular languages in another way. First, we define the syntax and the semantics of regular expressions, and then we show that they describe the same class of languages as the class that can be generated by regular grammars.

Definition 13. (Regular expressions). *Let T be an alphabet and $V = T \cup \{\emptyset, \lambda, +, \cdot, {}^*, (,)\}$ be defined as its extension, such that $T \cap \{\emptyset, \lambda, +, \cdot, {}^*, (,)\} = \emptyset$. Then, we define the regular expressions over T as expressions over the alphabet V in an inductive way:*

- *Base of the induction:*
  - *$\emptyset$ is a (basic) regular expression,*
  - *λ is a (basic) regular expression,*
  - *every $a \in T$ is a (basic) regular expression;*
- *Induction steps*

- *if p and r are regular expressions, then $(p+r)$ is also a regular expression,*
- *if p and r are regular expressions, then $(p \cdot r)$ is also a regular expression (we usually eliminate the sign $\cdot$ and write (pr) instead $(p \cdot r)$),*
- *if r is a regular expression, then r^* is also a regular expression.*

Further, every regular expression can be obtained from the basic regular expressions using finite number of induction steps.

Definition 14 (Languages described by regular expressions). *Let T be an alphabet. Then, we define languages described by the regular expressions over the alphabet T following their inductive definition.*

- *Basic languages:*
  - *$\emptyset$ refers to the empty language $\{\}$,*
  - *λ refers to the language $\{\lambda\}$,*
  - *for every $a \in T$, the expression refers to the language $\{a\}$;*
- *Induction steps:*
  - *if p and r refer to the languages L_p and L_r, respectively, then the regular expressions $(p+r)$ refers to the language $L_p \cup L_r$,*
  - *if p and r refer to the languages L_p and L_r, respectively, then the regular expressions $(p \cdot r)$ or (pr) refers to the language $L_p \cdot L_r$,*
  - *if r refers to a language L_r then r^* refers to the language L_r^*.*

The language operations used in the above definition are the *regular operations*:

- the addition (+) is the union of two languages,
- the product is the concatenation of two languages, and
- the reflexive and transitive closure of concatenation is the (Kleene-star) iteration of languages.

Two regular expressions are equivalent if they describe the same language. Here are some examples:

$(p+r)$	$\equiv$	$(r+p)$	(commutativity of union)
$((p+r)+q)$	$\equiv$	$(p+(r+q))$	(associativity of union)
$(r+\emptyset)$	$\equiv$	r	(additive zero element, unit element for union)
$((pr)q)$	$\equiv$	$(p(rq)$	(associativity of concatenation)
$(r\lambda)$	$\equiv$	$(\lambda r) \equiv r$	(unit element for concatenation)
$((p+r)q)$	$\equiv$	$((pq)+(rq))$	(left distributivity)
$(p(r+q))$	$\equiv$	$((pr)+(pq))$	(right distributivity)
$(r\emptyset)$	$\equiv$	$\emptyset$	(zero element for concatenation)
λ^*	$\equiv$	λ	(iteration of the unit element)
$\emptyset^*$	$\equiv$	λ	(iteration of the zero element)
(rr^*)	$\equiv$	(r^*r)	(positive iteration)

Actually, the values of the last equivalence are frequently used, and they are denoted by r^+, i.e., $r^+ \equiv rr^*$ by definition. This type of iteration which does not allow for 0 times iteration, i.e., only positive numbers of iterations are allowed, is usually called Kleene-plus iteration.

With the use of the above equivalences we can write most of the regular expressions in a shorter form: some of the brackets can be eliminated without causing any ambiguity in the language described. The elimination of the brackets is usually based on the associativity (both for the union and for the concatenation). Some further brackets can be eliminated by fixing a precedence order among the regular operations: the unary operation (Kleene-)star is the strongest, then the concatenation (the product), and (as usual) the union (the addition) is the weakest.

We have regular grammars to generate languages and regular expressions to describe languages, but these concepts are not independent. First we will prove one direction of the equivalence between them.

Theorem 3. *Every language described by a regular expression can be generated by a regular grammar.*

Proof. The proof goes by induction based on the definition of regular expressions. Let r be a basic regular expression, then

- if r is $\emptyset$, then the empty language can be generated by the regular grammar

  $$(S, A, T, S, \{A \to a\});$$

- if r is λ, then the language $\{\lambda\}$ can be generated by the regular grammar

  $$(S, T, S, \{S \to \lambda\});$$

- if r is a for a terminal $a \in T$, then the language $\{a\}$ is generated by the regular grammar

  $$(S, T, S, \{S \to a\}).$$

If r is not a basic regular expression then the following cases may occur:

- r is $(p+q)$ with some regular expressions p,q such that the regular grammars $G_p = (N_p, T, S_p, P_p)$ and $G_q = (N_q, T, S_q, P_q)$ generate the languages described by expression p and q, respectively, where $N_p \cap N_q = \emptyset$ (this can be done by renaming nonterminals of a grammar without affecting the generated language). Then let

  $$G = (N_p \cup N_q \cup \{S\}, T, S, P_p \cup P_q \cup \{S \to S_p, S \to S_q\}),$$

  where $S \notin N_p \cup N_q$ is a new symbol. It can be seen that G generates the language described by expression r.

- r is $(p \cdot q)$ with some regular expressions p,q such that the regular grammars $G_p = (N_p, T, S_p, P_p)$ and $G_q = (N_q, T, S_q, P_q)$ generate the languages described by expression p and q, respectively, where $N_p \cap N_q = \emptyset$. Then let

  $$G = (N_p \cup N_q, T, S_p, P_q \cup \{A \to uB \mid A \to uB \in P_p, A,B \in N_p, u \in T^*\} \cup \{A \to uS_q \mid A \to u \in P_p, A \in N_p, u \in T^*\}).$$

  It can be shown that G generates the language described by expression r.

- r is a regular expression of the form q^* for a regular expression q. Further let $G_q = (N_q, T, S_q, P_q)$ be a regular grammar that generates the language described by expression q. Then, let

  $$G = (N_q \cup \{S\}, T, S, P_q \cup \{S \to \lambda, S \to S_q\} \cup \{A \to uS_q \mid A \to u \in P_q, A \in N_q, u \in T^*\}),$$

  where $S \notin N_q$. It can be shown that G generates the language described by expression r.

Since every regular expression built by finitely many applications of the induction step, for any regular expression one can construct a regular grammar such that the grammar generates the language described by the expression.

QED.

When we want to have a grammar that generates the language given by a regular expression, the process can be faster, if we know that every singleton, i.e., language containing only one word, can easily be obtained by a grammar having only one nonterminal (the start symbol) and only one production that allows to generate the given word in one step.

Example 20. *Let* $r = (cbb)^*(ab+ba)$ *be a regular expression (that describes a language over the alphabet* $\{a,b,c\}$*). Give a regular grammar that generates this language.*

Solution:

Let us build up r *from its subexpressions. According to the above observation, the language* $\{cbb\}$ *can be generated by the grammar*

$$G_{cbb} = (\{S_{cbb}\},\{a,b,c\},S_{cbb},\{S_{cbb} \rightarrow cbb\}).$$

Now, let us use our induction step to obtain the grammar $G_{(cbb)^*}$ *that generates the language* $(cbb)^*$*: then*

$$G_{(cbb)^*} = (\{S_{cbb}, S_{(cbb)^*}\},\{a,b,c\},S_{(cbb)^*},$$

$$\{S_{cbb} \rightarrow ccb, S_{(cbb)^*} \rightarrow \lambda, S_{(cbb)^*} \rightarrow S_{cbb}, S_{(cbb)} \rightarrow cbb\, S_{(cbb)}\}).$$

The languages $\{ab\}$ *and* $\{ba\}$ *can be generated by grammars*

$$G_{ab} = (\{S_{ab}\},\{a,b,c\},S_{ab},\{S_{ab} \rightarrow ab\})$$

and

$$G_{ab} = (\{S_{ba}\},\{a,b,c\},S_{ba},\{S_{ba} \rightarrow ba\}),$$

respectively. Their union, $ab+ba$*, then is generated by the grammar*

$$G_{ab+ba} = (\{S_{ab},S_{ba},S_{ab+ba}\},\{a,b,c\},S_{ab+ba},$$

$$\{S_{ab} \rightarrow ab, S_{ba} \rightarrow ba, S_{ab+ba} \rightarrow S_{ab}, S_{ab+ba} \rightarrow S_{ba}\})$$

according to our induction step.

Finally, we need the concatenation of the previous expressions $(cbb)^*$ *and* $(ab+ba)$*, and it is generated by the grammar*

$$G_{(cbb)^*(ab+ba)} = (\{S_{cbb}, S_{(cbb)^*},S_{ab},S_{ba},S_{ab+ba}\}, \{a,b,c\}, S_{(cbb)^*},$$

$$\{S_{cbb} \rightarrow cbbS_{ab+ba}, S_{(cbb)^*} \rightarrow S_{ab+ba}, S_{(cbb)^*} \rightarrow S_{cbb}, S_{(cbb)^*} \rightarrow cbb\, S_{(cbb)^*},$$

$$S_{ab} \rightarrow ab, S_{ba} \rightarrow ba, S_{ab+ba} \rightarrow S_{ab}, S_{ab+ba} \rightarrow S_{ba}\})$$

due to our induction step. The problem is solved.

Exercise 12. *Give a regular expression that describes the language containing exactly those words that contain three consecutive a's over the alphabet* $\{a,b\}$.

Exercise 13. *Give a regular expression that describes the language containing exactly those words that do not contain two consecutive a's (over the alphabet* $\{a,b\}$*).*

Exercise 14. *Give a regular grammar that generates the language* $0^*(1+22)(2^*+00)$.

Exercise 15. *Give a regular grammar that generates the language* $0+1(1+0)^*$.

Exercise 16. *Give a regular grammar that generates the language* $(a+bb(b+(cc)^*))^*(ababa+c^*)$.

3. 2.3. Finite Automata as Language Recognizers

In this section we first define several variations of the finite automata distinguished by the properties of the transition function.

Definition 15 (Finite automata). *Let $A = (Q, T, q_0, \delta, F)$. It is a finite automaton (recognizer), where Q is the finite set of (inner) states, T is the input (or tape) alphabet, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final (or accepting) states and δ is the transition function as follows.*

- $\delta : Q \times (T \cup \{\lambda\}) \to 2^Q$ (*for nondeterministic finite automata with allowed λ-transitions*);
- $\delta : Q \times T \to 2^Q$ (*for nondeterministic finite automata without λ-transitions*);
- $\delta : Q \times T \to Q$ (*for deterministic finite automata, λ can be partially defined*);
- $\delta : Q \times T \to Q$ (*for completely defined deterministic finite automata* (*it is not allowed that δ is partial function, it must be completely defined*).

One can observe, that the second variation is a special case of the first one (not having λ-transitions). The third variation is a special case of the second one having sets with at most one element as images of the transition function, while the fourth case is more specific allowing sets exactly with one element.

One can imagine a finite automaton as a machine equipped with an input tape. The machine works on a discrete time scale. At every point of time the machine is in one of its states, then it reads the next letter on the tape (the letter under the reading head), or maybe nothing (in the first variations), and then, according to the transition function (depending on the actual state and the letter being read, if any) it goes to a/the next state. It may happen in some variations that there is no transition defined for the actual state and letter, then the machine gets stuck and cannot continue its run.

There are two widely used ways to present automata: by Cayley tables or by graphs. When an automaton is given by a Cayley table, then the 0th line and the 0th column of the table are reserved for the states and for the alphabet, respectively (and it is marked in the 0th element of the 0th row). In some cases it is more convenient to put the states in the 0th row, while in some cases it is a better choice to put the alphabet there. We will look at both possibilities. The initial state should be the first among the states (it is advisable to mark it by a → sign also). The final states should also be marked, they should be circled. The transition function is written into the table: the elements of the set $\delta(q,a)$ are written (if any) in the field of the column and row marked by the state q and by the letter a. In the case when λ-transitions are also allowed, then the 0th row or the column (that contains the symbols of the alphabet) should be extended by the empty word (λ) also. Then λ-transitions can also be indicated in the table.

Automata can also be defined in a graphical way: let the vertices (nodes, that are drawn as circles in this case) of a graph represent the states of the automaton (we may write the names of the states into the circles). The initial state is marked by an arrow going to it not from a node. The accepting states are marked by double circles. The labeled arcs (edges) of the graph represent the transitions of the automaton. If $p \in \delta(q,a)$ for some $p,q \in Q$, $a \in T \cup \{\lambda\}$, then there is an edge from the circle representing state q to the circle representing state p and this edge is labeled by a. (Note that our graph concept is wider here than the usual digraph concept, since it allows multiple edges connecting two states, in most cases these multiple edges are drawn as normal edges having more than 1 labels.)

In this way, implicitly, the alphabet is also given by the graph (only those letters are used in the automaton which appear as labels on the edges).

In order to provide even more clarification, we present an example. We describe the same nondeterministic automaton both by a table and by a graph.

Example 21. *Let an automaton be defined by the following Cayley table:*

T Q	$\to q_0$	q_1	$\subset q_2 \supset$	$\subset q_3 \supset$
a	q_1	q_1	q_2, q_3	-
b	q_0	q_0	-	q_3
c	q_0	q_2	-	q_1, q_2, q_3

Figure 2.2. shows the graph representation of the same automaton.

2.2. ábra - The graph of the automaton of Example 21 [20].

These automata are used to accept words, and thus, languages:

Definition 16. (Language accepted by finite automaton). *Let $A = (Q, T, q_0, \delta, F)$ be an automaton and $w \in T^*$ be an input word. We say that w is accepted by A if there is a run of the automaton, i.e., there is an alternating sequence $q_0\ t_1\ q_1 \ldots q_{k-1}\ t_k\ q_k$ of states and transitions, that starts with the initial state q_0, ($q_i \in Q$ for every i, they are not necessarily distinct, e.g., $q_i = q_j$ is allowed even if $i \neq j$) and for every of its transition t_i of the sequence*

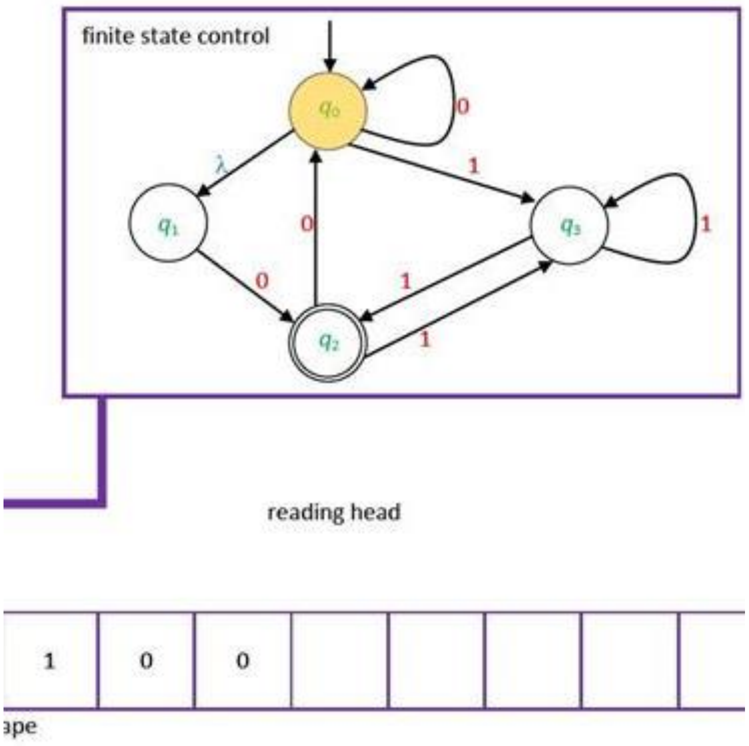
- $t_i : q_i \in \delta(q_{i-1}, a_i)$ *in nondeterministic cases*
- $t_i : q_i = \delta(q_{i-1}, a_i)$ *in deterministic cases,*

where $a_1 \ldots a_k = w$, and $q_k \in F$. This run is called an accepting run.

All words that A accepts form $L(A)$, the language accepted (or recognized) by the automaton A.

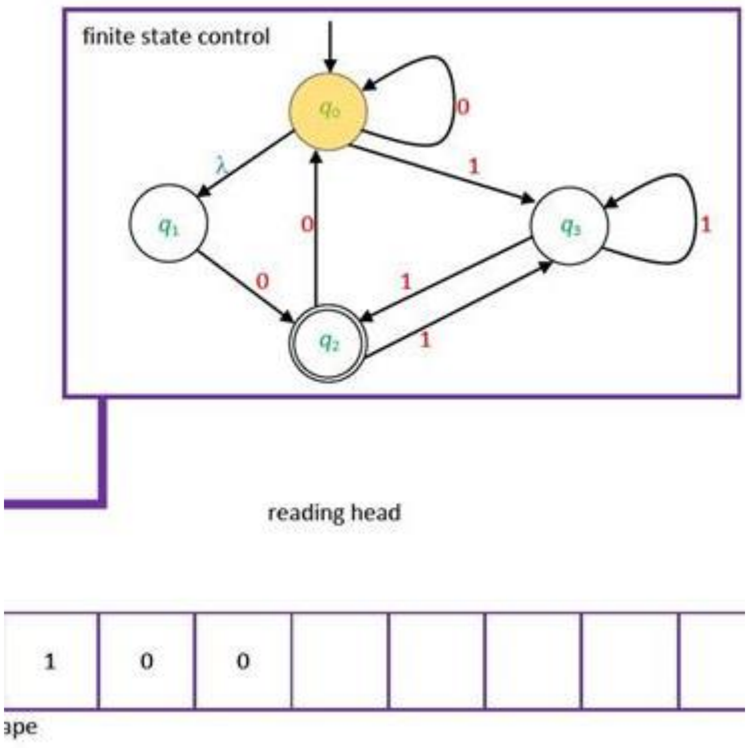
Example 22. *Let A be the automaton drawn in the next animations. We show a non-accepting run of a non-deterministic automaton A (with λ-transitions) in Animation 3. [21]*

Animation 3.



However the word 1100 is accepted by A, since it has also an accepting run that is shown in Animation 4. [21]

Animation 4.



These finite automata are also called finite-state acceptors or Rabin-Scott automata. Let us see the language class(es) that can be accepted by these automata.

Two automata are equivalent if they accept the same language.

We have defined four types of finite automata and by the definition it seems that the latter ones are more restricted than the former ones. However, it turns out that all four versions characterize the same language class:

Theorem 4. *For every finite automaton there is an equivalent (completely defined) deterministic finite automaton.*

Proof. The proof is constructive. Let $A = (Q, T, q_0, \delta, F)$ be a nondeterministic finite automaton (allowing λ-transitions). Let us define, first, the λ-closure of an arbitrary set q' of states.

- let $U_1(\{q'\}) = \{q'\}$,
- let $U_{i+1}(\{q'\}) = U_i(q') \cup \{p \in Q | \ \exists r \in U_i(q')$ such that $p \in \delta(r,\lambda)\}$, for $i > 1$.

Since Q is finite, there is a value k such that $U_k(q') = U_{k+1}(q')$, let us denote this set by $U(q')$. Practically, this set contains all the states that can be reached starting from a state of q' by only λ-transitions.

Now we are ready to construct the automaton $A' = (Q', T, U(q_0), \delta', F')$, where $Q' = 2^Q$, $F' \subset Q'$ includes every element $q' \in Q'$ such that $q' \cap F \neq \emptyset$. The transition function δ' is defined as follows: $\delta'(q',a) = \bigcup_{q \in q'} U(\delta(q,a)),$ for any $a \in T$ and $q' \in Q'$. Actually while this can be done for all subsets of Q, subsets which cannot be reached by transitions from $U(q_0)$ by δ' can be deleted (these useless states are not needed).

One can observe that A' is a completely defined deterministic automaton. Also, every run of A has an equivalent run of A', in fact, A' simulates every possible run of A on the input at the same time. Conversely, if A' has an accepting run, then A also has at least one accepting run for the same input. Therefore, A and A' accept the same language, consequently they are equivalent.

QED

Our previous proof gives an algorithm for the "determinization" of any finite automaton having only states reachable from the initial state as we will see it in details in Example 23. [23] Note that even if we deleted these useless states, the automaton may not be minimal in the sense that the same language can be accepted by a completely defined deterministic finite automaton having less number of states than our automaton.

Example 23. *Let a nondeterministic automaton be defined by the following Cayley table (note that in this algorithm the rows refer to the states of the automaton and the columns to the letters of the alphabet, and in this automaton λ-transitions are allowed):*

Q T	***a***	***b***	**λ**
$\rightarrow q_0$	q_0,q_1	q_2	-
q_1	q_1	-	q_2
$\subset q_2 \supset$	q_0	q_1	-

We start with the λ-closure of the initial state $U(q_0) = \{q_0\}$. This set will count as the initial state of the new automaton: let it be in the first row of the table of this new automaton. Let us see which sets of states can be obtained from this set by using the letters of the alphabet:

- *by letter a the set $\{q_0, q_1\}$ is obtained, however, its λ-closure is $\{q_0, q_1, q_2\}$;*
- *by letter b the set $\{q_2\}$ is obtained and its λ-closure is $\{q_2\}$.*

Let us write these two sets in the second and third row of the table. Now let us see what sets of states can be reached from these sets. First, let us see the set $\{q_0, q_1, q_2\}$.

- *by letter a the set $\{q_0, q_1\}$ is obtained, however, its λ-closure is $\{q_0, q_1, q_2\}$;*
- *by letter b the set $\{q_1, q_2\}$ is obtained and its λ-closure is $\{q_1, q_2\}$.*

Since this latter set is not in the table yet, it is added to the fourth row. Now let us see the set $\{q_2\}$.

- *by letter a the set $\{q_0\}$ is obtained, and its λ-closure is $\{q_0\}$;*
- *by letter b the set $\{q_1\}$ is obtained and its λ-closure is $\{q_1, q_2\}$.*

Since both of these two sets are already in the table we do not need to add a new row. Finally, let us analyse the set $\{q_1,q_2\}$ (that is the last row of the table).

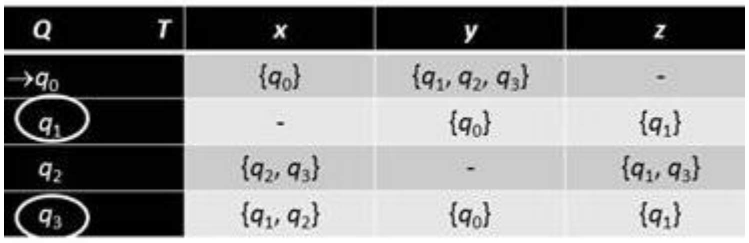
- *by letter a the set $\{q_0, q_1\}$ is obtained, and its λ-closure is $\{q_0, q_1, q_2\}$;*
- *by letter b the set $\{q_1\}$ is obtained and its λ-closure is $\{q_1, q_2\}$.*

These sets are in the table. So the table is filled. The initial state of the new deterministic automaton is $\{q_0\}$. The final states are: $\{q_0, q_1, q_2\}$, $\{q_2\}$, and $\{q_1, q_2\}$. The next table shows the resulting deterministic finite automaton:

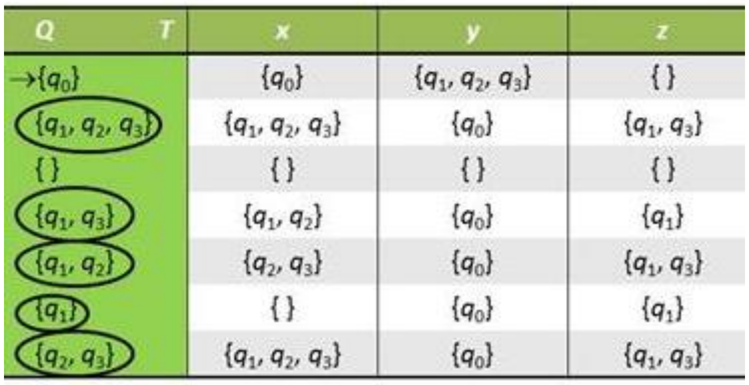
Q T	***a***	***b***
$\rightarrow \{q_0\}$	$\{q_0, q_1, q_2\}$	$\{q_2\}$
$\subset\{q_0, q_1, q_2\}\supset$	$\{q_0, q_1, q_2\}$	$\{q_1, q_2\}$
$\subset\{q_2\}\supset$	$\{q_0\}$	$\{q_1, q_2\}$
$\subset\{q_1, q_2\}\supset$	$\{q_0, q_1, q_2\}$	$\{q_1, q_2\}$

Example 24. *Animation 5. [23] shows an example how to obtain a completely defined deterministic automaton that is equivalent to the original nondeterministic automaton.*

Animation 5.



Q T	x	y	z
$\rightarrow q_0$	$\{q_0\}$	$\{q_1, q_2, q_3\}$	-
q_1	-	$\{q_0\}$	$\{q_1\}$
q_2	$\{q_2, q_3\}$	-	$\{q_1, q_3\}$
q_3	$\{q_1, q_2\}$	$\{q_0\}$	$\{q_1\}$



Q T	x	y	z
$\rightarrow\{q_0\}$	$\{q_0\}$	$\{q_1, q_2, q_3\}$	{ }
$\{q_1, q_2, q_3\}$	$\{q_1, q_2, q_3\}$	$\{q_0\}$	$\{q_1, q_3\}$
{ }	{ }	{ }	{ }
$\{q_1, q_3\}$	$\{q_1, q_2\}$	$\{q_0\}$	$\{q_1\}$
$\{q_1, q_2\}$	$\{q_2, q_3\}$	$\{q_0\}$	$\{q_1, q_3\}$
$\{q_1\}$	{ }	$\{q_0\}$	$\{q_1\}$
$\{q_2, q_3\}$	$\{q_1, q_2, q_3\}$	$\{q_0\}$	$\{q_1, q_3\}$

Let $A = (Q, T, q_0, \delta, F)$ be a deterministic finite automaton such that each of its states is reachable from its initial state (there are no useless states). Then we can construct the minimal deterministic finite automaton that is equivalent to A in the following way:

Let us divide the set of states into two groups obtaining the classification $C_1 = \{F, Q \backslash F\}$. (We denote the class where state q is by $C_1[q]$.)

Then, for $i > 1$ the classification C_i is obtained from C_{i-1}: the states p and q are in the same class by C_i if and only if they are in the same class by C_{i-1} and for every $a \in T$ they behave similarly: $\delta(p,a)$ and $\delta(q,a)$ are in the same class by C_i.

Set Q is finite and, therefore, there is a classification C_m such that it is the same as C_{m+1}.

Then, we can define the minimal completely defined deterministic automaton that is equivalent to A: its states are the groups of the classification C_m, the initial state is the group containing the initial state of the original automaton, the final states are those groups that are formed from final states of the original automaton, formally:

$$(C_m, T, C_m[q_0], \delta_{C_m}, F_{C_m}),$$

where $\delta_{C_m}(C_m[q], a) = C_m[\delta(q,a)]$ for every $C_m[q] \in C_m$, $a \in T$ and $F_{C_m} = \{C_m[q] | \ q \in F\}$.

It may happen that there are some words $w \in T^*$ that are not prefixes of any words of a regular language L. Then, the minimal completely defined deterministic automaton contains a sink state, that is the state where the word w and other words with the same property lead the automaton. When we want to have a minimal deterministic finite automaton for these languages, allowing partial (not completely defined) finite automata, then we may delete this sink state (with the transitions into it) by decreasing the number of the states by one.

Let us see yet another example. When applying the minimization algorithm it is more convenient to put the states to the 0th row of the table and the letters of the alphabet to the 0th column of the table.

Example 25. *Let the deterministic automaton A be given as follows:*

T Q	**→ q_0**	**q_1**	**⊂q_2⊃**	**q_3**	**⊂q_4⊃**	**⊂q_5⊃**	**⊂q_6⊃**
a	q_2	q_5	q_1	q_1	q_2	q_1	q_0
b	q_1	q_0	q_3	q_4	q_5	q_3	q_2

Give a minimal deterministic automaton that is equivalent to A.

Solution:

Before applying the algorithm we must check which states can be reached from the initial state: from q_0 one can reach the states q_0, q_2, q_1, q_3, q_5, q_4. Observe that the automaton cannot enter state q_6, therefore, this state (column) is deleted. The task is to minimize the automaton

T Q	**→ q_0**	**q_1**	**⊂q_2⊃**	**q_3**	**⊂q_4⊃**	**⊂q_5⊃**
a	q_2	q_5	q_1	q_1	q_2	q_1
b	q_1	q_0	q_3	q_4	q_5	q_3

by the algorithm. When we perform the first classification of the states $C_1 = \{Q_1, Q_2\}$ by separating the accepting and non-accepting states: $Q_1 = \{q_2, q_4, q_5\}$, $Q_2 = \{q_0, q_1, q_3\}$ then we have:

	Q	***Q_1***			***Q_2***		
T		**⊂q_2⊃**	**⊂q_4⊃**	**⊂q_5⊃**	**→q_0**	**q_1**	**q_3**
a		Q_2	Q_1	Q_2	Q_1	Q_1	Q_2
b		Q_2	Q_1	Q_2	Q_2	Q_2	Q_1

Then $C_2 = \{Q_{11}, Q_{12}, Q_{21}, Q_{22}\}$ with $Q_{11} = \{q_2, q_5\}$, $Q_{12} = \{q_4\}$, $Q_{21} = \{q_0, q_1\}$, $Q_{22} = \{q_3\}$. Then according to this classification we have

	Q	***Q_{11}***		***Q_{12}***	***Q_{21}***		***Q_{22}***
T		**⊂q_2⊃**	**⊂q_5⊃**	**⊂q_4⊃**	**→q_0**	**q_1**	**q_3**
a		Q_{21}	Q_{21}	Q_{11}	Q_{11}	Q_{11}	Q_{21}
b		Q_{22}	Q_{22}	Q_{11}	Q_{21}	Q_{21}	Q_{12}

Since $C_3 = C_2$ we have the solution, the minimal deterministic finite automaton equivalent to A:

T Q	**⊂Q_{11}⊃**	**⊂Q_{12}⊃**	**→Q_{21}**	**Q_{22}**
a	Q_{21}	Q_{11}	Q_{11}	Q_{21}
b	Q_{22}	Q_{11}	Q_{21}	Q_{12}

We conclude this subsection by a set of exercises.

Exercise 17. *Give a finite automaton that accepts the language of words that contain the consecutive substring baab over the alphabet {a,b}.*

Exercise 18. *Let the automaton A be given in a Cayley table as follows:*

T Q	**→q_0**	**q_1**	**q_2**	**q_3**	**⊂q_4⊃**
0	q_0	q_1	q_2	q_4	q_1
1	q_1	q_2	q_3	q_3	q_4

Draw the graph of A. What is the type of A (e.g., nondeterministic with allowed λ-transitions)?

Exercise 19. *The graph of the automaton A is given in Figure 2.3.*

2.3. ábra - The graph of the automaton of Exercise 19 [25].

Describe A by utilizing a Cayley table. What is the type of A (e.g., deterministic with partially defined transition function)? What is the language that A recognizes?

Exercise 20. *Let a nondeterministic automaton with λ-transitions, A, be defined by the following Cayley table:*

Q T	**a**	**b**	**c**	**λ**
→q_0	q_0, q_3	q_1	-	-
⊂q_1⊃	q_1	q_3	-	⊂q_2⊃
q_2	-	q_0	q_2	-
q_3	q_1	q_1	q_2	q_0

Give a completely defined deterministic automaton that is equivalent to A.

Exercise 21. *Let a nondeterministic automaton A be defined by the following table:*

Q T	**0**	**1**
→q_0	q_0, q_1	q_1
q_1	q_1	q_2, q_3
q_2	q_0	-
⊂q_3⊃	q_3	q_0, q_1, q_2

Give a completely defined deterministic automaton that accepts the same language as A.

Exercise 22. *Let a nondeterministic automaton A be defined by the graph shown in Figure 2.4.*

2.4. ábra - The graph of the automaton of Exercise 22. [26]

Give a completely defined deterministic automaton that accepts the same language as A.

Exercise 23. *Let the deterministic automaton A be given as follows:*

T Q	**→q_0**	**q_1**	**⊂q_2⊃**	**q_3**	**⊂q_4⊃**	**⊂q_5⊃**	**q_6**	**⊂q_7⊃**
a	q_6	q_3	q_1	q_7	q_1	q_0	q_4	q_1
b	q_4	q_7	q_1	q_2	q_3	q_1	q_5	q_6
c	q_5	q_2	q_6	q_1	q_3	q_6	q_0	q_6

Give a minimal deterministic automaton that is equivalent to A.

Exercise 24. *Let the deterministic automaton A be given by the following table.*

T Q	**→q_0**	**q_1**	**⊂q_2⊃**	**q_3**	**q_4**	**q_5**	**q_6**	**⊂q_7⊃**	**q_8**
0	q_3	q_5	q_7	q_0	q_2	q_0	q_4	q_2	q_5
1	q_8	q_1	q_3	q_2	q_6	q_7	q_3	q_5	q_2

Give a minimal deterministic automaton that is equivalent to A. (Hint: check first which states can be reached from the initial state.)

Exercise 25. *Let a nondeterministic automaton A be defined by the graph shown in Figure 2.5.*

2.5. ábra - The graph of the automaton of Exercise 25. [26]

Give a minimal deterministic automaton that is equivalent to A.

3.1. 2.3.1. Synthesis and Analysis of Finite Automata

Now we are going to show that exactly the class of regular languages can be described with the help of the finite automata. First we show that every regular language (type 3 language) can be accepted by finite automata.

Theorem 5. *Every language generated by a regular grammar is accepted by a finite automaton.*

Proof. The proof is constructive. Let $G = (N, T, S, P)$ be a regular grammar in normal form. Then, let the finite automaton

$$A = (Q, T, q_0, \delta, F)$$

be defined as follows:

let $Q = N \cup \{F'\}$ (where $F' \notin N$),

$q_0 = S$.

Let the transition function δ be defined by the elements of P: let $B \in \delta(A,a)$ if $A \rightarrow aB \in P$; and let $F' \in \delta(A,a)$ if $A \rightarrow a \in P$. Further, let the set of accepting states be $\{F'\}$ if $S \rightarrow \lambda$ is not in P and let $F = \{F',S\}$ if $S \rightarrow \lambda \in P$.

One can see that the successful derivations in the grammar and the accepting runs of the automaton have a one-to-one correspondence.

QED.

Example 26. Let

$$G = (\{S, A, B, C\}, \{a, b, c\}, S,$$

```
{   S → abbA,
    S → baaB,
    S → λ,
    A → aS,
    A → aC,
    B → bC,
    C → aS,
    C → cc
})
```

generating the language L(G). Give a finite automaton that accepts the language L(G).

Solution:

First we must exclude the rules containing more than 1 terminals, as we did in the proof of Theorem 2. [13] In this way the grammar

$$G' =(\{S, A, B, C, X_1, X_2, X_3, X_4, X_5\}, \{a, b, c\}, S,$$

```
{   S → aX_1,
    X_1 → bX_2,
    X_2 → bA,
    S → bX_3,
    X_3 → aX_4,
    X_4 → aB,
    S → λ,
    A → aS,
    A → aC,
    B → bC,
    C → aS,
    C → cX5,
    X5 → c
})
```

can be obtained. Then, we can draw the graph of automaton A as it can be seen in Figure 2.6.

2.6. ábra - The graph of the automaton of Example 26. [27]

Now we are going to show that the class of finite automata cannot accept more languages than the class that can be described by regular expressions.

Theorem 6. *Every language accepted by a finite automaton can be described by a regular expression.*

Proof. The proof is constructive. We present an algorithm that shows how one can construct a regular expression from a finite automaton. We can restrict ourselves to deterministic finite automaton, since we have already seen that they are equivalent to the nondeterministic finite automata. Let the states of a deterministic finite automaton be 1,2,... for the sake of simplicity, i.e., let $A = (\{1,...\, n\}, T, 1, \delta, F)$ be given. Let $r_{i,j}^{k}$ denote the regular expression that describes the language accepted by the automaton $(\{1,...,k\} \cup \{i,j\}, T, i, \delta', \{j\})$, where δ' is the restriction of δ containing only transitions from the set $\{i\} \cup \{1,...,k\}$ to the set $\{1,...,k\} \cup \{j\}$ $(1 \le i,j \le n, 0 \le k \le n$ and 1,..., 0 means the empty set).

Then, $r_{i,i}^{0}$ describes the regular language that is given by direct transition(s) from state i to j. Therefore, $r_{i,i}^{0}$ gives the language $\{\lambda\} \cup \{a|\ \ \delta\ (i,a) = i\}$ (this language contains the empty word and possibly some one-letter-long words, i.e., it is described by a basic regular expression or finite union of basic regular expressions).

Further, $r_{i,j}^{0}$ describes the language $\{a|\ \ \delta\ (i,a) = j\}$ if $i \neq j$ (it can contain some one-letter-long words). These regular expressions can easily be obtained and proven to describe the languages mentioned.

Now we use induction on the upper index:

$$r_{i,j}^{k} = r_{i,k}^{k-1}\left(r_{k,k}^{k-1}\right)^{*} r_{k,j}^{k-1} + r_{i,j}^{k-1}$$

for $1 \le k \le n$. This expression can be seen intuitively: starting from state i we could reach state j by using the first k states in our path, either without using state k (the part $r_{i,j}^{k-1}$ refers to this case) or by a path reaching state $k\left(r_{i,k}^{k-1}\right)$, and then reaching it arbitrarily many times (including 0 times: $\left(r_{k,k}^{k-1}\right)^{*}$, and finally reaching state j from state $k\left(r_{k,j}^{k-1}\right)$.

Finally, $\bigcup_{j\in F} r_{1,j}^{k}$ gives a regular expression that describes exactly the language accepted by A. In this way it is constructively proven that for any finite automaton one can construct a regular expression that describes the language accepted by the automaton.

QED.

Example 27. *Let the Cayley table of automaton A be given as follows:*

T Q	**→*q*₁**	**⊂*q*₂⊃**	**⊂*q*₃⊃**
a	q_2	q_3	q_1
b	q_3	q_2	-

Describe the accepted language L(A) by a regular expression.

Solution:

Let us describe the regular expressions $r^0_{i,j}$ *for i,j ∈* {1,2,3} *by using the transitions of A.*

$r^0_{1,1} = \lambda,$	$r^0_{1,2} = a,$	$r^0_{1,3} = b,$
$r^0_{2,1} = \varnothing,$	$r^0_{2,2} = \lambda + b,$	$r^0_{2,3} = a,$
$r^0_{3,1} = a,$	$r^0_{3,2} = \varnothing,$	$r^0_{3,3} = \lambda.$

Now by using the inductive step we compute $r^1_{i,j}$ *for i,j ∈* {1,2,3}.

- $r^1_{1,1} = r^0_{1,1}\left(r^0_{1,1}\right)^* r^0_{1,1} + r^0_{1,1} = \lambda,$
- $r^1_{1,2} = r^0_{1,1}\left(r^0_{1,1}\right)^* r^0_{1,2} + r^0_{1,2} = a,$
- $r^1_{1,3} = r^0_{1,1}\left(r^0_{1,1}\right)^* r^0_{1,3} + r^0_{1,3} = b,$
- $r^1_{2,1} = r^0_{2,1}\left(r^0_{1,1}\right)^* r^0_{1,1} + r^0_{2,1} = \varnothing,$
- $r^1_{2,2} = r^0_{2,1}\left(r^0_{1,1}\right)^* r^0_{1,2} + r^0_{2,2} = \lambda + b,$
- $r^1_{2,3} = r^0_{2,1}\left(r^0_{1,1}\right)^* r^0_{1,3} + r^0_{2,3} = a,$
- $r^1_{3,1} = r^0_{3,1}\left(r^0_{1,1}\right)^* r^0_{1,1} + r^0_{3,1} = a,$
- $r^1_{3,2} = r^0_{3,1}\left(r^0_{1,1}\right)^* r^0_{1,2} + r^0_{3,2} = aa,$
- $r^1_{3,3} = r^0_{3,1}\left(r^0_{1,1}\right)^* r^0_{1,3} + r^0_{3,3} = ab + \lambda.$

Now we can continue by computing the expressions $r^2_{i,j}$:

- $r^2_{1,1} = r^1_{1,2}\left(r^1_{2,2}\right)^* r^1_{2,1} + r^1_{1,1} = \lambda,$
- $r^2_{1,2} = r^1_{1,2}\left(r^1_{2,2}\right)^* r^1_{2,2} + r^1_{1,2} = ab^*,$
- $r^2_{1,3} = r^1_{1,2}\left(r^1_{2,2}\right)^* r^1_{2,3} + r^1_{1,3} = ab^*a + b,$
- $r^2_{2,1} = r^1_{2,2}\left(r^1_{2,2}\right)^* r^1_{2,1} + r^1_{2,1} = \varnothing,$
- $r^2_{2,2} = r^1_{2,2}\left(r^1_{2,2}\right)^* r^1_{2,2} + r^1_{2,2} = b^*,$
- $r^2_{2,3} = r^1_{2,2}\left(r^1_{2,2}\right)^* r^1_{2,3} + r^1_{2,3} = b^*a,$

- $r^2_{3,1} = r^1_{3,2}\left(r^1_{2,2}\right)^* r^1_{2,1} + r^1_{3,1} = a,$
- $r^2_{3,2} = r^1_{3,2}\left(r^1_{2,2}\right)^* r^1_{2,2} + r^1_{3,2} = aab^*,$
- $r^2_{3,3} = r^1_{3,2}\left(r^1_{2,2}\right)^* r^1_{2,3} + r^1_{3,3} = aab^*a + ab + \lambda.$

To describe L(*A*) *we need* $r^3_{1,2} + r^3_{1,3}$. *Let us compute it:*

- $r^3_{1,2} + r^2_{1,3}\left(r^2_{3,3}\right)^* r^2_{3,2} + r^2_{1,2} = \left(ab^*a + b\right)\left(aab^*a + ab\right)^* aab^* + ab^*,$
- $r^3_{1,3} + r^2_{1,3}\left(r^2_{3,3}\right)^* r^2_{3,3} + r^2_{1,3} = \left(ab^*a + b\right)\left(aab^*a + ab\right)^* + ab^*a + b = \left(ab^*a + b\right)\left(aab^*a + ab\right).$

Thus, the regular expression

> $(ab^*a + b)(aab^*a + ab)^*aab^* + ab^* + (ab^*a + b)(aab^*a + ab)^* = (ab^*a + b)(aab^*a + ab)^* (\lambda + aab^*) + aab^*$

describes L(*A*). *The problem is solved.*

Now we have proven that the three concepts we have discussed in this chapter, the regular grammars, regular expressions and finite automata are equivalent in the sense that each of them characterize exactly the class of regular languages (see Figure 2.7).

2.7. ábra - The equivalence of the three types of descriptions (type-3 grammars, regular expressions and finite automata) of the regular languages.

The aim of the analysis of a finite automaton is the task to describe the accepted language in another way, usually, by regular expressions. The synthesis of a finite automaton is the construction of the automaton that accepts a regular language that is usually given by a regular expression. Kleene has proven the equivalence of finite automata and regular expressions.

The minimization algorithm is very important, since the minimal, completely defined, deterministic finite automaton is (the only known) unique identification of a regular language. In this way, we can decide if two regular languages (given by regular expressions, grammars or automata) coincide or not.

We close this subsection by a set of exercises.

Exercise 26. *Let*

> $G = (\{S, A, B\}, \{0,1\}, S,$

```
{   S → 000A,
    S → 111B,
    A → λ,
    A → 0S,
    A → 11,
    B → 1S,
    B → 000
})
```

generating language L(*G*). *Give a finite automaton that accepts language L*(*G*). *(Hint: first transform the grammar to normal form.)*

Exercise 27. *Let*

$G = (\{S, A, B, C\}, \{a,b\}, S,$

```
{   S → aaaA,
    S → bbB,
    S → C,
    A → S,
    A → baB,
    A → ba,
    B → S,
    B → C,
    B → b,
    B → λ,
    C → B,
    C → aA
})
```

generating language $L(G)$. Give a finite automaton that accepts language $L(G)$.

Exercise 28. *Let*

$G = (\{S, A, B\}, \{a, b, c\}, S,$

```
{   S → abA,
    S → bccS,
    A → bS,
    A → c,
    A → B,
    B → S,
    B → aA,
    B → bcc,
    B → λ
})
```

generating language $L(G)$. Give a finite automaton that accepts language $L(G)$.

Exercise 29. *Let the automaton A be defined by the following table:*

T Q	$\rightarrow q_1$	q_2	$\subset q_3 \supset$
0	q_1	q_3	q_3
1	q_2	q_2	q_1

Give a regular expression that describes the language accepted by automaton A.

Exercise 30. *Let automaton A, accepting language $L(A)$, be defined by the Cayley table:*

T Q	$\rightarrow \subset q_1 \supset$	q_2	$\subset q_3 \supset$
a	q_1	q_3	-
b	-	q_2	q_1
c	q_2	q_2	-

Give a regular expression that describes language $L(A)$.

Exercise 31. *Let automaton A be as it is shown in Figure 2.8. Give a regular expression that defines the same language as A.*

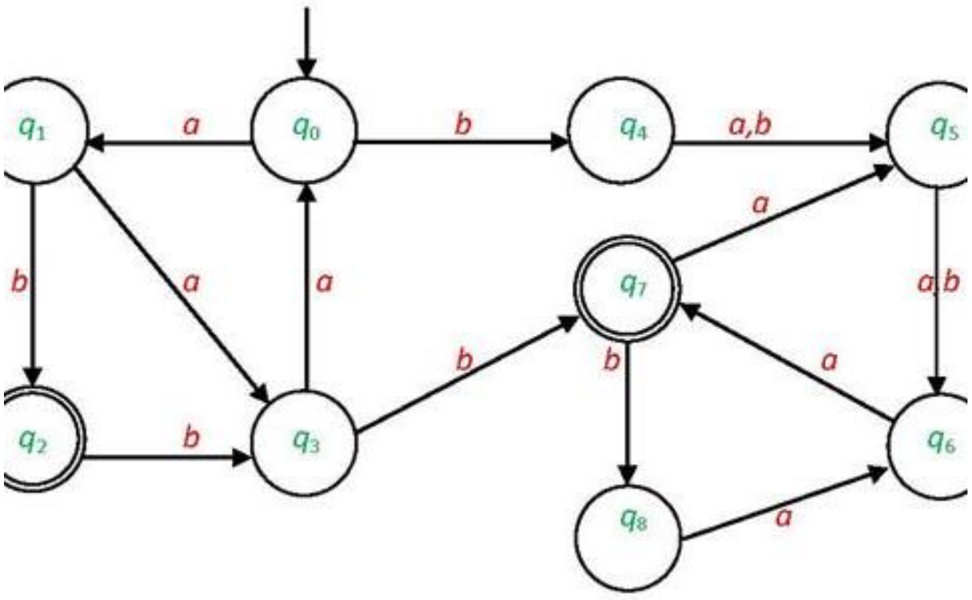
2.8. ábra - The graph of the automaton of Exercise 31. [31]

3.2. 2.3.2. The Word Problem

The word problem is to decide whether any given word *w* belongs to a given regular language (or not). This can be done very efficiently by using a deterministic finite automaton that accepts the given language. (Such an automaton can be constructed from a regular expression, first by Theorem 3. [18] and then/or from a grammar by Theorem 5. [27] and then/or from a nondeterministic finite automaton by Theorem 4. [22]) Reading the word *w* can be done by $|w|$ steps, and then, if the automaton accepts *w*, i.e., it arrived at an accepting state in this run, then *w* is an element of the language. Otherwise, (if there were some undefined transitions and the automaton gets stuck, or by reading the word finally a non accepting state is reached) *w* does not belong to the given regular language. The decision on an input word of length *n* is done in at most *n* steps, therefore, this is a real time algorithm (i.e., linear time algorithm with coefficient 1). There is no faster algorithm that can read the input, so the word problem for regular languages can be solved very efficiently.

Exercise 32. *Let automaton A be given as it can be seen in Figure 2.9.*

2.9. ábra - The graph of the automaton of Exercise 32. [32]



```
Decide whether the words

   abab,
   baba,
   aaaabbb,
   bbbaaaab,
   baabaabaabb and
   aaaabbbababaaa are in L(A).
```

Exercise 33. *Let the nondeterministic automaton A be defined by the Cayley table:*

Q T	**0**	**1**	**λ**
$\rightarrow q_0$	q_1, q_4	q_5	q_2
q_1	q_3	q_1, q_4	-
q_2	q_1	q_2	-
$\subset q_3 \supset$	-	-	q_2
q_4	-	q_4	-

Q T	**0**	**1**	**λ**
$\subset q_5 \supset$	q_5	q_5	q_0

Decide which of the words 0011, 0101, 0110, 01110010, 100, 1011110 belong to the accepted language of A.

Exercise 34. *Let*

$G = (\{S, A, B\}, \{a, b, c\}, S,$

```
{   S → abcS,
    S → bcA,
    S → acB,
    A → aS,
    A → a,
    B → bS,
    B → bcc,
    S → ccc
})
```

be a regular grammar. Decide which of the following words can be generated by G:

```
    abcccc,
    acbcc,
    acbbacbccc,
    bca,
    bcacacc,
    bcaabcacbcc,
    ccc,
    cccacbc.
```

Exercise 35. *Given the regular language*

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

decide if the following words are in the described language or not:

```
    aaaaa,
    aaabaa,
    ababa,
    abbabaaba.
```

4. 2.4. Properties of Regular Languages

In the next part of this section we concentrate on the closure properties of the class of regular languages.

4.1. 2.4.1. Closure Properties

By the constructive proof of Theorem 3 [18], it is also shown that the class of regular languages is closed under the regular operations. Now let us consider the set theoretical operations: intersection and complement.

Theorem 7. *The class of regular languages is closed under intersection and complement.*

Proof. The proof is constructive in both cases, and deterministic finite automata are used. Let us start with the complement. Let a regular language be given by a complete deterministic finite automaton $A = (Q, T, q_0, \delta, F)$ that recognizes it. This automaton has exactly one run for every word of T^*, and accepts a word if this run is finished in an accepting state. Then $\bar{A}=(Q,T,q_0,\delta,Q\setminus F)$ recognize exactly those words that are not accepted by A, and thus the finite automaton $\bar{A}$ accepts the complement of the original regular language.

For the intersection, let two regular languages L_1 and L_2 be given by complete deterministic automata $A = (Q, T, q_0, \delta,F)$ and $A' = (Q', T, q'_0, \delta', F')$ that recognize them, respectively. Then, let $A^\cap = (Q \times Q', T, (q_0, q'_0), \delta'', F\times F')$, with transition function $\delta''((q,q'), a) = (\delta(q,a), \delta'(q',a))$ for every $q \in Q$, $q' \in Q'$ and $a \in T$. The states are formed by pairs of the states of the automata A and A'. Thus, $A^\cap$ simulates the work of these two automata simultaneously and accepts exactly those words that are accepted by both of these machines. Thus the intersection of the languages L_1 and L_2 is accepted by a finite automaton, and thus it is also a regular language.

QED.

Example 28. *Let the automaton A and A' accept the languages L(A) and L(A'), respectively. Let them be defined in the following way: the table of A as shown below.*

T Q	**$\rightarrow q_1$**	**q_2**	**$\subset q_3 \supset$**
a	q_3	q_2	q_3
b	q_2	q_2	q_3

the table of A' as shown below

T Q'	**$\rightarrow q'_1$**	**$\subset q'_2 \supset$**
a	q'_1	q'_1
b	q'_2	q'_2

Give an automaton that accepts the complement of L(A) and an automaton that accepts the intersection of L(A) and L(A'). What are the languages accepted by these automata?

Solution:

Let us take the automaton that accepts the complement of L(A) by interchanging the role of accepting and non-accepting states in A. Let $\bar{A}$ be defined by the following Cayley table:

T Q	**$\rightarrow\subset q_1 \supset$**	**$\subset q_2 \supset$**	**q_3**
a	q_3	q_2	q_3
b	q_2	q_2	q_3

Now let us construct $A^\cap$ by using the Cartesian product of the sets of states Q and Q'.

T Q	**$\rightarrow(q_1, q'_1)$**	**(q_2, q'_1)**	**(q_3, q'_1)**	**(q_1, q'_2)**	**(q_2, q'_2)**	**$\subset(q_3, q'_2)\supset$**
a	(q_3, q'_1)	(q_2, q'_1)	(q_3, q'_1)	(q_3, q'_1)	(q_2, q'_1)	(q_3, q'_1)
b	(q_2, q'_2)	(q_2, q'_2)	(q_3, q'_2)	(q_2, q'_2)	(q_2, q'_2)	(q_3, q'_2)

Actually, A accepts the language $a(a+b)^*$(words starting by letter a), and A' accepts the language $(a+b)^*b$ (words that ends with letter b), the automaton $\bar{A}$ accepts the language $\lambda + b(a+b)^*$ (words do not start with letter a over the alphabet {a,b}), while $A^\cap$ accepts the language $a(a+b)^*b$ (words starting with a and ending with b).

Exercise 36. *Let the table of A be*

T Q	**$\rightarrow q_1$**	**q_2**	**$\subset q_3 \supset$**
0	q_1	q_3	q_2
1	q_2	q_2	q_2

and the table of A' be

T Q'	$\rightarrow q'_1$	$\subset q'_2 \supset$	$\subset q'_3 \supset$
0	q'_3	q'_2	q'_3
1	q'_2	q'_1	q'_2

Give an automaton that accepts the intersection of the languages accepted by A and A'.

Exercise 37. *Let the language $L(A)$ be defined as the accepted language of the automaton A as it is shown in Figure 2.10.*

2.10. ábra - The graph of the automaton of Exercise 37.

Give an automaton that accepts the complement of $L(A)$. (Hint: first the equivalent completely defined deterministic automaton must be obtained.)

Exercise 38. *Let the table of A be*

T Q'	$\rightarrow q_1$	q_2	$\subset q_3 \supset$	$\subset q_4 \supset$
a	q_1	q_3	q_1	q_1
b	q_1	q_4	q_2	q_2
c	q_2	q_4	q_4	q_3

and the table of A' be

T Q'	$\rightarrow \subset q'_1 \supset$	q'_2
a	q'_1	q'_2
b	q'_2	q'_1
c	q'_2	q'_1

They accept the languages $L(A)$ and $L(A')$, respectively.

Give automata that accept the complement of $L(A)$ and $L(A')$. Give an automaton that accepts $L(A) \cap L(A')$.

4.2. 2.4.2. Myhill-Nerode theorem

In the next part we show an if and only if characterization of the class of regular languages.

Let us define congruence relations on T^* with the following property: for every $u, v, w \in T^*$ if $u \sim v$, then $uw \sim vw$. These relations are called *right-congruences*. A congruence relation is of finite index, if the number of classes of T^* is finite.

Theorem 8. (Myhill-Nerode theorem). *A language over the alphabet T is regular if and only if there is a finite index right-congruence relation on T^* such that the language is obtained as (the union of) some of the classes induced by the relation.*

Let a language L be given. Roughly speaking two words, u and v are equivalent if their every possible continuation w behaves in the same manner, i.e., $uw \in L$ if and only if $vw \in L$. If this equivalence relation induces finitely many classes on T^*, then, and only then, L is regular.

Actually, this fact is also related to the minimal, completely defined, accepting deterministic finite automaton (that uniquely identifies the given regular language): the partitions of T^* can be assigned to the states of the minimal automaton: the partition containing the empty word λ is assigned to the initial state; those partitions that contain words such that their empty continuation is in L are assigned to the accepting states. The transitions can also be easily defined by using the partitions, by checking in which partition the words are that are given by the previous one by extending it with exactly one letter.

Now, we are going to give an example for how this theorem can be used to show that some of the languages are not regular.

Example 29. *Let us analyze language $L = \{a^n b^n | n \in \mathbb{N}\}$. Let us partition the words of a^* into classes: assume that a^k and a^m are in the same class, i.e., $a^k \sim a^m$. Then, for each possible continuation ($w \in \{a,b\}^*$) they behave in the same manner, e.g., for b^k the word $a^k b^k \in L$ and thus, $a^m b^k \in L$ also. But it can only be if $m = k$, and so every element of a^* is equivalent to only itself and not to any other element of this set. Therefore, language L induces an infinite index right-congruence relation, thus L is not regular.*

Exercise 39. *Show that the language containing exactly the words having the same number of 0's and 1's (over the alphabet {0,1}) is not regular.*

(Apply the Myhill-Nerode theorem.)

Exercise 40. *Show that the language $L = \{a^{n^2} | n \in \mathbb{N}\}$ (over the alphabet $\{a\}$), i.e., the unary words having square number length, is not regular.*

(Apply the Myhill-Nerode theorem.)

5. 2.5. Finite Transducers

Transducers are machines which do not only have input, but output as well. One can imagine them as automata with two tapes: an input and an output tape. In this book, we consider only the simplest transducers: they are finite and they give an output letter as a response to every input letter. In this section, we briefly describe two types of finite state transducers.

5.1. 2.5.1. Mealy Automata

We start by giving the formal definition of Mealy automata.

Definition 17. *A Mealy automaton is an ordered sextuple $A = (Q, T, V, q_0, \delta, \mu)$, where Q, T, q_0, δ are the same as at the completely defined deterministic finite state acceptors, i.e., Q is the finite set of states, T is the input alphabet, $q_0 \in Q$ is the initial state, $\delta : Q \times T \to Q$ is the transition function; and V is the output alphabet and $\mu : Q \times T \to V$ is the output function.*

Notice that there are no final (or accepting) states. These automata are not used to accept languages.

A Mealy automaton can be defined by a Cayley table or by a graph. When a Cayley table is used to describe a Mealy automaton, then both the values $\delta(q,a)$ and $\mu(q,a)$, as pairs are written to the cell identified by the state q and by the letter a. When a graph is used to describe a Mealy automaton, then we can put a/x to an arrow meaning that the transition represented by the arrow is performed by reading an input letter a, while an output letter $x \in V$ is written to the output tape. Here is an example.

Example 30. *Let the Mealy automaton A be given by the following Cayley table*

T Q	$\to q_1$	q_2	q_3	q_4
a	(q_4, x)	(q_3, y)	(q_1, y)	(q_2, x)
a	(q_1, y)	(q_1, x)	(q_2, x)	(q_2, y)
a	(q_1, x)	(q_1, x)	(q_4, y)	(q_3, y)

The same automaton given by graph can be seen in Figure 2.11.

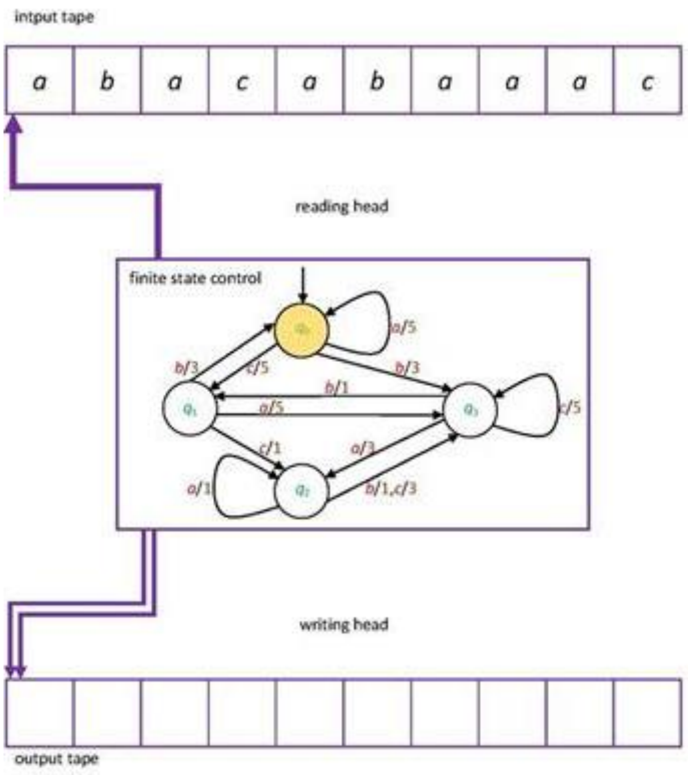
2.11. ábra - The graph of the Mealy automaton of Example 30. [36]

Let us make some sample runs of the automaton:

- *Let the input be abcabc, then the output is xyxxyx.*
- *Let the input be aaaabbbb, then the output is xxyyyyyy.*
- *Let the input be ccabcabc, then the output is xxxyxxyx.*
- *Let the input be abcbbccabac, then the output is xyxyyxxxyyy.*

Example 31. *Animation 6. [37] shows an example of a Mealy automaton, how it produces output for a given input.*

Animation 6.



We say that two Mealy automata are equivalent if they assign the same output word for every input word $u \in T^*$. The number of states of equivalent automata can be various. However, there is particular Mealy automaton for each equivalent class that has a minimal number of states. This automaton can be obtained from any automaton of the class by the minimization algorithm.

Now we present the minimization algorithm for Mealy automata. First, as with the finite state recognizers, we should check which states can be reached from the initial state. The states that cannot be reached can simply be erased from the automaton (table or graph) together with the transitions from them.

When we have a Mealy automaton, such that each of its states is reachable from the initial state with some input words (i.e., reading the given input word the automaton arrives to this particular state), then we can start an analogous algorithm that was used for minimizing finite state recognizers.

Only the initial step of the algorithm differs from the one shown previously: since we have no accepting states, the initial classification is done in another way. Let two states p and q be in the same class, i.e., $C_1[p] = C_1[q]$ if and only if for every input letter $a \in T$ the equality $\mu(p,a) = \mu(q,a)$ is fulfilled.

The other steps of the algorithm are similar to the previously described algorithm: based on the previous classification the next classification is obtained by separating those states for which there is an input letter such that the transition function with this letter brings them to different classes.

Let us see an example.

Example 32. *Let the Mealy automaton A be given as follows:*

T Q	$\to q_1$	q_2	q_3	q_4	q_5	q_6
a	(q_2, x)	(q_1, z)	(q_6, x)	(q_5, z)	(q_4, x)	(q_5, x)
b	(q_3, y)	(q_2, y)	(q_4, y)	(q_4, y)	(q_3, y)	(q_2, y)

Give the minimal Mealy automaton that is equivalent to A.

Solution:

One can easily check that every state can be reached from the initial state. Then classification $C_1 = \{Q_1, Q_2\}$, where $Q_1 = \{q_1, q_3, q_5, q_6\}$ (having output x for input a and output y for input b) and $Q_2 = \{q_2, q_4\}$ (having output z for input a and output y for input b). Then, the transition function reflecting this classification is as follows:

	Q	Q_1				Q_2	
T		$\to q_1$	q_3	q_5	q_6	q_2	q_4
a		Q_2	Q_1	Q_2	Q_1	Q_1	Q_1
b		Q_1	Q_2	Q_1	Q_2	Q_2	Q_2

Then, Q_1 is divided into two subgroups in the classification $C_2 = \{Q_{11}, Q_{12}, Q_2\}$ with $Q_{11} = \{q_1, q_5\}$ and $Q_{12} = \{q_3, q_6\}$. ($Q_2 = \{q_2, q_4\}$ remains the same group.) The transition function reflecting these groups is as follows:

	Q	Q_{11}		Q_{12}		Q_2	
T		$\to q_1$	q_5	q_3	q_6	q_2	q_4
a		Q_2	Q_2	Q_{11}	Q_{12}	Q_{11}	Q_{11}
b		Q_{12}	Q_{12}	Q_2	Q_2	Q_2	Q_2

Only Q_{12} is divided (to its elements) and thus $C_3 = \{Q_{11}, Q_{121}, Q_{122}, Q_2\}$ with $Q_{121} = \{q_3\}$ and $Q_{122} = \{q_5\}$. Then the transitions become:

	Q	Q_{11}		Q_{121}	Q_{122}	Q_2	
T		$\to q_1$	q_5	q_3	q_6	q_2	q_4
a		Q_2	Q_2	Q_{11}	Q_{122}	Q_{11}	Q_{11}
b		Q_{121}	Q_{121}	Q_2	Q_2	Q_2	Q_2

Since $C_4 = C_3$, we can give the minimal Mealy automaton (writing also the values of the output function into the table):

T Q	$\to Q_{11}$	Q_{121}	Q_{122}	Q_2
a	(Q_2, x)	(Q_{11}, x)	(Q_{122}, x)	(Q_{11}, z)
b	(Q_{121}, y)	(Q_2, x)	(Q_2, y)	(Q_2, y)

Exercise 41. *Let the Mealy automaton A be given with its table as follows:*

T Q	$\to q_0$	q_1	q_2	q_3
0	(q_0, a)	(q_3, a)	(q_1, b)	(q_2, b)

T Q	$\to q_0$	q_1	q_2	q_3
1	(q_1, b)	(q_1, a)	(q_2, b)	(q_2, a)

Draw the graph of automaton A.

Exercise 42. *Let the Mealy automaton A be given by its graph as it is shown in Figure 2.12.*

2.12. ábra - The graph of the Mealy automaton of Exercise 42. [39]

Describe the same automaton with a Cayley table.

What is the output of this automaton for the input strings aaabb, abbba, bbbaabb and aabbbaabab?

Give a minimal Mealy automaton that is equivalent to A.

Exercise 43. *Give a minimal Mealy automaton that is equivalent to the following one:*

T Q	$\to q_1$	q_2	q_3	q_4	q_5	q_6	q_7	q_8
a	$(q_8, 0)$	$(q_3, 1)$	$(q_8, 0)$	$(q_1, 1)$	$(q_8, 0)$	$(q_4, 0)$	$(q_3, 1)$	$(q_5, 1)$
b	$(q_2, 1)$	$(q_1, 0)$	$(q_7, 0)$	$(q_2, 1)$	$(q_7, 1)$	$(q_2, 1)$	$(q_6, 0)$	$(q_3, 1)$

Exercise 44. *Give a minimal Mealy automaton that is equivalent to the one defined by the following table.*

T Q	$\to q_1$	q_2	q_3	q_4	q_5
a	(q_2, x)	(q_2, y)	(q_2, x)	(q_5, y)	(q_4, y)
b	(q_4, x)	(q_3, x)	(q_1, x)	(q_3, x)	(q_3, x)
c	(q_5, y)	(q_5, x)	(q_1, y)	(q_5, x)	(q_2, x)

5.2. 2.5.2. Moore Automata

In this subsection another type of finite transducers are investigated.

Definition 18. *A Moore automaton is an ordered sextuple A = (Q, T, V, q_0, δ, η), where Q, T, V, q_0, δ are the same as at the Mealy automata, and η : Q × V is the output function.*

Notice that the difference between the Mealy and the Moore automata is due to their output function. While with the Mealy automata the output is produced during the transition (depending on both the state that the automaton was in and on the read input letter), at the Moore automata the output letter is produced after the transition is finished and the output letter depends only on the state the automaton reached by the transition.

The Moore automata can also be defined by Cayley table and by graph. Since with the Moore automata the output depends only on the state the automaton has reached, the output letters are written to the states (above the states, when the states are in the 0th row of table) and inside the circles of the states as pairs containing the state and the output assigned to the state on the graphs. Here is an example:

Example 33. *Let the Moore automaton A be given by its graph as it is shown in Figure 2.13.*

2.13. ábra - The graph of the Mealy automaton of Exercise 33. [39]

Describe the same automaton using a Cayley table. Give the output for input strings

```

aabb,
baabaa and
abaababb.
```

Solution:

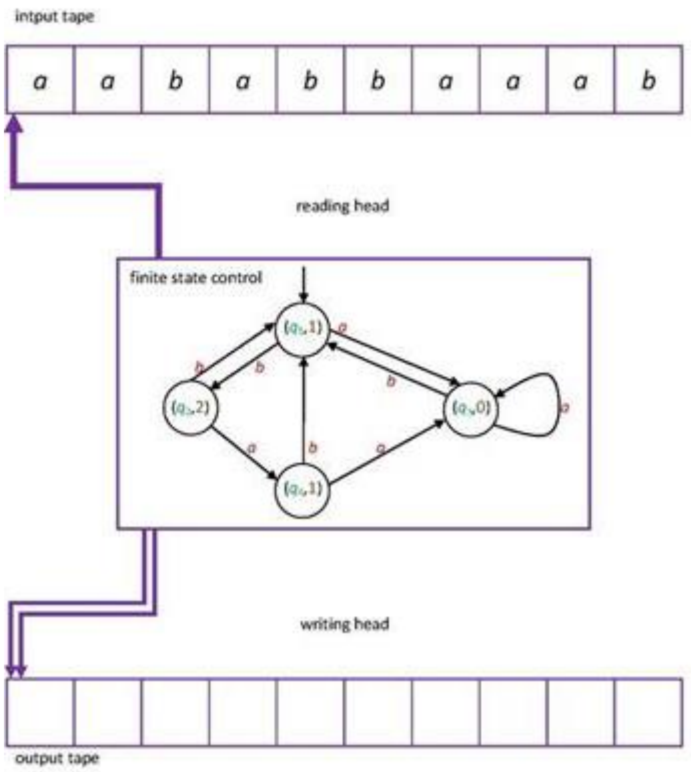
V	0	1	0	1
T Q	$\rightarrow q_0$	q_1	q_2	q_3
a	q_2	q_3	q_0	q_3
b	q_1	q_2	q_3	q_2

The example runs give the outputs as follows:

- *For input aabb the output is 0010.*
- *For input baabaa the output is 111000.*
- *For input abaababb the output is 01110010.*

Example 34. *Animation 7. [40] shows a Moore automaton at work.*

Animation 7.



Now we can generalize the equivalence relation between finite transducers: a Mealy/Moore automaton A is equivalent to a Mealy/Moore automaton A' if and only if for every input string $u \in T^*$ they produce the same output string.

The Moore automata can also be minimized and its algorithm is very similar to the previously described minimization algorithms. The only difference is that in this case the first classification is done based on the output letters assigned to the states, i.e., the states p and q are in the same class by classification C_1 if and only if $\eta(p) = \eta(q)$.

Let us see an example of how the algorithm works for the Moore automata.

Example 35. *The Moore automata A is defined by its Cayley table as follows:*

V	**x**	**y**	**y**	**y**	**x**	**x**	**x**
T Q	**$\rightarrow q_1$**	**q_2**	**q_3**	**q_4**	**q_5**	**q_6**	**q_7**
a	q_2	q_5	q_1	q_5	q_2	q_3	q_4
b	q_7	q_4	q_6	q_2	q_4	q_6	q_1

Find a minimal Moore automaton that is equivalent to A.

Solution:

First we check if every state is reachable from the initial state. It can be seen that states q_3 and q_6 are not reachable, therefore we erase them. We need to minimize the automaton

V	**x**	**y**	**y**	**x**	**x**
T Q	**$\rightarrow q_1$**	**q_2**	**q_4**	**q_5**	**q_7**
a	q_2	q_5	q_5	q_2	q_4
b	q_7	q_4	q_2	q_4	q_1

Classification $C_1 = \{Q_1, Q_2\}$ is based on the output function: $Q_1 = \{q_1, q_5, q_7\}$ (having output x) and $Q_2 = \{q_2, q_4\}$ (having output y). Then, the transitions using the classification become:

	Q	**Q_1**			**Q_2**	
T		**$\rightarrow q_1$**	**q_5**	**q_7**	**q_2**	**q_4**
a		Q_2	Q_2	Q_2	Q_1	Q_1
b		Q_1	Q_2	Q_1	Q_2	Q_2

Then, Q_1 is divided into two subclasses, therefore $C_2 = \{Q_{11}, Q_{12}, Q_2\}$ where $Q_{11} = \{q_1, q_7\}$ and $Q_{12} = \{q_5\}$. Then, we have:

	Q	**Q_{11}**		**Q_{12}**	**Q_2**	
T		**$\rightarrow q_1$**	**q_7**	**q_5**	**q_2**	**q_4**
a		Q_2	Q_2	Q_2	Q_{12}	Q_{12}
b		Q_{11}	Q_{11}	Q_2	Q_2	Q_2

Thus $C_3 = C_2$, and we can describe the minimal Moore automaton as follows:

V	***x***	***x***	***y***
T Q	**$\rightarrow Q_{11}$**	**Q_{12}**	**Q_2**
a	Q_2	Q_2	Q_{12}
b	Q_{11}	Q_2	Q_2

We note here that the minimization methods for finite automata presented in this book are using the Aufenkamp-Hohn algorithm.

Exercise 45. *A Moore automaton A is given by the following Cayley table:*

V	**x**	**x**	**y**	**z**	**z**	**y**
T Q	**$\rightarrow q_0$**	**q_1**	**q_2**	**q_3**	**q_4**	**q_5**
a	q_4	q_5	q_3	q_0	q_4	q_5

V	***x***	***x***	***y***	***z***	***z***	***y***
T Q	$\rightarrow q_0$	q_1	q_2	q_3	q_4	q_5
b	q_1	q_2	q_4	q_5	q_5	q_5

Give the same automaton by a graph. What is the output of the automaton for the input words

```
abbaab and
bbaabbaabbbb?
```

Exercise 46. *The Moore automaton A is defined by the graph shown in Figure 2.14.*

2.14. ábra - The graph of the Mealy automaton of Exercise 46. [42]

Give it by a Cayley table. Give its output for the input

```
cabbaccc,
aabbccabcabcbbcaca and
acaabacbbbcccaa.
```

Exercise 47. *The Moore automata A is defined by its Cayley table as follows:*

V	***0***	***1***	***0***	***2***	***0***	***2***	***0***	***1***	***1***
T Q	$\rightarrow q_1$	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_9
a	q_1	q_5	q_8	q_5	q_9	q_1	q_2	q_3	q_3
b	q_2	q_4	q_6	q_2	q_6	q_8	q_4	q_5	q_1
c	q_3	q_9	q_8	q_2	q_8	q_8	q_7	q_6	q_8

Find a minimal Moore automaton that is equivalent to A.

Exercise 48. *The Moore automata A is defined by its Cayley table as follows:*

V	***x***	***y***	***y***	***y***	***z***	***z***
T Q	$\rightarrow q_1$	q_2	q_3	q_4	q_5	q_6
a	q_2	q_4	q_4	q_2	q_2	q_3
b	q_3	q_2	q_6	q_4	q_6	q_5

Give the minimal Moore automaton that is equivalent to A.

5.3. 2.5.3. Automata Mappings

Finally, we devote this subsection to a brief analysis of the mappings $T^* \rightarrow V^*$ that can be obtained by the Mealy and the Moore automata.

We provide a theorem that the classes of the Mealy and the Moore automata have the same efficiency.

Theorem 9. (Gill's theorem). *For every Mealy automaton there exists an equivalent Moore automaton and for every Moore automaton there exists an equivalent Mealy automaton.*

For a Moore automaton there is a very simple way to construct a Mealy automaton that defines the same mapping from T^* to V^*. Roughly speaking, the output letter should move from a state to each of the transitions (arrows in the graph) into the given state. The other direction, that is not detailed here, can be done my multiplying the number of states (using the set $Q \times V$).

Now let us see some of the important properties of the mappings that can be defined by finite transducers.

Theorem 10. (Raney's theorem). *The automata mappings have the following two properties:*

- *They are length preserving, i.e., for any input string $w \in T^*$its length is the same as the length of the output $u \in V^*$ given by w.*
- *They are prefix keeping, i.e., the image of the prefix will also be prefix, more formally: for every $w,v \in T^*$the output for the string wv will start with the output string assigned to w.*

We note here that there are automata mappings that cannot be defined by the finite state Mealy or Moore automata, but only by their infinite state variants. We have chosen not to discuss these infinite variants in our book.

3. fejezet - Linear Languages

Summary of the chapter: *In this chapter, we deal with a family of languages between the regular and context-free languages of the Chomsky hierarchy, i.e., the linear languages. We give an example for a non-regular linear language. A normal form for linear grammars is proven. The class of one-turn pushdown automata recognizes exactly the class of linear languages. This class is closed under union, but it is not closed under concatenation, Kleene-star, complement and intersection.*

1. 3.1. Linear Grammars

First let us recall the definition of linear grammars.

Definition 19. (Linear grammars). *A grammar $G = (N, T, S, P)$ is linear if each of its productions has one of the following forms: $A \to u$, $A \to uBv$, where $A,B \in N$, $u,v \in T^*$. The languages that can be generated by linear grammars are the linear languages.*

This class of languages are between the (classes of) type 3 and type 2 languages of the Chomsky hierarchy, thus we may call them type 2.5 languages. The linear grammars inherit the property of the regular grammars that there is at most one nonterminal in any sentential form. However, this nonterminal is not restricted to be at the end of the sentential form (as it was in the regular case), it can be in an arbitrary place.

Now, we give an example, where the nonterminal is in the middle of the sentential forms in every derivation.

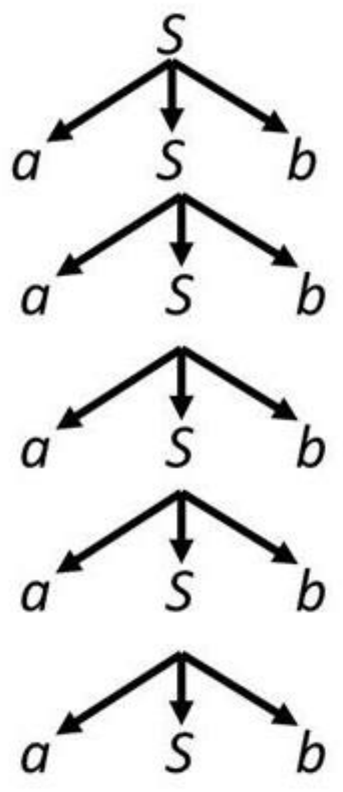
Example 36. *Let*

$$G = (\{S\}, \{a,b\}, S, \{S \to aSb, S \to \lambda\}).$$

Then, every (finished) derivation in this grammar has the following form: applying the first rule n times ($n \in \mathbb{N}$, $n \geq 0$) and then applying the second rule. Hence, the generated language is $\{a^nb^n | \ n \in \mathbb{N}\}$. See also Animation 8. [44] for an example derivation in this grammar.

Animation 8.

$\}, \{a, b\}, S, \{S \rightarrow aSb, S \rightarrow$



$aaaaaSbbbbb$

Remember that in Example 29. [36] we have shown that this language is not regular, and thus, with Example 36 [44] we have just proven that the class of linear languages strictly includes the class of regular languages.

Theorem 11. *Every linear language can be generated by a grammar having productions in the following forms, only:*

$A \rightarrow aB, A \rightarrow Ba, A \rightarrow a, S \rightarrow \lambda.$

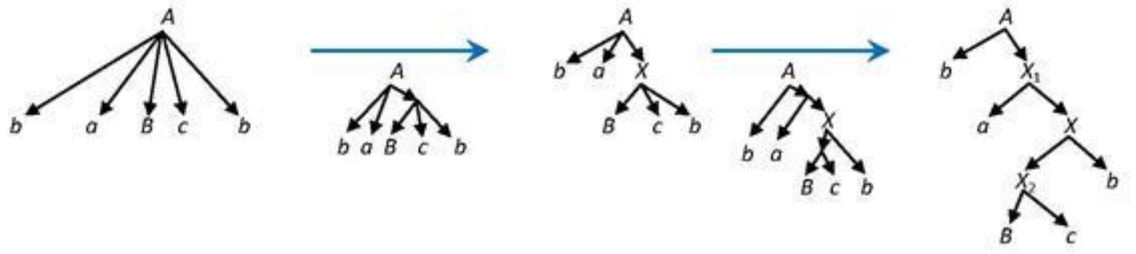
(We call a grammar with this property a linear grammar in normal form.)

Proof. The proof is constructive. Given $G = (N, T, S, P)$ a linear grammar, we show how one can construct a grammar G' that is in normal form and equivalent to G.

If there is a (are) rule(s) of the form $A \to \lambda$ $(A \neq S)$, then first the Empty-word lemma (Theorem 1. [9]) is applied and grammar $G'' = (N'', T, S', P'')$ is obtained that may only contain λ in the right hand side of the rule $S' \to \lambda$. Then in G'' either $S = S'$, $N'' = N$ or $S' \notin N$ and in this latter case $N'' = N \cup \{S'\}$.

Now as an intermediate step, we replace each of the rules $A \to uBv$ (where $u \neq \lambda$, $v \neq \lambda$) with rules $A \to uX$, $X \to Bv$, where X is a newly introduced nonterminal, i.e., $X \notin N''$. After the substitution of each rule of this form grammar $G''' = (N''', T, S', P''')$ is obtained and it is equivalent to the original grammar G. (See the left side of Figure 3.1., for an example.)

3.1. ábra - In derivations the rules with long right hand side (left) are replaced by chains of shorter rules in two steps, causing a binary derivation tree in the resulting grammar (right).



Now let us eliminate the rules having more than one terminal in their right hand side (i.e., they have long right hand side). Actually, rules of the form $A \to a_1 \dots a_k$ for $k > 1$, where $a_i \in T$, $i \in \{1,\dots,k\}$ and $A \to a_1 \dots a_kB$ for $k > 1$, where $a_i \in T$, $i \in \{1,\dots,k\}$, $B \in N'''$ can be substituted in the same manner as we have eliminated them in regular grammars (see the proof of Theorem 2 [13] for details). We present a similar technique for the rules of the form $A \to Ba_1 \dots a_k$ for $k > 1$, where $a_i \in T$, $i \in \{1,\dots,k\}$, $B \in N'''$, since rules of this type were not present in regular grammars. Every rule of the above form is substituted by a chain of shorter rules introducing new nonterminals into the grammar: let the new nonterminals $X_1,\dots, X_{k-1}$ be introduced and put to the set N''', and instead of the actual rule the next set of rules is added to P''':

$$A \to X_1a_k,\ X_1 \to X_2a_{k-1},\ \dots,\ X_{k-2} \to X_{k-1}a_2,\ X_{k-1} \to Ba_1.$$

(See the right hand side of Figure 3.1, for an example.)

Now a grammar $G'''' = (N'''', T, S' ,P'''')$ is obtained and the set of productions P'''' can contain only rules of the following forms

$$A \to a,\ A \to aB,\ A \to Ba,\ A \to B,\ S' \to \lambda$$

$(A, B \in N'''', a \in T)$. Now, as a final step of our (algorithm) proof we need to exclude the chain rules (rules of the form $A \to B$). This step can be done in a similar way as we showed in the proof of Theorem 2 [13]: first the set $U(A)$ is determined for each nonterminal A, and then the grammar $G' = (N'''', T, S', P')$ is obtained having $P' = \{A \to r|\ \exists\ B \in N''''$ such that $B \to r \in P''''$, $r \notin N''''$ and $B \in U(A)\}$. This grammar is in a normal form and it generates the same language as G, so the (construction) proof is finished.

QED.

Example 37. *Let*

$$G = (\{S, A, B\},\{1,3,7\}, S,$$

```
{   S → 11S37,
    S → 7A,
    S → B313,
    A → 333B7777,
```

$A \to S,\ B \to \lambda,\ B \to 731$
}).

Give a linear grammar in a normal form that is equivalent to G.

Solution:

Since there is a rule $B \longrightarrow \lambda$ in the grammar, we start by applying the Empty-word lemma. Then set $U = \{B\}$ and it is the set of nonterminals from which the empty word λ can be derived. Consequently, we obtain the grammar

$G'' = (\{S, A, B\}, \{1,3,7\}, S,$

{ $S \to 11S37,$
$S \to 7A,$
$S \to B313,\ S \to 313,$
$A \to 333B7777,\ A \to 3337777,$
$A \to S,$
$B \to 731$
}).

We have the rules $S \longrightarrow 11S37$ and $A \longrightarrow 333B7777$ having terminals on both sides of the nonterminal in the right hand side, thus, the intermediate step results in the grammar

$G''' = (\{S, A, B, X_1, X_2\}, \{1,3,7\}, S,$

{ $S \to 11X_1,\ X_1 \to S37,$
$S \to 7A,$
$S \to B313,$
$S \to 313,$
$A \to 333X_2,\ X_2 \to B7777,$
$A \to 3337777,$
$A \to S,$
$B \to 731$
}).

Now let us replace the rules with more than one terminals on their right hand side by chains of rules having exactly one terminal on their right hand sides:

$G'''' = (\{S, A, B, X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9, X_{10}, X_{11}, X_{12}, X_{13}, X_{14}, X_{15}, X_{16}, X_{17}, X_{18}, X_{19}, X_{20}, X_{21}\}, \{1,3,7\}, S,$

{ $S \to 1X_3,\ X_3 \to 1X_1,$
$X_1 \to X_47,\ X_4 \to S3,$
$S \to 7A,$
$S \to X_53,\ X_5 \to X_61,\ X_6 \to B3,$
$S \to 3X_7,\ X_7 \to 1X_8,\ X_8 \to 3,$
$A \to 3X_9,\ X_9 \to 3X_{10},\ X_{10} \to 3X_2,$
$X_2 \to X_{11}7,\ X_{11} \to X_{12}7,\ X_{12} \to X_{13}7,\ X_{13} \to B7,$
$A \to 3X_{14},\ X_{14} \to 3X_{15},\ X_{15} \to 3X_{16},\ X_{16} \to 7X_{17},\ X_{17} \to 7X_{18},\ X_{18} \to 7X_{19},\ X_{19} \to 7,$
$A \to S,$
$B \to 7X_{20},\ X_{20} \to 3X_{21},\ X_{21} \to 1$
}).

However, grammar G'''' contains the chain rule $A \longrightarrow S$, and thus

$U(S) = \{S\}, U(A) = \{A,S\},$

for the other nonterminals the trivial sets are obtained since they do not appear in any chain rules: $U(B) = \{B\}$ and $U(X_i) = \{X_i\}$ *(for $1 \le i \le 21$). Thus, the result is:*

$G' = (\{S, A, B, X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9, X_{10}, X_{11}, X_{12}, X_{13}, X_{14}, X_{15}, X_{16}, X_{17}, X_{18}, X_{19}, X_{20}, X_{21}\}, \{1,3,7\}, S,$

{ $S \to 1X_3,\ A \to 1X_3,$

```
    X₃ → 1X₁,
    X₁ → X₄7,
    X₄ → S3,
    S → 7A,  A → 7A,
    S → X₅3,  A → X₅3,
    X₅ → X₆1,
    X₆ → B₃,
    S → 3X₇,  A → 3X₇,
    X₇ → 1X₈,
    X₈ → 3,
    A → 3X₉,
    X₉ → 3X₁₀,
    X₁₀ → 3X₂,
    X₂ → X₁₁7,
    X₁₁ → X₁₂7,
    X₁₂ → X₁₃7,
    X₁₃ → B7,
    A → 3X₁₄,
    X₁₄ → 3X₁₅,
    X₁₅ → 3X₁₆,
    X₁₆ → 7X₁₇,
    X₁₇ → 7X₁₈,
    X₁₈ → 7X₁₉,
    X₁₉ → 7,
    B → 7X₂₀,
    X₂₀ → 3X₂₁,
    X₂₁ → 1
}).
```

It is linear, in normal form and equivalent to G.

As special cases of linear grammars the right-linear grammars (i.e., our regular grammars) and also the so-called left-linear grammars are defined.

Definition 20. (Left-linear grammars). *A grammar $G = (N, T, S, P)$ is left-linear if each of its productions has one of the following forms: $A \longrightarrow u$, $A \longrightarrow Bu$, where $A,B \in N$, $u \in T^*$.*

We state the following interesting result about the languages that can be generated by left-linear grammars, without proof.

Theorem 12. *The language class that can be generated by left-linear grammars is exactly the class of regular languages.*

We note here that even though regular languages can be generated by using left linear or right linear rules, using both in the same grammar leads to a different language class.

Finally, in this section, we provide a few exercises.

Exercise 49. *Give a linear grammar that generates the language*

$$\{0^m 1^n 2^n | \; n, m \in \mathbb{N}\}.$$

Exercise 50. *Give a linear grammar that generates the language*

$$\{a^{3n} b^{2n} | \; n \in \mathbb{N}\}.$$

Exercise 51. *Give a linear grammar in normal form that generates the language*

$\{ucv| \; u, v \in \{a,b\}^*$ such that the number of a's are the same in u and $v\}$.

Exercise 52. *Let*

$G = (\{S, A, B, C\}, \{a, b, c\}, S,$

```
{   S → aaSbc,
    S → B,
    S → cccC,
    A → λ,
    A → aaa,
    A → aCabc,
    B → A,
    B → Bbbb,
    C → cAb,
    C → c
}).
```

Give a linear grammar in normal form that is equivalent to G.

2. 3.2. One-Turn Pushdown Automata

It will be shown in the next chapter, that the class of pushdown automata accepts exactly the class of context-free languages (Section 4.6). The class of linear languages can be recognized by a class of special pushdown automata, called one-turn pushdown automata. We will present these automata in detail in Subsection 4.6.4., when we are familiar with the concept of pushdown automata.

3. 3.3. Closure Properties

In this section we show that the class of linear languages is closed under union, but it is not closed under other regular operations and under other set-theoretical operations.

Theorem 13. *The class of linear languages is closed under union, i.e., the union of any two linear languages is also linear.*

Proof. The proof is constructive. Let L_1 and L_2 be linear languages. Let the linear grammars $G_1 = (N_1, T, S_1, P_1)$ and $G_2 = (N_2, T, S_2, P_2)$ generate the languages L_1 and L_2 such that $N_1 \cap N_2 = \emptyset$ (this can be done by renaming nonterminals of a grammar without affecting the generated language). Then, let

$$G = (N_1 \cup N_2 \cup \{S\}, T, S, P_1 \cup P_2 \cup \{S \to S_1, S \to S_2\}),$$

where $S \notin N_1 \cup N_2$, is a new symbol. It can be seen that G generates the language $L_1 \cup L_2$.

QED.

Theorem 14. *The class of linear languages is not closed under concatenation and Kleene-star.*

Instead of a formal proof we offer a suggestion:

Let us consider the language

$$L = \{a^n b^n \mid n > 0\}.$$

The languages $L \cdot L$ and L^* are not linear languages.

Theorem 15. *The class of linear languages is not closed under complement and intersection.*

Proof. Let us start with the intersection. Observe that both of the languages

$$L_1 = \{a^j b^j c^k \mid j, k \in \mathbb{N}\} \text{ and } L_2 = \{a^k b^j c^j \mid j, k \in \mathbb{N}\}$$

are linear. The intersection of these two languages is

$$L = L_1 \cap L_2 = \{a^j b^j c^j \mid j \in \mathbb{N}\}.$$

As we will prove it in Example this language is not context-free, and therefore it is not linear. This proves the non closure under intersection.

We are going to prove now that the class is not closed under complement. Consider the following language: $\{wcw \mid w \in \{a,b\}^*\}$ over the alphabet $\{a,b,c\}$. It is called the language of "marked-copy". In Example 43. [57]

we prove that this language is not context-free, and thus it is not linear. However, the complement of this language is a linear language.

QED.

Exercise 53. *Give a linear grammar that generates the complement of the language of marked-copy. Hints: it can be done as union of linear languages. A word can be in this complement if,*

- *it does not contain any c,*
- *it does contain at least two c's,*
- *it is of the form u c v, with u,v* $\in \{a,b\}^*$, but $|u| \neq |v|$,
- *it is of the form u c v, with u,v* $\in \{a,b\}^*$, *and* $|u| = |v|$, *but there is a mismatch letter:* $u = u_1 x u_2$ *and* $v = u_1$yu2, *where* $x,y \in \{a,b\}$, *but* $x \neq y$.

Exercise 54. *Give a grammar that generates the union of the languages generated by grammars* G_1 *and* G_2, *where*

$G_1 = (\{S_1, A_1, B_1\}, \{a,b,c\}, S_1,$

```
{   S₁ → aaS₁ccc,
    S₁ → A₁,
    A₁ → bB₁b,
    B₁ → bB₁,
    B₁ → b
})
```

and

$G_2 = (\{S_2, A_2, B_2, C_2\}, \{a,b,c\}, S_2,$

```
{   S₂ → cccS₂aa,
    S₂ → bA₂,
    A₂ → A₂b,
    A₂ → cB₂aa,
    A₂ → C₂,
    B₂ → bB₂,
    B₂ → baccab,
    C₂ → C₂c,
    C₂ → A₂
}).
```

4. fejezet - Context-free Languages

Summary of the chapter: *This chapter will mainly deal with the properties of the type-2 language class of the Chomsky hierarchy, called context-free languages. This language class has many practical applications used in various areas of computer science. We will mention some of the most important ones. First, we discuss the notation techniques used to describe the syntax of programming languages, the Backus-Naur form, and the syntax diagram. Second, we introduce a normal form for context-free languages. This normal form will be used in Section 4.5., which is dedicated to parsing. The first pumping lemma, the Bar-Hillel lemma will be explained, and the closure properties of the context-free language class will be proven. In the last part of this chapter we introduce the pushdown automaton, we show its features, and its applications.*

1. 4.1. Notation Techniques for Programming Languages

Notation techniques were introduced as simple methods to describe different parts of programming languages. These parts contain terminal and nonterminal symbols. Terminals are given, and nonterminals can be built up from terminals and already defined nonterminals by using simple operations. These operations are the following:

1. Concatenation, when symbols are written after each other.
2. Alternation is a selection from different possibilities.
3. Option is a special selection between a symbol and the empty word.
4. Repetition, when a symbol can be repeated any (≥ 0) number of times.

In this section we introduce two well known techniques, the Backus-Naur form (BNF) and the Syntax diagram, but many others have been introduced for a variety of reasons. For example, the Extended Backus-Naur form is an extended version of the standard BNF.

1.1. 4.1.1. Backus-Naur Form

BNF was designed by Peter Naur in 1963 as a simplified version of the notation technique of John Backus. It was used first to describe the programming language ALGOL60. Table 4.1. shows the marking of the operations used by BNF.

4.1. táblázat - Operations of the BNF metasyntax.

Definition	Concatenation	Alternation	Option	Repetition
:: =		|	[]	{}

As you can see, concatenation does not have any special mark, we just write the symbols after each other. We use a terminal symbol as it is, for example, the mark of one as a number is 1. For nonterminals we use their names between angle brackets. We have a special mark to define nonterminal symbols, followed by the description of the nonterminal.

Example 38. *In this example, we describe a non-negative binary number using BNF metasyntax.*

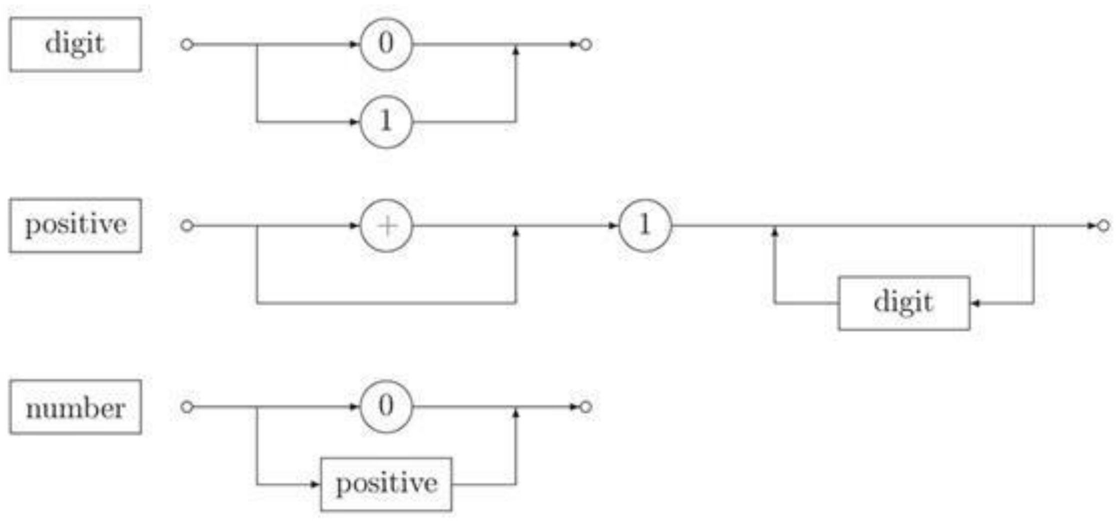
```
< digit > ::= 0 | 1
< positive > ::= [ + ] 1 { < digit > }
< number > ::= 0 | < positive >
```

1.2. 4.1.2. Syntax Diagram

A syntax diagram is a graphical notation technique. It uses simple graphs, each of them has an entry and an end point. The concatenation, alternation, option and repetition operations are implemented in the structure of the graph.

Example 39. *Figure 4.1. describes a non-negative binary number using the syntax diagram.*

4.1. ábra - Syntax diagram.



2. 4.2. Chomsky Normal Form

A generative grammar is said to be λ-free grammar if none of its production rules contains the empty word λ on the right hand side. We have to note that each $\lambda \notin L$ context-free language can be generated by some λ-free context-free grammar.

Definition 21. *The grammar $G = (N, T, S, P)$ is in Chomsky normal form, if all of its production rules has the form:*

1. *$A \rightarrow BC$ or*

2. *$A \rightarrow a$,*

where $A, B, C \in N$ and $a \in T$.

This normal form was introduced by Noam Chomsky for λ-free context-free languages. Using the Chomsky normal form instead of the universal context-free grammar makes it more simple to store the grammar in the memory of the computer, to calculate using the grammar and to prove theorems about context-free languages. First, we have to prove that each λ-free context-free language can be generated by a Chomsky normal form grammar.

Theorem 16. *For each λ-free context-free grammar $G = (N, T, S, P)$ one can give Chomsky normal form grammar $G' = (N', T, S, P')$ such that $L(G) = L(G')$.*

Proof. We are going to give a constructive proof of this theorem. We are going to show the necessary steps to construct a Chomsky normal form grammar $G' = (N', T, S, P')$ which is equivalent to the original λ-free context-free grammar $G = (N, T, S, P)$. It can easily be seen that we get equivalent grammars after each step.

1. First of all we create the grammar $G_1 = (N_1, T, S, P_1)$ such that all of its production rules are of the form:

   a. $A \rightarrow p$, or

   b. $A \rightarrow a$,

where $A \in N$, $a \in T$ and $p \in N^+$. In this step we eliminate each terminal symbol from the production rules whose right-hand side contains more than one letter. To do this we introduce new nonterminal symbols for each such terminal symbol. Let the set of new nonterminals be $N_{new} = \{X| \ a \in T, A \to paq \in P, | \ pq| \neq 0\}$ and let $N_1 = N \cup N_{new}$. Then let the set P_1 be the union of 3 different sets:

a. $\{A \to p| \ A \to p \in P, p \in \{T \cup N^+\}\} \subset P_1$ (we keep the rules for which the right hand side contains just one terminal, or contains only nonterminal symbols),

b. $\{X \to a| \ X \in N_{new}\} \subset P_1$ (we add new rules, the right hand side contains the terminal symbol, and the left hand side contains the nonterminal introduced for it),

c. $\{A \to p_0X_1p_1X_2... X_np_n| \ A \to p_0a_1p_1a_2... a_np_n \in P, a_1, a_2,...,a_n \in T, X_1, X_2,...,X_n \in N_{new}, p_0, p_1,...,p_n \in N^*, | \ p_0a_1p_1a_2... a_np_n| \geq 2\} \in P_1$ (here we change each appearance of the terminals to the nonterminal introduced for it in each rule, with right hand side of at least two letters).

Now we have the sets N_1 and P_1 and the grammar $G_1 = (N_1, T,S,P_1)$ satisfies the above conditions. It can be easily shown that $L(G_1) = L(G)$.

2. The next step is to eliminate the long rules. We create the grammar $G_2 = (N_2, T, S, P_2)$ such that all of its production rules are of the form:

   a. $A \to B$, or

   b. $A \to BC$, or

   c. $A \to a$,

   where $A, B, C \in N$ and $a \in T$. To reach our goal, we have to replace the long rules in P_1 with short ones in P_2. For each rule $A \to B_1B_2...B_k \in P_1$, $k \geq 3$ we introduce new nonterminals $Z_1, Z_2,...,Z_{k-2}$. The set N_2 contains these new nonterminals and the nonterminals contained by the set N_1. In the set P_2 we keep those productions rules from the set P_1 whose right hand side contains at most two letters and instead of each $A \to B_1B_2...B_k \in P_1$, $k \geq 3$ rule we introduce the rules

$$\begin{aligned} A &\to B_1Z_1, \\ Z_1 &\to B_2Z_2, \\ &\vdots \\ Z_{k-3} &\to B_{k-2}Z_{k-2}, \\ Z_{k-2} &\to B_{k-1}B_k. \end{aligned}$$

   The grammar $G_2 = (N_2, T, S, P_2)$ has no long rules and $L(G_2) = L(G_1)$.

3. The third step is to eliminate the rules of the form $A \to B$, where $A, B \in N$.

   First, for each nonterminal letter A let us collect all nonterminal letters $B_1, B_2, ..., B_k$ such that A can be derived from B_i, $1 \leq i \leq k$. Let $U(A) = \{B_1, B_2, ..., B_k\}$ for each nonterminal A. The following formulas make this pocedure simple:

   a. $U_1(A) = \{A\}$

   b. $U_{i+1}(A) = U_i(A) \cup \{B| \ B \to C \in P_2, C \in U_i(A)\}$

   c. if $U_k(A) = U_{k+1}(A)$ then $U(A) = U_k(A)$

   When we have set U for each nonterminal letter, we can define set P' with the following formula: $P' = \{B \to p| \ A \to p \in P_2, B \in U(A), p \notin N_2\}$. Then $N' = N_2$, $G' = (N', T, S, P')$ and $L(G') = L(G_2) = L(G_1) = L(G)$.

QED.

Example 40. *In this example we have a λ-free context-free grammar G, and we are going to create the grammar G' which is in Chomsky normal form and generates the same language as G.*

$G = (\{S, A, B\}, \{a,b,c\}, S, P)$
$P = \{$
$S \to ABaba,$
$A \to c,$
$A \to B,$
$A \to AS,$
$B \to AbA,$
$B \to S$
$\}$

1. *The terminals a and b appear in a rule which has more than 1 letter on the right hand side, so we have to add two new nonterminals X_a and X_b to the set of nonterminals: $N_1 = \{S, A, B, X_a, X_b\}$. Now we add two new rules ($X_a \to a$ and $X_b \to b$) to the set of production rules and replace the terminal symbol a by X_a and b by X_b in the rules which have more than one letter on the right hand side. Now we have*

$G_1 = (\{S, A, B, X_a, X_b\}, \{a,b,c\}, S, P_1),$
$P_1 = \{$
$A \to c,$
$A \to B,$
$A \to AS,$
$B \to S,$
$X_a \to a,$
$X_b \to b,$
$S \to ABX_aX_bX_a,$
$B \to AX_bA$
$\}.$

2. *In the set P_1 there are two long rules, $S \to ABX_aX_bX_a$ and $B \to AX_bA$. We add new nonterminals Z_1, Z_2, Z_3 to the first rule and Z_4 to the second one, and replace the rule $S \to ABX_aX_bX_a$ by rules*

$S \to AZ_1,$
$Z_1 \to BZ_2,$
$Z_2 \to X_aZ_3,$
$Z_3 \to X_bX_a,$
and the rule $B \to AX_bA$ *by rules*$B \to AZ_4,$
$Z_4 \to X_bA.$
Now we have$G_2 = (\{S, A, B, C, D, Z_1, Z_2, Z_3, Z_4\}, \{a,b,c\}, S, P_2),$
$P_2 = \{$
$A \to c,$
$A \to B,$
$A \to AS,$
$B \to S,$
$X_a \to a,$
$X_b \to b,$
$S \to AZ_1,$
$Z_1 \to BZ_2,$
$Z_2 \to X_aZ_3,$
$Z_3 \to X_bX_a,$
$B \to AZ_4,$
$Z_4 \to X_bA$
$\}.$

3. *$U(B) = \{B,A\}$ and $U(S) = \{S,B,A\}$, and finally we have:*

$G' = (\{S, A, B, C, D, Z_1, Z_2, Z_3, Z_4\}, \{a,b,c\}, S, P').$
$P' = \{$
$A \to c,$
$A \to AS,$
$X_a \to a,$
$X_b \to b,$
$S \to AZ_1,\ {}_B \to AZ_1,\ A \to AZ_1,$
$Z_1 \to BZ_2,$
$Z_2 \to X_aZ_3,$
$Z_3 \to X_bX_a,$
$B \to AZ_4,\ A \to AZ_4,$

```
Z4 → X_bA
}
```

Exercise 55. *In this exercise we have a λ-free context-free grammar G, and you have to create a grammar G' which is in Chomsky normal form and generates the same language as G.*

```
        G = ({S, A, B}, {x,y,z}, S, P)
P = {
   S → BB,
   A → S,
   A → xxzz,
   A → y,
   B → AxzxA,
   B → A
}
```

Exercise 56. *Create a grammar G' which is in Chomsky normal form and generates the same language as G.*

```
        G = ({S, A, B}, {x,y}, S, P)
P = {
   S → ABBAB,
   S → x,
   A → BB,
   A → S,
   A → B,
   B → ASA,
   B → y
}
```

Exercise 57. Create a grammar G' which is in Chomsky normal form and generates the same language as G.

```
        G = ({S, X, Z}, {x,y}, S, P)
P = {
   S → XZ,
   S → ZX,
   X → xy,
   X → S,
   Z → S,
   Z → yx,
   Z → X,
   Z → ZZ
}
```

Exercise 58. *Create a grammar G' which is in Chomsky normal form and generates the same language as G.*

```
        G = ({S}, {a,+,*,(,)}, S, P)
P = {
   S → S+S,
   S → S*S,
   S → (S),
   S → a
}
```

Exercise 59. *Create a grammar G' which is in Chomsky normal form and generates the same language as G.*

```
        G = ({S, A, B}, {x,y}, S, P)
P = {
   S → AxxB,
   S → A,
   S → B,
   B → A,
   A → y,
   A → SB
}
```

3. 4.3. Pumping Lemma for Context-free Languages

Although it is easy to find the exact position of a grammar in the Chomsky hierarchy, it is sometimes much more challenging to find the position of a language in the Chomsky hierarchy. The Bar-Hillel lemma is the first pumping - also called iteration - lemma, which gives properties shared by all context-free languages. Thus, if a language does not satisfy the conditions of the lemma, it is not context-free. This lemma - and its variations - can be used to show that a language is not context-free. On the other hand, languages satisfying the conditions may be not context-free.

Theorem 17. (Bar-Hillel lemma) *For each context-free language L there exists an integer $n \geq 1$ such that each string $p \in L$, $|p| \geq n$ can be written in a form $p = uvwxy$, where $|vwx| \leq n$, $|vx| \geq 1$ and $uv^iwx^iy \in L$ holds for each integer $i \geq 0$.*

Proof. *Let* $L_1 = L \setminus \{\lambda\}$. It is ovious that if language L_1 satisfies the above conditions then language $L = L_1 \cup \{\lambda\}$ also holds the above conditions, so it is enough to prove the lemma for λ-free context-free languages.

Theorem 16 [52] shows that each λ-free context-free language can be generated by a Chomsky normal form grammar. Further on let us assume that grammar G generating L_1 is in Chomsky normal form.

Let us mark the number of nonterminals in grammar G by k, and let $n = 2^k+1$. Let p be a word generated by grammar G, and let $|p| \geq n$. In this case, the height of the derivation tree of p is more than k+2, where the last step is a nonterminal to terminal derivation. Let us investigate the last k+2 height part of the longest path of the derivation tree. There must be a nonterminal A which appears twice, because the number of the nonterminals in G is less than the number of the nonterminals in this part. So there must be terminal words v, x such that $A \Rightarrow^* vAx$. Here $|vx| \geq 1$ because A has two different occurrences in the path, and the length of the generated word is increased by one in each derivation step, except for the derivation steps when we change a nonterminal to a terminal symbol. Also, there is a terminal word w, which can be derived from the last appearance of A on the path, so $A \Rightarrow^* w$ also holds. Moreover, there are terminal words u, y such that $S \Rightarrow^* uAy$. Based on these facts it is easy to show that

```
         S ⇒*uAy ⇒*uwyS ⇒*uAy ⇒*uvAxy ⇒*uvwxyS ⇒*uAy ⇒*uvAxy ⇒*uvvAxxy ⇒*uvvwxxy
⋮
```

This proves that $uv^iwx^iy \in L$ holds for each integer $i \geq 0$.

Finally, $|vwx| \leq n$, because the word vwx was derived with a derivation subtree of height at most k+2, - where the last step was a nonterminal to terminal derivation, - so the length of the word vwx is maximum 2^k which is less than n.

QED.

Example 41. *The following classical example shows an application of the Bar-Hillel lemma. We are going to prove that language $L = \{a^jb^jc^j | j \geq 0\}$ is not context-free. In order to do this suppose to the contrary that language L is context-free. Let $j \geq (n / 3)$, then, by the Bar-Hillel lemma, $a^jb^jc^j$ can be written in a form uv^iwx^iy such that $|vx| \geq 1$ and $uv^iwx^iy \in L$ holds for each integer $i \geq 0$. First, neither of v nor x should contain two or more different letters, because repeating them would change the form of the words in L. So v is a unary word (some power of a letter, e.g. aa...a) and x is a unary word as well. In this case, when we increase the integer i, we change the number of one or two different letters, but we cannot change the number of each letter, which is a contradiction.*

Example 42. *Let us consider the language $L = \{a^jb^kc^jd^k | j, k \geq 0\}$. Suppose to the contrary that language L is context-free. Then, the word $a^nb^nc^nd^n \in L$ can be written in a form uvwxy such that $uv^iwx^iy \in L$ for each $i \geq 0$. Suppose that the word v contains the letter a. In this case, the word x cannot contain the letter c, because $|vwx| \leq n$. Also, if the word v contains the letter b, then the word x cannot contain the letter d. In this case, we cannot increase the number of the letters a and c at the same time, and also we cannot increase the number of the letters b and d together, which means that this language does not satisfy the conditions of the Bar-Hillel lemma, consequently it is not context-free.*

Example 43. *In this example we consider the language $L = \{wcw|\ w \in \{a,b\}^*\}$, and we use the Bar-Hillel lemma to prove that this language is not context-free. Suppose to the contrary that the language L is context-free, in this case the word $a^nb^nca^nb^n$ can be written in a form uvwxy such that $uv^iwx^iy \in L$ for each $i \geq 0$. The only possible solution is that the word v contains letters before the letter c and the word x contains letters after the letter c, because in other cases the number of the letters before and after c will not be the same. Now, $|vwx| \leq n$, so the word v can contain only letter b and the word x can contain only letter a, which is a contradiction.*

Example 44. *In this example we show that the language $L = \{a^p|\ p$ pime$\}$ is not context-free. Suppose to the contrary that the language L is context-free. Let $p \geq n$, so a^p can be written in a form uvwxy such that $|vx| \geq 1$ and $uv^iwx^iy \in L$ holds for each integer $i \geq 0$. Now, let $q = |vx|$, $r = |uwy|$, so $a^{r+i\cdot q} \in L$ holds for each integer $i \geq 0$. This means that $r+i\cdot q$ is a prime for each integer $i \geq 0$. Here $r \neq 1$, because $1+i\cdot q = 1$ for $i = 0$, and 1 is not a prime number. Let $i=r$, then $r+r\cdot q$ should be a prime, but $r+r\cdot q = r\cdot(1+q)$ is not a prime, so we have a contradiction.*

4. 4.4. Closure Properties

Theorem 18. *The context-free language class is closed under regular operations.*

Proof. We are going to give a constructive proof for each regular operation one by one. We use two context-free languages, L_1 and L_2. Let the grammar $G_1 = (N_1, T, S_1, P_1)$ such that $L(G_1) = L_1$, and let the grammar $G_2 = (N_2, T, S_2, P_2)$ such that $L(G_2) = L_2$. Without loss of generality we can suppose that $N_1 \cap N_2 = \emptyset$. We are going to give the context-free grammars G_{Un}, G_{Co} and G_{Kl}, such that $L(G_{Un}) = L_1 \cup L_2$, $L(G_{Co}) = L_1 \cdot L_2$ and $L(G_{Kl}) = L_1^*$.

1. Union

   To create the grammar G_{Un} we need a new start symbol S, such that $S \cap N_1 = S \cap N_2 = S \cap T = \emptyset$. Then, let

   $$G_{Un} = (N_1 \cup N_2 \cup \{S\}, T, S, P1 \cup P_2 \cup \{S \rightarrow S_1, S \rightarrow S_2\}).$$

2. Concatenation

   For the grammar G_{Co} we also need a new start symbol S, such that $S \cap N1 = S \cap N_2 = S \cap T = \emptyset$. Then, let

   $$G_{Co} = (N_1 \cup N_2 \cup \{S\}, T, S, P_1 \cup P_2 \cup \{S \rightarrow S_1S_2\}).$$

3. Kleene star

   For the grammar G_{Kl} we again use a new start symbol S, where $S \cap N_1 = S \cap T = \emptyset$. Then, let

   $$G_{Kl} = (N_1 \cup \{S\}, T, S, P1 \cup \{S \rightarrow \lambda, S \rightarrow SS_1\}).$$

QED.

Theorem 19. The context-free language class is not closed under intersection.

Proof. It is easy to prove this theorem, because we need only one counterexample. Let the language $L_1 = \{a^ib^ic^j|\ i, j \geq 0\}$. L_1 is context-free, because we have a context-free grammar $G_1 = (\{S, A\}, \{a,b,c\}, S, \{S \rightarrow Sc, S \rightarrow A, A \rightarrow aAb, A \rightarrow \lambda\})$ such that $L_1 = L(G_1)$. Let the language $L_2 = \{a^ib^jc^j|\ i, j \geq 0\}$. The language L_2 is also context-free, because we have a context-free grammar $G_2 = (\{S, A\}, \{a,b,c\}, S, \{S \rightarrow aS, S \rightarrow A, A \rightarrow bAc, A \rightarrow \lambda\})$ such that $L_2 = L(G_2)$. The intersection of these two languages is $L_1 \cap L_2 = \{a^jb^jc^j|\ j \geq 0\}$, and in the Example 41 [56] it is proven by the Bar-Hillel lemma that this language is not context-free.

QED.

Theorem 20. *The context-free language class is not closed under the complement operation.*

Proof. Now we use proof by contradiction. The set theoretical version of one of the well known De Morgan's laws says that $\overline{A \cup B} = \bar{A} \cap \bar{B}$. With a slight modification, we have $L_1 \cap L_2 = \overline{\bar{L}_1 \cup \bar{L}_2}$. Suppose to the contrary that the context-free languages are closed under the complement operation. In this case, if L_1 and L_2 are context-free languages, the language defined by the right hand side of the expression must be context-free, however, since the left hand side may be non-context-free as in the previous theorem, a contradiction.

QED.

Theorem 21. *The intersection of a context-free language and a regular language is always context-free.*

Proof. We are going to give a constructive proof for this theorem. Let L_1 be a context-free language, and let L_2 be a regular language, where $L_2 = L_{21} \cup L_{22} \cup ... \cup L_{2n}$, and where each $L_{2i}(1 \leq i \leq n)$ is accepted by a deterministic finite automaton which has only one final state. We can do it without loss of generality, because it is well established fact that each regular language can be written in this form. Now

$$L_1 \cap L_2 = L_1 \cap \bigcup_{i=1}^{n} L_{2i} = \bigcup_{i=1}^{n} \left(L_1 \cap L_{2i} \right),$$

because intersection distributes over union. Since context-free languages are closed under union, we only have to show that $L_1 \cap L_{2i}$ is context-free for each i. Let the grammar $G = (N, T, S, P)$ be a Chomsky normal form grammar such that $L(G) = L_1 \setminus \{\lambda\}$, and let $DFA_i = (Q, T, q_0, \delta, q_f)$ be a deterministic finite automaton such that $L(DFA_i) = L_{2i}$. Now, we are going to define the context-free grammar $G' = (N', T, S', P')$ such that $L(G') = L_1 \cap L_{2i}$. The set of the nonterminals of G' is $N' = \{A_{[q_1,B,q_2]}|$ for each $q_1, q_2 \in Q, B \in N\}$. The production rules of the grammar G' are the following:

1. $S' \to A_{[q_0,S,q_f]} \in P'$,

2. $A_{[q_1,B,q_3]} \to A_{[q_1,C,q_2]}A_{[q_2,D,q_3]} \in P'$ for each possible $q_1, q_2, q_3 \in Q$, if $B \to CD \in P$,

3. $A_{[q_1,B,q_2]} \to a \in P'$ if $B \to a \in P$ and $\delta(q_1,a) = q_2$,

4. $S' \to \lambda \in P'$, if $\lambda \in L_1$ and $q_0 = q_f$.

It is easy to see that the grammar G' generates the word p, if and only if it is generated by grammar G, and it is accepted by the automaton L_{2i} as well. Each production rule of the grammar G' is context-free, so the language generated by the grammar G' is context-free.

QED.

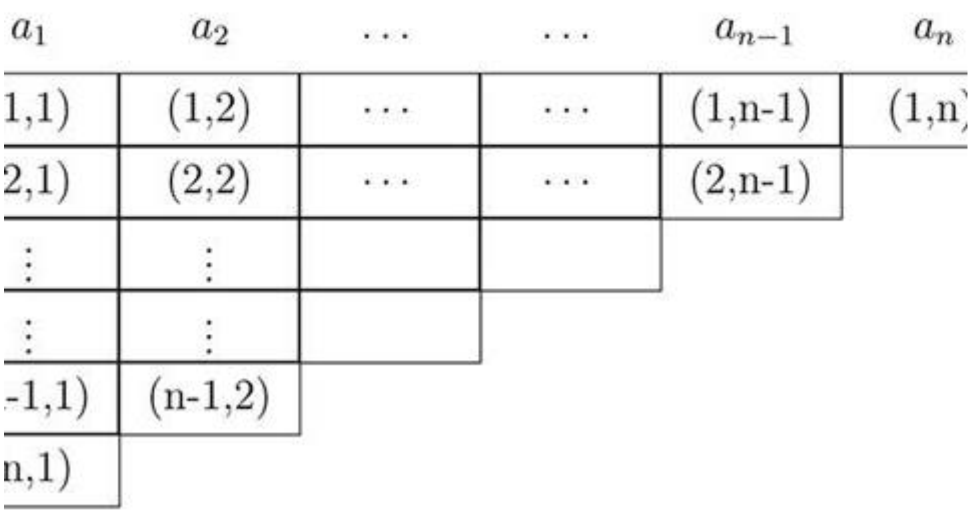
5. 4.5. Parsing

In formal language theory, parsing - or the so called syntactic analysis - is a process when the parser determines if a given string can be generated by a given grammar. This is very important for compilers and interpreters. For example, it is not too difficult to create a context-free grammar G_P generating all syntactically correct Pascal programs. Then, we can use a parser to decide - about a Pascal program written by a programmer - if the program is in the generated language $L(G_P)$. When the program is in the generated language, it is syntactically correct.

5.1. 4.5.1. The CYK Algorithm

We have a given Chomsky normal form grammar $G = (N, T, S, P)$ and a word $p = a_1a_2... a_n$. The Cocke-Younger-Kasami algorithm is a well known method to decide wether $p \in L(G)$ or $p \notin L(G)$. To make our decision, we have to fill out an $n \times n$ size triangular matrix M in the following way: Over the cells of the first line, we write the letters $a_1,a_2,...,a_n$, starting from the first letter, one after the other. Then, the cell $M(i,j)$ contains each nonterminal symbol A, if and only if the subword $a_ja_{j+1...}a_{j+i-1}$ can be derived from A. (Formally: $A \in M(i,j)$ if and only if $A \to^* a_ja_{j+1}...a_{j+i-1}$.) This means that the first cell of the first line contains the nonterminal A, if and only if $A \to a_1 \in P$. The cell $M(1,j)$ contains the nonterminal A, if and only if $A \to a_j \in P$. It is also quite easy to fill out the cells of the second line of the matrix. The nonterminal A is in the cell $M(2,j)$, if and only if there exists nonterminals $B, C \in N$ such that $B \in M(1,j)$, $C \in M(1,j+1)$, and $A \to BC \in P$. From this point the algorithm becomes more complex. From the third line, we use the following formula: $A \in M(i,j)$, if and only if there exists nonterminals $B, C \in N$ and integer k such that $B \in M(k,j)$, $C \in M(i\text{-}k, j+k)$ and $A \to BC \in P$. This algorithm is finished when the cell $M(n,1)$ is filled out. Remember, the nonterminal A is in the cell $M(i,j)$, if and only if the word $a_ja_{j+1}...a_{j+i-1}$ can be derived from A. This means that the nonterminal S is in the cell $M(n,1)$, if and only if the word a_1a_2...an can be derived from S. So the grammar G generates the word p, if and only if the cell $M(n,1)$ contains the start symbol S.

4.2. ábra - The triangular matrix M for the CYK algorithm.

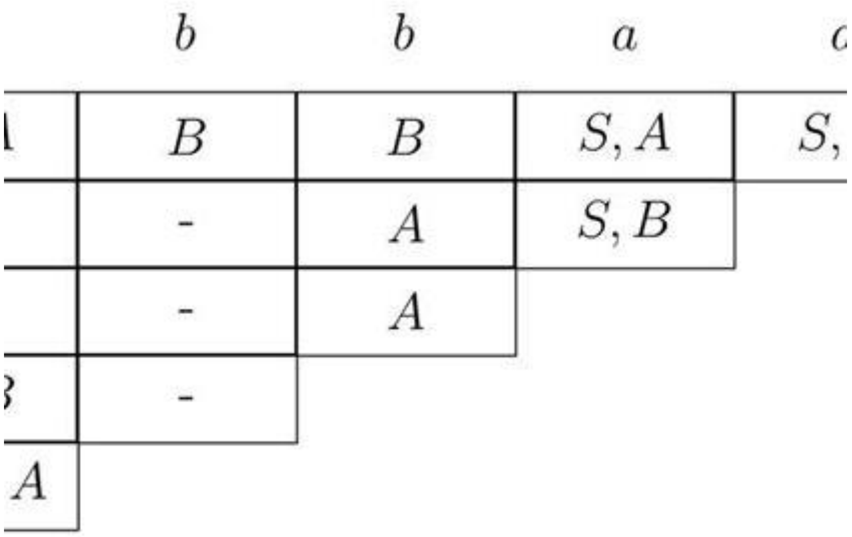


a_1	a_2	$\cdots$	$\cdots$	a_{n-1}	a_n
1,1)	(1,2)	$\cdots$	$\cdots$	(1,n-1)	(1,n
2,1)	(2,2)	$\cdots$	$\cdots$	(2,n-1)	
$\vdots$	$\vdots$				
$\vdots$	$\vdots$				
-1,1)	(n-1,2)				
n,1)					

Example 45. In this example, we use the CYK algorithm to show that the grammar G generates the word abbaa.

```
           G = ({S, A, B}, {a,b}, S, P)
P = {
   S → SA,
   S → AB,
   A → BS,
   B → SA,
   S → a,
   A → a,
   B → b
}
```

4.3. ábra - The triangular matrix M for the CYK algorithm of the Example 45 [59].



	b	b	a	a
	B	B	S, A	$S,$
	-	A	S, B	
	-	A		
	-			
A				

As you can see, S ∈ M(5,1), so abbaa ∈ L(G).

Exercise 60. *Use the CYK algorithm to show that the grammar G generates the word baabba.*

```
           G = ({S, A, B, X, Y, Z}, {a,b}, S, P)
P ={
    S → AY,
    Y → XB,
    X → BA,
    X → ZA,
    Z → BX,
    A → b,
    B → a
}
```

Exercise 61. *Use the CYK algorithm to decide if the word cbacab can be generated by grammar G.*

```
            G = (S, A, B, C, D}, {a,b,c}, S, P)
P = {
    S → AB,
    A → CA,
    A → SS,
    B → CD,
    A → b,
    C → a,
    C → b,
    D → c
}
```

5.2. 4.5.2. The Earley Algorithm

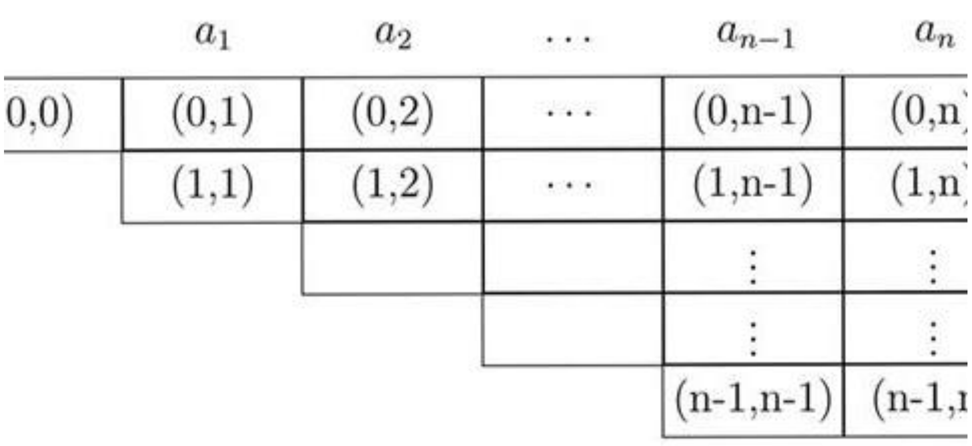
The Earley algorithm is designed to decide if a context-free grammar generates a terminal word. Sometimes it is not comfortable to create and use an equivalent Chomsky normal form grammar for a λ-free context-free grammar, because the Chomsky normal form grammar could have many more production rules than the original grammar. This is why the Earley algorithm is more widely used than the CYK algorithm for computerized lexical analysis. Although the Earley algorithm looks more complicated for humans, - and actually, it is more complicated compared to the the very simple CYK algorithm, - but after the implementation, there is no difference between the complexity of the two algorithms for computers.

Now we are going to show the steps of the λ-free version of the Earley algorithm. It can work with rules having form $A \to \lambda$ as well, with minor modification, but in practice we do not need the extended version.

5.2.1. Earley Algorithm

Let $G = (N, T, S, P)$ be a λ-free, context-free grammar, and $p = a_1a_2... a_n \in T^+$, with integer $n > 0$. We are going to fill out the cells of an $(n+1) \times (n+1)$ triangular matrix M, except for the last cell $M(n,n)$. Over the cells of the first line of the matrix, we write the letters $a_1, a_2,..., a_n$, starting from the second cell and first letter, one after the other. The elements of the matrix are production rules from P, where the right hand side of each rule contains a dot character.

4.4. ábra - The triangular matrix M for the Earley algorithm.



The steps of the algorithm are the following:

1. Let $S \rightarrow .q \in M(0,0)$ if $S \rightarrow q \in P$, and let $j = 0$.
2. Let $A \rightarrow .q \in M(j,j)$ if $A \rightarrow q \in P$, and there exists an integer $k \leq j$ such that $B \rightarrow r.At \in M(k,j)$.
3. Let $j = j+1$ and let $i = j-1$.
4. Let $A \rightarrow ra_j.t \in M(i,j)$ if $A \rightarrow r.a_jt \in M(i,j-1)$.
5. Let $A \rightarrow rB.t \in M(i,j)$ if there exists an integer $i \leq k < j$ such that $A \rightarrow r.Bt \in M(i,k)$, and $B \rightarrow q. \in M(k,j)$.
6. If $i > 0$ then $i = i-1$ and goto 4.

   If $i = 0$ and $j < n$ then goto 2.

   If $i = 0$ and $j = n$ then finished.

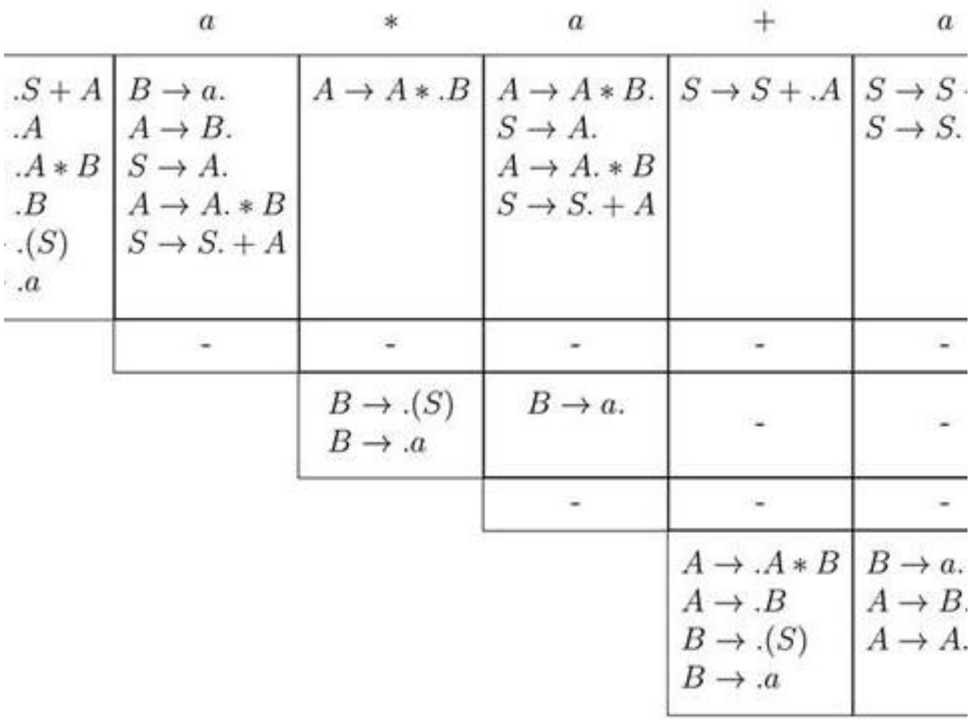
Here $q \in (T \cup N)^+$, $A, B \in N$, $r, t \in (T \cup N)^*$, and of course i,j,k are integers.

Grammar G generates the word p ($p \in L(G)$), if and only if there is a production rule in $M(0,n)$, whose left hand side is the start symbol S, and there is a dot at the end of the right hand side of the rule.

Examplee 46. *In this example, we have a λ-free context-free grammar G, and we have to decide if the word a*a+a can be generated by this grammar.*

```
            G = ({S, A, B}, {a,+,*,(,)}, S, P)
P = {
   S → S+AA → A*BB → (S)
   S → AA → BB → a
}
```

4.5. ábra - The triangular matrix M for the Earley algorithm of the Example 46. [61]



	a	$*$	a	$+$	a
$.S+A$ $.A$ $.A*B$ $.B$ $.(S)$ $.a$	$B \rightarrow a.$ $A \rightarrow B.$ $S \rightarrow A.$ $A \rightarrow A.*B$ $S \rightarrow S.+A$	$A \rightarrow A*.B$	$A \rightarrow A*B.$ $S \rightarrow A.$ $A \rightarrow A.*B$ $S \rightarrow S.+A$	$S \rightarrow S+.A$	$S \rightarrow S$ $S \rightarrow S.$
	-	-	-	-	-
		$B \rightarrow .(S)$ $B \rightarrow .a$	$B \rightarrow a.$	-	-
			-	-	-
				$A \rightarrow .A*B$ $A \rightarrow .B$ $B \rightarrow .(S)$ $B \rightarrow .a$	$B \rightarrow a.$ $A \rightarrow B.$ $A \rightarrow A.$

*As you can see, the top right cell contains a rule, whose left hand side is the start symbol S, and there is a dot at the end of the right hand side of the rule, so a+a*a ∈ L(G).*

Exercise 62. *Use the Earley algorithm to decide if the word* 100110 *can be generated by grammar G.*

```
            G = ({S, A, B}, {0,1}, S, P)
P = {
   S → 0A1,
   S → 1B0,
   A → B1,
   B → S1,
   A → 0,
   B → 1
}
```

Exercise 63. *Use the Earley algorithm to decide if the word bbabb can be generated by grammar G.*

```
            G = ({S, A, B}, {a,b}, S, P)
P = {
   S → BAB → bAB,
   A → BAb,
   B → SbA,
   A → a,
   B → b
}
```

6. 4.6. Pushdown Automata

Finite automata can accept regular languages, so we have to extend its definition so as it could accept context-free languages. The solution for this problem is to add a stack memory to a finite automaton, and the name of this solution is "pushdown automaton". The formal definition is the following:

Definition 22. *A pushdown automaton (PDA) is the following 7-tuple:*

$$PDA = (Q, T, Z, q_0, z_0, \delta, F)$$

where

- *Q is the finite nonempty set of the states,*
- *T is the set of the input letters (finite nonempty alphabet),*
- *Z is the set of the stack symbols (finite nonempty alphabet),*
- *q_0 is the initial state, $q_0 \in Q$,*
- *z_0 is the initial stack symbol, $z_0 \in Z$,*
- *δ is the transition function having a form $Q \times \{T \cup \{\lambda\}\} \times \mathrm{Z} \rightarrow 2^{Q \times Z^*}$, and*
- *F is the set of the final states, $F \subseteq Q$.*

In order to understand the operating principle of the pushdown automaton, we have to understand the operations of finite automata and the stack memory. Finite automata were introduced in Chapter 2, and we studied them through many pages. The stack is a LIFO (last in first out) memory, which has two operations, PUSH and POP. When we use the POP operation, we read the top letter of the stack, and at the same time we delete it. When we use the PUSH operation, we add a word to the top of the stack.

The pushdown automaton accepts words over the alphabet T. At the beginning the PDA is in state q_0, we can read the first letter of the input word, and the stack contains only z_0. In each step, we use the transition function to change the state and the stack of the PDA. The PDA accepts the input word, if and only if it can read the whole word, and it is in a final state when the end of the input word is reached.

More formally, in each step, the pushdown automaton has a configuration - also called instantaneous description - (q,v,w), where $q \in Q$ is the current state, $v \in T^*$ is the unread part of the input word, and $w \in Z^*$ is the whole word contained by the stack. At the beginning, the pushdown automaton is in its initial configuration: (q_0, p, z_0), where p is the whole input word. In each step, the pushdown automaton changes its configuration, while using the transition function. There are two different kinds of steps, the first is the standard, the second is the so called λ-step.

1. The standard step is when the PDA reads its current state, current input letter, the top stack symbol, it finds an appropriate transition rule, it changes its state, it moves to the next input letter and changes the top symbol of the stack to the appropriate word. Formally, we can say the PDA can change its configuration from (q_1, av, zw) to (q_2, v, rw) in one step, if it has a transition rule $(q_2, r) \in \delta(q_1, a, z)$, where $q_1, q_2 \in Q$, $a \in T$, $z \in Z$, $v \in T^*$, $w \in Z^*$. Denote this transition $(q_1, av, zw) \vdash_{PDA} (q_2, v, rw)$.

2. The λ-step is when the PDA reads its current state, it does not read any input letters, it reads the top stack symbol, it finds an appropriate transition rule, and it changes its state, it does not move to the next input letter and it changes the top letter of the stack to the given word. Formally, we can say again that the PDA can change its configuration from (q_1, v, zw) to (q_2, v, rw) in one step, if it has a transition rule $(q_2, r) \in \delta(q_1, \lambda, z)$, where $q_1, q_2 \in Q$, $z \in Z$, $v \in T^*$, and $w \in Z^*$. Mark: $(q_1, v, zw) \vdash_{PDA} (q_2, v, rw)$.

We can say that the PDA can change its configuration from (q_1, v, w) to (q_2, x, y) in finite many steps, if there are configurations $C_0, C_1, \ldots, C_n$ such that $C_0 = (q_1, v, w)$, $C_n = (q_2, x, y)$, and $C_i \vdash_{PDA} C_{i+1}$ holds for each integer $0 \leq i < n$. Mark: $(q_1, v, w) \vdash^*_{PDA} (q_2, x, y)$.

Finally, we can define the language accepted by the pushdown automaton:

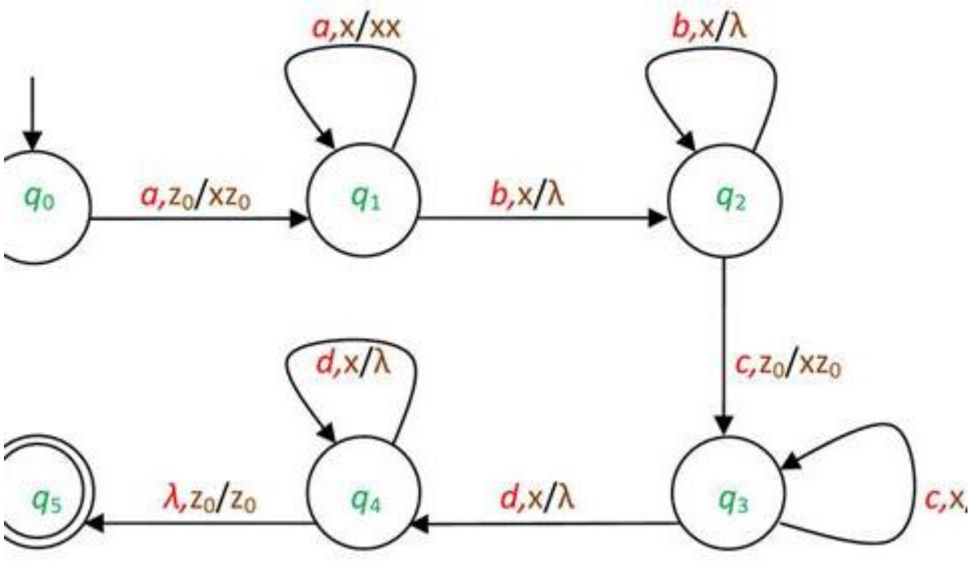
$$L(PDA) = \{p \mid p \in T^*, (q_0, p, z_0) \vdash^*_{PDA} (q_f, \lambda, y), q_f \in F, y \in Z^*\}.$$

Example 47. *This simple example shows the description of a pushdown automaton which accepts the language* $L = \{a^i b^i c^j d^j \mid i, j \geq 1\}$.

```
      PDA  =  ({q₀,q₁,q₂,q₃,q₄,q₅},{a,b,c,d},{x,z₀},q₀,z₀,δ,{q₅}),
  δ(q₀,a,z₀)  =  {(q₁,xz₀)},
  δ(q₁,a,x)  =  {(q₁,xx)},
  δ(q₁,b,x)  =  {(q₂,λ)},
  δ(q₂,b,x)  =  {(q₂,λ)},
  δ(q₂,c,z₀)  =  {(q₃,xz₀)},
  δ(q₃,c,x)  =  {(q₃,xx)},
  δ(q₃,d,x)  =  {(q₄,λ},
  δ(q₄,d,x)  =  {(q₄,λ)},
  δ(q₄,λ,z₀)  =  {(q₅,z₀)}.
```

The Figure 4.6. shows the graphical notation of this pushdown automaton.

4.6. ábra - The graphical notation for the Example 47. [63]

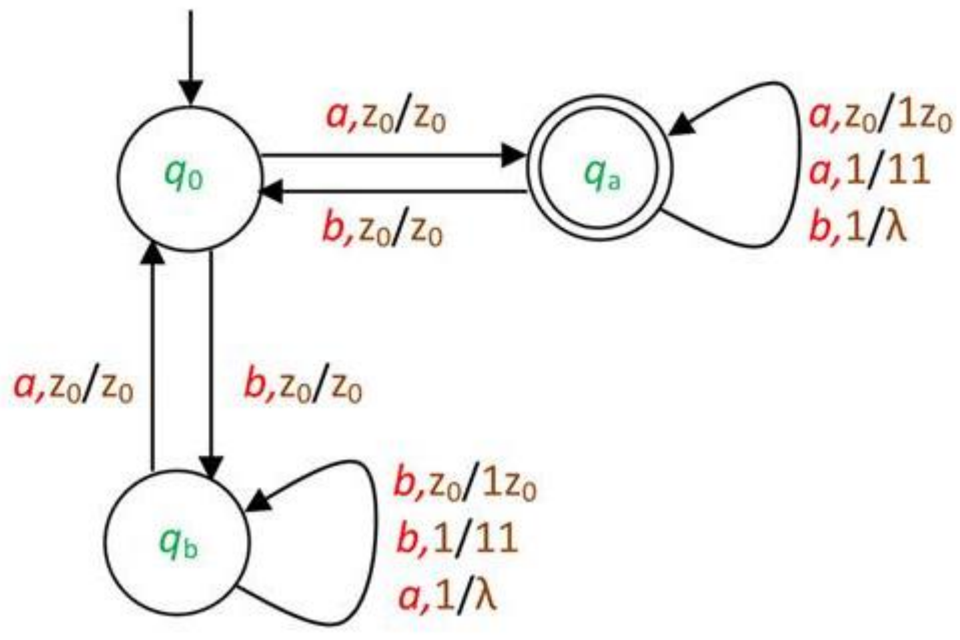

$$\begin{tikzcd}
\arrow[r] & q_0 \arrow[r, "{a,z_0/xz_0}"] & q_1 \arrow[loop above, "{a,x/xx}"] \arrow[r, "{b,x/\lambda}"] & q_2 \arrow[loop above, "{b,x/\lambda}"] \arrow[d, "{c,z_0/xz_0}"] \\
 & q_5 & q_4 \arrow[l, "{\lambda,z_0/z_0}"'] \arrow[loop above, "{d,x/\lambda}"] & q_3 \arrow[l, "{d,x/\lambda}"'] \arrow[loop right, "{c,x,}"]
\end{tikzcd}$$

Example 48. *This example shows the description of a pushdown automaton which accepts the language of words over the alphabet {a,b} containing more a's than b's.*

$$
\begin{aligned}
&PDA = (\{q_0,q_a,q_b\},\{a,b\},\{1,z_0\},q_0,z_0,\delta,\{q_a\}),\\
&\delta(q_0,a,z_0) = \{(q_a,z_0)\},\\
&\delta(q_0,b,z_0) = \{(q_b,z_0)\},\\
&\delta(q_a,a,z_0) = \{(q_a,1z_0)\},\\
&\delta(q_a,a,1) = \{(q_a,11)\},\\
&\delta(q_a,b,1) = \{(q_a,\lambda)\},\\
&\delta(q_a,b,z_0) = \{(q_0,z_0)\},\\
&\delta(q_b,b,z_0) = \{(q_b,1z_0)\},\\
&\delta(q_b,b,1) = \{(q_b,11)\},\\
&\delta(q_b,a,1) = \{(q_b,\lambda)\},\\
&\delta(q_b,a,z_0) = \{(q_0,z_0)\}.
\end{aligned}
$$

The Figure 4.7. shows the graphical notation of this pushdown automaton.

4.7. ábra - The graphical notation for the Example 48. [64]



Exercise 64. *Create a pushdown automaton, which accepts the language* $L = \{a^ib^j \mid 0 \leq i \leq j \leq 2i\}$.

Exercise 65. *Create a pushdown automaton, which accepts the language* $L = \{a^ib^jc^k \mid i = j \text{ or } j = k\}$.

Exercise 66. *Create a pushdown automaton, which accepts the language* $L = \{a^ib^jc^k \mid i = j \text{ or } i = k\}$.

6.1. 4.6.1. Acceptance by Empty Stack

There is another method for accepting words with a pushdown automaton. It is called "acceptance by empty stack". In this case, the automaton does not have any final states, and the word is accepted by the pushdown automaton if and only if it can read the whole word and the stack is empty when the end of the input word is reached. More formally, the language accepted by automaton

$$PDA_e = (Q, T, Z, q_0, z_0, \delta)$$

by empty stack is

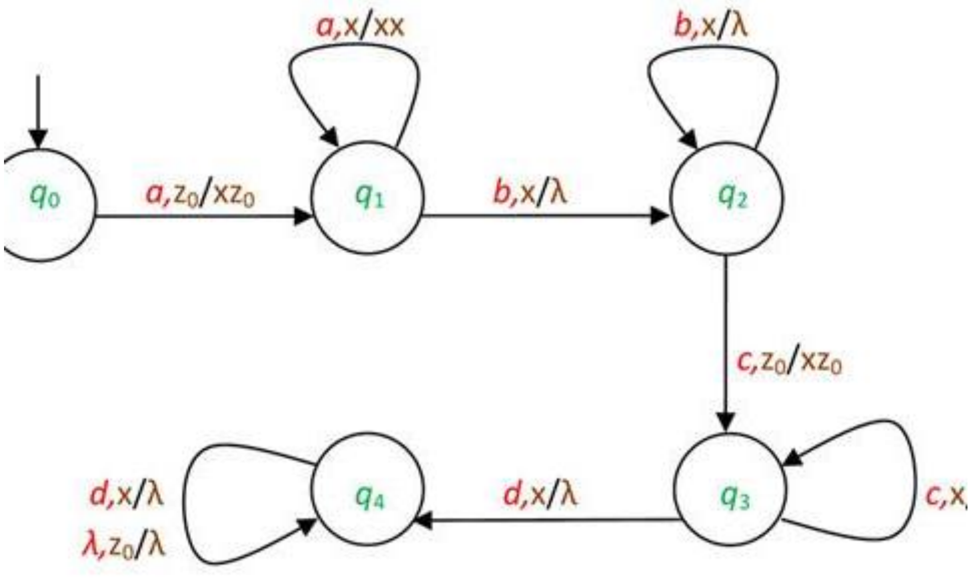
$$L(PDA_e) = \{p \mid p \in T^*, (q_0,p,z_0) \vdash^*_{PDA_e} (q,\lambda,\lambda), q \in Q\}.$$

Example 49. *This example shows the description of a pushdown automaton which accepts by empty stack the language $L = \{a^ib^ic^jd^j \mid i, j \geq 1\}$.*

```
        PDA = ({q0,q1,q2,q3,q4,q5},{a,b,c,d},{x,z0},q0,z0,δ,{q5}),
δ(q0,a,z0) = {(q1,xz0)},
δ(q1,a,x) = {(q1,xx)},
δ(q1,b,x) = {(q2,λ)},
δ(q2,b,x) = {(q2,λ)},
δ(q2,c,z0) = {(q3,xz0)},
δ(q3,c,x) = {(q3,xx)},
δ(q3,d,x) = {(q4,λ},
δ(q4,d,x) = {(q4,λ)},
δ(q4,λ,z0) = {(q4,λ}.
```

The Figure 4.8. shows the graphical notation of this pushdown automaton.

4.8. ábra - The graphical notation for the Example 49. [65]

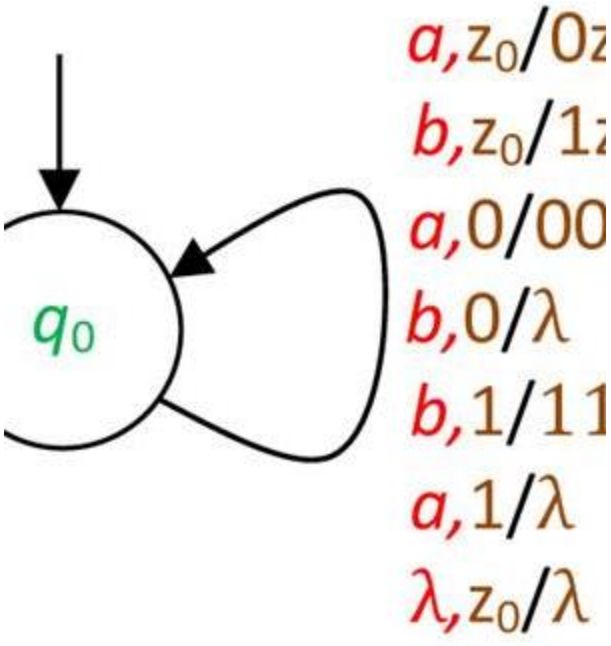


Example 50. *In this example, we are going to show the description of a pushdown automaton which accepts by empty stack the language with words over the alphabet {a,b} containing the same number of a's and b's.*

```
        PDA = ({q0},{a,b},{0,1,z0},q0,z0,δ),
δ(q0,a,z0) = {(q0,0z0)},
δ(q0,b,z0) = {(q0,1z0)},
δ(q0,a,0) = {(q0,00)},
δ(q0,b,0) = {(q0,λ)},
δ(q0,b,1) = {(q0,11)},
δ(q0,a,1) = {(q0,λ)},
δ(q0,λ,z0)={(q0,λ)}.
```

The Figure 4.9. shows the graphical notation of this pushdown automaton.

4.9. ábra - The graphical notation for the Example 50. [65]



The language class accepted by pushdown automata by final states and the language class accepted by pushdown automata by empty stack are the same. To prove this, we use two lemmas. First, we prove that for each PDA we can give PDA$_e$ such that $L(PDA_e) = L(PDA)$, second we show the reverse case.

Lemma 1. *For each PDA* = $(Q, T, Z, q_0, z_0, \delta, F)$ *we can give PDA$_e$* = $(Q', T, Z, q_0, z_0, \delta')$ *such that* $L(PDA_e) = L(PDA)$.

Proof. We are going to define a pushdown automaton PDA$_e$, which works the same way as the pushdown automaton PDA does, but each time when the original automaton goes into a final state, the new automaton goes into the state q_f, as well. Then, PDA$_e$ clears out the stack, when it is in the state q_f. Formally, let $Q' = Q \cup \{q_f\}$ where $\{q_f\} \cap Q = \emptyset$, and the transition function is the following:

1. Let $(q_2,r) \in \delta'(q_1,a,z)$ if $(q_2,r) \in \delta(q_1,a,z)$, for each $q_1, q_2 \in Q, a \in T \cup \{\lambda\}, z \in Z, r \in Z^*$,

2. let $(q_f,\lambda) \in \delta'(q_1,a,z)$ if $(q_2,r) \in \delta(q_1,a,z)$, for each $q_1 \in Q, q_2 \in F, a \in T \cup \{\lambda\}, z \in Z, r \in Z^*$, and

3. let $\delta'(q_f,\lambda,z) = \{(q_f,\lambda)\}$ for each $z \in Z$.

QED.

Lemma 2. *For each PDA$_e$* = $(Q, T, Z, q_0, z_0, \delta)$ *we can give PDA* = $(Q', T, Z', q'0, z'_0, \delta', F)$ *such that* $L(PDA) = L(PDA_e)$.

Proof. Again, we have a constructive proof. The automaton PDA first puts the initial stack symbol of the automaton PDA*e* over the new initial stack symbol. Then it simulates the original PDA*e* automaton, but each time when the original automaton clears the stack completely, the new automaton goes into the new final state q_f. The automaton PDA defined below accepts the same language with final states which is accepted by the original automaton PDA$_e$ with empty stack. Let $Q' = Q \cup \{q'_0,q_f\}$, where $\{q'_0\} \cap Q = \{q_f\} \cap Q = \emptyset$, let $Z' = Z \cup \{z'_0\}$, where $\{z'_0\} \cap Z = \emptyset$, and let $F = \{q_f\}$, so $\{q_f\}$ is the only final state, q'_0 is the new initial state, and z'_0 is the new initial stack symbol. The transition function is the following:

1. Let $\delta'(q'_0, \lambda, z'_0) = \{(q_0,z_0z'_0)\}$,

2. let $(q_2,r) \in \delta'(q_1,a,z)$ if $(q_2,r) \in \delta(q_1,a,z)$, for each $q_1, q_2 \in Q, a \in T \cup \{\lambda\}, z \in Z, r \in Z^*$, and

3. let $\delta'(q,\lambda,z'_0) = \{(q_f,\lambda)\}$ for each $q \in Q$.

QED.

Theorem 22. *The language class accepted by pushdown automata with final states is the same as the language class accepted by pushdown automata with empty stack.*

Proof. This theorem is a direct consequence of Lemma 1. [66] and Lemma 2. [66]

QED.

The proof of Lemma 2. [66] has the following consequences:

For each pushdown automaton we can give another pushdown automaton which accepts the same language with only one final state.

For each pushdown automaton we can give another pushdown automaton which accepts the same language with a final state and with empty stack at the same time.

6.2. 4.6.2. Equivalence of PDAs and Context-free Grammars

In this subsection we are going to prove that the languages accepted by pushdown automata are the context-free languages. Again, we are going to give constructive proofs. First, we demonstrate that for each pushdown automaton we can give a context-free grammar generating the same language as accepted by the PDA_e with empty stack, then we show how to construct a PDA_e which accepts the language generated by a context-free grammar.

Lemma 3. *For each* $PDA_e = (Q, T, Z, q_0, z_0, \delta)$ *we can give a context-free grammar* $G = (N, T, S, P)$ *such that* $L(G) = L(PDA_e)$.

Proof. The set of input letters of the PDA*e* and the set of terminal symbols of grammar G are the same. The set of nonterminal letters is $N = \{S\} \cup \{A_{[q,z,t]} |$ *for each* $q, t \in Q, z \in Z\}$. The production rules are the following:

1. $S \rightarrow A_{[q_0,z_0,q]} \in P$ for each $q \in Q$,

2. $A_{[q,z,t]} \rightarrow aA_{[t,z_1,q_1]}A_{[q_1,z_2,q_2]}\ldots A_{[q_{n-1},z_n,q_n]} \in P$ for each possible $q_1,\ldots, q_n \in Q$, if $(t, z_1z_2\ldots z_n) \in \delta(q,a,z)$, where $a \in T \cup \{\lambda\}$,

3. $A_{[q,z,t]} \rightarrow a \in P$, if $(t,\lambda) \in \delta(q,a,z)$, where $a \in T \cup \{\lambda\}$.

Grammar G simulates the work of the PDA_e in the following way: During the generating process the prefix of the generated word contains the the first part of the input word - which is already read by the pushdown automaton. It is followed by a nonterminal word $A_{[q,z_1,q_1]}A_{[q_1,z_2,q_2]}\ldots A_{[q_{n-1},z_n,q_n]}$, where $z_1z_2\ldots z_n$ is the word contained by the stack, q is the current state and $q_1,q_2,\ldots q_n$ can be any state. $A_{[q,z,t]}$ meaning that the automaton moves from state q to state t and removes the stack symbol z. The generated word keeps this structure during the generating process. When the automaton reaches the end of the input word and its stack is empty, then the word generated by the grammar contains the whole input word and does not contain any nonterminal symbols.

QED.

Lemma 4. *For each context-free grammar* $G = (N, T, S, P)$ *we can give a pushdown automaton* $PDA_e = (Q, T, Z, q_0, z_0, \delta)$ *such that* $L(PDA_e) = L(G)$.

Proof. The set of input letters of the PDA_e and the set of terminal symbols of grammar G are the same. Let $Q = \{q_0\}$, $Z = N \cup T$, , and $z_0 = S$. The production rules are very simple.

1. Let $(q_0,r) \in \delta(q_0,\lambda,A)$, if $A \rightarrow r \in P$, and

2. let $(q_0,\lambda) \in \delta(q_0,a,a)$ for each $a \in$ T.

During the computation of the PDAe, we use λ-steps to simulate the work of grammar G. The current word is always in the stack memory. We can remove the letters one by one, reading them from the input and clearing them at the same time from the top of the stack. The process is finished, when each letter is read and the stack is empty.

QED.

Theorem 23. *A language is context-free, if and only if it is accepted by some pushdown automaton.*

Proof. This theorem is a direct consequence of Lemma 3. [67] and Lemma 4. [67]

QED.

Finally, we have to note that for each context-free language we can give a pushdown automaton, which has only one state and accepts the context-free language by empty stack. This statement is a direct consequence of the proof of Lemma 4. [67]

6.3. 4.6.3. Deterministic PDA

Definition 23. *The pushdown automaton* $PDA_d = (Q, T, Z, q_0, z_0, \delta, F)$ *is deterministic, if*

1. $\delta(q,a,z)$ *has at most one element for each tripleq* $\in Q$, $a \in T \cup \{\lambda\}$, *and* $z \in Z$, *and*

2. *if* $\delta(q,\lambda,z)$, $q \in Q$, $z \in Z$ *has an element, then* $\delta(q,a,z) = \emptyset$ *for each* $a \in T$.

The language class accepted by deterministic pushdown automata with final states is a proper subset of the language class accepted by pushdown automata.

Definition 24. *The class of languages accepted by deterministic pushdown automata is called the class of deterministic context-free languages.*

In section 4.6.1. we have proven that the language class accepted by pushdown automata by final states and the language class accepted by pushdown automata by empty stack are the same. However, it is different for the deterministic case. The language class accepted by deterministic pushdown automata with empty stack is a proper subset of the language class accepted by deterministic pushdown automata with final states. Let us mark the deterministic pushdown automata accepting by empty stack with PDA_{de}. We can summarize these properties in the following expression:

$$L(PDA_{de}) \subset L(PDA_d) \subset L(PDA_e) = L(PDA).$$

6.4. 4.6.4. One-turn Pushdown Automata

In this subsection, we define a subclass of pushdown automata as it has already been mentioned in Subsection 3.2.

Definition 25. *The one-turn pushdown automaton (OTPDA) is a pushdown automaton* $PDA = (Q, T, Z, q_0, z_0, \delta, F)$ *with the following property:*

- *The set of states* $Q = Q_1 \cup Q_2$, *where* $Q_1 \cap Q_2 = \emptyset$,

- $q_0 \in Q_1$ *is the initial state,*

- $\delta : Q \times \{T \cup \{\lambda\}\} \times Z \to 2^{Q \times Z^*}$ *is the transition function such that each of its transitions is*

  - *either of the form* $(q',z') \in \delta(q,a,z)$ *with* $q \in Q_1$, $,q' \in Q$, $a \in T \cup \{\lambda\}$, $z \in Z$, $z' \in Z^+$,

  - *or of the form* $(q',z') \in \delta(q,a,z)$ *for* $q, q' \in Q_2$, $a \in T \cup \{\lambda\}$, $z \in Z$, $z' \in Z \cup \{\lambda\}$.

According to the above definition, it is clear that in a run once the automaton reaches a state $q' \in Q_2$, then it can never go back to a state in Q_1: after the stack content has been decreasing in a step, it cannot increase again. This fact also appears in the name of these special automata: their runs have (at most) one turn point (measuring the stack content).

Example 51. *Let* $PDA = (\{q_0,q_1,q_f\}, \{0,1,2\}, \{y,z\}, q_0, z, \delta, q_f)$ *be a one-turn pushdown automaton with*

Q

```
             $_1$ = {$q_0$},
  $Q_2$ = {$q_1$, $q_f$},
```

and δ is given by the following transitions:

```
  ($q_1$, $z$) ∈ δ ($q_0$,2,$z$),
  ($q_0$, $yz$) ∈ δ ($q_0$,0,$z$),
  ($q_0$, $yy$) ∈ δ ($q_0$,0,$y$),
  ($q_1$, $y$) ∈ δ ($q_0$,2,$y$),
  ($q_1$, λ) ∈ δ ($q_1$,1,$y$),
  ($q_f$, λ) ∈ δ($q_1$,2,$z$).
```

(For all other triplets of $\{q_0,q_1,q_f\} \times \{0,1,2\} \times \{y,z\}$ *there is no transition available. Thus, if PDA reaches a configuration defining a triplet non-listed above, it causes the process to stop without accepting.)*

The work of the automaton can be described as follows:

- *if the input is 22, then by reading the first 2, it changes its state to* q_1 *(first transition) and then, it reaches* q_f *by the last transition, and thus this input is accepted. (Observe that no other input starting with 2 can be accepted.)*
- *if the input is of the form* 0^n21^n2*, then by the second transition PDA is starting to count and by the third transition it is continuing to count the number of* 0*'s by pushing as many y's into the stack as the number of* 0*'s already read. Then, by reading a 2 PDA is at its turning point (having n y in the stack), and it changes its state to* q_1*. (Observe that there were no transitions defined for reading a* 1 *before this point.) Then, PDA is reading* 1*'s and counting their number by decreasing the number of y's in the stack (popping them out one by one). Finally, if and only if the number of* 0*'s are the same as the number of* 1*'s, then PDA can accept the output by reading the last* 2*.*
- *PDA is not accepting any other input.*

Thus, this pushdown automaton accepts the language

$$L(PDA) = \{0^n21^n2 \mid n \in \mathbb{N}\}.$$

We are going to present the following theorem without a proof.

Theorem 24. *The class of languages accepted by one-turn pushdown automata and the class of linear languages coincide.*

If there is at most one possible transition in every possible configuration of a one-turn pushdown automaton, then it is a deterministic one-turn pushdown automaton. Observe that *PDA* is deterministic in Example 51. [68] The deterministic variant of the one-turn pushdown automaton is weaker than the non-deterministic one, and thus the class of them accepts a proper subclass of linear languages, namely, the deterministic linear languages.

Example 52. *The language*

$$\{a^nb^n \mid n \in \mathbb{N}\} \cup \{a^nb^{3n} \mid n \in \mathbb{N}\}$$

is a union of two languages that are deterministic linear, however, it is not deterministic linear itself.

This chapter is concluded by some exercises.

Exercise 67. *Give a one-turn pushdown automaton that recognizes the language of palindromes (a palindrome is a word that is identical to its reverse). (Hint: this language cannot be accepted by deterministic OTPDA.)*

Exercise 68. *Give a deterministic one-turn pushdown automaton that recognizes the language*

$$\{a^nb^mc^{2n+3} \mid n, m \in \mathbb{N}\}$$

over the alphabet $\{a,b,c\}$.

Exercise 69. *Give a one-turn pushdown automaton that accepts the language*

$\{uc^*(c+d)ddv \mid u, v \in \{a,b\}^*$ *such that the number of a's in u and v are the same*$\}$.

5. fejezet - Context-Sensitive Languages

Summary of the chapter: *In this chapter, we deal with a family of languages generated by context-sensitive grammars of the Chomsky hierarchy, i.e., the context-sensitive languages. We are going to prove that monotone grammars generate the same language class. Normal forms of these grammars, such as Kuroda and Révész normal forms are provided. An example for a non-context-free, context-sensitive language is also given. The language class accepted by linear bounded automata coincides with the class of context-sensitive languages. This class is closed under the regular operations (union, concatenation, Kleene-star) and under the set theoretic operations complement and intersection. The word problem is solvable for these languages but no efficient algorithm is known for the general case.*

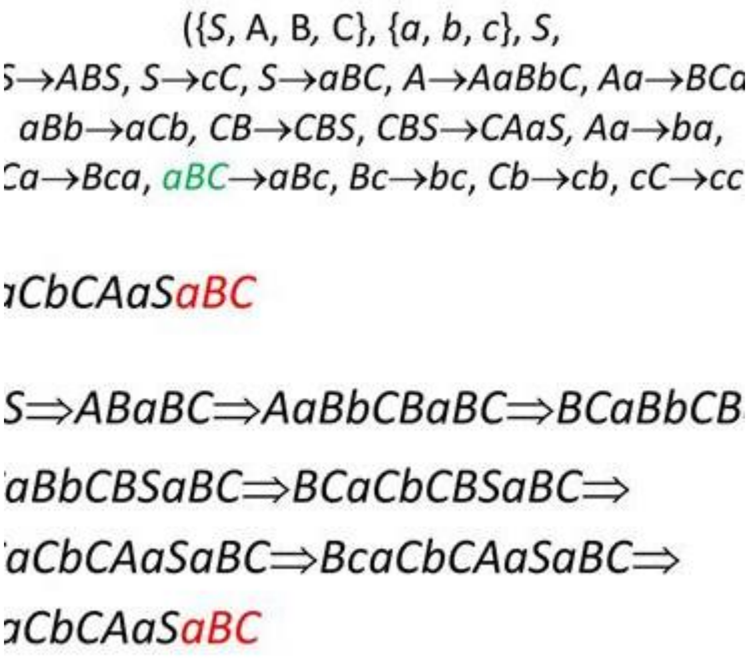
1. 5.1. Context-Sensitive and Monotone Grammars

For better understanding, we start this section by recalling the definition of context-sensitive grammars and languages.

Definition 26 (Context-sensitive grammars). *A grammar $G = (N, T, S, P)$ is context-sensitive if each of its productions has one of the following forms: $pAq \to puq$, where $A \in N$, $p, q \in (N \cup T)^*$, $u \in (N \cup T)^+$; $S \to \lambda$, and if $S \to \lambda \in P$, then S does not occur in the right hand side of any rule in P. The languages that can be generated by context-sensitive grammars are the context-sensitive languages.*

Example 53. *Animation 9. [71] shows an example for a context-sensitive grammar with a sample derivation.*

Animation 9.



({S, A, B, C}, {a, b, c}, S,
S→ABS, S→cC, S→aBC, A→AaBbC, Aa→BCa
aBb→aCb, CB→CBS, CBS→CAaS, Aa→ba,
Ca→Bca, aBC→aBc, Bc→bc, Cb→cb, cC→cc

aCbCAaSaBC

S⇒ABaBC⇒AaBbCBaBC⇒BCaBbCB
aBbCBSaBC⇒BCaCbCBSaBC⇒
aCbCAaSaBC⇒BcaCbCAaSaBC⇒
aCbCAaSaBC

We present yet another definition:

Definition 27. (Monotone grammars). *A grammar $G = (N, T, S, P)$ is monotone (or length-non-decreasing) if for each of its rules $p \to q$ ($p \in (N \cup T)^*N(N \cup T)^*$, $q \in (N \cup T)^+$)$|p| \leq |q|$, but the possible rule $S \to \lambda$. If $S \to \lambda$ is contained in P, then S does not occur in the right hand side of any rules of the grammar.*

It is an interesting property of monotone grammars that the terminals can be rewritten: this definition is so general that it allows to this (if there is a nonterminal close to that terminal), for example, by rule $aaaaB \to cccCCcc$ (with $a, c \in T; B, C \in N$).

According to the definitions, it is obvious that every context-sensitive grammar is monotone. The opposite will also be proven in this section, but first we are going to investigate a few normal forms.

1.1. 5.1.1. Normal Forms

In this subsection, two normal forms are presented.

Definition 28. (Kuroda normal form). *A monotone grammar $G = (N, T, S, P)$ is in Kuroda normal form, if it is monotone, and each of its rules is in one of the following forms:*

$$AB \to CD, A \to BC, A \to B, A \to a, S \to \lambda$$

($A,B,C,D \in N, a \in T$).

Since grammars in Kuroda normal form are monotone, in case $S \to \lambda$ is in the set of productions, the start symbol S cannot be in any right hand side of a rule. Kuroda normal form is a normal form, therefore we have the following theorem:

Theorem 25. *There is an equivalent grammar in Kuroda normal form for every monotone grammar.*

Proof. Let a monotone grammar $G = (N, T, S, P)$ be given. The proof is constructive: we present an algorithmic way to obtain the grammar in Kuroda normal form that is equivalent to G. Since G is monotone, the generated language $L(G)$ contains the empty word λ, if and only if there is a production $S \to \lambda$ in G. We need to deal only with the rules of the form $p \to q$ with $|p| \leq |q|$.

As a first step of our proof (algorithm), we obtain a grammar G'' that generates the same language as G, moreover, it has rules containing terminals only in rules of the form $A \to a$. So for each terminal a let us introduce a new nonterminal X_a ($X_a \notin N$), and replace each occurrence of all terminals in every rule by their new nonterminals (for example, a is replaced by X_a in every rule, both left and right hand side); and add the rules of the form $X_a \to a$ to the set of productions for each terminal:

$N'' = N \cup \{X_a | a \in T\}$,

$G'' = (N'', T, S,$

$\{p' \to q' | p \to q \in P$, and p' and q' are obtained from p and q, respectively, by replacing the occurrences of each terminal to the appropriate new nonterminal$\} \cup \{X_a \to a | a \in T\})$.

Observe that in G'' only nonterminals are rewritten. It can be seen that G' generates the same language as G, and the terminals can be derived only in the last steps of the derivations.

Now, if a rule of the monotone grammar G'' is not allowed to be in a grammar in Kuroda normal form, then this rule must have longer right hand side than it is allowed in Kuroda normal form (i.e., 2). Let us substitute each of these rules by a sequence of appropriate rules.

Let a rule $A_1... A_m \to B_1... B_n$ in P. Based on the definition of monotone grammars it is clear that $m \leq n$. Then

- if $n \leq 2$, then the rule is allowed in Kuroda normal form, and we leave it as is.
- if $m = 1$ and $n < 2$, we can do the same replacement as we have done at the Chomsky normal form for context-free grammars (see the proof of Theorem 16. [52]):

$$A_1 \to B_1... B_n$$

is replaced by the set of rules

$$A\ 1 \rightarrow B_1X_1,\ X_1 \rightarrow B_2X2,\ \dots\ X_{n\text{-}2} \rightarrow B_{n\text{-}1}B_n,$$

where $X_1,\dots,X_{n\text{-}2}$ are new nonterminals, were not in the grammar so far.

- if $m \geq 2$, $n > 2$, then

$$A\ _1\dots\ A_m \rightarrow B_1\dots\ B_n$$

is replaced by the set of rules

$$A\ _1\ A\ _2 \rightarrow B_1X_1,\ X_1A_3 \rightarrow B_2X_2,\ \dots\ X_{m\text{-}2}A_m \rightarrow B_{m\text{-}1}X_{m\text{-}1},$$

$$X_{m\ \text{-}1} \rightarrow B_mX_m,\ \dots\ ,\ X_{n\text{-}2} \rightarrow B_{n\text{-}1}B_n,$$

where $X_1,\dots,X_{n\text{-}2}$ are new nonterminals, not used in the grammar before. See also Figure 5.1.

5.1. ábra - In derivations the rules with long right hand side are replaced by chains of shorter rules.

The resulting grammar generates the same language as G, and it is in Kuroda normal form.

QED.

Example 54. Let

$$G = (\{S, A, B, C\}, \{0,1,2\}, S,$$

```
   {S → ABAB00,
    ABA → A111A,
    A111 → B221,
    B → 2, B → CC,
    BB → CBA,
    C → S,
    C → 021
}).
```

Give a Kuroda normal form grammar that is equivalent to G.

Solution:

In the first step, by introducing the nonterminals $D\ _0$, D_1, D_2 (using them instead of the terminals) we obtain *G''* as follows:

$$G'' = (\{S, A, B, C\ , D_0, D_1, D_2\}, \{0,1,2\}, S,$$

```
   {S → ABABD_0D_0,
    ABA → AD_1D_1D_1A,
    AD_1D_1D_1 → BD_2D_2D_1,
    B → D_2, B → CC,
    BB → CBA,
    C → S,
    C → D_0D_2D_1,
    D_0 → 0,
    D_1 → 1,
    D_2 → 2
}).
```

Now, we need to replace the rules which have too many letters (having right hand side longer than 2):

the original rule		*replaced by the set of rules*
$S \rightarrow ABABD_0D_0$:		$S \rightarrow AX_1, X_1 \rightarrow BX_2, X_2 \rightarrow AX_3, X_3 \rightarrow BX_4, X_4 \rightarrow D_0D_0,$
$ABA \rightarrow AD_1D_1D_1A$:		$AB \rightarrow AX_5, X_5A \rightarrow D_1X_6, X_6 \rightarrow D_1X_7, X_7 \rightarrow D_1A,$
$AD_1D_1D_1 \rightarrow BD_2D_2D_1$:		$AD_1 \rightarrow BX_8, X_8D_1 \rightarrow D_2X_9, X_9D_1 \rightarrow D_2D_1,$
$BB \rightarrow CBA$:		$BB \rightarrow CX_{10}, X_{10} \rightarrow BA,$
$C \rightarrow D_0D_2D_1$:		$C \rightarrow D_0X_{11}, X_{11} \rightarrow D_2D_1.$

All the other rules are kept, thus we have the solution

$G' = (\{S, A, B, C, D_0, D_1, D_2, X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9, X_{10}, X_{11}\}, \{0,1,2\}, S,$

$\{S \rightarrow AX_1, X_1 \rightarrow BX_2, X_2 \rightarrow AX_3, X_3 \rightarrow BX_4, X_4 \rightarrow D_0D_0,$

$AB \rightarrow AX_5, X_5A \rightarrow D_1X_6, X_6 \rightarrow D_1X_7, X_7 \rightarrow D_1A, AD_1 \rightarrow BX_8,$

$X_8D_1 \rightarrow D_2X_9, X_9D_1 \rightarrow D_2D_1, B \rightarrow D_2, B \rightarrow CC, BB \rightarrow CX_{10}, X_{10} \rightarrow BA,$

$C \rightarrow S, C \rightarrow D_0X_{11}, X_{11} \rightarrow D_2D_1, D_0 \rightarrow 0, D_1 \rightarrow 1, D_2 \rightarrow 2\}).$

The next observation was proven by György Révész, so this normal form is caled *Révész normal form*. Every rule $AB \rightarrow CD$ of a Kuroda normal form grammar can be replaced by a chain of rules

$AB \rightarrow AX, AX \rightarrow YX, YX \rightarrow YD, YD \rightarrow$ CD,

where X and Y are newly introduced nonterminals that are used only in these rules in the new grammar.

Definition 29. (Révész normal form). *A grammar $G = (N, T, S, P)$ is in Révész normal form, if each of its rules is in one of the following forms:*

$AB \rightarrow AC, AB \rightarrow CB, A \rightarrow BC, A \rightarrow B, A \rightarrow a, S \rightarrow \lambda$

($A,B,C \in N$, $a \in T$ and S does not occur in the right hand side of any rule if $S \rightarrow \lambda \in P$).

By using Révész's observation the following result is obtained:

Theorem 26. *There is an equivalent grammar in Révész normal form for every monotone grammar.*

Example 55. *Let*

$G = (\{S, A, B\}, \{a,b,c\}, S,$

```
   {S → BaB,
    BaB → BABa,
    A → BbB,
    A → c,
    B → BABB,
    B → AbbA,
    B → aB,
    B → c,
    S → λ
}).
```

Give a Révész normal form grammar that is equivalent to G.

Solution:

First, we obtain grammar G' that is in Kuroda normal form and generates the same language as G. Thus,

$G'' = (\{S, A, B, C_a, C_b, C_c\}, \{a,b,c\}, S,$

$$\begin{array}{l}
\{S \rightarrow BC_aB, \\
\ BC_aB \rightarrow BABC_a, \\
\ A \rightarrow BC_bB, \\
\ A \rightarrow C_c, \\
\ B \rightarrow BABB, \\
\ B \rightarrow AC_bC_bA, \\
\ B \rightarrow C_aB, \\
\ B \rightarrow C_c, \\
\ S \rightarrow \lambda \\
\}), \text{ and}
\end{array}$$

$G' = (\{S, A, B, C_a, C_b, C_c, X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8\}, \{a,b,c\}, S,$

$$\begin{array}{l}
\{S \rightarrow BX_1,\ X_1 \rightarrow C_aB, \\
\ BC_a \rightarrow BX_2,\ X_2B \rightarrow AX_3,\ X3 \rightarrow BC_a, \\
\ A \rightarrow BX_4,\ X_4 \rightarrow C_bB, \\
\ A \rightarrow C_c, \\
\ B \rightarrow BX_5,\ X_5 \rightarrow AX_6,\ X_6 \rightarrow BB, \\
\ B \rightarrow AX_7,\ X_7 \rightarrow C_bX_8,\ X_8 \rightarrow C_bA, \\
\ B \rightarrow C_aB, \\
\ B \rightarrow C_c, \\
\ S \rightarrow \lambda \\
\}).
\end{array}$$

Further, we need to replace the following rule by a chain of rules:

the original rule		*replaced by the set of rules*
$X_2B \rightarrow AX_3$		$X_2B \rightarrow X_2Y_1$, $X_2Y_1 \rightarrow Y_2Y_1$, $Y_2Y_1 \rightarrow Y_2X_3$, $Y_2X_3 \rightarrow AX_3$.

Thus, the Révész normal form grammar that is equivalent to G is

$G''' = (\{S, A, B, C_a, C_b, C_c, X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, Y_1, X_2\}, \{a,b,c\}, S,$

$\{S \rightarrow BX_1, X_1 \rightarrow C_aB, BC_a \rightarrow BX_2,$

$X_2B \rightarrow X_2Y_1, X_2Y_1 \rightarrow Y_2Y_1, Y_2Y_1 \rightarrow Y_2X_3, Y_2X_3 \rightarrow AX_3$

$X_3 \rightarrow BC_a, A \rightarrow BX_4, X_4 \rightarrow C_bB, A \rightarrow C_c,$

$B \rightarrow BX_5, X_5 \rightarrow AX_6, X_6 \rightarrow BB,$

$B \rightarrow AX_7, X_7 \rightarrow C_bX_8, X_8 \rightarrow C_bA, B \rightarrow C_aB, B \rightarrow C_c, S \rightarrow \lambda\}).$

One may observe that grammars in Révész normal form satisfy the conditions of the definition of context-sensitive grammars, and thus one can construct an equivalent context-sensitive grammar for any monotone grammar, i.e., the following theorem is proven.

Theorem 27. *The class of languages generated by monotone grammars coincides with the class of context-sensitive languages.*

By the previous theorem, we may use any monotone grammar to generate a context-sensitive language.

As we have shown by the Empty-word lemma (Theorem 1. [9]) every context-free language can be generated by context-sensitive grammars. Now, we are going to give an example that proves that the class of context-free languages is strictly included in the class of context-sensitive languages.

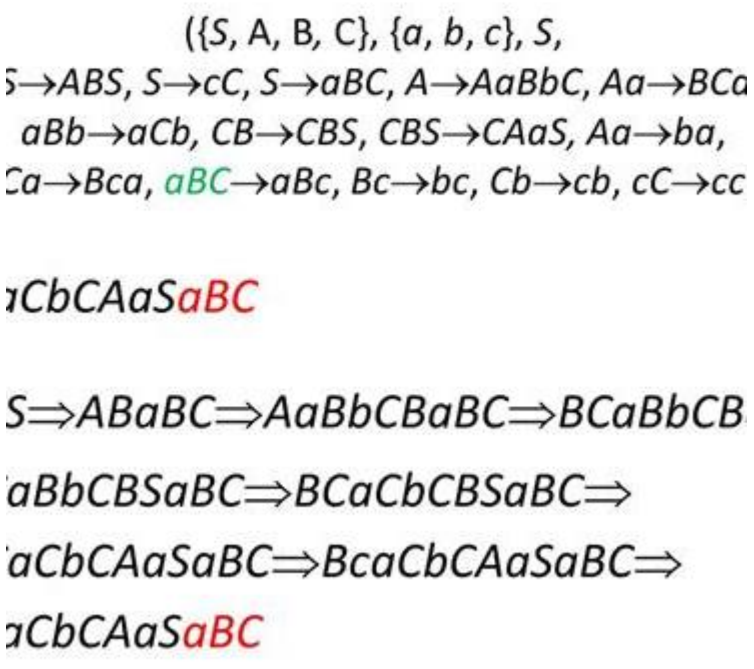
Example 56. *Let*

$G = (\{S, A, B, C\}, \{a,b,c\}, S,$

```
   {S → λ,
    S → abc,
    S → aABC,
    A → aABC,
    A → aBC,
    CB → BC,
    aB → ab,
    bB → bb,
    bC → bc,
    cC → cc
}).
```

Then λ and abc can be derived directly from S. Then every other (finished) derivation in this grammar applies $S \rightarrow aABC$, and then n times the rule $A \rightarrow aABC$ ($n \in \mathbb{N}$, $n \geq 0$) and finally the rule $A \rightarrow aBC$. In this way the sentential form starts with $n + 2$ a's and it contains $n + 2$ B's and C's. Then, every B must be positioned before the C's in a terminating derivation. Hence the generated language is $\{a^j b^j c^j \mid j \in \mathbb{N}\}$. See Animation 10. [76] for a terminal derivation in this grammar.

Animation 10.



({S, A, B, C}, {a, b, c}, S,
S→ABS, S→cC, S→aBC, A→AaBbC, Aa→BCa
aBb→aCb, CB→CBS, CBS→CAaS, Aa→ba,
Ca→Bca, aBC→aBc, Bc→bc, Cb→cb, cC→cc

aCbCAaSaBC

S⇒ABaBC⇒AaBbCBaBC⇒BCaBbCB
aBbCBSaBC⇒BCaCbCBSaBC⇒
aCbCAaSaBC⇒BcaCbCAaSaBC⇒
aCbCAaSaBC

Remember that in Example 41. [56] we have shown that this language is not context-free, but as it can be seen from Example 56. [75], it is context-sensitive.

We finish this section with some exercises.

Exercise 70. *Give a monotone grammar that generates the language*

$\{a^n b^m c^n d^m \mid n, m \in \mathbb{N}\}$.

Exercise 71. *Let*

$G = (\{S, A, B\}, \{0,1\}, S,$

```
   {S → SAS,
    SA → B0B0S,
    S → 1,
    A → S0S,
    B0B0 → 0S0S
}).
```

Give a Kuroda normal form grammar that is equivalent to G.

Exercise 72. *Let*

$G = (\{S, A, B, C\}, \{d,e\}, S,$

```
   {S → λ,
    S → BeBe,
    C → BeBe,
    BeBe → dAdA,
    eB → dede,
    Bd → CAC,
    A → ede,
    B → dd
}).
```

Give a Révész normal form grammar that generates the same language as G.

Exercise 73. *Let*

$G = (\{S, A, B, C, D\}, \{a,b,c\}, S,$

```
   {S → λ,
    S → AaBb,
    Aa → ccBbBb,
    Bb → CACA,
    bB → DaDa,
    DaD → CAC,
    bBbBb → ABCaD,
    A → a,
    B → bb,
    C → D,
    C → ccc
}).
```

Give a Révész normal form grammar that generates the same language as G.

Exercise 74. Let

$G = (\{S, A, B, C\}, \{a,b,c,d\}, S,$

```
   {S → BBC,
    S → SAB,
    A → cdC,
    cdA → CBbb,
    Bb → aa,
    bbA → dbd,
    A → a,
    A → d,
    B → b,
    C → SdS,
    C → cd
}).
```

Give a grammar in Révész normal form that generates the same language as G.

2. 5.2. Linear Bounded automata

We mention here that a special class of Turing machines, the class of linear bounded automata recognizes exactly the class of context-sensitive languages. All the details will be shown in Subsection 6.4., when the concept of the Turing machines have already been introduced.

3. 5.3. Properties of Context-Sensitive Languages

3.1. 5.3.1. Closure Properties

In this section, we prove that the class of context-sensitive languages is closed under union, concatenation and Kleene-star. It is also closed under the other set theoretical operations.

Theorem 28. *The class of context-sensitive languages is closed under the regular operations.*

Proof. The proof is constructive. Let L_1 and L_2 be two context-sensitive languages. Let the monotone grammars $G_1 = (N_1, T, S_1, P_1)$ and $G_2 = (N_2, T, S_2, P_2)$ be in Kuroda normal form and generate the languages L_1 and L_2, respectively, such that $N_1 \cap N_2 = \emptyset$ (this can be done by renaming nonterminals of a grammar without affecting the generated language).

First, we show the closure under union.

- If $\lambda \notin L_1 \cup L_2$, then let

  $$G = (N_1 \cup N_2 \cup \{S\}, T, S, P_1 \cup P_2 \cup \{S \to S_1, S \to S_2\}),$$

  where $S \notin N_1 \cup N_2$, a new symbol. It can be seen that G generates the language $L_1 \cup L_2$.

- If $\lambda \in L_1 \cup L_2$ (i.e., $S_1 \to \lambda \in P_1$ and/or $S_2 \to \lambda \in P_2$), then let

  $$G = (N_1 \cup N_2 \cup \{S\}, T, S,$$

  $$P_1 \cup P_2 \cup \{S \to S_1, S \to S_2, S \to \lambda\} \setminus \{S_1 \to \lambda, S_2 \to \lambda\}),$$

  where $S \notin N_1 \cup N_2$. In this way, G generates the language $L_1 \cup L_2$.

The closure under concatenation is proven for the following four cases:

- If $\lambda \notin L_1$ and $\lambda \notin L_2$, then let

  $$G = (N_1 \cup N_2 \cup \{S\}, T, S, P_1 \cup P_2 \cup \{S \to S_1S_2\}),$$

  where $S \notin N_1 \cup N_2$ a new symbol.

- If $\lambda \in L_1$ and $\lambda \notin L_2$, then let

  $$G = (N_1 \cup N_2 \cup \{S\}, T, S, P1 \cup P_2 \cup \{S \to S_1S_2, S \to S_2\} \setminus \{S_1 \to \lambda\}),$$

  where S is a new symbol.

- If $\lambda \notin L_1$ and $\lambda \in L_2$, then let

  $$G = (N_1 \cup N_2 \cup \{S\}, T, S, P_1 \cup P_2 \cup \{S \to S_1S_2, S \to S_1\} \setminus \{S_2 \to \lambda\}),$$

  where $S \notin N_1 \cup N_2$ a new symbol.

- If $\lambda \in L_1$ and $\lambda \in L_2$, then let

  $$G = (N_1 \cup N_2 \cup \{S\}, T, S,$$

  $$P_1 \cup P_2 \cup \{S \to S_1S_2, S \to S_1, S \to S_2, S \to \lambda\} \setminus \{S_1 \to \lambda, S_2 \to \lambda\}),$$

  where $S \notin N_1 \cup N_2$.

It can be easily seen that G generates the language L_1L_2.

Finally, let us consider the closure under Kleene-star. Let now $G_1 = (N_1, T, S_1, P_1)$ and $G_2 = (N_2, T, S_2, P_2)$ be in Kuroda normal form and generate the languages L (both G_1 and G_2) such that $N_1 \cap N_2 = \emptyset$.

Let

$G = (N_1 \cup N_2 \cup \{S, S'\},$

$P_1 \cup P_2 \cup \{S \to \lambda, S \to S_1, S \to S_1S_2, S \to S_1S_2S', S' \to S_1, S' \to S_1S_2, S' \to S_1S_2S'\} \setminus \{S_1 \to \lambda, S_2 \to \lambda\}),$

where $S, S' \notin N_1 \cup N_2$, they are new symbols. Then G generates the language L^*.

QED.

The closure of the class of context-sensitive languages under complementation was a famous problem and was open for more than 20 years. In the 1980's, Immerman, Szelepcsényi solved this problem independently. We present this result without proof.

Theorem 29. *The class of context-sensitive languages is closed under complementation.*

Theorem 30. *The class of context-sensitive languages is closed under intersection.*

Proof. The proof uses the fact that this class is closed both under union and complementation. Let us consider the context-sensitive languages L_1 and L_2. Then, the complement of each of them is context-sensitive according to the theorem of Immerman and Szelepcsényi. Their union is also context-sensitive, as we have proven constructively. The complement of this language is also context-sensitive. However, this language is the same as $L_1 \cap L_2$ by the De Morgan law.

QED.

Exercise 75. *Give a monotone grammar that generates the language of marked-copy:*

$\{wcw | \ w \in \{a,b\}^*\}$

over the alphabet $\{a,b,c\}$. *(Hint: use context-sensitive rules to allow some nonterminals to terminate, i.e., to change them to terminals only at the correct place.)*

Exercise 76. *Give context-sensitive grammars that generate the union and the concatenations of the languages generated by grammars* G_1 *and* G_2, *where*

$G_1 = (\{S_1, A_1, B_1, C_1\}, \{a,b,c\}, S_1,$

```
   {S₁ → λ,
    S₁ → C₁aA₁,
    S₁ → B₁,
    aA₁ → B₁B₁,
    B₁ → B₁b,
    B₁bb → cba,
    C₁ → cA₁
}) and
```

$G_2 = (\{S_2, A_2, B_2, C_2\}, \{a,b,c\}, S_2,$

```
   {S₂ → cccS₂aA₂a,
    S₂ → bA₂,
    ccS₂ → C₂bA₂,
    bA₂ → A₂b,
    A₂ → cB₂C₂a,
    A₂ → aa,
    cB₂ → bB₂,
    bB₂ → baccabB₂,
    C₂ → C₂c,
    C₂c → A₂S₂,
    C₂cc → cccc
}).
```

Exercise 77. *Let*

$G_1 = (\{S, A, B\}, \{0,1\}, S,$

```
   {S → λ,
    S → AB,
    A → 0AB,
    0A→ 1A,
    AB → 11
}) and
```

$G_2 = (\{S, B, C, D\}, \{0,1\}, S,$

```
   {S → λ,
    S → BC,
    S → D,
    B → BC,
    B → 1,
    1C → 1D0,
    D → DD,
    D → 11,
    1D → C0,
    BDC → 00D11
}).
```

Give context-sensitive grammars that generate

- $L(G_1) \cup L(G_2)$,
- $L(G_1)\, L(G_2)$,
- $L(G_2)\, L(G_1)$,
- $(L(G_1))^*$ and
- $(L(G_2))^*$.

3.2. 5.3.2. About the Word Problem

The word problem of context-sensitive grammars can be solved. Since this language class is accepted by linear bounded automata, it can be solved in linear space in a nondeterministic manner. It is known that polynomial space is sufficient with a deterministic algorithm: the required space is $c|w|^2$, where c is a constant and $|w|$ is the length of the word. It is an open problem if linear space (i.e., $c|w|$ with a constant c) is sufficient or not. Regarding time complexity, there is no deterministic or nondeterministic algorithm is known that can solve the word problem in polynomial time (for arbitrary context-sensitive grammar/language).

Now, we are going to present a naive solution to the word problem (which is very inefficient), but at the same time it shows that the problem is solvable. So let $G = (N, T, S, P)$ and w be given. If the input word is the empty word ($w = \lambda$), then it is in the language, if and only if $S \rightarrow \lambda \in P$. If $|w| > 0$, then we may consider only the rules $u \rightarrow v$ of P with the property $|u| \le |v|$. Then, let us use a bread-first search algorithm. Let the initial node of the graph be labeled by S. Consider the productions as possible operators on the sentential forms. Then, the search-graph can be obtained by applying every applicable rule for every node. This graph is usually infinite, but we need to obtain only a finite portion of it. (Every label must appear at most once in the graph.) Since each of the rules has the monotone property we may cut those branches of the search-space that contain a longer sentential form than w (the solution cannot be in the continuation of such a branch). When we have obtained all the portions of the search-graph representing sentential forms not longer than w, then we can check whether w is a node of the graph, or not. If so, then it can be derived, it is in the generated language, else it is not.

Exercise 78. *Check whether the words abc and ccc are in the language generated by the grammar*

$G = (\{S, A, B, C, D, E\}, \{a,b,c\}, S,$

```
   {S → ab,
    S → BS,
    S → ABS,
    A → a,
    A → BCE,
    AB → BA,
    B → b,
    BC → bc,
    CE → abc,
    D → b,
    ED → aE,
    E → DD,
    bA → SS
}).
```

6. fejezet - Recursively Enumerable Languages and Turing Machines

Summary of the chapter: *In this chapter, we discuss the most universal language class, the class of recursively enumerable languages, and the most universal automaton, the Turing machine. In the first section, we investigate the recursive and the recursively enumerable languages, and their closure and other properties. The second and third section is dedicated to the Turing machine. We will show two different applications. First, we are going to use the Turing machine as a universal language acceptor, then we show how we can use it as a simple but universal computing device.*

1. 6.1. Recursive and Recursively Enumerable Languages

At the beginning of this chapter we introduce the recursive languages and the recursively enumerable languages. These two language classes are fundamental in computability theory. There are many equivalent definitions, however, we are going to use these two definitions now, and we are going to show how these definitions fit for the concept of the Turing machine later.

Definition 30. *The language $L \in V^*$ is recursive, if there is an algorithm, which decides about any word $p \in V^*$, whether or not $p \in L$.*

We can say a language L is recursive, if the word problem can be solved in L. In Chapter 1. we define the class of the recursively enumerable languages as languages which can be generated by unrestricted generative grammars. Now, we give another definition.

Definition 31. *The language $L \in V^*$ is recursively enumerable, if there is a procedure, which specifies all the elements of L.*

The two definitions of the recursively enumerable languages are equivalent. If there is a procedure, which specifies all elements of the language L, then there is a generative grammar which generates the language L, and if there is a generative grammar generating the language L, then this grammar itself is a procedure, which can be used to specify all elements of the language L.

It is obvious that each recursive language is a recursively enumerable, because we can list the words over an alphabet V, and select those words which are contained by the language L.

Theorem 31. The language L is recursive, if and only if both L and $\bar{L}$ are recursively enumerable.

Proof. First, we show that, if $L \in V^*$ and $\bar{L} = V^* \setminus L$ are recursively enumerables, then L - and also $\bar{L}$ - is recursive. The language L is recursively enumerable, so there is a procedure which lists the elements of L. Let us denote these words p_1, p_2,.... However, $\bar{L}$ is recursively enumerable as well, so we have another procedure which lists the elements of $\bar{L}$. Let us denote these words r_1,r_2,.... Now we can combine these two procedures, to use the first one, and then the second one, alternately. What we receive is the list of all words over the alphabet V: p_1, r_1, p_2, r_2,... and we know about each one if it belongs to the language L or not.

Now, we show that if L is recursive, then L and $\bar{L}$ is recursively enumerable. We already mentioned that recursive languages are recursively enumerable, because we can list all the words over an alphabet V, and add the current word to the language if it is in L. The same algorithm can be used for the language $\bar{L}$, we can list the words again, and we add them to the language if they are not in the language L.

QED.

The following theorem shows the connection between the context-sensitive and recursive languages.

Theorem 32. *Each context-sensitive language is recursive, but there are recursive languages which are not context-sensitive.*

Proof. The first part of the theorem states that the word problem can be solved for each context-sensitive language. This was shown already in Section 5.3.2.

For the second part, let us create the list of each possible context-sensitive generative grammar $G_i = (N_i, T_i, S_i, P_i)$, which generates numbers. (The set of the terminal letters of each grammar G_i are digits, so $T_i = \{0,1,2,3,4,5,6,7,8,9\}$ for each G_i.) Now, we define language L, which contains the sequential numbers of grammars whose generated language does not contain its own sequential number (position in the list): $L = \{i| \ i \notin L(G_i)\}$.

We can create a list of all context-sensitive generative grammars which generates numbers, and we can decide whether or not a context-sensitive grammar generates its position in the list, so language L is recursive.

Now, suppose to the contrary that language L is context-sensitive. In this case, there is a context-sensitive grammar G_k, such that $L(G_k) = L$. Then, by the definition of L, if $k \in L(G_k)$, then $k \notin L$ is a contradiction, and if $k \notin L(G_k)$, then $k \in L$ is also a contradiction. The only possible solution is that language L is not context-sensitive.

QED.

The next theorem shows that there are languages which are not in the class of recursively enumerable languages, so the recursively enumerable language class does not contain all possible languages. The concept of the proof is similar to the previous proof.

Theorem 33. *There exists a language L, which is not recursively enumerable.*

Proof. Let us create the list of each generative grammar $G_i = (N_i, T_i, S_i, P_i)$ which generates numbers. (The set of the terminals of each grammar G_i are numbers: $T_i = \{0,1,2,3,4,5,6,7,8,9\}$ for each grammar G_i). We have to note that it is easy to create an ordered list of all possible generative grammars which generates numbers. Now, we define language L, which contains the numbers of the grammars which does not generate the number of its position in the list: $L = \{i| \ i \notin L(G_i)\}$.

Now, suppose to the contrary that the language L is recursively enumerable. In this case, there is a generative grammar G_k, such that $L(G_k) = L$. Then, by the definition of L, if $k \in L(G_k)$, then $k \notin L$ is a contradiction, and if $k \notin L(G_k)$, then $k \in L$ is also a contradiction. The only possible solution is that language L is not recursively enumerable.

QED.

We have already shown that the class of context-sensitive languages is a proper subset of the class of recursive languages. It has also been proven that the class of all languages is a proper superset of the class of recursively enumerable languages. Finally, we note that the complementer language of language L defined in the proof of Theorem 33. [83] is recursively enumerable, but not recursive. We can summarize our results in the following formula:

$$CS \subsetneq R \subsetneq RE \subsetneq AL$$

where CS stands for context-sensitive languages, R stands for recursive languages, RE stands for recursively enumerable languages, and AL stands for all languages.

1.1. 6.1.1. Closure Properties

Theorem 34. *The class of recursive languages is closed under complement operation.*

Proof. Theorem 31. [82] states that language L is recursive, if and only if L and $\bar{L}$ are both recursively enumerable. However, we can apply the same theorem for the language $\bar{L}$, and what we receive as a result is that $\bar{L}$ is recursive, if $\bar{L}$ and L are both recursively enumerable, which is the same condition, so if L is recursive, then $\bar{L}$ is recursive, as well.

QED.

Theorem 35. *The class of recursively enumerable languages is not closed under complement operation.*

Proof. The proof of this theorem is easy, we need only one example, where the language itself is recursively enumerable, and the complement language is not recursively enumerable. In the proof of the Theorem 33. [83] we have defined a language L which is not recursively enumerable. The complement language $\overline{L}$ is recursively enumerable. Here we have a recursively enumerable language $\overline{L}$, whose complement is not recursively enumerable.

QED.

Theorem 36. *The class of recursively enumerable languages is closed under regular operations.*

Proof. We give a constructive proof here. Let the languages L_1 and L_2 be recursively enumerable. Let the grammar $G1 = (N_1, T, S_1, P_1)$ such that $L(G_1) = L_1$, and let the grammar $G_2 = (N_2, T, S_2, P_2)$ such that $L(G_2) = L_2$. Without loss of generality we can suppose that $N_1 \cap N_2 = \emptyset$, and the terminal symbols appear only in rules having the form $A \rightarrow a$, where $A \in N, a \in T$. We define generative grammars G_{Un}, and G_{Co}, such that $L(G_{Un}) = L_1 \cup L_2$, and $L(G_{Co}) = L1 \cdot L_2$.

1. Union

   Let S be a new start symbol, such that $S \cap N_1 = S \cap N_2 = S \cap T = \emptyset$, and let

   $$G_{Un} = (N_1 \cup N_2 \cup \{S\}, T, S, P_1 \cup P_2 \cup \{S \rightarrow S_1, S \rightarrow S_2\}).$$

2. Concatenation

   Let S be a new start symbol, such that $S \cap N_1 = S \cap N_2 = S \cap T = \emptyset$, and let

   $$G_{Co} = (N_1 \cup N_2 \cup \{S\}, T, S, P_1 \cup P_2 \cup \{S \rightarrow S_1S_2\}).$$

3. Kleene star

   In order to create a grammar generating the language $L(G_{Kl}) = L_1^*$ we use two grammars. Let the grammar $G_1 = (N_1, T, S_1, P_1)$, and let the grammar $G_2 = (N_2, T, S_2, P_2)$ such that $L(G_1) = L(G_2) = L_1 \setminus \{\lambda\}$, and $N_1 \cap N_2 = \emptyset$. Without loss of generality we can suppose that the terminal symbols appear only in rules having the form $A \rightarrow a$, where $A \in N, a \in T$. For the grammar G_{Kl} we again use a new start symbol S, where $S \cap N_1 = S \cap N_2 = S \cap T = \emptyset$. Then, let

   $$G_{Kl} = (N_1 \cup N_2 \cup \{S\}, T, S, P_1 \cup P_2 \cup \{S \rightarrow \lambda, S \rightarrow S_1, S \rightarrow S_1S_2S\}).$$

QED.

Theorem 37. *The class of recursively enumerable languages is closed under intersection.*

Proof. By applying the definition of the recursively enumerable language, we can create a list of the elements of the recursively enumerable language L_1 without repetitions, and yet another list, which contains the elements from the recursively enumerable language L_2 without repetitions. Then, we can create a list, which contains one element from the list of the language L_1, and then one element of the list of the language L_2 alternately. We move an element from this combined list into the list of the $L_1 \cap L_2$, if the element appears twice.

QED.

Finally, we have to note that recursive languages are also closed under regular operations and intersection.

1.2. 6.1.2. Normal Forms

Definition *The grammar $G = (N, T, S, P)$ is in Révész normal form, if all of its production rules has the following forms:*

1. $S \rightarrow \lambda$,

2. $A \rightarrow a$,

3. $A \rightarrow B$,

4. $A \rightarrow BC$,

5. $AB \rightarrow AC$,

6. $AB \rightarrow CB$, *or*

7. $AB \rightarrow B$,

where $A, B, C \in N$ and $a \in T$.

This normal form for unrestricted grammars was introduced by György Révész. Compared to the Kuroda normal form, the differences are the following:

- It allows the rule $S \rightarrow \lambda$ for grammars generating the empty word,
- instead of the rule $AB \rightarrow CD$ it allows two rules, namely $AB \rightarrow AC$ and $AB \rightarrow CB$,
- the only ,,really plus" rule, which makes the difference between the monotone and unrestricted grammars is the last production rule: $AB \rightarrow B$.

As you can see, there is not a huge difference between the unrestricted grammars and the context-sensitive grammars. For generative grammars generating context-sensitive languages, only one production rule form is enough to be able to generate any recursively enumerable language.

Even more surprising results were proven by Viliam Geffert. His results put limitations not just to the form of the production rules, but also to the number of the nonterminal symbols. We introduce his results as theorems, without proofs.

Theorem 38. *For each recursively enumerable language L we can give an unrestricted generative grammar G* = (N, T, S, H) *such that*

- *grammar G generates the language L,* $(L(G) = L)$,
- *G has exactly 5 nonterminal symbols,* $(N = \{S, A, B, C, D\})$, *and*
- *each rule has a form:*
  - $S \rightarrow p$ *where S is the start symbol, and p is a nonempty word,* $(p \in (N \cup T)^+)$,
  - $AB \rightarrow \lambda$, *or*
  - $CD \rightarrow \lambda$.

Theorem 39. *For each recursively enumerable language L we can give an unrestricted generative grammar G* = (N, T, S, H) *such that*

- *grammar G generates the language L,* $(L(G) = L)$,
- *G has exactly 4 nonterminal symbols,* $(N = \{S, A, B, C\})$, *and*
- *each rule has a form:*
  - $S \rightarrow p$ *where S is the start symbol, and p is a nonempty word,* $(p \in (N \cup T)^+)$,
  - $AB \rightarrow \lambda$, *or*
  - $CC \rightarrow \lambda$.

Theorem 40. *For each recursively enumerable language L we can give an unrestricted generative grammar G* = (N, T, S, H) *such that*

- *grammar G generates the language L,* $(L(G) = L)$,

- *G has exactly 3 nonterminal symbols, ($N = \{S,A,B\}$), and*
- *each rule has a form:*
    - *$S \rightarrow p$ where S is the start symbol, and p is a nonempty word, ($p \in (N \cup T)^+$),*
    - *$AA \rightarrow \lambda$, or*
    - *$BBB \rightarrow \lambda$.*

Theorem 41. *For each recursively enumerable language L we can give an unrestricted generative grammar $G = (N, T, S, H)$ such that*

- *grammar G generates the language L, ($L(G) = L$),*
- *G has exactly 3 nonterminal symbols, ($N = \{S, A, B\}$), and*
- *each rule has a form:*
    - *$S \rightarrow p$ where S is the start symbol, and p is a nonempty word, ($p \in (N \cup T)^+$), or*
    - *$ABBBA \rightarrow \lambda$.*

Theorem 42. *For each recursively enumerable language L we can give an unrestricted generative grammar $G = (N, T, S, H)$ such that*

- *grammar G generates the language L, ($L(G) = L$),*
- *G has exactly 4 nonterminal symbols, ($N = \{S, A, B, C\}$), and*
- *each rule has a form:*
    - *$S \rightarrow p$ where S is the start symbol, and p is a nonempty word, ($p \in (N \cup T)^+$), or*
    - *$ABC \rightarrow \lambda$.*

2. 6.2. Turing Machine, the Universal Language Acceptor

Turing machines play a fundamental role in the algorithms and computational theory. The concept of Turing machine was invented by Alan Turing in 1937. This simple hypothetical device is able to compute all the functions which are algorithmically computable. Before we deal with the Turing machine as a universal tool for describing algorithms, we introduce the Turing machine as a universal language definition device.

The basic concept is that the Turing machine manipulates a string on a two-way infinite tape according to transition rules, and decides whether or not the input string belongs to a language accepted by the Turing machine. The tape contains an infinite number of cells, and each cell contains one letter. At the beginning, the tape contains the input string, and the rest of the cells contain a special tape symbol called a blank symbol. There is a head, which can read and write the content of the current cell of the tape, and can move both to the left and to the right. At the beginning, the head is over the first letter of the input string. The Turing machine also has its own inner state, which can be changed in each step. At the beginning, the inner state of the Turing machine is the initial state. The transition rules are the "program" of the Turing machine.

In each step the machine reads the letter contained by the current cell of the tape, and also reads its own inner state, then writes a letter into the current cell, changes its inner state and moves the head to the left or to the right, or stays in the same position. Sometimes, it does not change its inner state, and sometimes it does not change the content of the current cell. The operations of the Turing machine are based on the transition rules.

Let us see the formal definition and the detailed description.

Definition 33. *The (nondeterministic) Turing machine (TM) is the following 7-tuple:*

$TM = (Q, T, V, q_0, \#, \delta, F)$

where

- *Q is the finite nonempty set of the states,*
- *T is the set of the input letters, (finite nonempty alphabet), $T \subseteq V$,*
- *V is the set of the tape symbols, (finite nonempty alphabet),*
- *q_0 is the initial state, $q_0 \in Q$,*
- *# is the blank symbol, $\# \in V$,*
- *δ is the transition function having a form $Q \times V \rightarrow 2^{Q\times V\times \{Left, Right, Stay\}}$, and*
- *F is the set of the final states, $F \subseteq Q$.*

We have a description of the parts of our device, and now we have to describe its operation. In each step, the Turing machine has its configuration (u,q,av), where $q \in Q$ is the current state, $a \in V$ is the letter contained by the current cell, and $u, v \in V^*$ are the words before and after the current letter, respectively. The first letter of the word u and the last letter of the word v cannot be the blank symbol, and the word uav is the "important" part of the tape, the rest of the tape contains only blank symbols. At the beginning, the Turing machine has its initial configuration: (λ, q_0, av), where av is the whole input word. In each step, the Turing machine changes its configuration according to the transition function. There are three possibilities, depending on the movement part of the applied rule.

1. The simplest case is when the applied transition rule has a form $(q_2, a_2, Stay) \in \delta(q_1, a_1)$. In this case, we just change the state and the symbol contained by the current cell of the tape according to the current rule. Formally, we say the TM can change its configuration from (u, q_1, a_1v) to (u, q_2, a_2v) in one step, if it has a transition rule $(q_2, a_2, Stay) \in \delta(q_1, a_1)$, where $q_1, q_2 \in Q$, $a_1, a_2 \in V$, and $u, v \in V^*$.

   Mark: $(u, q_1, a_1v) \vdash_{TM} (u, q_2, a_2v)$.

2. The next possibility is when the applied transition rule has a form $(q_2, a_2, Right) \in \delta(q_1, a_1)$. In this case, we change the state and the symbol contained by the current cell of the tape according to the current rule, and move the head to the right. Formally, we say the TM can change its configuration from (u, q_1, a_1v) to (ua_2, q_2, v) in one step, if it has a transition rule $(q_2, a_2, Right) \in \delta(q_1, a_1)$, where $q_1, q_2 \in Q$, $a_1, a_2 \in V$, and $u, v \in V^*$.

   It is denoted by $(u, q_1, a_1v) \vdash_{TM} (ua_2, q_2, v)$.

   Here, we have a special case, namely, if $a_2 = \#$ and $u = \lambda$, then $(u, q_1, a_1v) \vdash_{TM} (\lambda, q_2, v)$.

3. The last possibility is when the applied transition rule has a form $(q_2, a_2, Left) \in \delta(q_1, a_1)$. In this case, we change the state and the symbol contained by the current cell of the tape according to the current rule, and move the head to the left. To formalize this case, we have to write the word u in a form $u = wb$, where b is the last letter of the word u. We say that the TM can change its configuration from (wb, q_1, a_1v) to (w, q_2, ba_2v) in one step, if it has a transition rule $(q_2, a_2, Left) \in \delta(q_1, a1)$, where $q_1, q_2 \in Q$, $a_1, a_2, b \in V$, and $w, v \in V^*$.

   It is denoted by $(wb, q_1, a_1v) \vdash_{TM} (w, q_2, ba_2v)$.

   Here, we also have a special case, namely, if $a_2 = \#$ and $v = \lambda$, then $(wb, q_1, a_1v) \vdash_{TM} (w, q_2, b)$.

Now, let X and Y be configurations of the same Turing machine. Then, we say that the Turing machine can change its configuration from X to Y in finite number of steps, if $X = Y$, or there are configurations $C_0, C_1, ..., C_n$ such that $C_0 = X$, $C_n = Y$, and $C_i \vdash_{TM} C_{i+1}$ holds for each integer $0 \leq i < n$.

It is denoted by $X \vdash^*_{TM} Y$.

A configuration is called a final configuration, if the Turing machine is in a final state. Now, we can define the language accepted by the Turing machine. The input word is accepted, if the Turing machine can change its configuration from the initial configuration to a final configuration in finite number of steps.

$L(TM) = \{p \mid p \in T^*, (\lambda, q_0, p) \vdash^*_{TM} (u, q_f, av), q_f \in F, a \in V, u, v \in V^*\}.$

Example 57. *Let the language L be the language of the palindromes over the alphabet* $\{a, b\}$*. (Palindromes are words that reads the same backward or forward.) This example shows the description of a Turing machine TM, which accepts the language L.*

```
     $TM = (\{q_0,q_1,q_2,q_3,q_4,q_5,q_f\}, \{a,b\}, \{a,b,\#\}, q_0,\#,\delta, \{q_f\})$
$\delta(q_0,\#) = \{(q_f,\#,Stay)\},$
$\delta(q_0,a) = \{(q_1,\#,Right)\},$
$\delta(q_0,b) = \{(q_2,\#,Right\},$
$\delta(q_1,a) = \{(q_1,a,Right)\},$
$\delta(q_1,b) = \{(q_1,b,Right)\},$
$\delta(q_1,\#) = \{(q_3,\#,Left)\},$
$\delta(q_3,a) = \{(q_5,\#,Left)\},$
$\delta(q_3,\#) = \{(q_f,\#,Stay)\},$
$\delta(q_2,a) = \{(q_2,a,Right)\},$
$\delta(q_2,b) = \{(q_2,b,Right)\},$
$\delta(q_2,\#) = \{(q_4,\#,Left)\},$
$\delta(q_4,\#) = \{(q_f,\#,Stay)\},$
$\delta(q_4,b) = \{(q_5,\#,Left)\},$
$\delta(q_5,a) = \{(q_5,a,Left)\},$
$\delta(q_5,b) = \{(q_5,b,Left)\},$
$\delta(q_5,\#) = \{(q_0,\#,Right)\}.$
```

Exercise 79. *Create a Turing machine, which accepts words over the alphabet* $\{a,b\}$ *containing the same number of a's and b's.*

Exercise 80. *Create a Turing machine, which accepts words over the alphabet* $\{a,b\}$ *if they are a repetition of another word.*

Exercise 81. *Create a Turing machine, which accepts binary numbers greater than 20.*

Our final note is that, although it is a simple construction, Turing machines can accept any language from the recursively enumerable language class. This statement is formulated in the following theorem.

Theorem 43. *A language L is recursively enumerable, if and only if there exists a Turing machine TM such that* $L = L(TM)$.

2.1. 6.2.1. Equivalent Definitions

There are several equivalent definitions for the Turing machine. In this subsection we are going to introduce some of them. Our first definition is the deterministic Turing machine, which has the same language definition power as the nondeterministic Turing machine. A Turing machine is called deterministic, if from each configuration it can reach at most one other configuration in one step. The formal definition is the following:

Definition 34 *The Turing machine* $TM_d = (Q, T, V, q_0, \#, \delta, F)$ *is deterministic, if the transition function* $\delta(q,a)$ *has at most one element for each pair* (q,a)*, where* $q \in Q$ *and* $a \in V$.

In other words, the Turing machine $TM_d = (Q, T, V, q_0, \#, \delta, F)$ is deterministic, if the form of the transition function δ is $Q \times V \to Q \times V \times \{Left, Right, Stay\}$.

Theorem 44. *For each Turing machine TM there exists a deterministic Turing machine* TM_d *such that* $L(TM) = L(TM_d)$.

Now, we are going to introduce the multitape Turing machine, which looks like a more powerful tool compared to the original Turing machine, but in reality it has the same language definition power. In this case, we have more than one tape, and we work on each tape in each step. At the beginning, the input word is written on the first tape, and the other tapes are empty. (Contains blank symbols in each position.) The multitape Turing machine is in initial state, and the head is over the first letter of the first tape. In each step the multitape Turing machine reads its own state and the symbols from the cells of each tape, then changes its state; it writes symbols into the current cell of each tape and moves the head to the left or to the right, or stays in a same position over each tape separately. Observing the formal definition, its only difference compared to the deterministic Turing machine is that it has more tapes, and as a result, it has a different transition function.

Definition 35. *The multitape Turing machine is the following octuple* $TM_m = (k, Q, T, V, q_0, \#, \delta, F)$, *where* Q, T, V, q_0, # *and* F *are the same as before, the integer* $k \geq 0$ *is the number of the tapes, and the form of the transition function* δ *is* $Q \times V^k \rightarrow Q \times (V \times \{Left, Right, Stay\})^k$.

As we have noted before, multitape Turing machines accept recursively enumerable languages as well.

Theorem 45. *For each multitape Turing machine* TM_m *there exists (a one-tape) Turing machine TM such that* $L(TM) = L(TM_m)$.

The reason for using the multitape Turing machine or the deterministic Turing machine instead of the original Turing machine is very simple. Sometimes it is more comfortable to use these alternative Turing machines for calculating or proving theorems.

For the same purpose, sometimes we use a Turing machine which has one tape, and this tape is infinite in one direction, and the other direction is "blocked". This version has a special tape symbol in the first cell, and when the Turing machine reads this symbol, the head moves to the right and does not change this special symbol. There is yet another possibility, when the Turing machine must not stay in the same position, in each step the head must move to the right or to the left. Sometimes we use only one final state, and sometimes we use a rejecting state as well, but all of these versions are equivalent to each other. Each of the Turing machine described above accepts recursively enumerable languages, and each recursively enumerable language can be accepted by each type of the above mention Turing machines.

3. 6.3. Turing Machine, the Universal Computing Device

We have discussed earlier that the Turing machine is not just a language definition tool. The original reason for introducing the Turing machine was to simulate mathematical algorithms which can calculate complicated functions. Later, it has been recognized that all algorithmically computable functions can be calculated by the Turing machine, as well. The statement which claims that a function is algorithmically computable if and only if it can be computed by the Turing machine is called Church's thesis. Church's thesis is not a theorem, it cannot be proven, because "algorithmically computable" is not a well defined expression in the thesis. In spite of the fact that Church's thesis is not a proven theorem, the thesis is accepted among scientists. The most important consequence of the thesis is that even the latest, and the strongest computer with a highly improved computer program can only compute the same things as a very simple Turing machine. Therefore we can use the Turing machine to show if something can be computed or not; and this is why the Turing machine plays a fundamental role in the algorithms and computational theory. Although this book focuses on formal languages, we cannot conclude this topic without illustrating the basics of the application of the Turing machine as a computing device.

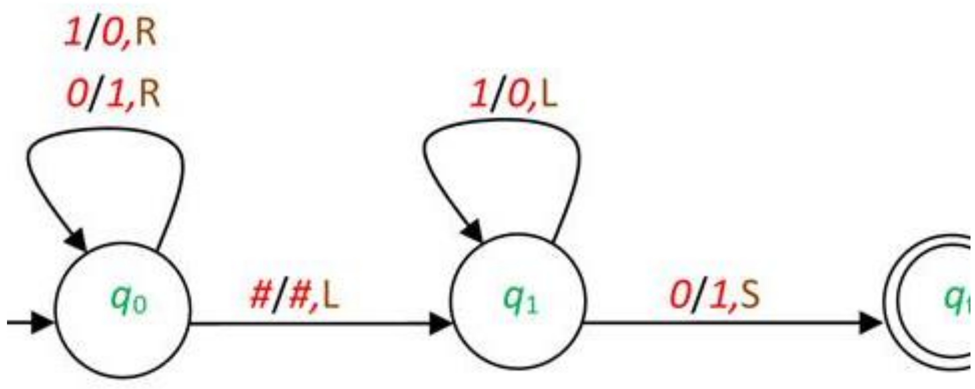
When we use the Turing machine to compute/calculate a function, we use it the same way as before. At the beginning, the input word is on the tape, and when the Turing machine reaches a final configuration (u,qf,av), the result of the computation/calculation is the word uav, which is the significant part of the tape. In the Example 58. [89] we show a Turing machine, which computes the two's complement of the input binary number.

Example 58. $TM = (\{q_0, q_1, q_f\}, \{0,1\}, \{0,1,\#\}, q_0, \#, \delta, \{q_f\})$

```
  δ(q0,1) = {(q0,0,Right)},
  δ(q0,0) = {(q0,1,Right)},
  δ(q0,#) = {(q1,#,Left},
  δ(q1,1) = {(q1,0,Left)},
  δ(q1,0) = {(qf,1,Stay)}.
```

The Figure 6.1. shows the graphical notation of this Turing machine.

6.1. ábra - The graphical notation for the Example 58. [89]



Exercise 82. *Create a Turing machine, which changes the first five 0 to 1 in an input binary number.*

Exercise 83. *Create a Turing machine, which adds five to the input binary number.*

Exercise 84. *Create a Turing machine, which changes the order of two input binary numbers, separated by space.*

Each equivalent definition of the Turing machine which was given at the end of the previous section can work here, so we can use the deterministic Turing machine, the multitape Turing machine, and the single tape Turing machine, which is infinite only in one direction, or we can extend the Turing machine with rejecting states, our choice will not influence the calculating power.

As we have already demonstrated, there are languages which are not recursively enumerable. This means that these languages cannot be generated by a generative grammar, and cannot be accepted by Turing machines. Also, there are functions, which cannot be computed by Turing machines. There is a well known example: the halting problem. There is a computer program given with an input, decide whether the program stops running after a while, or goes into an infinite loop. The same problem with Turing machine appears to be the following: given a description of a Turing machine and the input word, decide whether the Turing machine stops running or keeps on running forever. Let us denote the description of a Turing machine with d_{TM} and the input word with w. The problem is to create a Turing machine which decides about each input word $d_{TM}\#w$ whether or not the Turing machine TM goes into an infinite loop with the input word w. It has been shown that this problem cannot be decided by a Turing machine, so there is no universal algorithm to decide if a given computer program goes into an infinite loop or not. The equivalent problem is to create a Turing machine which accepts an input word $d_{TM}\#w$, if the Turing machine TM stops with the input word w. As one can see, computing/calculating a function or accepting a language is not so distant from each other. We also have to point out that the halting problem cannot be solved, because we suppose that the Turing machine has an infinite tape. The problem can be solved in a computer with a finite memory, however, the algorithm is not efficient in practice. We have already shown that there are functions which cannot be algorithmically computed, consequently there are problems which cannot be solved. However, there are many problems, which can be solved, and we would like to know the complexity of the algorithms computing the solutions. For this reason, scientists have introduced the time complexity of the algorithms. The time complexity of an algorithm is not a constant number. For a longer input the Turing machine needs longer time to compute, so the time complexity of an algorithm is a function for each algorithm, and the parameter of the function is the length of the input word w. This length is commonly denoted by n, so $n = |w|$, and the time complexity of the Turing machine TM is denoted by $T(n)$. We can investigate the space complexity of an algorithm as well. Let us denote by $S(n)$ the function which shows how many cells we use on the tape of the Turing machine TM for an input word of length n. Of course, the time complexity and the space complexity are not independent from each other. If the time is limited, we have a limitation on the number of steps, so we can go to a limited distance from the initial position on the tape, which means that the space is also limited. The most important time complexity classes are the followings:

- constant time, when the calculating time is fixed, does not depend on the length of the input, denoted by $O(1)$,
- logarithmic time, when the calculating time is not more than a logarithmic function of the length of the input word, denoted by $O(\log n)$,

- linear time, when the calculating time is not more than a linear function of the length of the input word, denoted by $O(n)$,
- polynomial time, when the calculating time is not more than a polynomial function of the length of the input word, denoted by $O(n^k)$,
- exponential time, when the calculating time is not more than an exponential function of the length of the input word, denoted by $O(2^{n_k})$.

Evidently, we can have the same complexity classes for the space used by a Turing machine, and most of our computer programs are deterministic, so these complexity classes can be similarly defined for deterministic Turing machines, as well. We know that the nondeterministic and the deterministic Turing machines have the same computational power, but we do not know if the problems which can be solved with nondeterministic Turing machines in polynomial time, and the problems which can be solved with deterministic Turing machines in polynomial time are the same or not. This is a major problem in algorithm theory, and it is called **P** = **NP?** problem.

Our last section is about the linear bounded automaton, which is a Turing machine with linear space complexity, and has a special role in the study of context-sensitive languages.

4. 6.4. Linear Bounded Automata

In this section, we present a special, bounded version of the Turing machines, by which the class of context-sensitive languages can be characterized - as we already mentioned in Subsection. This version of the Turing machine can work only on the part of the tape where the input is/was. These automata are called linear bounded automata (LBA).

Definition 36. *Let LBA* = $(Q, T, V, q_0, \sharp, \delta, F)$ *be a Turing machine, where*

$$\delta : Q \times (V \setminus \{\sharp\}) \to 2^{Q\times V\times \{Left,\ Right,\ Stay\}}$$

and

$$\delta : Q \times \{\sharp\} \to 2^{Q\times \{\sharp\}\times \{Left,\ Right,\ Stay\}}.$$

Then LBA is a (nondeterministic) linear bounded automaton.

One can observe that ♯ signs cannot be rewritten to any other symbol, therefore, the automaton can store some results of its subcomputations only in the space provided by the input, i.e., in fact, the length of the input can be used during the computation, only.

Example 59. *Give an LBA that accepts the language* $\{a^ib^ic^i |\ i \in \mathbb{N}\}$.

Solution:

Idea:

- *The automaton rewrites the first a to A, and changes its state, looks for the first b.*
- *The automaton rewrites the first b to B, and changes its state, looks for the first c.*
- *The automaton rewrites the first c to C, and changes its state, looks (backward) for the first a.*
- *The capital letters A,B,C are read without changing them.*
- *The above movements are repeated.*
- *If finally only capital letters remain between the border ♯ signs, then the automaton accepts (the input).*

Formally, let

$$LBA = (\{q_0, q_1, q_2, q_3, q_4, q_f\}, \{a,b,c\}, \{a,b,c,A,B,C,\sharp\}, q_0, \sharp, \delta, \{q_f\})$$

be a deterministic LBA, where δ consists of the next transitions:

1. δ (q_0, ♯) = (q_f, ♯, *Right*) – *the empty word is accepted by LBA.*
2. δ (q_0, *a*) = (q_1, *A*, *Right*) – *the first (leftmost) a is rewritten to A and LBA changes its state.*
3. δ (q_0, *B*) = (q_0, *B*, *Left*) – *the capital letters B and C are skipped in state* q_0,
4. δ (q_0, *C*) = (q_0, *C*, *Left*) – *by moving the head to the left.*
5. δ (q_1, *a*) = (q_1, *a*, *Right*) – *letter a is skipped in state* q_1*to the right.*
6. δ (q_1, *B*) = (q_1, *B*, *Right*) – *capital B is also skipped.*
7. δ (q_1, *b*) = (q_2, *B*, *Right*) – *the leftmost b is rewritten by B and the state becomes* q_2.
8. δ (q_2, *b*) = (q_2, *b*, *Right*) – *letter b is skipped in state* q_2*to the right.*
9. δ (q_2, *C*) = (q_2, *C*, *Right*) – *capital C is also skipped in this state.*
10. δ (q_2, *c*) = (q_3, *C*, *Left*) – *the leftmost c is rewritten by C and LBA changes its state to* q_3.
11. δ (q_3, *a*) = (q_3, *a*, *Left*) – *letters a,b are skipped in state* q_3
12. δ (q_3, *b*) = (q_3, *b*, *Left*) – *by moving the head of the automaton to the left.*
13. δ (q_3, *C*) = (q_3, *C*, *Left*) – *capital letters C,B are skipped in state* q_3
14. δ (q_3, *B*) = (q_3, *B*, *Left*) – *by moving the head of the automaton to the left.*
15. δ (q_0, *A*) = (q_3, *A*, *Right*) – *the head is positioned after the last A and the state is changed to* q_0.
16. δ (q_4, *B*) = (q_3, *B*, *Right*) – *if there is a B after the last A the state is changed to* q_4.
17. δ (q_4, *B*) = (q_4, *B*, *Right*) – *in state* q_4*capital letters B and C are skipped*
18. δ (q_4, *C*) = (q_4, *C*, *Right*) – *by moving the head to the right.*
19. δ (q_f, ♯) = (q_4, ♯, *Left*) – *if in* q_4*there were only capital letters on the tape, LBA accepts.*

The work of the automaton can be described as follows: it is clear by transition 1, that λ is accepted.

Otherwise the head reads the first letter of the input: if the input starts with an a, then it is replaced by A and q_1 *is the new state. If the first letter of the input is not a, then LBA gets stuck, i.e., it is halting without acceptance, since there are no defined transitions in this state for the other input letters.*

In state q_1 *LBA looks for the first b by moving to the right (skipping every a and B, if any; and halting without acceptance by finding other letters before the first b). When a b is found it is rewritten to a B and the automaton changes its state to* q_2.

In q_2 *the first c is searched, the head can move through on b's and C's (but not on other letters) to the right. When it finds a c it rewrites by a C and changes the state to q3 and starts to move back to the left.*

In q_3 *the head can go through on letters a,B,b,C to the left and when finds an A it steps to the right and the state becomes* q_0. *In* q_0*if there is an a under the head, then it is rewritten by an A and the whole loop starts again.*

If in q_0 *a letter B is found (that could happen when every a is rewritten by an A already), LBA changes its state to* q_4. *In this state by stepping to the right LBA checks if every b and c is already rewritten and if so, i.e., their number is the same as the number of a's and their order was correct (the input is in the language* $a^*b^*c^*$*), then LBA reaches the marker ♯ sign after the input and accepts.*

When at any stage some other letter is being read than it is expected (by the previous description), then LBA halts without accepting the input.

Thus, the accepted language is exactly $\{a^i b^i c^i \mid i \in \mathbb{N}\}$.

To establish a connection between the classes of context-sensitive languages and linear bounded automata we present the following theorem.

Theorem 46. *The class of languages accepted by linear bounded automata and the class of context-sensitive languages coincide.*

Proof. We do not give a formal proof here, instead we present the idea of a proof. A context-sensitive language can be defined by a monotone grammar. In a monotone grammar (apart from the derivation of the empty word, if it is in the language), the length of the sentential form cannot be decreased by any step of the derivation. Consequently, starting from the derived word, and applying the rules in a way which is an inverse, the length is monotonously decreasing till we obtain the start symbol. In this way every context-sensitive language can be accepted by an LBA.

The other way around, the work of an LBA can be described by a grammar, working in inverse way of the generation (starting from a word the start symbol is the target). These grammars are similar to the previously used monotone grammars, and thus, if an LBA is accepting a language L, then L is context-sensitive.

QED.

Actually, there are other models of LBA, in which the workspace (the maximal tape-length during the computation) is limited by $c_1 \cdot | w | + c_0$, where w is the input and $c_0, c_1 \in \mathbb{R}$ constants. The accepting power of these models are the same as of those that have been presented.

However, the deterministic model is more interesting, since it is related to a long-standing and still open question.

It is known that every context-sensitive language can be accepted by deterministic Turing machines, using at most $c_2 \cdot | w |^2 + c_1 \cdot | w | + c_0$ space during the computations, where c_2, c_1, c_0 are constants. However, it is neither proven nor disproved that deterministic linear bounded automata (using $c_1 \cdot | w | + c_2$ space) can recognize every context-sensitive language. This is still a *famous* open problem.

Exercise 85. *Give a linear bounded automaton that accepts the language*

$$\{a^i b^j a^i b^j \mid i, j \in \mathbb{N}\}.$$

Exercise 86. *Give a linear bounded automaton that accepts the language*

$$\{a^{2^i} \mid i \in \mathbb{N}\},$$

i.e., the language of powers of two in unary coding.

Exercise 87. *Give a linear bounded automaton that accepts the set of primitive words over the alphabet* $\{a,b\}$. *(A word w is primitive if it is not of the form u^n for any word $u \neq w$.)*

7. fejezet - Literature

For further reading the following books in related topics are recommended:

1. Bel-Enguix, Gemma, Jiménez-López, Maria Dolores and Martín-Víde, Carlos (eds.): *Recent Developments in Formal Languages and Applications*, Springer, Berlin, 2008.

2. Ésik, Zoltán, Martín-Vide, Carlos and Mitrana, Victor (eds.): *Recent Advances in Formal Languages and Applications*, Springer 2006.

3. Harrison, Michael A.: *Introduction to Formal Language Theory*, Addison-Wesley, 1978.

4. Herendi, Tamás and Nagy, Benedek: *Parallel Approach of Algorithms*, Typotex, Budapest, 2014.

5. Hopcroft, John E. and Ullmann, Jeffrey D.: *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, Massachusetts, Menlo Park, California, London, Amsterdam, Don Mills, Ontario, Sidney, 1979.

6. Linz, Peter: *An introduction to formal languages and automata*, Fourth edition, Jones and Bartlett Publishers 2006.

7. Martín-Vide, Carlos, Mitrana, Victor and Păun, Gheorghe (eds.) *Formal Languages and Applications*, Springer, Berlin, 2004.

8. Révész, György E.: *Introduction to Formal Languages*, McGraw-Hill, New York, St Louis, San Francisco, Auckland, Bogota, Hamburg, Johannesburg, London, Madrid, Mexico, Montreal, New Delhi, Panama, Paris, Sao Paulo, Singapore, Sydney, Tokyo, Toronto, 1983.

9. Rozenberg, Grzegorz and Salomaa, Arto (eds.): *Handbook of Formal Languages*, 3 volumes, Springer, Heidelberg, 1997.

10. Salomaa, Arto: *Formal Languages*, Academic Press, New York, London, 1973.