VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FIT

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Theoretical Computer Science

TCS

Textbook

Milan Češka, Tomáš Vojnar

January 2, 2009

Contents

Chapter 1

Introduction

0:20

This textbook is intended for one semester course in theoretical computer science at the master level. The text covers the theory of regular and context-free languages and essential parts of the theory of computability. A basic introduction to the complexity theory is also included in this textbook.

The theory of formal languages, computability and complexity represent classical but very important part of theoretical computer science. Their origins are bind with such names of famous scientists as N. Chomsky, A. Turing or A. Church. Motivations to find a mathematical model for natural languages description on one side and to define models for effective computation, i.e. computation which can be done by manipulating symbols on the other side have led to the fascinating theories. They represent not only priceless and inspiring heritage always used in applications, but in fact they are philosophical and theoretical basements for contemporary computers and information technology.

The including theories are highly relevant to practice. They provide conceptual tools that practitioners used in computer engineering. Designing new programming language, writing a compiler, finding a computer solution of new problems, evaluate the time or space complexity of designed or referenced algorithm, and verify implemented program are typical activities for which knowledge of the theory of formal languages, computability and complexity are needed for professionals.

The subject is quite mathematical in flavor and certain degree of previous mathematical experience is essential for students. Specially the should be familiar with elementary discrete mathematics notions of sets, function, relation, product, partial order, equivalence relation, proposition and predicate logic, graph and tree. Students should also have a repertoire of basic proof techniques at their disposal including the proof by contradiction and the proof by mathematical induction.

⏲	Needed study time.	?	Question or example to be solved.
🧭	Goal	💻	Computer exercise or example
DEF	Definition	x+y	Example
!	Important part	📖	Reference
🔍	Part of extension	✓	Solution
⛰1		Σ	Summary
⛰2	Difficult part	☺	Tutor remark, comment
⛰3		◁	Interesting part

Table 1.1: Description of used pictograms

1.1 Description of the course

1.1.1 Learning objectives

To acquaint students with more advanced parts of the formal language theory, with basics of the theory of computability, and with basic terms of the complexity theory.

1.1.2 Description

An overview of the applications of the formal language theory in modern computer science and engineering (compilers, system modeling and analysis, linguistics, etc.), the modeling and decision power of formalisms, regular languages and their properties, minimization of finite-state automata, context-free languages and their properties, Turing machines, properties of recursively enumerable and recursive languages, computable functions, undecidability, undecidable problems of the formal language theory, the complexity, Turing complexity, the P and NP classes.

1.1.3 Knowledge and skills required for the course

Basic knowledge of discrete mathematics concepts including graph theory and mathematical logic.

1.1.4 Syllabus of lectures

1. An overview of the applications of the formal language theory, the modeling and decision power of formalisms, operations over languages, grammars, Chomsky hierarchy of languages.
2. Regular languages and their properties, Kleene's theorem, Nerod's theorem, Pumping lemma.
3. Minimization of finite-state automata, the relation of indistinguishability of automata states, construction of a reduced finite-state automaton.
4. Closure properties of regular languages, regular languages as a Boolean algebra, decidable problems of regular languages.
5. Context-free languages and their properties, normal forms of context-free grammars, unambiguous and deterministic context-free languages, Pumping lemma for context-free languages.
6. Closure properties of context-free languages, closure property with respect to substitution and its consequences, decidable problems of context-free languages.
7. Turing machines (TM's), the language accepted by a TM, recursively enumerable and recursive languages and problems, TM's and functions, methods of constructing TM's.
8. Modifications of TM's, TM's with a tape infinite on both sides, with more tapes, nondeterministic TM's, automata with two push-down stacks, automata with counters.
9. TM's and type-0 languages, diagonalization, properties of recursively enumerable and recursive languages, linearly bounded automata and type-1 languages.
10. Computable functions, initial functions, primitive recursive functions, mu-recursive functions, the relation of TM's and computable functions.
11. The Church-Turing thesis, universal TM's, undecidability, the halting problem, reductions, the Post's correspondence problem.
12. Undecidable problems of the formal language theory.
13. An introduction to the computational complexity, the Turing complexity, the P and NP classes and beyond.

1.1.5 Evaluation of the course

A mid-term exam evaluation (max. 20 points), an evaluation of projects (max 20 points), and a term exam evaluation (max. 60 points).

Chapter 2

Languages, grammars, and their classification

10:00

The goal of the chapter is to understand the notion of formal language, the mathematical description of formal languages, basic algebraic operations over formal languages, grammatical specification of formal languages, and finally the classification of formal languages based on the grammars classification.

2.1 Languages

The base for introducing a concept of the language are notions an *alphabet* and a *string*.

DEF

Definition 2.1 An alphabet is a nonempty set of elements called *symbols of the alphabet*.

In some cases we can even work with infinite alphabets, but in the following applications we will use finite alphabets only.

Example 2.1 As examples we can give the *Latin alphabet* - the set of 52 symbols being the upper and lowercase Latin letters, the *Greek alphabet*, the *binary alphabet* represented by $\{0, 1\}$ resp. $\{\text{true}, \text{false}\}$ or alphabets of programming languages.

DEF

Definition 2.2 A *string* (also a *word* or a *sentence*) over given alphabet is any finite sequence of symbols of the alphabet . An empty sequence of symbols, i.e. the sequence with no symbol is called the *empty string*. The empty string will be denoted by ϵ.

We can formally define strings over an alphabet Σ in the following manner:

(1) The empty string ϵ is a string over Σ,

(2) if x is a string over Σ and $a \in \Sigma$, then xa is a string over Σ,

(3) y is a string over Σ if and only if y can be constructed using rules (1) and (2).

Example 2.2 If $A = \{a, b, +\}$ is the alphabet then

$$\epsilon \qquad a \qquad b \qquad + \qquad aa \qquad a+b \qquad +ab$$

are some strings over A. Ordering of symbols in a string is significant; strings $a + b$, $+ab$ are different. Symbol b, for example, is a string A, as $\epsilon b = b$ (the rule (2)).

!

Convention 2.1 We will use the following notation for alphabets, symbols, and strings:

- upper case Greek letters for alphabets,
- lower case Latin letters $a, b, c, d, f, \ldots$ for symbols,
- lower case Latin letters $t, u, v, w, \ldots$ for strings.

Now we introduce important operations over strings.

DEF

Definition 2.3 Let x and y be strings over Σ. The *concatenation* of strings x and y is the string xy (the string y is attached to the string x). The concatenation is obviously associative, i.e. $x(yz) = (xy)z$, but not commutative, $xy \neq yx$.

Example 2.3 If $x = ab, y = ba$, then $xy = abba, yx = baab, x\epsilon = \epsilon x = ab$, ϵ is the empty string.

DEF

Definition 2.4 Let $x = a_1a_2 \ldots a_n$ be a string over $\Sigma, a_i \in \Sigma$ for $i = 1, \ldots, n$. The *reversal* of string x is the string $x^R = a_na_{n-1} \ldots a_2a_1$; i.e. symbols of x are written in reverse order.

Example 2.4 If $x = abbc$, then $x^R = cbba$. Clearly $\epsilon^R = \epsilon$.

DEF

Definition 2.5 Let w be a string over Σ. The string z is called a *substring* of the string w, if there are strings x and y such that $w = xzy$. The string x_1 is called a *prefix* of a string w, if there is a string y_1 such that $w = x_1y_1$. Similarly, the string y_2 is called a *suffix* of a string w, if there is a string x_2 such that $w = x_2y_2$. If $y_1 \neq \epsilon$, resp. $x_2 \neq \epsilon$, then x_1 *proper prefix*, resp. y_2 is *proper suffix* of the string w.

x+y

Example 2.5 If $w = abbc$, then

$$\epsilon \qquad a \qquad ab \qquad abb \qquad abbc$$

are all prefixes of w (first four are proper one),

$$\epsilon \qquad c \qquad bc \qquad bbc \qquad abbc$$

are all suffixes w (first four are proper one) a

$$a \qquad bb \qquad abb$$

are some substrings of w.

Obviously, a prefix and a suffix are substrings of any string w, the empty string is the substring, prefix, and suffix of any string.

Definition 2.6 The *length of a string* is a nonnegative integer representing number of symbols in the string. The length of the string x is denoted by $|x|$. If $x = a_1a_2 \dots a_n$, $a_i \in \Sigma$ for $i = 1, \dots n$, then $|x| = n$. The length of the empty string equals zero, i.e. $|\epsilon| = 0$.

Convention 2.2 A string or substring which contains just k occurrences of symbol a will be denoted as a^k. E.g.

$$a^3 = aaa, \quad b^2 = bb, \quad a^0 = \epsilon.$$

Definition 2.7 Let Σ be an alphabet. By the symbol Σ^* (read sigma star) we denote the set of all strings over alphabet Σ including the empty string, by the symbol Σ^+ we denote the set of all strings over alphabet Σ excluding the empty string¨, i.e. $\Sigma^* = \Sigma^+ \cup \{\epsilon\}$. The set L, such that $L \subseteq \Sigma^*$ (eventually $L \subseteq \Sigma^+$, if $\epsilon \notin L$), is called a *language* L over alphabet Σ. Thus it means that a language is any subset of strings over the given alphabet. A string x, $x \in L$, is called the *sentence* (sometimes also the world) of the language L.

Example 2.6 Let Σ be an alphabet. The powerset 2^{Σ^*} represents all possible languages over Σ.

Let L_1 a L_2 be the languages defined over the binary alphabet, it means $L_1, L_2 \subseteq \{0,1\}^*$, in this way:

$$\begin{aligned} L_1 &= \{0^n1^n \mid n \geq 0\} \text{ i.e.} \\ L_1 &= \{\epsilon, 01, 0011, 000111, \dots\} \\ L_2 &= \left\{xx^R \mid x \in \{0,1\}^+\right\} \text{ i.e.} \\ L_2 &= \{00, 11, 0000, 0110, 1001, 1111, \dots\} \end{aligned}$$

Example 2.7 Let us consider the alphabet of the programming language Pascal. Pascal is an infinite set of strings which can be derived from its graph of syntax [6]. For example the string

```
program P; begin end.
```

is a sentence of Pascal, in contrary to the string

```
procedure S; begin a := a mod b end
```

which does not belong to the language, because it represents only some part of a possible programm (the substring of a possible sentence).

Pay now an attention to operations which can be defined over languages. These operations can be assigned to two groups. The first group contain usual set operations as *union*, *intersection*, and *complement* over languages. If L_1 is, for example, a language over alphabet Σ, then its complement L_2 is the language given as the set difference $L_2 = \Sigma^* \setminus L_1$.

The second group of operations reflect the specific feature of languages, i.e. the fact that elements of languages are strings. We will define two operations from this group - concatenation and iteration of languages.

Definition 2.8 Let L_1 be a language over alphabet Σ_1, L_2 be a language over alphabet Σ_2. The *concatenation of languages* L_1 and L_2 is the language $L_1 \cdot L_2$ over alphabet $\Sigma_1 \cup \Sigma_2$ defined in the following way:

$$L_1 \cdot L_2 = \{xy \mid x \in L_1, y \in L_2\}$$

Concatenation of languages is defined on the base of concatenation of strings and has the same algebraic properties, i.e. is associative, but not commutative.

Example 2.8 Let:

$$\begin{aligned} P &= \{A, B, \ldots, Z, a, b, \ldots, z\}, \\ C &= \{0, 1, \ldots, 9\} \quad \text{be the alphabets and} \\ L_1 &= P \\ L_2 &= (P \cup C)^* \quad \text{be the languages} \end{aligned}$$

The concatenation $L_1 \cdot L_2$ is the language of all identifiers (sequences of letters and digits which start with letter).

Using the concatenation of languages, or more specifically, the powers of languages we can define an important operation over languages – an iteration of language.

Definition 2.9 Let L be a languages over alphabet Σ. An *iteration* L^* of the language L and *positive iteration* L^+ of the languages L are defined in the following way:

$$\begin{aligned} L^0 &= \{\epsilon\} && (1) \\ L^n &= L \cdot L^{n-1} \text{ pro } n \geq 1 && (2) \\ L^* &= \bigcup_{n \geq 0} L^n && (3) \\ L^+ &= \bigcup_{n \geq 1} L^n && (4) \end{aligned}$$

Theorem 2.1 If L is a language, then:

$$\begin{aligned} L^* &= L^+ \cup \{\epsilon\} \quad \text{and} && (1) \\ L^+ &= L \cdot L^* = L^* \cdot L && (2) \end{aligned}$$

Proof: The proposition 1 stems from the definition of iteration and positive iteration:

$$L^* = L^0 \cup L^1 \cup L^2 \cup \ldots, \qquad L^+ = L^1 \cup L^2 \cup \ldots$$

$$L^* = L^0 \cup L^+ = L^+ \cup L^0 = L^+ \cup \{\epsilon\}$$

The proposition 2 is a result of concatenation of $L \cdot L^*$ resp. $L^* \cdot L$:

$$L \cdot L^* = L \cdot (L^0 \cup L^1 \cup L^2 \cup \ldots) = L^1 \cup L^2 \cup L^3 \cup \cdots = L^+,$$

and application of distributive low $L_1(L_2 \cup L_3) = L_1L_2 \cup L_1L_3$ (see Theorem 2.2). Similarly, using the alternative definition of the power $L^n = L^{n-1}L, n \geq 0$ we get $L^* \cdot L = L^+$. $\square$

Example 2.9 If $L_1 = \{(p\}, L_2 = \{,p\}, L_3 = \{)\}$ then the language $L_1 \cdot L_2^* \cdot L_3$ has strings:

$$(p) \qquad (p,p) \qquad (p,p,p) \qquad (p,p,p,p)\ldots$$

Definition 2.10 An algebraic structure $\langle A, +, \cdot, 0, 1\rangle$ is called a semiring with the additive unit element 0 and the multiplicative unit element 1 if and only if

(1) $\langle A, +, 0\rangle$ is a commutative monoid

(2) $\langle A, \cdot, 1\rangle$ is a monoid

(3) the distributive low with holds for the operation $\cdot$ with respect to $+$: $a \cdot (b + c) = a \cdot b + a \cdot c$ pro $a, b, c \in A$.

Theorem 2.2 The algebra of languages $\langle 2^{\Sigma^*}, \cup, \cdot, \emptyset, \{\epsilon\}\rangle$, where $\cup$ si the union and $\cdot$ is the concatenation of languages is the semiring.

Proof:

1. $\langle 2^{\Sigma^*}, \cup, \emptyset\rangle$ is the commutative monoid ($\cup$ is commutative and associative operation and $L \cup \emptyset = \emptyset \cup L = L$ for all $L \in 2^{\Sigma^*}$).
2. $\langle 2^{\Sigma^*}, \cdot, \{\epsilon\}\rangle$ is the monoid: $L \cdot \{\epsilon\} = \{\epsilon\} \cdot L = L$ for all $L \in 2^{\Sigma^*}$.
3. For all $L_1, L_2, L_3 \in 2^{\Sigma^*}$:
   $L_1(L_2 \cup L_3) = \{xy \mid (x \in L_1) \wedge (y \in L_2 \vee y \in L_3)\} =$
   $= \{xy \mid (x \in L_1 \wedge y \in L_2) \vee (x \in L_1 \wedge y \in L_3)\} =$
   $= \{xy \mid x \in L_1 \wedge y \in L_2\} \cup \{xy \mid x \in L_1 \wedge y \in L_3\} =$
   $= L_1L_2 \cup L_1L_3.$

□

Example 2.10 Let $L_1 = \{xy, z, x\}, L_2 = \{a, bc\}$ be languages. Specify languages L_1L_2, L_2^2, L_2^*.

$\boxed{x+y}$

Solution:

(1) $L_1L_2 = \{xy, z, x\} \cdot \{a, bc\} = \{xya, xybc, za, zbc, xa, xbc\}$

(2) $L_2^2 = L_2 \cdot L_2 = \{aa, abc, bca, bcbc\}$

(3) $L_2^* = \{w \mid w$ is the empty string or string over alphabet $\{a, b, c\}$ in which any occurrence of c immediately precedes an occurrence of symbol b and behind any occurrence of symbol b immediately succeeds symbol $c\}$.

2.2 Grammar

A concept of grammar is strongly connected with the language representation problem. The trivial representation in the form of enumeration of sentences is useless not only for infinite language but for large finite languages too. Also the usual mathematical language (used in previous examples) can be only used for description of languages with the very simple structure.

The grammar, as a most popular tool for language representation, fulfils the basic requirement for the representation - is finite even the specified language is infinite. The grammar uses two disjoint alphabet:

1. set N of nonterminal symbols
2. set Σ of terminal symbols

Nonterminal symbols, or *nonterminals*, ply the role of auxiliary variables describing some syntactical tokens - syntactical categories.

The set of *terminal symbols*, shortly *terminals*, is identical with the language alphabet. The union $N \cup \Sigma$, is called *the vocabulary of the grammar.*

Convention 2.3 The following conventions will be used for writing terminal and nonterminal symbols and for strings over the grammar vocabulary:

1. a, b, c, d denote terminal symbols
2. A, B, C, D, S denote nonterminal symbols
3. $U, V, \ldots, Z$ denote terminal symbols or denote or terminal symbols
4. $\alpha, \beta, \ldots, \omega$ denote strings of terminal and nonterminal symbols
5. $u, v, \ldots, z$ denote strings of terminal symbols only
6. string in angle parentheses $\langle, \rangle$ denotes a nonterminal symbol (the convention used in BNF).

Example 2.11 The grammar vocabulary for definition of the language of identifiers can be in the form:

$$N = \{\langle \text{identifier} \rangle, \langle \text{letter} \rangle, \langle \text{digit} \rangle\}$$

$$\Sigma = \{A, B, \ldots, Z, \quad a, b, \ldots, z, \quad 0, 1, \ldots, 9\}$$

Hence it has 65 symbols.

The grammar is a generative system in which the strings of terminal symbols are able to be generated from the distinguished nonterminal. These strings are just the elements, sentences, of the language defined by the given grammar. p

The heart of the grammar is a finite set P of *rewriting rules* (called also productions). Any rewriting rule is in the form of ordered pair (α, β) of strings; it prescribes the possible substitution of the string β instead (in place) of string α, which, as a substring, is occurring in the string generated in the grammar. The string α has at least one nonterminal symbol, string β is an element of the set $(N \cup \Sigma)^*$. Formally the set P of rewriting rules is a subset of Cartesian product:

$$P \subseteq (N \cup \Sigma)^* N (N \cup \Sigma)^* \times (N \cup \Sigma)^*$$

Example 2.12 Let us suppose that the pair (AB, CDE) is one of rewriting rule of a grammar and also suppose that the string $x = FGABH$ has been generated applying another rules in the grammar. If we now apply the rule (AB, CDE) to the string x , we obtain the string $y = FGCDEH$ (by replacing the string AB in x by the string CDE). We say that the string y has been derived from the string x according the rule (AB, CDE).

Now we are prepared to define formally the grammar and other concepts which can be used to specify the language generated by the grammar.

Definition 2.11 A grammar G is four-tuple $G = (N, \Sigma, P, S)$, where

- N is a finite set of nonterminal symbols

- Σ is a finite set of nonterminal symbols, $N \cap \Sigma = \emptyset$
- P is a finite subset of Cartesian product $(N \cup \Sigma)^* N (N \cup \Sigma)^* \times (N \cup \Sigma)^*$
- $S \in N$ is the start symbol of the grammar

An element (α, β) of the set P is called *the rewriting rule* (shortly *the rule*) and will be write in the form $\alpha \rightarrow \beta$. The string α, resp. β is called the *left hand side* resp. the *right hand side* of rewriting rule.

Example 2.13

$$G = (\{A, S\}, \{0, 1\}, P, S)$$
$$P = \{S \rightarrow 0A1, 0A \rightarrow 00A1, A \rightarrow \epsilon\}$$

Example 2.14 The grammar defining the language of identifiers could be:

$$\begin{array}{rcl}
G & = & (N, \Sigma, P, S),\ \text{where} \\
N & = & \{\langle\text{identifier}\rangle, \langle\text{letter}\rangle, \langle\text{digit}\rangle\} \\
\Sigma & = & \{A, B, \ldots, Z, \quad a, b, \ldots, z, \quad 0, 1, \ldots, 9\} \\
P & = & \{\langle\text{identifier}\rangle \rightarrow \langle\text{letter}\rangle, \\
 & & \langle\text{identifier}\rangle \rightarrow \langle\text{identifier}\rangle\langle\text{letter}\rangle, \\
 & & \langle\text{identifier}\rangle \rightarrow \langle\text{identifier}\rangle\langle\text{digit}\rangle, \\
 & & \langle\text{letter}\rangle \rightarrow A, \\
 & & \quad\quad\vdots \\
 & & \langle\text{letter}\rangle \rightarrow Z, \\
 & & \langle\text{digit}\rangle \rightarrow 0, \\
 & & \quad\quad\vdots \\
 & & \langle\text{digit}\rangle \rightarrow 9\} \\
S & = & \langle\text{identifier}\rangle
\end{array}$$

The set P contains 65 rewriting rules.

Convention 2.4 If the set P contains rules of the form $\alpha \rightarrow \beta_1, \alpha \rightarrow \beta_2, \ldots, \alpha \rightarrow \beta_n$, then we can use notation $\alpha \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$ to shorten the grammar specification.

Definition 2.12 Let $G = (N, \Sigma, P, S)$ be a grammar and let λ a μ be strings from $(N \cup \Sigma)^*$. The strings λ and μ are in binary relation $\underset{G}{\Rightarrow}$ called the *direct derivation* if and only if the strings λ a μ can be written in the form

$$\begin{array}{rcl}
\lambda & = & \gamma\alpha\delta \\
\mu & = & \gamma\beta\delta
\end{array}$$

where γ and δ are strings from $(N \cup \Sigma)^*$ and $\alpha \to \beta$ is a rewriting rule from P.

If the strings λ a μ satisfy the relation of direct derivation, then we will write $\lambda \underset{G}{\Rightarrow} \mu$ and we say that the string μ can be directly derived the string λ in the grammar G. When it is clear which grammar we are talking about, we shall drop an index under symbol $\Rightarrow$.

Example 2.15 Let us consider the grammar from the example 2.13 and the strings $\lambda = 000A111$ a $\mu = 0000A1111$. If we put

x+y

$$\begin{aligned} \lambda &= \underbrace{00}_{\gamma}\,\underbrace{0A}_{\alpha}\,\underbrace{111}_{\delta} \\ \mu &= \underbrace{00}_{\gamma}\,\underbrace{00A1}_{\beta}\,\underbrace{111}_{\delta} \end{aligned}$$

We see that $000A111 \Rightarrow 0000A1111$, as $0A \to 00A1$ is the rule in the grammar.

Example 2.16 If $\alpha \to \beta$ is the rule in a grammar G then $\alpha \Rightarrow \beta$. It stems from definition 2.12, if we put $\gamma = \delta = \epsilon$.

Definition 2.13 Let $G = (N, \Sigma, P, S)$ be a grammar and λ and let μ be strings from $(N \cup \Sigma)^*$. The strings λ and μ are in binary relation $\Rightarrow^+$ called *derivation* if and only if there is a sequence of direct derivations $\nu_{i-1} \Rightarrow \nu_i$ $i = 1, \dots, n$, $n \geq 1$ such that:

DEF

$$\lambda = \nu_0 \Rightarrow \nu_1 \Rightarrow \dots \Rightarrow \nu_{n-1} \Rightarrow \nu_n = \mu$$

This sequence is called *the derivation of length n.* If $\lambda \Rightarrow^+ \mu$ then we say that string μ is *generated* from the string λ in G. The relation $\Rightarrow^+$ is obviously the transitive closure of the relation direct derivation $\Rightarrow$. The symbol $\Rightarrow^n$ denotes n power of relation $\Rightarrow$.

Example 2.17 In the grammar from the example 2.13 is

$$0^n A 1^n \Rightarrow 0^{n+1} A 1^{n+1} \qquad n > 0$$

(since $0A \to 00A1$, see example 2.15) and hence also $0A1 \Rightarrow^+ 0^n A 1^n$ for any $n > 1$.

Definition 2.14 If for strings λ and μ hold $\lambda \Rightarrow^+ \mu$ or $\lambda = \mu$ in G then we write $\lambda \Rightarrow^* \mu$. The relation $\Rightarrow^*$ is the transitive and reflexive closure of the relation of direct derivation $\Rightarrow$.

DEF

Example 2.18 The $0A1 \Rightarrow^{+} 0^nA1^n, \quad n > 1$ from example 2.17 can be extended to the relation

$$0A1 \Rightarrow^{*} 0^nA1^n, \quad n > 0$$

Note 2.1 If we reach a string which is composed from terminal symbols only in a sequence of derivations then the generation process has to terminate (as the consequent of the form of rewriting rules). That is why the grammar symbols are called terminals and on the other hand nonterminals.

Definition 2.15 Let $G = (N, \Sigma, P, S)$ be a grammar. The string $\alpha \in (N \cup \Sigma)^*$ is called the *sentential form* if $S \Rightarrow^* \alpha$, i.e. the string α is generated from the start symbol S. The sentential form composed from terminal symbols only is called the *sentence. language* $L(G)$, generated by the grammar G is the set of all sentences:

$$L(G) = \{w \mid S \ \Rightarrow^* w \wedge w \in \Sigma^*\}$$

Example 2.19 The language generated by grammar G from the example 2.13 is the set

$$L(G) = \{0^n1^n \mid n > 0\},$$

since

$$\begin{array}{lll} S & \Rightarrow & 0A1 \\ S & \Rightarrow^* & 0^nA1^n \qquad n > 0 \qquad \text{(see example 2.17 a 2.18)} \\ S & \Rightarrow^* & 0^n1^n \qquad n > 0 \qquad \text{(after application of the rule } A \rightarrow \epsilon) \end{array}$$

2.3 Chomsky classification of grammar

The Chomsky classification of grammar (and languages), known also as Chomsky hierarchy of languages, determine four types of grammars according conditions for forms of rewriting rules being in P. These types are type 0, type 1. type 2 a type 3.

2.3.1 Type 0

A grammar of the type 0 has the rules according to the definition 2.11 of the grammar:

$$\alpha \rightarrow \beta, \alpha \in (N \cup \Sigma)^* N (N \cup \Sigma)^*, \beta \in (N \cup \Sigma)^*$$

That is why the grammar of type 0 is also called the *unrestricted* grammar.

Example 2.20 An example of an unrestricted grammar:

$$\begin{array}{rcl} G & = & (\{A, B\}, \{a, b\}, P, A) \text{ with rules} \\ A & \rightarrow & AbB \mid a \\ AbB & \rightarrow & baB \mid BAbB \\ B & \rightarrow & b \mid \epsilon \end{array}$$

2.3.2 Type 1

A grammar of the type 1 has the rules of the form:

$$\alpha A\beta \to \alpha\gamma\beta, \quad A \in N, \quad \alpha, \beta \in (N \cup \Sigma)^*, \quad \gamma \in (N \cup \Sigma)^+ \text{ or } S \to \epsilon$$

Grammars of the type 1 are also called the *context-sensitive grammars*, since, using this grammar, we are able to describe that a nonterminal A can be replaced by the string γ only in the case when its right context in the given sentential form is the string β and its left context is the string α.

The rules $\alpha A\beta \to \alpha\beta$ are nor allowed in context sensitive grammars, e.i. nonterminals can not be replaced by the empty string. The only exception is the rule $S \to \epsilon$ (S is the start symbol), which is used in the case when ϵ belongs to the language generated by the grammar. The consequent of this fact is the property of derivations: strings in derivations are not shorten, e.i. if $\lambda \Rightarrow \mu$ then $|\lambda| \leq |\mu|$ (except $S \Rightarrow \epsilon$). This property is sometimes used as the alternative for context-sensitive grammars definition.

Example 2.21 An example of a context-sensitive grammar:

$$\begin{array}{rcll} G & = & (\{A, S\}, \{0, 1\}, P, S) & \text{with rules} \\ S & \to & 0A1 \mid 01 & \\ 0A & \to & 00A1 & (\alpha = 0, \beta = \epsilon, \gamma = 0A1) \\ A & \to & 01 & \end{array}$$

2.3.3 Type 2

Grammars of the type 2 has the rules of the form:

$$A \to \gamma, \quad A \in N, \quad \gamma \in (N \cup \Sigma)^*$$

Grammar of type 2 are also called *context-free grammars* since the substitution for nonterminal A according rewriting rules can done without respect to the context of A. In the context-free grammar the rule of the form $A \to \epsilon$ can be used. But in the chapter 4 we will show that any context-free grammar can be transformed to the equivalent grammar without such rules except the rule $S \to \epsilon$ eventually, and without any occurrences of the start symbol on the right-hand side of any rule.

Example 2.22 An example of a context-free grammar:

$$\begin{array}{rcll} G & = & (\{S\}, \{0, 1\}, P, S) & \text{with rules} \\ S & \to & 0S1 \mid \epsilon & \end{array}$$

Note that the language generated by this grammar is the same as the languages generated by the grammar from the example 2.21:

$$L(G) = \{0^n 1^n\}, \qquad n \geq 0$$

2.3.4 Type 3

A grammar of the type 3 has the rules of the form:

$$A \to xB \quad \text{or} \quad A \to x; \quad A, B \in N, \quad x \in \Sigma^*$$

Grammars with this form of rules are called *right-linear grammars* (the only possible occurrence of nonterminal in right-hand side of any rule is the right most symbol). We will show in the next chapter that any right-linear grammar can be transformed to the special right-linear grammar with rules of the form:

$$A \to aB \quad \text{nebo} \quad A \to a; \quad A, B \in N, \quad a \in \Sigma \quad \text{nebo} \quad S \to \epsilon$$

This type of grammar is called the *regular grammar*, specifically *right-regular grammar*. Hence grammars of type 3 are also called *regular grammars*.

Example 2.23 An example of a grammar of type 3:

$$\begin{array}{rcl} G & = & (\{A, B\}, \{a, b, c\}, P, A) \quad \text{with rules} \\ A & \to & aaB \mid ccB \\ B & \to & bB \mid \epsilon \end{array}$$

Definition 2.16 The language generated by the grammar of type $i, i = 0, 1, 2, 3$, is called the language of type i. Alternatively we speak about *recursively enumerable* languages ($i = 0$) and with analogy to corresponding grammars about *context-sensitive* ($i = 1$), *context-free* ($i = 2$) and *regular* ($i = 3$) languages.

Theorem 2.3 Let $\mathcal{L}_i, i = 0, 1, 2, 3$ be the class of all languages of the type i. Then $\mathcal{L}_0 \supseteq \mathcal{L}_1 \supseteq \mathcal{L}_2 \supseteq \mathcal{L}_3$.

Proof: From the definition of grammar of the type i stems, that any grammar of the type 1 is also the grammar of the type 0 and any grammar of the type 3 is also the context-free grammar. The inclusion $\mathcal{L}_1 \supseteq \mathcal{L}_2$ stems from the alternative definition of a context-sensitive grammar and mentioned fact that any context-free grammar can be transformed to the equivalent grammar without such rules except the rule $S \to \epsilon$ eventually and without any occurrences of the start symbol on right-hand side of any rule.

□

The next theorem which is the more strength form of the theorem 2.3 belongs to the basic theorems of the formal languages theory. This theorem determines so called Chomsky hierarchy of formal languages. The theorem will be proved in next chapters.

Theorem 2.4 Let $\mathcal{L}_i, i = 0, 1, 2, 3$ be the class of all languages of the type i. Then $\mathcal{L}_0 \supset \mathcal{L}_1 \supset \mathcal{L}_2 \supset \mathcal{L}_3$.

Formal languages are an universal tool for coding information and descriptions of objects in various applications which can be represented and processed by computer programs. The algebraic nature of operation over formal languages enables the composite solutions of complex problems using the principle of problem decomposition, a formal description of components and a composition of resulting solution. Chomsky hierarchy of grammars and formal languages is standard scheme for application problems classification and the way how to specify the modeling and decision power of given class of different formal models.

2.4 Exercises

Exercise 2.4.1 Give well orders (according to the set inclusion) of the following languages:

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

(b) $a^*b^*c^*$

(c) $\{w \mid w \in \{a, b\}^*$ and the number of a's in w equals the number of b's $\}$

Exercise 2.4.2 Decide the the type of the following grammars according to the Chomsky classification of grammars. The rewriting rules are given only. Capitals are nonterminals (S is the start symbol) and lower-case letters and digits are terminals. symbols.

$$\begin{array}{llll}
\text{(a)} & S \rightarrow bSS \mid a & & \\
\text{(b)} & S \rightarrow 00S1 \mid \epsilon & & \\
\text{(c)} & S \rightarrow bA & A \rightarrow eB & B \rightarrow gC \\
 & C \rightarrow iD & D \rightarrow n & \\
\text{(d)} & S \rightarrow \text{sur } A & S \rightarrow \text{ in } A & S \rightarrow \text{ bi } A \\
 & S \rightarrow \text{ pro } A & A \rightarrow \text{ jection} & \\
\text{(e)} & S \rightarrow SZ_1 & Y_1Z_3 \rightarrow YZ_3 & ZZ_1 \rightarrow Z_2Z_1 \\
 & S \rightarrow X_1Z_1 & YZ_3 \rightarrow YZ & Z_2Z_1 \rightarrow Z_2Z \\
 & X_1 \rightarrow aY_1 & Y_1Y \rightarrow Y_1Z_2 & Z_2Z \rightarrow Z_1Z \\
 & X_1 \rightarrow aX_1Y_1 & Y_1Y_2 \rightarrow YY_2 & Y \rightarrow b \\
 & Y_1Z_1 \rightarrow Y_1Z_3 & YY_2 \rightarrow YY_1 & Z \rightarrow c \\
\text{(f)} & S \rightarrow YXY & ZY \rightarrow XXY & YX \rightarrow YZ \\
 & X \rightarrow a & ZX \rightarrow XXZ & Y \rightarrow \epsilon
\end{array}$$

Construct an examples of derivations of sentences in the grammars (a)–(f)and describe the languages $L(G)$ generated by these grammars.

Exercise 2.4.3 The following conventions are often used describing the syntax of programming languages constructs: The brackets [and] closed a string which can alternatively occurred or be omitted, e.g.

$$\langle\texttt{statement}\rangle \rightarrow \texttt{if}\ \langle\texttt{expr}\rangle\ \texttt{then}\ \langle\texttt{statement}\rangle\ \texttt{else}\ [\langle\texttt{statement}\rangle]$$

The brackets { and } then describes possible iterations of enclosed string including 0number iteration as in the following example of the definition of composed statement:

$$\langle\texttt{statement}\rangle \rightarrow \texttt{begin}\ \langle\texttt{statement}\rangle\ \{;\ \langle\texttt{statement}\rangle\}\ \texttt{end}\ .$$

Give an equivalent description using context-free rules for the following constructs:

(1) $A \rightarrow \alpha\ [\beta]\ \gamma$

(2) $A \rightarrow \alpha\ \{\beta\}\ \gamma$

Exercise 2.4.4 Give the grammar of the type 3 which generates identifiers composed from at most 6 characters and starting with the first letter I, J, K, L, M or N (integer variables in the programming language Fortran).

Exercise 2.4.5 Give the grammar of the type 3 for numbers of the programming language Pascal.

Exercise 2.4.6 Give the grammar of the type 2 which generates all strings w over binary alphabet such that the number of 0's in w equals the number of 1's.

Exercise 2.4.7 Show that the grammar $G = (\{S, B, C\}, \{a, b, c\}, P, S)$, of the type 0, where P has the rules

$$\begin{aligned}
S &\rightarrow SaBC \mid aBC\\
BC &\rightarrow CB\\
Ca &\rightarrow aC\\
aB &\rightarrow Ba\\
CB &\rightarrow BC\\
Ba &\rightarrow aB\\
B &\rightarrow b\\
aC &\rightarrow Ca\\
C &\rightarrow c
\end{aligned}$$

generates sentences in which the number of occurrences of symbols a, b, c equal.

Exercise 2.4.8 Let N be the set of nonterminals symbols and Σ the set of terminals. Show that the rules of the form $A \rightarrow r$, where $A \in N$ and r r is a regular expression (see the next chapter for definition eventually) over alphabet $(N \cup \Sigma)$ can be described by a context-free grammar.

According to the designed solution construct an equivalent context-free grammar to the rule $A \rightarrow a\ (C\ +\ D)$, where the symbols *, $^+$, (,) are the meta symbols of a regular expression.

Exercise 2.4.9 Give grammars which generate the following languages:

(a) $L = \{ww^R \mid w \in \{a, b\}^*\ \}$

(b) $L = \{wcw^R \mid w \in \{a, b\}^+\ \}$

(c) The language of arithmetic expressions with operators $+, *, /$ and $-$.The minus is a unary as well as a binary operator.

(d) The language identifiers (give grammar with the left recursion, the right recursion, and with the minimal number of rules).

Chapter 3

Regular languages

20:00

The aim of this chapter is thorough understanding the alternative means of the specification of the regular languages, proof techniques, that verify their equivalence, and the formal algorithmic notations of their mutual conversions. It leads to understanding of the basic properties of regular languages, which makes this class the most applicable in wide range of the technical and the nontechnical domains. This chapter determines the theory frames, that are used in the following parts of the study text.

3.1 Languages accepted by finite automata and deterministic finite automata

3.1.1 Nondeterministic finite automaton

Definition 3.1
A nondeterministic finite automaton (NFA) is a 5-tuple $M = (Q, \Sigma, \ \delta, q_0, F)$, where

(1) Q is a finite set of states

(2) Σ is a finite set of input symbols (input alphabet)

(3) δ is a mapping $Q \times \Sigma \to 2^Q$ (2^Q is set of subset of Q)

(4) $q_0 \in Q$ is a initial state

(5) $F \subseteq Q$ is a set of final states

The mapping δ is called the *transition function*, if $\delta : Q \times \Sigma \to Q$, then the automaton M is *a deterministic finite automaton*.

The operation of the finite automaton M is given by sequence of the transitions which are defined by the transition function δ. This function determines a new state q_j of the automaton M according to a actual state q_i and just read symbol $a \in \Sigma$ of input string. If M is a deterministic finite automaton then δ returns the only state q_j; in case of a nondeterministic finite automaton δ returns a set of states $Q_j, \ Q_j \in 2^Q$.

Definition 3.2 Let $M = (Q, \Sigma, \delta, q_0, F)$ is a finite automaton, We denote by a *configuration* of the automaton M a couple $C = (q, w)$ from $Q \times \Sigma^*$. The configuration (q_0, w) is the *initial configuration*, a configuration $(q, \epsilon), \ q \in F$ is the *final configuration*. A *transition* of automaton M is

δ	**z**	**c**	$\cdot$	e	$\sharp$
$\mathbf{q_0}$	q_8	q_7	q_6	q_4	
$\mathbf{q_1}$					
$\mathbf{q_2}$		q_2			q_1
$\mathbf{q_3}$		q_2			
$\mathbf{q_4}$	q_3	q_2			
$\mathbf{q_5}$		q_5		q_4	q_1
$\mathbf{q_6}$		q_5			
$\mathbf{q_7}$		q_7	q_6	q_4	q_1
$\mathbf{q_8}$		q_7	q_6	q_4	

Table 3.1: The transition function of the automaton M

represented by binary relation $\vdash_M$ on the set of configurations C. Let the set $\delta(q,a)$ contains the state q' (realize $\delta(q,a) = Q_j,\ Q_j \in 2^Q,\ q' \in Q_j$), then $(q,aw)\vdash_M(q',w)$ for all $w \in \Sigma^*$. Denote by the symbol $\vdash^k_M,\ k \geq 0$ the k-power ($C\vdash^0 C'$ if and only if $C = C'$), by the symbol $\vdash^+_M$ the transitive closure and by the symbol $\vdash^*_M$ the transitive and reflexive closure of relation. If it is obvious, that if we consider the automaton M, then we write the mentioned relations only in the form $\vdash,\ \vdash^k,\ \vdash^+,\ \vdash^*$.

DEF

Definition 3.3 The input string w is *accepted* by the finite automaton M, if and only if $(q_0,w)\vdash^* (q,\epsilon),\ q \in F$. We denote by $L(M)$ the language accepted by finite automaton M. The language $L(M)$ is set of all strings accepted by the finite automaton M:

$$L(M) = \{w \mid (q_0,w)\overset{*}{\vdash}(q,\epsilon) \land q \in F\}$$

x+y

Example 3.1 The finite deterministic automaton

$$M = (\{g_0,q_1,q_3,q_4,q_5,q_6,q_7,q_8\},\ \{c,z,_{10},\cdot,\sharp\},\ \delta,q_0,\{q_1\}),$$

whose transition function δ is given by the table 3.1, accepts the language which contains notation of natural numbers as we know it from some programming languages. The symbol c stands for an element of set $\{0,1,\dots,9\}$, the symbol z stands for an element of set $\{+,-\}$; symbol $\sharp$ ends the notation of a number.

For example this sequence of configurations leads to accepting of the input string $zc.c\mathtt{e}zc\sharp$ (e.g number $+3.1e-5$):

$$\begin{array}{rcl}
(q_0, zc.c\mathtt{e}zc\sharp) & \vdash & (q_8, c.c\mathtt{e}zc\sharp) \\
 & & (q_7, .c\mathtt{e}zc\sharp) \\
 & & (q_6, c\mathtt{e}zc\sharp) \\
 & & (q_5, \mathtt{e}zc\sharp) \\
 & & (q_4, zc\sharp) \\
 & & (q_3, c\sharp) \\
 & & (q_2, \sharp) \\
 & & (q_1, \epsilon)
\end{array}$$

Since $(q_0, zc.c\mathtt{e}zc\sharp) \vdash^* (q_1, \epsilon)$, the string $zc.c\mathtt{e}zc\sharp$ belongs to $L(M)$.

Example 3.2 The nondeterministic final automaton

$$M_1 = (\{q_0, q_1, q_2, q_F\},\ \{0, 1\},\ \delta,\ q_0,\ \{q_F\})$$

has the transition function defined as follows:

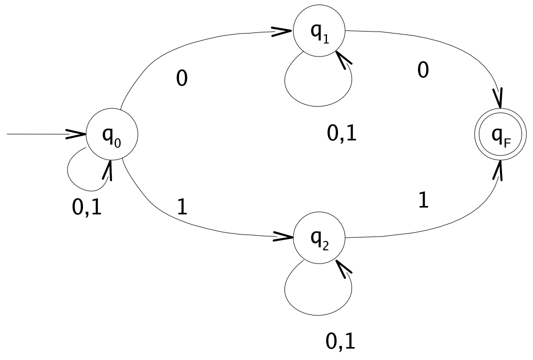
$$\begin{array}{lll}
\delta: & \delta(q_0, 0) = \{q_0, q_1\} & \delta(q_0, 1) = \{q_0, q_2\} \\
 & \delta(q_1, 0) = \{q_1, q_F\} & \delta(q_1, 1) = \{q_1\} \\
 & \delta(q_2, 0) = \{q_2\} & \delta(q_2, 1) = \{q_2, q_F\} \\
 & \delta(q_F, 0) = \emptyset & \delta(q_F, 1) = \emptyset
\end{array}$$

The alternative techniques for representation of transition function δ can be: 1. the matrix of transitions or 2. the transition diagram.

1. the matrix of transitions

	0	1
q_0	$\{q_0, q_1\}$	$\{q_0, q_2\}$
q_1	$\{q_1, q_F\}$	$\{q_1\}$
q_2	$\{q_2\}$	$\{q_2, q_F\}$
q_F	$\emptyset$	$\emptyset$

2. the transition diagram



Example 3.3 Consider NFA M_1 from example 1.5. where
$(q_0, 1010) \vdash (q_0, 010) \vdash (q_1, 10) \vdash (q_1, 0) \vdash (q_f, \epsilon)$ and hence: $(q_0, 1010) \overset{*}{\vdash} (q_f, \epsilon)$ But for instance $(q_0, \epsilon) \overset{*}{\vdash} (q_f, \epsilon)$ doesn't hold.

The language $L(M_1)$ can be described as: $L(M_1) = \{w \mid w \in \{0,1\}^* \wedge$ w is terminated by the symbol, which has already occurred in the string $w\}$.

A relationship between nondeterministic and deterministic finite automata comes under basic pieces of knowledge in formal language theory.

Theorem 3.1 Any *nondeterministic* finite automaton M can be transformed to the equivalent *deterministic* finite automat M' such that $L(M) = L(M')$.

Proof:

1. Primarily we will find the algorithm to transform $M \rightarrow M'$:

Algorithm 3.1
The conversion between nondeterministic finite automaton and equivalent deterministic finite automaton.

Input: A nondeterministic finite automat $M = (Q, \Sigma, \delta, q_0, F)$

Output: A deterministic finite automat $M' = (Q', \Sigma, \delta', q_0', F')$

Method:

1. Let $Q' = (2^Q \setminus \{\emptyset\}) \cup \{nedef\}$
2. Let $q_0' = \{q_0\}$
3. For all $S \in 2^Q \setminus \{\emptyset\}$ and for all $a \in \Sigma$ set : $\delta'(S, a) = \bigcup_{q \in S} \delta(q, a)$
   If $\delta'(S, a) = \emptyset$, let $\delta'(S, a) = nedef$.
4. Set $F' = \{S \mid S \in 2^Q \wedge S \cap F \neq \emptyset\}$

2. Now we will prove $L(M) = L(M')$ that means it holds:
   (a) $L(M) \subseteq L(M')$ and at the same time,
   (b) $L(M') \subseteq L(M)$.

We leave this part of proof as an exercise.

□

Example 3.4 Consider the nondeterministic finite automaton
$M_2 = (\{S, A, B, C\},\ \{a, b,\ c\}, delta,\ S,\ \{A\})$ where δ is defined as follows: $\delta(S, a) = \{A\},\ \delta(S, c) = \{B\},\ \delta(B, b) = \{B, C\},\ \delta(C, a) = \{B\},\ \delta(C, c) = \{A\}$.

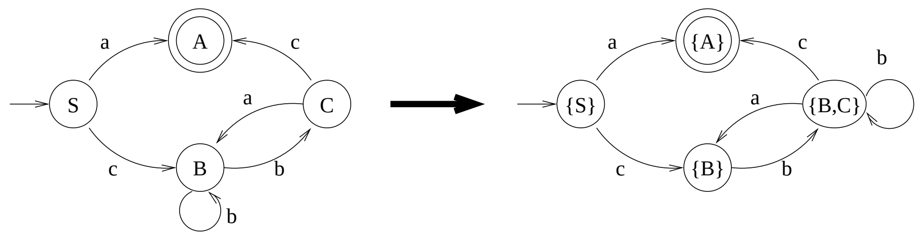
We will use short method to find the transition function δ' of equivalent DFA because usually many states from 2^Q are unreachable.

1. Initial state: $\{S\}$

2. $\delta'(\{S\}, a) = \{A\}$ — the final state
   $\delta'(\{S\}, c) = \{B\}$

3. $\delta'(\{B\}, b) = \{B, C\}$

4. $\delta'(\{B, C\}, a) = \delta(B, a) \cup \delta(C, a) = \{B\}$
   $\delta'(\{B, C\}, b) = \{B, C\}$ $\delta'(\{B, C\}, c) = \{A\}$



NFA DFA

3.1.2 Linear and regular grammar

Before we will prove the equivalence of the class $\mathcal{L}_3$ and the class of languages accepted by finite automata we show that every languages of the type 3 can be generated by special type of grammar.

Definition 3.4 A grammar $G = (N, \Sigma, P, S)$ with rules of the form:

DEF

$$\begin{array}{ll} A \to xB & A, B \in N, x \in \Sigma^* \text{ or} \\ A \to x & x \in \Sigma^*, \end{array}$$

resp. with rules of the form:

$$\begin{array}{ll} A \to Bx & A, B \in N \\ A \to x & x \in \Sigma^* \end{array}$$

is called the*right-linear*, resp. the *left-linear* grammar.
A grammar $G = (N\Sigma, P, S)$ with rules of the form:

$$\begin{array}{rll} & A \to aB & A, B \in N, a \in \Sigma \\ & A \to a & a \in \Sigma \\ \text{eventually} & S \to \epsilon & \text{if } \epsilon \in L(G), \end{array}$$

resp. with rules of the form:

$$\begin{array}{rll} & A \to Ba & A, B \in N, a \in \Sigma \\ & A \to a & a \in \Sigma \\ \text{eventually} & S \to \epsilon & \text{if } \epsilon \in L(G) \end{array}$$

is called the *right-regular*, resp. the *left-regular* grammar.

Note 3.1 A grammar $G = (N, \Sigma, P, S)$ with rules of the form $A \to xBy$ or $A \to x$, where $A, B \in N$ and $x, y \in \Sigma^*$ is called the *linear grammar*.

Denote:

- $\mathcal{L}_{PL}$ – all languages generated by right-linear grammars,
- $\mathcal{L}_{LL}$ – all languages generated by left-linear grammars,
- $\mathcal{L}_{L}$ – all languages generated by linear grammars.

It holds: $\mathcal{L}_{PL} = \mathcal{L}_{LL}$ and $\mathcal{L}_{PL} \subset \mathcal{L}_{L}$.

Theorem 3.2 Any right-linear grammar $G = (N, \Sigma.P, S)$, it means a grammar contains only rules of the form $A \to xB$ or $A \to x$, where $A, B \in N$, $x \in \Sigma^*$, can be transformed to the grammar $G' = (N', \Sigma, P', S')$, which contains only rules of the form $A \to aB$ or $A \to \epsilon$, whereas $L(G) = L(G')$.

Proof: We construct the set $P'$of rewriting rules of grammar the G' as follows:

(1) all rules from P of the form $A \to aB$ and $A \to \epsilon$ where $A, B \in N$, $a \in \Sigma$ insert into the set P'

(2) every rule of the form $A \to a_1 a_2 \dots a_n B$; $A, B \in N$, $a_i \in \Sigma$, $n \geq 2$ from the set P replace by the rules:

$$\begin{array}{rcl} A & \to & a_1 A_1 \\ A_1 & \to & a_2 A_2 \\ \vdots & & \\ A_{n-1} & \to & a_n B \end{array}$$

in the set P'.

We add new established nonterminals $A_1, \dots, A_{n-1}$ to the set N'. The derivation $A \underset{G}{\Rightarrow} a_1 \dots a_n B$ obviously corresponds to the derivation $A \underset{G'}{\Rightarrow}^{n} a_1 a_2 \dots a_n B$.

(3) We replace every rule of form $A \to a_1 \dots a_n$, $a_i \in \Sigma$, $n \geq 2$ from the set P by the rules:

$$\begin{array}{rcl} A & \to & a_1 A'_1 \\ A'_1 & \to & a_2 A'_2 \\ \vdots & & \\ A'_{n-1} & \to & a_n A'_n \\ A'_n & \to & \epsilon \end{array}$$

in the set P'. The derivation $A \underset{G}{\Rightarrow} a_1 a_2 \dots a_n$ corresponds to the derivation $A \underset{G'}{\Rightarrow}^{n+1} a_1 a_2 \dots a_n$.

(4) We replace remaining, so-called simple rules of the form $A \to B$, $A, B \in N$, as follows:

(a) We construct for each $A \in N$ the set $N_A = \{B \mid A \Rightarrow^* B\}$.

(b) We extend the set of rules P' as follows: If $B \rightarrow \alpha$ is in P and it is not simple rule, then we add the rules $A \rightarrow \alpha$ for each A, which satisfies $B \in N_A$, to P'.

The general algorithm eliminating simple rules in a context-free grammar, which is mentioned and proved in the next chapter (algorithm 4.5) was used in the case 4b of the previous proof. This algorithm contains also a computation of the set N_A.

□

Example 3.5 Using the previous theorem we will transform the right-linear grammar $G = (\{X, Y\}, \{a, b, c\}, P, X)$, where P contains rules:

$\boxed{\mathtt{x+y}}$

$$\begin{array}{rcl} X & \rightarrow & abc \mid Y \mid \epsilon \\ Y & \rightarrow & aY \mid cbX \end{array}$$

In P' (according to 1)there will be the rules:

$$X \rightarrow \epsilon, Y \rightarrow aY$$

According to 2 we replace the rule $Y \rightarrow cbX$ by the rules

$$Y \rightarrow cZ, Z \rightarrow bX$$

According to 3 we replace the rule $Y \rightarrow abc$ by the rules

$$Y \rightarrow aU, U \rightarrow bV, V \rightarrow cW, W \rightarrow \epsilon$$

According to 4 we eliminate the rule $X \rightarrow Y$. Because $N_X = \{X, Y\}$ we add the rules

$$X \rightarrow aY, X \rightarrow cZ, X \rightarrow aU$$

to P'

The resulting grammar is in the form $G' = (\{X, Y, Z, U, V, W\}, \{a, b, c\}, P', X)$ where P' contains the rules:

$$\begin{array}{rcl} X & \rightarrow & \epsilon \mid aY \mid cZ \mid aU \\ Y & \rightarrow & aY \mid cZ \\ Z & \rightarrow & bX \\ U & \rightarrow & bV \\ V & \rightarrow & cW \\ W & \rightarrow & \epsilon \end{array}$$

Theorem 3.3 Any language of the type 3 can be generated by a regular grammar.

Proof: We obtain the desired regular grammar from the grammar constructed according to the theorem 3.2 by removing rules with the empty string on the right side. This process is described in the algorithm 4.4 in chapter the next chapter; meanwhile we demonstrate the process in an exercise.

□

Example 3.6 We transform the grammar from the exercise 3.5 to the regular grammar by removing the rule $W \rightarrow \epsilon$ (the equivalent new rule will be $V \rightarrow c$) and the rule $X \rightarrow \epsilon$ (the equivalent new rule will be $Z \rightarrow b$). Because $\epsilon \in L(G)$ and X is on the right side of the rule $Z \rightarrow bX$, we have to establish a new initial symbol X' and remove the simple rule $X' \rightarrow X$. The resulting grammar is in the form:

$$G'' = (\{X', X, Y, Z, U, V\}, \{a, b, c\}, P'', X'),$$

P'' contains rules:

$$\begin{array}{rcl}
X' & \rightarrow & \epsilon \mid aY \mid cZ \mid aU \\
Y & \rightarrow & aY \mid cZ \\
Z & \rightarrow & bX \mid b \\
X & \rightarrow & aY \mid cZ \mid aU \\
U & \rightarrow & bV \\
V & \rightarrow & c
\end{array}$$

3.1.3 Equivalence of the class $\mathcal{L}_3$ and the class of languages accepted by finite automata

We denote by $\mathcal{L}_M$ the class of languages accepted by finite automata and we show, that holds $\mathcal{L}_3 = \mathcal{L}_M$.

Theorem 3.4 Let L be a language of the type 3 . Then there exists a finite automaton M, for which holds $L = L(M)$, it means $\mathcal{L}_3 \subseteq \mathcal{L}_M$.

Proof:

According to the theorem 3.2 we can generate the language L by the grammar $G = (N, \Sigma, P, S)$ of the type 3, with rules of the form $A \rightarrow aB$ or $A \rightarrow \epsilon$ where $A, B \in N, a \in \Sigma$. We construct the finite (nondeterministic) automaton M

$$M = (Q, \Sigma, \delta, q_0, F) \text{ where}$$

(1) $Q = N$

(2) $\Sigma = \Sigma$

(3) $\delta : \delta(A, a)$ contains B for each rule $A \rightarrow aB$ z P

(4) $q_0 = S$

(5) $F = \{A \mid A \to \epsilon \text{ is the rule from } P\}$

We show, that $L(G) = L(M)$ holds.

We prove it by the induction. Let the inductional hypothesis is in the form:

> For each $A \in N$ it holds $A \underset{G}{\Rightarrow}^{i+1} w, w \in \Sigma^*$, if and only if $(A, w) \vdash_M^i (C, \epsilon)$ for some $C \in F$

First we prove this hypothesis for $i = 0$. Obviously holds

$$A \Rightarrow \epsilon \text{ if and only if } (A, \epsilon) \overset{0}{\vdash} (A, \epsilon) \text{ pro } A \in F$$

Now assume, that the hypothesis holds for i and let $w = ax, a \in \Sigma, |x| = i$. Then $A \Rightarrow^{i+1} w$ holds if and only if $A \Rightarrow aB \Rightarrow^i x$. According to the definition of the function δ, $\delta(a, A)$ contains B (due to direct derivation $A \Rightarrow aB$). On the base of the inductive hypothesis it holds $B \Rightarrow^i x$, if and only if $(B, x) \vdash^{i-1} (C, \epsilon), C \in F$. Summarizing these facts, it holds:

$$A \Rightarrow aB \Rightarrow^i ax = w, \text{ if and only if } (A, ax) \vdash (B, x) \overset{i-1}{\vdash} (C, \epsilon), C \in F$$

$$\text{tj.} \quad A \Rightarrow^{i+1} w, \text{ if and only if } (A, w) \overset{i}{\vdash} (C, \epsilon), C \in F;$$

by that we prove that the inductive hypothesis holds for all $i \geq 0$.

Particulary it holds:

$$S \Rightarrow^* w, \text{ if and only if } (S, w) \overset{*}{\vdash} (C, \epsilon), C \in F$$

Thus $L(G) = L(M)$. □

Example 3.7 Let the grammar $G = (\{X, Y, Z, U, V, W\}, \{a, b, c\}, P, X)$, P contains the rules

$\boxed{\texttt{x+y}}$

$$\begin{array}{rcl} X & \to & \epsilon \mid aY \mid cZ \mid aU \\ Y & \to & aY \mid cZ \\ Z & \to & bX \\ U & \to & bV \\ V & \to & cW \\ W & \to & \epsilon \end{array}$$

We construct a finite automaton, which accepts the language $L(G)$.

Follow the theorem 3.4. We represent the transition function δ by the transition diagram (pict. 3.1), in which the final states are marked double circle and initial state is marked small arrow without an output node.

$$\begin{array}{rcl} M & = & (Q, \Sigma, \delta, q_0, F) \text{ where} \\ Q & = & \{X, Y, Z, U, V, W\} \\ \Sigma & = & \{a, b, c\} \\ \delta & : & \text{see Fig. 3.1} \\ q_0 & = & x \\ F & = & \{X, W\} \end{array}$$

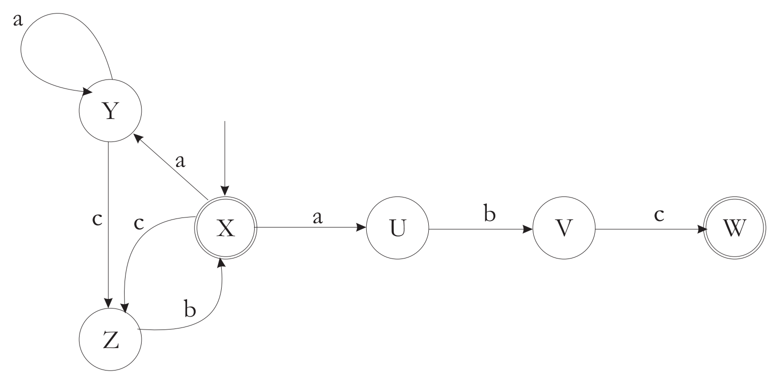


Figure 3.1: The transition diagram of the automaton M

Theorem 3.5 Let $L = L(M)$ for some finite automaton M. Then there exists a grammar G of the type 3 for which $L = L(G)$ holds, it means $\mathcal{L}_M \subseteq L_3$.

Proof: Let $M = (Q, \Sigma, \delta, q_0, F)$. Because any nondeterministic finite automaton can be transformed to the equivalent deterministic finite automaton we assume, without loss of generality, that M is the deterministic finite automaton. Let G is the grammar $G = (Q, \Sigma, P, q_0)$ of the type 3, its set P of rewriting rules is defined as follows:

(1) if $\delta(q, a) = r$, then P contains the rule $q \rightarrow ar$

(2) if $p \in F$, then P contains the rule $p \rightarrow \epsilon$

In the next part of the proof we continue analogically to the proof of the Theorem 3.4. $\square$

Example 3.8 We construct the grammar of the type 3 on the base of the exercise 3.1. This grammar will generate the language which contains notations of numbers. The transition function of the corresponding deterministic finite automaton is defined by table 3.1. The resulting grammar will be in the form:

$$G = (\{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8\}, \{c, z, \mathtt{e}, ., \sharp\}, P, q_0)$$

P contains rules

$$\begin{aligned}
q_0 &\rightarrow zq_8 \mid cq_7 \mid \cdot q_6 \mid \mathtt{e}q_4 \\
q_1 &\rightarrow \epsilon \\
q_2 &\rightarrow cq_2 \mid \sharp q_1 \\
q_3 &\rightarrow cq_2 \\
q_4 &\rightarrow zq_3 \mid cq_2 \\
q_5 &\rightarrow cq_5 \mid \mathtt{e}q_4 \mid \sharp q_1 \\
q_6 &\rightarrow cq_5 \\
q_7 &\rightarrow cq_7 \mid \cdot q_6 \mid \mathtt{e}q_4 \mid \sharp q_1 \\
q_8 &\rightarrow cq_7 \mid \cdot q_6 \mid \mathtt{e}q_4
\end{aligned}$$

Let us remark, that the transformation of this grammar to the regular grammar is easy, if you set q_1 as the empty string and remove rule $q_1 \rightarrow \epsilon$.

Theorem 3.6 The class of languages, which are accepted by finite automata,is equivalent with the class of languages of the type 3 according the Chomsky hierarchy.

Proof: The statement immediately follows from the theorems 3.4 and 3.5.
□

Theorem 3.7 Any language of the type 3 can be generated by a left-linear grammar.

Proof: We leave the whole proof as an exercise. First using definition of the regular set we can prove the statement: if L is a regular set, then L^R is also a regular set. Next we show: if $G = (N, \Sigma, P, S)$ is right-linear grammar, then left-linear grammar G', with the set of rules in the form $P' = \{A \rightarrow \alpha^R \mid A \rightarrow \alpha \text{ is in } P\}$, generates the language $(L(G))^R$. □

Example 3.9 The grammar $G' = (\{X, Y\}, \{a, b, c\}, P', X)$ where P' contains rules

$$\begin{aligned}
X &\rightarrow cba \mid Y \mid \epsilon \\
Y &\rightarrow Ya \mid Xbc
\end{aligned}$$

is the left-linear grammar, for which holds $L(G') = (L(G))^R$, where G is the right-linear grammar from exercise 3.5.

DEF

Definition 3.5 The left-linear grammar G = (N, Σ, P, S) is called *regular* (more precisely left regular), if rules in P are in the form $A \rightarrow Ba$ or $A \rightarrow a$. if L(G) contains empty string ϵ, then only one allowed rule with right side containing empty string is $S \rightarrow \epsilon$ and in this case S doesn't have to be on the right side of any rule.

Let us remark that the construction of a left regular grammar equivalent with the left linear grammar is analogous to construction of a right regular grammar equivalent with the right linear grammar. This construction illustrates the following exercise:

x+y

Example 3.10 Consider the grammar from the exercise 3.9, with rules:

$$\begin{array}{rcl} X & \rightarrow & cba \mid Y \mid \epsilon \\ Y & \rightarrow & Ya \mid Xbc \end{array}$$

The first step establishes only rules in the form $A \rightarrow Ba$ or $A \rightarrow \epsilon$:

$$\begin{array}{rcl} X & \rightarrow & Ua \mid Ya \mid Zc \mid \epsilon \\ Y & \rightarrow & Ya \mid Zc \\ U & \rightarrow & Vb \\ V & \rightarrow & Wc \\ W & \rightarrow & \epsilon \\ Z & \rightarrow & Xb \end{array}$$

The second step removes rules in the form $A \rightarrow \epsilon$ and modifies the grammar to the final form:

$$\begin{array}{rcl} X' & \rightarrow & Ua \mid Ya \mid Zc \mid \epsilon \\ Y & \rightarrow & Ya \mid Zc \\ U & \rightarrow & Vb \\ V & \rightarrow & c \\ Z & \rightarrow & Xb \mid b \\ X & \rightarrow & Ua \mid Ya \mid Zc \end{array}$$

See the difference between rules of this grammar and rules of the right regular grammar from the exercise 3.6, which we obtained from the right linear grammar (the exercise 3.5)

The algorithm of the construction a nondeterministic finite automaton from the proof of the theorem 3.4 can be easily modified for the transormation the regular grammar to an equivalent nondeterministic finite automaton.

Algorithm 3.2
The construction of a nondeterministic finite automaton to the equivalent right regular grammar.

Input: A right regular grammar $G = (N, \Sigma, P, S)$, with rules in the form $A \rightarrow aB$ and $A \rightarrow a$ where $A, B \in N$ and $a \in \Sigma$, eventually $S \rightarrow \epsilon$, if $\epsilon \in L(G)$.

Output: A nondeterministic finite automaton $M = (N, \Sigma, \delta, q_0, F)$, for which $L(M) = L(G)$ holds.

Method:

(1) Let $Q = N \cup \{q_F\}$, where q_F represent a finial state.

(2) The set of input symbols of the automaton M is identical with the set of terminals of the grammar G.

(3) Define the transition function δ as follows:

  (a) if $A \rightarrow aB$ is the rule from P, then $\delta(A, a)$ contains the state B

  (b) if $A \rightarrow a$ is the rule from P, then $\delta(A, a)$ contains q_F

(4) $q_0 = S$

(5) if $S \rightarrow \epsilon$ is rule from P, then $F = \{S, q_F\}$, in other case $F = \{q_F\}$.

Theorem 3.8 Let G be a right regular grammar and let M be the finite automaton from the algorithm 3.2, then $L(M) = L(G)$ holds.

Proof: The proof of this theorem is a modification of the proof of the theorem 3.4; we leave it as an exercise. □

Example 3.11 Consider the grammar $G = (\{S, B\}, \{0, 1\}, P, S)$ with rules

$$\begin{aligned} S &\rightarrow 0B \mid \epsilon \\ B &\rightarrow 0B \mid 1S \mid 0 \end{aligned}$$

On the base of the algorithm 3.2 the transition diagram of the automaton accepting the language $L(G)$ has the form:

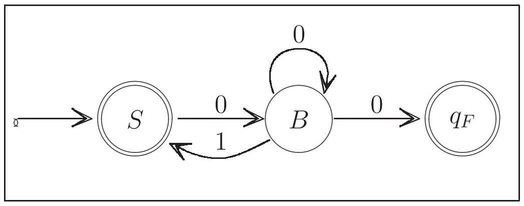


Figure 3.2: The transition diagram

Algorithm 3.3
The construction of an equivalent nondeterministic finite automaton to the given left regular grammar.

Input: A left regular grammar $G = (N, \Sigma, P, S)$ with the rules in the form $A \to Ba$ and $A \to a$ where $A, B \in N$ and $a \in \Sigma$, eventually $S \to \epsilon$, if $\epsilon \in L(G)$.

Output: A nondeterministic finite automaton $M = (N, \Sigma, \delta, q_0, F)$, for which $L(M) = L(G)$ holds.

Method:

(1) Let $Q = N \cup \{q_0\}$

(2) The set of the input symbols of the automaton M is identical with the set of terminals of the grammar G.

(3) Define the transition function δ as follows:

  (a) If $A \to Ba$ is the rule from P, then $\delta(B, a)$ contains state A.

  (b) If $A \to a$, then $\delta(q_0, a)$ contains state A.

(4) q_0 is an initial state of automaton M.

(5) If $S \to \epsilon$ is the rule from P then $F = \{S, q_0\}$, in the other case $F = \{S\}$

Theorem 3.9 Let G be a left regular grammar and let M be the finite automaton from the algorithm 3.3, then $L(M) = L(G)$ holds.

Proof: First we prove, that $\epsilon \in L(G)$, if and only if $\epsilon \in L(M)$. According to the construction of the automaton M, $q_0 \in F$ if and only if there is the rule $S \to \epsilon$ in P thus

$$S \Rightarrow \epsilon, \text{ if and only if } (q_0, \epsilon) \overset{0}{\vdash} (q_0, \epsilon),\ q_0 \in F$$

Now we prove, that for arbitrary $w \in \Sigma^+$ holds

$$S \Rightarrow^+ w, \text{ if and only if } (q_0, w) \overset{+}{\vdash} (S, \epsilon),\ S \in F$$

Let $w = ax,\ a \in \Sigma,\ x \in \Sigma^*$. Let the inductive hypothesis is in the form:

$$S \Rightarrow^i Ax \Rightarrow ax, \text{ if and only if } (q_0, ax) \vdash (A, x) \overset{i}{\vdash} (S, \epsilon),\ A \in N,\ S \in F,\ |x| = i$$

For $i = 0$ we obtain:

$$S \Rightarrow a, \text{ if and only if } (q_0, a) \vdash (S, \epsilon),$$

what follows straight from the definition of transitions of the automaton M ($\delta(q_0, a)$ contains S, if and only if $S \to a$ is in P).

For $i \geq 1$ is necessary to prove:

(a) $Ax \Rightarrow ax$, if and only if $(q_0, ax) \vdash (A, x)$

(b) $S \Rightarrow^i Ax$, if and only if $(A, x) \vdash^i (S, \epsilon)$

We leave the proof of statements a) and (b) as an exercise. $\square$

Example 3.12 Consider the grammar $G = (\{S, I, N\}, \{c, p, \sharp\}, P, S)$ with the rules:

$$\begin{array}{lcl} S & \rightarrow & I\sharp \mid N\sharp \\ I & \rightarrow & p \mid Ip \mid Ic \\ N & \rightarrow & c \mid Nc \end{array}$$

If c stands for an arabian numeral and p stands for a letter, then $L(G)$ is language of identifiers (nonterminal I) and integer numbers without sign (nonterminal N). Each sentence of the language $L(G)$ ends with the terminal character $\sharp$.

On the base of the algorithm 3.3 the transition diagram of automaton accepting language the $L(G)$ is in the form:

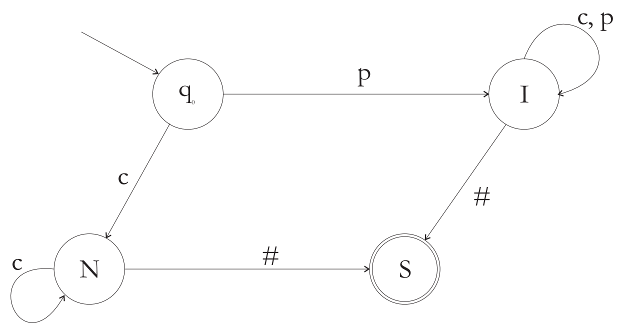


Figure 3.3: The transition diagram

3.2 Minimization of deterministic finite automata

A conversion process between a deterministic finite automaton and the minimal final automaton consist of several steps, in which we eliminate unreachable states, reduce undistinguishable states, convert automaton to reduced deterministic finite automaton and eliminate redundant states, that don't influent an input string acceptance.

Definition 3.6 A deterministic finite automaton $M = (Q, \Sigma, \delta, q_0, F)$ is called *the complete deterministic finite automaton*, if for all $q \in Q$ and for all $a \in \Sigma$ holds: $\delta(q, a) \in Q$, it means δ is a total function on $Q \times \Sigma$.

Definition 3.7 Let $M = (Q, \Sigma, \delta, q_0, F)$ be a finite automaton. A state $q \in Q$ is called *reachable*, if there is $w \in \Sigma^*$ for which $(q_0, w) \overset{*}{\underset{M}{\vdash}} (q, \varepsilon)$ holds. A state is called *unreachable*, if it is not reachable.

Algorithm 3.4
The elimination of unreachable states

Input: A deterministic finite automaton $M = (Q, \Sigma, \delta, q_0, F)$.

Output: A deterministic finite automaton M' without unreachable states for which $L(M) = L(M')$ holds.

Method:

```
1. $i := 0$
2. $S_i := \{q_0\}$
3. repeat
4.    $S_{i+1} := S_i \cup \{q \mid \exists p \in S_i \ \exists a \in \Sigma : \delta(p, a) = q\}$
5.    $i := i + 1$
6. until $S_i = S_{i-1}$
7. $M' := (S_i, \Sigma, \delta_{|S_i}, q_0, F \cap S_i)$
```

A minimization algorithm of a deterministic finite automaton is based on a concept of undistinguishable states.

Definition 3.8 Let $M = (Q, \Sigma, \delta, q_0, F)$ be a complete DFA. We say, that *a string $w \in \Sigma^*$ distinguishes the states $q_1, q_2 \in Q$* , if $(q_1, w) \overset{*}{\underset{M}{\vdash}} (q_3, \varepsilon) \wedge (q_2, w) \overset{*}{\underset{M}{\vdash}} (q_4, \varepsilon)$ for some q_3, q_4 and *only the one* from states q_3, q_4 is in F. We say that the states $q_1, q_2 \in Q$ are *k-distinguishable* (denote by $q_1 \overset{k}{\equiv} q_2$) if and only if there is any $w \in \Sigma^*$, $|w| \leq k$, which distinguishes q_1 and q_2. The states q_1, q_2 are *undistinguishable* (denote by $q_1 \equiv q_2$) if and only if for all $k \geq 0$ these state are k-undistinguishable.

It is easy to prove that relation $\equiv$ is the equivalence on Q, it means that this relation is reflexive, symmetric and transitive.

Definition 3.9 A DFA M is called *reduced*, if all states from Q are reachable and any two state from Q are undistinguishable.

Theorem 3.10 Let $M = (Q, \Sigma, \delta, q_0, F)$ is a complete DFA and $|Q| = n$, $n \geq 2$. It holds that $\forall q_1, q_2 \in Q : q_1 \equiv q_2 \Leftrightarrow q_1 \overset{n-2}{\equiv} q_2$.

Proof: The part $\Rightarrow$ of the proof is trivial, it follows straight from the definition.

The part $\Leftarrow$:

1. If $|F| = 0$ or $|F| = n$, then holds $q_1 \overset{n-2}{\equiv} q_2 \quad \Rightarrow \quad q_1 \equiv q_2$ (all states are final, or all states are not final).

2. Let $|F| > 0 \wedge |F| < n$. We show, that $\equiv \, = \overset{n-2}{\equiv} \subseteq \overset{n-3}{\equiv} \subseteq \ldots \subseteq \overset{1}{\equiv} \subseteq \overset{0}{\equiv}$ holds:

- Obviously:
  (a) $\forall q_1, q_2 \in Q : q_1 \overset{0}{\equiv} q_2 \Leftrightarrow (q_1 \notin F \wedge q_2 \notin F) \vee (q_1 \in F \wedge q_2 \in F)$.
  (b) $\forall q_1, q_2 \in Q\ \forall k \geq 1 : q_1 \overset{k}{\equiv} q_2 \Leftrightarrow \forall a \in \Sigma : \delta(q_1, a) \overset{k-1}{\equiv} \delta(q_2, a)$.
- The relation $\overset{0}{\equiv}$ is the equivalence and it determines the decomposition $\{F, Q \setminus F\}$.
- If $\overset{k+1}{\equiv} \neq \overset{k}{\equiv}$, then $\overset{k+1}{\equiv}$ is a refinement (the proper refinement) of $\overset{k}{\equiv}$, it means that the decomposition $\overset{k+1}{\equiv}$ contains at least one more class then the decomposition $\overset{k}{\equiv}$.
- If for some k the identity $\overset{k+1}{\equiv} = \overset{k}{\equiv}$ holds, then also $\overset{k+1}{\equiv} = \overset{k+2}{\equiv} = \overset{k+3}{\equiv} = ...$ holds according to (b) and thus $\overset{k}{\equiv}$ is the desired equivalence.
- Because F or $Q \setminus F$ contains no more than $n-1$ elements, we obtain a relation $\equiv$ after no more than $n-2$ refinements $\overset{0}{\equiv}$.

□

Algorithm 3.5
The construction of the reduced automaton.

Input: A complete DFA $M = (Q, \Sigma, \delta, q_0, F)$.

Output: A reduce DFA $M' = (Q', \Sigma, \delta', q_0', F')$, $L(M) = L(M')$.

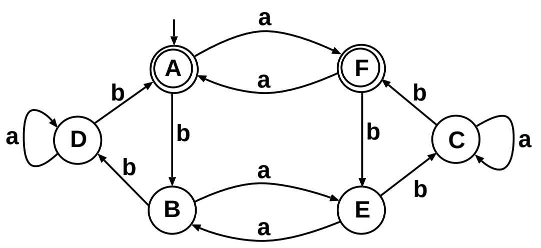
Method:

```
 1. Using the algorithm 3.4 we remove all unreachable states .
 2. $i := 0$
 3. $\overset{0}{\equiv} := \{(p, q) \mid p \in F \Longleftrightarrow q \in F\}$
 4. repeat
 5.     $\overset{i+1}{\equiv} := \{(p, q) \mid p \overset{i}{\equiv} q \wedge \forall a \in \Sigma : \delta(p, a) \overset{i}{\equiv} \delta(q, a)\}$
 6.     $i := i + 1$
 7. until $\overset{i}{\equiv} = \overset{i-1}{\equiv}$
 8. $Q' := Q/ \overset{i}{\equiv}$
 9. $\forall p, q \in Q\ \forall a \in \Sigma : \delta'([p], a) = [q] \Leftrightarrow \delta(p, a) = q$
10. $q_0' = [q_0]$
11. $F' = \{[q] \mid q \in F\}$
```

Note 3.2 The expression $[x]$ stands for the class of an equivalence which is determined by the element x.

Example 3.13 Transform the given deterministic finite automaton (defined by the transition diagram) to the equivalent reduce deterministic finite automaton.

x+y



Solution: We show in the solution how the steps of the algorithm look like: 3.5.

1. No unreachable states.
3. $\overset{0}{\equiv} = \{\{A,F\},\{B,C,D,E\}\}$
5.1. $\overset{1}{\equiv} = \{\{A,F\},\{B,E\},\{C,D\}\}$

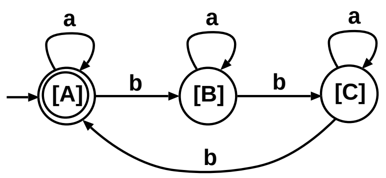
$\overset{0}{\equiv}$	δ	a	b
I:	A	F_I	B_{II}
	F	A_I	E_{II}
II:	B	E_{II}	D_{II}
	C	C_{II}	F_I
	D	D_{II}	A_I
	E	B_{II}	C_{II}

5.2. $\overset{2}{\equiv} = \{\{A,F\},\{B,E\},\{C,D\}\} = \overset{1}{\equiv} = \equiv$

$\overset{1}{\equiv}$	δ	a	b
I:	A	F_I	B_{II}
	F	A_I	E_{II}
II:	B	E_{II}	D_{III}
	E	B_{II}	C_{III}
III:	C	C_{III}	F_I
	D	D_{III}	A_I

8. $Q' = \{[A],[B],[C]\}$, where $[A] = \{A,F\}$, $[B] = \{B,E\}$, $[C] = \{C,D\}$

9–11.



3.3 Regular sets and regular expressions

3.3.1 Regular sets

Definition 3.10 Let Σ be a finite alphabet. We recursively define *regular sets* over the alphabet Σ as follows:

DEF

(1) $\emptyset$ (the empty set) is a regular set over Σ

(2) $\{\epsilon\}$ (the set containing only the empty string) is a regular set over Σ

(3) $\{a\}$ for all $a \in \Sigma$ is regular set over Σ

(4) if P and Q are regular sets over Σ, then $P \cup Q$, $P \cdot Q$ a P^* are also regular sets over Σ

(5) Regular sets are just the sets which arise by application of the rules 1–4.

The class of regular sets is the smallest class of languages which contains the sets $\emptyset, \{\epsilon\}, \{a\}$ for all symbols a and which is closed under the following operations: union, product and iteration.

Example 3.14 $L = (\{a\} \cup \{d\}) \cdot (\{b\}^*) \cdot \{c\}$ is the regular set over $\Sigma = \{a, b, c, d\}$.

We show, that the class of the regular sets forms the class $\mathcal{L}_3$, it means the class of the languages of the type 3 of Chomsky hierarchy..

Theorem 3.11 Let Σ is a finite alphabet. Then

$$\emptyset \qquad (1)$$

$$\{\epsilon\} \qquad (2)$$

$$\{a\} \quad \text{for all } a \in \Sigma \qquad (3)$$

are the languages of the type 3 over the alphabet Σ.

Proof:

(1) $G = (\{S\}, \Sigma, \emptyset, S)$ is the grammar of the type 3, $L(G) = \emptyset$

(2) $G = (\{S\}, \Sigma, \{S \rightarrow \epsilon\}, S)$ is the grammar of the type 3, $L(G) = \{\epsilon\}$

(3) $G_a = (\{S\}, \Sigma, \{S \rightarrow a\}, S)$ is the grammar of the type 3, $L(G_a) = \{a\}$

$\square$

Theorem 3.12 Let L_1 a L_2 are the languages of the type 3 over the alphabet Σ. Then

$$L_1 \cup L_2 \qquad (1)$$

$$L_1 \cdot L_2 \qquad (2)$$

$$L_1^* \qquad (3)$$

are the languages of the type 3 over the alphabet Σ.

Proof: Because L_1 and L_2 are the language of the type 3, there are some grammars of the type 3 $G_1 = (N_1, \Sigma, P_1, S_1)$ a $G_2 = (N_2, \Sigma, P_2, S_2)$ for which $L(G_1) = L_1$ and $L(G_2) = L_2$ hold. We assume without loss of generality, that the set N_1 and N_2 are disjunctive.

(1) Let $G_3 = (N_1 \cup N_2 \cup \{S_3\}, \Sigma, P_1 \cup P_2 \cup \{S_3 \to S_1, S_3 \to S_2\}, S_3)$ is the grammar of the type 3, where S_3 is a new nonterminal, $S_3 \notin N_1$, $S_3 \notin N_2$. Obviously $L(G_3) = L(G_1) \cup L(G_2)$, because for any derivation $S_3 \underset{G_3}{\Rightarrow}^+ w$ there is a derivation $S_1 \underset{G_1}{\Rightarrow}^+ w$, or $S_2 \underset{G_2}{\Rightarrow}^+ w$ and vice versa . $L(G_3)$ is the language of the type 3 because G_3 is the grammar of the type 3.

(2) Let $G_4 = (N_1 \cup N_2, \Sigma, P_4, S_1)$ is the grammar of the type 3, whose set of rewriting rules P_4 is defined as follows:

   (a) if $A \to xB$ v P_1, then $A \to xB$ is in P_4
   (b) if $A \to x$ v P_1, then $A \to xS_2$ is in P_4
   (c) all rules from P_2 are in P_4.

   If $S_1 \underset{G_1}{\Rightarrow}^+ w$, then $S_1 \underset{G_4}{\Rightarrow}^+ wS_2$. Next if $S_2 \underset{G_2}{\Rightarrow}^+ x$, then $S_2 \underset{G_4}{\Rightarrow}^+ x$ and thus $S_1 \underset{G_4}{\Rightarrow}^+ wx$ for arbitrary w and x. From this follows $L(G_1) \cdot L(G_2) \subseteq L(G_4)$.
   Assume, that holds $S_1 \underset{G_4}{\Rightarrow}^+ w$. Because no rules in the form $A \to x$ from the set P_1 exist in P_4, we can write this derivation in the form $S_1 \underset{G_4}{\Rightarrow}^+ x \quad S_2 \underset{G_4}{\Rightarrow}^+ xy$, where $w = xy$, whereas all rules used in the derivation $S_1 \underset{G_4}{\Rightarrow}^+ xS_2$ have been derived according to (a) and (b). Then there has to be a derivation $S_1 \underset{G_1}{\Rightarrow}^+ x$ and $S_2 \underset{G_2}{\Rightarrow}^+ y$ and thus $L(G_4) \subseteq L(G_1) \cdot L(G_2)$. However, from this follows $L(G_4) = L(G_1) \cdot L(G_2)$.

(3) Let $G_5 = \{N_1 \cup \{S_5\}, \Sigma, P_5, S_5\}$, $S_5 \notin N_1$ and the set P_5 are created as follows:

   (a) if $A \to xB$ v P_1, then $A \to xB$ is in P_5
   (b) if $A \to x$ v P_1, then $A \to xS_5$ and $A \to x$ are in P_5
   (c) $S_5 \to S_1$ and $S_5 \to \epsilon$ are in P_5.

   Now we have to prove the statement :
   The derivation

$$S_5 \underset{G_5}{\Rightarrow}^+ x_1S_5 \underset{G_5}{\Rightarrow}^+ x_1x_2S_5 \underset{G_5}{\Rightarrow}^+ \dots \underset{G_5}{\Rightarrow}^+ x_1x_2\dots x_{n-1}S_5 \underset{G_5}{\Rightarrow}^+ x_1x_2\dots x_{n-1}x_n$$

exists if and only if there is the derivation

$$S_1 \underset{G_1}{\Rightarrow}^+ x_1,\ S_1 \underset{G_1}{\Rightarrow}^+ x_2, \dots, S_1 \underset{G_1}{\Rightarrow}^+ x_n.$$

We leave this proof as an exercise.
From this statement directly follows $L(G_5) = (L(G_1))^*$. $L(G_5)$ is the language of the type 3 because G_5 is the grammar of the type 3. □

3.3.2 Regular expressions

An usual notation for a representation of regular sets are regular expressions.

Definition 3.11 Regular expressions over Σ are recursively defined as follows:

(1) $\emptyset$ is a regular expression denoting the regular set $\emptyset$,

(2) ε is a regular expression denoting the regular set $\{\varepsilon\}$,

(3) a is a regular expression denoting the regular set $\{a\}$ for all $a \in \Sigma$,

(4) if p, q are regular expression denoting the regular sets P and Q, then

   (a) $(p + q)$ is regular expression denoting the regular set $P \cup Q$,
   (b) (pq) is regular expression denoting the regular set $P \cdot Q$,
   (c) (p^*) is regular expression denoting the regular set P^*.

(5) There are not other regular expressions over Σ.

Convention 3.1 We introduce the following conventions for the more readable notations of some regular expressions:

1. A regular expression p^+ stands for a regular expression pp^*.
2. We fix priorities of the operators: * and $^+$ (iteration – the most hight priority), $\cdot$ (concatenation), $+$ (alternative) to minimize a number of used brackets.

Example 3.15

1. 01 corresponds to $\{01\}$.
2. 0^* corresponds to $\{0\}^*$.
3. $(0 + 1)^*$corresponds to $\{0, 1\}^*$.
4. $(0 + 1)^*011$ stands for the set of strings over $\{0, 1\}$ ending by 011.
5. $(a + b)(a + b + 0 + 1)^*$ stands for the set of strings over $\{a, b, 0, 1\}$, which begin by symbol a or b.

We have defined the regular expression. We need a set of axioms to be able to derive identities over the regular expressions. Such set is *Kleene algebra*.

Definition 3.12

DEF

Klenee algebra contains non-empty set with two important constants 0 and 1, two binary operations + and · and unary operation *, which satisfy following axioms:

$a+(b+c)=(a+b)+c$	associativity of +	[A.1]
$a+b=b+a$	a commutativity of +	[A.2]
$a+a=a$	an idempotent +	[A.3]
$a+0=a$	0 is an identity for +	[A.4]
$a(bc)=(ab)c$	an association ·	[A.5]
$a1=1a=a$	1 is an identity for ·	[A.6]
$a0=0a=0$	0 is an annihilator pro ·	[A.7]
$a(b+c)=ab+ac$	distributivity from the left	[A.8]
$(a+b)c=ac+bc$	distributivity from the right	[A.9]
$1+aa^*=a^*$		[A.10]
$1+a^*a=a^*$		[A.11]
$b+ac\le c\Rightarrow a^*b\le c$		[A.12]
$b+ca\le c\Rightarrow ba^*\le c$		[A.13]

The symbol $\le$ in A.12 a A13 represents ordering which is defined as follow: $a\le b \overset{def}{\Longleftrightarrow} a+b=b$.

Example 3.16 Some examples of Kleene algebra

x+y

(1) A class 2^{Σ^*} of all subsets Σ^* with the constants $\emptyset$ and ε and the binary operations $\cup$, $\cdot$ and the unary operation *.

(2) A class of all regular subsets Σ^* with the constants $\emptyset$ and ε and the binary operations $\cup$, $\cdot$ and the unary operation *.

(3) A class of all binary relations over a set X with constants the empty relation, the identity relation and the operations $\cup$, the composition of relations and the reflexive and transitive closure of composition.

(4) Kleene algebras of matrixes.

Note 3.3 The axioms A.12 and A.13 can be used to replace by following equivalent axioms (a proof see in [3]):

$ac\le c\Rightarrow a^*c\le c$	[A.14]
$ca\le c\Rightarrow ca^*\le c$	[A.15]

There are some useful theorems of Kleene algebra which is possible to derive from the axioms.

(1)	$0^*=1$		
(2)	$1+a^*=a^*$		
(3)	$a^*=a+a^*$		
(4)	$a^*a^*=a^*$		
(5)	$a^{**}=a^*$		
(6)	$(a^*b)^*a^*=(a+b)^*$	"denesting rule"	[R.16]
(7)	$a(ba)^*=(ab)^*a$	"shifting rule"	[R.17]
(8)	$a^*=(aa)^*+a(aa)^*$		

Proof: To be short, we mention only a proof of the first two theorems, the others see in [3].

(1) $0^* = 1$: $0^* \overset{A.10}{=} 1 + 00^* \overset{A.7}{=} 1 + 0 \overset{A.4}{=} 1$.

(2) $1 + a^* = a^*$ – to be short, we do not mention an usage of A.1, A.2:

- (a) $1 + a^* \leq a^*$:
  - i. A.10: $a^* = 1 + aa^*$
  - ii. A.3: $a^* + a^* = 1 + aa^*$
  - iii. A.10: $1 + aa^* + a^* = 1 + aa^*$
  - iv. A.3: $1+1+aa^*+a^* = 1+aa^*$, otherwise $1+a^*+1+aa^* = 1 + aa^*$
  - v. def. $\leq$: $1 + a^* \leq 1 + aa^*$
  - vi. A.10: $1 + a^* \leq a^*$
- (b) $a^* \leq 1 + a^*$:
  - i. $1 + a^* = 1 + a^*$
  - ii. A.3: $1 + a^* + a^* = 1 + a^*$
  - iii. def. $\leq$: $a^* \leq 1 + a^*$
- (c) antisymmetry of $\leq$.

$\square$

Some properties of Kleene algebras are sometimes more easy to prove for particular examples of these algebras, e.g. for Kleene algebra of regular expression is possible to use theory of set. In the next parts we will directly work with Kleene algebra over regular sets.

Example 3.17 Let p, resp. q, denote the regular set P, resp. Q. Then $p+q$ denotes $P \cup Q$ and $q + p$ denotes $Q \cup P$. $P \cup Q = Q \cup P$ (commutativity of union of sets) $\Rightarrow p + q = q + p$.

3.3.3 Equations over regular expressions

Definition 3.13 An equation, which contains a variable and coefficients representing regular expressions is called *the equation over regular expressions.*

Example 3.18 Consider the equation

$$X = aX + b$$

over regular expressions over the alphabet $\{a, b\}$

$$X = aX + b$$

The solution of this equation is the regular expression $X = a^*b$.

Proof: $LS = a^*b$, $RS = a(a^*b) + b = a^+b + b = (a^+ + \epsilon)b = a^*b$. Not always there is a single solution of an equation over regular expressions.

$\square$

Example 3.19 Let $X = pX + q$ is the equation over regular expressions, where p, q are regular expressions and p denote the regular set P, such that $\epsilon \in P$. Then

$$X = p^*(q + r)$$

is the solution of this equation for an arbitrary language r (r does not have to correspond even to a regular set).
Proof:
$LS = p^*(q + r)$, $RS = p(p^*(q + r)) + q = pp^*(q + r) + q = p^*(q + r) + q = p^*(q + r)$. It is necessary to realize that $\epsilon \in P$.

$\square$

Usually we try to find "the least solution", so called *the least fix point* of the given equation.

Theorem 3.13 The least fix point of the equation $X = pX + q$ is:

$$X = p^*q$$

Proof:
$LS = p^*q$, $RS = pp^*q + q = (pp^* + \epsilon)q = p^*q$

$\square$

3.3.4 Systems of equations over regular expressions

Example 3.20 Let us consider the system of equations

$$\begin{aligned} X &= a_1X + a_2Y + a_3 \\ Y &= b_1X + b_2Y + b_3 \end{aligned}$$

The solution is:

$$\begin{aligned} X &= (a_1 + a_2b_2^*b_1)^*(a_3 + a_2b_2^*b_3) \\ Y &= (b_2 + b_1a_1^*a_2)^*(b_3 + b_1a_1^*a_3) \end{aligned}$$

Proof: We leave this prof as an exercise. $\square$

Definition 3.14 A system of equations over regular expressions is in *a standard form* with respect to variables $\Delta = \{X_1, X_2, ..., X_n\}$, if the system is in the form

$$\bigwedge_{i \in \{1,...,n\}} X_i = \alpha_{i0} + \alpha_{i1}X_1 + \alpha_{i2}X_2 + ... + \alpha_{in}X_n$$

where α_{ij} are some regular expressions over the alphabet Σ, $\Sigma \cap \Delta = \emptyset$.

Theorem 3.14 If the system of equations over regular expressions is in the standard form, then there exists the least fix point of this equations and there exists an algorithm for finding this fix point.

Proof: We express particular variables using a solution of the equation $X = pX + q$ as regular expressions with variables whose number is step by step decreased: For example we express the variable X_n as the regular expression over Σ and $X_1, ..., X_{n-1}$ from the equation for X_n. We substitute for X_n into the equation for X_{n-1} and after it we repeat this process. It is possible (but not necessary) to modify this order.

□

Example 3.21 Solve the following system of equations over regular expressions:

$$\begin{array}{lrcl} (1) & X_1 & = & (01^* + 1)X_1 + X_2 \\ (2) & X_2 & = & 11 + 1X_1 + 00X_3 \\ (3) & X_3 & = & \varepsilon + X_1 + X_2 \end{array}$$

Solution:

1. We substitute the expression for X_3 from (3) into (2). We obtain the system:

$$\begin{array}{lrcl} (4) & X_1 & = & (01^* + 1)X_1 + X_2 \\ (5) & X_2 & = & 11 + 1X_1 + 00(\varepsilon + X_1 + X_2) = 00 + 11 + (1 + 00)X_1 + 00X_2 \end{array}$$

2. We express from (4) the variable X_1 using the solution of the equation $X = pX + q$ (the theorem 3.13):

$$\begin{array}{lrcl} (6) & X_1 & = & (01^* + 1)^* X_2 = (0 + 1)^* X_2 \end{array}$$

3. By the substitution into (5) we get:

$$\begin{array}{lrcl} (7) & X_2 & = & 00 + 11 + (1 + 00)(0 + 1)^* X_2 + 00X_2 = \\ & & = & 00 + 11 + (1 + 00)(0 + 1)^* X_2 \end{array}$$

4. By the computation X_2 as solution of the equation $X = pX + q$ we get:

$$\begin{array}{lrcl} (8) & X_2 & = & ((1 + 00)(0 + 1)^*)^*(00 + 11) \end{array}$$

5. By the substitution into (6) we get:

$$\begin{array}{lrcl} (9) & X_1 & = & (0 + 1)^*((1 + 00)(0 + 1)^*)^*(00 + 11) = (0 + 1)^*(00 + 11) \end{array}$$

6. By the substitution into (3) we get:

$$\begin{array}{lrcl} (10) & X_3 & = & \varepsilon + (0 + 1)^*(00 + 11) + ((1 + 00)(0 + 1)^*)^*(00 + 11) = \\ & & = & \varepsilon + ((0 + 1)^* + ((1 + 00)(0 + 1)^*)^*)(00 + 11) = \\ & & = & \varepsilon + (0 + 1)^*(00 + 11) \end{array}$$

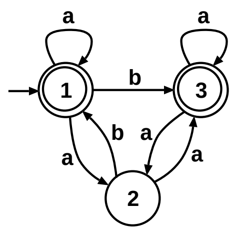
Theorem 3.15 Any language generated by the grammar of the type 3 is a regular set: $\mathcal{L}_3 \subseteq \mathcal{L}_R$

Proof:

1. Let $L \in \mathcal{L}_3$ is an arbitrary language of the type 3. We already know that, this language is accepted by a finite automaton $M = (Q, \Sigma, \delta, q_0, F)$. Let $Q = \{q_0, q_1, ..., q_n\}$.
2. We create a system of equations with variables $X_0, X_1, ..., X_n$ in the standard form. The equation for X_i describes a set of strings accepted by M in the case when the final state is Q_i.
3. By the solution of this system we obtain the regular expression for the variables X_0 which represents the language L.

$\square$

Example 3.22 The language L describes a regular expression, which is solution of this system for the variable X_1.

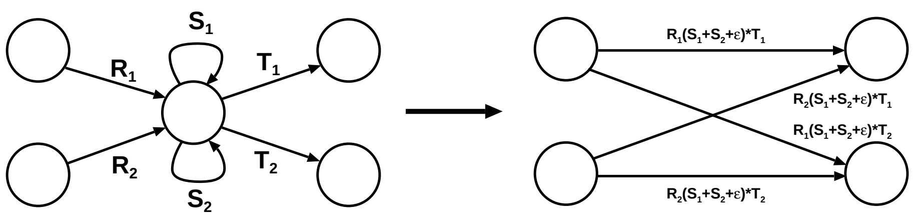


$$\begin{array}{lcllll}
X_1 & = & \varepsilon + & aX_1 + & aX_2 + & bX_3 \\
X_2 & = & & bX_1 + & & aX_3 \\
X_3 & = & \varepsilon + & & aX_2 + & aX_3
\end{array}$$

Note 3.4 An alternative conversion of a finite automaton to the corresponding regular expression.

A regular transition graph is a generalization of a finite automaton. It allows to use the set of initial states and a regular expression on the edge of transitions.

Any transition graph is possible to convert to *the regular transition graph with one transition*, from which we can obtain the searched regular expression. We establish a new initial and final state, which we connect with original initial and final states using ϵ transitions. After it we step by step remove all original states as is illustrated below:



3.4 Conversion of a regular expression to the finite automaton

Now we show a direct conversion of a regular expression to the corresponding deterministic automaton. In the first step we convert a regular expression to so-called extended finite automata that we convert to deterministic finite automata in the second step.

DEF

Definition 3.15 *An extended finite automaton* (EFA) is 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$, where

(a) Q is a finite set of states,

(b) Σ is a finite input alphabet,

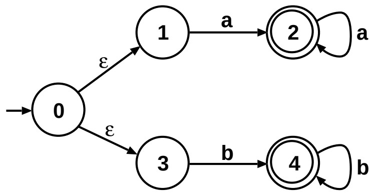
(c) δ is a mapping $Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q$,

(d) $q_0 \in Q$ is a initial state,

(e) $F \subseteq Q$ is a set of final states.

x+y

Example 3.23 $M = (\{0, 1, 2, 3, 4\}, \{a, b\}, \delta, 0, \{2, 4\})$



$$L(M) = aa^* + bb^* = a^+ + b^+$$

DEF

Definition 3.16 A crucial part of an algorithm for conversion of EFA to DFA is a calculation of the function which for the given state determines a set of all states which are reachable by using of ε edges of the transition diagram of the transition function δ. Denote this function by ε-closure:

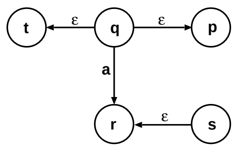
$$\varepsilon\text{-closure}(q) = \{p \mid \exists w \in \Sigma^* : (q, w) \overset{*}{\vdash} (p, w)\}$$

We generalize the function ε-closure in such way that an argument of the function could be a set $T \subseteq Q$:

$$\varepsilon\text{-closure}(T) = \bigcup_{s \in T} \varepsilon\text{-closure}(s)$$

x+y

Example 3.24



$$\varepsilon\text{-closure}(\{q, r, s\}) = \{p, q, r, s, t\}$$

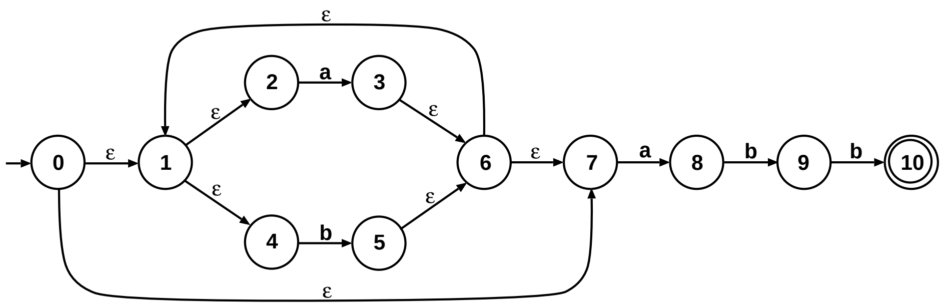
For a practical computation of ε-closure we can establish for example a relation $\xrightarrow{\varepsilon}$ on a set Q as follows:

$$\forall q_1, q_2 \in Q : q_1 \xrightarrow{\varepsilon} q_2 \overset{def}{\Longleftrightarrow} q_2 \in \delta(q_1, \varepsilon)$$

Then ε-closure$(q) = \{q \in Q \mid q \xrightarrow{\varepsilon}{}^{*} q\}$.

We can use Warshall algorithm to compute ε-closure. We complete a diagonal by ones (the reflexive closure) and than from the relevant rows of the matrix the resulted set could be specified.

Example 3.25



$$\varepsilon\text{-closure}(3) = \{3, 6, 7, 1, 2, 4\}$$
$$\varepsilon\text{-closure}(\{1, 0\}) = \{0, 1, 2, 4, 7\}$$

Algorithm 3.6
The conversion of an extended finite automaton to a deterministic finite automat.

Input: An extended finite automaton $M = (Q, \Sigma, \delta, q_0, F)$.

Output: A deterministic finite automat $M' = (Q', \Sigma, \delta', q'_0, F')$, $L(M) = L(M')$.

Method:

1. $Q' := 2^Q \setminus \{\emptyset\}$.
2. $q'_0 := \varepsilon$-closure(q_0).
3. $\delta' : Q' \times \Sigma \to Q' \cup \{nedef\}$ is computed as follows:
   - Let $\forall T \in Q', a \in \Sigma : \overline{\delta}(T, a) = \bigcup_{q \in T} \delta(q, a)$.
   - Then for each $T \in Q'$, $a \in \Sigma$:
     (a) if $\overline{\delta}(T, a) \neq \emptyset$, then $\delta'(T, a) = \varepsilon$-closure$(\overline{\delta}(T, a))$,
     (b) otherwise $\delta'(T, a) = nedef$.
4. $F' := \{S \mid S \in Q' \wedge S \cap F \neq \emptyset\}$.

Example 3.26 We apply the algorithm 3.6 on the automaton from the exercise 3.25

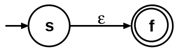
1. Let A be the initial state, then $A = \varepsilon\text{-closure}(0) = \{0, 1, 2, 4, 7\}$.
2. $\delta'(A, a) = \varepsilon\text{-closure}(\{3, 8\}) = \{1, 2, 3, 4, 6, 7, 8\} = B$.
3. $\delta'(A, b) = \varepsilon\text{-closure}(\{5\}) = \{1, 2, 4, 5, 6, 7\} = C$.
4. $\delta'(B, a) = \varepsilon\text{-closure}(\{3, 8\}) = B$.
5. $\delta'(B, b) = \varepsilon\text{-closure}(\{5, 9\} = \{1, 2, 4, 5, 6, 7, 9\} = D$.
6. $\delta'(C, a) = \varepsilon\text{-closure}(\{3, 8\}) = B$.
7. $\delta'(C, b) = \varepsilon\text{-closure}(\{5\}) = C$.
8. $\delta'(D, a) = \varepsilon\text{-closure}(\{3, 8\}) = B$.
9. $\delta'(D, b) = \varepsilon\text{-closure}(\{5, 10\} = \{1, 2, 4, 5, 6, 7, 10\} = E$.
10. $\delta'(E, a) = \varepsilon\text{-closure}(\{3, 8\}) = B$.
11. $\delta'(E, b) = \varepsilon\text{-closure}(\{5\}) = C$.
12. A set of final states $F = \{E\}$.

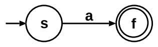
Algorithm 3.7
The conversion of a regular expression to an extended finite automaton.

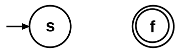
Input: A regular expression r describing the regular set R over Σ.

Output: An extended finite automaton M for which holds $L(M) = R$.

Method: 1. We decompose r into its primitive component according to recursive definition of regular sets (expressions).

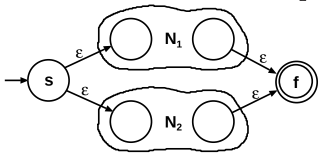
2. (a) We construct for the expression ε the automaton: 

   (b) We construct for the expression $a, a \in \Sigma$ the automaton: 

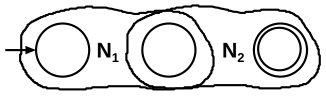
   (c) We construct for the expression $\emptyset$ the automaton: 

   (d) Let N_1 is the automaton accepting the language which is specified by the expression r_1 and let N_2 is the automaton accepting the language which is specified by athe expression r_2.

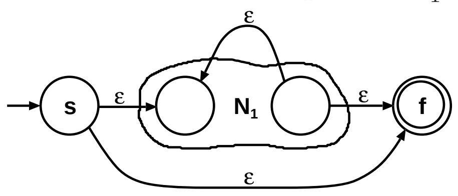
      i. We construct for the expression $r_1 + r_2$ the automaton:



      ii. We construct for the expression $r_1 r_2$ the automaton:



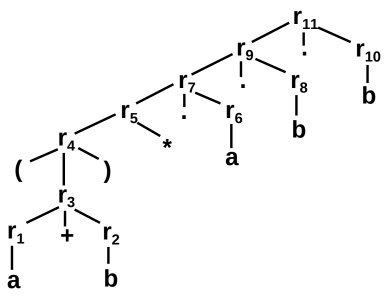
iii. We construct for the expression r_1^* the automaton:

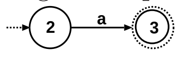


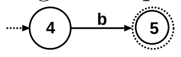
Example 3.27 Construct an extended finite automaton for the regular expression $(a+b)^*abb$:

x+y

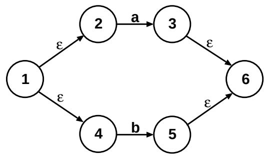
1. The decomposition of given regular expression is expressed by the tree:



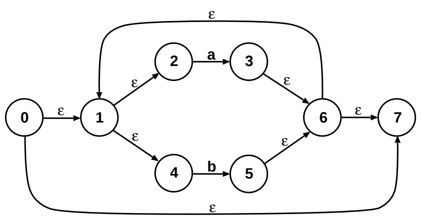
2. (a) The regular expression $r_1 = a$ corresponds to the automaton N_1: 

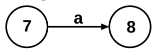
   (b) The regular expression $r_2 = b$ corresponds to the automaton N_2: 

   (c) The regular expression $r_1 + r_2$ corresponds to the automaton N_3:

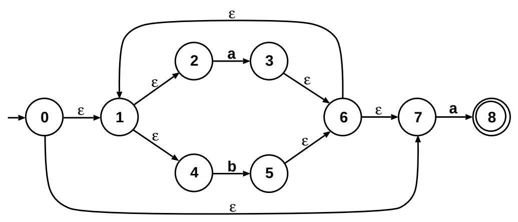


   (d) The automaton N_4 for $r_4 = (r_3)$ is same as the automaton N_3, so we construct directly the automaton N_5 for the expression $r_5 = r_4^* = (a+b)^*$:



   (e) The regular expression $r_6 = a$ corresponds to the automaton N_6: 

(f) The regular expression $r_7 = r_5 r_6$ corresponds to the automaton N_7:



(. . .)
We continue until we obtain the automaton from the exercise 3.25.

The conversion of a regular expression to the extended finite automaton creates a lot of inner states and therefore this conversion is usually followed by *minimization of DFA* (the algorithm 3.5).

Example 3.28 Decide, whether the deterministic finite automaton from the exercise 3.26 is in the reduce form.

Σ

Comparing regular grammars, finite automata and regular expressions we can summarize that

1. grammars of the type 3 (right/left regular grammars, right/left linear grammars),
2. (extended/nondeterministic/deterministic) finite automata and
3. regular expressions

have the equivalent expressive power.

$$\begin{tikzcd}
 & \text{grammars of the type 3} \arrow[dl, bend right] \arrow[dr, bend left] & \\
\text{finite automata} \arrow[ur, bend right] \arrow[rr, bend left] & & \text{regular expressions} \arrow[ul, bend left] \arrow[ll, bend left]
\end{tikzcd}$$

3.5 Properties of regular languages

Analogous to the other classes of languages, we will examine the following properties of regular languages:

(1) structural properties,

(2) closure properties and

(3) decidable problems for regular languages.

3.5.1 Structural properties of regular languages

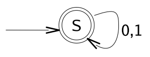
Theorem 3.16 Any finite language is a regular language.

Proof: Let $L = \{w_1, w_2, \ldots, w_n\}$, $w_i \in \Sigma^*$. Then $L = L(G)$, where the grammar $G = (\{S\}, \Sigma, \{S \to w_1, S \to w_2, \ldots, S \to w_n\}, S)$. G is obviously the grammar of the type 3.

□

It is clear that the opposite implication does not hold:

Example 3.29 Construct a grammar of the type 3 that generates the language $\{0,1\}^*$.
Solution:



$$\Rightarrow \quad G = (\{S\},\ \{0,1\},\ \{S \to \varepsilon, S \to 0S, S \to 1S\},\ S)$$

Pumping lemma

As far as we are interesting in the the property, whether a language belongs to a given class of languages, there is useful theorem so called *pumping lemma*. Informally the pumping lemma says that in any enough long sentence of any infinite regular language there exist rather short sequence (a substring of the sequence) which is possible to omit or repeat arbitrarily-times and always we obtain strings (sentences) which belong to the given languages.

Theorem 3.17 Let L is an infinity regular language. Then there exists integer constant $p \geq 0$ (so called the pumping constant) for which:

$$\begin{array}{lll} w \in L \ \wedge \ |w| \geq p & \Rightarrow & w = x\ y\ z \ \wedge \\ & & 0 < |y| \leq p \ \wedge \\ & & x\ y^i\ z \in L \text{ pro } i \geq 0. \end{array}$$

Proof: Let $L = L(M)$, $M = (Q, \Sigma, \delta, q_0, F)$ be a finite automaton where $|Q| = n$ and let $p = n$. If $w \in L$ and $|w| \geq n$, then M accepts the string w by "passing" *at least* $n+1$ *configurations* of M and hence at least two of these configurations containing equal state, thus:

$$(q_0, w) = (q_0, xyz) \overset{*}{\vdash} (r, yz) \overset{k}{\vdash} (r, z) \overset{*}{\vdash} (q_F, \varepsilon),\ q_F \in F$$

for some state $r \in Q$ and some constant k, $0 < k \leq n$.

Then of course exists the sequence of configurations:

$$\begin{array}{ll} (q_0, xy^iz) & \overset{*}{\vdash} (r, y^iz) \\ & \overset{+}{\vdash} (r, y^{i-1}z) \\ & \vdots \\ & \overset{+}{\vdash} (r, z) \\ & \overset{*}{\vdash} (q_F, \varepsilon) \end{array}$$

It indeed means that $xy^iz \in L(M)$, and not only for $i > 0$, but even for case when $i = 0$: $(q_0, xz) \overset{*}{\vdash} (r, z) \overset{*}{\vdash} (q_F, \epsilon),\ q_F \in F$.

□

The theorem 3.17 is often used to prove that given languages is not regular.

$\boxed{\texttt{x+y}}$

Example 3.30 Prove that the language $L = \{0^n1^n \mid n \geq 1\}$ is not regular.
Proof: Let us suppose that L is a regular language. Then according to the theorem 3.17 the sentence 0^k1^k can be written for enough large k in the form $xyz = 0^k1^k$, $0 < |y| \leq k$, and holds that $xy^iz \in L$ pro $i \geq 0$.
There exist 3 possibilities for a searched substring y:

$$0\underbrace{0\,0\,.\,.}_{y}\underbrace{.\,0\ \ 1\,1}_{y}\underbrace{1\ldots}_{y}1$$

$y \in \{0\}^+$	but then $xy^iz \notin L$ — different number of $0's$ and $1's$
$y \in \{1\}^+$	but then $xy^iz \notin L$ — different number of $0's$ and $1's$
$y \in \{0\}^+ \cdot \{1\}^+$	but then $xy^iz \notin L$ (not all $0's$ go before all $1's$)

Hence the substring y can not be chosen and thus $L \notin \mathcal{L}_3$.

□

Example 3.31 Prove that the language $L = \{a^q \mid q$ is prime a number$\}$ is not the regular language.
Proof: Let us suppose that L is a regular language.Then according to the theorem 3.17 there exist p such that any sentence $a^q \in L$, where $q \geq p$, can be written in the form $a^q = xyz$, $0 < |y| \leq p$, and $xy^iz \in L$ holds for $i \geq 0$.
Choose a prime number r greater than p. According the pumping lemma $a^r \in L$ and $|a^r| = r \geq p$ implies $a^r = xyz$, $0 < |y| \leq p$, and $xy^iz \in L$ pro $i \geq 0$.
Let $|y| = k$ and choose $i = r + 1$. But then $|xy^{r+1}z| = |xyz| + |y^r| = r + r.k = r.(k + 1)$, which is not the prime number and hence $xy^iz \notin L$. We obtain a controversy.

□

3.5.2 Myhill-Nerode theorem

The Myhill-Nerode theorem characterizes some fundamental relation between finete automata over the alphabet Σ and an equivalence relation on strings from Σ^*. The theorem describe some necessary and sufficient conditions for regular languages. It means that we can use this theorem to prove weather a given language is or is not regular. Moreover it provides a formal base for elegant proof of existence of unique (up to isomorphism) minimal DFA for the given finite automaton.

Right congruence and prefix equivalence

Let us recall that an *equivalence relation* $\sim$ is a binary relation, which is *reflexive, symmetric, and transitive* . An index of equivalence $\sim$ is a number of classes of the decomposition $\Sigma^*/\sim$. If there is infinitely many classes, we define this index as ∞.

DEF

Definition 3.17 Let Σ be an alphabet and $\sim$ be an equivalence on Σ^*. The equivalence $\sim$ is a *right congruence* (is invariant from the right) if for each $u, v, w \in \Sigma^*$

$$u \sim v \Longrightarrow uw \sim vw$$

◁

Theorem 3.18 The equivalence $\sim$ on Σ^* is the right congruence if and only if for each $u, v \in \Sigma^*$, $a \in \Sigma$ the implication $u \sim v \Longrightarrow ua \sim va$ holds.

Proof: "$\Rightarrow$" is trivial, "$\Leftarrow$" is easy to prove by induction with respect to the length of w.

□

DEF

Definition 3.18 Let L be any (not necessarily regular) language over an alphabet Σ. On the set Σ^* we define relation $\sim_L$ called *the prefix equivalence* for L as follows:

$$u \sim_L v \overset{def}{\Longleftrightarrow} \forall w \in \Sigma^* : uw \in L \Longleftrightarrow vw \in L$$

!

The Myhill-Nerode theorem

Theorem 3.19 Let L be a language over Σ. Then the following statements are equivalent:

1. L is the language accepted by a deterministic finite automaton.
2. L is the union of some classes of the decomposition determined by the right congruence with the finite index on Σ^*.
3. the relation $\sim_L$ has the finite index.

Proof: We prove the following implications:

(a) $1 \Rightarrow 2$

(b) $2 \Rightarrow 3$

(c) $3 \Rightarrow 1$

Let us note that the statement of theorem then follows from definition of equivalence ($a \Leftrightarrow b \overset{def}{\Longleftrightarrow} a \Rightarrow b \wedge b \Rightarrow a$) and from the basic tautology $((a \Rightarrow b \wedge b \Rightarrow c) \Rightarrow a \Rightarrow c)$.

(a) The proof of the implication $1 \Rightarrow 2$

If L is accepted by DFA, then L is the union of some classes of decomposition determined by the right congruence on Σ^* with the finite index. For DFA $M = (Q, \Sigma, \delta, q_0, F)$ we establish *a generalize transition function* $\hat{\delta} : Q \times \Sigma^* \to Q$ such that $\forall q_1, q_2 \in Q, w \in \Sigma^* : \hat{\delta}(q_1, w) = q_2 \Leftrightarrow (q_1, w) \overset{*}{\underset{M}{\vdash}} (q_2, \varepsilon)$.
For the given L accepted by the finite automaton M we construct the relation $\sim$ with the required properties:

(1) Let $M = (Q, \Sigma, \delta, q_0, F)$ a δ be a total function.

(2) Let $\sim$ be a binary relation on Σ^* such that $u \sim v \iff \hat{\delta}(q_0, u) = \hat{\delta}(q_0, v)$.

(3) We show that $\sim$ has the required properties:

i. $\sim$ is *an equivalence*: it is reflexive, symmetric and transitive.

ii. $\sim$ has *a finite index*: the number of equivalent classes can not be greater than the number of states of the automaton.

iii. $\sim$ is *a right congruence*: Let $u \sim v$ and $a \in \Sigma$. Then $\hat{\delta}(q_0, ua) = \delta(\hat{\delta}(q_0, u), a) = \delta(\hat{\delta}(q_0, v), a) = \hat{\delta}(q_0, va)$ and hence $ua \sim va$.

iv. L is *an union of some classes* $\Sigma^* \backslash \sim$: The union of classes which correspond to the states from the set F.

(b) The proof of the implication $2 \Rightarrow 3$

If there exists a relation $\sim$ satisfying condition 2, then $\sim_L$ has a finite index.

Let $u \sim v$. We show that also holds $u \sim_L v$, it means $\forall w \in \Sigma^* : uw \in L \Leftrightarrow vw \in L$. We know that $uw \sim vw$ and since L is an union of some classes of decomposition $\Sigma^* \backslash \sim$, also holds $uw \in L \Leftrightarrow vw \in L$. Thus we know that $\sim \subseteq \sim_L$ (it means $\sim_L$ is the largest right congruence with the given properties). Any classes of $\sim$ is contained in some class $\sim_L$. The index $\sim_L$ can not be greater than the index $\sim$. $\sim$ has a finite index and hence also $\sim_L$ has a finite index.

(c) The proof of the implication $3 \Rightarrow 1$

If $\sim_L$ has a finite index, then L is accepted by some finite automaton.
We construct $M = (Q, \Sigma, \delta, q_0, F)$ accepting L:

(1) $Q = \Sigma^* \backslash \sim$ (states are the equivalence classes of the decomposition of Σ^* with respect to the relation $\sim$),

(2) $\forall a \in \Sigma^*, u \in \Sigma : \delta([u], a) = [ua]$,

(3) $q_0 = [\varepsilon]$,

(4) $F = \{[x] \mid x \in L\}$.

The mentioned construction *is correct*, it means that $L = L(M)$: We can prove by induction with respect to a length of string v that $\forall v \in \Sigma^* : \hat{\delta}([\varepsilon], v) = [v]$ holds. Moreover $v \in L \Longleftrightarrow [v] \in F \Longleftrightarrow \hat{\delta}([\varepsilon], v) \in F$.

$\square$

As it has been mentioned, we can use the Myhill-Nerode theorem to prove weather given language is regular (or is not regular). The following exercise shows such usage.

Example 3.32 Prove that *a language* $L = \{a^n b^n \mid n \geq 0\}$ *is not regular.*

Proof: No strings $\varepsilon, a, a^2, a^3, ...$ are $\sim_L$-equivalent, since $a^i b^i \in L$, but $a^i b^j \notin L$ for $i \neq j$. Hence the relation $\sim_L$ has infinitely many classes (infinite index). According to the Myhill-Nerode theorem L can not be accepted by any finite automaton.

$\square$

The next use of the Myhill-Nerode theorem is elegant proof of existence of a minimal determinist finite automaton for the given regular language (expression).

Theorem 3.20 The number of states of an arbitrary minimal DFA accepting L is equal to the index of $\sim_L$. (This DFA exists if and only if $\sim_L$ has a finite index.)

Proof: Any DFA (let consider the DFA without unreachable states) defines a certain right congruence with the finite index and viceversa. If L is a regular language then $\sim_L$ is the largest right congruence with the finite index such that L is an union of some classes of corresponding decompositions. The finite automaton which correspond to $\sim_L$ (see the proof $3 \Rightarrow 1$ from the Myhill-Nerode theorem), is thus the minimal finite automaton accepting L.

$\square$

3.5.3 Closure properties of the regular languages

Theorem 3.21 The class of regular languages *is closed* (among others) under the operations $\cup$ (union), $\cdot$ (concatenation) a * (iteration).

Proof: It follows from definition of regular sets and from equivalence of regular sets and regular languages. $\square$

Theorem 3.22 The class of regular languages is *the Boolean algebra.*

Proof:

Denote by $\mathcal{L}_3$ the class of all regular languages over an alphabet Σ. We show that the algebraic structure $< \mathcal{L}_3, \cup, \cap, \ ', \Sigma^*, \emptyset >$ is the Boolean algebra.

1. Clearly $\Sigma^* \in \mathcal{L}_3$ and $\emptyset \in \mathcal{L}_3$. These languages are obviously "the largest element(1)", and "the smallest element (0) " of the consideration algebra.
2. The closure under union ($\cup$) is trivial.
3. We prove the closure under complement over an alphabet Δ, $\Delta \subseteq \Sigma$. Let us construct *a totaly defined* finite automaton M accepting the language L.

   $$M = (Q, \Delta, \delta, q_0, F)$$

   such that $L = L(M)$. Then the finite automaton M'

   $$M' = (Q, \Delta, \delta, q_0, Q \setminus F)$$

   clearly accepts the language $\Delta^* \setminus L$ (the complement L in Δ^*).

   The complement with respect to Σ^* can be expressed as:

   $$\overline{L} = L(M') \cup \Sigma^*(\Sigma \setminus \Delta)\Sigma^*$$

   which is according to the theorem 3.18 a regular language.
4. The closure under the intersection follows from de Morgan laws:

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

   and hence $L_1, L_2 \in \mathcal{L}_3 \Rightarrow L_1 \cap L2 \in \mathcal{L}_3$.

□

Theorem 3.23 Let $L \in \mathcal{L}_3$ and $L^R = \{w^R \mid w \in L\}$. Then $L^R \in \mathcal{L}_3$.

Proof: Let M be a finite automaton such that $L = L(M)$. We can construct the finite automaton M' such that $L(M') = L^R$.
We leave the construction of M' as an exercise.

□

3.5.4 Decidable problems of the regular languages

This subsection is concerned with the following fundamental decidable problems of regular languages:
The problem of *non-emptiness*: $L \neq \emptyset$?,
The problem of *membership*: $w \in L$? and
The problem of *equivalence*: $L(G_1) = L(G_2)$?

Theorem 3.24 The problem of *non-emptiness* and the problem of *membership* are decidable in the class $\mathcal{L}_3$.

Proof: For the language $L \in \mathcal{L}_3$ we construct the deterministic finite automaton M, such that $L = L(M)$:

$$M = (Q, \Sigma, \delta, q_0, F)$$

non-emptiness: $L(M) \neq \emptyset \Longleftrightarrow \exists q \in Q : (q \in F \wedge q \text{ is reachable from } q_0)$

membership: $w \in L \Leftrightarrow (q_0, w) \overset{*}{\vdash} (q, \varepsilon) \wedge q \in F$The both predicates are easily decidable. □

Theorem 3.25 Let $L_1 = L(G_1)$ and $L_2 = L(G_2)$ are two languages generated by regular grammars G_1 and G_2. Then the problem of the*equivalence*, it means $L(G_1) = L(G_2)$ is decidable.

Proof: Let $M_1 = (Q_1, \Sigma_1, \delta_1, q_0^1, F_1)$, resp. $M_2 = (Q_2, \Sigma_2, \delta_2, q_0^2, F_2)$ are the finite automata accepting the language L_1, resp. L_2 such that $Q_1 \cap Q_2 = \emptyset$. We construct the finite automaton M in this way:

$$M = (Q_1 \cup Q_2, \Sigma_1 \cup \Sigma_2, \delta_1 \cup \delta_2, q_0^1, F_1 \cup F_2)$$

and compute the relation $\equiv$ on $Q_1 \cup Q_2$ for the automaton M that specifies undistinguishable states of M.
Then $L(G_1) = L(G_2) if and only if q_0^1 \equiv q_0^2$

□

Regular languages, involving all finite languages, represent the class of formal languages with very high number of applications, especially because of their decision power. Regular expressions, grammars of the type 3 and nondeterministic and deterministic finite automata belong to most widespread means of their specification. Regular expressions create one of the instance of Kleene algebra, that enables to solve standardized equation systems over regular expressions. Conversion algorithms between specification means are often the base for special language processors, for instance constructors of lexical analyzers of compilers.

3.6 Exercises

Exercise 3.6.1 For the nondeterministic finite automaton

$$M = (\{q_0, q_1, q_2, q_3, q_F\}, \{1, 2, 3\}, \delta, q_0, \{q_F\}),$$

where the transition function δ is defined by the following table:

	$\mathbf{\Sigma}$		
Q	1	2	3
$\mathbf{q_0}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$	$\{q_0, q_3\}$
$\mathbf{q_1}$	$\{q_1, q_F\}$	$\{q_1\}$	$\{q_1\}$
$\mathbf{q_2}$	$\{q_2\}$	$\{q_2, q_F\}$	$\{q_1\}$
$\mathbf{q_3}$	$\{q_3\}$	$\{q_3\}$	$\{q_3, q_F\}$
$\mathbf{q_F}$	$\emptyset$	$\emptyset$	$\emptyset$

construct a finite deterministic automaton M' such that $L(M') = L(M)$.

Exercise 3.6.2 Let $L_1 = L(M_1)$ and $L_2 = L(M_2)$ are the languages accepting by the finite automata $M_1 = (Q_1, \Sigma_1, \delta_1, q_0^1, F_1)$ and $M_2 = (Q_2, \Sigma_2, \delta_2, q_0^2, F_2)$. With an analogy to the theorem 3.12 construct the automata M_3, M_4 a M_5, such that:

$$L(M_3) = L_1 \cup L_2,\ L(M_4) = L_1 \cdot L_2 \text{ a } L(M_5) = L_1^*.$$

Exercise 3.6.3 For the right linear grammar, which contains the rules

$$\begin{array}{lcl} A & \rightarrow & B \mid C \\ B & \rightarrow & 0B \mid 1B \mid 011 \\ C & \rightarrow & 0D \mid 1C \mid \epsilon \\ D & \rightarrow & 0C \mid 1D \end{array}$$

construct a right regular grammar, which generates same language. The nonterminal A is the start symbol of the grammar.

Exercise 3.6.4 Construct a regular grammar, which generates the language accepting by the automaton M from the exercise 3.6.1.

Exercise 3.6.5 Construct a finite automaton, which accepts the language generated by the grammar from the exercise 3.6.3.

Exercise 3.6.6 On the basis of the algorithm, which is "inverse" to the algorithm 3.3, construct a left regular grammar for the languages of real numbers for programming languages. The automat accepting this language is mentioned in the exercise 3.1.

Exercise 3.6.7 Following the grammar from the exercise 2.4.5 construct a finite deterministic automaton which accepts the languages of numbers for the programming language Pascal.

Exercise 3.6.8 The grammar $G = (\{S, A_n, A_{n-1}, \ldots, A_0\}, \{a, b\}, P, S)$ with the rules

$$\begin{array}{rcl} S & \rightarrow & A_n \\ A_n & \rightarrow & A_{n-1}A_{n-1} \\ A_{n-1} & \rightarrow & A_{n-2}A_{n-2} \\ \vdots & & \\ A_2 & \rightarrow & A_1A_1 \\ A_1 & \rightarrow & A_0A_0 \\ A_0 & \rightarrow & a \mid b \end{array}$$

describes for $n \geq 0$ a regular language over the alphabet $\{a, b\}$.

(a) Specify the languages $L(G)$ by a grammar of the type 3 grammar.

(b) Transform this grammar to a regular grammar.

(c) Transform this grammar to an equivalent NFA an DFA.

(d) Describe the language specified by the grammar G in the form of a regular expression.

(e) For the obtained regular expression construct an equivalent extended finite automaton and an equivalent deterministic finite automaton.

(f) Compare DFA, which was constructed in (c) and (e). Show that the automat accepting the language $L(G)$ do not need more than $2^n + 1$ states.

Chapter 4

Context-free languages

15:00

The aim of this chapter is deep understanding terms and facts of the context-free grammars theory, which are a fundament of modern applications in programming languages compilers, verification and formal specification of computer systems. The chapter introduces basic transformations of context-free grammars preserving equivalence of the context-free languages and leads to the development of algorithms and proofs of these transformations.

Definition 4.1 A context-free grammar G is a 4-tuple $G = (N, \Sigma, P, S)$

(1) N is a finite set of non-terminal symbols

(2) Σ is a finite set of terminal symbols

(3) P is a finite set of rules (rewriting rules, productions) $A \to \alpha$, $A \in N$ and $\alpha \in (N \cup \Sigma)^*$

(4) $S \in N$ is the start symbol of the grammar.

Using the word *grammar* we will mean a context-free grammar in this chapter.

Example 4.1 The grammar $G = (\{S, A, B\}, \{a, b, c, d\}, P, S)$, where P contains the rules

$$\begin{array}{rcl} S & \to & AB \\ A & \to & aAb \mid ab \\ B & \to & cBd \mid cd \end{array}$$

generates the context-free language $L(G) = \{a^n b^n c^m d^m \mid \ n \geq 1,\ m \geq 1\}$

The language $L(G)$ is a concatenation of two context-free languages generated by the grammars

$$\begin{array}{rcl} G_1 & = & (\{A\}, \{a, b\}, \{A \to aAb, A \ \to ab\}, A) \\ G_2 & = & (\{B\}, \{c, d\}, \{B \to cBd, B \to cd\}, B) \end{array}$$

Let us remark, that the language $L(G_1) = \{a^n b^n \ \mid n \leq 1\}$ and therefore also the language $L(G)$ is not regular, because a finite state automaton cannot "count" the same number of occurrence of symbols a and b for arbitrary n.

On the other hand, the language $\{a^n b^n c^n d^n \ \mid \ n \leq 1\}$ and even the language $\{a^n b^n c^n \mid n \leq 0\}$ is not context-free.

4.1 Derivation tree

An important means of graphical representation of a sentence structure (it's derivation) is a rooted, vertex-weighted tree, which is called a derivation tree or a syntax tree.

Let us remind that a tree is an oriented acyclic graph with these characteristics:

(1) There is only one vertex, so-called *root* with no incoming edges.

(2) There is exactly one edge coming in the rest of vertices.

Vertices with no outcoming edges are called *leaf vertices* (leaves). When drawing a tree we usually follow the convention: the root is on the very top, all the edges are oriented down. If we follow this convention, we can omit arrows, which denote the orientation of the edges. Vertex 1 on the

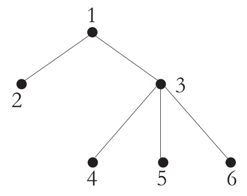


Figure 4.1: Example of a tree

figure 4.1 is a root, vertices 2,4,5,6 are the leafs.

DEF

Definition 4.2 Let δ be a sentence or a sentential form generated by a grammar $G = (N, \Sigma, P, S)$ and let $S = \nu_0 \Rightarrow \nu_1 \Rightarrow \nu_2 \ldots \Rightarrow \nu_k = \delta$ be it's derivation in G. *A derivation tree* corresponding to the derivation is a labeled tree with following characteristics:

(1) Vertices of the derivation tree are (labeled by) symbols from the set $N \cup \Sigma$; the root of the tree is labeled by the start symbol S.

(2) To a direct derivation $\nu_{i-1} \Rightarrow \nu_i,\ i = 0, 1, \ldots, k$, where

$$\begin{aligned}&\nu_{i-1} = \mu A \lambda,\ \mu, \lambda \in (N \cup \Sigma)^*,\ A \in N\\&\nu_i = \mu\alpha\lambda \text{ a}\\&A \rightarrow \alpha,\ \alpha = X_1 \ldots X_n \text{ is a rule from } P,\end{aligned}$$

corresponds exactly n edges $(A, X_j), 1 \leq j \leq n$ coming out of the vertex A, which are sorted from left to right in the order $(A, X_1), \ldots (A, X_n)$.

(3) Labeling of the derivation tree leaves from left to right forms a sentential form or a sentence δ (corollary of (1) and (2)).

Example 4.2 By the grammar from example 4.1 we can generate the string *aabbcd*, e.g. by the derivation:

$$S \Rightarrow AB \Rightarrow aAbB \Rightarrow aabbB \Rightarrow aabbcd$$

The derivation tree corresponding to the derivation is on the figure 4.2. The used rules are also.

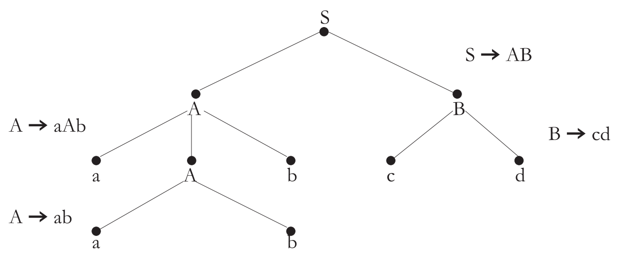


Figure 4.2: Derivation tree

Note 4.1 A vertex of a derivation tree, which is labeled with a terminal symbol must be a leaf of the derivation tree.

When constructing a derivation tree corresponding to a given derivation we repeatedly apply the step (2) from definition 4.2. The application is illustrated on the figure 4.3.

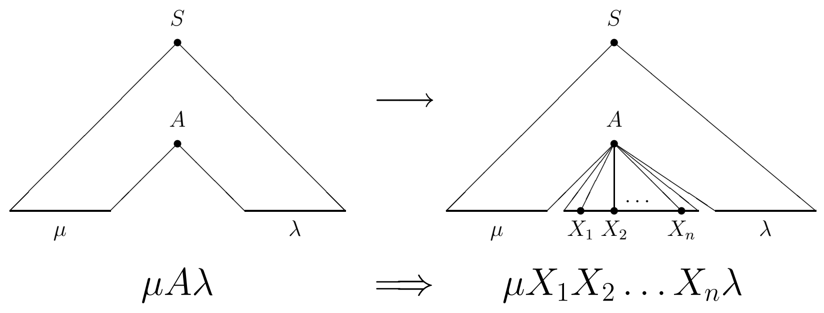


Figure 4.3: Construction of a derivation tree

Example 4.3 Consider the the grammar

$$G = (\{\langle\text{expression}\rangle, \langle\text{term}\rangle, \langle\text{factor}\rangle\}, \{+, -, *, /, (,), i\}, P, \langle\text{expression}\rangle),$$

which is often used for a description of an arithmetical expression with binary operations $+, -, *, /$. A terminal symbol i corresponds to an identifier. The set P contains the following rules:

$$\begin{aligned}
\langle\text{expression}\rangle &\rightarrow \langle\text{term}\rangle \mid \langle\text{expression}\rangle + \langle\text{term}\rangle \mid \langle\text{expression}\rangle - \langle\text{term}\rangle \\
\langle\text{term}\rangle &\rightarrow \langle\text{factor}\rangle \mid \langle\text{term}\rangle * \langle\text{factor}\rangle \mid \langle\text{term}\rangle / \langle\text{factor}\rangle \\
\langle\text{factor}\rangle &\rightarrow (\langle\text{expression}\rangle) \mid i
\end{aligned}$$

It is easy to show that for example the strings i, (i), $i * i$, $i * i + i$, $i * (i + i)$ are sentences of the language $L(G)$.

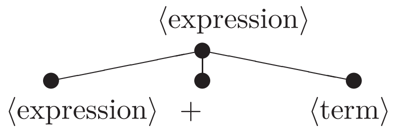
Let us assume, that we shall construct a derivation tree for the sentential form $\langle\text{expression}\rangle + \langle\text{term}\rangle * \langle\text{factor}\rangle$. At first we show, that this string is indeed a sentential form:

$$\langle\text{expression}\rangle \Rightarrow \langle\text{expression}\rangle + \langle\text{term}\rangle \Rightarrow \langle\text{expression}\rangle + \langle\text{term}\rangle * \langle\text{factor}\rangle$$

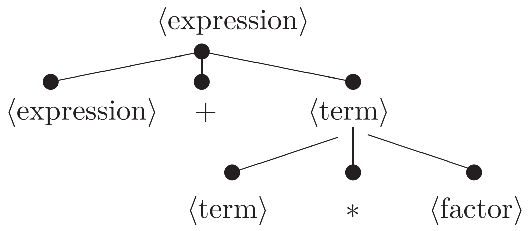
We start designing the derivation tree from the start symbol:

⟨expression⟩

The following construction corresponds to the first direct derivation:



After visualization of the second direct direction we obtain the resulting derivation tree.



Given a derivation of a sentential form, there is exactly one corresponding derivation tree. The proof arises from the construction of the derivation tree. Let us have a look at the reverse implication: Given a derivation tree of a sentential form there is exactly one corresponding derivation. We consider the grammar G from example 4.1.

$$\begin{aligned} S &\rightarrow A\ B \\ A &\rightarrow aAb \mid ab \\ B &\rightarrow cBd \mid cd \end{aligned}$$

We have illustrated the derivation tree for the derivation

$$S \Rightarrow AB \Rightarrow aAbB \Rightarrow aabbB \Rightarrow aabbcd$$

on the figure 4.2. Now we demonstrate, that there are also different derivations of the sentence *aabbcd*.

$$S \Rightarrow AB \Rightarrow Acd \Rightarrow aAbcd \Rightarrow aabbcd$$

or

$$S \Rightarrow AB \Rightarrow aAbB \Rightarrow aAbcd \Rightarrow aabbcd$$

The given derivations of the sentence *aabbcd* differ in order, in which non-terminals for direct derivations were chosen. However,this order does not reflect in the resulting derivation tree and therefore the derivation trees corresponding to the second and the third mentioned derivations of the sentence *aabbcd* are identical with the derivation tree on the figure 4.2. The statement, that for a given derivation tree of sentential form there is exactly one corresponding derivation is thus false.

Let us remark, that there are no more derivations of the sentence *aabbcd* in the grammar from example 4.2. In the first and the second mentioned derivations the rules were applied according to a certain canonical way, which is characteristic for the most common syntactic analyzers of programming languages compilers. Hence we introduce for these special derivations their own names.

Definition 4.3 Let $S = \alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n = \alpha$ be a derivation of a sentential form α. We call the derivation a *leftmost* (*rightmost*) derivation of the sentential form α if in each string $\alpha_i,\ i = 1, \ldots, n-1$ the leftmost (rightmost) non-terminal was rewritten.

The first derivation of the sentence *aabbcd* is then a leftmost derivation, the second one is a rightmost one.

If $S = \alpha_0 \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n = w$ is a leftmost derivation of a sentence w, then each $\alpha_i,\ i = 0, 1, \ldots, n-1$ is of the form $x_i A_i \beta_i$, where $x_i \in \Sigma^*,\ A_i \in N$ a $\beta_i \in (N \cup \Sigma^*)$. In order to obtain the sentential form α_{i+1} the non-terminal A_i is rewritten. For the rightmost derivation the situation is vice versa.

4.2 Phrase of sentential form

Definition 4.4 Let $G = (N, \Sigma, P, S)$ be a grammar and let a string $\lambda = \alpha\beta\gamma$ be a sentential form. A substring β is called *a phrase of the sentential form* λ with respect to non-terminal A from N, if the following conditions hold:

$$\begin{aligned} S &\Rightarrow^* \alpha A \gamma \\ A &\Rightarrow^+ \beta \end{aligned}$$

The substring β is *the simple phrase of the sentential form* λ, if the following conditions hold:

$$\begin{aligned} S &\Rightarrow^* \alpha A \gamma \\ A &\Rightarrow \beta \end{aligned}$$

Example 4.4 The task is to find all phrases and all simple phrases of the sentential forms $\langle\text{expression}\rangle + \langle\text{term}\rangle * \langle\text{factor}\rangle$ in the grammar, which describes arithmetical expressions. As we have already mentioned, the derivation of this sentential form is of form:

$$\langle\text{expression}\rangle \Rightarrow \langle\text{expression}\rangle + \langle\text{term}\rangle \Rightarrow \langle\text{expression}\rangle + \langle\text{term}\rangle * \langle\text{factor}\rangle$$

Because

$$\begin{aligned}\langle\text{expression}\rangle &\Rightarrow^* \langle\text{expression}\rangle + \langle\text{term}\rangle \qquad \text{and}\\ \langle\text{term}\rangle &\Rightarrow \langle\text{term}\rangle * \langle\text{factor}\rangle\end{aligned}$$

holds, the string $\langle\text{term}\rangle * \langle\text{factor}\rangle$ is the simple phrase of the sentential form $\langle\text{expression}\rangle + \langle\text{term}\rangle * \langle\text{factor}\rangle$ with respect to the non-terminal $\langle\text{term}\rangle$.

Further only:

$$\begin{aligned}\langle\text{expression}\rangle &\Rightarrow^* \langle\text{expression}\rangle\\ \langle\text{expression}\rangle &\Rightarrow^+ \langle\text{expression}\rangle + \langle\text{term}\rangle * \langle\text{factor}\rangle\end{aligned}$$

and this implies, that the sentential form $\langle\text{expression}\rangle + \langle\text{term}\rangle * \langle\text{factor}\rangle$ is a phrase of itself with respect to the start symbol. This fact is a corollary of a trivial case in definition of the phrase, when the strings α and γ are empty. There are no more phrases of the sentential form $\langle\text{expression}\rangle + \langle\text{term}\rangle * \langle\text{factor}\rangle$.

The notion of phrase is a pivotal notion for syntactic analysis. The whole class of syntactic analyzers is built on methods of searching the leftmost simple phrase of a sentential form (a sentence). Because further we use the notion of the leftmost simple phrase very often, we introduce the special name *l-phrase* for it.

In the sentential form $\langle\text{expression}\rangle + \langle\text{term}\rangle * \langle\text{factor}\rangle$ there is only one simple phrase $\langle\text{term}\rangle * \langle\text{factor}\rangle$. This phrase is therefore at once l-phrase.

The notion of a phrase of a sentential form is very closely related to the notion of a subtree of the corresponding derivation tree. By a subtree of a derivation tree we understand that part of the tree, which is specified by a certain vertex, so-called root of the subtree, together with all vertices, which are reachable from the root of the subtree via corresponding edges, including these edges.

Example 4.5 The subtrees of the derivation tree of the sentence *aabbcd* from the figure 4.2 are the subtrees on the figure 4.4.

Let us assume, that a non-terminal A is the root of a subtree of the derivation tree. If β is a string of leaves of the subtree, then indeed $A \Rightarrow^+ \beta$ holds. Let α be a string of leaves on the left side, γ a string of leaves on the right side from β. Then $S \Rightarrow^* \alpha\beta\gamma$ holds; S is the root of the derivation tree. It means, that β is a phrase of the sentential form $\alpha\beta\gamma$ with respect to the non-terminal A. The situation is illustrated on the figure 4.5.

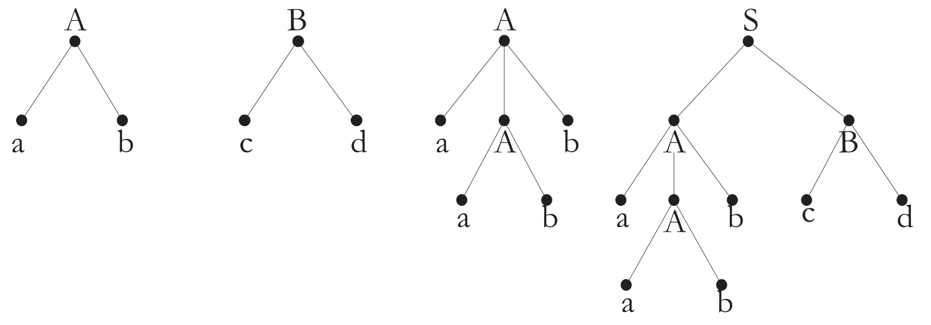


Figure 4.4: Subtrees of a derivation tree

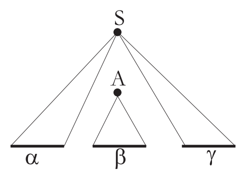


Figure 4.5: Phrase of a sentential form

A subtree of a derivation tree then corresponds to the phrase of incident sentential form. The phrase is formed by leaves of the subtree. A simple phrase corresponds to a subtree, which is result of a direct derivation $A \Rightarrow \beta$.

Example 4.6 Find all phrases of the sentence *abcd* in the grammar from example 4.1. We first construct the derivation tree. For it's construction we create (e.g.) leftmost derivation of the sentence:

$$S \Rightarrow AB \Rightarrow aAbB \Rightarrow aabbB \Rightarrow aabbcBd \Rightarrow aabbccdd$$

Marked subtrees correspond to further mentioned phrases defined for the incident non-terminals. Superscripts identify occurrences of the same non-terminal (see 4.6)

Phrase	Non-terminal
aabbccdd	S
aabb	A^1
ab	A^2
ccdd	B^1
cd	B^2

The phrases *ab* and *cd* are simple, *ab* is a l-phrase.

4.3 Ambiguous grammars

Definition 4.5 Let G be a grammar. We say that a sentence w generated

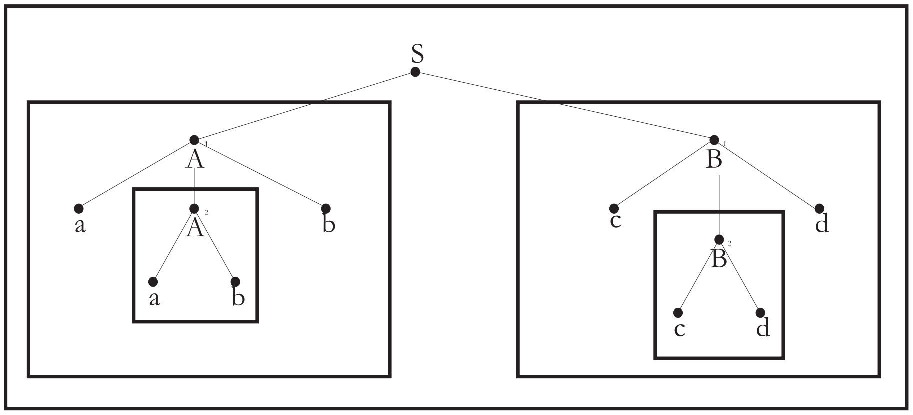


Figure 4.6: Subtrees of a derivation tree

in G is *ambiguous*, if there are at least two different derivation trees with their leaves forming the sentence w. *A grammar G is ambiguous*, if G generates at least one ambiguous sentence. In the opposite case we say the grammar is called *unambiguous*.

Notice, that we define an ambiguous grammar, not an ambiguous language. In many cases it is possible to eliminate the ambiguousness of sentences using another suitable transformations of ambiguous grammar, indeed without changing the language generated by the new unambiguous grammar. However there are languages, which can not be generated by any unambiguous grammar, so-called *inherently ambiguous languages*.

Example 4.7 Let us consider the grammar $G = (\{E\}, \{+, -, *, /, (,), i\}, P, E)$ with the set of the rules P

$\boxed{x+y}$

$$E := E \ + \ E \ \mid E \ - \ E \ \mid E \ * \ E \ \mid E \ / \ E \ \mid (\ E \ ) \ \mid i$$

The language $L(G)$ is equal to the language generated by the grammar from example 4.3 and is formed by arithmetical expressions with binary operations.

This grammar is unlike the grammar from example 4.3 ambiguous. Let us take for example the sentence $i + i * i$ and all possible derivation trees corresponding to the sentence (see figure 4.7).

The existence of two different derivation trees corresponding to the sentence $i + i * i$ proves, that the grammar G is ambiguous. If $+$ denotes summation and $*$ multiplication, then it is not clear, if the first operation is multiplication (the first derivation tree) or summation (the second derivation tree). On the other hand the unambiguous grammar from example 4.3 respects the usual priority of the operations (multiplication has bigger priority than summation), which is apparent from the only one derivation tree of the sentence $i + i * i$ on the figure 4.8

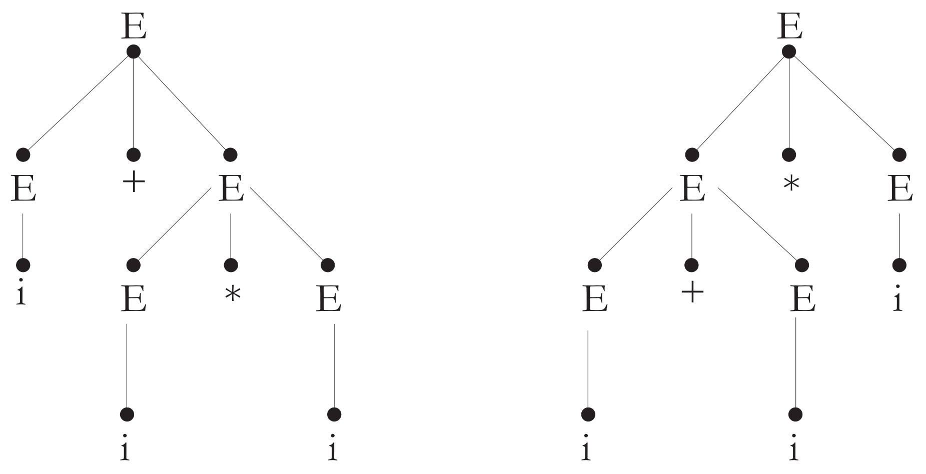


Figure 4.7: Derivation tree of the sentence $i + i * i$

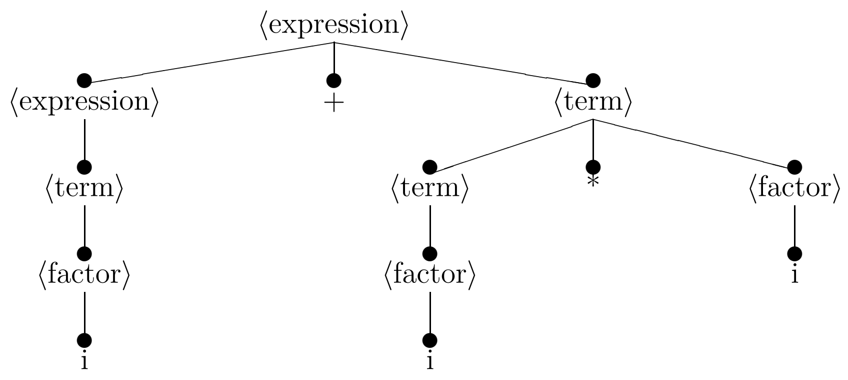


Figure 4.8: Derivation tree of the $i + i * i$ in G_2

It has been proven, that the problem, if a given context-free grammar is or is not ambiguous, is undecidable, i.e. there is no algorithm, which would reveal ambiguousness of each context-free grammar in finite time.

Example 4.8 A grammar containing a rule of form $A \rightarrow A$ is obviously ambiguous. This rule can be omitted without changing the language generated by the reduced grammar.

Note 4.2 Ambiguousness of a grammar, not generating an inherent ambiguous language, is generally considered a negative property, because it leads to sentences, which have several interpretations. However, on the other hand, an ambiguous grammar might be simpler than the corresponding unambiguous grammar (see example 4.7). This fact is sometimes used when constructing compilers in such way, that an ambiguous grammar is applied and undesirable interpretations of an ambiguous sentence are eliminated by an additional semantic rule.

Example 4.9 One of the well-known instances of ambiguousness in programming languages are constructions with `then` and `else`. Actually, the rewriting rules

$$\begin{aligned} S &\rightarrow \texttt{if}\ b\ \texttt{then}\ S\ \texttt{else}\ S \\ S &\rightarrow \texttt{if}\ b\ \texttt{then}\ S \\ S &\rightarrow p \end{aligned}$$

where b denotes a boolean expression and p other than a conditional statement, lead to an ambiguous interpretation of the conditional statement. For example there are two different derivation trees for the sentence

$$\texttt{if}\ b\ \texttt{then}\ \texttt{if}\ b\ \texttt{then}\ p\ \texttt{else}\ p$$

Whereas Algol 60 does not allow this construction (after `then` there has to be a compound statement), language Pascal solves ambiguousness with the semantic rule: "a given `else` relates to the closest previous `then`". Let us note, that also this rule can be captured syntactically:

$$\begin{aligned} S_1 &\rightarrow \texttt{if}\ b\ \texttt{then}\ S_1 \mid\ \texttt{if}\ b\ \texttt{then}\ S_2\ \texttt{else}\ S_1 \mid p \\ S_2 &\rightarrow \texttt{if}\ b\ \texttt{then}\ S_2\ \texttt{else}\ S_2 \mid p \end{aligned}$$

With use of these rewriting rules we obtain only one derivation tree (fig 4.9) for the sentence $\texttt{if}\ b\ \texttt{then}\ \texttt{if}\ b\ \texttt{then}\ p\ \texttt{else}\ p$

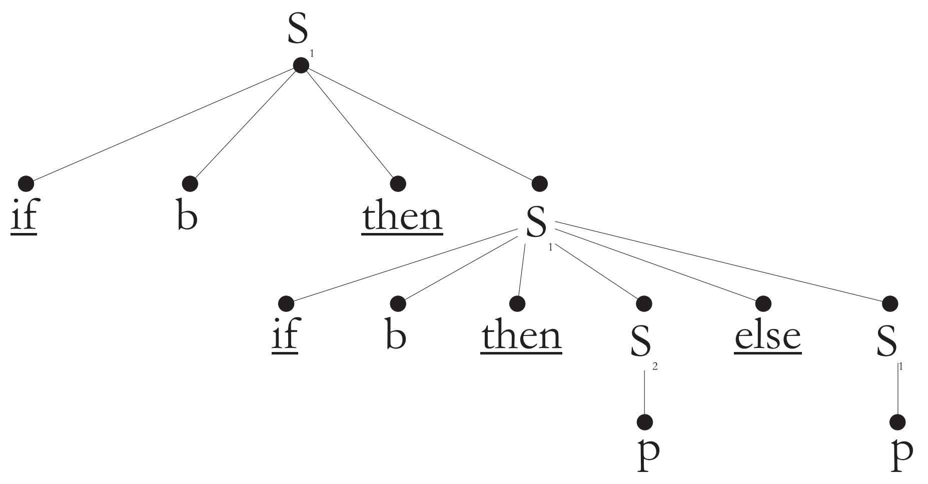


Figure 4.9: Derivation tree of conditional statement

Example 4.10 Find a context-free grammar, which generates correctly formed regular expressions over an alphabet $\{a, b\}$.
Solution:

A very simple solution to this problem is given by recursive definition of the regular set 3.10 $G^0_{RV} = (\{R\},\ \{a, b, \epsilon, +, ^*, (,)\}, P, R)$, where

R is the only non-terminal describing syntactic category regular expression

ϵ denotes an empty string as a part of an expression

$+$ corresponds to the operation union of regular sets

* corresponds to the operation iteration of a regular set

Concatenation is not explicitly marked. We construct the rules building the set P on the basis of the definition of a regular set:

$$R \to a \mid b \mid \epsilon \mid R + R \mid RR \mid R^* \mid (R)$$

The constructed grammar G^0_{RV} is simple, however it has a property, which is usually undesirable – ambiguousness. In order to construct an unambiguous grammar we use analogy with the grammar for arithmetical expression. Besides the highest syntactical category R describing the whole regular expression, we introduce further non-terminals:

K for subexpressions formed by operation concatenation

I for subexpressions formed by operation iteration

P for primitive regular expressions

The resulting grammar can be in the form:

$$\begin{array}{rcl} G_{RV} & = & (\{R, K, I, P\}, \{a, b, \epsilon, +, {}^*, (,)\}, PP, R) \text{ with the rules} \\ R & \to & R + K \mid K \\ K & \to & KI \mid I \\ I & \to & I^* \mid P \\ P & \to & a \mid b \mid \epsilon \mid (R) \end{array}$$

4.4 Sentence parsing

A construction of a derivation or a derivation tree for a given sentence or a sentential form is called *parsing* or *syntactic analysis* of the sentence or the sentential form. A *syntactic analyzer* (a *parser*) is a program, which performs parsing of sentences of certain language.

Parsing algorithms can be divided according to way of construction of a sentence derivation, i.e. the way of creating a derivation tree into two basic groups:

- top-down parsing
- bottom-up parsing

When doing top-down parsing we start building a derivation tree from the start symbol (the root of the derivation tree) and with step-by-step direct derivations we reach terminal symbols (the leaves of the derivation tree), which form the analyzed sentence. A problem is in right choice of direct derivations, i.e. in order of using rewriting rules.

When doing bottom-up parsing we start building a derivation tree from the leaves and by using direct reductions we reach the root (the start symbol of the grammar).

An essential problem of this class of parsers is finding the first substring of a sentence (in next steps of sentential forms), which can be reduced to certain non-terminal – a root of a subtree of the derivation tree. This substring, as we know, is called l-phrase.

Example 4.11 We illustrate the difference between both types of parsing on example of the grammar with the rules (S is the start symbol)

$\boxed{\texttt{x+y}}$

$$\begin{aligned} S &\rightarrow aSd \mid aAd \\ A &\rightarrow bAc \mid bc \end{aligned}$$

which generates the language $L = \{a^n b^m c^m d^n \mid n \geq 1, m \geq 1\}$.

On the figure 4.10 there is demonstrated a construction of a derivation tree with use of the top-down method together with the corresponding leftmost derivation of the sentence *abbccd*.

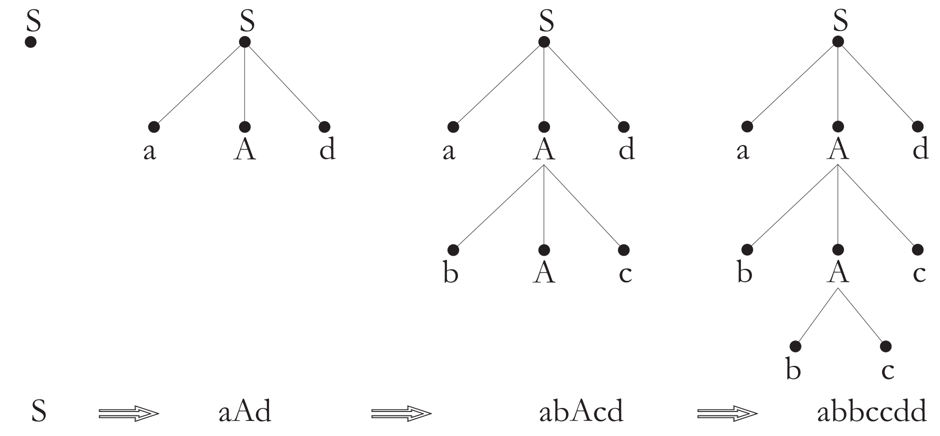


Figure 4.10: Top-down syntactic analysis

On the figure 4.11 there is illustrated a construction of derivation tree with use of the bottom-up method. The corresponding right derivation is written from right to left.

Let us come back to basic problems of parsing. When doing top-down parsing, we construct the leftmost sentence derivation. Let us assume, that in certain step of the parsing A is the leftmost non-terminal, which is to be rewritten. Further let us assume, that the grammar contains n rules with the left side A:

$$A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$$

How do we recognize, to which string α_i the non-terminal A has to be rewritten? Similarly, when doing bottom-up parsing, when in each step the

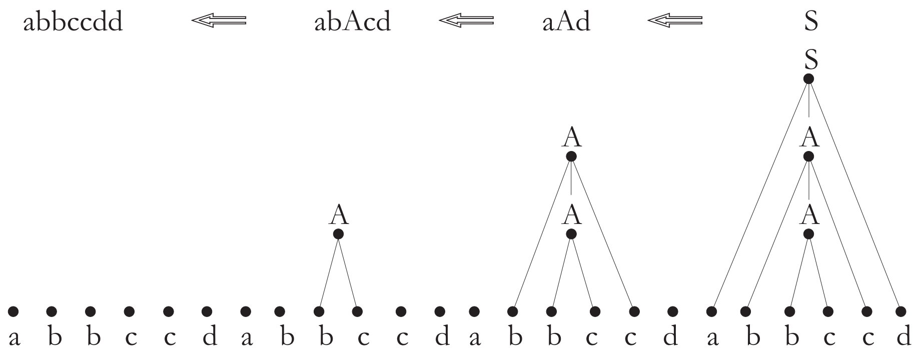

abbccdd ⟸ abAcd ⟸ aAd ⟸ S
S
A
A
A
A
A
a b b c c d a b b c c d a b b c c d a b b c c d

Figure 4.11: Bottom-up syntactic analysis

l-phrase of a sentential form is reduced, the main problem is in determining the beginning and the end of the l-phrase.

One of the solutions to this problems is random choice one of the possible alternatives. If later it turns out that the chosen alternative was not correct, the process of parsing has to be "returned" and a different alternative to be considered. (For example with top-down parsing we try to rewrite A with strings $\alpha_1, \alpha_2 \ldots$ in such way, that the prefix of obtained leftmost derivation, which contains only terminal symbols, is identic to the prefix of the analyzed sentence). This type of analysis is called *parsing with backtracking*. Even if the number of backtracks is limited, it is obvious, that the parsing with backtracking is very time-consuming. Besides that, the backtracks are source of complications when performing semantic evaluation of the compiled program.

A practical solution to problems of parsing of programming languages are so-called deterministic grammars (chapter 6.), which allow to determine the correct alternative in each step of analysis according to the context of processed substring of the sentential form. This type of parsing is called *deterministic parsing*. For illustrating the role of the context let us consider the sentence $i + i * i$ in the grammar from example 4.3. After processing the substring $i + i$ the context represented by the terminal $*$ help us to determine the phrase, which is not $i + i$, but $i * i$.

4.5 Transformations of context-free grammars

Generally, there is no algorithmic method, which would allow us to construct a grammar generating a language with arbitrary structure. The theory of computability even shows, that infinitely many formal languages are not possible to describe with any finite formal system, hence with any grammar. However, there is a number of useful transformations of grammars, which allow to modify the grammar without changing the generated language. In this paragraph, we describe some of the transformations via mathematical notation of the corresponding algorithms.

Definition 4.6 We say, that grammars G_1 and G_2 are *equivalent*, if and only if $L(G_1) = L(G_2)$, i.e. if languages generated by them are the same.

In some cases, it may happen, that a constructed grammar contains useless symbols and rewriting rules. Apart from the fact, that this might be a consequence of the mistake in the grammar construction, it is very likely, that syntactic analysis will be less efficient. Thus it is important to remove useless symbols and useless rules from the grammar.

For example let us consider the grammar $G = (\{S, A\}, \{a, b\}, P, S)$, where $P = \{S \to a, A \to b\}$. Apparently, the non-terminal A and the terminal b cannot appear in any sentential form. Therefore we can remove them from the grammar G together with the useless rule $A \to b$ without changing the language $L(G)$.

Definition 4.7 Let $G = (N, \Sigma, P, S)$ be a grammar. A symbol $X \in (N \cup \Sigma)$ is called *useless* in grammar G, when there is no derivation of form $S \Rightarrow^* wXy \Rightarrow^* wxy$ where strings w, x, y are in Σ^*.

In order to find out if a non-terminal A is useless or not we describe an algorithm determining, if there is a string of terminal symbols which can be generated from the non-terminal A, i.e. we discover whether $\{w \mid A \Rightarrow^* w,\ w \in \Sigma^*\} = \emptyset$.

The algorithm can be extended to an algorithm, which determines, if a language generated by a context-free grammar is non-empty.

Algorithm 4.1
Is $L(G)$ non-empty?

Input: A grammar $G = (N, \Sigma, P, S)$.

Output: YES if $L(G) \neq \emptyset$, NO in the other case.

Method: We construct sets $N_0, N_1, \dots$ recursively this way:

(1) $N_0 = \emptyset, i = 1$

(2) $N_i = \{A \mid A \to \alpha \text{ je v } P \wedge \alpha \in (N_{i-1} \cup \Sigma)^*\} \cup N_{i-1}$

(3) If $N_i \neq N_{i-1}$, then $i = i+1$ and go back to step 2. If $N_i = N_{i-1}$, then $N_t = N_i$

(4) If the start symbol S is in N_t, then output is YES, in the other case NO.

Note 4.3 If a set of non-terminals N has n elements, the algorithm 4.1 has to end within $n + 1$ iterations of step 2, because $N_t \subseteq N$, .

Theorem 4.1 Algorithm 4.1 gives output YES, if and only if $S \Rightarrow^* w$ for some $w \in \Sigma^*$.
Proof: First we prove by induction for i the implication:

$$\text{If } A \in N_i, \text{ then } A \Rightarrow^* w \text{ for some } w \in \Sigma^* \qquad (1)$$

For $i = 0$ the implication holds, because $N_0 = \emptyset$. Let us assume, that (1) holds for i, and that A is an element of the set N_{i+1}. If furthermore $A \in N_i$, then the induction step is trivial. If A is in $N_{i+1} \setminus N_i$, then there is a rule $A \to X_1 \ldots X_k$, where X_j is either a terminal symbol, or $X_j \in N_i,\ j = 1, \ldots, k$. Thus for each j there is a string $w_j \in \Sigma^*$ such that $X_j \Rightarrow^* w_j$ and

$$A \Rightarrow X_1 \ldots X_k \Rightarrow^* w_1 X_2 \ldots X_k \Rightarrow^* \ldots \Rightarrow^* w_1 \ldots w_k = w$$

Now we prove the opposite implication:

$$\text{If } A \ \Rightarrow^n w, \text{ then } A \in N_i \text{ for some } i \qquad (2)$$

again by induction.

If $n = 1$, then obviously $i = 1$. Let us assume, that (2) holds for n and that $A \Rightarrow^{n+1} w$ .Then we can write $A \Rightarrow X_1 \ldots X_k \Rightarrow^n w$, where $w = w_1 \ldots w_k$ and $X_j \Rightarrow^{n_j} w_j$ for $j = 1, \ldots, k,\ n_j \leq n$. (The substrings w_j are phrases of the string w with respect to the non-terminals X_j in case, that $X_j \in N$, then $X_j \in N_{i_j}$ for some i_j. Let $i_j = 0$, if $X_j \in \Sigma$. Let $i = 1 + max(i_1, \ldots, i_k)$. Then, according to the definition, $A \in N_i$. If now we put $A = S$ in the implications (1) and (2), we get proof of the theorem 4.1. □

Definition 4.8 We call a symbol $X \in (N \cup \Sigma)$ *unreachable* in the grammar $G = (N, \Sigma, P, S)$, if X cannot appear in any sentential form.

Algorithm 4.2
The elimination of unreachable symbols

Input: A grammar $G = (N, \Sigma, P, S)$.

Output: A grammar $G' = (N', \Sigma', P', S)$, for which following holds:

(i) $L(G') = L(G)$

(ii) For each X in $(N' \cup \Sigma')$ there are strings α a β in $(N' \cup \Sigma')^*$ such that
$S \Rightarrow^* \alpha X \beta$ in the grammar G'.

Method:

(1) We put $V_0 = \{S\}$ and $i = 1$

(2) We construct $V_i = \{X \mid A \to \alpha X \beta \in P \wedge A \in V_{i-1}\} \cup V_{i-1}$

(3) If $V_i \neq V_{i-1}$, put $i = i + 1$ and go to step 2.
If $V_i = V_{i-1}$, then
$N' = V_i \cap N$
$\Sigma' = V_i \cap \Sigma$
$P' \subseteq P$ contains the rules, which are constructed only with use of symbols in V_i.

The algorithm 4.2 is similar to the algorithm 4.1. Because $V_i \subset N \cup \Sigma$, the number of repetitions of step (2) in the algorithm 4.2 is finite. The proof of the algorithm is done by induction for i for this equivalence:

$$S \Rightarrow^* \alpha X \beta, \text{ if and only if } X \in V_i \text{ for some } i.$$

Now we formulate an algorithm for elimination of useless symbols.

Algorithm 4.3
The elimination of useless symbols.

Input: A grammar $G = (N, \Sigma, P, S)$ generating non-empty language.

Output: A grammar $G' = (N', \Sigma', P', S)$, for which following holds:

(i) $L(G) = L(G')$
(ii) Any symbol in $N' \cup \Sigma$' is not useless

Method:

(1) Apply the algorithm 4.1 on the grammar G with aim of obtaining the set N_t. Put $\overline{G} = (N_t, \Sigma, P_1, S)$, where P_1 contains the rules formed only with use of symbols in $N_t \cup \Sigma$
(2) Apply the algorithm 4.2 on the grammar $\overline{G}$. The result is a grammar $G' = (N', \Sigma', P', S)$, which does not contain any useless symbols.

In step (1) of the algorithm 4.3 all non-terminals, which cannot generate any terminal strings are removed from G. Then in step (2) all the symbols, which are unreachable are eliminated. The order of using the algorithms 4.1 and 4.2 is essential, reverse order does not guarantee the resulting grammar without useless symbols.

Theorem 4.2 The grammar G' from the algorithm 4.3 is equivalent to the grammar G and does not contain any useless symbols.

Proof: We prove by contradiction, that grammar G' does not contain useless symbols. We assume, that $A \in N'$ is a useless symbol. According to the definition of useless symbols, we have to consider following two cases:

1. $S \underset{G'}{\Rightarrow}^* \alpha A \beta$ does not hold for each α and β. In such case, we get a contradiction with step (2) of the algorithm 4.3.

2. $S \underset{G'}{\Rightarrow}^* \alpha A\beta$ holds for some α and β, but $A \underset{G'}{\Rightarrow}^* w, w \in \Sigma'^*$ does not hold. Then A is not eliminated in step (2) of the algorithm 4.3 and furthermore, if $A \underset{G}{\Rightarrow}^* \gamma B\delta$, then B is not eliminated in step (2). However, if $A \underset{G}{\Rightarrow}^* w$, then also $A \underset{G'}{\Rightarrow}^* w$ and therefore if $A \underset{G'}{\Rightarrow}^* w$ does not hold, then $A \underset{G}{\Rightarrow}^* w$ does not hold as well, which is a contradiction with step (1) of the algorithm 4.3.

Proof, that G' does not contain even useless terminal is done similarly. □

Example 4.12 Let us consider the grammar $G = (\{S, A, B\}, \{a, b\}, P, S)$, where P contains the rules:

$$\begin{aligned} S &\rightarrow a \mid A \\ A &\rightarrow AB \\ B &\rightarrow b \end{aligned}$$

If we apply the algorithm 4.3 to G, then we get $N_t = \{S, B\}$ in step 1, so $\overline{G} = (\{S, B\}, \{a, b\}, \{S \rightarrow a, B \rightarrow b\}, S\})$. After application of the algorithm 4.2 on $\overline{G}$ we get $V_2 = V_1 = \{S, a\}$. The result is then the equivalent grammar

$$G' = (\{S\}, \{a\}, \{S \rightarrow a\}, S).$$

If we first applied the algorithm 4.2, we would find out, that all symbols in G are reachable and the algorithm 4.2 would not change the grammar G. After application of the algorithm 4.1 we would get $N_t = \{S, B\}$, so the resulting grammar would be $\overline{G}$ and not G'.

Another important transformation, which we describe now, is a transformation eliminating rules of form $A \rightarrow \epsilon$ (ϵ is an empty string), so-called ϵ-rules from a grammar. If $L(G)$ should contain an empty string, then it is not possible for G not to contain any ϵ-rule. The following definition of a grammar without ϵ-rules respects this fact.

Definition 4.9 We call a grammar $G = (N, \Sigma, P, S)$ *grammar without ϵ-rules*, if either P does not contain any ϵ-rule, or, in case $\epsilon \in L(G)$, there is only one ϵ-rule of form $S \rightarrow \epsilon$ and the start symbol S does not appear on any right side of any rule in P.

Algorithm 4.4
The transformation to the grammar without ϵ-rules.

Input: A grammar $G = (N, \Sigma, P, S)$

Output: A equivalent grammar $G' = (N', \Sigma', P', S')$ without ϵ-rules.

Method:

(1) Construct $N_\epsilon = \{A \mid A \in N \text{ a } A \Rightarrow^* \epsilon\}$. A construction of the set N_ϵ is analogous to the construction of N_t from 4.1.

(2) Let P' be a set of rules, which we construct this way:

a) If $A \rightarrow \alpha_0 B_1 \alpha_1 B_2 \ldots B_k \alpha_k$ is in P, $k \geq 0$ and each B_i is in N_ϵ for $1 \leq i \leq k$, but any of symbols of the strings α_j is not in N_ϵ, $0 \leq j \leq k$, then add to P' the new subset of rules of form

$$A \rightarrow \alpha_0 X_1 \alpha_1 X_2 \ldots X_k \alpha_k$$

where X_i is either B_i or ϵ. Do not add the ϵ-rule $A \rightarrow \epsilon$ which appears if all $\alpha_i = \epsilon$.

b) If $S \in N_\epsilon$ then add to P' rules

$$S' \rightarrow \epsilon \mid S$$

S' is new start symbol. Put $N' = N \cup \{S'\}$ If $S \notin N_\epsilon$, then $N' = N$ and $S' = S$

(3) The resulting grammar is $G' = (N', \Sigma', P', S')$

Example 4.13 Let us consider the grammar $G = (\{S\}, \{a, b\}, P, S)$, where P contains the rules $S \rightarrow aSbS \mid bSaS \mid \epsilon$.

After application of the algorithm 4.4 we obtain a grammar G' without ϵ-rules. $G' = (\{S', S\}, \{a, b\}, P', S')$, where P' contains rules

$$\begin{array}{lcl} S' & \rightarrow & S \ \mid \epsilon \\ S & \rightarrow & aSbS \mid abS \mid aSb \mid ab \mid bSaS \mid baS \mid bSa \mid ba \end{array}$$

Theorem 4.3 The algorithm 4.3 transforms a grammar G to an equivalent grammar G' without ϵ-rules.
Proof: G' obviously does not contain ϵ-rules except for the eventual rule $S \rightarrow \epsilon$, if $\epsilon \in L(G)$. The proof, that $L(G) = L(G')$ is done by induction for the length of the string w in the equivalence

$$A \underset{G'}{\Rightarrow}^* w \text{ if and only if } w \neq \epsilon \wedge A \underset{G}{\Rightarrow}^* w$$

Another useful transformation is an elimination of rules of form $A \rightarrow B$.
□

Definition 4.10 A rewriting rule $A \rightarrow B, A, B \in N$ is called *simple rule.*

Algorithm 4.5
The elimination of simple rules.

Input: A grammar G without ϵ-rules.

Output: An equivalent grammar G' without simple rules.

Method:

(1) For each $A \in N$ construct a set $N_A = \{B \mid A \Rightarrow^* B\}$ this way:

a) $N_0 = \{A\}, i = 1$
b) $N_i = \{C \mid B \to C$ je v P a $B \in N_{i-1}\} \cup N_{i-1}$
c) If $N_i \neq N_{i-1}$, put $i = i + 1$ and repeat step b). In the other case $N_A = N_i$.

(2) Construct P' this way: If $B \to \alpha$ is in P and is not a simple rule, then for each A, for which $B \in N_A$, add rules $A \to \alpha$ to P'

(3) The resulting grammar is $G = (N, \Sigma, P', S)$

Example 4.14 Let us consider a grammar with the rewriting rules:

$\boxed{\texttt{x+y}}$

$$\begin{array}{lcl} E & \to & E + T \mid T \\ T & \to & T * F \mid F \\ F & \to & (E) \mid i \end{array}$$

This grammar differs from the grammar from exercise 4.3 only with alternative denotation of non-terminals and generates therefore the same language of arithmetical expressions After application of the algorithm 4.5:

$$\begin{array}{lcl} N_E & = & \{E, T, F\} \\ N_T & = & \{T, F\} \\ N_F & = & \{F\} \end{array}$$

and the resulting set of the rewriting rules, not containing simple rules is:

$$\begin{array}{lcl} E & \to & E + T \mid T * F \mid (E) \mid i \\ T & \to & T * F \mid (E) \mid i \\ F & \to & (E) \mid i \end{array}$$

Theorem 4.4 The algorithm 4.4 transforms an input grammar G to an equivalent grammar G' without simple rules.
Proof: The grammar G' obviously does not contain simple rules. In order to show, that $L(G) = L(G')$, we prove, that $L(G') \subseteq L(G)$ and also $L(G) \subseteq L(G')$.

1. $L(G') \subseteq L(G)$

   Let $w \in L(G')$. Then there is a derivation $S = \alpha_0 \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n = w$ in G'.

   If a rule $A \to \beta$ was used for a derivation $\alpha_i \underset{G'}{\Rightarrow} \alpha_{i+1}$, then there is a non-terminal B (eventually $A = B$) such that, $A \underset{G}{\Rightarrow}^* B$ and $B \underset{G}{\Rightarrow} \beta$ and hence $A \underset{G}{\Rightarrow}^* \beta$ and $\alpha_i \underset{G}{\Rightarrow}^* \alpha_{i+1}$. From this $S \underset{G}{\Rightarrow}^* w$ holds and w is therefore in $L(G)$.

2. $L(G) \subseteq L(G')$

   Let $w \in L(G)$ and $S = \alpha_0 \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n = w$ is a leftmost derivation of the string w in the grammar G. Let $i_1, i_2, \ldots i_k$ be a sequence of indexes formed only by those j, for which no simple rule was used in a leftmost derivation $\alpha_{j-1} \Rightarrow \alpha_j$. Particularly $i_k = n$, because the last applied rule in the derivation of the string w cannot be a simple rule. The derivation of the string w in G is a leftmost derivation, and therefore by application of simple rules we only replace the leftmost non-terminal by other non-terminal (represents left side of a rule in G'). Hence $S \underset{G'}{\Rightarrow} \alpha_{i_1} \underset{G'}{\Rightarrow} \ldots \underset{G'}{\Rightarrow} \alpha_{i_1} = w$ holds and therefore $w \in L(G')$.

We proved equivalence of the grammars G and G'. □

Definition 4.11 We call a grammar G *cycle-free*, if there is no derivation of the form $A \Rightarrow^+ A$ for any A from N. If G is a cycle-free grammar without ϵ-rules and does not contain any useless symbols, then we call G *a proper grammar.*

Theorem 4.5 If L is a context-free language, then there is a proper grammar G such that $L = L(G)$.

Proof: Arises from the algorithms 4.1–4.5. A grammar, which does not contain ϵ-rules and simple rules, is obviously a cycle-free grammar.

□

Note 4.4 A grammar, which contains cycles, is ambiguous and cannot be used for example for construction of a deterministic syntactic analyzer. On the other hand existence of ϵ-rules and simple rules does not imply existence of a cycle (i.e. a derivation of form $A \Rightarrow^+ A$ for some $A \in N$).

Another transformation, which we introduce is an elimination of the left recursion. This transformation is important for the top-down analysis usage.

Definition 4.12 Let $G = (N, \Sigma, P, S)$ be a grammar. A rewriting rule in P is called *left recursive (right recursive)*, if it is of the form $A \rightarrow A\alpha$ $(A \rightarrow \alpha A)$, $A \in N, \alpha \in (N \cup \Sigma)^*$. If in G there is a derivation $A \Rightarrow^+ \alpha A \beta$, for some $A \in N$, we call the grammar G *recursive.* If $\alpha = \epsilon$, then we talk about a *left recursive grammar*, if $\beta = \epsilon$, then we call G *right recursive.*

Let us remark, then if a language $L(G)$ is infinite, then G has to be recursive.

Before we formulate an algorithm for the left recursion elimination, we show how we can eliminate the left recursive rewriting rules.

Definition 4.13 A rule $A \rightarrow \alpha$, with the left side formed by the non-terminal A, is called A-rule.

Theorem 4.6 Let $G = (N, \Sigma, P, S)$ be a grammar where all A-rules in P are

$$A \to A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

None of the strings β_i does not start with the non-terminal A. The grammar $G' = (N \cup \{A'\}, \Sigma, P', S)$ where P' containing instead of A-rules of G the new rules

$$\begin{array}{lcl} A & \to & \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n \mid \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A' \\ A' & \to & \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_m \mid \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \end{array}$$

is equivalent to the grammar G, i.e. $L(G) = L(G')$.
Proof: The given transformation replaces the left recursive rules with the right recursive rules. If we denote languages $L_1 = \{\beta_1, \beta_2, \ldots, \beta_n\}$ and $L_2 = \{\alpha_1, \alpha_2, \ldots, \alpha_m\}$ we see, that in G it is possible to derive the strings forming language $L_1 L_2^*$ from the non-terminal A. Exactly these strings can be derived from A also in the grammar G'. The figure 4.12 illustrates the effect of the described transformations. $\square$

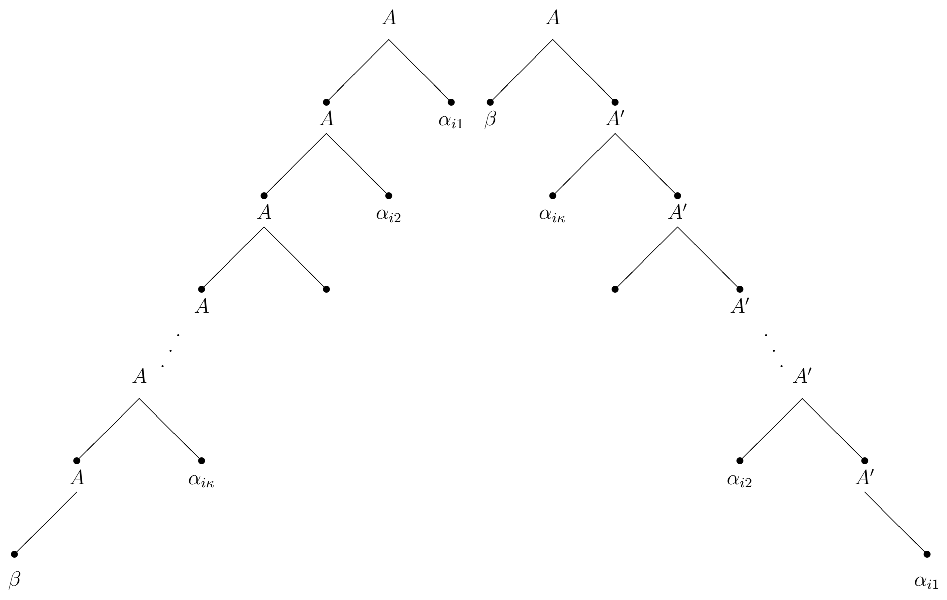


Figure 4.12: Elimination of the left recursion: the left tree corresponds to derivation in G and the right tree corresponds to derivation in G'

Example 4.15 Consider the grammar G from the previous exercise with the rules

x+y

$$\begin{array}{lcl} E & \to & E + T \mid T \\ T & \to & T * F \mid F \\ F & \to & (E) \mid i \end{array}$$

After the elimination of the left recursive rules we obtain an equivalent grammar G' with the rules

$$\begin{array}{rcl} E & \rightarrow & T \mid TE' \\ E' & \rightarrow & +T \mid +TE' \\ T & \rightarrow & F \mid FT' \\ T' & \rightarrow & *F \mid *FT' \\ F & \rightarrow & (E) \mid i \end{array}$$

Theorem 4.7 Let $G = (N, \Sigma, P, S)$ be a grammar, and let $A \rightarrow \alpha B \beta$, where $B \in N$ and $\alpha, \beta \in (N \cup \Sigma)^*$, be a rule from P. Let $B \rightarrow \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_n$ be all the B-rules from P. Then the grammar $G' = (N, \Sigma, P', S)$, where

$$P' = P - \{A \rightarrow \alpha B \beta\} \cup \{A \rightarrow \alpha\gamma_1\beta \mid \alpha\gamma_2\beta \mid \ldots \mid \alpha\gamma_n\beta\}.$$

is equivalent to G, i.e. $L(G) = L(G')$.
Proof: The result of the transformation is the elimination of the rule $A \rightarrow \alpha B \beta$ from the grammar G in such way, that for the non-terminal B we "substitute" all the right sides of the B-rules. Formally, the proof is done by showing validity of the conjunction $L(G) \subseteq L(G) \wedge L(G') \subseteq L(G)$. The effect of the transformation is illustrated on the figure 4.13, which shows derivation trees of the string *aabbb* in the grammar G resp. G'. The grammar G contains the rule $A \rightarrow aAA$ and two A-rules $A \rightarrow aAA \mid b$. If we put $\alpha = a, B = A, \beta = A$, we can eliminate the rule $A \rightarrow aAA$ by introducing these A-rules in the grammar G':

$$A \rightarrow aaAAA \mid abA \mid b$$

$\square$

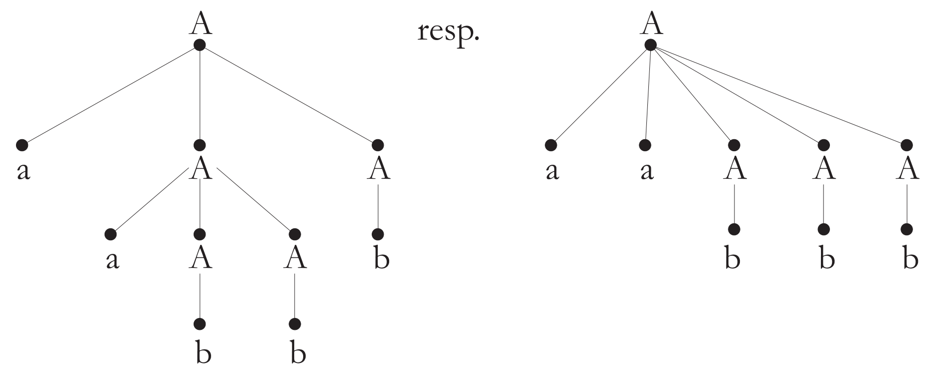


Figure 4.13: Derivation trees of the sentence *aabbb* in G resp. G'

Algorithm 4.6 Left recursion elimination

Input: A proper grammar $G = (N, \Sigma, P, S)$

Output: An equivalent grammar G' without left recursion

Method:

(1) Let $N = \{A_1, A_2, \dots, A_n\}$. We transform the grammar this way: if $A_i \rightarrow \alpha$ is a rule, then α starts either with a terminal or a non-terminal $A_j,\ j > i$. For this purpose put $i = 1$.

(2) Let $A_i \rightarrow A_i\alpha_1 \mid \dots \mid A_i\alpha_m \mid \beta_1 \mid \dots \mid \beta_p$ are all the A_i-rules and let each β_i does not start with any non-terminal A_k, if $k \leq i$. Replace all the A_i-rules by these rules:

$$\begin{array}{lcl} A_i & \rightarrow & \beta_1 \mid \dots \mid \beta_p \mid \beta_1 A_i' \mid \dots \mid \beta_p A_i' \\ A_i' & \rightarrow & \alpha_1 \mid \dots \mid \alpha_m \mid \alpha_1 A_i' \mid \dots \mid \alpha_m A_i' \end{array}$$

where A_i' is a new non-terminal. All the A_i-rules then begin with a terminal or a non-terminal $A_k, k > i$.

(3) If $i = n$, we have found the resulting grammar G'. In the other case, put $i = i + 1$ and $j = 1$.

(4) Each rule of form $A_i \rightarrow A_j\alpha$ replace with $A_i \rightarrow \beta_1\alpha \mid \dots \mid \beta_p\alpha$ where $A_j \rightarrow \beta_1 \mid \dots \mid \beta_p$ are all the A_j-rules. After this transformation all the A_j-rules start either with a terminal or a non-terminal $A_k, k > j$, so all the A_i-rules have this property as well.

(5) If $j = i - 1$, then go to step (2). Otherwise put $j = j + 1$ and repeat step (4).

Theorem 4.8 Each context free language can be generated by a grammar without left recursion.

Proof: Let G be a proper grammar. The algorithm 4.6 uses only the transformations from theorems 4.6 and 4.7 and hence $L(G) = L(G')$.

Further we have to prove, that G' is not a left recursive grammar. A formal proof is not given here (see [1]). However notice, that step (2) eliminates left recursive rules. Rules originated by the application of step (4) are neither left recursive. Because step (2) is applied on each non-terminal symbol, the resulting grammar cannot contain any left recursive rules. □

Example 4.16 Let G be a grammar with the rules:

$\boxed{x+y}$

$$\begin{array}{lcl} A & \rightarrow & BC \mid a \\ B & \rightarrow & CA \mid Ab \\ C & \rightarrow & AB \mid CC \mid a \end{array}$$

Let $A_1 = A, A_2 = B$ and $A_3 = C$. In each step we introduce only new rules for those non-terminals, whose rules are changed.

step (2),	i=1,	without change
step (4),	i=2, j=1,	$B \rightarrow CA \mid BCb \mid ab$
step (2),	i=2,	$B \rightarrow CA \mid ab \mid CAB' \mid abB'$
		$B' \rightarrow CbB' \mid Cb$
step (4),	i=3, j=1,	$C \rightarrow BCB \mid aB \mid CC \mid a$
step (4),	i=3, j=2,	$C \rightarrow CACB \mid abCB \mid CAB'CB \mid abB'CB \mid aB$
		$C \rightarrow CC \mid a$
step (2),	i=3,	$C \rightarrow abCB \mid abB'CB \mid aB \mid a \mid abCBC'$
		$C \rightarrow abB'CBC' \mid aBC' \mid aC'$
		$C' \rightarrow ACBC' \mid AB'CBC' \mid CC' \mid ACB$
		$C' \rightarrow AB'CB \mid C$

4.6 Chomsky normal form

Definition 4.14 A grammar $G = (N, \Sigma, P, S)$ is in *Chomsky normal form* (CNF), if each rule in P is in one of the forms:

(1) $A \rightarrow BC, A, B, C \in N$ or

(2) $A \rightarrow a$, $a \in \Sigma$ or

(3) If $\epsilon \in L(G)$, then $S \rightarrow \epsilon$ is a rule in P and the start symbol S does not appear on any right side of any rule.

Chomsky normal form of a grammar is a means simplifying a representation of a context-free language. Now we describe an algorithm of transformation of any context-free grammar into Chomsky normal form.

Algorithm 4.7 The transformation into Chomsky normal form.

Input: A proper grammar $G = (N, \Sigma, P, S)$ without simple rules.

Output: A grammar $G' = (N', \Sigma, P', S')$ in CNF, where $L(G) = L(G')$

Method: We obtain an equivalent grammar G' in CNF from the grammar G this way:

(1) The set of rules P' contains all the rules of form $A \rightarrow a$ from P

(2) The set of rules P' contains all the rules of form $A \rightarrow BC$ from P

(3) If a rule $S \rightarrow \epsilon$ is in P, then $S \rightarrow \epsilon$ is also in P'

(4) For each rule of form $A \rightarrow X_1 \dots X_k$, where $k > 2$ from P add this set of rules to P'. We denote a non-terminal X_i, where $X_i \in N$, or a new non-terminal if $X_i \in \Sigma$ by the symbol X'_i:

$$\begin{array}{rcl} A & \rightarrow & X'_1\langle X_2 \dots X_k\rangle \\ \langle X_2 \dots X_k\rangle & \rightarrow & X'_2\langle X_3 \dots X_k\rangle \\ & \vdots & \\ \langle X_{k-1}X_k\rangle & \rightarrow & X'_{k-1}X'_k \end{array}$$

where each symbol $\langle X_i \dots X_k\rangle$ denotes a new non-terminal symbol.

(5) For each rule of form $A \rightarrow X_1X_2$, where some of the symbols X_1 or X_2 is in Σ add the rule $A \rightarrow X_1'X_2'$ to P'.

(6) For each new non-terminal of form a' add the rule $a' \rightarrow a$ to P'. The resulting grammar is $G' = (N', \Sigma, P', S')$; set N' contains all the non-terminals $\langle X_i \dots X_k\rangle$ a a'.

Theorem 4.9 Let L be a context-free language. Then there is a context-free grammar G in CNF, such that $L = L(G)$.
Proof: According to theorem 4.5 a language L has a proper grammar G. Now it is enough to prove, that the result of the algorithm 4.7 is an equivalent grammar G' i.e. $L(G) = L(G')$. The equivalence though results directly from the application of theorem 4.7 to each rule in P', which has non-terminals of form a' and $\langle X_i...X_j\rangle$. The result is the grammar G'. $\Box$

Example 4.17 Let G be a grammar with the rules

x+y

$$\begin{array}{rcl}
S & \rightarrow & aAB \mid BA \\
A & \rightarrow & BBB \mid a \\
B & \rightarrow & AS \mid b
\end{array}$$

A Chomsky normal form $G' = (N', \{a, b\}, P', S)$ contains the rules:

$$\begin{array}{rcl}
S & \rightarrow & a'\langle AB\rangle \mid BA \\
A & \rightarrow & B\langle BB\rangle \mid a \\
B & \rightarrow & AS \mid b \\
\langle AB\rangle & \rightarrow & AB \\
\langle BB\rangle & \rightarrow & BB \\
a' & \rightarrow & a
\end{array}$$

The new set of non-terminals is thus $N' = \{S, A, B, \langle AB\rangle, \langle BB\rangle, a'\}$

Example 4.18 We assume, that a grammar G is in CNF and that a derivation of a string $w \in L(G)$ in G has the length p. What is the length of the sentence w?

x+y

Solution: Let $|w| = n$. Then for a derivation of the sentence it was necessary to apply n-times a rule of form $A \rightarrow a$ and (n-1)-times a rule of form $A \rightarrow BC$. We have

$$p = n + n - 1 = 2n - 1$$

and we see, that p is always an odd number. The solution of the problem is hence

$$|w| = \frac{p+1}{2}$$

4.7 Greibach normal form

Definition 4.15 A grammar $G = (N, \Sigma, P, S)$ is in Greibach normal form (GNF), if G is a grammar without ϵ-rules and if each rule (except for the eventual rule $S \to \epsilon$) is of the form $A \to a\alpha$ where $a \in \Sigma$ and $\alpha \in N^*$.

Lemma 4.1 Let $G = (N, \Sigma, P, S)$ be a grammar without left recursion. Then there is a linear order $<$ defined on the set of non-terminal symbols N such that, if $A \to B\alpha$ in P, then $A < B$.
Proof: Let r be a relation on the set N where ArB holds if and only if $A \Rightarrow^+ B\alpha$ for some $\alpha \in (N \cup \Sigma)^*$. Definition of the left recursion implies, that r is a partial order (r is transitive). Each partial order then can be extended to a linear order. $\Box$

Now we formulate an algorithm for the transformation of a grammar G into Greibach normal form.

Algorithm 4.8 The transformation into Greibach normal form.

Input: A proper grammar without left recursion $G = (N, \Sigma, P, S)$

Output: An equivalent grammar G' in GNF

Method:

(1) According to lemma 4.1 construct a linear order $<$ on N such that each A-rule starts with a terminal or a non-terminal B where $A < B$. Let
$$N = \{A_1, A_2, \ldots, A_n\} \text{ and } A_1 < A_2 < \cdots < A_n.$$

(2) Put $i = n - 1$

(3) If $i = 0$ go to step (5), if $i \neq 0$ replace each rule of form $A_i \to A_j\alpha$, where $j > i$ with the rules $A_i \to \beta_1\alpha \mid \ldots \mid \beta_m\alpha$, where $\to \beta_1 \mid \ldots \mid \beta_m$ are all the A_j-rules. (Each string $\beta_1 \ldots \beta_m$ starts with a terminal.)

(4) Put $i = i - 1$ and repeat step (3).

(5) In this moment all the rules (with exception of the rule $S \to \epsilon$) start with a terminal symbol. In each rule $A \to aX_1 \ldots X_k$ replace those symbols X_j, which are terminal symbols, with a new non-terminal X'_j.

(6) For each X'_j from step (5) add the rules $X'_j \to X_j$

Theorem 4.10 Let L be a context-free language. Then there is a grammar G in GNF such that $L = L(G)$.
Proof: Definition of the order $<$ implies, that all the A_n-rules start with a terminal, and that after repetitive processing of step (3) of the algorithm 4.8 all the A_i-rules for $i = 1, \ldots, n-1$ start with terminal symbols, too. Steps (5) and (6) transform the non-start terminal symbols of the right sides to the non-terminals, without changing the generated language. $\Box$

Example 4.19 Let us transform a grammar G with the rules

$$\begin{aligned}
E &\rightarrow T \mid TE' \\
E' &\rightarrow +T \mid +TE' \\
T &\rightarrow F \mid FT' \\
T' &\rightarrow *F \mid *FT' \\
F &\rightarrow (E) \mid i \qquad \text{into GNF.}
\end{aligned}$$

Solution: According to lemma 4.1 $E < T < F$. As the linear order on the set of non-terminals we take the order

$$E' < E < T' < T < F$$

All the F-rules start with a terminal (corollary of the fact, that F is the greatest element in the order $<$). The preceding symbol T has the rules $T \rightarrow F \mid FT'$ and after application of step (3) we get the rules $T \rightarrow (E) \mid i \mid (E)T' \mid iT'$.

The resulting grammar in GNF contains rules:

$$\begin{aligned}
E &\rightarrow (E)' \mid i \mid (E)'T' \mid iT' \mid (E)'E' \mid iE' \mid (E)'T'E' \mid iT'E' \\
E' &\rightarrow +T \mid +TE' \\
T &\rightarrow (E)' \mid i \mid (E)'T' \mid iT' \\
T' &\rightarrow *F \mid *FT' \\
F &\rightarrow (E)' \mid i \\
)' &\rightarrow)
\end{aligned}$$

Disadvantage of the algorithm 4.8 is a large amount of new rules. There is an alternative algorithm of transformation into GNF [1], which does not increase the number of rules of grammar so much. On the other hand it introduces more non-terminals.

Context-free grammars allow description of the most constructs of current high-level programming languages. The way of obtaining a derivation tree for a sentence in a given context-free grammar is the basis of syntactic analysis and classification of syntactic analyzers, compilers and language processors. A number of transformations of the context-free languages and the existence of normal forms of context-free languages allows eliminating some of the non desired properties of grammars or derivations in the grammar. These transformations are often used in practical applications and also within the frame of proof techniques.

4.8 Exercises

Exercise 4.8.1 Modify the grammar G_{RV} from example 4.10 in such way, that notation of regular expressions would also allow:

(a) explicit use of the operator $\cdot$ concatenation (for example $a \cdot b$)

(b) use of the operator $^+$ in sense $r^+ = rr^*$

Exercise 4.8.2 The grammar $G = (\{\langle\texttt{declaration}\rangle, \langle\texttt{options}\rangle, \langle\texttt{option}\rangle, \langle\texttt{mode}\rangle, \langle\texttt{scale}\rangle, \langle\texttt{precision}\rangle, \langle\texttt{base}\rangle\}, \{\texttt{declare}, \texttt{id}, \texttt{real}, \texttt{complex}, \texttt{fixed}, \texttt{floating}, \texttt{single}, \texttt{double}, \texttt{binary}, \texttt{decimal}\}, P, \langle\texttt{declaration}\rangle)$ with the rules

$$\begin{array}{rcl}
\langle\texttt{declaration}\rangle & \rightarrow & \texttt{declare id } \langle\texttt{options}\rangle \\
\langle\texttt{options}\rangle & \rightarrow & \langle\texttt{option}\rangle\ \langle\texttt{options}\rangle \mid \epsilon \\
\langle\texttt{option}\rangle & \rightarrow & \langle\texttt{mode}\rangle \mid \langle\texttt{scale}\rangle \mid \langle\texttt{precision}\rangle \mid \langle\texttt{base}\rangle \\
\langle\texttt{mode}\rangle & \rightarrow & \texttt{real} \mid \texttt{complex} \\
\langle\texttt{scale}\rangle & \rightarrow & \texttt{fixed floating} \\
\langle\texttt{precision}\rangle & \rightarrow & \texttt{single} \mid \texttt{double} \\
\langle\texttt{base}\rangle & \rightarrow & \texttt{binary} \mid \texttt{decimal}
\end{array}$$

describes a declaration of a simple variable. The grammar though allows to create declarations, which contain redundant or contradictory information, for instance

```
declare X real fixed real floating
```

Write a grammar, which allows only declarations without redundancies and contradictions. Is it suitable to solve the problem by definition of the syntax of the language?

Exercise 4.8.3 For a computation of transitive closure of a binary relation defined on a finite set (for example on a dictionary of a grammar) the representation of relation via a boolean matrix and operations on the matrix is very often used. If A is a boolean matrix representing a binary relation R on a set M (i.e. $A[p,q] = \text{true}$, if $(p,q) \in R$ and $A[p,q] = \text{false}$, if $(p,q) \notin R$, $p,q \in M$ and $A[p,q]$ is an element of the matrix A in a row denoted by p and a column denoted by q), then the transitive closure of the relation R is the relation R^+, which is represented by matrix A^+:

$$A^+ = A + A^2 + \ldots + A^n, \tag{3}$$

where $n = \min(|M| - 1, |R|)$ and the operations of summation, resp. multiplication are interpreted as disjunction, resp. conjuction. For the computation of transitive closure there is a more efficient way, than computation according to 3, called, *Warshall algorithm.*

Algorithm 4.9 Warshall algorithm for computation of the transitive closure of a binary relation.

Input: A boolean matrix A representing binary relation R.

Output: A boolean matrix B representing binary relation R^+.

Method:

```
(1) Put $B = A$ and $i = 1$.
(2) For each $j$, if $B[j,i] = $ true, then put $B[j,k] = B[j,k] + B[i,k]$
    for $k = 1, \ldots, n$.
(3) Put $i = i + 1$.
(4) If $i \leq n$, go to step (2); in the other case $B$ is the resulting
    matrix.
```

Show, that the algorithm 4.9 computes indeed the transitive closure of a binary relation.

Exercise 4.8.4 Construct a leftmost and a rightmost derivation and a derivation tree of the sentence $i * (i + i - i)$ in the grammar from example 4.3. Find all phrases, simple phrases and l-phrases of the sentence.

Exercise 4.8.5 Let $G = (N, \Sigma, P, S)$ be a grammar and $\alpha_1, \alpha_2 \in (N \cup \Sigma)^*$ strings. Design algorithms, which find out, if $\alpha_1 \underset{G}{\Rightarrow}^* \alpha_2$ resp. $\alpha_1 \underset{G}{\Rightarrow} \alpha_2$ holds.

Exercise 4.8.6 Let $G = (\{S, A, B\}, \{a, b\}, P, S)$ be a grammar, with the rules

$$\begin{array}{rcl} S & \rightarrow & bA \mid aB \\ A & \rightarrow & a \mid aS \mid bAA \\ B & \rightarrow & b \mid bS \mid aBB \end{array}$$

Show, that the grammar is ambiguous (consider for example the sentence $aabbab$). Try to find an equivalent unambigious grammar.

Exercise 4.8.7 A grammar with the rules

$$\begin{array}{rcl} S & \rightarrow & A \mid B \\ A & \rightarrow & aB \mid bS \mid b \\ B & \rightarrow & AB \mid Ba \\ C & \rightarrow & AS \mid b \end{array}$$

transform in an equivalent grammar without useless symbols.

Exercise 4.8.8 Find a grammar without ϵ-rules, which is equivalent to the grammar

$$\begin{aligned} S &\rightarrow ABC \\ A &\rightarrow BB \mid \epsilon \\ B &\rightarrow CC \mid a \\ C &\rightarrow AA \mid b \end{aligned}$$

Exercise 4.8.9 For the grammar

$$\begin{aligned} S &\rightarrow A \mid B \\ A &\rightarrow C \mid D \\ B &\rightarrow D \mid E \\ C &\rightarrow S \mid a \mid \epsilon \\ D &\rightarrow S \mid b \\ E &\rightarrow S \mid c \mid \epsilon \end{aligned}$$

find an equivalent proper grammar.

Exercise 4.8.10 Formulate an algorithm, which tests, if the grammar G is or is not a cycle-free grammar.

Exercise 4.8.11 Eliminate the left recursion in the grammar

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow BS \mid b \\ B &\rightarrow SA \mid a \end{aligned}$$

Exercise 4.8.12 Transform the grammar

$$G = (\{S, T, L\}, \{a, b, +, -, *, /, [,]\}, P, S)$$

where P contains the rules

$$\begin{aligned} S &\rightarrow T + S \mid T - S \mid T \\ T &\rightarrow L * T \mid L/T \mid L \\ L &\rightarrow [S] \mid a \mid b \end{aligned}$$

into Chomsky normal form.

Exercise 4.8.13 Transform the grammar $G = (\{S, A, B\}, \{a, b\}, P, S)$ with the rules

$$\begin{aligned} S &\rightarrow Ba \mid Ab \\ A &\rightarrow Sa \mid AAb \mid a \\ B &\rightarrow Sb \mid BBa \mid b \end{aligned}$$

into Greibach normal form.

Chapter 5

Pushdown automata and their relation to context-free languages

10:00

The goal of this chapter is acquiring means and methods for alternative description of the context-free languages by various kinds of pushdown automata. We will prove the relation between context-free grammars and nondeterministic and deterministic pushdown automata. The chapter leads to rigorous understanding properties of context-free languages and introduces some further knowledge about the theory of context-free languages.

A pushdown automaton is a one-way nondeterministic automaton fitted with a stack representing an infinite memory. A scheme of a pushdown automaton is presented on the figure 5.1.

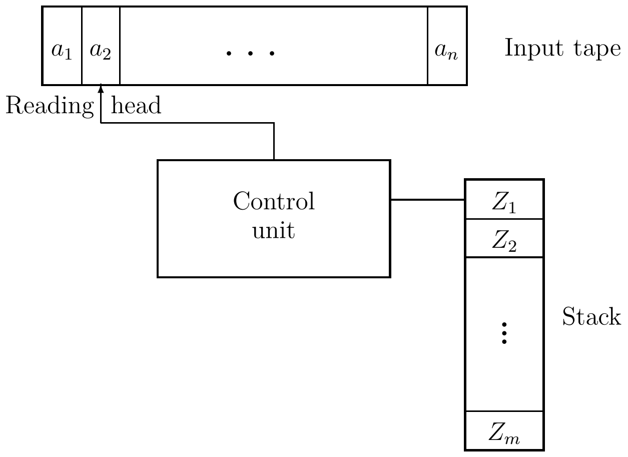


Figure 5.1: Scheme of a pushdown automaton

An input tape is divided into unit records, which contain exactly one input symbol each. A content of a unit record can be read by reading head, but can not be rewritten (read only input tape). In a certain time slot the reading head can stay on the given record or move one record to right (one-way automaton). A finite control unit performs the operations of moving the reading head and storing information into the stack according to a transition function defined over an input alphabet, a set of states of the control unit and a symbol at the top of the stack.

5.1 Definition of a pushdown automaton

Definition 5.1
A pushdown automaton PDA P is a 7-tuple

$$P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F), \text{ where}$$

(1) Q is a finite set of state symbols representing inner states of the control unit

(2) Σ is a finite input alphabet; its items are input symbols

(3) Γ is a finite alphabet of stack symbols

(4) δ is a mapping from a set $Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma$ to a finite set of subsets of set $Q \times \Gamma^*$ describing a transition function

(5) $q_0 \in Q$ is an initial state of the control unit

(6) $Z_0 \in \Gamma$ is a symbol, which is stored into the stack at the beginning of computation – so-called *start symbol* of the stack

(7) $F \subseteq Q$ is a set of final (accepting) states.

Definition 5.2 *A configuration* of a pushdown automaton P is a triplet

$$(q, w, \alpha) \in Q \times \Sigma^* \times \Gamma$$

where

(1) q is a current state of the control unit

(2) w is yet unread part of input string; the first symbol of w is under the reading head. If $w = \epsilon$, then all the symbols of input string have been already read.

(3) α is a content of the stack. Unless otherwise indicated, we represent the stack as a string, whose left most symbol corresponds to the top of the stack. If $\alpha = \epsilon$, then the stack is empty.

Definition 5.3 *A transition* of a pushdown automaton P is represented by a binary relation $\vdash_P$ (or $\vdash$ in case it is obvious, that the automaton P is considered), which is defined on a set of configurations of the pushdown automaton P. The relation

$$(q, aw, Z\alpha) \vdash (q', w, \gamma\alpha)$$

holds, if $\delta(q, a, Z)$ contains the item (q', γ) for some $q \in Q, a \in (\Sigma \cup \{\epsilon\}), w \in \Sigma^*, Z \in \Gamma$ and $\alpha, \gamma \in \Gamma^*$.

We interpret the relation $\vdash_P$ this way. If the pushdown automaton P is in a state q and a symbol Z is at the top the stack, then after reading an input symbol $a \neq \epsilon$, the automaton can move to a state q',whereas the reading head moves to right and the symbol Z from the top of the stack is replaced by a string γ.

If $a = \epsilon$, the reading head does not move, which means, that the automaton transitions to a new state and a new content of the stack is not determined by the next input symbol. This type of transition is called *ϵ-transition*. Note that ϵ-transition can happen also when all the input symbols have been already read.

The relations $\vdash^i, \vdash^+, \vdash^*$ are defined as usual. The relation $\vdash^+$, resp. $\vdash^*$ is transitive resp. transitive and reflexive closure of the relation $\vdash$, $\vdash^i$ is i-th power of the relation $\vdash, i \geq 0$.

An initial configuration of a pushdown automaton is of form (q_0, w, Z_0) for $w \in \Sigma^*$, e.g. the automaton is in the initial state q_0, the string w is on the input tape and the start symbol Z_0 is on the stack. *A final configuration* is of form (q, ϵ, α), where $q \in F$ is the final state and $\alpha \in \Gamma^*$.

Definition 5.4 If a relation $(q_0, w, Z_0) \vdash^* (q, \epsilon, \alpha)$ holds for some $q \in F$, $\alpha \in \Gamma^*$ and a string $w \in \Sigma^*$, then w is *accepted* by the pushdown automaton P. The set $L(P)$ of all strings accepted by the pushdown automaton P, is called *language accepted by the pushdown automaton.*

DEF

Example 5.1 Consider the language $L = \{0^n 1^n \mid n \geq 0\}$. The pushdown automaton P, which accepts the language L, is

x+y

$$P = (\{q_0, q_1, q_2\}, \{0, 1\}, \{Z, 0\}, \delta, q_0, Z, \{q_0\})$$

where

$$\begin{aligned}
\delta(q_0, 0, Z) &= \{(q_1, 0Z)\} \\
\delta(q_1, 0, 0) &= \{(q_1, 00)\} \\
\delta(q_1, 1, 0) &= \{(q_2, \epsilon)\} \\
\delta(q_2, 1, 0) &= \{(q_2, \epsilon)\} \\
\delta(q_2, \epsilon, Z) &= \{(q_0, \epsilon)\}
\end{aligned}$$

The automaton works as follows. It copies all symbols 0 from an input string to the stack. If a symbol 1 appears, then it removes one symbol 0 from the stack. Besides this, the automaton requires that all the symbols 0 go before all the symbols 1. For example for the input string 0011 the automaton P executes these transitions:

$$\begin{aligned}
(q_0, 0011, Z) &\vdash (q_1, 011, 0Z) \\
&\vdash (q_1, 11, 00Z) \\
&\vdash (q_2, 1, 0Z) \\
&\vdash (q_2, \epsilon, Z) \\
&\vdash (q_0, \epsilon, \epsilon)
\end{aligned}$$

We show that $L(P) = L$ really holds. From the definition of the transition function δ and the relation $\vdash$ it follows that $(q_0, \epsilon, Z) \vdash^0 (q_0, \epsilon, Z)$ and hence $\epsilon \in L(P)$. Further

$$\begin{array}{rcl}
(q_0, 0, Z) & \vdash & (q_1, \epsilon, OZ) \\
(q_1, 0^i, 0Z) & \vdash^i & (q_1, \epsilon, 0^{i+1}Z) \\
(q_1, 1, 0^{i+1}Z) & \vdash & (q_2, \epsilon, 0^iZ) \\
(q_2, 1^i, p^iZ) & \vdash^i & (q_2, \epsilon, Z) \\
(q_2, \epsilon, Z) & \vdash & (q_2, \epsilon, Z)
\end{array}$$

holds. Together

$$(q_0, 0^n1^n, Z) \overset{2n+1}{\vdash} (q_0, \epsilon, \epsilon) \qquad \text{for } n \geq 1$$

and hence $L \subseteq L(P)$.

The proof of the reverse inclusion $L(P) \subseteq L$, e.g. the proof, that the pushdown automaton accepts only the strings of form $0^n1^n, n \geq 1$, is more difficult.

If the pushdown automaton P accepts a input string then it has to pass the sequence of states q_0, q_1, q_2, q_0. If $(q_0, w, Z) \vdash^i (q_1, \epsilon, \alpha)$ then $w = 0^i$ and $\alpha = 0^iZ$. Similarly if $(q_2, w, \alpha) \vdash^i (q_2, \epsilon, \beta)$ then $w = 1^i$ and $\alpha = 0^i\beta$. However it means, that the relation $(q_1, w, \alpha) \vdash (q_2, \epsilon, \beta)$ holds, if and only if $w = 1$ and $\alpha = 0\beta$ and the relation $(q_2, w, Z) \vdash^* (q_0, \epsilon, \epsilon)$ holds, if and only if $w = \epsilon$. Thus if $(q_0, w, Z) \vdash^i (q_0, \epsilon, \alpha)$ holds for some $i > 0$, then $w = 0^n1^n, i = 2n + 1$ and $\alpha = \epsilon$, i.e. $L(P) \subseteq L$.

5.2 Various types of pushdown automata

In this paragraph we introduce two types of pushdown automata, which help us to make relation between pushdown automata and context-free languages clear.

Definition 5.5 *An extended pushdown automata EPDA* is a 7-tuple

DEF

$$P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

where δ is a mapping from a finite subset $Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma^*$ to a set of subsets of $Q \times \Gamma^*$. The rest of the symbols have the same meaning as in the basic definition 5.1.

The relation $(q, aw, \alpha\gamma) \vdash (q', w, \beta\gamma)$ holds, if $\delta(q, a, \alpha)$ contains (q', β) for $q \in Q, a \in \Sigma \cup \{\epsilon\}$ and $\alpha \in \Gamma^*$. This relation corresponds to a transition, in which a top string of the stack is removed and replaced by the string β. (Note that in the basic definition there is $\alpha \in \Gamma$ instead of $\alpha \in \Gamma^*$). The language defined by an automaton P is

$$L(P) = \{w \mid (q_0, w, Z_0) \overset{*}{\vdash} (q, \epsilon, \alpha), q \in F, \alpha \in \Gamma^*\}$$

The extended pushdown automaton can execute, unlike the basic definition, transitions also in case the stack is empty.

Example 5.2 Consider the extended pushdown automaton P accepting the language $L = \{ww^R \mid w \in \{a,b\}^*\}$.

$\boxed{\texttt{x+y}}$

$$P = (\{q,p\},\{a,b\},\{a,b,S,Z\},\delta,q,Z,\{p\})$$

where the mapping δ is defined as follows:

$$\begin{aligned}
\delta(q,a,\epsilon) &= \{(q,a)\}\\
\delta(q,b,\epsilon) &= \{(q,b)\}\\
\delta(q,\epsilon,\epsilon) &= \{(q,S)\}\\
\delta(q,\epsilon,aSa) &= \{(q,S)\}\\
\delta(q,\epsilon,bSb) &= \{(q,S)\}\\
\delta(q,\epsilon,SZ) &= \{(p,\epsilon)\}
\end{aligned}$$

For an input string $aabbaa$ the automaton P performs this transitions:

$$\begin{aligned}
(q,aabbaa,Z) &\vdash (q,abbaa,aZ)\\
&\vdash (q,bbaa,aaZ)\\
&\vdash (q,baa,baaZ)\\
&\vdash (q,baa,SbaaZ)\\
&\vdash (q,aa,bSbaaZ)\\
&\vdash (q,aa,SaaZ)\\
&\vdash (q,a,aSaaZ)\\
&\vdash (q,a,SaZ)\\
&\vdash (q,\epsilon,aSaZ)\\
&\vdash (q,\epsilon,SZ)\\
&\vdash (q,\epsilon,\epsilon)
\end{aligned}$$

We can see that the automaton P works this way: at first it stores into the stack a prefix of the input string, afterwards the automaton stores on the top of the stack the symbol S denoting a middle of the input string; then the top string aSa or bSb is replaced by the symbol S. This example illustrates a nondeterministic character of the pushdown automaton, because the mapping does not determine, whether an input symbol will be read and store on the top of the stack ($\delta(q,x,\epsilon) = (q,x), x \in \{a,b\}$), or whether ϵ-transition will be performed and the central symbol S will be stored on the top of the stack ($\delta(q,\epsilon,\epsilon) = (q,S)$).

Definition of extended pushdown automaton implies, that if a language L is accepted by a pushdown automaton, then it is also accepted by an extended pushdown automaton. We prove that also the opposite implication holds.

Theorem 5.1 Let $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ is an extended pushdown automaton. Then there is a pushdown automaton P_1 such that $L(P_1) = L(P)$.

Proof: Let $m = \max\{|\alpha| \mid \delta(q, a, \alpha) \neq \emptyset$ for some $q \in Q$ and $\alpha \in \Gamma^*\}$ A pushdown automaton P_1 will be constructed to simulate the automaton P. Because the automaton P does not determine transitions according to the top of the stack but according to a top string of the stack the automat P_1 will store m top symbols in kind of "buffer" of the control unit in order to know which m top symbols are in the stack of the automaton P at the beginning of every transition. If the automaton P replaces k top symbols by a string of length l, then the automaton P_1 does the same in the buffer. If $l < k$, then P_1 performs $(k-l)$ ϵ-transitions, which moves $(k-l)$ symbols from the top of the stack to the buffer. The automaton P_1 can afterwards simulate the next transition of the automaton P. If $l \geq k$, then the symbols are moved from the buffer to the stack.

A construction of the pushdown automaton P_1 is formally described as follows.

$$P_1 = (Q_1, \Sigma_1, \Gamma_1, \delta_1, q_1, Z_1, F_1) \qquad \text{where}$$

(1) $Q_1 = \{[q, \alpha] \mid q \in Q, \alpha \in \Gamma_1^*$ a $0 \leq |\alpha| \leq m\}$

(2) $\Gamma_1 = \Gamma \cup \{Z_1\}$

(3) The mapping δ_1 is defined this way:

a) Let us assume that $\delta(q, a, X_1 \dots X_k)$ contains $(r, Y_1 \dots Y_l)$.

(i) If $l \geq k$, then for all $Z \in \Gamma_1$ a $\alpha \in \Gamma_1^*$ such that $|\alpha| = m - k$

$$\delta_1([q, X_1 \dots X_k\alpha], a, Z) \text{ obsahuje } ([r, \beta], \gamma Z)$$

where $\beta\gamma = Y_1 \dots Y_l\alpha$ a $|\beta| = m$.

(ii) If $l < k$, then for all $Z \in \Gamma_1$ a $\alpha \in \Gamma_1^*$ such that $|\alpha| = m - k$

$$\delta_1([q, X_1 \dots X_k\alpha], a, Z) \text{ contains } ([r, Y_1 \dots Y_l\alpha Z], \epsilon)$$

b) For all $q \in Q, Z \in \Gamma_1$ a $\alpha \in \Gamma_1^*$ such that $|\alpha| < m$

$$\delta_1([q, \alpha], \epsilon, Z) = \{([q, \alpha, Z], \epsilon)\}$$

These rules lead to filling the buffer (containing m symbols).

(4) $q_1 = [q_0, Z_0, Z_1^{m-1}]$. At the beginning the buffer contains the symbol Z_0 on the top and $m-1$ symbols Z_1 on the next positions. The symbols Z_1 are special markers denoting the bottom of the stack.

(5) $F_1 = \{[q, \alpha] \mid q \in F, \alpha \in \Gamma_1^*\}$

It is possible to show that

$$(q, aw, X_1 \dots X_k X_{k+1} \dots X_n) \underset{P}{\vdash} (r, w, Y_1 \dots Y_l X_{k+1} \dots X_n)$$

holds if and only if $([q, \alpha], aw, \beta) \vdash^{+}_{P_1} ([r, \alpha'], w, \beta')$ where

$$\begin{aligned} \alpha\beta &= X_1 \ldots X_n Z_1^m \\ \alpha'\beta' &= Y_1 \ldots Y_l X_{k+1} \ldots X_n Z_1^m \\ |\alpha| &= |\alpha'| = m \end{aligned}$$

and between these two configurations of the automaton P_1 there is no configuration in which the buffer has length m.

Hence a relation

$$(g_0, w, Z_0) \underset{P}{\vdash} (q, \epsilon, \alpha) \qquad \text{for } q \in F, \alpha \in \Gamma^*$$

holds if and only if

$$([q_0, Z_0, Z_1^{m-1}], w, Z_1) \overset{*}{\underset{P_1}{\vdash}} ([q, \beta], \epsilon, \gamma)$$

where $|\beta| = m$ a $\beta\gamma = \alpha Z_1^m$. Thus $L(P) = L(P_1)$. □

DEF

Definition 5.6 Let $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, \emptyset)$ be a pushdown or an extended pushdown automaton. A string w is accepted by *empty stack*, if $(q_0, w, Z_0) \vdash^{+} (q, \epsilon, \epsilon)$ holds. Let us denote $L_\epsilon(P)$ the set of all strings which are accepted pushdown automaton P by empty stack.

Theorem 5.2 Let L be a language accepted by a pushdown automaton P where $P = (Q, \Sigma, \Gamma, \delta,\ q_0, Z_0,\ F), L = L(P)$. It is possible to construct a pushdown automaton P' such that $L_\epsilon(P') = L$.

Proof: We will construct the automaton P' to simulate the automaton P. Any time the automaton P reaches the accepting state, the automaton P' moves to a special state q_ϵ, where its stack is cleaned out. However, we have to consider a situation where the automaton P reaches a configuration with empty stack, but it is not in a final state. To avoid the cases when the automaton P' accepts some string, which is not accepted by the automaton P, we add to the stack's alphabet of the automaton P' a special marker. This marker denotes the bottom of the stack and can be popped from the stack only if the automaton P' is in the state q_ϵ. Formally, let

$$P = (Q \cup \{q_\epsilon, q'\}, \Sigma, \Gamma \cup \{Z'\}, \delta', q', Z', \emptyset), \quad \text{where}$$

the symbol $\emptyset$ denotes that the automaton P accepts a string by empty stack. The mapping δ is defined as follows :

(1) if $\delta(q, a, Z)$ contains (r, γ), then $\delta'(q, a, Z)$ contains (r, γ) for all $q \in Q, a \in \Sigma \cup \{\epsilon\}$ and $Z \in \Gamma$.

(2) $\delta(q', \epsilon, Z') = \{(q_0, Z_0 Z')\}$. The first transition of the pushdown automaton P' causes storing the string $Z_0 Z'$ into the stack, where Z' is a special marker denoting the the bottom of the stack. Afterwards the automaton moves to the initial state of the automaton P.

(3) For each $q \in F$ and $Z \in \Gamma \cup \{Z'\}$ $\delta'(q, \epsilon, Z)$ contains the item (q_ϵ, ϵ).

(4) For each $Z \in \Gamma \cup \{Z'\}$ where $\delta(q_\epsilon, \epsilon, Z) = \{(q_\epsilon, \epsilon)\}$.

Now, obviously

$$\begin{aligned}(q', w, Z') &\vdash_{P'} (q_0, w, Z_0 Z')\\ &\vdash^{n}_{P'} (q, \epsilon, Y_1 \dots Y_r)\\ &\vdash_{P'} (q_\epsilon, \epsilon, Y_2 \dots Y_r)\\ &\vdash^{r-1}_{P'} (q_\epsilon, \epsilon, \epsilon) \qquad \text{where } Y_r = Z',\end{aligned}$$

holds if and only if

$$(q_0, w, Z_0) \overset{n}{\underset{P}{\vdash}} (q, \epsilon, Y_1 \dots Y_{r-1})$$

for $q \in F$ and $Y_1 \dots Y_{r-1} \in \Gamma^*$. Thus $L_\epsilon(P') = L(P)$.

The previous theorem holds also conversely. □

Theorem 5.3 Let $P = (q, \Sigma, \Gamma, \delta, q_0, Z_0, \emptyset)$ be a pushdown automaton. It is possible to construct a pushdown automaton P' such that $L(P) = L_\epsilon(P')$.

Proof: The pushdown automaton P' has a special marker Z' denoting the bottom of the stack. If the P's stack is empty (the marker Z' is on the top of the stack of the automaton P') the automaton P' moves to the new accepting state $q_f : F' = \{q_f\}$. We leave the formal construction of the automaton P' as an exercise. □

5.3 Equivalence of context-free languages and languages accepted by pushdown automata

First of all we show a construction of a nondeterministic top-down parser for a language generated by a context-free grammar.

Theorem 5.4 Let $G = (N, \Sigma, P, S)$ be a context-free grammar. We can construct a pushdown automaton R such that $L_\epsilon(R) = L(G)$ from the grammar G .

Proof: The pushdown automaton R is constructed in order to form a leftmost derivation of an input string in the grammar G. Let $R = (\{q\}, \Sigma, N \cup \Sigma, \delta, S, \emptyset)$, where δ is defined as follows:

(1) If $A \to \alpha$ is a rule from P, then $\delta(q, \epsilon, A)$ contains (q, α).

(2) $\delta(q, a, a) = \{(q, \epsilon)\}$ for all $a \in \Sigma$.

Now we show that
$A \Rightarrow^m w$, if and only if $(q, w, \epsilon) \vdash^n (q, \epsilon, \epsilon)$ for some $m, n \geq 1$.

The "only if" part we prove by induction with respect to m. Let us assume that $A \Rightarrow^m w$. If $m = 1$ and $w = a_1 \ldots a_k, k \geq 0$ then

$$\begin{array}{rcl} (q, a_1 \ldots a_k, A) & \vdash & (q, a_1 \ldots a_k) \\ & \vdash^k & (q, \epsilon, \epsilon) \end{array}$$

Now assume that $A \Rightarrow^m w$ holds for $m > 1$. The first step of this derivation has to be of form: $A \Rightarrow X_1 X_2 \ldots X_k$, whereas $X_i \Rightarrow^{m_i} x_i$ pro $m_i < m, 1 \leq i \leq k$ and $x_1 x_2 \ldots x_k = w$. Hence

$$(q, w, A) \vdash (q, w, X_1 \ldots X_k)$$

If $X_i \in N$, then according to the induction hypothesis

$$(q, x_i, X_i) \overset{*}{\vdash} (q, \epsilon, \epsilon)$$

If $X_i = x_i, x_i \in \Sigma$, then

$$(q, x_i, X_i) \vdash (q, \epsilon, \epsilon)$$

Finely we can see that the sequence of transitions

$$(q, x_1 \ldots x_k, A) \vdash (q, x_1 \ldots x_k, X_1 \ldots X_k) \overset{*}{\vdash} (q, x_2 \ldots x_k, X_2 \ldots X_k) \ldots \overset{*}{\vdash} (q, \epsilon, \epsilon)$$

correspond to the leftmost derivation

$$A \Rightarrow X_1 \ldots X_k \Rightarrow^{m_1} x_1 X_2 \ldots X_k \Rightarrow^{m_2} \ldots \Rightarrow^{m_k} x_1 x_2 \ldots x_k = w$$

The "if" part, i.e.:
if $(q, w, A) \vdash^n (q, \epsilon, \epsilon)$, then $\Rightarrow^+ w$
we prove by induction with respect to n. For $n = 1, w = \epsilon$ and $A \rightarrow \epsilon$ is a rule in P. Let us assume that the proven relation holds for each $n' < n$. Then the first transition of the automaton R has to be of form:

$$(q, w, A) \vdash (q, w, X_1 \ldots X_k)$$

and $(q, x_i, X_i) \vdash^{n_i} (q, \epsilon, \epsilon)$ for $1 \leq i \leq k$, where $w = x_1 \ldots x_k$. Afterwards the rule $A \rightarrow X_1 \ldots X_k$ is in P and the derivation $X_i \Rightarrow^+ x_i$ follows from the induction hypothesis. If $X_i \in \Sigma$ then $X_i \Rightarrow^0 x_i$ and thus
$A \Rightarrow X_1 \ldots X_k \Rightarrow^* x_1 X_2 \ldots X_k \Rightarrow^* \ldots \Rightarrow^* x_1 \ldots x_{k-1} x_k = w$
is the leftmost derivation of the string w from A. If we take $S = A$ as a special case, we obtain $S \Rightarrow^+ w$, if and only if $(q, w, s) \vdash^+ (q, \epsilon, \epsilon)$ and thus $L_\epsilon(R) = L(G)$.

□

Example 5.3 We construct a pushdown automaton P such that $L_\epsilon(P) = L(G)$ to the grammar G with the rules

$$\begin{array}{rcl} E & \to & E + T \mid T \\ T & \to & T * F \mid F \\ F & \to & (E) \mid i \end{array}$$

Solution:

$$P = (\{q\}, \{+, *, (,), i\}, \{+, *, (,), i, E, T, F\}, \delta, q, E, \emptyset)$$

where δ is the mapping

$$\begin{array}{rcl} \delta(q, \epsilon, E) & = & \{(q, E + T), (q, T)\} \\ \delta(q, \epsilon, T) & = & \{(q, T * F), (q, F)\} \\ \delta(q, \epsilon, F) & = & \{(q, (E)), (q, i)\} \\ \delta(q, a, a) & = & \{(q, \epsilon)\} \quad \text{for all} \quad a \in \{+, *, (,), i\} \end{array}$$

For the input string $i + i * i$ the pushdown automaton P performs this sequence of the transitions:

$$\begin{array}{rcl} (q, i + i * i, E) & \vdash & (q, i + i * i, E + T) \\ & \vdash & (q, i + i * i, T + T) \\ & \vdash & (q, i + i * i, F + T) \\ & \vdash & (q, i + i * i, i + T) \\ & \vdash & (q, +i * i, +T) \\ & \vdash & (q, i * i, T) \\ & \vdash & (q, i * i, T * F) \\ & \vdash & (q, i * i, F * F) \\ & \vdash & (q, i * i, i * F) \\ & \vdash & (q, *i, *F) \\ & \vdash & (q, i, F) \\ & \vdash & (q, i, i) \\ & \vdash & (q, \epsilon, \epsilon) \end{array}$$

Now we show the construction of an extended pushdown automaton performing bottom-up parsing for a languages generated by context-free grammar G.

Theorem 5.5 Let $G = (N, \Sigma, P, S)$ be a context-free grammar. The extended pushdown automaton $R = (\{q, r\}, \Sigma, N \cup \Sigma \cup \{\$\}, \delta, q, \$, \{r\})$, where the mapping δ is defined as follows:

(1) $\delta(q, a, \epsilon) = \{(q, a)\}$ for all $a \in \Sigma$.

(2) If $A \to \alpha$ is a rule from P, then $\delta(q, \epsilon, \alpha)$ contains (q, A).

(3) $\delta(q, \epsilon, S\$) = \{(r, \epsilon)\}$,

accepts the language $L(G)$, e.g. $L(R) = L(G)$.

Proof: We show that the automaton R creates the rightmost derivation by step-by-step reductions of the l-phrase starting with the terminal string (placed on the input tape) and ending with the initial symbol S. The automaton R performs bottom-up parsing.

The change of the convention: The top of the stack we will now indicate *on the right*. The induction hypothesis, which we will prove by induction with respect to n, has the form:

$$S \underset{rm}{\Rightarrow}^{*} \alpha A y \underset{rm}{\Rightarrow}^{n} xy \text{ implies } (q, xy, \$) \overset{*}{\vdash} (q, y, \$\alpha A),$$

where $\underset{rm}{\Rightarrow}$ denotes the right derivation. For $n = 0$ this hypothesis holds, because $\alpha A = x$ and thus the automaton R performs the transitions according to $\delta(q, a, \epsilon) = \{(q, a)\}, a \in \Sigma$, moving the string x into the stack.

Now let us assume that the induction hypothesis holds for each derivation shorter than n. We can write

$$\alpha A \gamma \underset{rm}{\Rightarrow} \alpha \beta y \underset{rm}{\Rightarrow}^{n-1} xy$$

If the string $\alpha\beta$ consists of terminal symbols only then

$$\alpha\beta = x \text{ and } (q, xy, \$) \vdash^{*} (q, y, \$\alpha\beta) \vdash (q, y, \$\alpha A).$$

If $\alpha\beta \notin \Sigma^{*}$, then we can write $\alpha\beta = \gamma B z$, where B is the rightmost nonterminal and according to the induction hypothesis

$$S \underset{rm}{\Rightarrow}^{*} \gamma B z y \underset{rm}{\Rightarrow}^{n-1} xy$$

implies $(q, xy, \$) \vdash^{*} (q, zy, \$\gamma B)$.

Therefore $(q, zy, \$\gamma B) \vdash^{*} (q, y, \$\gamma B z) \vdash (q, y, \$\alpha A)$ holds and thus we proved the induction hypothesis. Because $(q, \epsilon, \$S) \vdash (r, \epsilon, \epsilon)$ we finally obtain $L(G) \subseteq L(R)$.

To prove that $L(R) \subseteq L(G)$ we show that the following implications holds:

$$\text{If } (q, xy, \$) \overset{n}{\vdash} (q, y, \$\alpha A), \text{ then } \alpha A y \Rightarrow^{*} xy$$

For $n = 0$ this implication holds. Consider that, this implication holds also for all sequences of the transitions shorter than n. Because the top of the stack is a nonterminal symbol, the last performed transition corresponds to the rule (2) in the mapping δ and thus we can write:

$$(q, xy, \$) \overset{n-1}{\vdash} (q, y, \$\alpha\beta) \vdash (q, y, \$\alpha A)$$

where $A \to \beta$ is the rule from P. Therefore a derivation $\alpha A y \Rightarrow \alpha\beta y \Rightarrow^{*} xy$ exists.

As a special case consider the implication:

$$\text{if} \quad (q, w, \$) \overset{*}{\vdash} (q, \epsilon, \$S), \quad \text{then} \quad S \Rightarrow^* w$$

However, because $(q, w, \$) \vdash^* (q, \epsilon, \$S) \vdash (r, \epsilon, \epsilon)$ holds, $L(R) \subseteq L(G)$ and together with the first part of the proof we obtain $L(G) = L(R)$.

$\square$

Let us remark, that the automaton R really represents a model of parser which creates the rightmost derivation of the input string by step-by-step reductions of l-phrases of sentential forms (the initial sentential form is the input string and the final sentential form is the initial symbol of the grammar). Immediately after the transition of the automaton the right sentential form $\alpha A x$ is represented by the content of the stack (the string αA) and by the unprocessed part of the input string (the string x). The next operation of the automaton is moving a prefix of the string x on the top of the stack and reduction of the l-phrase given by the top string on the stack. This type of parsing is called, as we have already mentioned, bottom-up parsing.

Example 5.4 Create a bottom-up parser R for the grammar with the rules

$$\begin{array}{rcl} E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * F \mid F \\ F & \rightarrow & (E) \mid i \end{array}$$

Let R be the extended pushdown automaton

$$R = (\{q, r\}, \{+, *, (,), i\}, \{\$, E, T, F, +, *, (,), i\}, \delta, q, \$, \{r\})$$

where δ is the mapping:

(1) $\delta(q, a, \epsilon) = \{(q, a)\}$ for all $a \in \{i, +, *, (,)\}$.

(2)

$$\begin{array}{rcl} \delta(q, \epsilon, E + T) & = & \{(q, E)\} \\ \delta(q, \epsilon, T) & = & \{(q, E)\} \\ \delta(q, \epsilon, T * F) & = & \{(q, T)\} \\ \delta(q, \epsilon, F) & = & \{(q, T)\} \\ \delta(q, \epsilon, (E)) & = & \{(q, F)\} \\ \delta(q, \epsilon, i) & = & \{(q, F)\} \end{array}$$

(3) $\delta(q, \epsilon, \$E) = \{(r, \epsilon)\}$

For the input string $i+i*i$ the extended pushdown automaton R can perform this sequence of the transitions:

$$\begin{aligned}
(q, i+i*i, \$) &\vdash (q, +i*i, \$i) \\
&\vdash (q, +i*i, \$F) \\
&\vdash (q, +i*i, \$T) \\
&\vdash (q, +i*i, \$E) \\
&\vdash (q, i*i, \$E+) \\
&\vdash (q, *i, \$E+i) \\
&\vdash (q, *i, \$E+F) \\
&\vdash (q, *i, \$E+T) \\
&\vdash (q, i, \$E+T*) \\
&\vdash (q, \epsilon, \$E+T*i) \\
&\vdash (q, \epsilon, \$E+T*F) \\
&\vdash (q, \epsilon, \$E+T) \\
&\vdash (q, \epsilon, \$E) \\
&\vdash (q, \epsilon, \epsilon)
\end{aligned}$$

To conlude this paragraph we show that context-free languages are equivalent to the languages accepted by pushdown automata, i.e. it is possible to construct a context-free grammar G to each pushdown automaton R, such that $L(R) = L(G)$.

Theorem 5.6 Let $R = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ be a pushdown automaton. Then there is a context-free grammar G, such that $L(G) = L(R)$.
Proof: We will construct the grammar G such that the leftmost derivation of the terminal string w directly corresponds to the sequence of transitions of the automaton R. We will use nonterminal symbols of form $[qZr]$ where $q, r \in Q, Z \in \Gamma$. The top of the stack is again indicated *on the left*.

Let the grammar $G = (N, \Sigma, P, S)$ be formally defined as follows:

(1) $N = \{[qZr] \mid q, r \in Q, Z \in \Gamma\} \cup \{S\}$

(2) If $\delta(q, a, Z)$ contains $(r, X_1 \dots X_k)$ $k \geq 1$, then add all rules of form

$$[qZs_k] \rightarrow a[rX_1s_1][s_1X_2s_2] \dots [s_{k-1}X_ks_k]$$

to the P for each sequence $s_1, s_2, \dots, s_k$ of the states from the set Q.

(3) If $\delta(q, a, Z)$ contains (r, ϵ), then add the rule $[qZr] \rightarrow a$ to the set P.

(4) For each state $q \in Q$ add the rule $S \rightarrow [q_0Z_0q]$ to the set P.

We can prove by induction that for all $q, r \in Q$ and $Z \in \Gamma$ $[qZr] \Rightarrow^m w$ holds if and only if $(q, w, Z) \vdash^n (r, \epsilon, \epsilon)$. A special case of this equivalence is $S \Rightarrow [q_0Z_0q] \Rightarrow^+ w$, if and only if $(q_0, w, Z_0) \vdash^+ (q, \epsilon, \epsilon), q \in F$. Hence $L_\epsilon(R) = L(G)$. □

Example 5.5 Let $P = (\{q_0, q_1\}, \{0, 1\}, \{X, Z_0\}, \delta, q_0, Z_0, \emptyset)$ where the mapping δ is defined as follows:

x+y

$$\begin{aligned}
\delta(q_0, 0, Z_0) &= \{(q_0, XZ_0)\}\\
\delta(q_0, 0, X) &= \{(q_0, XX)\}\\
\delta(q_0, 1, X) &= \{(q_1, \epsilon)\}\\
\delta(q_1, 1, X) &= \{(q_1, \epsilon)\}\\
\delta(q_1, \epsilon, X) &= \{(q_1, \epsilon)\}\\
\delta(q_0, \epsilon, Z_0) &= \{(q_1, \epsilon)\}
\end{aligned}$$

We counstruct a grammar $G = (N, \Sigma, P, S)$ such that $L(P) = L(G)$. The sets of nonterminals and terminals have the form:

$$\begin{aligned}
N &= \{S, [q_0Xq_0], [q_0Xq_1], [q_1Xq_0], [q_1Xq_1], [q_0Z_0q_0], [q_0Z_0q_1], [q_1Z_0q_0],\\
&\quad [q_1Z_0q_1]\}\\
\Sigma &= \{0, 1\}
\end{aligned}$$

The set N contains some nonterminals which are unreachable in G. To avoid later elimination of these nonterminals we construct the rules of the the grammar G starting with the S-rules and we add only those nonterminals which appear on the right sides of the constructed rules.

According to (4) we construct the S-rules:

$$S \rightarrow [q_0Z_0q_0] \qquad S \rightarrow [q_0Z_0q_1]$$

Now we add the rules for the nonterminal $[q_0Z_0q_0]$

$$\begin{aligned}
[q_0Z_0q_0] &\rightarrow 0[q_0Xq_0][q_0Z_0q_0]\\
[q_0Z_0q_0] &\rightarrow 0[q_0Xq_1][q_1Z_0q_0]
\end{aligned}$$

following from $\delta(q_0, 0, Z_0) = \{(q_0, XZ_0)\}$.

The rules for nonterminal $[q_0Z_0q_1]$

$$\begin{aligned}
[q_0Z_0q_1] &\rightarrow 0[q_0Xq_0][q_0Z_0q_1]\\
[q_0Z_0q_1] &\rightarrow 0[q_0Xq_1][q_1Z_0q_1]
\end{aligned}$$

are requested by the mapping $\delta(q_0, 0, Z_0) = \{(q_0, XZ_0)\}$.

The rules for the remaining nonterminals are:

$$\begin{aligned}
[q_0Xq_0] &\rightarrow 0[q_0Xq_0][q_0Xq_0]\\
[q_0Xq_0] &\rightarrow 0[q_0Xq_1][q_1Xq_0]\\
[q_0Xq_1] &\rightarrow 0[q_0Xq_0][q_0Xq_1]\\
[q_0Xq_1] &\rightarrow 0[q_0Xq_1][q_1Xq_1]
\end{aligned}$$

because, $\delta(q_0, 0, X) = \{(q_0, XX)\}$;

$$\begin{aligned}
[q_0Xq_1] &\rightarrow 1,\ \text{because } \delta(q_0, 1, X) = \{(q_1, \epsilon)\}\\
[q_1Z_0q_1] &\rightarrow \epsilon,\ \text{because } \delta(q_1, \epsilon, Z_0) = \{(q_1, \epsilon)\}\\
[q_1Xq_1] &\rightarrow \epsilon,\ \text{because } \delta(q_1, \epsilon, X) = \{(q_1, \epsilon)\}\\
[q_1Xq_1] &\rightarrow 1,\ \text{because } \delta(q_1, 1, X) = \{(q_1, \epsilon)\}.
\end{aligned}$$

Note that for the nonterminals $[q_1 X q_0]$ and $[q_1 Z_0 q_0]$ no rules are defined and thus no terminal strings can be derived from the nonterminals $[q_0 Z_0 q_0]$ and $[q_0 X q_0]$. If we remove the rules containing some of the four mentioned nonterminals we obtain the equivalent grammar generating the language $L(P)$:

$$\begin{aligned}
S &\rightarrow [q_0 Z_0 q_1] \\
[q_0 Z_0 q_1] &\rightarrow 0[q_0 X q_1][q_1 Z_0 q_1] \\
[q_0 X q_1] &\rightarrow 0[q_0 X q_1][q_1 X q_1] \\
[q_1 Z_0 q_1] &\rightarrow \epsilon \\
[q_0 X q_1] &\rightarrow 1 \\
[q_1 X q_1] &\rightarrow \epsilon \\
[q_1 X q_1] &\rightarrow 1
\end{aligned}$$

5.4 Deterministic pushdown automaton

In the previous paragraph we have shown that for each context-free grammar it is possible to construct a pushdown automaton, which represents parser for the sentences generated by the given grammar. This parser is generally nondeterministic. From the point of view of applications of the theory of formal languages in compilers so-called deterministic context-free languages are important. This languages can be parsed by deterministic parsers. In this paragraph we will define a class of the pushdown automata which can move only to one configuration in every moment, it means we will define deterministic pushdown automata and deterministic languages corresponding to them.

DEF

Definition 5.7 A pushdown automaton $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ is called a deterministic pushdown automaton, if for each $q \in Q$ and $Z \in \Gamma$ the following holds: Either for all $a \in \Sigma$ $\delta(q, a, Z)$ contains at most one item and $\delta(q, \epsilon, Z) = \emptyset$ or $\delta(q, a, Z) = \emptyset$ for all $a \in \Sigma$ and $\delta(q, \epsilon, Z)$ contains at most one item.

Because the mapping $\delta(q, a, Z)$ contains at most one item we will instead of $\delta(q, a, Z) = \{(r, \gamma)\}$ write $\delta(q, a, Z) = (r, \gamma)$.

x+y

Example 5.6 A deterministic pushdown automaton, which accepts the languages
$L = \{wcw^R \mid w \in \{a, b\}^+\}$, is of form:

$$P = (\{q_0, q_1, q_2\}, \{a, b, c\}, \{Z, a, b\}, \delta, q_0, Z, \{q_2\})$$

where the mapping δ is defined as follows:

$$\begin{aligned}
\delta(q_0, X, Y) &= (q_0, XY) \quad \text{for all } X \in \{a, b\} \text{ a } Y \in \{Z, a, b\} \\
\delta(q_0, c, Y) &= (q_1, Y) \quad \text{for all } Y \in \{a, b\} \\
\delta(q_1, X, X) &= (q_1, \epsilon) \quad \text{for all } X \in \{a, b\} \\
\delta(q_1, \epsilon, Z) &= (q_2, \epsilon)
\end{aligned}$$

The automaton P works like this: Until it reads the middle symbol c, it stores he symbols of the input string into the stack. Afterwards the automaton moves to the state q_1 and compares the next input symbols with the symbols on the stack.

We can also define an extended deterministic pushdown automaton.

Definition 5.8 An extended pushdown automaton $R = (q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ is called *an extended deterministic pushdown automaton*, if the following holds:

DEF

(1) For each $q \in Q, a \in \Sigma \cup \{\epsilon\}$ and $\gamma \in \Gamma^*$ $\delta(q, a, \gamma)$ contains at most one item.

(2) If $\delta(q, a, \alpha) \neq \emptyset, \delta(q, a, \beta) \neq \emptyset$ and $\alpha \neq \beta$, the string α is not a suffix of the string β and β is not a suffix of α.

(3) If $\delta(q, a, \alpha) \neq \emptyset$ and $\delta(q, \epsilon, \beta) \neq \emptyset$, then the string α is not a suffix of the string β and β is not a suffix α.

Note 5.1 The definition 5.8 assumes that the top of the stack is on the right, if the top is on the left then we have to replace the word "suffix" with the word "prefix" in the previous definition.

Unfortunately the relation between a pushdown automaton and a deterministic pushdown automaton is not the same as the relation between nondeterministic and deterministic finite automaton. It means that not all languages accepted by the pushdown automaton can be accepted by deterministic pushdown automaton.

Definition 5.9 The language L is called *a deterministic context-free language*, if there is a deterministic pushdown automaton P such that $L(P) = L$.

DEF

Theorem 5.7 A deterministic extended pushdown automaton has equivalent expressive power as a deterministic pushdown automaton.

Theorem 5.8 A deterministic pushdown automaton has strictly smaller expressive power than a nondeterministic pushdown automaton.

Proof: (We show only the idea of the proof)
The context-free language $L = \{ww^R \mid w \in \Sigma^+\}$ can not be accepted by

any deterministic pushdown automaton. Informally a deterministic pushdown automaton have no chance to guess the end of the string w and the beginning of the string w^R.

$\square$

Note 5.2 For the alternative proof of the theorem 5.8 we can use the fact that the class of the deterministic context-free languages is closed to the complement(see in next section). Consider the context free language $\overline{\{a^n b^n c^n | n \geq 1\}}$.

The problem whether a given context-free language is a deterministic context-free language *is generally undecidable* (similar as undecidability of the multivalent).

5.5 Properties of context-free languages

In this section we will deal with some properties of context-free languages in the similar form as in the case of regular languages

5.5.1 Structural properties

Pumping lemma

Theorem 5.9 Let L be a context-free language. Then there is a constant k such that if $z \in L$ and $|z| \geq k$, then z can be written in the form:

$$z = uvwxy, vx \neq \varepsilon, |vwx| \leq k$$

and for all $i \geq 0$ is $uv^i wx^i y \in L$.

Proof: Let $L = L(G)$ and $G = (N, \Sigma, P, S)$ be a grammar in CNF.

1. First we will prove this implication :
   If $A \Rightarrow^+ \alpha$ then $|\alpha| \leq 2^{m-1}$, where m is the number of vertexes of the longest path in the corresponding derivation tree.

   For the proof we use the induction

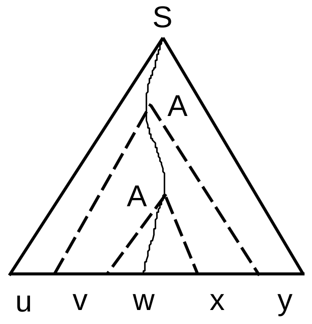
   (a) $m = 2$

   A =>$^+$ a
   $$\begin{tikzcd} A \arrow[d] \\ a \end{tikzcd}$$
   $$\begin{tikzcd} & A \arrow[dl] \arrow[dr] & \\ B & & C \end{tikzcd}$$
   A =>$^+$ BC

   (b) Consider that the statement holds for some m and the longest path contains $m+1$ vertices. Then the rule of form $A \rightarrow BC$ was applied in the first step of a derivation and we can apply the induction hypothesis on the subtrees with the roots B and C. We obtain:

$$|\alpha| \leq 2^{m-1} + 2^{m-1} = 2^{(m-1)+1} = 2^{(m+1)-1}$$

2. Let $|N| = n$ and $k = 2^{n+1}$. Consider an arbitrary sentence z such that $|z| \geq k$. Then the longest path in the corresponding derivation tree contains at least $n + 1$ vertices and necessarily at least 2 from them are marked by the same nonterminal. Denote this nonterminal by symbol A. See the following picture.



The strings v, x can not be empty because the applied rule is of form $A \to BC$. Now consider the derivation of the string z of form:
$S \Rightarrow^* uAy \Rightarrow^+ uvAxy \Rightarrow^+ uvwxy = z$
This means that in the grammar G there is also the derivation:
$S \Rightarrow^* uAy \Rightarrow^+ uvAxy \Rightarrow^+ uvvAxxy \Rightarrow^+ uv^2wx^2y$, because $A \Rightarrow^+ w$, and thus the derivation $S \Rightarrow^* uv^iwx^iy$ for arbitrary $i > 0$, which is to be proved.

$\square$

Lemma 5.1 The language $L = \{a^nb^nc^n \mid n \geq 1\}$ is not a context-free language.

Proof: We will use pumping lemma to prove this statement: We cannot select the strings v and x so that by the iteration of these strings the number of the symbols a, b, c is preserved and at the same time the order of the symbols a, b, c remains unchanged.

$\square$

Note 5.3 The language $L = \{a^nb^nc^n \mid n \geq 1\}$ is a typical context sensitive languages.

5.5.2 Closure properties

Substitution of languages

Definition 5.10 Let $\mathcal{L}$ be a class of languages and $L \subseteq \Sigma^*$ be a language from the class $\mathcal{L}$. Let $\Sigma = \{a_1, a_2, ..., a_n\}$ for some $n \in \mathbb{N}$ and let languages denoted by $L_{a_1}, L_{a_2}, ..., L_{a_n}$ are also languages from the class $\mathcal{L}$. We say that the class $\mathcal{L}$ is *closed under the substitution*, if for each selection of the languages $L, L_{a_1}, L_{a_2}, ..., L_{a_n}$ also the language $\sigma_{L_{a_1}, L_{a_2}, ..., L_{a_n}}(L)$

$$\sigma_{L_{a_1}, L_{a_2}, ..., L_{a_n}}(L) = \{x_1x_2...x_m \mid b_1b_2...b_m \in L \wedge \forall i \in \{1, ..., m\} : x_i \in L_{b_i}\}$$

is in the class $\mathcal{L}$.

Example 5.7 Let $L = \{0^n1^n \mid n \geq 1\}$, $L_0 = \{a\}$, $L_1 = \{b^mc^m \mid m \geq 1\}$. By the substitution of the languages L_0 and L_1 into L we obtain thelanguage

$$L' = \{a^nb^{m_1}c^{m_1}b^{m_2}c^{m_2}...b^{m_n}c^{m_n} \mid n \geq 1 \wedge \forall i \in \{1, ..., n\} : m_i \geq 1\}$$

Morphism of languages

Definition 5.11 Let Σ and Δ be alphabets and $L \subseteq \Sigma^*$ be a language over the alphabet Σ. The mapping $h : \Sigma^* \to \Delta^*$ is called *a morphism over words*, if $\forall w = a_1a_2...a_n \in \Sigma^*$ holds $h(w) = h(a_1)h(a_2)...h(a_n)$. *The morphism of the language* $h(L)$ is then defined this way: $h(L) = \{h(w) \mid w \in L\}$.

Morphism of languages is *a special case of substitution*, where each substituted language has exactly one sentence.

Closure under substitution

Theorem 5.10 *The class of context-free languages is closed under substitution.*

Proof:

1. According to definition of the substitution let $\Sigma = \{a_1, a_2, ..., a_n\}$ be an alphabet of a context-free language L and L_a for $a \in \Sigma$ arbitrary context-free languages. Let $G = (N, \Sigma, P, S)$ and $G_a = (N_a, \Sigma_a, P_a, S_a)$ for $a \in \Sigma$ be grammars for which $L = L(G)$ and $L_a = L(G_a)$ for $a \in \Sigma$.

2. Let us assume that $N \cap N_a = \emptyset$ and $N_a \cap N_b = \emptyset$ for each $a, b \in \Sigma$, $a \neq b$. Let us construct a grammar $G' = (N', \Sigma', P', S)$ this way:

   (a) $N' = N \cup \bigcup_{a\in\Sigma} N_a$.

   (b) $\Sigma' = \bigcup_{a\in\Sigma} \Sigma_a$.

   (c) Let h be a morphism on $N \cup \Sigma$ such that

      i. $h(A) = A$ for $A \in N$ and

      ii. $h(a) = S_a$ for $a \in \Sigma$

   and let $P' = \{A \to h(\alpha) \mid (A \to \alpha) \in P\} \cup \bigcup_{a\in\Sigma} P_a$.

3. Consider an arbitrary sentence $a_{i_1}a_{i_2}...a_{i_m} \in L$ and sentences $x_j \in L_{a_j}$, $1 \leq j \leq m$. Then $S \overset{*}{\underset{G'}{\Rightarrow}} S_{a_{i_1}}S_{a_{i_2}}....S_{a_{i_m}} \overset{*}{\underset{G'}{\Rightarrow}} x_1S_{a_{i_2}}....S_{a_{i_m}} \overset{*}{\underset{G'}{\Rightarrow}} ... \overset{*}{\underset{G'}{\Rightarrow}} x_1x_2...x_m$ and thus $L' \subseteq L(G')$. Similarly $L(G') \subseteq L'$.

$\square$

Closure of $\mathcal{L}_2$ under miscellaneous language operations

Theorem 5.11 The context-free languages are closed under:

1. union,

2. concatenation,

3. iteration,

4. positive iteration,

5. morphism.

Proof: Let L_a and L_b be context free languages.

1. The closure under $\cup$ follows from substitution of L_a, L_b into the language $\{a, b\}$.
2. The closure under $\cdot$ follows from substitution of L_a, L_b into the language $\{ab\}$.
3. The closure under* follows from substitution of L_a into the language $\{a\}^*$.
4. The closure under$^+$ follows from substitution of L_a into the language $\{a\}^+$.
5. Let h be a given morphism and $L'_a = \{h(a)\}$ for $a \in \Sigma$. By substitution of the languages L'_a into the language L we obtain the language $h(L)$.

□

Theorem 5.12 Context-free languages are closed under intersection with the regular languages.

Proof: We can easily construct a pushdown automaton accepting the corresponding intersection – we construct the intersection on the finite control, the stack operations are preserved.

□

Definition 5.12 If $h : \Sigma^* \rightarrow \Delta^*$ is the morphism, then we can define *an inverse morphism over words* from Δ^* this way $h^{-1}(w) = \{x \in \Sigma^* \mid h(x) = w\}$ and *an inverse morphism of the language* L over Δ as $h^{-1}(L) = \{x \in \Sigma^* \mid h(x) \in L\}$.

Theorem 5.13 Context-free languages are closed under inverse morphism.

Proof: Consider an arbitrary morphism $h : \Sigma^* \rightarrow \Delta^*$. Let us establish the auxiliary alphabet $\overline{\Sigma}$, which contains $\overline{a}$ such that $\overline{a} \notin \Sigma$ for each $a \in \Sigma$. Closure of $\mathcal{L}_2$ under inverse morphism follows from closure under substitution, intersection with regular languages and morphism: $h^{-1}(L) = h_1(\sigma(L) \cap L_1^*)$, where:

1. σ is a substitution such that $\forall c \in \Delta : L_c = \overline{\Sigma}^* c \overline{\Sigma}^*$,
2. $L_1 = \{\overline{a}w \mid \overline{a} \in \overline{\Sigma} \wedge w \in \Delta^* \wedge h(a) = w\}$ is a regular language and
3. $h_1 : (\overline{\Sigma} \cup \Delta)^* \rightarrow \Sigma^*$ is a morphism such that
(1) $\forall \overline{a} \in \overline{\Sigma} : h_1(\overline{a}) = a$ and
(2) $\forall c \in \Delta : h_1(c) = \varepsilon$.

□

Non-closure of $\mathcal{L}_2$ under intersection and complement

Theorem 5.14 Context-free languages *are not* closed under intersection and complement.

Proof:

1. The non-closure under $\cap$:

   Consider the context-free languages $L_1 = \{a^m b^m c^n \mid n, m \geq 1\}$ and $L_2 = \{a^m b^n c^n \mid m, n \geq 1\}$. However the language $L_1 \cap L_2 = \{a^n b^n c^n \mid n \geq 1\}$ is not a context-free language (this was proved by pumping lemma).

2. The non-closure under the complement:

   Consider that the context-free languages are closed under complement. From De Morgan's laws and from the closure under union follows the closure under intersection $L_1 \cap L_2 = \overline{\overline{L_1 \cap L_2}} = \overline{\overline{L_1} \cup \overline{L_2}}$, which is a contradiction.

$\square$

5.5.3 Decidable and undecidable problems for context-free languages

Theorem 5.15 The following problems are *decidable*, it means these problems are algorithmically solvable:

1. The problem of *nonemptiness of the language* $L(G)$ for an arbitrary context-free grammar G.

2. The problem of *membership of a string* $w \in \Sigma^*$ *into the language* $L(G)$ for an arbitrary context-free grammar G.

3. The problem of *finality of the language* $L(G)$ for an arbitrary context-free grammar G.

Proof:

1. For the decision of the nonemptiness we can use the algorithm 4.1 which iteratively determine the set N_t of the nonterminals generating the terminal strings (mentioned in the lecture 4). Then $L(G) \neq \emptyset \Leftrightarrow S \in N_t$.

2. For the decision of the membership of the string w we can for instance determine the intersection of the nondeterministic PDA corresponding to the given grammar with a finite automaton accepting exactly the string w and then verify the nonemptiness of this intersection.

3. The problem of the finality can be decided by using pumping lemma for context-free languages:

(a) According to pumping lemma for context-free languages there is a constant $k \in \mathbb{N}$ such that for each sentence $w \in L$, $|w| \geq k$, we can write as $uvwxy$, where $vx \neq \varepsilon$ a $|vwx| \leq k$, a $\forall i \in \mathbb{N}$: $uv^iwx^iy \in L$ for each context-free language L.

(b) For the decision of the finitness it is sufficient to verify whether any string from Σ^* of the length between k and $2k-1$ does not belong to the given language: If there is such string it can be "pumped" and we obtain infinity many string belonging to the given language. If such string does not exist, $k-1$ is the upper limit for the length of the strings in L. If there would be a string of the length $2k$ or longer belonging to L, we can (according to pumping lemma) find vwx and omit vx. According to the fact that $0 < |vx| \leq k$, we would necessarily obtain, by repeated omitting, the string from L of the length between k and $2k-1$.

(c) For determination of the constant k it is sufficient to represent the language L by a context-free grammar in CNF with n nonterminals and select $k = 2^n$ (see the proof of the pumping lemma).

□

Undecidable problems for $\mathcal{L}_2$

Theorem 5.16 The following problems are *undecidable*, e.g. these problems are not algorithmically solvable:

1. The problem of *the equivalence of languages generated by context-free grammars* it means the question whether $L(G_1) = L(G_2)$ for the two context-free grammars G_1, G_2 is undecidable.

2. The problem of *inclusion of languages generated by context-free grammars*, it means that the question whether $L(G_1) \subseteq L(G_2)$ for the two context-free grammars G_1, G_2 is undecidable.

Proof: The proof uses the reduction from the undecidable Post correspond problem – see further. The undecidability of equivalence implies the undecidability of the inclusion because $A = B \Leftrightarrow A \subseteq B \wedge B \subseteq A$.

□

In the following chapters we will mention some another undecidable problems for $\mathcal{L}_2$, for instance *the universality* ($L(G) \overset{?}{=} \Sigma^*$), *the regularity* and so on.

5.5.4 Closure properties of deterministic context-free languages

Theorem 5.17 Deterministic context-free languages are closed under intersection with regular languages and under complement.

Proof: (the idea) The closure under intersection with regular languages we prove similarly as in case of languages accepted by nondeterministic PDAs. The closure under complement we prove similarly as in case of languages accepted by deterministic FA – we use the final and non-final states replacement. Moreover we have to consider these two problems: (a) A deterministic PDA does not have to read the whole input word (the PDA reaches the configuration from where it cannot continue or it performs infinitely many ε-steps) and (b) A deterministic PDA has read whole input word but afterwards it performs the sequence of the ε-steps going through both final and nonfinal states. The solution of these problems can be found in recommended literature.

□

Theorem 5.18 Deterministic context-free languages *are not* closed under intersection and union.

Proof: To prove that deterministic context-free languages are not closed under intersection we use same method as in the case of nondeterministic PDAs (notice, that languages $\{a^m b^m c^n \mid n, m \geq 1\}$ and $\{a^m b^n c^n \mid n, m \geq 1\}$ are accepted by a deterministic PDA). The nonclosure of deterministic context-free languages under union follows from De Morgan's laws.

□

Theorem 5.19 Deterministic context-free languages *are not* closed under concatenation and iteration.

Proof: (the idea)
Consider the languages $L_1 = \{a^m b^m c^n \mid m, n \geq 1\}$ and $L_2 = \{a^m b^n c^n \mid m, n \geq 1\}$. These languages are accepted by a deterministic PDA, but the language $L_1 \cup L_2$ is not accepted by any deterministic PDA. (Intuitively the deterministic PDA can not guess wether it should check the first or the second equation and thus whether it should store on the stack the symbols a or b.)

1. The nonclosure under concatenation.
   The language $L_3 = 0L_1 \cup L_2$ is obviously a deterministic context-free language. The language 0^* is also a deterministic context-free language (even a regular one), but it is not difficult to realize, that the language 0^*L_3 is not a deterministic context-free language. It is sufficient to consider that $0a^*b^*c^*$ is a deterministic context-free language (even a regular one) and $0^*L_3 \cap 0a^*b^*c^* = 0L_1 \cup 0L_2 = 0(L_1 \cup L_2)$.

2. The nonclosure under iteration.
   Consider the language $(\{0\} \cup L_3)^* \cap 0a^+b^+c^+ = 0(L_1 \cup L_2)$.

□

5.5.5 Some further interesting properties of context-free languages

DEF

Definition 5.13 A context-free grammar $G = (N, \Sigma, P, S)$ has the property of self-imbedding, if there is $A \in N$ and $u, v \in \Sigma^+$ such that $A \Rightarrow^+ uAv$ and A is not the useless nonterminal. A context-free language has the property of self-imbedding, if each grammar which generates this language has the property of self-imbedding.

Regularity

Theorem 5.20 A context-free language has the property of self-imbedding if and only if it is not regular.

Proof: We can use Greibach normal form (definice 4.15).

□

The mentioned theorem can be useful to prove that a given grammar does not generate a regular language. However we repeat, that the problem whether a given context-free grammar generates a regular language is algorithmically undecidable.

Chomsky and Schützenberger theorem: This theorem covers the close relation between context-free languages and parentheses problem.

DEF

Definition 5.14 For $n \geq 0$ we denote ZAV_n languages consisting from all well-formed strings of parentheses of n types. This languages – denoted also Dyck languages – are generated by grammars with the rules of the form:

$$S \rightarrow [^1\ S\ ]^1 \mid [^2\ S\ ]^2 \mid \cdots \mid [^n\ S\ ]^n \mid SS \mid \varepsilon$$

Theorem 5.21 (The Chomsky-Schützenberger) Each context-free language is a morphism of intersection of a language of the parentheses and a regular set. Formally, for each $L \in \mathcal{L}_2$ there is $n \geq 0$, a regular set R and a morphism h such that $L = h(ZAV_n \cap R)$

Proof: See recommended literature. □

Parikh theorem

This theorem again covers structure of context-free languages. It deals with the question what we obtain if we ignore the order of particular symbols in sentences and explore only the number of their occurrence (it means that we consider the arbitrary permutation of symbols in the given string).

DEF

Definition 5.15 Consider an alphabet $\Sigma = \{a_1, ..., a_k\}$. *The Parikh function* is a function $\psi : \Sigma^* \rightarrow \mathbb{N}^k$ defined for $w \in \Sigma^*$ as $\psi(w) = (\#a_1(w), ..., \#a_k(w))$, where $\#a_i(w)$ states for the number of occurrences of the symbol a_i in w.

DEF

Definition 5.16 A subset of a set of vectors $\mathbb{N}^k$ is called *a linear set*, if it is given by a base $u_0 \in \mathbb{N}^k$ and periods $u_1, ..., u_m \in \mathbb{N}^k$ as

$$\{u_0 + a_1u_1 + ... + a_mu_m \mid a_1, ..., a_m \in \mathbb{N}\}.$$

A subset $\mathbb{N}^k$ is called *a semilinear set*, if it is a union of finite number of linear sets.

Theorem 5.22 (The Parikh theorem) For an arbitrary context-free language L, $\psi(L)$ is a semilinear set.

Proof: See recommended literature. □

For each semilinear set S *we can find a regular set* $R \subseteq \Sigma^*$ such that $\psi(R) = S$. Hence the Parikh theorem is sometimes formulated this way: The commutative image of each context-free languages corresponds to a regular language. Moreover semilinear sets can be represented by finite automata directly as sets of numerical vectors in binary coding.

Nondeterministic pushdown automata are an equivalent specification means for context-free languages. There are different equivalent types of these automata, which are used as models for different types of parsers. The class of languages accepted by deterministic pushdown automata(so-called deterministic context-free languages) is a proper subclass of the class of context-free languages. Context-free languages are not closed under intersection and complement. The problem of equivalence of two context-free grammars is undecidable. We can use pumping lemma to prove that a given language is not a context-free language. This lemma also characterizes basic structural patterns of the sentences of an infinite context-free language.

5.6 Exercises

Exercise 5.6.1 Construct a pushdown automaton accepting the language

$$L = \{a^n b^m \mid n \leq m \leq 2n\}$$

Exercise 5.6.2 Construct a pushdown automata modeling the top-down parsing and bottom-up parsing for the grammars

(a)

$$S \rightarrow aSb \mid \epsilon$$

(b)

$$\begin{aligned} S &\rightarrow AS \mid b \\ A &\rightarrow SA \mid a \end{aligned}$$

(c)

$$\begin{aligned} S &\rightarrow SS \mid A \\ A &\rightarrow 0A1 \mid S \mid 01 \end{aligned}$$

Exercise 5.6.3 Find a grammar generating the language $L(P)$, where

$$P = (\{q_0, q_1, q_2\}, \{a, b\}, \{Z_0, A\}, \delta, q_0, Z_0, \{q_2\});$$

and the mapping δ is:

$$\begin{aligned}
\delta(q_0, a, Z_0) &= (q_1, AZ_0) \\
\delta(q_0, a, A) &= (q_1, AA) \\
\delta(q_1, a, A) &= (q_0, AA) \\
\delta(q_1, \epsilon, A) &= (q_2, A) \\
\delta(q_2, b, A) &= (q_2, \epsilon)
\end{aligned}$$

Chapter 6

Turing machines

15:00

This chapter aims to define the Turing machine formally, set the rules and conventions for modular construction of Turing machine, bring comprehension of the mechanism of definition of language accepted and decided by Turing machine, and to prove the equivalence of the class of type 0 languages and the class of languages accepted by Turing machines. The aim of this chapter is also to analyze appropriate languages and an important modification of Turing machine defining context-sensitive languages.

6.1 Basic concepts of Turing machines

6.1.1 Church thesis

Church (Church-Turing) thesis*: Turing machines (and systems equivalent to them) define, by their computational power, what we consider effectively computable.*

Church thesis *is not a theorem*, we cannot prove in a formal way that something corresponds to our intuitive ideas, nevertheless it is supported by a number of arguments:

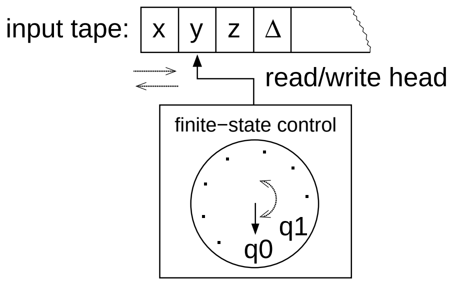
- Turing machines are *very robust* – we will see that their various modifications do not change their computational power (determinism x nondeterminism, number of input tapes, ...).
- A number of totally *different models of computation* (λ-calculus, partially recursive function, Minsk machines, ...), whose power is equal to Turing machines, has been proposed.
- No *computation process*, which we can call effectively computable and we cannot execute it on Turing machine, is known.[1]

6.1.2 Turing machine

A Turing machine consists of a finite-state control unit, unidirectionally unbounded tape and reading/writing head. In one computation step, a Turing machine first reads the symbol under the head. Consequently, depending upon the symbol currently read and the state of the control unit,

[1]There are formalized computational processes which are executable for example on a TM with an oracle (prompt) which atomically decides a problem which is undecidable by a standard TM (e.g. halting problem), but we don't consider them as effectively computable processes.

it can rewrite the read symbol, change the state and move the head right or left by one cell. A computation of the Turing machine is controlled by a finite set of rules – a transition relation that is part of the control unit. The computation is a series of steps performed successively which starts in the initial configuration of the machine (a configuration is a triple consisting of the state of the control unit, the tape content and the head position).



DEF

Definition 6.1.1 *A* Turing machine *(TM) is a 6-tuple of the form* $M = (Q, \Sigma, \Gamma, \delta, q_0, q_F)$*, where:*

- Q *is a finite set of control states,*
- Σ *is a finite set of symbols called input alphabet,* $\Delta \notin \Sigma$*,*
- Γ *is a finite set of symbols,* $\Sigma \subset \Gamma$*,* $\Delta \in \Gamma$*, called the tape alphabet,*
- *a partial function* $\delta : (Q \setminus \{q_F\}) \times \Gamma \to Q \times (\Gamma \cup \{L, R\})$*, where* $L, R \notin \Gamma$*, called the transition function,*
- q_0 *is the initial state,* $q_0 \in Q$ *and*
- q_F *is the final state,* $q_F \in Q$*.*

6.1.3 Configuration of Turing machine

The symbol Δ denotes the so-called *blank* (empty symbol) which occurs in tape areas, that haven't been used yet (but can be also written on tape later).

A *configuration of the tape* is a pair consisting of an infinite string representing the tape contents and a head position on this string – more precisely, it is an element of the set $\{\gamma\Delta^\omega \mid \gamma \in \Gamma^*\} \times \mathbb{N}$.

We write the configuration of the tape as $\Delta xyz\underline{z}\Delta x\Delta\Delta...$ (the underline marks the head position).

A *configuration of the machine* consists of the control state and the tape configuration – formally, it is an element of the set $Q \times \{\gamma\Delta^\omega \mid \gamma \in \Gamma^*\} \times \mathbb{N}$.

6.1.4 Transition function of TM

DEF

For an arbitrary string $\gamma \in \Gamma^\omega$ and a number $n \in \mathbb{N}$ we denote by γ_n the symbol on the position n of the string γ and by $s^n_b(\gamma)$ the string that is formed from γ by substituting γ_n for b.

A computation step of a TM M is defined as a binary relation $\underset{M}{\vdash}$ such that $\forall q_1, q_2 \in Q\ \forall \gamma \in \Gamma^\omega\ \forall n \in \mathbb{N}\ \forall b \in \Gamma$:

- $(q_1, \gamma, n) \underset{M}{\vdash} (q_2, \gamma, n+1)$ for $\delta(q_1, \gamma_n) = (q_2, R)$ – an operation of a *move to the right* when γ_n is under head,
- $(q_1, \gamma, n) \underset{M}{\vdash} (q_2, \gamma, n-1)$ for $\delta(q_1, \gamma_n) = (q_2, L)$ a $n > 0$ – an operation of a *move to the left* when γ_n is under head and
- $(q_1, \gamma, n) \underset{M}{\vdash} (q_2, s_b^n(\gamma), n)$ for $\delta(q_1, \gamma_n) = (q_2, b)$ – an operation of *writing* b when γ_n is under head.

6.1.5 Computation of TM

DEF

A computation of a TM M starting from configuration K_0 is a sequence of configurations K_0, K_1, K_2, ... in which $K_i \underset{M}{\vdash} K_{i+1}$ for all $i \geq 0$, and which is either

- *infinite*, or
- *finite* with final configuration (q, γ, n). In this case, we distinguish between the following *types of termination*:
  1. *normal* – move to the final state, i. e. $q = q_F$, and
  2. *abnormal*:
     (a) the δ function is not defined for (q, γ_n), or
     (b) the head is on the leftmost position of the tape and it moves to the left,
     i. e. $\delta(q, \gamma_n) = (q', L)$ for some $q' \in Q$.

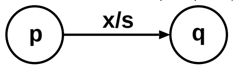
6.1.6 Note – alternative definitions of TM

Alternative definitions of TM are sometimes used and their *mutual convertibility* can be easily shown:

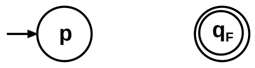
- instead of having a single final state q_F, a set of finite states is allowed,
- instead of having a single final state q_F, two final states q_{accept} and q_{reject} are used,
- a special end-marker symbol is written in the first cell of the tape which cannot be rewritten and for which the δ function does not allow a move to the left,
- when using both two previous modifications, the δ function is usually defined as total,
- the symbol rewriting and the head move are joined into a single operation
- etc.

6.1.7 Graphical representation of TM

Graphical representation of a *transition*(x – the symbol read, s – write/L/R):

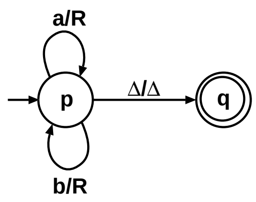


Graphical representation of the *initial* and the *finite state*:



$\boxed{x+y}$

Example 6.1.1 *TM that moves its head to the right up to the first occurence of the symbol* Δ*,including the current position (e.g.* $\Delta\underline{a}ab\Delta ... \longrightarrow \Delta aab\underline{\Delta} ...$*):*

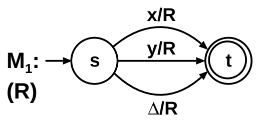


6.1.8 Modular construction of TM

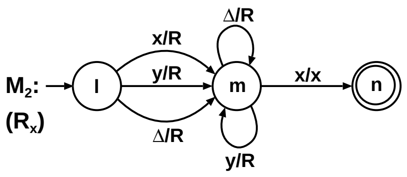
Analogously to the program development as a composition of simpler subprograms, procedures and functions, the TM can be constructed modularly as a *combination* of simpler TMs.

Example 6.1.2 *Consider the three following components:*

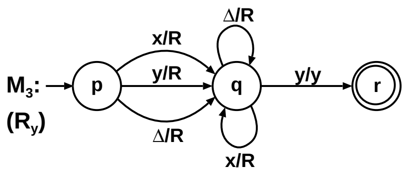
- M_1 *moves head to the right by one symbol :*



- M_2 *finds the first occurence of the symbol* x *on the right (starting at the current head position):*

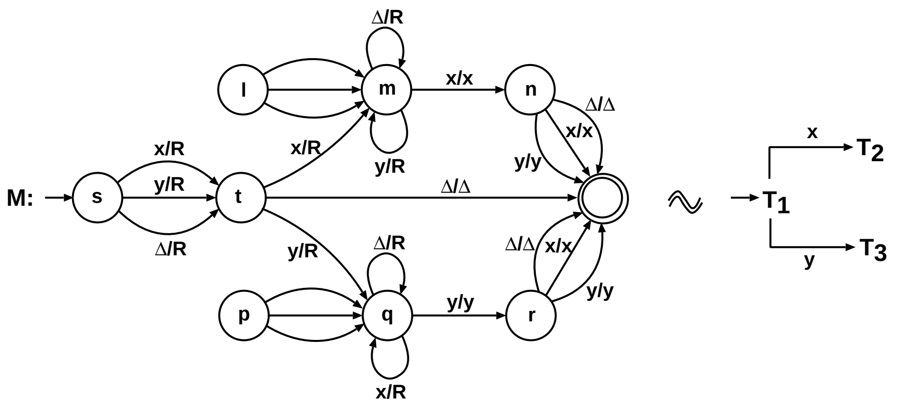


- M_3 *finds the first occurence of the symbol* y *on the right:*



$\boxed{x+y}$

The following combination of M_1, M_2 and M_3 creates a TM M, that finds the second occurence of a (non-blank) symbol from the initial position. M is depicted in the form of the so-called *composite diagram* in the right part of the figure:



6.1.9 Composite diagram of TM

A convention for a *simplified notation of a composite diagram*:

1. *A sequence of machines* $\longrightarrow A \longrightarrow B \longrightarrow C$ is shortened to $\longrightarrow ABC$.

2. The *parametric convention*: $\xrightarrow{x,y,z}\}\xrightarrow{\quad\omega\quad}$, where ω is the value of the symbol which was on the tape.

Example 6.1.3
$$\begin{tikzcd} {} \arrow[r] & M_1 \arrow[r, "{x,y}"] & \} \arrow[r, "\omega"] & M_2 \arrow[ll, bend right, "\omega"'] \end{tikzcd}$$

M_1 terminates with x which will get to input of M_2, and the control is passed back to M_1 only when M_2 terminates with x. Analogously for y.

This situation can be also illustrated as follows:
$$\begin{tikzcd} {} \arrow[r] & M_1 \arrow[r, "x"] \arrow[dr, "y"'] & M_2 \arrow[l, bend right, "x"'] \\ & & M_2 \arrow[ul, bend left, "y"] \end{tikzcd}$$

3. *"Branching"*
$$\begin{tikzcd} & & M_2 \\ {} \arrow[r] & M_1 \arrow[ur, "x"] \arrow[dr, "y"'] & \\ & & M_3 \end{tikzcd}$$
or
$$\begin{tikzcd} & & M_2 \\ {} \arrow[r] & M_1 \arrow[ur, "x"] \arrow[dr, "\neg x"'] & \\ & & M_3 \end{tikzcd}$$
.

Beware of the *difference between the two following constructions*:

- $\longrightarrow M_1 \longrightarrow M_2$ – it moves always, when appropriate edges in M_2 are defined (i.e. edges from q_F in M_1 for all edges from q_0 in M_2 are added – it means matching of q_F from M_1 and q_0 from M_2).

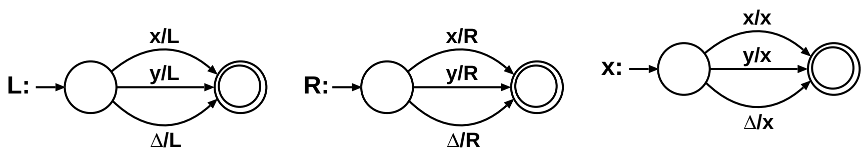
- $\longrightarrow M_1 \xrightarrow{x} M_2$ – only the edge x is added (provided it is in M_2).

6.1.10 Basic building blocs of TM

Consider $\Gamma = \{x, y\}$. A *basic building blocs of TM* are usually the following machines (it can be easily modified for another Γ):

DEF

1. *Machines L, R, x*:

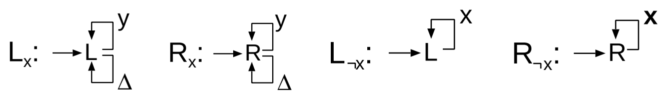


to the left *to the right* *write x on the current position*

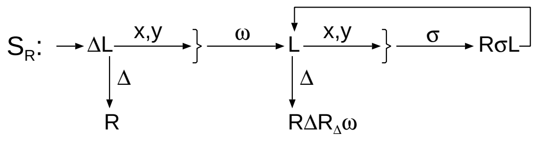
x+y

Example 6.1.4 *Machine* $\longrightarrow R \longrightarrow y \longrightarrow L$ *i.e.* $\longrightarrow RyL$ *transforms tape* $\Delta xyxy\underline{x}\Delta\Delta...$ *to* $\Delta xyxy\underline{x}y\Delta...$

2. *Machines L_x, R_x, $L_{\neg x}$ and $R_{\neg x}$*:



3. *Machines S_R and S_L* for the *shift of tape content*: Machine S_R moves the string of non-blank symbols situated on the left from the current position of the head to the right by one symbol. Machine S_L behaves similarly.



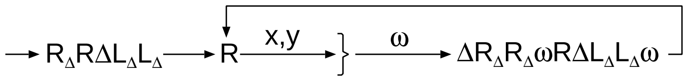
Example 6.1.5 *Computation of machines S_R and S_L is illustrated in following examples:*

- $\Delta xyy\underline{x}x\Delta\Delta... \xrightarrow{S_R} \Delta\Delta xy\underline{y}x\Delta\Delta...$
- $\Delta yxy\underline{\Delta}\Delta xxy\Delta... \xrightarrow{S_R} \Delta\Delta yx\underline{y}\Delta xxy\Delta...$
- $xy\Delta\underline{y}x\Delta\Delta... \xrightarrow{S_R} xy\Delta\underline{\Delta}x\Delta\Delta...$
- $\Delta\underline{y}yxx\Delta\Delta... \xrightarrow{S_L} \Delta\underline{y}xx\Delta\Delta\Delta...$

6.1.11 Examples of TM

x+y

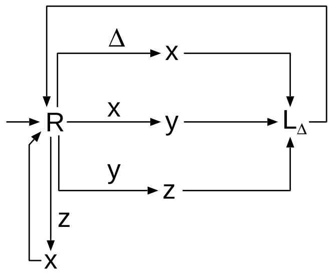
Example 6.1.6 *A* copy machine *– transforms* $\underline{\Delta}w\Delta$ *to* $\Delta w\underline{\Delta}w\Delta$*:*



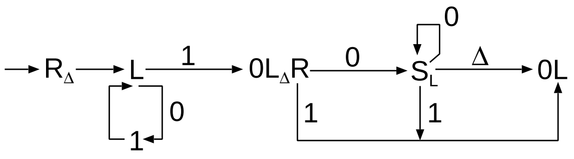
$R_\Delta R\Delta L_\Delta L_\Delta$
R
x,y
ω
$\Delta R_\Delta R_\Delta \omega R\Delta L_\Delta L_\Delta \omega$

Example 6.1.7 String generation*:*

- *The following machine successively generates all strings over* $\{x, y, z\}$ *in this order:* $\varepsilon, x, y, z, xx, yx, zx, xy, yy, zy, xz, yz, zz, xxx, yxx, \ldots$.
- *We assume that the machine starts with the tape configuration* $\underline{\Delta}w\Delta$*, where* w *is a string of the mentioned sequence, from which the string generation will begin.*



Example 6.1.8 *A machine* decrementing a positive number *written in the binary numeral system:*



6.2 Turing machines as languages acceptors

6.2.1 Language accepted by TM

A Turing machine normally halts (by a transition to the state q_F) only for some words from the set Σ^* written on the tape in an initial configuration. Therefore, each Turing machine uniquely determines the set of words from Σ^* – *the language accepted by Turing machine.*

We can consider computational problems as languages. The class of languages, for which a Turing machine that accepts this languages exists, represents the class of all computable problems.

DEF

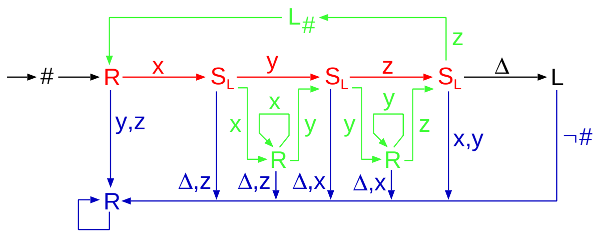
Definition 6.2.1

1. *A* string $w \in \Sigma^*$ is accepted by a TM $M = (Q, \Sigma, \Gamma, \delta, q_0, q_F)$*, if M starts the computation from an initial tape configuration* $\underline{\Delta}w\Delta...$ *and from the initial state* q_0 *and if it halts by a move to the final state* q_F*, i.e.* $(q_0, \Delta w \Delta^\omega, 0) \overset{*}{\underset{M}{\vdash}} (q_F, \gamma, n)$ *for some* $\gamma \in \Gamma^*$ *and* $n \in \mathbb{N}$*.*

2. *A set* $L(M) = \{w \mid w$ *is accepted by TM* $M\} \subseteq \Sigma^*$ *is called* language accepted by a TM M*.*

x+y

Example 6.2.1 *The following Turing machine M accepts the language* $L(M) = \{x^n y^n z^n \mid n \geq 0\}$*:*



The machine works in this manner: $x^n y^n z^n \to x^{n-1} y^n z^n \to x^{n-1} y^{n-1} z^n \to x^{n-1} y^{n-1} z^{n-1} \to \ldots \to \varepsilon$*.*

The machine that accepts the same language, with the transition function defined by the following table:

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_F)$$
$$Q = \{q_0, q_1, q_2, q_3, q_F\}$$
$$\Sigma = \{x, y, z\}$$
$$\Gamma = \{x, y, z, \times, \Delta\}$$

δ	Δ	x	y	z	$\times$
q_0	q_1, R				
q_1	q_F	$q_2, \times$			q_1, R
q_2			$q_3, \times$		q_2, R
q_3				$q_4, \times$	q_3, R
q_4	q_1, R	q_1, L	q_1, L	q_1, L	q_1, L

The machine is gradually crossing out (with symbol $\times$*) one* x*, one* y *and one* z *symbols. In state* q_1 *it seeks for* x*, then in state* q_2 *it seeks for* y*, and then in state* q_3 *it seeks for* z*. In state* q_4 *it returns to the beginning of the word, where it starts to cross out next triplet. It accepts only if the input word is empty, or if it has achieved to cross out all triplets.*

6.3 Modifications of TM

We will investigate a couple of ways how to extend Turing machine capabilities. Because the original TM is very robust, not even the seemingly more general modifications lead to mechanisms with higher computational power – the class of languages accepted by following modified Turing machines remains the same.

6.3.1 Acceptance by a specific tape configurations

There are situations when it might be convenient that Turing machine notifies us about the correct termination of a computation in other way, for instance by writing the computation result information on the tape.

We can define the *string acceptance by a TM* in such way that TM starts with the tape configuration $\underline{\Delta}w\Delta...$ and halts with the tape configuration $\underline{\Delta}Y\Delta...$, $Y \in \Gamma \setminus \Sigma$, ($Y$ *denotes Yes*).

Theorem 6.3.1 Given a TM M which accepts $L(M)$ by a transition to q_F, we can always construct a machine M'', which accepts $L(M)$ by halting with the tape configuration $\underline{\Delta}Y\Delta....$

Poof. (Idea)

$$\begin{tikzcd}
M'': & {} \arrow[r] & R_\Delta S_R R{*} L_\Delta L\#R \arrow[r] & M' \arrow[r] & R_* \arrow[r] & \Delta L \arrow[loop right, "\neg\#"] \arrow[d, "\#"] \\
 & & & & & \Delta R Y L
\end{tikzcd}$$

- M'' starts with tape $\#\underline{\Delta}w * \Delta....$
- M' is constructed by adding transitions to M (M having a set of states Q) that solve situations when the machine operates at the positions of the $*$ and $\#$ symbols:
    1. $\forall q \in Q : \delta(q, \#) = (q, L)$ – "securing the fall over the left end of the tape" (abnormal termination) which might have happened originally.
    2. When reading symbol $*$ it is (possibly) necessary to extend the working space – we activate the submachine $\Delta R * L$.

□

We can also easily *transform backwards* from acceptance by tape configurations of the form $\underline{\Delta}Y\Delta...$ to acceptance by q_F.

Remark 6.3.1

- *Beware of the fact that we can view TMs also as* computing mechanisms implementing functions $\Sigma^* \to \Gamma^*$ *by transforming an initial non-blank prefix of its tape to another non-blank prefix when moving to halting state.*

- *Considering that TM does not have to accept all its inputs, functions implemented by TM are, generally,* partial.
- *We will look more closely on computations of functions by TM later.*

6.3.2 Multi-tape Turing machines

A straightforward way how to extend Turing machines is to integrate not one, but a number of tapes. Multi-tape TM disposes of k unidirectionally infinite tapes, where each of them has its own reading/writing head and are all connected with one finite-state unit. During one step, a machine first reads the current symbols on all tapes, then it moves the head or rewrites a symbol, and finally – according to the symbols read – changes the state of the control unit.

DEF

More formally – consider a TM which has k *tapes* with tape alphabets $\Gamma_1, \Gamma_2, ..., \Gamma_k$ and k *corresponding heads* with a transition function in the form

$$\delta : (Q \setminus \{q_F\}) \times \Gamma_1 \times \Gamma_2 \times ... \times \Gamma_k \longrightarrow Q \times \bigcup_{i \in \{1,...,k\}} \{i\} \times (\Gamma_i \cup \{L, R\}),$$

where i denotes the tape to which it is written (on which the head is moved).

x+y

Example 6.3.1 *We construct a 2-tape machine that will accept the already known language $x^n y^n z^n$. The use of the second tape will effect the efficiency of the computation. We assume that in an initial configuration the tapes have the following form:*
– first tape: $\underline{\Delta} w \Delta\Delta \ldots$
– second tape: $\underline{\Delta}\Delta \ldots.$
The machine is defined in the following way:

$$\begin{aligned}&M = \{Q, \Sigma, \Gamma_1, \Gamma_2, \delta, q_0, q_F\} \text{ where:}\\&Q = \{q_0, q_1, q_2, q_3, q_F\}\\&\Sigma = \{x, y, z\}\\&\Gamma_1 = \{x, y, z, \Delta\}\\&\Gamma_2 = \{X, Y \Delta\}\end{aligned}$$

δ	Δ, Δ	x, Δ	x, X	y, Δ	y, X	y, Y	z, Δ	z, Y
q_0	q_1, R, R							
q_1	q_F	q_1, x, X	q_1, R, R	q_2, y, L				
q_2					q_2, y, Y	q_2, R, L	q_3, z, R	
q_3	q_F							q_3, R, R

The machine reads a word on the first (input) tape only once from left to the right. At the same time, it checks – using the second tape – the form of the word as follows: First by writing symbols X to the second tape, it records symbols x which are read on the input. Then, for each y read on input it rewrites one symbol X on the second tape with symbol Y (and *finishes this part at the beginning of the second tape). Finally, for each z on the input it moves the head of the second tape to the right. Word is accepted, if the head of the second tape points precisely to the field behind all Y s.*

The machine accepting the same language in a similar way, written by a diagram:

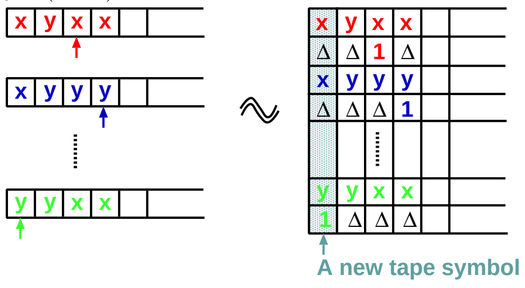
$$\begin{tikzcd}
{} \arrow[r, "\Delta^1\Delta^2"] & R^1R^2 \arrow[rr, "\Delta^1\Delta^2"] \arrow[dd, "\Delta^1\Delta^2"'] \arrow[ddr, "*"] & & X^2R^1R^2 \arrow[loop right, "x^1\Delta^2"] \arrow[d, "y^1\Delta^2"] \arrow[ddl, "*"'] \\
& & & L^2 \arrow[dl, "*"'] \arrow[dd, "X^2"] \\
& \Delta^1\Delta^2 \arrow[r, "*"'] & \mathsf{LOOP} & \\
& & & Y^2 \arrow[ul, "*"'] \arrow[d] \arrow[d, bend left, "y^1X^2"] \\
& R^1R^2 \arrow[uu, "\Delta^1\Delta^2"] \arrow[uur, "*"] \arrow[loop left, "z^1Y^2"] & R^2 \arrow[l, "y^2"'] \arrow[uu, "*"] & R^1L^2 \arrow[l, "z^1\Delta^2"'] \arrow[uul, "*"]
\end{tikzcd}$$

Action of the first and the second head are distinguished by upper index. LOOP *is a looping TM. Star label of an arrow is an abbreviation of all pairs of symbols distinct from labels of other arrows coming out from the given node.*

Example 6.3.2 *Design a multi-tape TM for conversion of numbers from binary notation to unary notation.*

Theorem 6.3.2 For every k-tape TM M, there exists a one-tape TM M', such that $L(M) = L(M')$.

Poof. (idea)



The proof is made by giving an algorithm that converts a k-tape TM to an equivalent one-tape TM:

- We assume that in a k-tape machine, the *accepted string* is written on the first tape initially, all other tapes are empty and all heads are on the leftmost position.

- We simulate the original k tapes by *extending the tape alphabet with $2k$-tuples*, in which the i-th element for odd i represents content of the $(\frac{i+1}{2})$-th tape and at the $(i+1)$-th position is Δ or 1 depending on whether the corresponding head points to that position or not respectively.

- *The number of combinations of symbols read in the original automaton is finite* and thus we can afford the extension mentioned above.

- During the simulation of a k-tape TM, we first transform the original content of the first tape to an equivalent content encoded in $2k$-tuples. Then, we simulate each step by several steps.

- During the simulation, we use *states in the form $(k+1)$-tuples*, where the first element is the state of the original TM and other elements are currently read symbols.

- *When determining next step of computation*, we consider especially this state, the actually read symbol is not important. When reading, we move to a special position on the left from useful tape content.

- *When determining next step of computation "we move" to the right* to the position, on which the simulated head of a tape that is about to be modified is. Then we make the modification and return to the left.

- *As a new current state* we consider $(k+1)$-tuple given by a new state of the simulated TM and k-tuple taken from the former state of the new TM modified at the place where the tape was modified.

- Moreover, it is necessary to correctly *simulate situations when any of the heads falls over the left end of the respective tape* and *conversion of still unused places of tape* with Δ to corresponding $2k$-tuple of blank symbols.

- If we formalized the described algorithm it would not be difficult to show that the resulting TM really simulates the original TM.

□

Conclusion:

An extension of the memory potential of TMs does not extend their abilities to accept languages!

Example 6.3.3 *Use the potential of multi-tape machine to design a TM accepting the language $x^n y^n z^n$.*

6.3.3 Nondeterministic Turing machines

The defined deterministic Turing machine determines the next step of computation on the basis of the read symbol and state. Nondeterministic TM, being in a certain configuration, chooses from finite number of options. Therefore, there exist more possible computations for one input configuration. The input word is accepted by a nondeterministic TM if and only if at least one of the possible computations is accepting. [2]

Definition 6.3.1 *A* nondeterministic TM *is a 6-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, q_F)$, where the meaning of its elements is the same as in the deterministic TM except for δ, which is of the form:*

$$\delta : (Q \setminus \{q_F\}) \times \Gamma \longrightarrow 2^{Q\times(\Gamma\cup\{L,R\})}$$

Definition 6.3.2 Language $L(M)$ accepted by a NTM $M = (Q, \Sigma, \Gamma, \delta, q_0, q_F)$ *is a set of strings $w \in \Sigma^*$ such that when M starts from q_0 with the initial tape content being $\underline{\Delta}w\Delta$...* it **can** halt *by moving to q_F.*

Example 6.3.4 *We construct 2-tape nondeterministic Turing machine that accepts language $L = \{ww^R|$ where $w \in \{x, y\}^*\}$. We use nondeterminism to "guess" the half of input word. It means that the machine first reads the word on the first tape and the symbols it read writes on the second tape. At certain point, the machine makes a nondeterministic transition into a state, when it starts to compare the content of the second tape with the part of the first tape that has not been read yet (it reads from right to left on the second tape). We assume that in the initial configuration the tapes are of the form:*
– first tape: $\underline{\Delta}w\Delta\Delta\dots$
– second tape: $\underline{\Delta}\Delta\dots$
As the machine is nondeterministic, its transition function δ returns a set of states. It means that a cell of table for the transition function can contain more than one item. The machine will look like this:

$M = \{Q, \Sigma, \Gamma_1, \Gamma_2, \delta, q_0, q_F\}$ *where:*
$Q = \{q_0, q_1, q_2, q_F\}$
$\Sigma = \{x, y\}$
$\Gamma_1 = \{x, y, \Delta\}$
$\Gamma_2 = \{x, y, \Delta\}$

δ	Δ,Δ	x,Δ	x,x	y,Δ	y,y	x,y	y,x	Δ,x	Δ,y
q_0	q_1,R,R								
q_1	q_0,R,R	q_1,x,x	q_1,R,R q_2,R,x	q_1,y,y	q_1,R,R q_2,R,y				
q_2	q_F	q_0,x,Δ	q_2,R,L	q_0,y,Δ	q_2,R,L	q_0,x,y	q_0,y,x	q_0,Δ,x	q_0,Δ,y

[2] as if the machine "was guessing" which one of the possible computation steps (if any) leads to the final state

Example 6.3.5 *Design a multi-tape nondeterministic TM accepting the language* $\{a^p|$ *where p is not a prime number*$\}$.

Theorem 6.3.3 For every NTM M there exists a DTM M' such that $L(M) = L(M')$.

Poof. (idea)

- *We will simulate the NTM M with a 3-tape DTM.* The meaning of tapes of this machine is following:
  - *Tape 1* contains the accepted input string.
  - *Tape 2* is the working tape. It contains the copy of the tape 1. There are suitable special characters at the boundaries of the copied string. Its content is erased and restored from the first tape after an unsuccessful attempt to accept the input.
  - *Tape 3* contains an encoded transition sequence which has been chosen. In case of non-acceptance, the content will be replaced with other sequence.
- The chosen *sequence of transitions is encoded* by a sequence of numbers assigned to transitions of the simulated machine.
- *Sequences of transitions on the tape 3 can be generated* by a modified TM for generation of string sequence $\varepsilon, x, y, z, xx, ...$
- The *simulation itself proceeds as follows*:
  1. Copy the content of the tape 1 to the tape 2.
  2. Generate the next sequence of transitions on the tape 3.
  3. Simulate an execution of the sequence from tape 3 on the content of the tape 2.
  4. If the currently investigated sequence leads to q_F of the simulated machine, the machine halts – the input string is accepted. Otherwise, erase the content of the tape 2 and return to the point 1.
- It is not difficult to see that the language accepted by machine designed in this manner is the same as the language accepted by the original NTM.

□

Conclusion:

By incorporating nondeterminism into TMs, their ability to accept languages does not increase!

6.4 Recursive languages and recursively enumerable languages

We establish a classification of languages (problems) according to their acceptability by Turing machines. We come to three basic classes of languages that represent decidable problems, partially decidable problems and algorithmically undecidable problems.

6.4.1 Recursive enumeration and recursiveness

Turing machine is called *total* (*total*), if and only if it halts for all inputs.

DEF

Definition 6.4.1 *A language $L \subseteq \Sigma^*$ is called*

- recursively enumerable*, if $L = L(M)$ for some TM M,*
- recursive*, if $L = L(M)$ for some* total *TM M.*

If M is a total Turing machine, then we say that M *decides the language* $L(M)$.

For every *recursive language* there exists a TM, which decides it, i.e. *halts for all input words* – this TM can be modified in a way that for all strings from the given language it halts with the tape content $\Delta Y \Delta\Delta$..., and otherwise halts with the tape content $\Delta N \Delta\Delta$....

TM which accepts a *recursively enumerable language* L halts for all $w \in L$, but for $w \notin L$ it can halt or it also *can loop infinitely.*

6.4.2 Decision problems

A *decision problem* P can be understood as a function f_P with the range $\{true, false\}$[3].

DEF

A *decision problem is usually specified by*:

- the range A_P representing the set of possible instances of the problem (inputs) and
- the subset $B_P \subseteq A_P$, $B_P = \{p \mid f_P(p) = true\}$ of instances for which f_P evaluates to $true$.

In the theory of formal languages we use the string over the suitable alphabet Σ to encode separate instances of problems. Then, the decision problem P can be naturally specified by the language $L_p = \{w \in \Sigma^* \mid w = code(p), p \in B_P\}$, where $code : A_P \to \Sigma^*$ is an injective function which for a given instance of problem returns the corresponding string (independently on f_P).

[3]a question with the possible answer yes/no

Example 6.4.1 *Examples of decision problems:*

- P_1 *– an oriented graph is strongly connected.*
- P_2 *– two context-free grammars are equivalent.*
- P_3 *–* n *is a prime number.*

Note: Further, we will talk about problems instead of decisions problems.

6.4.3 Decidability of TM problems

Definition 6.4.2 *Let* P *be a problem specified by a language* L_P *over* Σ*. We call the problem* P *as:*

- decidable*, if* L_P *is recursive language, i.e. there exists a TM which decides* L_P *(accepts all strings* $w \in L_P$ *and rejects all strings* $w \in \Sigma^* \setminus L_P$*),*
- undecidable*, when its not decidable, and*
- partially decidable*, if* L_P *is a recursively enumerable language, i.e. there exists a TM which accepts all strings* $w \in L_P$ *and every string* $w \in \Sigma^* \setminus L_P$ *is either rejected or a computation for that* w *is infinite.*

Note: From the definition 8.2 it follows that all decidable problems are also partially decidable. But there are undecidable problems which are not even partially decidable.

6.5 TM and languages of type 0

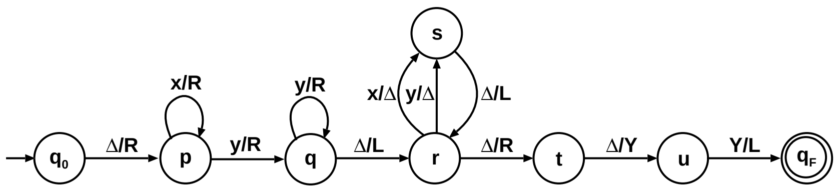
We will deduce an important correspondence between grammars of type 0 and Turing machines. Languages generated by grammars of type 0 are just languages accepted by Turing machines (recursive enumerate). We will show that for each TM, there exists a grammar of type 0 generating the language accepted by this TM and vice versa, that for each grammar of type 0 there exist a TM accepting language generated by that grammar.

6.5.1 Languages accepted by TM are of type 0

A *configuration of TM is given* (1) by state of the control unit, (2) tape content and (3) position of the head.

We denote by $[\Delta x q y z \Delta ...]$ the configuration of the TM in a state q and with the tape configuration $\Delta x \underline{y} z \Delta ...$.

Example 6.5.1 *A TM with initial configuration of the tape* $\underline{\Delta} w \Delta$ *accepting the language* $\{x^m y^n \mid m \geq 0, n > 0\}$ *by the tape configuration* $\underline{\Delta} Y \Delta$*:*



The sequence of the configurations when accepting xxy:

1. $[\textcolor{red}{q_0}\Delta xxy\Delta...]$
2. $[\Delta \textcolor{red}{p}xxy\Delta...]$
3. $[\Delta x\textcolor{red}{p}xy\Delta...]$
4. $[\Delta xx\textcolor{red}{p}y\Delta...]$
5. $[\Delta xxy\textcolor{red}{q}\Delta...]$
6. $[\Delta xx\textcolor{red}{r}y\Delta...]$
7. $[\Delta xx\textcolor{red}{s}\Delta\Delta...]$
8. $[\Delta x\textcolor{red}{r}x\Delta\Delta...]$
9. $[\Delta x\textcolor{red}{s}\Delta\Delta\Delta...]$
10. $[\Delta \textcolor{red}{r}x\Delta\Delta\Delta...]$
11. $[\Delta \textcolor{red}{s}\Delta\Delta\Delta\Delta...]$
12. $[\textcolor{red}{r}\Delta\Delta\Delta\Delta\Delta...]$
13. $[\Delta \textcolor{red}{t}\Delta\Delta\Delta\Delta...]$
14. $[\Delta \textcolor{red}{u}Y\Delta\Delta\Delta...]$
15. $[\textcolor{red}{q_F}\Delta Y\Delta\Delta\Delta...]$

Theorem 6.5.1 Every language accepted by a TM (i.e. every recursively enumerable language) is a type-0 language.

Poof. Let $L = L(M)$ for some TM $M = (Q, \Sigma, \Gamma, \delta, q_0, q_F)$. *We construct a grammar* $G = (N, \Sigma, P, S)$ *of type 0, such that* $L(G) = L(M)$. *The grammar* G *allows to create the derivations corresponding to the inversion of sequences of configurations of TM* M *when accepting any string* $w \in L(M)$*:*

1. $N = \{S\} \cup Q \cup (\Gamma \setminus \Sigma) \cup \{[,]\}$ (we assume that all sets are pairwise disjoint).

2. P is the smallest subset containing the following rules:

   (a) $S \to [q_f \Delta Y \Delta]$,

   (b) $\Delta] \to \Delta\Delta]$ – *complement* Δ,

   (c) $\textcolor{red}{q}\textcolor{green}{y} \to \textcolor{red}{p}\textcolor{blue}{x}$, if $\delta(p, \textcolor{blue}{x}) = (q, \textcolor{green}{y})$,

   (d) $x\textcolor{red}{q} \to \textcolor{red}{p}x$, if $\delta(p, x) = (q, \textcolor{blue}{R})$,

   (e) $\textcolor{red}{q}yx \to y\textcolor{red}{p}x$ for all $y \in \Gamma$, if $\delta(p, x) = (q, \textcolor{blue}{L})$,

   (f) $[q_0\Delta \to \varepsilon, \quad \Delta\Delta] \to \Delta], \quad \Delta] \to \varepsilon$ – *ensuring* $[q_0\Delta w\Delta...\Delta] \overset{+}{\underset{G}{\Rightarrow}} w$.

Now we can easily deduce that $w \in L(M)$, if and only if there exists a derivation $S \Rightarrow_G [q_F\Delta Y\Delta] \Rightarrow_G ... \Rightarrow_G [q_0\Delta w\Delta...] \Rightarrow_G ... \Rightarrow_G w$, and that $L(G) = L(M)$.

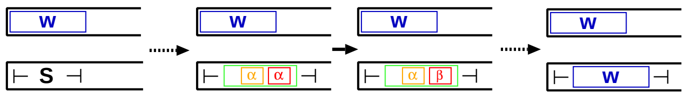
□

6.5.2 Languages of type 0 are accepted by TM

Theorem 6.5.2 Every type-0 language is accepted by some TM (i.e. is recursively enumerable).

Poof. Let $L = L(G)$ for $G = (N, \Sigma, P, S)$ be a language of type 0. *We construct a nondeterministic 2-tape TM M, such that $L(G) = L(M)$:*

- the first tape contains an input string w.
- On the second tape, M *tries to create the derivation of w by simulating the use of rewriting rules* $(\alpha \to \beta) \in P$:



1. Initially, the machine writes the symbol S on the second tape.
2. The machine repeatedly simulates the processing of rules $(\alpha \to \beta) \in P$ on the second tape. It nondeterministically chooses a rule and also an occurence of α on the tape. When rewriting α to β, $|\alpha| \neq |\beta|$, it can use the shift of part of useful content of the tape to the left or right.
3. The machine compares the final content of the second tape with the content of the first tape. If they are same, it halts by moving to q_F. Otherwise, it moves the head to the left until it halts abnormally.

Now we can see that $L(G) = L(M)$. Moreover, $TM\ M$ can be, analogously to multi-tape DTM, converted to a 1-tape NTM and afterwards to a 1-tape DTM. □

6.5.3 Languages of type 0 = languages accepted by TM

Theorem 6.5.3 The class of languages accepted by TM (i.e. the class recursively enumerable languages) is equal to the class of languages of type 0.

Poof. Consequence of two previous theorems. □

6.6 Properties of recursive and recursively enumerable languages

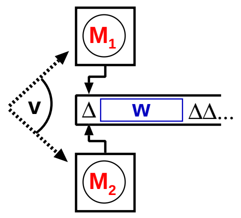
Recursive and recursively enumerable languages are closed under operations of intersection, union and concatenation. The class of recursive languages is also closed under complement.

6.6.1 Closure under $\cup$, $\cap$, . and $*$

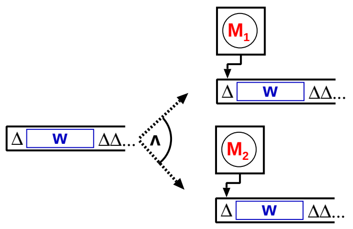
Theorem 6.6.1 The classes of recursive and recursively enumerable languages are closed under operations $\cup$, $\cap$, . and $*$.

Poof. Let L_1, L_2 be languages accepted by TM M_1, M_2. Obviously, we can assume that the sets of states of TM M_1 and TM M_2 are disjoint.

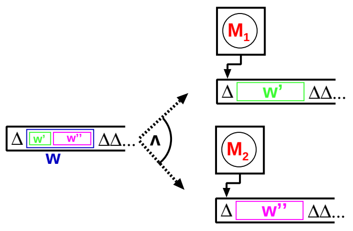
- We construct a NTM $M_{L_1 \cup L_2}$, such that $L(M_{L_1 \cup L_2}) = L_1 \cup L_2$. We unify machines M_1 and M_2 element by element, we establish a new initial state, nondeterministic transitions over Δ/Δ from the new initial state to both original initial states and we merge the original final states to a single final state.



- We construct a three-tape TM $M_{L_1 \cap L_2}$, such that $L(M_{L_1 \cap L_2}) = L_1 \cap L_2$. It copies the input from the first tape to the second, and on that tape, it simulates the machine M_1. If it accepts, it copies the input from the first tape to the third tape, and on that tape, it simulates the machine M_2. If it accepts, then the machine $M_{L_1 \cap L_2}$ accepts too.



- We construct a three-tape NTM $M_{L_1.L_2}$, such that $L(M_{L_1.L_2}) = L_1.L_2$. It copies a nondeterministically chosen prefix of the input from the first tape to the second tape. On that tape, it simulates the machine M_1. If it accepts, it copies the rest of the input from the first tape to the third tape and on that tape, it simulates the machine M_2. If it accepts, then the machine $M_{L_1.L_2}$ accepts too.



- We construct a 2-tape NTM $M_{L_1^*}$, such that $L(M_{L_1^*}) = L_1^*$, which is a generalization of the previous machine. It copies the input from the first tape to the second by parts and on the second tape, it repeatedly simulates the machine M_1. A content of the second tape is delimited with the special characters and it is erased after every simulation of the machine M_1. The machine allows to move the right delimiter further to the right in case when more space is needed.

If the machines M_1 and M_2 are total, it is possible to construct the machines according to foregoing rules as *total* too (for $M_{L_1 \cup L_2}$, $M_{L_1 \cap L_2}$, $M_{L_1 . L_2}$ it is immediate, for $M_{L_1^*}$ we do not admit reading of empty substring of input from the first to the second tape – we only admit to accept empty input in a one-off manner). That proves closure under mentioned operations also for *recursive languages*.

□

6.6.2 (Non-)Closure under complement

Theorem 6.6.2 The class of recursive languages is closed under complement.

Poof. A TM M accepting recursive language L always halts. We can modify M to M', which always moves to unique state q_{reject} while rejecting a string. A TM $\overline{M}$, such that $L(\overline{M}) = \overline{L}$, can be obtained from M' by swapping q_F and q_{reject}. □

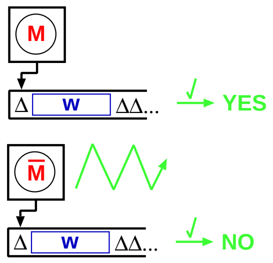
The *class of recursively enumerable languages is not closed under complement!*

- The construction mentioned above cannot be used – looping remains.
- A proof of non-closure will be given in the following chapter.

Theorem 6.6.3 If the languages L and $\overline{L}$ are recursively enumerable, then both of them are recursive.

Poof.

Let M be a *total* TM, such that $L(M) = L$, and $\overline{M}$ be *total* TM, such that $L(\overline{M}) = \overline{L}$. *We construct a total TM accepting the language L in the following way*:



- We will use two tapes. On the first we will simulate M, on the second $\overline{M}$. The *simulation will be interleaved step by step*: a step of M, a step of $\overline{M}$, a step of M, etc.
- We accept if and only if M accepts and we reject by abnormal halting if and only if $\overline{M}$ accepts. One of these situations will occur in finite number of steps.

An existence of total TM for $\overline{L}$ follows from closure of recursive languages under complement.

□

One of the consequences of the foregoing theorems is that for any L and $\overline{L}$, one of the following situations always occurs:

- both L and $\overline{L}$ are recursive,
- neither L nor $\overline{L}$ is recursively enumerable,
- one of these languages is recursively enumerable, but not recursive, the second is not recursively enumerable.

6.7 Linear bounded automata

The last class of Chomsky hierarchy, where we have not seen a counterpart in the form of computing device [4] so far, are grammars of type 1 – context-sensitive grammars. Now we introduce a slightly constrained form of Turing machines that fills this gap.

6.7.1 Linear bounded automata

DEF

A *linear bounded automaton* (LBA) is a *nondeterministic* TM that never leaves the part of the tape where an input is written.

Formally, we can define LBA as a NTM, whose tape alphabet contains special symbol, which serves as a unique end-marker of an input on the tape, and which cannot be rewritten, and from which no right move can be made.

[4] regular grammars correspond to finite automata, context-free grammars correspond to pushdown automata and grammars type 0 correspond to Turing machines

Deterministic LBA can be naturally defined as a (deterministic) TM that never leaves the part of the tape where the an input is written.

It is not known whether deterministic LBA is strictly weaker than LBA or not.

6.7.2 LBA and context-sensitive languages

Theorem 6.7.1 *A* class of languages that can be generated by context-sensitive grammars is the same as the class of languages that can be accepted by LBAs.

Poof.

- We consider the definition of context-free grammars as grammars with rules of the form $\alpha \to \beta$, where $|\alpha| \leq |\beta|$, or $S \to \varepsilon$.
- *LBA* $\longrightarrow$ *G1*:
  - We use similar construction as for *TM* $\longrightarrow$ *G0*.
  - At the beginning, we generate appropriate working area which won't change afterwards: we omit the non-context-sensitive rule $\Delta\Delta] \to \Delta]$.
  - We avoid non-context-sensitive rules $[q_0\Delta \to \varepsilon$ a $\Delta] \to \varepsilon$ (1) by establishing a special final nonterminals integrating the original information and a flag, that denotes if we deal with the first/last symbol, and (2) integration of the symbol representing the control state and the head position with the following tape symbol.
- *G1* $\longrightarrow$ *LBA*:
  - We use a similar construction as for *G0* $\longrightarrow$ *TM*. We do not allow the range of second tape to exceed the range of the first tape.

$\square$

6.7.3 Context-sensitive a recursive languages

Theorem 6.7.2 Every context-sensitive languages is recursive.

Poof.

- *The number of configurations which can appear while accepting w by corresponding LBA M is finite, as it is impossible to extend the working area of the tape.* The number of configurations can be bounded by function c^n for suitable constant c – exponential function follows from the necessity to consider occurrences of all possible symbols at all possible places on the tape.

- The length of notation of any number from the interval $0, ..., c^n - 1$ will never exceed n, if we use c-ary system.
- *We can construct a total 2-tape LBA equivalent with M*:
  - On the first tape, we simulate M.
  - On the second tape, we count the number of steps. If we exceed the maximum given by the exponential function, we reject the input.

□

Theorem 6.7.3 *Not* every recursive language is context-sensitive.

Poof. (Idea) We can use technique of diagonalization which will be presented further. □

6.7.4 Properties of context-sensitive languages

Theorem 6.7.4 The class of context-sensitive languages is closed under operations of $\cup$, $\cap$, $.$, $*$ and complement.

Poof.

- Closure under $\cup$, $\cap$, $.$ and $*$ can be shown in the same way as for recursively enumerable languages.
- Proof of closure under complement is rather complicated (notice that LBA is *nondeterministic* and we cannot use the construction for recursive languages) – the proof can be found in the recommended literature.

□

Let us remark, that we already know that for context-sensitive languages

- *membership of sentence into a language* (recursiveness) can be decided and
- *inclusion of languages* cannot be decided (does not even hold for context-free languages).

Further we can show that *emptiness of language* cannot be decided for context-sensitive languages (by reduction from the Post correspondence problem – see the next chapter).

The Turing machine is an equivalent specification device for any type 0 language and, according to the Church thesis, it is the most powerful formal instrument (together with number of others) for specification of formal languages. It represents a robust mathematical instrument for description of languages and algorithms, whose modeling power is influenced by neither nondeterminism or determinism of machine, nor by its memory resources (multiple tapes). The important subclass of Turing machines, linear bounded automata, determines the class of context-sensitive languages.

6.8 Exercises

Exercise 6.8.1 Explain the meaning of the Church thesis.

Exercise 6.8.2 Define the notions of the Turing machine and languages accepted by Turing machines.

Exercise 6.8.3 Define the notion of multi-tape and nondeterministic Turing machine and decide whether these extensions extends their ability to accept languages.

Exercise 6.8.4 Explain the difference between total and non-total Turing machines and mention the classes of languages which they are able to accept.

Exercise 6.8.5 What is the relation between the class of languages accepted by Turing machines and languages of type 0?

Exercise 6.8.6 Decide, under which language operations the classes of recursive and recursively enumerable languages are closed.

Exercise 6.8.7 Define the linear bounded automata and decide, what is the relation between the class of the languages accepted by linear bounded automata and the class of the type 1 languages.

Chapter 7

Bounds of decidability

15:00

According to the Church thesis, the class of algorithmically solvable – computable problems is defined by Turing machines. By using Turing machines, we have established the basic classification of problems: decidable, partially decidable and undecidable problems. We will now look more closely into features of these particular classes and into problems which belong to these classes. We will also find answer for questions like:

– Are there languages (problems) which are not recursively enumerable (partially decidable)?

– Which languages (problems) are not recursive (decidable)?

– How can we specify to which class a particular language (problem) belongs?

We will also define a device for specification of languages (algorithms) equivalent to Turing machines – partially recursive functions.

7.1 Languages outside of the class 0

There exist languages outside of the class 0 (algorithmically unsolvable problems). We can even say, that (partial) decidability (algorithmic solvability) is an exception among languages.

7.1.1 Existence of languages outside of class 0

Theorem 7.1.1 For each alphabet Σ there exists a language over Σ which is not recursive (which is not a language of the class 0).

Poof.

1. Any language from the class 0 over the alphabet Σ can be accepted by a TM with $\Gamma = \Sigma \cup \{\Delta\}$: If M uses more symbols, we can encode them as certain sequences of symbols from $\Sigma \cup \{\Delta\}$ and construct a TM M' which simulates M.

2. Now, we can systematically list all TMs with $\Gamma = \Sigma \cup \{\Delta\}$. We start with machines having two states, then with three states, etc.

   Conclusion: The set of all such machines (and hence, the set of languages of type 0) is countable.

3. However, the set Σ^* contains uncountable number of strings and therefore the *set* 2^{Σ^*}*, including all languages, is uncountable* – see the proof of the following lemma.
4. The validity of the theorem follows from the difference of cardinalities of countable and uncountable sets.

□

Lemma 7.1.1 For a nonempty and finite set Σ, the set 2^{Σ^*} is uncountable.

Poof. We will use *diagonalization* (first time used by Cantor in the proof of different cardinality of $\mathbb{N}$ and $\mathbb{R}$) to prove the lemma

- We assume that 2^{Σ^*} is countable. Then, according to the definition of countability there is a *bijection* $f : \mathbb{N} \longleftrightarrow 2^{\Sigma^*}$.
- We order Σ^* into a sequence $w_1, w_2, w_3, ...$, e.g. ε, x, y, xx, xy, yx, yy, xxx, ... for $\Sigma = \{x, y\}$. Now, the function f can be represented as an *infinite matrix*:
$$\begin{array}{cccccc} & w_0 & w_1 & w_2 & \dots & w_i & \dots \\ L_0 = f(0) & a_{00} & a_{01} & a_{02} & \dots & a_{0i} & \dots \\ L_1 = f(1) & a_{10} & a_{11} & a_{12} & \dots & a_{1i} & \dots \\ L_2 = f(2) & a_{20} & a_{21} & a_{22} & \dots & a_{2i} & \dots \\ \dots \end{array}$$
where $a_{ij} = \begin{cases} 0, \text{ if } w_j \notin L_i, \\ 1, \text{ if } w_j \in L_i. \end{cases}$
- Consider the language $\overline{L} = \{w_i \mid a_{ii} = 0\}$. $\overline{L}$ differs from every language $L_i = f(i)$, $i \in \mathbb{N}$:
  - if $a_{ii} = 0$, then w_i belongs to the language,
  - if $a_{ii} = 1$, then w_i does not belong to the language.
- At the same time $\overline{L} \in 2^{\Sigma^*}$, therefore f is not surjective, which is a contradiction.

□

7.2 Halting problem

The problem, whether the computation of a given TM halts, is the first problem which we show not to be recursive. In order to prove that this problem is undecidable we establish *an universal Turing machine* by using a suitable encoding of a TM. The universal Turing machine is able to simulate a computation of any other TM (an input of the universal machine consists of a TM which is to be simulated and of an initial configuration of that TM). As we will show, the notion of the universal Turing machine is important not only for the proof of undecidability of the halting problem.

7.2.1 Encoding of TM

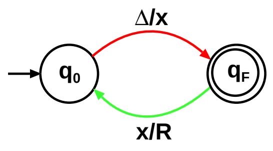
An encoding system for TM involves (1) encoding of states (so that all states – including q_0 and q_F – will be distinguished), (2) encoding of symbols from Γ and (3) encoding of the transition function δ.

An encoding of states: We order the set of states Q into a sequence q_0, q_F, q, p, ..., t. The state q_j is encoded as 0^j.

An encoding of symbols and commands L/R: Assume that $\Gamma = \Sigma \cup \{\Delta\}$. We order Σ into a sequence a_1, a_2, ..., a_n and we choose the following codes: $\Delta \mapsto \varepsilon,\ L \mapsto 0,\ R \mapsto 00,\ a_i \mapsto 0^{i+2}$.

A transition $\delta(p, x) = (q, y)$, where $y \in \Gamma \cup \{L, R\}$, is represented as a 4-tuple (p, x, q, y) and encoded *by concatenating the codes* for p, x, q, y while using 1 *as a separator*, i.e. as $\langle p\rangle 1\langle x\rangle 1\langle q\rangle 1\langle y\rangle$, where $\langle\_\rangle$ denotes the code of _.

The code of a *TM* is a *sequence of codes of its transitions* separated and bordered by 1.

Example 7.2.1 

code: 1110100010100011001

7.2.2 Universal TM

Universal TM establishes the concept of *"programmable" machine*, which allows to specify a particular TM (i.e. a program) and input data (on which that particular TM works) in the input string.

We will encode the TM, which is to be simulated, in the way mentioned above. The input string will be encoded as a sequence of corresponding codes of symbols separated and bordered by 1. The codes of machine and input string will be separated e.g. by #.

Example 7.2.2 *A TM from the previous page having xxx on its input:*

1110100010100011001#1000100010001

The universal TM, which processes this input, can be designed as a 3-tape machine:

- the input (and afterwards the output) is on the *first tape*,
- *the second tape* is used for simulation of the working tape of the original machine and
- the current state of the simulated machine and the actual position of the head (position of the head i is encoded as 0^i) is written on the *third tape*.

The Universal machine works as follows:

1. The machine checks if the input has a correct form ($M\#w$). If not, it halts abnormally.
2. It copies w to the second tape and writes the code of the initial configuration on the third tape.
3. It finds current symbol s under the head of the simulated machine on the second tape; then it finds the transition executable from the state written at the beginning of the third tape for the symbol s on the first tape. If there is no such transition available, the machine halts abnormally.
4. The machine will perform changes on the second and the third tape corresponding to the simulated transition (rewriting of the current symbol, changing the position of the head, changing the control state).
5. If the state q_F of the simulated machine has not been reached the computation continues on the item 2. Otherwise, the machine deletes the first tape, copies the content of the second tape on the first tape and halts normally (it performs transition into the final state).

This machine can be transformed to a *single-tape universal TM*, which we will denote as T_U.

7.2.3 Halting problem of TM

Theorem 7.2.1 Halting problem of TM – the problem wheather a given TM M halts for a given input string w **is not decidable** but is **partially decidable**.

!

Poof.

- Halting problem corresponds to the language $HP = \{\langle M\rangle\#\langle w\rangle \mid M \textit{ halts with } w\}$, where $\langle M\rangle$ is the code of TM M and $\langle w\rangle$ is the code of w.
- *Partial decidability* can be easily proved by using a modified T_U which halts by accepting an input $\langle M\rangle\#\langle w\rangle$, if and only if M halts on w – the modification grounds in transforming an abnormal halting to halting by a move to the state q_F.
- We show *undecidability* by using *diagonalization*:
  1. For $x \in \{0,1\}^*$, let M_x be a TM with code x, if x is a legal code of a TM. Otherwise, M_x is identified with an arbitrarily chosen TM – e.g. one which halts immediately for any input.
  2. Now we can create a sequence $M_\varepsilon, M_0, M_1, M_{00}, M_{01}, M_{10}, M_{11}, M_{000}, ...$ containing all TMs over $\Sigma = \{0,1\}$ indexed by strings from $\{0,1\}^*$.

3. Consider an infinite matrix

$$\begin{array}{lllllll}
 & \varepsilon & 0 & 1 & 00 & 01 & 10 \quad \dots \\
M_\varepsilon & H_{M_\varepsilon,\varepsilon} & H_{M_\varepsilon,0} & H_{M_\varepsilon,1} & H_{M_\varepsilon,00} & H_{M_\varepsilon,01} & \dots \\
M_0 & H_{M_0,\varepsilon} & H_{M_0,0} & H_{M_0,1} & H_{M_0,00} & H_{M_0,01} & \dots \\
M_1 & H_{M_1,\varepsilon} & H_{M_1,0} & H_{M_1,1} & H_{M_1,00} & H_{M_1,01} & \dots \\
M_{00} & H_{M_{00},\varepsilon} & H_{M_{00},0} & H_{M_{00},1} & H_{M_{00},00} & H_{M_{00},01} & \dots \\
M_{01} & H_{M_{01},\varepsilon} & H_{M_{01},0} & H_{M_{01},1} & H_{M_{01},00} & H_{M_{01},01} & \dots \\
\dots & & & & & &
\end{array}$$

where $H_{M_x,y} = \begin{cases} \mathbf{L}, \text{ if } M_x \text{ is looping on } y, \\ \mathbf{H}, \text{ if } M_x \text{ halts on } y. \end{cases}$

4. *Assume that there exists a* complete *TM* K accepting the language HP. Then, for an input $\langle M\rangle\#\langle w\rangle$, the machine K
   - halts normally (accepts), if and only if M halts on w,
   - halts abnormally (reject), if and only if M is looping on w.

5. *We construct a TM* N, which for an input $x \in \{0,1\}^*$:
   - constructs M_x from x and writes $\langle M_x\rangle\#x$ to its tape.
   - simulates K on $\langle M_x\rangle\#x$, and accepts, if K rejects, and moves to an infinite cycle, if K accepts.

   Notice that N *actually complements the diagonal* of the mentioned matrix:

$$\begin{array}{lllllll}
 & \varepsilon & 0 & 1 & 00 & 01 & 10 \quad \dots \\
M_\varepsilon & H_{M_\varepsilon,\varepsilon} & H_{M_\varepsilon,0} & H_{M_\varepsilon,1} & H_{M_\varepsilon,00} & H_{M_\varepsilon,01} & \dots \\
M_0 & H_{M_0,\varepsilon} & H_{M_0,0} & H_{M_0,1} & H_{M_0,00} & H_{M_0,01} & \dots \\
M_1 & H_{M_1,\varepsilon} & H_{M_1,0} & H_{M_1,1} & H_{M_1,00} & H_{M_1,01} & \dots \\
M_{00} & H_{M_{00},\varepsilon} & H_{M_{00},0} & H_{M_{00},1} & H_{M_{00},00} & H_{M_{00},01} & \dots \\
M_{01} & H_{M_{01},\varepsilon} & H_{M_{01},0} & H_{M_{01},1} & H_{M_{01},00} & H_{M_{01},01} & \dots \\
\dots & & & & & &
\end{array}$$

6. we obtain that

$$\begin{array}{llll}
N \text{ halts on } x & \Leftrightarrow & K \text{ rejects } \langle M_x\rangle\#\langle x\rangle & \text{(definition of } N) \\
 & \Leftrightarrow & M_x \text{ is looping on } x & \text{(assumption about } K).
\end{array}$$

7. But this means that N *differs from each* M_x at least on one string – namely, the string x. This is a *contradiction* with the fact that the sequence $M_\varepsilon, M_0, M_1, M_{00}, M_{01}$,
   M_{10}, $M_{11}, M_{000}, \dots$ contains all TMs over $\Sigma = \{0,1\}$.

   This contradiction results from the assumption that there exists a TM K which decides whether a given TM M halts on a given input x or not.

□

We have shown that the language HP is recursively enumerable and not recursive, and therefore the halting problem of TM is partially decidable. Then, from the theorem 8.6 it follows that the ***complement of the halting problem*** *is not even partially decidable* and the *language co-*$HP = \{\langle M\rangle\#\langle w\rangle \mid M$ *doesn't halt on* $w\}$ *is an example of a language that is not even recursively enumerable.* We will see another example of such language later.

7.3 Reduction

A reduction is a basic technique of problems classification with respect to computability. It is an algorithmic transformation of one problem to another problem. Informally, it is a computable reduction function f which, for an instance I of a problem P, assigns an instance $f(I)$ of a problem Q in the way that the solution of $f(I)$ is a solution of I [1].

7.3.1 Proof of undecidability by using reduction

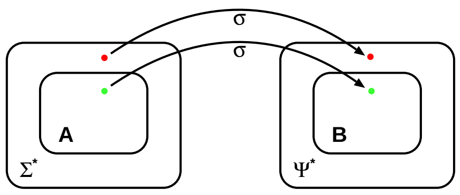
Along with diagonalization, the technique of *reduction* is one of the most used techniques for proving that a given problem is not decidable (partially decidable), or that a language is not recursive (recursively enumerable):

- we know that the language A is not recursive (recursively enumerable),
- we investigate the language B,
- we show that A *can be transformed by a complete TM to* B,
- but this means that B is also not recursive (recursively enumerable) – otherwise we could use the complete TM (the non-complete TM) accepting B and particular reductions to construct the complete TM (the non-complete TM) accepting A, which is a contradiction.

This argument also shows that reduction can be also used in proofs that a given problem is recursive (partially recursive).

DEF

Definition 7.3.1 *Let* A*,* B *be languages,* $A \subseteq \Sigma^*$*,* $B \subseteq \Psi^*$*.* A reduction of the language A to the language B is a recursively enumerable function $\sigma : \Sigma^* \to \Psi^*$ such that that $w \in A \Leftrightarrow \sigma(w) \in B$*.*



[1] P and Q are decision problems

If there exists a reduction of the language A to the language B, we say that A is reducible to B, which we denote as $A \leq B$.

Theorem 7.3.1 *Let $A \leq B$.*

1. *If the language A is not recursively enumerable, then neither the language B is recursively enumerable.*

2. *If the language A is not recursive, then neither the language B is recursive.*

$\overline{1}$. *If the language B is recursively enumerable, then the language A is recursively enumerable too.*

$\overline{2}$. *If the language B is recursive, then the language A is recursive too.*

Poof. We will prove that if $A \leq B$ then the statement $\overline{1}$ holds i.e. if the language B is recursively enumerable, then the language A is recursively enumerable too:

- Let M_R be a complete TM computing the reduction function σ from A to B and M_B is a TM accepting B.
- *We construct the M_A accepting A*:
  1. M_A simulates M_R on the input w; it transforms the content of the tape to $\sigma(w)$.
  2. M_A simulates a computation of M_B on $\sigma(w)$.
  3. If M_B halts and accepts, M_A also halts and accepts, otherwise M_A halts abnormally or loops.
- Apparently, the following holds:

$$\begin{array}{rcll} M_A \text{ accepts } w & \Leftrightarrow & M_B \text{ accepts } \sigma(w) & \\ & \Leftrightarrow & \sigma(w) \in B & \\ & \Leftrightarrow & w \in A & \text{(definition of reduction).} \end{array}$$

The proposition (1) is a contraposition of ($\overline{1}$); The proposition ($\overline{2}$) can be proved similarly as ($\overline{1}$) using the complete TM M_B; The proposition (2) is the contraposition of[2] *($\overline{2}$).*

□

7.4 Membership problem and other problems

The reduction from the halting problem brings many "bad news" about undecidability of many important problems, e.g. the problem if a given word belongs to a given language.

[2]contraposition: $p \rightarrow q \Leftrightarrow \neg q \rightarrow \neg p$ (Modus tollens).

7.4.1 Membership problem for $\mathcal{L}_0$

Theorem 7.4.1 Membership problem *of a string w to a language L of type 0* ***is not decidable****, but it* ***is partially decidable.***

Poof. Partial decidability is obvious: we simply use the T_U which will simulate the TM M, $L(M) = L$, on the given string w. We show the *undecidability* by using the *reduction from halting problem* :

- Any *TM M can be easily modified to M' which accepts w if and only if M halts on the input w (by accepting or rejecting)*. It is sufficient to:
  - add all the "missing" transitions, leading to q_F
  - add a unique symbol marking the left edge of the tape
  - add a transition to q_F, that will be used whenever the head moves to this marking symbol.
- This transformation can be easily realized at the code level of Turing machines by a *complete TM* – it will implement the *reduction of language HP to language $MP = \{\langle M\rangle\#\langle w\rangle \mid w \in L(M)\}$*.
- Thus, we are able to reduce HP by a complete TM to MP and at the same time we know that HP is not recursive. Then it follows from the theorem 10.3 (2) that MP is not recursive, and therefore membership problem is not decidable for languages of type 0.

□

Similarly as in the case of halting problem we can prove, using the theorem 8.6, that

- **complement of the membership problem** is not even partially decidable and
- the language co-$MP = \{\langle M\rangle\#\langle w\rangle \mid w \notin L(M)\}$ is an another example of a language which is not even recursively enumerable.

7.4.2 Examples of other problems for TM

$\boxed{\texttt{x+y}}$

By constructing a suitable *total TM* (and by the proof of its correctness of such construction for more complicated cases), we can show that e.g. *the following problems are decidable*:

- A given TM has at least 2005 states.
- A given TM will perform more than 2005 steps on an input ε.
- A given TM will perform more than 2005 steps on *some* input.

By constructing a suitable *(non-total) TM* and *by the proof of non-recursiveness of reductions*, we can show that the *following problems are partially decidable*:

- The language of a given TM is non-empty.
- The language of a given TM contains at least 2005 words.

By a proof by reduction we can show that languages corresponding to the following problems *are not even partially decidable*:

- The language of a given TM is non-empty.
- The language of a given TM contains at most 2005 words.
- The language of a given TM is finite (regular, context-free, context-sensitive, recursive).

7.5 Post correspondence problem

The Post correspondence problem, similarly as the halting problem, is important as a basis when proving undecidability of number of problems. For many of them, the reduction from HP would be hard to find.

7.5.1 Post correspondence problem

Definition 7.5.1

DEF

- The Post system *over the alphabet* Σ *is given by a non-empty list* S *of tuples of non-empty strings over* Σ, $S = \langle(\alpha_1, \beta_1), ..., (\alpha_k, \beta_k)\rangle$, $\alpha_i, \beta_i \in \Sigma^+$, $k \geq 1$.
- A solution of a Post system *is every non-empty sequence of natural numbers* $I = \langle i_1, i_2, ..., i_m\rangle$, $1 \leq i_j \leq k$, $m \geq 1$, *such that:*

$$\alpha_{i_1}\alpha_{i_2}...\alpha_{i_m} = \beta_{i_1}\beta_{i_2}...\beta_{i_m}$$

  (Notice: m is not bounded and indices may repeat!)

- Post correspondence problem (PCP) is: *Does a solution exists for a given Post system?*

Example 7.5.1

x+y

- *Consider a Post system* $S_1 = \{(b, bbb), (babbb, ba), (ba, a)\}$ *over* $\Sigma = \{a, b\}$. *This system has a solution* $I = \langle 2, 1, 1, 3\rangle$: $babbb\ b\ b\ ba = ba\ bbb\ bbb\ a$.
- *On the contrary, a Post system* $S_2 = \{(ab, abb), (a, ba), (b, bb)\}$ *over* $\Sigma = \{a, b\}$ *does not have a solution, because* $|\alpha_i| < |\beta_i|$ *for* $i = 1, 2, 3$.

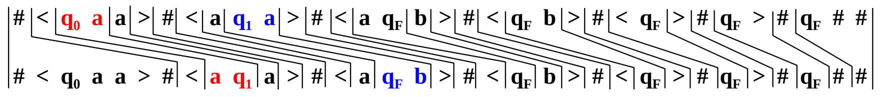
7.5.2 Undecidability of PCP

Theorem 7.5.1 Post correspondence problem is undecidable.

Poof. (Idea) It can be shown that undecidability of PCP follows from the undecidability of the so-called *initial PCP*, in which we require that its solution always starts with the integer 1. Undecidability of the initial PCP can be shown *by a reduction from the membership problem of TM*:

- A configuration of a computation of a TM can be encoded as a string: we bound the used content of the tape by special marks (we remember only this content), we insert the control state at the actual position of the head of the tape.
- The sequence of configurations of TM, when accepting the string, will be represented as a concatenation of strings, which arises from the solution of PCP.
- One of the concatenations considered will be always longer (except the last phase): at the beginning it will contain the initial configuration and then it will be always one step ahead. In the last phase of computation we "pad" the concatenation (if the computation of the simulated TM will be possible to be halted by accepting).
- We will model the computation of TM in the way that we always successively prolong one concatenation by actual configuration of the simulated TM and at the same time, we generate a new configuration of TM in the second concatenation.
- *The tuples of PCP will model following steps*:
  - Inserting the initial configuration of the simulated TM to one of the concatenations, e.g. right-sided (#, #initial_configuration), we use $\# \notin \Gamma$ as a separator of configurations.
  - Copying of symbols on the tape before and behind the actual head position (z, z) for all $z \in \Gamma \cup \{\#, <, >\}$, where $<, >$ borders the used part of the tape.
  - The basic change of configuration: rewriting $\delta(q_1, a) = (q_2, b)$: $(q_1 a, q_2 b)$, move to the right $\delta(q_1, a) = (q_2, R)$: $(q_1 a, a q_2)$, move to the left $\delta(q_1, b) = (q_2, L)$: $(a q_1 b, q_2 a b)$ for all $a \in \Gamma \cup \{<\}$. Moreover, we need to handle a movement of the head to $>$: reading Δ, extending the used part of the tape.
  - Rules for "padding" of both concatenations when accepting: we allow to add a symbol in the neighborhood of q_F on the left side, without adding it on the right side.

- A simulation of computation of TM, which reads a, moves the head to the right, rewrites a to b and halts, would look like this for the input aa:



# < q0 a a > # < a q1 a > # < a qF b > # < qF b > # < qF > # qF > # qF # #
# < q0 a a > # < a q1 a > # < a qF b > # < qF b > # < qF > # qF > # qF # #

- *The correctness of the construction* can be proved by induction on the length of the computation.

□

7.5.3 Undecidability of the reductions from PCP

Reductions from PCP or its complement are very often used for proofs of undecidability.

As an example, we will show *the proof of undecidability of the problem of emptiness of the language generated by a given context-sensitive grammar*:

- We use *a reduction from the complement of PCP*. To a given list $S = (\alpha_1, \beta_1), ..., (\alpha_k, \beta_k)$, which defines an instance of PCP, the reduction assigns a context-sensitive grammar G, such that the PCP based on S has no solution, if and only if $L(G) = \emptyset$.

- Consider the languages L_α, L_β over $\Sigma \cup \{\#, 1, ..., k\}$ (we assume that $\Sigma \cap \{\#, 1, ..., k\} = \emptyset$):

  - $L_\alpha = \{\alpha_{i_1} ... \alpha_{i_m} \# i_m ... i_1 \mid 1 \leq i_j \leq k, j = 1, ..., m, m \geq 1\}$,
  - $L_\beta = \{\beta_{i_1} ... \beta_{i_m} \# i_m ... i_1 \mid 1 \leq i_j \leq k, j = 1, ..., m, m \geq 1\}$.

- It is obvious that L_α, L_β are context-sensitive (they are even deterministic context-free), thus $L_\alpha \cap L_\beta$ is also context-sensitive (theorem 8.10) and we can effectively design the grammar G, which generates this language (e.g. by construction of LBA).

- Evidently, $L_\alpha \cap L_\beta$ contains exactly the strings $u\#v$, where v corresponds to the inversion of the solution of the given instance of PCP.

- The reduction therefore maps a given instance of PCP to a grammar G.

□

7.5.4 Summary of some languages properties

We will now show the summary of some important properties of various classes of languages; we have proved some of them already, proofs of others can be found in literature[3] (reduction from PCP is often used in order to prove undecidability):[4]

	Reg	DCF	CF	CS	Rec	RE
$w \in L(G)$?	R	R	R	R	R	N
Is $L(G)$ empty? finite?	R	R	R	N	N	N
$L(G) = \Sigma^*$?	R	R	N	N	N	N
$L(G) = R$, $R \in \mathcal{L}_3$?	R	R	N	N	N	N
$L(G_1) = L(G_2)$?	R	R	N	N	N	N
$L(G_1) \subseteq L(G_2)$?	R	N	N	N	N	N
$L(G_1) \in \mathcal{L}_3$?	A	R	N	N	N	N
Is $L(G_1) \cap L(G_2)$ of the same type?	A	N	N	A	A	A
Is $L(G_1) \cup L(G_2)$ of the same type?	A	N	A	A	A	A
Is the complement of $L(G)$ of the same type?	A	A	N	A	A	N
Is $L(G_1).L(G_2)$ of the same type?	A	N	A	A	A	A
Is $L(G)^*$ of the same type?	A	N	A	A	A	A
Is G ambiguous?	R	N	N	N	N	N

7.6 Rice theorem

Rice theorem is an important instrument for computability classification of languages (based on the reduction from HP and $co - HP$). It shows that **decidability is an exception which proves the rule.**

7.6.1 Rice theorem – first part

Theorem 7.6.1 ***Every* nontrivial property of recursively enumerable languages is undecidable.**

DEF

Definition 7.6.1 *Let Σ be an alphabet.* A property of recursively enumerable sets *is a mapping P : \{ recursively enumerable subsets of set Σ^* \} $\rightarrow \{\bot, \top\}$, where $\top$ or $\bot$ represent true or false.*

Example 7.6.1 *The emptiness property can be represented as a mapping*

$$P(A) = \begin{cases} \top, & \textit{if } A = \emptyset, \\ \bot, & \textit{if } A \neq \emptyset. \end{cases}$$

Let us stress that now we talk about properties of recursively enumerable sets, *not* about properties of TMs, which accept them. Therefore, the following properties are not properties of recursively enumerable sets:

- A TM M has at least 2005 states.

[3] e.g. I. Černá, M. Křetínský, A. Kučera. Automaty a formální jazyky I. FI MU, 1999.
[4] R = decidable, N = undecidable, A = implicitly true

- A TM M halts for all inputs.

Definition 7.6.2 *The property of recursively enumerable sets is* nontrivial *if it is neither always true (for all r.e. sets) nor always false.*

7.6.2 Proof of the 1. part of Rice theorem

Poof.

- Let P be a nontrivial property of the r.e. sets. We assume, without loss of generality, that $P(\emptyset) = \bot$, we can proceed analogically for $P(\emptyset) = \top$.
- Because P is a nontrivial property, there exists a r.e. set A, such that $P(A) = \top$. Let K be a TM accepting A.
- *We reduce HP to* $\{\langle M\rangle \mid P(L(M)) = \top\}$*.* We construct $\sigma(\langle M\rangle\#\langle w\rangle) = \langle M'\rangle$ from $\langle M\rangle\#\langle w\rangle$, where M' is 2-tape TM, which on input x:
  1. Stores x on the 2. tape.
  2. Writes w on the 1. tape – w is "saved" in the control of M'.
  3. Simulates M on the 1. tape – M is also "saved" in the control of M'.
  4. If M halts on w, it simulates K on x and accepts, if K accepts.
- We obtain:
  - M halts on $w \Rightarrow L(M') = A \Rightarrow P(L(M')) = P(A) = \top$,
  - M loops on $w \Rightarrow L(M') = \emptyset \Rightarrow P(L(M')) = P(\emptyset) = \bot$,

  Then we have a real reduction of HP to $\{\langle M\rangle \mid P(L(M)) = \top\}$.

  As HP is not recursive, neither $P(L(M))$ is recursive, and therefore it is not decidable whether $L(M)$ satisfies the property P.

□

7.6.3 Rice theorem – second part

Definition 7.6.3 *We call a property P of r.e. sets* monotonic*, if for all two r.e. sets A, B, such that $A \subseteq B$, holds $P(A) \Rightarrow P(B)$.*

DEF

Example 7.6.2 *The following properties are* monotonic*:*

- *A is infinite.*
- *$A = \Sigma^*$.*

On the contrary, the following properties are non-monotonic*:*

- *A is finite.*
- *$A = \emptyset$.*

Theorem 7.6.2 ***Every* nontrivial property of recursively enumerable languages is not even partially undecidable.**

Poof. By reduction from co-HP – see e.g. D. Kozen. Automata and Computability. □

7.7 Alternatives to Turing machine

The following mechanisms of computation have equivalent computing power as TM: *automata with (one) queue*:

- Consider a machine with a finite control, (unbounded) FIFO queue and transitions, which allows to read from the beginning of the queue and to write symbols from the queue alphabet Γ to the end of the queue.
- Evidently, by "rotating" the queue, we can simulate the tape of TM.

Pushdown automata with two (and more) stacks have equivalent computing power as TMs too:

- Intuitively: we have a content of tape of simulated TM in one stack; if we want to change it (generally not only on its top), we move a part of it to the second stack so that we can get to the required place, then we perform the appropriate change and return back the removed part of the stack.
- Note: we also know that we can implement a queue by using two stacks.

Another computing models with the same power as TM are *automata with counters (for two and more counters) and with operations* $+1$, -1 *and test of* 0:

- These automata have the finite control and k counters, which can be independently incremented, decremented and tested for zero (a transition is conditioned by a zero test on some counter).
- We can simulate two stacks by using *four counters*:

  - For PDAs, it is sufficient to have $\Gamma = \{0, 1\}$: we can encode different symbols with certain number of 0 separated by 1. Then the content of the stack has a nature of a number in binary notation. Insertion of 0 corresponds to multiplication by 2, removal of 0 corresponds to division by 2. Similarly for the case of insertion/removal of 1.
  - We can simulate a binary stack with two counters: when multiplying/dividing by 2 we subtract 1 (or 2) from one counter and add 2 (or 1) to the second counter.

- Even *two counters* are sufficient:
  - The content of four counters i, j, k, l can be encoded as $2^i 3^j 5^k 7^l$.
  - Then it is possible to realize addition/subtraction as multiplication/division by 2, 3, 5, or 7.

Another Turing complete computing mechanisms are λ-*calculus* or *partially-recursive function* (see following chapter).

7.8 Computable functions

7.8.1 Basics of the theory of recursive functions

We will try to identify functions, which are "computable", i.e. enumerable in general sense (regardless of particular computing system). In order to lower the extreme size of the class of these functions, which is given by variety of domains and ranges, we will consider – while considering the possibility of encoding – the functions of these forms:

$$\boxed{f : \mathbb{N}^m \to \mathbb{N}^n}$$

where $\mathbb{N} = \{0, 1, 2, \ldots\}, \quad m, n \in \mathbb{N}$.
Convention: n-tuple $(x_1, x_2, \ldots, x_n) \in \mathbb{N}^n$ will be denoted as $\overline{x}$
Classification of partial functions:

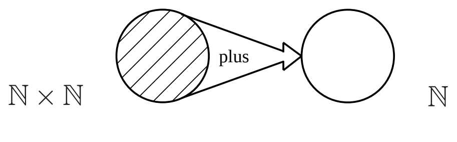
DEF

- *Total function over the set* X[5] – the domain is the whole X
- *Strictly partial function over the set* X – there is at least one element $x \in X$ for which the function is undefined.

x+y

Example 7.8.1 *Total function* plus

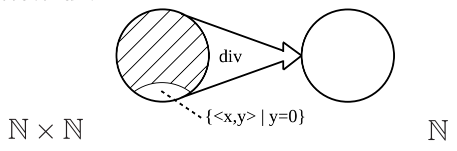
$$plus : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$
$$plus(x, y) = x + y$$



[5]In the following text, the set X will usually be $\mathbb{N}^k$.

Example 7.8.2 *Strictly partial function* div

$$div : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$$
$$div(x, y) = \textit{whole part } x/y, \textit{ if } y \neq 0$$



7.8.2 Initial function

The hierarchy of computable functions is based on so-called *initial functions* which are sufficiently elementary and which form "building blocks" of higher functions.

Those functions are:

DEF

1. *Zero function* : $\xi() = 0$
   maps "empty n-tuple" $\mapsto 0$

2. *Successor function* : $\sigma : \mathbb{N} \rightarrow \mathbb{N}$
   $\sigma(x) = x + 1$

3. *Projection* : $\pi^n_k : \mathbb{N}^n \rightarrow \mathbb{N}$
   Chooses k-th item from a n-tuple, e.g.: $\pi^3_2(7, 6, 4) = 6$ and $\pi^2_1(5, 17) = 5$

   Special case: $\pi^n_0 : \mathbb{N}^n \rightarrow \mathbb{N}^0$, i.e. e.g. $\pi^3_0(1, 2, 3) = ()$

7.8.3 Primitive recursive function

Now we define three ways of creating new, more complicated functions. *Primitive recursive functions* will be created from initial functions in this way:

DEF

1. *Combination*:
   By a combination of two functions $f : \mathbb{N}^k \rightarrow \mathbb{N}^m$ and $g : \mathbb{N}^k \rightarrow \mathbb{N}^n$, we get a function for which:

   $$\boxed{\begin{array}{l} f \times g : \mathbb{N}^k \rightarrow \mathbb{N}^{m+n} \\ f \times g(\overline{x}) = (f(\overline{x}), g(\overline{x})),\ \overline{x} \in \mathbb{N}^k \end{array}}$$

   E.g.: $\pi^3_1 \times \pi^3_3(4, 12, 8) = (4, 8)$

2. *Composition*: Composition of two functions $f : \mathbb{N}^k \rightarrow \mathbb{N}^m$ and $g : \mathbb{N}^m \rightarrow \mathbb{N}^n$ is a function for which:

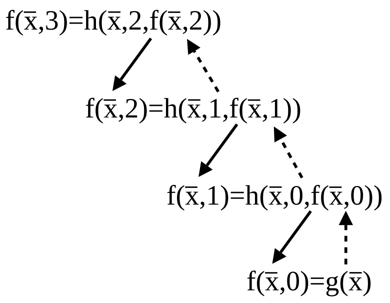
   $$\boxed{\begin{array}{l} g \circ f : \mathbb{N}^k \rightarrow \mathbb{N}^n \\ g \circ f(\overline{x}) = g(f(\overline{x})),\ \overline{x} \in \mathbb{N}^k \end{array}}$$

   E.g.: $\sigma \circ \xi() = 1$
   $\sigma \circ \sigma \circ \xi() = 2$

3. *Primitive recursion* is a technique which allows to create a function $f : \mathbb{N}^{k+1} \to \mathbb{N}^m$, which is based on two other functions $g : \mathbb{N}^k \to \mathbb{N}^m$ and $h : \mathbb{N}^{k+m+1} \to \mathbb{N}^m$, by equations:

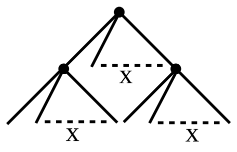
$$\boxed{\begin{array}{l} f(\overline{x}, 0) = g(\overline{x}) \\ f(\overline{x}, y+1) = h(\overline{x}, y, f(\overline{x}, y)),\ \overline{x} \in \mathbb{N}^k \end{array}}$$

The illustration of computation schema (for $y = 3$):



$\boxed{x+y}$

Example 7.8.3 *Assume that we want to define a function $f : \mathbb{N}^2 \to \mathbb{N}$, whose values $f(x, y)$ give the number of vertices of x-ary tree of the depth y, which is regular (complete):*



Obviously:

1. $f(x, 0) = 1$
2. *tree of the depth y has x^y leaves. When increasing the depth by one to $y + 1$, we have to add $x^{y+1} = x^y \cdot x$ vertices*

Thus the function f can be defined by the following formula: $f(x, 0) = x^0$, $f(x, y + 1) = f(x, y) + x^y \cdot x$.
E.g.: $f(3, 2) = f(3, 1) + 3^1 \cdot 3 = f(3, 0) + 3^0 \cdot 3 + 3^1 \cdot 3 = 1 + 3 + 9 = 13$

Example 7.8.4 *Consider function $plus : \mathbb{N}^2 \to \mathbb{N}$. It can be defined by using primitive recursion in this way:*

$$\begin{array}{l} plus(x, 0) = \pi_1^1(x) \\ plus(x, y + 1) = \sigma \circ \pi_3^3(x, y, plus(x, y)) \end{array}$$

which expresses:

1. $x + 0 = x$
2. $x + (y + 1) = (x + y) + 1 = \sigma(x + y)$

DEF

Definition 7.8.1 The class of primitive recursive functions *contains all functions that can be created from the initial functions by:*

1. *combination,*
2. *composition and*
3. *primitive recursion.*

Theorem 7.8.1 *Every primitive recursive function is a total function.*

Poof. Initial functions are total. By applying combination, composition and primitive recursion on total functions, we obtain total functions. $\square$

7.8.4 Examples of primitive recursive functions

The class of primitive recursive functions contains the majority of functions which are typical in computer applications. We will show how to define them from initial functions using combination, composition and primitive recursion.

Convention: We will sometimes use the notation $h(x, y, z) = \mathit{plus}(x, z)$ or $h(x, y, z) = x+z$ instead of the functional notation of type $h \equiv \mathit{plus} \circ (\pi^3_1 \times \pi^3_3)$.

Constant function: We establish function κ^n_m, which assigns the constant value $m \in \mathbb{N}$ to any n-tuple $\overline{x} \in \mathbb{N}^n$

$$\kappa^0_m \equiv \underbrace{\sigma \circ \sigma \circ \ldots \circ \sigma}_{m-\text{times}} \xi$$

κ^n_m is recursive primitive function also for $n > 0$:

$$\kappa^n_m(\overline{x}, 0) = \kappa^{n-1}_m(\overline{x})$$
$$\kappa^n_m(\overline{x}, y + 1) = \pi^{n+1}_{n+1}(\overline{x}, y, \kappa^n_m(\overline{x}, y))$$

x+y

E.g.: $\kappa^2_3(1, 1) = \pi^3_3(1, 0, \kappa^2_3(1, 0)) = \kappa^2_3(1, 0) = \kappa^1_3(1) = \kappa^1_3(0) = \kappa^0_3() = 3$. By combination of functions κ^n_m we obtain constants from $\mathbb{N}^n$, $n > 1$.

E.g.: $\kappa^3_2 \times \kappa^3_5(x, y, z) = (2, 5)$

Multiplication function:

$$\mathit{mult}(x, 0) = \kappa^1_0(x)$$
$$\mathit{mult}(x, y + 1) = \mathit{plus}(x, \mathit{mult}(x, y))$$

Exponentiation function: $\exp : \mathbb{N}^2 \to \mathbb{N}$ - analogically - see exercise.
Predecessor function:

$$\begin{aligned} pred(0) &= \xi() \\ pred(y+1) &= \pi_1^2(y, pred(y)) \end{aligned}$$

Note: *pred* is a total function: $pred(0) = 0$
Monus function:

$$\begin{aligned} monus(x,0) &= \pi_1^1(x) \\ monus(x,y+1) &= pred(monus(x,y)) \end{aligned}$$

Meaning: $monus(x,y) = \begin{cases} x - y & \text{if } x \geq y \\ 0 & \text{otherwise} \end{cases}$
Notation: $monus(x,y) \equiv x \dot{-} y$
Eq function (equal): $eg(x,y) = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{if } x \neq y \end{cases}$

DEF

Definition 7.8.2 $eq(x,y) = 1 \dot{-} ((y \dot{-} x) + (x \dot{-} y))$ *or more formally* $eq \equiv monus \circ (\kappa_1^2 \times (plus \circ ((monus \circ (\pi_2^2 \times \pi_1^2)) \times monus \circ (\pi_1^2 \times \pi_2^2))))$

x+y

Example 7.8.5 $eq(5,3) = 1 \dot{-} ((3 \dot{-} 5) + (5 \dot{-} 3)) = 1 \dot{-} (0+2) = 1 \dot{-} 2 = 0$

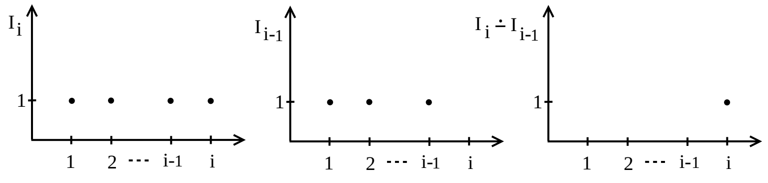
Function $\neg$ *eq*: $\neg eq \equiv monus \circ (\kappa_1^2 \times eq)$ $(\equiv 1 \dot{-} eq)$
Tabular function:
Consider functions of type:

$$f(x) = \begin{cases} 3 & \text{if } x = 0 \\ 5 & \text{if } x = 4 \\ 2 & \text{in other cases} \end{cases}$$

which tend to be defined by table. These functions can be formed using the characteristic function

$$\varphi_i(x) = \begin{cases} 1 & \text{if } x = i \\ 0 & \text{otherwise} \end{cases}$$

which can be expressed as $monus(I_i, I_{i-1})$, where $I_i(x) = eq(x \dot{-} i, 0)$



The tabular functions can be now formed by finite sum of multiples of constants and functions φ_i and $\neg\varphi_i$.

E.g.: The mentioned function $f(x)$ can be expressed in the form:

$$f \equiv mult(3, \varphi_0) + mult(5, \varphi_4) + mult(2, mult(\neg\varphi_0, \neg\varphi_4))$$

Quo function (quotient):

$$quo(x, y) = \begin{cases} \text{an integer part of division } x/y \text{ if } y \neq 0 \\ 0 \text{ if } y = 0 \end{cases}$$

This function can be defined by primitive recursion:

$$quo(0, y) = 0$$
$$quo(x + 1, y) = quo(x, y) + eq(x + 1, mult(quo(x, y), y) + y)$$

7.8.5 Functions outside of primitive recursive functions

There are functions which are computable and are not primitive recursive functions. All strictly partial functions (as *div*), and also some total functions have this property. An example of such total function was presented by W. Ackermann (1928) and it is called the *Ackermann function.* It is defined by the equations :

$$\boxed{\begin{array}{l} A(0, y) = y + 1 \\ A(x + 1, 0) = A(x, 1) \\ A(x + 1, y + 1) = A(x, A(x + 1, y)) \end{array}}$$

Theorem 7.8.2 *There exists a total function from* $\mathbb{N}$ *to* $\mathbb{N}$*, which is not primitive recursive.*

Poof. The definitions of functions, which are primitive recursive, can be perceived as strings and we can order them lexicographically and denote them as $f_1, f_2, \ldots, f_n, \ldots$

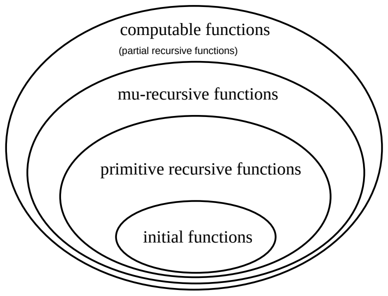
We define a function $f : \mathbb{N} \to \mathbb{N}$ so that $f(n) = f_n(n) + 1$ for $\forall n \in \mathbb{N} \setminus \{0\}$. Clearly, f is total and computable. But f is not primitive recursive (if it was, then $f \equiv f_m$ for some $m \in \mathbb{N}$. But then $f(m) = f_m(m)$ which is contradiction to the definition of the function f).

□

DEF

Definition 7.8.3 *The class of total computable functions is called* μ-recursive functions*.*

We get the following classification of functions:



7.8.6 Partial recursive functions

We will establish a technique known as *minimization* to extend the class of total computable functions. This technique allows to create function $f : \mathbb{N}^n \to \mathbb{N}$ from another function $g : \mathbb{N}^{n+1} \to \mathbb{N}$ by a formula, in which $f(\overline{x})$ is the smallest y such that:

1. $g(\overline{x}, y) = 0$
2. $g(\overline{x}, z)$ is defined for $\forall z < y,\ z \in \mathbb{N}$

We denote this construction as:

$$\boxed{f(\overline{x}) = \mu y[g(\overline{x}, y) = 0]}$$

x+y

Example 7.8.6 $f(x) = \mu y[plus(x, y) = 0]$ *i.e.* $f(x) = \begin{cases} 0 & pro\ \ x = 0 \\ undef. & otherwise \end{cases}$

Example 7.8.7 $div(x, y) = \mu t[((x + 1) \dot{-} (mult(t, y) + y)) = 0]$

Example 7.8.8 $i(x) = \mu y[monus(x, y) = 0]$ *i.e. identical function*

Indeed, the function defined by minimization is computable. A computation of the value $f(\overline{x})$ involves a computation of $g(\overline{x}, 0), g(\overline{x}, 1), \dots$ until we get:

- $g(\overline{x}, y) = 0$ $\quad (f(\overline{x}) = y)$,
- $g(\overline{x}, z)$ is undefined $\quad (f(\overline{x})$ is undefined).

DEF

Definition 7.8.4 The class of partial recursive functions *is a class of partial functions, which can be formed from initial functions by application of:*

- *combination,*
- *composition,*
- *primitive recursion and*
- *minimization.*

7.9 Relationship between the computable functions and the Turing machines

A Turing machine can be "modified" in a way, that it will be able to compute functions. The Turing machine will work as follows: starting in an initial configuration with input parameters of function on the tape, it writes the result of the computation (resulting function value) to the output, and only in this case it halts normally. We will show that recursive functions and Turing machines define the concept of computability of functions in the same way, i.e. functions computable by Turing machines are just recursive functions.

7.9.1 Turing-computable functions

Definition 7.9.1 *The Turing machine* $M = (Q, \Sigma, \Gamma, \delta, q_0, q_F)$ computes (enumerates) *a partial function* $f : \Sigma^{*m} \rightarrow \Sigma_1^{*n}$, $\Sigma_1 \subseteq \Gamma, \Delta \notin \Sigma_1$, *if for every* $(w_1, w_2, \ldots, w_m) \in \Sigma^{*m}$ *and corresponding initial configuration* $\underline{\Delta} w_1 \Delta w_2 \Delta \ldots \Delta w_m \Delta\Delta\Delta$ *of the machine* M *the following holds:*

1. *in the case that* $f(w_1, \ldots, w_m)$ *is defined,* M *halts and the tape contains* $\underline{\Delta} v_1 \Delta v_2 \Delta \ldots \Delta v_n \Delta\Delta\Delta$, *where* $(v_1, v_2, \ldots, v_n) = f(w_1, \ldots, w_m)$
2. *in the case that* $f(w_1, \ldots, w_m)$ *is not defined,* M *loops (never halts) or halts abnormally.*

The partial function, which can be computed by some Turing machine is called the *Turing-computable* function.

Example 7.9.1 *Turing machine computing the function* $f(w_1, w_2, w_3) = (w_1, w_3)$.

$$\begin{tikzcd} {} \arrow[r] & R_\Delta R_\Delta \arrow[r] & S_L L \arrow[loop right, "\neg\Delta"] \end{tikzcd}$$

Example 7.9.2 *Let* L *be any language.*

Function $f(w) = \begin{cases} |w| & \text{if } w \in L \\ 0 & \text{if } w \notin L \end{cases}$
is not Turing-computable.

Remark 7.9.1 *The definition of the computation of a function by a Turing machine didn't assume any special position of the head in the final configuration. Without loss of generality, we can assume that* M *halts in the configuration* $\underline{\Delta} v_1 \Delta \ldots \Delta v_n \Delta\Delta\Delta$.

7.9.2 Turing-computability of partial recursive functions

We will consider a Turing machine with alphabet $\Sigma = \{0, 1\}$ and partial functions in the form of $f : \mathbb{N}^m \to \mathbb{N}^n$, m-tuples and n-tuples will be encoded according to the following pattern:

$$\boxed{\Delta 11\Delta 10\Delta 100\Delta\Delta \quad \text{represents the triplet} \quad (3, 2, 4)}$$

Theorem 7.9.1 *Every partial recursive function is Turing-computable.*

Poof.

1. First, it is necessary to find Turing machines which compute the initial functions ξ, σ, π.

   (a) $\xi : \;\to R0L$ (b) (c) see exercise

2. Then we describe the construction of Turing machines for application of combination, composition, primitive recursion and minimization:

   (a) Combination: Let the Turing machine M_1 and M_2 computes the partial function g_1 and g_2 respectively. The machine M which computes $g_1 \times g_2$ will be a 3-tape Turing machine, which will begin with duplication of the input to the 2. and the 3. tape. Then M simulates M_1 using the 2. tape and M_2 using the 3. tape. If M_1 and M_2 halts properly, M erases the 1. tape and copies the content of the 2. and the 3. tape (separated by blank) to it and halts.

   (b) Composition: Function $g_1 \circ g_2$ is realized by machine $\to M_2 M_1$.

   (c) Primitive recursion: Consider:

   $$\begin{aligned} f(\overline{x}, 0) &= g(\overline{x}) \\ f(\overline{x}, y+1) &= h(\overline{x}, y, f(\overline{x}, y)) \end{aligned}$$

   where g is a partial function computed by a machine M_1 and h partial function, computed by a machine M_2. Function f is computed by a Turing machine which works in the following way:

      i. If the last item of the input is 0, then it will delete this item, returns the head back to the beginning and simulates the machine M_1.

      ii. If the last item of the input is not 0, then the tape must contain a sequence $\Delta\overline{x}\Delta y + 1\Delta\Delta\Delta$ for some $y + 1 > 0$. Then:

A. Using the machines for copying and decrementing, transform the content of the tape to the sequence:

$$\Delta\overline{x}\Delta y\Delta\overline{x}\Delta y-1\Delta\ldots\Delta\overline{x}\Delta 0\Delta\overline{x}\Delta\Delta,$$

then move the head right behind the 0 and simulate the machine M_1.

B. Now, the content of the tape is

$$\Delta\overline{x}\Delta y\Delta\overline{x}\Delta y-1\Delta\ldots\Delta\overline{x}\Delta 0\Delta g(\overline{x})\Delta\Delta,$$

which is equivalent to

$$\Delta\overline{x}\Delta y\Delta\overline{x}\Delta y-1\Delta\ldots\Delta\overline{x}\Delta 0\Delta f(\overline{x},0)\Delta\Delta.$$

Move the head before the last $\overline{x}$ and simulate the machine M_2. This leads to

$$\Delta\overline{x}\Delta y\Delta\overline{x}\Delta y-1\Delta\ldots\Delta\overline{x}\Delta 1\Delta h(\overline{x},0,f(\overline{x},0))\Delta\Delta,$$

which is equivalent to

$$\Delta\overline{x}\Delta y\Delta\overline{x}\Delta y-1\Delta\ldots\Delta\overline{x}\Delta 1\Delta f(\overline{x},1)\Delta\Delta.$$

C. Continue to apply the machine M_2 on the rest of the tape until M_2 is applied on $\Delta\overline{x}\Delta y\Delta f(\overline{x},y)$ and the tape is reduced to the form $\Delta h(\overline{x},y,f(\overline{x},y))\Delta\Delta$, which is equivalent to the required output $\Delta f(\overline{x},y+1)\Delta\Delta$.

(d) Minimization: We consider the function $\mu y[g(\overline{x},y)=0]$, where the partial function g is computed by M_1. We construct a 3-tape machine M, which works as follows:

i. It writes 0 on the 2. tape.
ii. It copies the content of the tape 1 and consequently of the tape 2 to the 3. tape.
iii. It simulates the machine M_1 on the 3. tape.
iv. If the 3. tape contains 0, it deletes the 1. tape, copies the 2. tape to the 1. tape and halts. Otherwise, it increments the content on the 2. tape, deletes the 3. tape and continues with step (2).

□

7.9.3 Representation of the Turing machine by partial recursive functions

To prove that Turing and Church thesis are equivalent, it remains to show that the computational power of Turing machines is restricted to computations of partial recursive functions. For this purpose, we consider the

Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, q_F)$ and set $b = |\Gamma|$. Now we will interpret the content of the tape as a positive number with the base b written reversly.

For instance: If $\Gamma = \{x, y, \Delta\}$ and if we interpret $x \approx 1, y \approx 2, \Delta \approx 0$, then the tape containing $\Delta yx\Delta\Delta y\Delta\Delta\ldots$ is equivalent to $02100200\ldots$, which after inversion represents the number $\ldots 00200120$ with the base 3 and thus 501.

With this interpretation we can understand the work of all Turing machines as a computation of the function $f : \mathbb{N} \to \mathbb{N}$, which for the given number corresponding to the initial content of the tape "assigns" the number corresponding to the content of the tape after halting of the machine. The nature of the function f is specified in the following theorem.

Theorem 7.9.2 *Every computation process performed by Turing machine is a computation process of some partial recursive function.*

Poof.

We will be working on the previous interpretation of work of a Turing machine M, i.e. for all $n \in \mathbb{N}$, $f(n)$ is defined by the content of the tape at the moment when the machine M halts. Next, we perform encoding of the states: $0 \approx q_0, 1 \approx q_F$, the rest of states is encoded by numbers $2, 3, \ldots, k-1$, assuming that $|Q| = k$. Now, both the states and the symbols have assigned numeric values and we can define functions, which summarize the transition diagram of the machine M' (M' is encoded machine M):

$$mov(p,x) = \begin{cases} 2 & \text{if} \ \ \delta(p,x) = (*, R) \\ 1 & \text{if} \ \ \delta(p,x) = (*, L) \\ 0 & \text{otherwise} \end{cases}$$

$$sym(p,x) = \begin{cases} y & \text{if} \ \ \delta(p,x) = (*, y) \\ x & \text{otherwise} \end{cases}$$

$$state(p,x) = \begin{cases} q & \text{if} \ \ \delta(p,x) = (q, *) \\ k & \text{if} \ \ p = 0 \ \text{ or } \ \delta(p,x) \ \text{ is undefined} \end{cases}$$

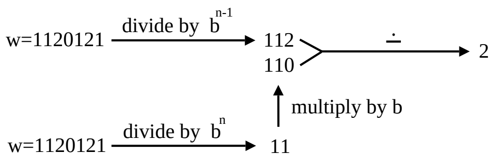
Function *sym*, *mov* and *state* are tabular functions (total) and thus they are primitive recursive.

Now we consider a configuration of Turing machine M' in the form of a triplet (w, p, n), where w is the content of the tape, p is the present state, n is the position of the head ($n \geq 1$,the leftmost position is equal to 1).

From the configuration, we can compute a symbol which is under the head using primitive recursive function *cursym*:

$$\boxed{cursym(w,p,n) = quo(w, b^{n-1}) \dot{-} mult(b, quo(w, b^n))}$$

Example 7.9.3 *Finding the n-th numeral of the string* $w = 1120121$ *for* $n = 5$ *and* $b = 3$*:*



$$112 = quo(w, b^{n-1})$$
$$110 = mult(b, quo(w, b^n))$$

Further functions that we define over the set of configurations are:

$$\boxed{\begin{aligned} nexthead(w,p,n) \quad = \quad & n \dot{-} eq(mov(p, cursym(w,p,n)), 1) \\ & + eq(mov(p, cursym(w,p,n), 2)) \end{aligned}}$$

determining the next position of the head (0 indicates abnormal move from 1. position on the left)

$$\boxed{nextstate(w,p,n) = state(p, cursym(w,p,n)) + mult(k, \neg nexthead(w,p,n))}$$

(normally, the 2. summand is equal to 0; this function will compute an illegal state bigger than $k-1$ for an abnormal move) and finally the function

$$\boxed{\begin{aligned} nexttape(w,p,n) \quad = \quad & (w \dot{-} mult(b^n, cursym(w,p,n))) \\ & + mult(b^n, sym(p, cursym(w,p,n))) \end{aligned}}$$

which computes an integer representing a new content of the tape, after making a transition from the configuration (w,p,n). By combination of the three previous functions we obtain a function *step*, which models 1. step of Turing machine, i.e. the transition to the new configuration:

$$\boxed{step = nexttape \times nextstate \times nexthead}$$

Now we define the function $run : \mathbb{N}^4 \to \mathbb{N}^3$; $run(w,p,n,t)$ performing t transitions from the configuration (w,p,n). Again we use primitive recursion:

$$\begin{aligned} & run(w,p,n,0) = (w,p,n) \\ & run(w,p,n,t+1) = step(run(w,p,n,t)) \end{aligned}$$

Resulting function value computed by the machine M' (on input w) is the value of the tape after reaching the terminal state (state 0). Number of required steps of the machine M is given by the function *stoptime*

$$stoptime(w) = \mu t[\pi_2^3(run(w,1,1,t)) = 0]$$

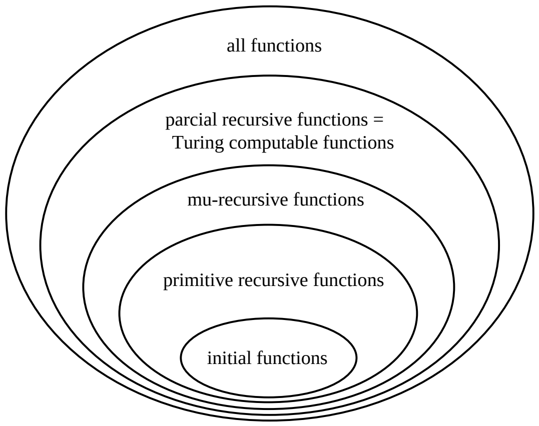
In conclusion, if $f : \mathbb{N} \to \mathbb{N}$ is a partial function computed by the machine M, then

$$\boxed{f(w) = \pi_1^3(run(w,1,1,stoptime(w)))}$$

It follows from its construction that f is a partial recursive function.

□

Lets summarize the obtained information in the figure:


all functions
parcial recursive functions = Turing computable functions
mu-recursive functions
primitive recursive functions
initial functions

There exist uncountably many formal languages, that cannot be described by Turing machines and therefore they are out of the class of recursively enumerable languages. The strictly undecidable languages and complements of partial decidable languages belong to these languages. Many of them define problems, which we can not solve algorithmically. The general and often used representation of problems is the halting problem of Turing machine and Post correspondence problem which are used for reductions.

Among a number of alternative formal systems equivalent to Turing machines, partial recursive functions, which have very close relation with functional programming, play an important role in description and analysis.

7.10 Exercise

Exercise 7.10.1 Define when a problem is decidable, undecidable and partially decidable.

Exercise 7.10.2 Give examples the problems which are decidable, partially decidable and not partially decidable.

Exercise 7.10.3 Define the Post correspondence problem and decide if it is decidable or at least partial decidable. Give an example of the Post system which has a solution and the system which does not have a solution.

Exercise 7.10.4 Define the concept of reduction and explain its usage on an example.

Exercise 7.10.5 Define the 1. and the 2. Rice theorem and explain their potential usage.

Exercise 7.10.6 Give examples of other computational models whose computational power is equivalent to Turing machines.

Exercise 7.10.7 Define the initial function, primitive recursive function and partial computable function.

Exercise 7.10.8 Does there exist any primitive recursive function which is not total? Explain your answer.

Exercise 7.10.9 Give examples of functions created by combination, composition, primitive recursion and minimization.

Exercise 7.10.10 Explain the relationship between partial computable functions and Turing machines.

Chapter 8

Complexity

8:00

The aim of this chapter is the comprehension of applications of Turing machines for description and classification of time and space complexity of computational problems. Also, standard complexity classes are introduced. These classes are closely related to matters of time and memory efficiency of computations and therefore reveal a lot about feasibility of these computations on real computers.

8.1 Basic notions of complexity

8.1.1 Algorithm complexity

The basic theoretical approach and the choice of Turing machine as a computing model follows from Church-Turing thesis:

Every algorithm can be implemented by a certain TM.

Turing machines allow us to classify problems (or functions) into two classes:

1. problems which are *algorithmically not even partially decidable* (or functions algorithmically uncomputable)

2. problems *algorithmically at least partially decidable* (or functions algorithmically computable).

Now, we will deal with the class of algorithmically (partially) decidable problems (computable functions) and with the question of their computational *complexity*.

We will understand the analysis of algorithm complexity as the analysis of complexity of computations of a corresponding TM. The aim of the analysis is to *express (quantify) the required resources (time, space) as a function depending on the length of an input string.*

8.1.2 Various cases of complexity analysis

First, it is necessary to determine the price of one certain computation of a certain TM (memory, computation time).

Let M be a TM. Thus, the complexity will be the function $Compl_M : \mathbb{N} \to \mathbb{N}$[1]. There are more ways how to define this function with respect to

[1]for given n, the function returns a measure of the computation complexity of M on strings of length n

the choice of potential computations on the input of respective length. We can focus on :

1. the analysis of *the complexity of the worst case*,

2. the analysis of *the complexity of the best case*,

3. the analysis of *the complexity of the average case*.

The average complexity of algorithm is defined as follows: If the algorithm (TM) leads to m various computations (cases) with the complexities c_1, c_2, ..., c_m, which occur with the probabilities p_1, p_2, ..., p_m, then *the average complexity of algorithm* is given as $\Sigma_{i=1}^{n} p_i c_i$.

Usually (at least at theoretical level) we pay *the most attention* to the complexity of the worst case.

8.1.3 Complexity of TM computations

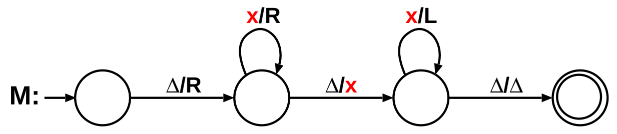
We will look at the computational complexity from two acpects – the computation time aspect and the aspect of memory requirements.

- *Time complexity* – the number of steps (transitions) performed by a TM from the beginning till the end of the computation.

- *Space (memory) complexity* – the number of "cells" of the tape of a TM which are required for a given computation.

$\boxed{\texttt{x+y}}$

Example 8.1.1 *Consider the following TM M:*



For the input $w = \Delta xxx\Delta\Delta...$:

- *the time complexity of the computation of M on w is equal to* 10,

- *the space complexity of M on w is equal to* 5.

Lemma 8.1.1 *If the* time complexity *of an computation performed by TM is equal to n, then the* space complexity *of this computation is not greater than $n + 1$.*

Poof. The statement is a simple implication following from the definition of time and space complexity. □

Now we will define functions specifying the complexity of TM depending on the computation time (the number of steps of a TM) and used space (the number of used cells of tape). We will consider the worst case analysis:

Definition 8.1.1 *We say that* 1-tape DTM (or NTM) M accepts the language L over the alphabet Σ in time $T_M : \mathbb{N} \to \mathbb{N}$, *if* $L = L(M)$ *and* M *accepts (or* can *accept) all* $w \in L$ *in at most* $T_M(|w|)$ *steps.*

DEF

Definition 8.1.2 *We say that* 1-tape DTM (or NTM) M accepts the language L over the alphabet Σ in the space $S_M : \mathbb{N} \to \mathbb{N}$, *if* $L = L(M)$ *and* M *accepts (or* can *accept) all* $w \in L$ *while using at most* $S_M(|w|)$ *cells of tape.*

Remark 8.1.1 *We don't account the cells of tape, on which input is written, to the space complexity, unless they are rewritten during the computation.*

Analogically, we can define a *computation of a certain function by given TM* in certain time, or space.

8.1.4 Complexity of atomic operations

In case of TM we assume that every computation step is equally demanding. We use the so-called *uniform price criteria*, where we assign the *same complexity* to each operation.

It can be different in computing systems other than TM. But we also use e.g. the so-called *logarithmic price criteria*, where we assign the price $\lfloor lgß \rfloor + 1$ to the operation, which manipulates with operand of size i, $i > 0$.

By this, we take into account the fact that the complexity of the operations is growing due to growing size of operands – the logarithm reflects growing size with respect to the binary encoding (we encode 2^n values in n bits)

The complexity analysis with such assumptions is not really precise. But usually it is important to *keep the information about how quickly the time/space required for computation is growing, depending on the length of input.* [2]

*T*he following example illustrates the importance of comparisons of the rate of growth of computational complexity:

x+y

Example 8.1.2 Comparison of the polynomial (more precisely quadratic – n^2) and exponential (2^n) time complexity*:*

[2]Algorithm A_1 whose demands grow less than demands of other algorithm A_2, does not have to be more suitable than A_2 for solving smaller instances.

length of input n	*time complexity* n^2	*time complexity* 2^n
10	*0.0001 s*	*0.0001 s*
20	*0.0004 s*	*0.1024 s*
30	*0.0009 s*	*1.75 min*
40	*0.0016 s*	*1.24 day*
50	*0.0025 s*	*3.48 year*
60	*0.0036 s*	*35.68 century*
70	*0.0049 s*	*3.65 mil. years*

8.1.5 Complexity of computations on TM and in other systems

It appears that for reasonable complexity criteria the computation complexity for various computation models, which are close to common computers (RAM, RASP machines), is *polynomially bounded* with the complexity of computations on TM (see further). Hence, the *complexity of computation on TM is not "too much" different from computations on common computers.*

- *RAM machines* have the memory with random access (one cell contains any natural number). Instruction such as LOAD, STORE, ADD, SUB, MULT, DIV (accumulator and constant/direct address/non-direct address), input/output, unconditional jump and conditional jump (zero test of accumulator), HALT. In RAM machine the program is part of control of the machine (instantly available), in *RASP* it is saved in the memory, with the operands as well.

- Functions $f_1(n), f_2(n) : \mathbb{N} \to \mathbb{N}$ are *polynomially bounded*, if there exist polynomials $p_1(x)$ and $p_2(x)$ such that $f_1(n) \leq p_1(f_2(n))$ and $f_2(n) \leq p_2(f_1(n))$ for all values n.

- *Logarithmic price criterion* can be used, when we consider a multiplication of two numbers. We are able to compute 2^{2^n} by using a simple cycle of type $A_{i+1} = A_i * A_i$ $(A_0 = 2)$, which can't be performed by TM with restricted alphabet in polynomial time (we need to access 2^n cells for saving/loading when using binary encoding).

Now we illustrate how to determine the complexity outside of the TM model:

$\boxed{\texttt{x+y}}$

Example 8.1.3 *Consider the following implementation of* comparison of two strings.

```
int str_cmp (int n, string a, string b) {
    int i;

    i = 0;
    while (i<n) {
      if (a[i] != b[i]) break;
      i++;
    }

    return (i==n);
}
```

- *In order to determine the complexity, we apply the* uniform price criterion. *For instance in a way that we consider the complexity of each row of the program (without declarations) as one unit. (We assume that no cycle is written in a single row)*
- The worst case complexity can be determined easily as $4n+3$*:*
  - *the cycle has* 4 *steps, it is performed* n *times, i.e.* $4n$ *steps,*
  - *the body of the function has* 3 *steps (including the cycle termination test).*

x+y

Example 8.1.4 *Consider the following implementation of the* insert-sort sorting method.

```
void insertsort(int n, int a[]) {
    int i, j, value;
    for (i=0; i<n; i++) {
        value = a[i];
        j = i - 1;
        while ((j >= 0) && (a[j] > value)) {
                a[j+1] = a[j];
                j = j - 1;
        }
        a[j+1] = value;
    }
}
```

- *In order to determine the complexity, we apply the* uniform price criterion. *For instance in a way that we consider the complexity of each row of the program (without declarations) as one unit. (We assume that no cycle is written in a single row)*
- The complexity can be determined easily as $2n^2+4n+1$*:*
  - *the inner cycle has* 4 *steps, it is performed* $0, 1, \ldots, n-1$ *times, i.e.* $4(0+1+\ldots+n-1) = 4\frac{n}{2}(0+n-1) = 2n^2-2n$ *steps,*
  - *the outer cycle has (excluding the inner cycle)* 6 *steps (including the inner cycle termination test) and it is performed* n *times, i.e.* $6n$ *steps,*
  - *termination of the outer cycle takes one step.*

8.1.6 Asymptotic restrictions of complexity

Usually, the exact information about algorithm complexity is useless for us. It says a little about the real time (space) of the computation:

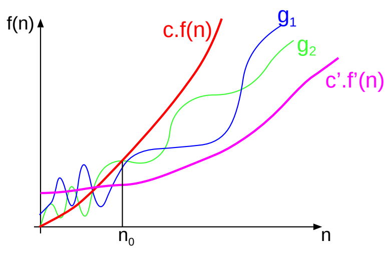
- by using "minor" modifications of considered algorithms, various additive and multiplicative constants arise very easily,
- e.g. a comparison of two strings can be accelerated till infinity in a way, that we will compare simultaneously 2, 3, 4, ... subsequent characters,
- but these modifications do not influence the speed growth of the time complexity considerably (the growth remains quadratic in the case mentioned above),
- moreover, we commit some inaccuracy by establishing various complexity criteria, when we analyze the complexity outside of the TM model.

We are interested just in the "important" part of information about complexity (comparison of strings – the time grows linearly, insert-sort – the time grows quadratically). Therefore, the complexity is described by the use of the so-called *asymptotic estimates of complexity*:

DEF

Definition 8.1.3 *Let $\mathcal{F}$ be a set of functions $f : \mathbb{N} \to \mathbb{N}$. For a given function $f \in \mathcal{F}$ we define sets of functions $O(f(n))$, $\Omega(f(n))$ and $\Theta(f(n))$ as follows:*

- Asymptotic upper bound *for the function $f(n)$ is the set $O(f(n)) = \{g(n) \in \mathcal{F} \mid \exists c, n_0 \in \mathbb{N}\ \forall n \in \mathbb{N} : n \geq n_0 \Rightarrow 0 \leq g(n) \leq c.f(n)\}$.*
- Asymptotic lower bound *for the function $f(n)$ is the set $\Omega(f(n)) = \{g(n) \in \mathcal{F} \mid \exists c, n_0 \in \mathbb{N}\ \forall n \in \mathbb{N} : n \geq n_0 \Rightarrow 0 \leq c.f(n) \leq g(n)\}$.*
- Asymptotic tight bound *for the function $f(n)$ is the set $\Theta(f(n)) = \{g(n) \in \mathcal{F} \mid \exists c_1, c_2, n_0 \in \mathbb{N}\ \forall n \in \mathbb{N} : n \geq n_0 \Rightarrow 0 \leq c_1.f(n) \leq g(n) \leq c_2.f(n)\}$.*



Example 8.1.5 *When using the asymptotic estimates of complexity, we can say that the complexity of our strings comparison is in $O(n)$ and the complexity of insert-sort is in $O(n^2)$.*

Example 8.1.6

- *We show that $2 \cdot n \in O(n)$ holds.*
  According to the definition we choose $c = 3$ and $n_0 = 1$, then it really holds that $\forall n \geq 1$ is $2 \cdot n \leq 3 \cdot n$
- *We show that $n^2 \notin O(n)$ holds.*
  We will perform a proof by contradiction. According to the definition, $\exists c, n_0 \in \mathbb{N}$ such that $\forall n \geq n_0$ holds that $n^2 \leq c \cdot n$. But then it also holds that $\forall n \geq n_0$ is $n \leq c$, which is a contradiction.

Example 8.1.7 *Decide if the following holds and explain why:*

- $n \cdot \log(n) \in O(n^2)$
- $n \cdot \log(n) \in O(n)$
- $3^n \in 2^{O(n)}$
- $6 \cdot n^3 + 50 \cdot n^2 + 6 \in O(n^3 - 8 \cdot n^2 - n - 5)$

8.2 Complexity classes

8.2.1 Complexity of problems

Now we move from the complexity of certain algorithms (Turing machines) to the *complexity of problems*.

We establish *complexity classes* as an *instrument for classification (creating the hierarchy) of problems according to their complexity*, i.e according to how efficient algorithms for this problems we know (analogically, we can talk about complexity of computation of functions). Similarly to finding the type of language, we will try to *place the problem into the complexity class as low as possible* – i.e. define the problem complexity as the complexity of its best solution.

Remark 8.2.1

As we will see, there exist problems whose solutions can be significantly *accelerated till infinity, which brings some restrictions to our effort.*

Placing *various problems into the same complexity class* can reveal some inner similarities of these problems and can allow getting solution of one problem by a transformation of that problem to another one (and use of e.g. various already developed instruments).

8.2.2 Complexity classes

DEF

Definition 8.2.1 *Lets have given the functions $t, s : \mathbb{N} \to \mathbb{N}$ and let T_M, or S_M, defines the time, or space, complexity of TM M. We define the following* time and space complexity classes of deterministic and non-deterministic TM*:*

- $DTime[t(n)] = \{L \mid \exists$ *1-tape DTM* $M : L = L(M)$ *a* $T_M \in O(t(n))\}$.
- $NTime[t(n)] = \{L \mid \exists$ *1-tape NTM* $M : L = L(M)$ *a* $T_M \in O(t(n))\}$.
- $DSpace[s(n)] = \{L \mid \exists$ *1-tape DTM* $M : L = L(M)$ *a* $S_M \in O(s(n))\}$.
- $NSpace[s(n)] = \{L \mid \exists$ *1-tape NTM* $M : L = L(M)$ *a* $S_M \in O(s(n))\}$.

We generalize the definition of complexity classes in the way so they can be based on the set of functions, rather than just on one single function.

Remark 8.2.2 *: Further we show that although non-determinism brings nothing important with respect to the computability, it can bring a lot with respect to the complexity.*

8.2.3 Time/space constructible functions

We usually construct the complexity classes over the so-called *time/space constructible functions*:

- The reason is to reach *intuitive hierarchic structure* of complexity classes – e.g. distinguishing of the classes $f(n)$ and $2^{f(n)}$, which (as we will see) is impossible for the classes based on general recursive functions.

DEF

Definition 8.2.2 *We call the function $t : \mathbb{N} \to \mathbb{N}$* time constructible*, if there exists a multi-tape TM, which for any input w halts after precisely $t(|w|)$ steps.*

Definition 8.2.3 *We call the function $s : \mathbb{N} \to \mathbb{N}$* space constructible*, if there exists a multi-tape TM, which for any input w halts with precisely $s(|w|)$ used cells of the tape.*

x+y

Example 8.2.1 *We show some*

- *time constructible functions:* $f(n) = n\log(n),\ f(n) = n\sqrt{n}$
- *time inconstructible functions:* $f(n) = c,\ f(n) = n$
- *space constructible functions:* $f(n) = log(n),\ f(n) = n^2$
- *space inconstructible functions:* $f(n) = c,\ f(n) = \log\log n$

Remark 8.2.3 If the language L over Σ is accepted by a machine in time/space complexity bounded by time/space constructible function, then it is also accepted by a machine which always halts for all $w \in \Sigma^*$*:*

- *For a time bound $t(n)$, we need to compute the number of necessary steps in advance and halt after using all of them.*
- *For a space bound, we compute the maximum number of configurations we can see (this can be computed from $s(n)$, $|Q|$, and $|\Delta|$). The maximum number of steps we can perform without looping follows from this.*

8.2.4 Most commonly used complexity classes

Deterministic/non-deterministic polynomial time:

$$\mathbf{P} = \bigcup_{k=0}^{\infty} DTime(n^k) \qquad \mathbf{NP} = \bigcup_{k=0}^{\infty} NTime(n^k)$$

Deterministic/non-deterministic polynomial space:

$$\mathbf{PSPACE} = \bigcup_{k=0}^{\infty} DSpace(n^k) \qquad \equiv \qquad \mathbf{NPSPACE} = \bigcup_{k=0}^{\infty} NSpace(n^k)$$

Remark 8.2.4 *:* **P** *is a particularly important class. It defines all practically well solvable problems.*

DEF

Remark 8.2.5 *: Problems from the class* **PSPACE** *are usually not solved in polynomial space – the space demands are increasing in exchange for at least partial decreasing of the time demands (e.g. from $O(2^{n^2})$ to $O(2^n)$).*

8.2.5 Classes under and above the polynomial complexity

Deterministic/non-deterministic logarithmic space:

$$\mathbf{LOGSPACE} = \bigcup_{k=0}^{\infty} DSpace(k\ lg\ n)$$

$$\mathbf{NLOGSPACE} = \bigcup_{k=0}^{\infty} NSpace(k\ lg\ n)$$

Deterministic/non-deterministic exponential time:

$$\mathbf{EXP} = \bigcup_{k=0}^{\infty} DTime(2^{n^k}) \qquad \mathbf{NEXP} = \bigcup_{k=0}^{\infty} NTime(2^{n^k})$$

Deterministic/non-deterministic exponential space:

$$\textbf{EXPSPACE} = \bigcup_{k=0}^{\infty} DSpace(2^{n^k}) \;\equiv\; \textbf{NEXPSPACE} = \bigcup_{k=0}^{\infty} NSpace(2^{n^k})$$

8.2.6 Classes over exponential complexity

DEF

Det./non-det. k-exponential time/space based on the tower of the exponentials $2^{2^{\vdots^{2}}}$ of the height k:

$$\textbf{k-EXP} = \bigcup_{l=0}^{\infty} DTime(2^{2^{\vdots^{2^{n^l}}}}) \qquad \textbf{k-NEXP} = \bigcup_{l=0}^{\infty} NTime(2^{2^{\vdots^{2^{n^l}}}})$$

$$\textbf{k-EXPSPACE} = \bigcup_{l=0}^{\infty} DSpace(2^{2^{\vdots^{2^{n^l}}}})$$

$$\textbf{k-NEXPSPACE} = \bigcup_{l=0}^{\infty} NSpace(2^{2^{\vdots^{2^{n^l}}}})$$

$$\textbf{k-EXPSPACE} \quad \equiv \quad \textbf{k-NEXPSPACE}$$

$$\textbf{ELEMENTARY} = \bigcup_{k=0}^{\infty} \textbf{k-EXP}$$

8.2.7 Top of hierarchy of complexity classes

On the *top of the hierarchy of complexity classes* are general classes of languages (functions) that we have already seen:

- *the class of primitive recursive functions* **PR** (implementable using nested cycles with fixed number of repeating – `for i=... to ...`),
- *the class of $\mu-$recursive functions* (recursive languages) **R** (implementable using cycles with not fixed number of repeating – `while ...`) a
- the class of *recursively computable functions* (recursively enumerable languages) **RE**.

8.3 Properties of complexity classes

We will now investigate properties of the defined complexity classes and relations between them.

8.3.1 Multi-tape machines

An introduction of multi-tape machines didn't influence the computability. Similarly, it won't bring about many novelties in the complexity theory. We will show that for each multi-tape machine there is an equivalent single-tape machine whose computations run in time which is at most polynomially higher.

Theorem 8.3.1 If the language L is accepted by some k-tape DTM M_k in time $t(n)$, then it is also accepted by some single-tape DTM M_1 in time $O(t(n)^2)$.

Poof. (idea) We will show how to simulate M_k by M_1 in time stated above:

- We write the concatenation of the content of all tapes of machine M_k on the tape of machine M_1. Next, the machine M_1 has to keep the information about actual position of heads of the machine M_k (e.g. symbols on the tape read by heads of the machine M_k will be underlined) and the information about the end of each tape of machine M_k (special separators).
- In the first phase of simulation of computation of the machine M_k by the machine M_1, the machine M_1 modifies its tape to required form.
- The machine M_1 has to go through its tape twice, in order to simulate one step of the machine M_k. In the first way through, it collects information about symbols which have been read by each head of the machine M_k. In the second way through, it modifies its tape accordingly.
- Problem occurs if the machine M_k writes any symbol to so far empty cell of its tape during simulation. In that case, the machine M_1 has to move the rest of the string on the tape by one cell to the right, in order to free required space for the new symbol.
- The machine M_k cannot have more than $t(n)$ symbols on any tape, where n is the length of input string. Therefore, the machine M_1 cannot have more than $k \cdot t(n)$ symbols. Simulation of one step of the machine M_k takes $4 \cdot k \cdot (t(n))$ steps (going twice through the tape and back) plus at most $k \cdot (t(n))$ steps (creating of an empty cell). Altogether, the number of steps of the machine M_1 is in $O(k^2 \cdot t(n)^2)$. Because k is the constant independent of input, the final time complexity of the machine M_1 is in $O(t(n)^2)$.

$\square$

Example 8.3.1 *Show that $L = \{w \in \Sigma^* |$ w is a palindrome $\} \in P$.*

8.3.2 Determinism and non-determinism

While non-determinism brings nothing new with respect to computability, it is quite different when considering complexity issues. Though the relations between deterministic and non-deterministic classes haven't been established yet, it seems that non-determinism strongly lowers the time demands of computation. Its power is well illustrated in the following point on computation of non-deterministic machine:

- While classical deterministic machine computes the result step by step, the non-deterministic machine can simply guess the result, write it down and then just check its correctness.

We can simulate any non-deterministic TM by a deterministic machine, however it results in the exponential growth in time:

Theorem 8.3.2 If the language L is accepted by any NTM M_n in time $t(n)$, then it is also accepted by any DTM M_d in time $2^{O(t(n))}$.

Poof. (idea) We will show, how M_d can simulate M_n in time mentioned above:

- We number the transitions of M_n as $1, 2, ..., k$.
- *M_d will subsequently simulate all possible sequences of transitions of M_n* (it will save the content of the input tape to the additional tape, in order to be always able to reload it; it will generate the sequence of transitions from $\{1, 2, ..., k\}^*$ to another tape and it will simulate this).
- In view of the the *possibility of infinite computations of M_n* we cannot traverse its possible computations in a depth-first manner – but if we will *traverse them in the breadth-first manner* (i.e. first all the strings from $\{1, 2, ..., k\}^*$ of the length 1, then 2, then 3, ...), we will surely find the shortest accepting sequence of transitions for M_n, if such exists.
- In this way, we traverse at most $O(k^{t(n)})$ paths, the simulation of each of them is in $O(t(n))$. Thus, in total we use at most $O(k^{t(n)})O(t(n)) = 2^{O(t(n))}$ time.

□

Example 8.3.2 *Show that the language $L = \{\phi | \ \phi$ is a satisfiable formula in CNF $\} \in NP$*

The relation between determinism and non-determinism is one of the most famous open problems in computer science. A lot has been said particularly about the problem of equivalence of polynomial time classes $P = NP$. However, its solution seems to be beyond today's capabilities. To show how complicated this problem is, consider for instance results which say that if P=NP, then this relation can't be proved by simulation [3], and also that if P≠NP, this solution cannot be proved by diagonalization.

It seems that non-determinism brings a great advantage with respect to time complexity of computations, but the situation is different with respect to space complexity:

Theorem 8.3.3 *(Savitch's theorem)* $NSpace[s(n)] \subseteq DSpace[s^2(n)]$ for all space constructible function $s(n) \geq lg\ n$.

Poof. Consider the NTM $M = (Q, \Sigma, \Gamma, \delta, q_0, q_F)$ deciding $L(M)$ in the space $s(n)$:

- There exists $k \in \mathbb{N}$ dependent only on $|Q|$ and $|\Gamma|$, such that for any input w, M traverses at most $k^{s(n)}$ configurations of length max. $s(n)$.
- This implies that M *will perform at most* $k^{s(n)} = 2^{s(n)lg\ k}$ *steps* for given w.
- Using DTM, we can easily implement the procedure $test(c, c', i)$ which *tests if M can get from the configuration c to c' in 2^i steps*:

```
**procedure** $test(c, c', i)$
**if** $(i = 0)$ **then return** $((c = c') \vee (c \underset{M}{\vdash} c'))$
**else for** all configurations $c''$ such that $|c''| \leq s(n)$ **do**
    **if** $(test(c, c'', i-1) \wedge test(c'', c', i-1))$ **then return** $true$
**return** $false$
```

- Notice that *by recursive invocations of the test, a tree of height i arises. The tree simulates a sequence of 2^i computation steps by sequences of its leaves.*
- Now, *for deterministic simulation of M* it suffices to traverse all accepting configurations c_F, such that $|c_F| \leq s(n)$, and to check if $test(c_o, c_f, \lceil s(n)lg\ k\rceil)$, where c_0 is the initial configuration.
- Each invocation of *test* takes $O(s(n))$ space, the depth of recursion is $\lceil s(n)lg\ k\rceil = O(s(n))$ and thus *we simulate M deterministic in the space* $O(s^2(n))$ in total.

[3]there cannot exist a polynomial-time bounded DTM simulating every polynomial NTM

- Let us add that $s(n)$ can be constructed in the space $O(s(n))$ (i.e. we are dealing with a space constructible function) and thus it does not influence the ideas mentioned above.

$\square$

Consequences of Savitch's theorem are the following inclusions (they have been already mentioned):

- **PSPACE** $\equiv$ **NPSPACE**,
- **k-EXPSPACE** $\equiv$ **k-NEXPSPACE**.

8.3.3 Space vs time

Intuitively, we can say that *while space can grow relatively slowly, the time can grow much faster*, because we can repeatedly traverse the same cells of tape – obviously it can't be the contrary (there is no reason to keep the unused space).

Theorem 8.3.4 *$NSpace[t(n)] \subseteq DTime[O(1)^{t(n)}]$* for all time constructible function *$t(n) \geq lg\ n$.*

Poof. We can use a construction which is quite similar to one used in Savitch's theorem – for more details, see the literature. $\square$

8.3.4 Closeness under complement

As the complement to the class we understand the class of languages which are complements to the languages of the given class. Therefore, if we mark the complement to the class $\mathcal{C}$ as *co*-$\mathcal{C}$, then $L \in \mathcal{C} \Leftrightarrow \overline{L} \in co\text{-}\mathcal{C}$. When deciding computational problems, this means deciding the complementary problem (emptiness x non-emptiness etc.).

Space classes are usually closed under complement:

Theorem 8.3.5 If $s(n) \geq lg\ n$, then $NSpace(s(n)) = co\text{-}NSpace(s(n))$.

Poof. The above statement is the Immerman–Szelepcsényi theorem – see the proof in literature. $\square$

The situation is different for *time classes*:

- Some classes such as ***P*** *or* ***EXP*** *are closed under a complement.*
- *The question of closeness under the complement remains opened for other important classes.* That's why it is reasonable to speak also about classes as:

– ***co-NP*** or
– ***co-NEXP***.

Theorem 8.3.6 Class **P** is closed under the complement.

Poof. (idea) The base is to show that if the language L over Σ can be accepted by a DTM M in polynomial time, then there also exists a DTM M' which decides L in polynomial time, i.e. $L = L(M')$ and there exists $k \in \mathbb{N}$, such that for all $w \in \Sigma^*$, M' halts in time $O(|w|^k)$:

- At the beginning of the computation, M' will determine the length of the input w and computes $p(|w|)$, where $p(n)$ is a polynomial defining complexity of acceptance by the machine M. It will save $p(|w|)$ symbols on the special additional tape.
- Subsequently, M' simulates M, and during every step, it deletes one symbol from the additional tape. If it deletes all symbols from this tape and M would not accept meanwhile, then M' halts the computation abnormally (refuses).
- Evidently, M' accepts all strings which are accepted by M. For that, $p(n)$ simulation steps are sufficient and it won't accept all strings, which M would not accept – if M doesn't accept in $p(n)$ steps, it won't accept at all. But M' will halt always in $O(p(n))$ steps.

□

8.3.5 Strictness of class hierarchy

From foregoing facts, we can summarize that the following holds:

- $\mathbf{LOGSPACE} \subseteq \mathbf{NLOGSPACE} \subseteq \mathbf{P} \subseteq \mathbf{NP}$
- $\mathbf{NP} \subseteq \mathbf{PSPACE} = \mathbf{NPSPACE} \subseteq \mathbf{EXP} \subseteq \mathbf{NEXP}$
- $\mathbf{NEXP} \subseteq \mathbf{EXPSPACE} = \mathbf{NEXPSPACE} \subseteq \mathbf{2\text{-}EXP} \subseteq \mathbf{2\text{-}NEXP}$
- $... \subset \mathbf{ELEMENTARY} \subset \mathbf{PR} \subset \mathbf{R} \subset \mathbf{RE}$[4]

Some questions of strictness of inclusions mentioned above remain open. However, from the so-called hierarchy theorem (we won't mention proper definition, because it is very technical – it can be found in literature) follows that *exponential "gaps" between classes are "strict"*:

- $\mathbf{LOGSPACE},\ \mathbf{NLOGSPACE} \subset \mathbf{PSPACE}$,
- $\mathbf{P} \subset \mathbf{EXP}$,

[4]We add without proof, that $\mathbf{ELEMENTARY} \subset \mathbf{PR}$.

- **NP** $\subset$ **NEXP**,
- **PSPACE** $\subset$ **NEXPSPACE**,
- **EXP** $\subset$ **2-EXP**, ...

8.3.6 Some other interesting results

Theorem 8.3.7 *(Blum's theorem)* For every total computable function $f : \mathbb{N} \to \mathbb{N}$, there exists a problem, whose each solution with some complexity $t(n)$ can be improved in a way, that the new solution has the complexity $f(t(n))$ for almost all $n \in \mathbb{N}$.

Poof. see literature. □

Due to Blum's theorem there exist problems whose solutions with complexity $t(n)$ can be accelerated infinitely to $lg\ t(n)$, $lg\ lg\ t(n)$, $lg\ lg\ lg\ t(n)$, ...

The necessity to work with *time constructible functions*, in order to avoid e.g. the $DTime[f(n)] = DTime[2^{f(n)}]$ for some $f(n)$, is expresses by the following theorem:

Theorem 8.3.8 *(Gap Theorem)* For each recursive function $\phi(n) > n$, there exists a recursive function $f(n)$, such that:

$$DTime[\phi(f(n))] = DTime[f(n)].$$

Poof. See literature – the proof is based on construction of a function f, such that no TM halts on input of the length n with number of steps between $f(n)$ and $\phi(f(n))$. □

8.3.7 $\mathcal{R}$ reduction, languages $\mathcal{C}$-hard a $\mathcal{C}$-complete

So far, we have used classes as an upper bound of problems complexity. Now notice the *lower bound – we introduce this through the concept of reducibility of a class of problems to given problem.*

$\mathcal{R}$ reduction is (similarly as in the chapter dealing with computability) a function which transforms a problem to another problem. But in the complexity theory, we require this function to satisfy further requirements, not only the mere computability:

Definition 8.3.1 *Let $\mathcal{R}$ be a class of functions. The language $L_1 \subseteq \Sigma_1^*$ is $\mathcal{R}$* reducible *(more precisely $\mathcal{R}$ many-to-one reducible) to the language $L_2 \subseteq \Sigma_2^*$ (we denote this as $L_1 \leq_{\mathcal{R}}^m L_2$), if there exists a function f from $\mathcal{R}$, such that $w \in L_1 \Leftrightarrow f(w) \in L_2$.*

If all problems of some class $\mathcal{R}$ reduce to a certain problem, then this problem is complete with respect to $\mathcal{R}$ reduction in this class (it is at least as hard as any other problem of this class):

DEF

Definition 8.3.2 *Let $\mathcal{R}$ be a class of functions and $\mathcal{C}$ a class of languages. The language L_0 is $\mathcal{C}$-*hard with respect to $\mathcal{R}$ reducibility*, if $\forall L \in \mathcal{C} : L \leq^m_{\mathcal{R}} L_0$.*

If a problem is hard for a class with respect to $\mathcal{R}$ reduction and, moreover, it belongs to that class, then it is complete for that class (the hardest problem from this class):

Definition 8.3.3 *Let $\mathcal{R}$ be a class of functions and $\mathcal{C}$ a class of languages. We say that the language L_0 is $\mathcal{C}$-*complete with respect to $\mathcal{R}$ reducibility*, if $L_0 \in \mathcal{C}$ and L_0 is $\mathcal{C}$-hard with respect to $\mathcal{R}$ reducibility.*

For classes $\mathcal{C}_1 \subseteq \mathcal{C}_2$ and a language L which is $\mathcal{C}_2$ complete with respect to the $\mathcal{R}$ reducibility, then exactly one of the following holds: either the class $\mathcal{C}_2$ itself is $\mathcal{R}$ reducible to $\mathcal{C}_1$ (meaning that each language of that class is $\mathcal{R}$ reducible to $\mathcal{C}_1$) or $L \in \mathcal{C}_2 \setminus \mathcal{C}_1$.

8.3.8 The most common types of $\mathcal{R}$ reduction and completeness

We will mention *the most common types of completeness* – notice that the reduction which is used is strong enough to make finding of complete problems with respect to it possible and, on the other hand, it is strong enough to prevent the respective classes from being trivially reducible to their (potential) subclasses:

- **NP**, **PSPACE**, **EXP** completeness is defined with respect to *polynomial reducibility* (i.e. reducibility using DTM running in polynomial time),
- **P**, **NLOGSPACE** completeness is defined with respect to *reducibility in deterministic logarithmic space*,
- **NEXP** completeness is defined with respect to *exponential reducibility* (i.e. reducibility using DTM running in exponential time).

The polynomial reduction is particularly important, because similarly to how the P class specifies practically solvable problems, polynomial reducibility corresponds to performable convertibility of problems.

8.4 Examples of problems complexity

Examples of the **LOGSPACE** *problems*:

- existence of the path between two vertices in an undirected graph.

Examples of the **NLOGSPACE**-*complete problems*:

- existence of the path between two vertices in directed graph,
- 2-SAT (*SATisfiability*), i.e. satisfiability of propositional formulas in the form of conjunction of disjunctions of two literals (literal is a propositional variable or its negation), e.g. $(x_1 \vee \neg x_3) \wedge (\neg x_2 \vee x_3)$.

Examples of the **P**-*complete problems*:

- satisfiability of Horn clauses $(p \wedge q \wedge ... \wedge t) \Rightarrow u$, where p, q, ... are atomic formulas of predicate logic,
- membership of a string in the language of a context-free grammar,
- vertices succession in depth-first search of graph (for given sorting of direct successors).

Examples of the **NP**-*complete problems*:

- 3-*SAT* and general *SAT* – see further,
- number of graph problems, e.g.:
  - existence of a clique of given size,
  - existence of a Hamiltonian path in an undirected graph,
  - existence of a oriented Hamiltonian path in a directed graph,
  - colorability – can the given undirected graph be colored with certain number of colors?,
  - vertex cover of undirected graph by set of vertices of certain size (i.e. by set of vertices, which are connected by all edges),
  - ...
- traveling salesman problem,
- *knapsack* – we have items with price and value, and we maximize the value in a way that price doesn't exceed a certain bound.

Examples of the *co*-**NP**-*complete problems*:

- equivalence of regular expressions without iteration.

Examples of the **PSPACE**-*complete problems*:

- equivalence of regular expressions,
- membership of a string in the language of context-sensitive grammar,
- model checking of formulas of linear temporal logic (LTL – propositional logic extended by temporal operators *until*, *always*, *eventually*, *next-time*) with respect to the size of a formula,

- the best turn in the game Sokoban.

Examples of the **EXP**-*complete problems*:

- the best turn in chess (generalized to the chessboard $n \times n$),
- model checking of processes with unlimited stack (recursion) with respect to fixed formula of branching time logic (CTL) – i.e. EXP in the size of process.
- inclusion for the so-called *visibly push-down* languages (operations push/pop, which are performed by accepting automaton, are part of input string).

Examples of the **EXPSPACE**-*complete problems*:

- equivalence of regular expressions extended by the quadrate operation (i.e. r^2).

k-EXP / k-EXPSPACE:

- deciding the satisfiability of formulas of *Presburger arithmetic* – i.e. arithmetic of natural numbers with addition, equality (not multiplication – this leads to the so-called Peano arithmetic, which is already undecidable) and first-order quantification (e.g. $\forall x, y : x \leq x + y$) is a problem, which is in **3-EXP** (**2-EXPSPACE**-complete).

Problems outside of **ELEMENTARY**:

- equivalence of regular expressions extended by the negation operation,
- decision of satisfiability of formulas of *WS1S* – arithmetic of natural numbers with operation $+1$ and first-order quantification and second-order quantification (i.e. for all/ there exists a value, or a set of values, such that ...),
- verification of reachability in the so-called *Lossy Channel Systems* – processes are communicating via unbounded, but lossy queue (anything can get lost anytime).

8.5 SAT-problem

SAT-problem (problem of satisfiability of Boolean formulas) was the first problem whose NP-completeness with respect to polynomial reduction was proven. In many cases, it can be used in the proof of NP-hardness.

8.5.1 Polynomial reduction

Definition 8.5.1 Polynomial reduction *of the language L_1 over the alphabet Σ_1 to the language L_2 over the alphabet Σ_2 is a function $f : \Sigma_1^* \rightarrow \Sigma_2^*$, for which the following holds:*

1. $\forall w \in \Sigma_1^* : w \in L_1 \Leftrightarrow f(w) \in L_2$
2. *f is Turing-computable in polynomial time*

If there exists a polynomial reduction of the language L_1 to L_2, we say, that L_1 reduces to *L_2 and we write $L_1 \leq_P^m L_2$.*

Theorem 8.5.1 If $L_1 \leq_P^m L_2$ and L_2 is in the class P, then L_1 is in the class P.

Poof. Let M_f be a Turing machine, which performs the reduction f of the language L_1 to L_2 and let $p(x)$ be its time complexity. For any $w \in L_1$, the computation $f(w)$ requires at most $p(|w|)$ steps and produces an output of maximal length $p(|w|)+|w|$. Let M_2 accepts the language L_2 in polynomial time given by a polynomial $q(x)$. Consider a Turing machine, which arises from the composition $\rightarrow M_f M_2$. This machine accepts the language L_1 in a way that for all $w \in L_1$ the machine will perform $\rightarrow M_f M_2$ at most $p(|w|) + q(p(|w|) + |w|)$ steps, which is a polynomial in $|w|$ and thus L_1 is in the same class P. $\square$

Example 8.5.1 *Let us consider the function $f : \{x, y\}^* \rightarrow \{x, y, z\}^*$ defined as $f(v) = vzv$. This function is a polynomial reduction of the language $L_1 = \{w | w \text{ is palindrome over } \{x, y\}\}$ to the language $L_2 = \{wzw^R | w \in \{x, y\}^*\}$.*

The previous theorem gives us a practical possibility to show that a certain language is in class P. Moreover, if we reformulate this theorem as follows: "If holds $L_1 \leq_P^m L_2$ and L_1 is not in P, then L_2 is also not in P", we can prove that certain language is not in P.

8.5.2 Satisfiability problem - SAT problem

Let $V = \{v_1, v_2, \ldots, v_n\}$ be a finite set of Boolean variables (atomic formulas of propositional calculus). By *literal*, we mean any variable v_i or a negation of a variable $\overline{v_i}$. Any propositional formula containing only literals bound by a logical connective $\vee$ (or) is called a *clause*. The examples of such clauses are $v_1 \vee \overline{v_2}$, $v_2 \vee v_3$, $\overline{v_1} \vee \overline{v_3} \vee v_2$.

A SAT-problem can be formulated as follows: Given the set of variables V and the set of clauses over V, is the set of clauses satisfiable?

We can encode every SAT-problem by only one string:
Let $V = \{v_1, v_2, \ldots, v_n\}$, we encode each literal v_i by a string of the length

m which contains only 0s except for the i-th position, which contains the symbol p, if it is literal v_i, or n, if it is literal $\overline{v_i}$. Each clause is represented by a list of encoded literals separated by the symbol /. SAT-problem will be the list of parenthesized clauses.

$\boxed{\texttt{x+y}}$

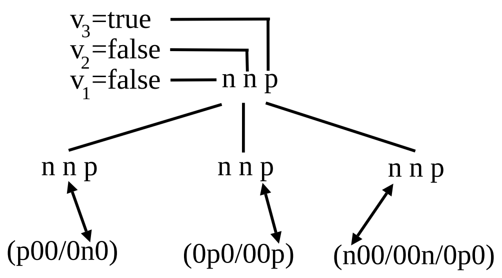
Example 8.5.2 *SAT-problem containing variables* v_1, v_2, v_3 *and clauses* $v_1 \vee \overline{v_2}$*,* $v_2 \vee v_3$*,* $\overline{v_1} \vee \overline{v_3} \vee v_2$ *will be represented as the string:*

$$(p00/0n0)(0p0/00p)(n00/00n/0p0)$$

Let L_{SAT} be the language containing strings, which represent satisfiable sets of clauses. The string $(p00/0n0)(0p0/00p)(n00/00n/0p0)$ is a member of L_{SAT} $(v_1 = F, v_2 = F, v_3 = T)$, contrary to the string $(p00/0p0)(n00/0p0)(p00/0n0)$ $(n00/0n0)$, which is a code of unsatisfiable set of the clauses $v_1 \vee v_2$, $\overline{v_1} \vee v_2$, $v_1 \vee \overline{v_2}$, $\overline{v_1} \vee \overline{v_2}$.

Assignment of truth values will be represented by a string from $\{p, n\}^+$, where p at the i-th position represents the assignment $v_i \approx$ true and n at the i-th position represents the assignment $v_i \approx$ false.

Then, testing whether a certain valuation is a model of a set of clauses (the set of clauses is satisfied under this valuation) or not is very simple and is illustrated in the following figure:



By using this principle, we can construct a non-deterministic Turing machine which accepts the language L_{SAT} in polynomial time. We choose a 2-tape Turing machine, which:

1. firstly, tests whether the input represents a set of clauses,

2. it generates a string from $\{n, p\}^m$ to the 2. tape in a non-deterministic way,

3. moves head on the 1. tape and tests if the input set of clauses is satisfiable under the evaluation on the 2. tape.

This process can be easily implemented with polynomial complexity of acceptance with respect to the length of the input string and thus $L_{SAT} \in NP$.

Theorem 8.5.2 Cook's theorem: *If* L *is any language from* NP*, then* $L \leq_P^m L_{SAT}$*.*

Poof. Because $L \in NP$, there exists a non-deterministic Turing machine M and a polynomial $p(x)$ such that for all $w \in L$, the machine M accepts w in at most $p(|w|)$ steps.

The core of the proof is a construction of the polynomial reduction f from L to L_{SAT}: For every $w \in L$, $f(w)$ will be a set of clauses which are satisfiable, if and only if M accepts w. The clauses can be divided into four groups which represent the following four statements:

1. At any time during the computation, M is exactly in one state, the head is over exactly one tape cell, and each tape cell contains exactly one symbol.

2. M starts its computation from its initial state, with its tape head over the leftmost tape cell, and with its tape containing a blank, followed by the input string w, followed by blanks.

3. At each step in the computation, the tape head position, current state, and tape content change according to some of M's transitions.

4. The computation reaches the machine's final state.

In the rest of the proof, we consider that:

- the states of M are represented as $q_1, q_2, \ldots, q_r$, where q_1 is the initial and q_2 is the final state

- the blank symbol is represented as x_1 and the nonblank tape symbols are represented as $x_2, x_3, \ldots, x_m$

For a translation of the four statements into clauses, we will use variables identified in figure 8.1. Each variable stands for some statement about either head, state, or content of some cell at some point of computation (at some time). The following two paragraphs explain ranges of indices used in the figure 8.1.

We know that the computation may consist of at most $p(|w|)$ steps (transitions), thus yielding at most $p(|w|) + 1$ configurations. For cases when the computation halts after number of steps which is less then $p(|w|)$, we assume, for simplicity, that M remains in its halt configuration for all "future" configurations after halting. The used indices of configurations (points of computation) range from 0 to $p(|w|)$ (they are denoted by symbol i in the figure).

It follows from the lemma 8.1.1 that M won't be able to move its head beyond the cell number $p(|w|) + 1$. The used cell indices range from 1 to $p(|w|) + 1$ (they are denoted by symbol j in the figure).

Now we are prepared to translate the four statements into clauses. Statement 4 is simplest since it can be represented by the clause containing one variable:

$$St_{p(n),2}$$

Variable	Range of subscripts	Corresponding statement	Number of variables of this type
$Hd_{i,j}$	$0 \leq i \leq p(n)$ $1 \leq j \leq p(n)+1$	M's tape is over cell j at time i	$[p(n)+1]^2$
$St_{i,j}$	$0 \leq i \leq p(n)$ $1 \leq j \leq r$	M is in state q_i at time i	$r[p(n)+1]$
$Cont_{i,j,k}$	$0 \leq i \leq p(n)$ $1 \leq j \leq p(n)+1$ $1 \leq k \leq m$	Cell j contains symbol x_k at time i	$m[p(n)+1]^2$

Figure 8.1: The variables used in the instance of SAT represented by $f(w)$ (head, state and content variables)

Statements 1 and 2 are a bit more involved. Their translation is shown in figures 8.2 and 8.3. The idea which is used in the translation of statement 1 is illustrated on the following example: Consider variables x, y and z. We need to ensure that exactly one of them becomes true (by constructing a suitable formula and requiring its satisfiability). This can be done by a conjunction of two formulas:

- $x \vee y \vee z$ – this formula evaluates to true if and only if at least one of the three variables is true
- $(\overline{x} \vee \overline{y}) \wedge (\overline{x} \vee \overline{z}) \wedge (\overline{y} \vee \overline{z})$ – this formula evaluates to true if and only if at most one of the three variables is true

The conjunction of the previous two formulas evaluates to true if and only if both of them evaluate to true, i.e. if and only if exactly one of the three variables is true.

Let us now consider statement 3. We translate it according to the transitions that can be performed by M. Recall that these transitions can be classified into three categories: those that move the tape head to the right, those that move the tape head to the left, and those that change the contents of the current cell. We will deal with each of the three cases separately. We denote the transition relation of M as δ_M.

Each transition in the first category requires the machine be in a particular state q_s with a particular symbol x_k in its current cell, and results in a shift to a new state q_t while the tape head is moved to the right. Clauses which are created for this type of transitions are specified in the figure 8.4. Notice that if we consider the following simple rewriting,

$$\begin{aligned} [\neg p_1 \vee \neg p_2 \vee \neg p_3 \vee p_4] &\iff [\neg(p_1 \wedge p_2 \wedge p_3) \vee p_4] \\ &\iff [(p_1 \wedge p_2 \wedge p_3) \implies p_4] \end{aligned}$$

Constraint	Clauses	Number of such clauses
M's tape head must be over at least one cell at any time	$Hd_{i,1} \vee Hd_{i,2} \vee \cdots \vee Hd_{i,p(n)+1}$ $\forall i \in \{0, 1, \ldots, p(n)\}$	$p(n) + 1$
M's tape head must be over at most one cell at any time	$\overline{Hd_{i,j}} \vee \overline{Hd_{i,k}}$ $\forall i \in \{0, 1, \ldots, p(n)\}$ $\forall j, k \in \{1, 2, \ldots, p(n) + 1\}, j < k$	$\frac{(p(n)+1)^2 p(n)}{2}$
M must be in at least one state at any time	$St_{i,1} \vee St_{i,2} \vee \cdots \vee St_{i,r}$ $\forall i \in \{0, 1, \ldots, p(n)\}$	$p(n) + 1$
M must be in at most one state at any time	$\overline{St_{i,j}} \vee \overline{St_{i,k}}$ $\forall i \in \{0, 1, \ldots, p(n)\}$ $\forall j, k \in \{1, 2, \ldots, r\}., j < k$	$(p(n) + 1)(\frac{r(r-1)}{2})$
Each tape cell must contain at least one symbol at any time	$Cont_{i,j,1} \vee Cont_{i,j,2} \vee \cdots \vee Cont_{i,j,m}$ $\forall i \in \{0, 1, \ldots, p(n)\}$ $\forall j \in \{1, 2, \ldots, p(n) + 1\}$	$(p(n) + 1)^2$
Each tape cell must contain at most one symbol at any time	$\overline{Cont_{i,j,k}} \vee \overline{Cont_{i,j,l}}$ $\forall i \in \{0, 1, \ldots, p(n)\}$ $\forall j \in \{1, 2, \ldots, p(n) + 1\}$ $\forall k, l \in \{1, 2, \ldots, m\}, k < l$	$(p(n) + 1)^2(\frac{m(m-1)}{2})$

Figure 8.2: Clauses representing statement 1

$$
\begin{array}{l}
St_{0,1} \\
Hd_{0,1} \\
Cont_{0,1,1} \\
Cont_{0,2,k_1} \\
Cont_{0,3,k_2} \\
\quad\vdots \\
Cont_{0,n+1,k_n} \\
Cont_{0,n+2,1} \\
\quad\vdots \\
Cont_{0,p(n)+1,1} \\
\hline
\end{array}
$$

Figure 8.3: Clauses representing statement 2

we can look at these clauses as an implication saying what must be the case when $St_{i,s} \wedge Hd_{i,j} \wedge Cont_{i,j,k}$ is true.

$$\begin{array}{lcl}
\forall (q_s, x_k) \rightarrow (q_t, R) \in \delta_M & & \overline{St_{i,s}} \vee \overline{Hd_{i,j}} \vee \overline{Cont_{i,j,k}} \vee Hd_{i+1,j+1} \\
\forall i \in \{0, 1, \ldots, p(n) - 1\} & \text{add} & \overline{St_{i,s}} \vee \overline{Hd_{i,j}} \vee \overline{Cont_{i,j,k}} \vee St_{i+1,t} \\
\forall j \in \{1, 2, \ldots, p(n)\} & & \overline{St_{i,s}} \vee \overline{Hd_{i,j}} \vee \overline{Cont_{i,j,k}} \vee Cont_{i+1,j,k}
\end{array}$$

Figure 8.4: Clauses created for rules which move the head to right

Transitions in the second category are treated similarly, as shown in figure 8.5. The third case is treated analogically too, as figure 8.6 shows. Notice the difference in the range of index j in all three categories. We can also see that each transition contributes no more than $3(p(n)+1)^2$ to the total number of clauses.

$$\begin{array}{lcl}
\forall (q_s, x_k) \rightarrow (q_t, L) \in \delta_M & & \overline{St_{i,s}} \vee \overline{Hd_{i,j}} \vee \overline{Cont_{i,j,k}} \vee Hd_{i+1,j-1} \\
\forall i \in \{0, 1, \ldots, p(n) - 1\} & \text{add} & \overline{St_{i,s}} \vee \overline{Hd_{i,j}} \vee \overline{Cont_{i,j,k}} \vee St_{i+1,t} \\
\forall j \in \{2, 3, \ldots, p(n) + 1\} & & \overline{St_{i,s}} \vee \overline{Hd_{i,j}} \vee \overline{Cont_{i,j,k}} \vee Cont_{i+1,j,k}
\end{array}$$

Figure 8.5: Clauses created for rules which move the head to left

$$\begin{array}{lcl}
\forall (q_s, x_k) \rightarrow (q_t, x_l) \in \delta_M & & \overline{St_{i,s}} \vee \overline{Hd_{i,j}} \vee \overline{Cont_{i,j,k}} \vee Hd_{i+1,j} \\
\forall i \in \{0, 1, \ldots, p(n) - 1\} & \text{add} & \overline{St_{i,s}} \vee \overline{Hd_{i,j}} \vee \overline{Cont_{i,j,k}} \vee St_{i+1,t} \\
\forall j \in \{1, 2, \ldots, p(n)\} & & \overline{St_{i,s}} \vee \overline{Hd_{i,j}} \vee \overline{Cont_{i,j,k}} \vee Cont_{i+1,j,l}
\end{array}$$

Figure 8.6: Clauses created for rules which rewrite the current symbol

To assure that cell contents change only in accordance with these rules we add the following clauses

$$\begin{array}{lcl}
\forall i \in \{0, 1, \ldots, p(n) - 1\} & & \\
\forall j \in \{1, 2, \ldots, p(n) + 1\} & \text{add} & Hd_{i,j} \vee \overline{Cont_{i,j,k}} \vee Cont_{i+1,j,k} \\
\forall k \in \{1, 2, \ldots, m\} & &
\end{array}$$

These clauses represent the statement: "If the tape head is not over cell j at time i, then cell j will remain unchanged at time $i+1$". This statement can be seen more clearly if we view the clause as $\overline{Hd_{i,j}} \implies \neg(Cont_{i,j,k} \wedge \overline{Cont_{i+1,j,k}})$. Note that there are only $mp(n)(p(n)+1)$ clauses of this kind.

Finally, we add the following clauses

$$\forall i \in \{0, 1, \ldots, p(n) - 1\} \quad \text{add} \quad \overline{St_{i,2}} \vee St_{i+1,2}$$

These clauses say that once in its final state, the machine will remain in its final state ($St_{i,2} \implies \overline{St_{i+1,2}}$).

Collecting all the clauses described together, we obtain a set of clauses that is satisfied by those and only those truth assignments that correspond to computations of M that accept the string w in time $p(|w|)$. Thus, we have obtained an instance of SAT that is satisfiable if and only if $w \in L$. In turn, the coded version of this instance is the string $f(w)$ we desire, since according to to this definition we have $w \in L$ if and only if $f(w) \in L_{SAT}$.

It remains to show that $f(w)$ can be computed in polynomial-time. To this end we note that the computation of $f(w)$ is essentially the process of listing the clauses represented by $f(w)$. But, the number of clauses to be listed, the number of literals in each clause, and the lengths of the strings representing each literal are all bounded by a polynomial in $|w|$. Thus, the string $f(w)$ can be computed in polynomial time.

□

Theorem 8.5.3 *SAT is* **NP**-*complete with respect to the polynomial reduction.*

Poof. Straightforward from the previous discussion. □

8.5.3 NP-complete languages

After the discovery of the Cook's theorem, it appeared that many other NP languages have the property similar to L_{SAT}, i.e. they are polynomial reductions of other NP languages.

These languages – as we already know – are called *NP-complete* languages.

If it was shown that any of these languages is in P, then it would must have held that $P = NP$; on the contrary, the proof that some of them is out of P would imply that $P \subset NP$.

8.5.4 Notable NP-complete problems

- *Satisfiability*: Is a boolean expression satisfiable?
- *Clique*: Does an undirected graph contain a clique of size k?
- *Vertex cover*: Does an undirected graph have a dominant set of the cardinality k?
- *Hamiltonian circuit*: Does an undirected graph have a Hamiltonian cycle?
- *Colorability*: Does an undirected graph have a chromatic number k?
- *Directed Hamiltonian circuit*: Does an undirected graph have a Hamiltonian cycle?

- *Set cover*: Given a class of the sets $S_1, S_2, \dots, S_n$, is there a subclass of k sets $S_{i_1}, S_{i_2}, \dots, S_{i_k}$ such that $\bigcup_{j=1}^{k} S_ij = \bigcup_{j=1}^{n} S_j$?
- *Exact cover*: Given a class of the sets $S_1, S_2, \dots, S_n$, is there a set cover induced by a subclass of pairwise disjoint sets?

8.5.5 Usage of reductions for proof of completeness

Example 8.5.3 *We will prove that the problem $3SAT = \{\phi \mid$ satisfiable formula ϕ in CNF contains at most 3 literals in each clause$\}$ is NP-complete.*

We have to show:

- *a) $3SAT \in NP$*
  NTM nondeterministically chooses an assignment of variables in the 3SAT formula and it checks, in polynomial time, whether this assignment is satisfiable.
- *b) $SAT \leq_p 3SAT$*
  We construct a polynomially computable function f, for which the following holds:
  $$A \in SAT \Leftrightarrow f(A) \in 3SAT$$
  The function f is defined as follows:
  Let $K = (a_1, a_2, a_3, ...a_n)$ be a clause in SAT which has more than 3 literals. We replace the clause K with the clauses $K_1 = (a_1, a_2, b)$ and $K_2 = (\neg b, a_3, ..., a_n)$, where the literal b does not occur in the original formula. It is obvious that the following holds:
  $$K_1 \cup K_2 \textit{ is satisfiable } \Leftrightarrow K \textit{ is satisfiable}$$
  Moreover, the clause K_2 contains one literal less than the original clause K. We continue analogically as long as the formula contains clauses containing more than 3 literals. It is obvious that the function f is polynomially computable and also fulfills all other requirements imposed on it.

Example 8.5.4 *We will prove that the problem $CLIQUE = \{(G, k) \mid G$ contains complete subgraph composed by k vertices$\}$ is NP-complete. We have to show :*

- *a) $CLIQUE \in NP$*
  NTM nondeterministically chooses k vertices and, in polynomial time, checks if they form the complete subgraph.
- *b) $SAT \leq_p CLIQUE$*
  We construct a polynomially computable function f, for which the following holds:
  $$A \in SAT \Leftrightarrow f(A) \in CLIQUE$$

The function f creates a graph G from the formula ϕ in CNF, and a number k such that the formula ϕ is satisfiable, if and only if the graph G contains a complete subgraph with k vertices. The number k corresponds to the number of clauses in the formula. Vertices of the graph G correspond to literals appearing in the formula. There is an edge between two vertices – which themselves correspond to two literals – if and only if these literals are not complementary and they appear in two separate clauses.

$\Leftarrow$:
Assume that the formula ϕ is satisfiable and that the function $H : \{x1, \dots, x_n\} \rightarrow \{0, 1\}$ is an assignment which satisfies ϕ. Then, for each clause, there exists at least one literal which evaluates to 1 under H. There exists an edge between all two vertices corresponding to these literals, because they are not complementary and they appear in separate clauses. Therefore, the composed graph is in $CLIQUE$.

$\Rightarrow$:
Assume that the vertices $k_1, \dots, k_k$ are vertices of a graph which form a complete subgraph. As there is an edge between all two vertices, the corresponding literals have to be in separate clauses and, at the same time, they can't be complementary. Therefore, we can assign the value 1 to these literals and we get satisfying assignment for ϕ.

Turing machines are often used also for definition and classification of time and space complexity of problems (algorithms). The most important complexity classes of problems (languages) are classes P, NP and NPC, however the other classes are significant for characterization of computation complexity too. The SAT problem is the representative problem of the class NPC.

8.6 Exercise

Exercise 8.6.1 Prove that the problem $DoubleSat = \{\phi \mid$ formula ϕ in CNF, which has two satisfying assignments $\}$ is NP-complete.

Exercise 8.6.2 Prove that the problem of deciding, if there exists a coloring of an undirected graph by three colors such that all two adjacent vertices have different color, is NP-complete.

Exercise 8.6.3 Consider the 2SAT problem. Decide if it is NP complete or it is in P.

Exercise 8.6.4 Define time and space complexity of a Turing machine's computation over a given language.

Exercise 8.6.5 Why do we define asymptotic bounds of complexity? For a given function f, define the set of functions $O(f(n))$, $\Omega(f(n))$ and $\Theta(f(n))$.

Exercise 8.6.6 Define the following complexity classes **P**, **NP**, **PSPACE**, **LOGSPACE**, **NLOGSPACE** and give examples of problems from these classes.

Exercise 8.6.7 Adduce the influence of the use of multi-tape TM and non-deterministic TM on computation complexity.

Exercise 8.6.8 Explain the Savitch's theorem and its meaning.

Exercise 8.6.9 Adduce the hierarchy of complexity classes and think about the strictness of each inclusions.

Exercise 8.6.10 Define general notions of reduction, hardness and completeness. Then, define polynomial reduction and explain its use on an example.

Exercise 8.6.11 Define the notion of NP-completeness and adduce examples of problems which are NP-complete.

Bibliography

[1] Aho, A. V, Ullman, J. D.: *The theory of parsing, translation and compiling*, EngelWood Cliffs, New Jersey, Prentice-Hall 1972.

[2] Kolář, J.: *Algebra a grafy*, Skriptum ES ČVUT, Praha, 1982.

[3] Kozen, D. C.: *A Completeness Theorem for KleeneAlgebras and the Algebra of Regular Events*, Technical Report TR 90–1123, 1990.

[4] Kozen, D. C.: *Automata and Computability*, Springer-Verlag, New York, Inc, 1997. ISBN 0-387-94907-0

[5] Rábová, Z., Češka, M.: *Základy systémového programování I.*, Učební texty vysokých škol. Ediční středisko VUT Brno, 1979.

[6] Rábová, Z., Češka, M., Honzík, J. M., Hruška, T.: *Programovací techniky*, Učební texty vysokých škol. Ediční středisko VUT Brno, 1985.

[7] Černá, I., Křetínský, M., Kučera, A.: *Automaty a formální jazyky I*, učební text FI MU, Brno, 1999.

[8] Hopcroft, J.E., Motwani, R., Ullman, J.D.: *Introduction to Automata Theory, Languages, and Computation*, Addison Wesley, 2. vydání, 2000. ISBN 0-201-44124-1

[9] Gruska, J.: *Foundations of Computing*, International Thomson Computer Press, 1997. ISBN 1-85032-243-0

[10] Bovet, D.P., Crescenzi, P.: *Introduction to the Theory of Complexity*, Prentice Hall Europe, Pearson Education Limited, 1994. ISBN 0-13-915380-2