Lecture Theoretical Computer Science II:

Foundations of Theoretical Computer Science

Lecture notes

of

V. Claus and E.-R. Olderog

translated by Ievgeniia Karashchuk

Winter semester 2010/11

Contents

Chapter I

Basic definitions

§1 Modelling in theoretical computer science

Characteristics of the theory:

- from the particular to the general
- use of modelling to answer general questions
- analysis and synthesis of models

Question: What is necessary to describe the syntax of a programming language?
Modelling of languages:
We consider CHOMSKY-languages and in particular

- regular languages
- context-free languages

We will show: Regular languages are recognized by finite automata.
Context-free languages are recognized by push-down automata.

Question: How can we describe parallel and communicating systems?
Modelling of processes and process calculi:

- Operators for parallelism and communication

Question: How can we describe time-critical systems?
Modelling of real-time automata:

- Extension of finite automata with clocks

Question: What tasks can be accomplished using computers?
Modelling of a computer:

- Turing machines (1936)

- Grammars (1959)

We will show: These approaches are equivalent. There exist non-computable functions. Examples of non-computable functions are considered. Finite and push-down automata can be seen as special cases of Turing machines.

Question: How much time and memory does the computation take?
Modelling of complexity:

- "fast", i. e., polynomial algorithms
- complexity classes P and NP

Open problem: Does $P = NP$ hold?

Topics of the lecture:

- Automata
- Formal languages and grammars
- Computability
- Complexity

§2 Logic, sets, relations and functions

Below we have put together the notations that will be used in these lecture notes.

- In logical formulae we use logical connectives $\neg$ (*negation*), $\wedge$ (*conjunction*), $\vee$ (*disjunction*), $\Rightarrow$ (*implication*) and $\Leftrightarrow$ (*equivalence*) as well as quantifiers $\forall$ (*universal quantifier*) and $\exists$ (*existential quantifier*).

- We can describe finite sets by listing their elements, for example, $\{a, b, c, d\}$ or $\{coffee, tea, sugar\}$. The *empty set* $\{\}$ is also denoted by $\emptyset$. An important example of an infinite set is the set $\mathbb{N}$ of *natural numbers*: $\mathbb{N} = \{0, 1, 2, 3, ...\}$.
  $x \in X$ denotes that x is an *element* of the set X, and $x \notin X$ denotes that x is *not* an element of the set X.
  The *principle of extensionality* holds, i. e., two sets are equal if they have the same elements. For example,
  $$\{a, b, c, d\} = \{a, a, b, c, d, d\}.$$
  $X \subseteq Y$ means that X is a *subset* of Y, i. e., $\forall\, x \in X : x \in Y$. Subsets can be *selected* from the given sets using logical formulae. For example,
  $$M = \{n \in \mathbb{N} \mid n \ mod\ 2 = 0\}$$
  describes the set of even natural numbers.
  For sets $X, Y \subseteq Z$ there exist standard set-theoretic operations

union:	$X \cup Y$	$=$	$\{z \in Z \mid z \in X \vee z \in Y\}$
intersection:	$X \cap Y$	$=$	$\{z \in Z \mid z \in X \wedge z \in Y\}$
difference:	$X - Y$	$=$	$X \setminus Y = \{z \in Z \mid z \in X \wedge z \notin Y\}$
complement:	$\overline{X}$	$=$	$Z - X$

Furthermore $X \dot{\cup} Y$ stands for the *disjoint union* of X and Y, i. e., additionally $X \cap Y = \emptyset$ holds.

$|X|$ and $card(X)$ denote the *cardinality* or *size* of the set X. $|X|$ is the number of elements in the set X when X is finite. E.g., $|\{coffee, tea, sugar\}| = 3$.

$\mathbb{P}(Z)$ and 2^Z denote the *powerset* of a set Z, i. e., the set of all subsets of Z: $\mathbb{P}(Z) = \{X \mid X \subseteq Z\}$. In particular, $\emptyset \in \mathbb{P}(Z)$ and $Z \in \mathbb{P}(Z)$ holds.

$X \times Y$ denotes the (*Cartesian*) *product* of two sets X and Y, consisting of all pairs where the first element is from X and the second from Y: $X \times Y = \{(x, y) \mid x \in X \wedge y \in Y\}$. In general, $X_1 \times ... \times X_n$ denotes the set of all n-tuples, where for every $i \in \{1, ..., n\}$ it holds that the set X_i contains the i-th component: $X_1 \times \cdots \times X_n = \{(x_1, ..., x_n) \mid x_1 \in X_1 \wedge ... \wedge x_n \in X_n\}$.

- Relations are special sets. A (*2-place* or *binary*) *relation* R of two sets X and Y is a subset of the product $X \times Y$, i. e., $R \subseteq X \times Y$. The infix-notation xRy is often used to denote element relation $(x, y) \in R$. The *domain* of R is defined by

$$dom(R) = \{x \in X \mid \exists\, y \in Y : (x, y) \in R\}$$

and the *range* of R by

$$ran(R) = \{y \in Y \mid \exists\, x \in X : (x, y) \in R\}.$$

The relation $R \subseteq X \times Y$ is

left-unique,	if	$\forall\, x_1, x_2 \in X, y \in Y : (x_1, y) \in R \wedge (x_2, y) \in R \Rightarrow x_1 = x_2$,
right-unique,	if	$\forall\, x \in X, y_1, y_2 \in Y : (x, y_1) \in R \wedge (x, y_2) \in R \Rightarrow y_1 = y_2$,
left-total,	if	$X = dom(R)$,
right-total,	if	$ran(R) = Y$.

The *identity relation* on X is denoted by id_X: $id_X = \{(x, x) \mid x \in X\}$. The *inverse relation* of $R \subseteq X \times Y$ is $R^{-1} \subseteq Y \times X$ that is defined as follows:

$$\forall x \in X, y \in Y : (x, y) \in R \Leftrightarrow (y, x) \in R^{-1}$$

The *composition* $\circ$ of two relations $R \subseteq X \times Y$ and $S \subseteq Y \times Z$ is defined as follows: for all $x \in X$ and $z \in Z$ it holds

$$(x, z) \in R \circ S \quad \Leftrightarrow \quad \exists\, y \in Y : (x, y) \in R \text{ and } (y, z) \in S.$$

By a 2-place relation *on the set* X we understand a relation $R \subseteq X \times X$. This relation is

reflexive, if $\forall\, x \in X : (x,x) \in R$,
or in terms of relations: $id_X \subseteq R$,

irreflexive, if $\neg\, \exists\, x \in X : (x,x) \in R$,
or in terms of relations: $R \cap id_X = \emptyset$,

symmetric, if $\forall\, x,y \in X : (x,y) \in R \Rightarrow (y,x) \in R$,
or in terms of relations: $R = R^{-1}$,

antisymmetric, if $\forall\, x,y \in X : (x,y) \in R \wedge (y,x) \in R \Rightarrow x = y$,
or in terms of relations: $R \cap R^{-1} \subseteq id_X$,

transitive, if $\forall\, x,y,z \in X : (x,y) \in R \wedge (y,z) \in R \Rightarrow (x,z) \in R$,
or in terms of relations: $R \circ R \subseteq R$.

R is an *equivalence relation* if R is reflexive, symmetric and transitive (note the initial letters *r-s-t* of these properties). The equivalence relation R on X divides the set X into disjoint subsets, i. e., each subset contains pairwise equivalent elements. $[x]_R$ is an *equivalence class* of x for element $x \in X$, i. e., the set

$$[x]_R = \{y \in X \mid (x,y) \in R\}.$$

An element of an equivalence class is called a *representative* of this class because the whole class can be identified with the representative and the relation R. Hence an arbitrary element can be chosen as a representative element of its class because

$$\forall x,y \in X : (x,y) \in R \Leftrightarrow [x]_R = [y]_R.$$

By the *index of* R we mean the set cardinality of all equivalence classes of R on X. Notation:

$$Index(R) = |\{\ [x]_R \mid x \in X\}|$$

If it is clear from the context which R is considered, we write $[x]$ instead of $[x]_R$.

An equivalence relation R on X is called a *refinement* of an equivalence relation S on X if $R \subseteq S$. Then every equivalence class of R is a subset of some equivalence class of S.

The *n-th power* of $R \subseteq X \times X$ is defined inductively:

$$R^0 = id_X \qquad \text{and} \qquad R^{n+1} = R \circ R^n$$

The *transitive closure* R^+ and the *reflexive transitive closure* R^* of R are defined as follows:

$$R^+ = \bigcup_{n \in \mathbb{N}\setminus\{0\}} R^n \qquad \text{and} \qquad R^* = \bigcup_{n \in \mathbb{N}} R^n$$

- Functions are special relations. A relation $f \subseteq X \times Y$ is called a *partial function* (or a *partial image*) from X to Y if f is right-unique. It is denoted as follows $f : X \xrightarrow{part} Y$. Instead of $(x,y) \in f$ we write $f(x) = y$. f is a (*total*) *function* from X to Y if f is additionally left-total. This is denoted by $f : X \longrightarrow Y$.

  A function $f : X \longrightarrow Y$ is

  injective, if f is left-unique,
  surjective, if f is right-total,
  bijective, if f is injective und surjective.

  A bijective function is also called *bijection*.

§3 Alphabets, strings and languages

In this lecture we will pay special attention to analysing formal languages. For this purpose the following notation will be used.

- *Alphabet* = finite set of characters (symbols) = character set
  We use A, B, Σ, Γ as typical names for alphabets and a, b as typical names for characters, i. e., elements of alphabets.

- *String* over an alphabet Σ = finite sequence of characters from Σ. In particular, there is the *empty string* ε. We use u, v, w as typical names for strings.

  Example : Let $\Sigma = \{1, 2, +\}$. Then $1 + 2$ and $2 + 1$ are strings over Σ.

Σ^*	=	set of all strings over Σ. $\Sigma \subseteq \Sigma^*$ holds.
Σ^+	=	set of all non-empty strings over Σ, i. e., $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$.
$\lvert u \rvert$	=	length of the string u = number of characters that occur in u. In particular: $\lvert\varepsilon\rvert = 0$.

  The symbols of a string u with the length n we denote by $u_1, \ldots, u_n$.

  The *concatenation* of two strings u and v consists of the characters from u, followed by the characters from v and is denoted by $u \cdot v$ or simply by uv.

  Example : The concatenation of $1+$ and $2 + 0$ is $1 + 2 + 0$.

  A string v is called *substring* of a string w if $\exists u_1, u_2 : w = u_1 v u_2$.
  A string v is called *prefix* of a string w if $\exists u : w = vu$.
  A string v is called *suffix* of a string w if $\exists u : w = uv$.
  There may be several *occurrences* of the same substring in string w.

  Example : The string $w = 1 + 1 + 1$ has two occurrences of the substring $1+$ and three occurrences of the substring 1.

- A (*formal*) *language* over an alphabet Σ is a subset of Σ^*. We use L as a typical name for a language.

  There exist standard set-theoretic operations on languages :

$L_1 \cup L_2$	(union)
$L_1 \cap L_2$	(intersection)
$L_1 - L_2$ or $L_1 \backslash L_2$	(difference)
$\overline{L} =_{df} \Sigma^* - L$	(complement)

  Furthermore, there are special operations on languages.
  The *concatenation* of strings is extended to *languages* L_1 and L_2 as follows:

  $$L_1 \cdot L_2 = \{u \cdot v \mid u \in L_1 \text{ and } v \in L_2\}.$$

  The *n-th power* of a language L is defined inductively:

  $$L^0 = \{\varepsilon\} \text{ and } L^{n+1} = L \cdot L^n$$

The (KLEENE) *star operator* (also KLEENE-*closure* or *iteration*) of a language L is

$$L^* = \bigcup_{n \in \mathbb{N}} L^n = \{w_1 \dots w_n \mid n \in \mathbb{N} \text{ and } w_1, \dots, w_n \in L\}.$$

For all L it holds that $\varepsilon \in L^*$.

Example : For $L_1 = \{1+, 2+\}$ and $L_2 = \{1+0, 1+1\}$ is

$$\begin{aligned} L_1 \cdot L_2 &= \{1+1+0, 1+1+1, 2+1+0, 2+1+1\}, \\ L_1^2 &= \{1+1+, 1+2+, 2+1+, 2+2+\}, \\ L_1^* &= \{\varepsilon, 1+, 2+, 1+1+, 1+2+, 2+1+, 2+2+, 1+1+1+, 1+1+2+, ...\}. \end{aligned}$$

§4 Bibliography

The following books are recommended for further reading:

- J.E. Hopcroft, R. Motwani & J.D. Ullmann: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, 2nd Edition, 2001.
- U. Schöning: Theoretische Informatik – kurzgefasst. Spektrum Akademischer Verlag, 5. Auflage, 2008.
- D. Harel: Algorithmics – The Spirit of Computing. Addison-Wesley, 1987.

The following sources were used during the preparation for the lecture:

J. Albert & T. Ottmann: Automaten, Sprachen und Maschinen für Anwender. BI 1983 (nur einführendes Buch).

E. Börger: Berechenbarkeit, Komplexität, Logik. Vieweg, Braunschweig 1986 (2. Auflage).

W. Brauer: Automatentheorie. Teubner Verlag, Stuttgart 1984.

E. Engeler & P. Läuchli: Berechnungstheorie für Informatiker. Teubner, Stuttgart 1988.

H. Hermes: Aufzählbarkeit, Entscheidbarkeit, Berechenbarkeit. 2. Auflage, Springer-Verlag, Berlin 1971.

J.E. Hopcroft & J.D. Ullmann: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, 1979.

A.R. Lewis & C.H. Papadimitriou: Elements of the Theory of Computation. Prentice Hall, Englewood Cliffs 1981.

K.R. Reischuk: Einführung in die Komplexitätstheorie. Teubner Verlag, Stuttgart 1990.

A. Salomaa: Computation and Automata. Cambridge University Press, Cambridge 1985.

A. Salomaa: Formal Languages. Academic Press, New York 1973.

F. Setter: Grundbegriffe der Theoretischen Informatik. Springer-Verlag, Heidelberg 1988 (einführendes Buch).

W. Thomas: Grundzüge der Theoretischen Infomatik. Vorlesung im Wintersemester 1989/90 an der RWTH Aachen.

Chapter II

Finite automata and regular languages

In the following chapter we deal with a simple model of a machine: the finite automaton. We will see that

- finite automata may be widely used in computer science,
- the languages recognized by finite automata have many structural properties, e.g. representability by regular expressions,
- questions about computations performed by finite automata are decidable.

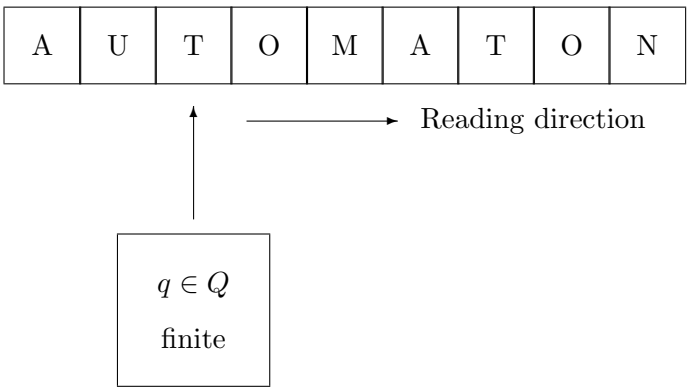
Moreover, finite automata can be easily implemented as tables in programs and as circuits.

§1 Finite automata

We want to use finite automata for recognizing languages. We can imagine a finite automaton as a machine with final states, which reads characters from a tape, can move the head only to the right and can print no new symbols on the tape.

Therefore, the transition function of finite automata is often defined as mapping $\delta : Q \times \Sigma \to Q$, where Q is a set of states and Σ is an input alphabet of the automaton. This automaton can be applied to the string $w \in \Sigma^*$ to check its acceptance.

Sketch:



The representation of automata as the so called *transition systems* is more suitable for graphical representation and definition of the acceptance behaviour of automata.

1.1 **Definition (deterministic finite automaton)**: A *deterministic finite automaton (acceptor)*, in short DFA, is a 5-tuple

$$\mathcal{A} = (\Sigma, Q, \delta, q_0, F) \quad \text{or} \quad \mathcal{A} = (\Sigma, Q, \rightarrow, q_0, F)$$

with following properties:

1. Σ is the finite *input alphabet*,
2. Q is a finite set of *states*,
3. $\delta : Q \times \Sigma \rightarrow Q$ is the *transition function*
   resp. $\rightarrow \subseteq Q \times \Sigma \times Q$ is a *deterministic transition relation*, i. e.,
   $$\forall\, q \in Q \quad \forall\, a \in \Sigma \quad \exists \text{ exactly } q' \in Q : (q, a, q') \in \rightarrow,$$
4. $q_0 \in Q$ is the *initial state*,
5. $F \subseteq Q$ is the set of *final states*.

Both representations of automata are related as follows :

$$\delta(q, a) = q' \Leftrightarrow (q, a, q') \in \rightarrow$$

The elements $(q, a, q') \in \rightarrow$ are called *transitions*. We mostly write $q \xrightarrow{a} q'$ instead of $(q, a, q') \in \rightarrow$. For given $a \in \Sigma$ we use $\xrightarrow{a}$ also as a binary relation $\xrightarrow{a} \subseteq Q \times Q$:

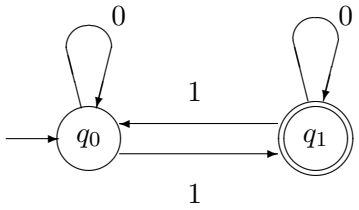
$$\forall\, q, q' \in Q : q \xrightarrow{a} q' \Leftrightarrow (q, a, q') \in \rightarrow$$

A DFA can be graphically represented by a *finite state diagram*. It is a directed graph which contains a vertex labelled with q for every state q of the automaton and a directed labelled edge for every transition $q \xrightarrow{a} q'$. The initial state q_0 is marked with an incoming arrow. Final states are labelled with an additional circle.

Example : Consider $\mathcal{A}_1 = (\{0,1\}, \{q_0, q_1\}, \rightarrow, q_0, \{q_1\})$ with

$$\rightarrow = \{(q_0, 0, q_0), (q_0, 1, q_1), (q_1, 0, q_1), (q_1, 1, q_0)\}.$$

Then $\mathcal{A}_1$ is represented by the following state diagram:



1.2 **Definition (Acceptance and reachability)**::
Let $\mathcal{A} = (\Sigma, Q, \rightarrow, q_0, F)$ resp. $\mathcal{A} = (\Sigma, Q, \delta, q_0, F)$ be a DFA.

1. We extend the transition relations $\xrightarrow{a}$ from input symbols $a \in \Sigma$ to strings of these symbols $w \in \Sigma^*$ and define the respective relations $\xrightarrow{w}$ inductively:

   - $q \xrightarrow{\varepsilon} q'$ if and only if $q = q'$
     or in terms of relations: $\xrightarrow{\varepsilon} = id_Q$
   - $q \xrightarrow{aw} q'$ if and only if $\exists\, q'' \in Q : q \xrightarrow{a} q''$ and $q'' \xrightarrow{w} q'$
     or in terms of relations: $\xrightarrow{aw} = \xrightarrow{a} \circ \xrightarrow{w}$

   Similarly we define the extended transition function δ^*

   $$\delta^* : Q \times \Sigma^* \rightarrow Q \text{ with } \delta^*(q, \varepsilon) = q \text{ and } \delta^*(q, aw) = \delta^*(\delta(q, a), w)$$

   for all $q, q' \in Q$, $a \in \Sigma$ and $w \in \Sigma^*$. It holds that:

   $$\delta^*(q, w) = q' \quad \Leftrightarrow \quad q \xrightarrow{w} q'$$

2. The *language accepted* by $\mathcal{A}$ is

   $$L(\mathcal{A}) = \{w \in \Sigma^* \mid \exists\, q \in F : q_0 \xrightarrow{w} q\} \text{ resp. } L(\mathcal{A}) = \{w \in \Sigma^* \mid \delta^*(q_0, w) \in F\}$$

   A language L is *finitely acceptable* if there is a DFA $\mathcal{A}$ with $L = L(\mathcal{A})$.

3. A state $q \in Q$ is *reachable* in $\mathcal{A}$, if $\exists w \in \Sigma^* : q_0 \xrightarrow{w} q$.

Example : For the automaton from the last example it holds that:

$$L(\mathcal{A}_1) = \{w \in \{0,1\}^* \mid w \text{ has an } \textit{odd} \text{ number of } 1's\}.$$

Remark : For all $a_1, \ldots, a_n \in \Sigma$ and $q, q' \in Q$:

$$q \xrightarrow{a_1 \cdots a_n} q' \quad \Leftrightarrow \quad \exists\, q_1, \ldots, q_n \in Q : q \xrightarrow{a_1} q_1 \quad \ldots \quad q_{n-1} \xrightarrow{a_n} q_n = q'.$$

or in terms of relations:

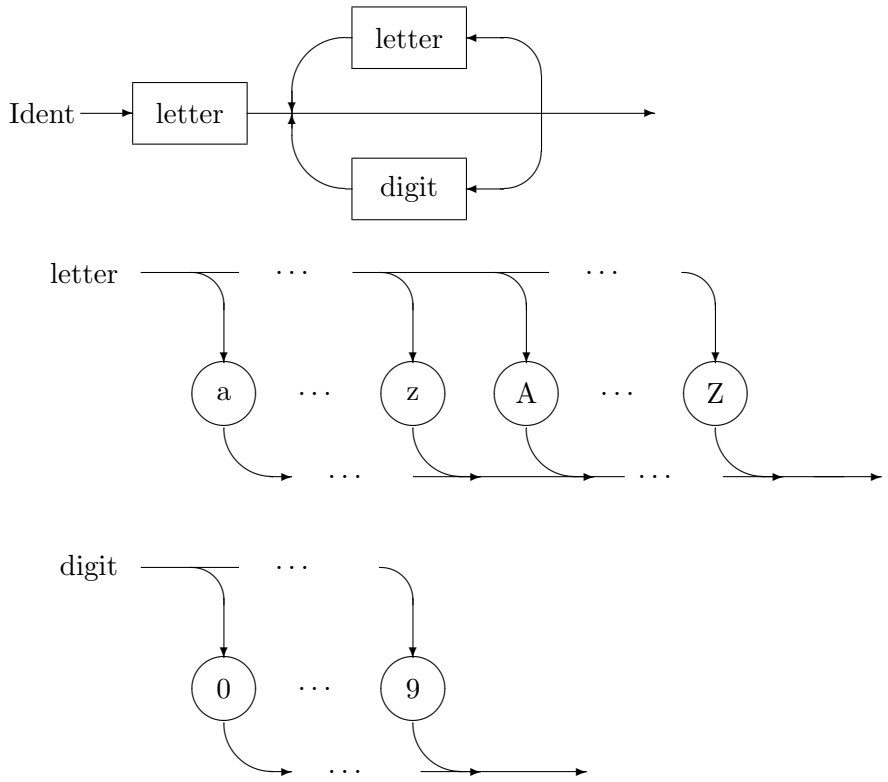
$$\overset{a_1 \cdot \ldots \cdot a_n}{\rightarrow} = \overset{a_1}{\rightarrow} \circ \cdots \circ \overset{a_n}{\rightarrow}$$

The sequence $q \overset{a_1}{\rightarrow} q_1 \quad \ldots \quad q_{n-1} \overset{a_n}{\rightarrow} q_n$ is also called *transitions sequence.*

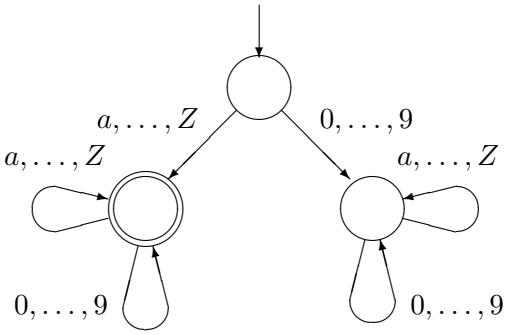
Example (from the field of compiler construction): A compiler for a programming language works in several phases, sometimes run in parallel:

- Lexical analysis : In this phase the so called *scanner* breaks the input text into the sequence of *tokens*. These are identifiers, keywords and delimiters.
- Syntax analysis : The generated token sequence is the input to the so called *parser*, which decides whether this sequence is a syntactically correct program.
- Code generation : The syntactic structure recognized by the parser is used to generate machine code.
- Optimisation : Execution time of the program should be improved mostly by local changes of machine code.

The lexical analysis is a simple task, which can be accomplished by finite automata. As an example let us consider the typical structure of identifiers in a programming language. We will consider a syntax diagram of MODULA



The identifiers constructed this way can be recognized by the following finite automaton:



For the sake of clarity we have labelled the edges of the state diagram with several symbols.

Example (from the field of operating systems): If several programs try to access shared resources in a multitasking environment, then these attempts need to be synchronized. For example, consider two programs P_1 and P_2 which share one printer. P_1 and P_2 should be synchronized in such a way that they do not send data to the printer simultaneously. We construct a finite automaton that would monitor the printer usage by P_1 and P_2.

- P_1 reports the beginning and the end of printing to the automaton by the symbols b_1 and e_1

- P_2 behaves similarly to P_1 and uses symbols b_2 and e_2 respectively.

At each point of time the behaviour of P_1 and P_2 regarding the printer is given by the finite string

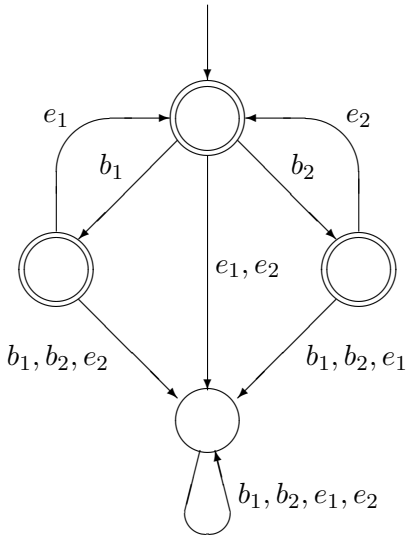
$$w \in \{b_1, e_1, b_2, e_2\}^*.$$

The automaton should accept this string, if

- every P_i uses the printer correctly, i. e., symbols b_i and e_i alternate in w, starting with b_i, $i = 1, 2$.

- P_1 and P_2 do not use the printer simultaneously, i. e., there is neither substring b_1b_2 nor b_2b_1 in string w.

For example, $w_1 = b_1e_1b_2e_2$ should be accepted, but $w_2 = b_1b_2e_1e_2$ should not be accepted. The following finite automaton fulfills the task:

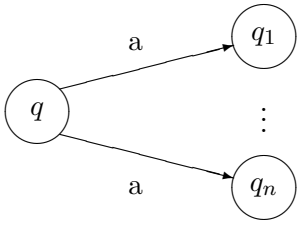


Non-determinism

In many applications finite automata can be simplified if *non-determinism* is allowed, i. e., we consider an *arbitrary* relation

$$\rightarrow \subseteq Q \times \Sigma \times Q$$

as transition relation. It may happen that for some $q \in Q, a \in \Sigma$ there are *several* successor states $q_1, \ldots, q_n$, so that (in graphical representation) it holds that



After reading a the automaton moves non-deterministically from q to one of the successor states $q_1, \ldots, q_n$. A special case is $n = 0$; then for $q \in Q$ and $a \in \Sigma$ there is no successor state q' with $q \xrightarrow{a} q'$. In this case the automaton *stops* and rejects symbol a. These remarks lead to the following definition.

1.3 **Definition** : A *non-deterministic finite automata* (or *acceptor*), in short NFA, is a 5-tuple

$$\mathcal{B} = (\Sigma, Q, \rightarrow, q_0, F),$$

where Σ, Q, q_0 and F are defined as in DFAs and for $\rightarrow$ it holds that:

$$\rightarrow \subseteq Q \times \Sigma \times Q.$$

Transitions are written ($q \xrightarrow{a} q'$) and extended to strings ($q \xrightarrow{w} q'$) as in DFAs.

A graphical representation by state diagram remains unchanged.

1.4 **Definition (acceptance and equivalence)**:

(i) The *language accepted* (or *recognized*) by NFA $\mathcal{B} = (\Sigma, Q, \to, q_0, F)$ is

$$L(\mathcal{B}) = \{w \in \Sigma^* \mid \exists\, q \in F : q_0 \overset{w}{\to} q\}.$$

By *NFA* we denote the class of languages accepted by NFAs.

(ii) Two NFAs $\mathcal{B}_1$ and $\mathcal{B}_2$ are called *equivalent* if $L(\mathcal{B}_1) = L(\mathcal{B}_2)$ holds.

An NFA recognizes a string w if while reading w it *can* reach one of the final states. It may be the case that, while reading w, there exist other transition sequences which end up in non-final states.

Obviously, a DFA is a special case of an NFA. Thus the equivalence between arbitrary finite automata is defined.

Example (suffix recognition): Consider an alphabet Σ and a string $v = a_1 \dots a_n \in \Sigma^*$ with $a_i \in \Sigma$ for $i = 1, \dots, n$.

We want to recognize the language

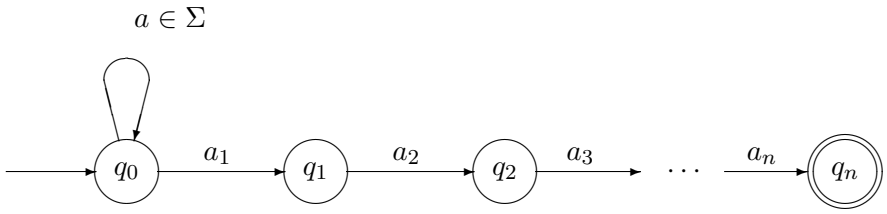
$$L_v = \{wv \mid w \in \Sigma^*\},$$

which consists of all strings over Σ with the suffix v.

For this purpose we consider the NFA

$$\mathcal{B}_v = (\Sigma, \{q_0, \dots, q_n\}, \to, q_0, \{q_n\}),$$

where $\to$ is defined by the following state diagram $\mathcal{B}_v$:



$\mathcal{B}_v$ behaves non-deterministically in the initial state q_0: while reading a string, $\mathcal{B}_v$ can decide in each occurrence of a_1 whether to try to recognize the suffix v. In order to do it, $\mathcal{B}_v$ goes to q_1 and now waits for $a_2 \dots a_n$ as a suffix. Should this not be the case, then $\mathcal{B}_v$ stops at some $i \in \{1, \dots, n\}$ and $a \neq a_i$.

The question arises: Do NFAs accept more languages than DFAs?

The answer is "no".

1.5 **Theorem (Rabin and Scott, 1959)**: For every NFA there exists an equivalent DFA.

Proof : Consider an NFA $\mathcal{B} = (\Sigma, Q, \to, q_0, F)$. We introduce a DFA $\mathcal{A}$ with $L(\mathcal{A}) = L(\mathcal{B})$ using the following *powerset construction*:

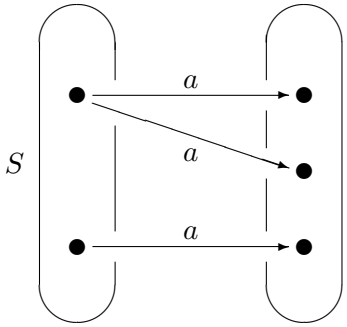
$$\mathcal{A} = (\Sigma, \mathbb{P}(Q), \to_{\mathcal{A}}, \{q_0\}, F_{\mathcal{A}}),$$

where for $S, S' \subseteq Q$ and $a \in \Sigma$ it holds that:

$$S \xrightarrow{a}_{\mathcal{A}} S' \text{ if and only if } S' = \{q' \in Q \mid \exists\, q \in S : q \xrightarrow{a} q'\},$$

Let the set of final states be $F_{\mathcal{A}} = \{S \subseteq Q \mid S \cap F \neq \emptyset\}$.

There exists a state in $\mathcal{A}$ for each subset S of the set of states Q of $\mathcal{B}$ (we denote such a state by S). $S \xrightarrow{a}_{\mathcal{A}} S'$ if and only if S' is the set of all successor states that can be reached by a-transitions of non-deterministic automaton $\mathcal{B}$ from states of S. It can be graphically represented by:



results in $S \xrightarrow{a}_{\mathcal{A}} S'$

We see that $\to_{\mathcal{A}}$ is deterministic, thus

$$\forall\, S \subseteq Q \quad \forall\, a \in \Sigma \quad \exists \text{ exactly one } S' \subseteq Q : S \xrightarrow{a}_{\mathcal{A}} S'.$$

Furthermore, for all $q, q' \in Q$, $S, S' \subseteq Q, a \in \Sigma$ the following holds:

(i) If $q \xrightarrow{a} q'$ and $q \in S$,
then $\exists\, S' \subseteq Q : \quad S \xrightarrow{a}_{\mathcal{A}} S'$ and $q' \in S'$.

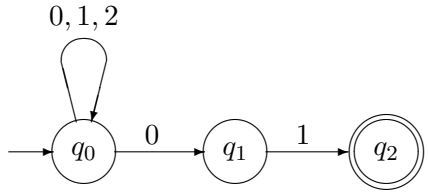
(ii) If $S \xrightarrow{a}_{\mathcal{A}} S'$ and $q' \in S'$,
then $\exists\, q \in Q : \quad q \xrightarrow{a} q'$ and $q \in S$.

This way we can easily show that $L(\mathcal{A}) = L(\mathcal{B})$. Let $w = a_1 \dots a_n \in \Sigma^*$ with $a_i \in \Sigma$ for $i = 1, \dots, n$. Then it holds that:

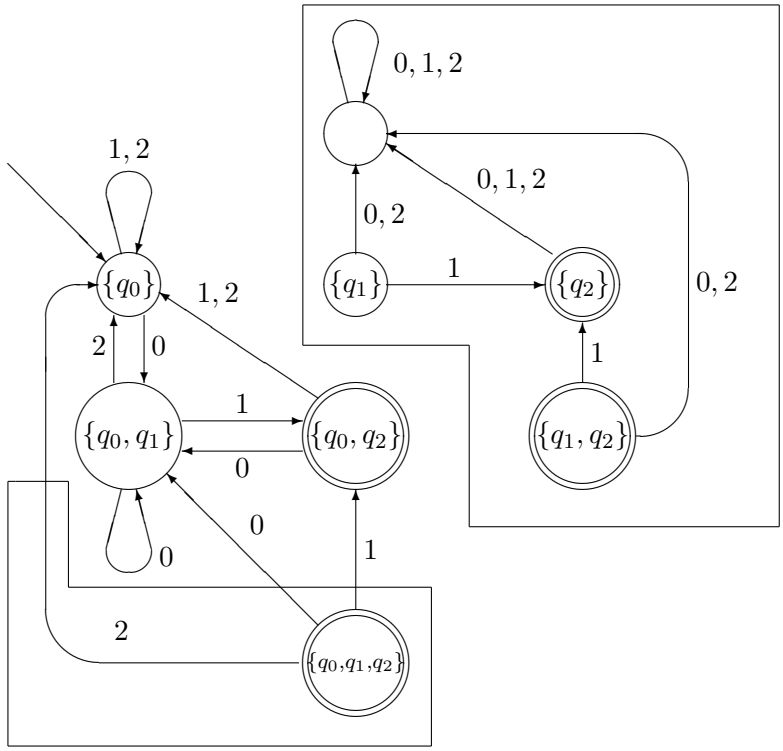
$$\begin{aligned}
w \in L(\mathcal{B}) \Leftrightarrow\ & \exists\, q \in F : \quad q_0 \xrightarrow{w} q \\
\Leftrightarrow\ & \exists\, q_1, \dots, q_n \in Q : \\
& q_0 \xrightarrow{a_1} q_1 \dots q_{n-1} \xrightarrow{a_n} q_n \text{ with } q_n \in F \\
\Leftrightarrow\ & \{\text{“}\Rightarrow\text{” follows from (i) and “}\Leftarrow\text{” follows from (ii)}\} \\
& \exists\, S_1, \dots, S_n \subseteq Q : \\
& \{q_0\} \xrightarrow{a_1}_{\mathcal{A}} S_1 \quad \dots \quad S_{n-1} \xrightarrow{a_n}_{\mathcal{A}} S_n \text{ with } S_n \cap F \neq \emptyset \\
\Leftrightarrow\ & \exists\, S \in F_A : \{q_0\} \xrightarrow{w}_{\mathcal{A}} S \\
\Leftrightarrow\ & w \in L(\mathcal{A}).
\end{aligned}$$

□

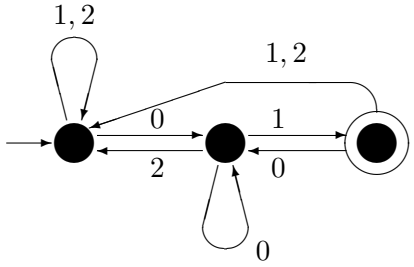
Example : We consider the suffix recognition of 01 in strings over the alphabet $\Sigma = \{0, 1, 2\}$. According to the previous example the language $L_{01} = \{w01 \mid w \in \Sigma^*\}$ is recognized by the NFA $\mathcal{B}_{01}$ with the following diagram:



Now we apply the powerset construction from the RABIN-SCOTT theorem to $\mathcal{B}_{01}$. As a result we get the following DFA $\mathcal{A}_{01}$:



The fragments in boxes □ are unreachable from the initial state $\{q_0\}$ of $\mathcal{A}_{01}$. Therefore, $\mathcal{A}_{01}$ can be simplified to the following equivalent DFA :



The number of states of this DFA can not be further minimized. There are examples, where the DFA given in the proof of the RABIN-SCOTT theorem cannot be further simplifed, i. e., if the given NFA has n states, then in the worst case the DFA actually needs 2^n states.

Spontaneous transitions

Automata can be defined even more conveniently, if ε-transitions in addition to non-determinism are allowed. These are the transitions, which an automaton makes spontaneously, without reading a symbol of the input string.

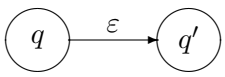
1.6 **Definition** : A *nondeterministic finite automaton (acceptor) with ε-transitions*, in short ε-NFA, is a 5-tuple

$$\mathcal{B} = (\Sigma, Q, \to, q_0, F),$$

where Σ, Q, q_0 and F are defined as in NFAs or DFAs and for the transition relation $\to$ it holds that:

$$\to \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q.$$

A transition $q \xrightarrow{\varepsilon} q'$ is called *ε-transition* and is represented in the state diagrams as follows:



In order to define the acceptance behaviour of ε-NFAs, we need an extended transition relation $q \overset{w}{\Rightarrow} q'$. For this purpose we make two terminological remarks.

- We define for every $\alpha \in \Sigma \cup \{\varepsilon\}$ a 2-place relation $\xrightarrow{\alpha}$ over Q, i. e., $\xrightarrow{\alpha} \subseteq Q \times Q$,

  $$\forall\, q, q' \in Q \text{ holds: } q \xrightarrow{\alpha} q' \Leftrightarrow (q, \alpha, q') \in \to .$$

  Here we use the standard infix notation for 2-place relations, i. e., $q \xrightarrow{\alpha} q'$ instead of $(q, q') \in \xrightarrow{\alpha}$. We call $\xrightarrow{\alpha}$ the *α-transition relation*.

- Therefore, we can use the composition $\circ$ for such transition relations. For all $\alpha, \beta \in \Sigma \cup \{\varepsilon\}$ we define $\xrightarrow{\alpha} \circ \xrightarrow{\beta}$ as follows: For $q, q' \in Q$ it holds that

  $$q \xrightarrow{\alpha} \circ \xrightarrow{\beta} q' \Leftrightarrow \exists\, q'' \in Q : q \xrightarrow{\alpha} q'' \text{ and } q'' \xrightarrow{\beta} q'.$$

1.7 **Definition** : Let $\mathcal{B} = (\Sigma, Q, \to, q_0, F)$ be an ε-NFA. Then a 2-place relation $\overset{w}{\Rightarrow} \subseteq Q \times Q$ is defined inductively for every string $w \in \Sigma^*$:

- $q \overset{\varepsilon}{\Rightarrow} q'$ if $\exists\, n \geq 0 :$ $\quad q \underbrace{\xrightarrow{\varepsilon} \circ \cdots \circ \xrightarrow{\varepsilon}}_{n\text{-times}} q'$

- $q \overset{aw}{\Rightarrow} q'$ if $q \overset{\varepsilon}{\Rightarrow} \circ \overset{a}{\to} \circ \overset{w}{\Rightarrow} q'$,

where $q, q' \in Q$, $a \in \Sigma$ and $w \in \Sigma^*$.

Remark : For all $q, q' \in Q$ and $a_1, \ldots, a_n \in \Sigma$:

(i) $q \overset{\varepsilon}{\Rightarrow} q$, but $q = q'$ does not follow from $q \overset{\varepsilon}{\Rightarrow} q'$.

(ii)
$$\begin{aligned} q \overset{a_1 \ldots a_n}{\Longrightarrow} q' &\Leftrightarrow q \overset{\varepsilon}{\Rightarrow} \circ \overset{a_1}{\to} \circ \overset{\varepsilon}{\Rightarrow} \cdots \circ \overset{\varepsilon}{\Rightarrow} \circ \overset{a_n}{\to} \circ \overset{\varepsilon}{\Rightarrow} q' \\ &\Leftrightarrow q \overset{a_1}{\Rightarrow} \circ \cdots \circ \overset{a_n}{\Rightarrow} q' \end{aligned}$$

(iii) It is decidable, whether the relation $q \overset{\varepsilon}{\Rightarrow} q'$ holds for the given states $q, q' \in Q$.

Proof : (i) and (ii) follow directly from the definition.

"(iii)": Let k be the number of states in Q. Then:

$$q \overset{\varepsilon}{\Rightarrow} q' \Leftrightarrow \exists\, n \leq k-1 : \quad q \underbrace{\overset{\varepsilon}{\to} \circ \cdots \circ \overset{\varepsilon}{\to}}_{n\text{-times}} q'.$$

If at least k transitions $\overset{\varepsilon}{\to}$ take place one after the other, then the states repeat in the path from q to q'. So in order to find all states reachable by ε-transitions from q, it is sufficient to check the finitely many sequences with maximum $k-1$ transitions $\overset{\varepsilon}{\to}$. Thus the decidability of $q \overset{\varepsilon}{\Rightarrow} q'$ is proved. □

1.8 **Definition (acceptance)**: Let $\mathcal{B} = (\Sigma, Q, \to, q_0, F)$ be a ε-NFA.

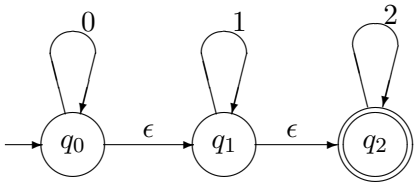
(i) The *language accepted* (or *recognized*) by $\mathcal{B}$ is

$$L(\mathcal{B}) = \{w \in \Sigma^* \mid \exists\, q \in F : q_0 \overset{w}{\Rightarrow} q\}.$$

(ii) Two ε-NFAs $\mathcal{B}_1$ and $\mathcal{B}_2$ are called *equivalent* if $L(\mathcal{B}_1) = L(\mathcal{B}_2)$.

Obviously NFAs are a special case of ε-NFAs. Therefore, the equivalence of NFAs and ε-NFAs can be defined.

Example : For the input alphabet $\Sigma = \{0, 1, 2\}$ we consider the ε-NFA $\mathcal{B}$ defined by the following state diagram:



Then:

$$L(\mathcal{B}) \;=\; \{w \in \{0,1,2\}^* \mid \exists\, k,l,m \geq 0 : w \;=\; \underbrace{0\ldots0}_{k\text{-times}}\ \underbrace{1\ldots1}_{l\text{-times}}\ \underbrace{2\ldots2}_{m\text{-times}}\ \}$$

1.9 **Theorem** : For every ε-NFA there exists an equivalent NFA.

Proof : Consider a ε-NFA $\mathcal{B} \;=\; (\Sigma, Q, \to, q_0, F)$.

We construct the following NFA $\mathcal{A} \;=\; (\Sigma, Q, \to_{\mathcal{A}}, q_0, F_{\mathcal{A}})$, where for $q, q' \in Q$ and $a \in \Sigma$ it holds that:

- $q \overset{a}{\to}_{\mathcal{A}} q'$ if and only if $q \overset{a}{\Rightarrow} q'$ in $\mathcal{B}$, therefore $\overset{\varepsilon}{\Rightarrow} \circ \overset{a}{\to} \circ \overset{\varepsilon}{\Rightarrow} q'$ in $\mathcal{B}$
- $q \in F_{\mathcal{A}}$ if and only if $\exists\, q' \in F : \quad q \overset{\varepsilon}{\Rightarrow} q'$

By definition $\mathcal{A}$ is a NFA without ε-transitions with $F \subseteq F_{\mathcal{A}}$. It remains to show that: $L(\mathcal{A}) = L(\mathcal{B})$. Let $w \;=\; a_1 \ldots a_n \in \Sigma^*$ with $n \geq 0$ and $a_i \in \Sigma$ for $i \;=\; 1, \ldots, n$. Then:

$$\begin{aligned}
w \in L(\mathcal{A}) \Leftrightarrow\ & \exists\, q \in F_{\mathcal{A}} : \quad q_0 \overset{w}{\to}_{\mathcal{A}} q \\
\Leftrightarrow\ & \exists\, q \in F_{\mathcal{A}} : \quad q_0 \overset{a_1}{\to}_{\mathcal{A}} \circ \cdots \circ \overset{a_n}{\to}_{\mathcal{A}} q \\
\Leftrightarrow\ & \exists\, q \in F_{\mathcal{A}} : \quad q_0 \overset{a_1}{\Rightarrow} \circ \cdots \circ \overset{a_n}{\Rightarrow} q \\
\Leftrightarrow\ & \{\text{“}\Rightarrow\text{”: choose } q' \text{ with } q \overset{\varepsilon}{\Rightarrow} q'. \\
& \quad \text{“}\Leftarrow\text{”: choose } q \;=\; q'.\} \\
& \exists\, q' \in F : \quad q \overset{a_1}{\Rightarrow} \circ \cdots \circ \overset{a_n}{\Rightarrow} q' \\
\Leftrightarrow\ & \exists\, q' \in F : \quad q \overset{w}{\Rightarrow} q' \\
\Leftrightarrow\ & w \in L(\mathcal{B})
\end{aligned}$$

In the special case $w \;=\; \varepsilon$ the above argument is reduced as follows:

$$\begin{aligned}
\varepsilon \in L(\mathcal{A}) \Leftrightarrow\ & \exists\, q_0 \in F_{\mathcal{A}} \\
\Leftrightarrow\ & \exists\, q \in F : \quad q_0 \overset{\varepsilon}{\Rightarrow} q \\
\Leftrightarrow\ & \varepsilon \in L(\mathcal{B})
\end{aligned}$$

□

Remarks :

(i) In the proof $F_{\mathcal{A}}$ could also be defined as follows:

$$F_{\mathcal{A}} \;=\; \begin{cases} F \cup \{q_0\} & \text{if} \quad \exists\, q \in F : \quad q_0 \overset{\varepsilon}{\Rightarrow} q \\ F & \text{otherwise} \end{cases}$$

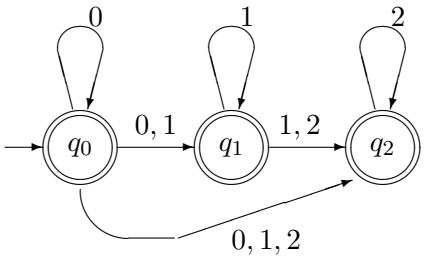
Compare with the proof in Hopcroft & Ullman.

(ii) NFA $\mathcal{A}$ can be computed from the given ε-NFA $\mathcal{B}$, because the relation $q \overset{\varepsilon}{\Rightarrow} q'$ is decidable:

Compare with the previous remark.

(iii) If there are ε-cycles in ε-NFA $\mathcal{B}$, then the set of states of NFA $\mathcal{A}$ can be reduced by the preceding contraction of ε-cycles; every ε-cycle can be replaced by one state.

Example : We apply the construction introduced in the proof to the ε-NFA $\mathcal{B}$ from the previous example. As the result we have the following NFA $\mathcal{A}$:



It is easy to check that $L(\mathcal{A}) = L(\mathcal{B})$ actually holds.

Therefore, we have the following result:

$$\text{DEA} = \text{NEA} = \varepsilon\text{-NEA},$$

i. e., the classes of languages accepted by DFAs, NFAs and ε-NFAs coincide; we call it the *class of finitely accepted languages*. If we want to show the properties of this class, we can choose the type of automaton most suited for this purpose.

§2 Closure properties

Now we explore under which operations the class of finitely accepted languages is closed. For this purpose we consider the set operations union, intersection, complement, difference as well as operations on languages such as concatenation and iteration (KLEENE star operator), which were introduced in Chapter I.

2.1 **Theorem** : The class of finitely acceptable languages is closed under the following operations:

1. union,
2. complement,
3. intersection,
4. difference,
5. concatenation,
6. iteration.

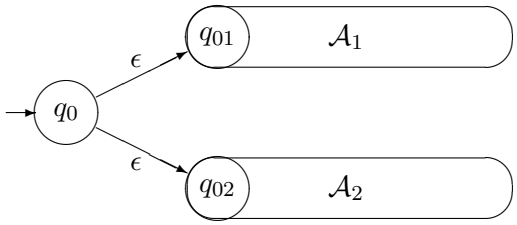
Proof : Let $L_1, L_2 \subseteq \Sigma^*$ be finitely acceptable. Then there are DFAs $A_i = (\Sigma, Q_i, \to_i, q_{0i}, F_i)$ with $L_i = L(\mathcal{A}_i)$, $i = 1, 2$, and $Q_1 \cap Q_2 = \emptyset$. We show that $L_1 \cup L_2, \overline{L_1}, L_1 \cap L_2, L_1 \backslash L_2, L_1 \cdot L_2$

and L_1^* are finitely acceptable. For $L_1 \cup L_2, L_1 \cdot L_2$ and L_1^* we shall provide the accepting ε-NFAs. This makes the task less complicated.

1. $L_1 \cup L_2$: Let us construct the ε-NFA $\mathcal{B} = (\Sigma, \{q_0\} \cup Q_1 \cup Q_2, \rightarrow, q_0, F_1 \cup F_2)$, where $q_0 \notin Q_1 \cup Q_2$ and

$$\rightarrow = \{(q_0, \varepsilon, q_{01}), (q_0, \varepsilon, q_{02})\} \cup \rightarrow_1 \cup \rightarrow_2$$

   hold. $\mathcal{B}$ is graphically represented as follows:



   It is clear that $L(\mathcal{B}) = L_1 \cup L_2$ holds.

2. $\overline{L_1}$: Consider the DFA $\mathcal{A} = (\Sigma, Q_1, \rightarrow_1, q_{01}, Q_1 \backslash F_1)$. Then for all $w \in \Sigma^*$ it holds that:

$$\begin{array}{rcl} w \in L(\mathcal{A}) & \Leftrightarrow & \exists\, q \in Q_1 \backslash F_1 :\ q_{01} \overset{w}{\rightarrow}_1 q \\ & \{\rightarrow_1 \text{ determ.}\} & \\ & \Leftrightarrow & \neg\exists\, q \in F_1 :\ q_{01} \overset{w}{\rightarrow}_1 q \\ & \Leftrightarrow & w \notin L(\mathcal{A}_1) \\ & \Leftrightarrow & w \notin L_1 \end{array}$$

   Therefore, $L(\mathcal{A}) = \overline{L_1}$ holds.

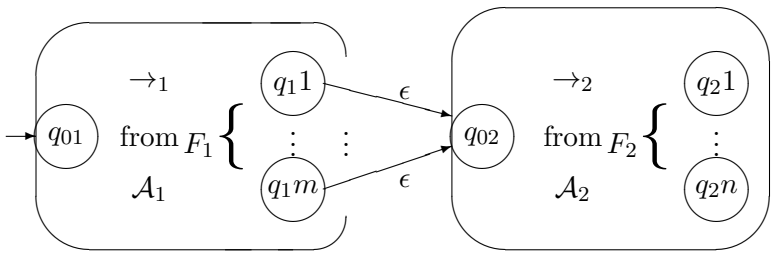
3. $L_1 \cap L_2$: is obvious, because $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$ holds.

4. $L_1 \backslash L_2$: is obvious, because $L_1 \backslash L_2 = L_1 \cap \overline{L_2}$ holds.

5. $L_1 \cdot L_2$: Construct the ε-NFA $\mathcal{B} = (\Sigma, Q_1 \cup Q_2, \rightarrow, q_{01}, F_2)$ with

$$\rightarrow = \rightarrow_1 \cup \{(q, \varepsilon, q_{02}) \mid q \in F_1)\} \cup \rightarrow_2 .$$

   $\mathcal{B}$ is graphically represented as follows:

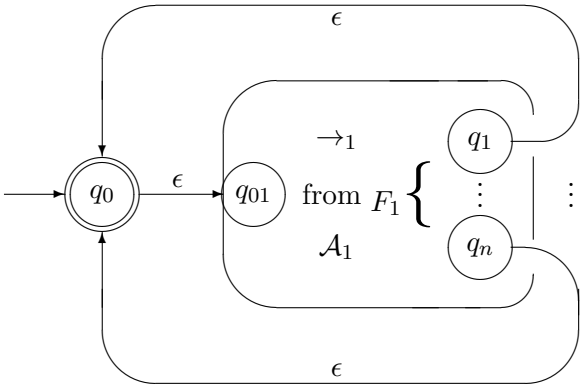


   It is easy to show that $L(\mathcal{B}) = L_1 \cdot L_2$ holds.

6. L_1^*: For the iteration we construct the ε-NFA
   $\mathcal{B} = (\Sigma, \{q_0\} \cup Q_1, \to, q_0, \{q_0\})$, where $q_0 \notin Q_1$ and

$$\to = \{(q_0, \varepsilon, q_{01})\} \cup \to_1 \cup \{(q, \varepsilon, q_0) \mid q \in F_1\}$$

   hold. $\mathcal{B}$ is graphically represented as follows:



   Again it is easy to show that $L(\mathcal{B}) = L_1^*$ holds.

Thus the proof is complete. □

Remark : There is also an interesting direct construction of accepting DFAs for the union and intersection . Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be as in the above proof. Then we consider the following transition relation $\to \subseteq Q \times \Sigma \times Q$ on the Cartesian product $Q = Q_1 \times Q_2$ of the sets of states:

for all $q_1, q_1' \in Q_1$ and $q_2, q_2' \in Q_2$ and $a \in \Sigma$ it holds that

$$(q_1, q_2) \xrightarrow{a} (q_1', q_2') \text{ if and only if } q_1 \xrightarrow{a}_1 q_1' \text{ and } q_2 \xrightarrow{a}_2 q_2'.$$

The relation $\xrightarrow{a}$ models the *simultaneous parallel progress* of automata $\mathcal{A}_1$ and $\mathcal{A}_2$ while reading the symbol a.

Consider the DFAs

$$A_\cup = (\Sigma, Q, \to, (q_{01}, q_{02}), F_\cup),$$
$$A_\cap = (\Sigma, Q, \to, (q_{01}, q_{02}), F_\cap)$$

with

$$F_\cup = \{(q_1, q_2) \in Q \mid q_1 \in F_1 \text{ or } q_2 \in F_2\},$$
$$F_\cap = \{(q_1, q_2) \in Q \mid q_1 \in F_1 \text{ and } q_2 \in F_2\}.$$

Then it holds that $L(\mathcal{A}_\cup) = L_1 \cup L_2$ and $L(\mathcal{A}_\cap) = L_1 \cap L_2$.

Proof : We show that the statement for $\mathcal{A}_\cap$ holds. For any $w \in \Sigma^*$ it holds that:

$$\begin{aligned} w \in L(\mathcal{A}_\cap) &\Leftrightarrow \exists\ (q_1, q_2) \in F_\cap :\ (q_{01}, q_{02}) \xrightarrow{w} (q_1, q_2) \\ &\Leftrightarrow \exists\ q_1 \in F_1, q_2 \in F_2 :\ q_{01} \xrightarrow{w}_1 q_1 \text{ and } q_{02} \xrightarrow{w}_2 q_2 \\ &\Leftrightarrow w \in L(\mathcal{A}_1) \text{ and } w \in L(\mathcal{A}_2) \\ &\Leftrightarrow w \in L_1 \cap L_2 \end{aligned}$$

□

§3 Regular expressions

With the help of regular expressions we can inductively describe the finitely acceptable languages. For this purpose we consider a fixed alphabet Σ.

3.1 **Definition (regular expressions and languages)**:

1. The *syntax of regular expressions over* Σ is given as follows:
   - $\emptyset$ and ε are regular expressions over Σ.
   - a is a regular expression over Σ for every $a \in \Sigma$.
   - If re, re_1, re_2 are regular expressions over Σ, then $(re_1 + re_2), (re_1 \cdot re_2), re^*$ are also regular expressions over Σ.
2. The *semantics of a regular expression* re *over* Σ is the language $L(re) \subseteq \Sigma^*$, which is inductively defined as follows:
   - $L(\emptyset) = \emptyset$
   - $L(\varepsilon) = \{\varepsilon\}$
   - $L(a) = \{a\}$ for $a \in \Sigma$
   - $L((re_1 + re_2)) = L(re_1) \cup L(re_2)$
   - $L((re_1 \cdot re_2)) = L(re_1) \cdot L(re_2)$
   - $L(re^*) = L(re)^*$
3. A language $L \subseteq \Sigma^*$ is called *regular* if there is a regular expression re over Σ with $L = L(re)$.

In order to save space by omitting some brackets we define priorities for the operators:

$$* \text{ binds stronger than } \cdot \text{ and } \cdot \text{ binds stronger than } +.$$

Besides we omit the outer brackets and use the associativity of $\cdot$ and $+$. The concatenation dot $\cdot$ is often omitted.

Example : We use regular expressions to describe some previously considered languages.

1. The language of identifiers considered by the lexical analysis is described by the regular expression
$$re_1 = (a + \ldots + Z)(a + \ldots + Z + 0 + \ldots + 9)^*.$$
2. The language over $\{b_1, e_1, b_2, e_2\}$ used for synchronization of two programs which share a printer is described by the following regular expression
$$re_2 = (b_1 e_1 + b_2 e_2)^*(\varepsilon + b_1 + b_2).$$

3. Let $\Sigma = \{a_1, \dots a_n, b_1, \dots, b_m\}$ and $v = a_1 \dots a_n$. Then the language $L_v = \{wv \mid w \in \Sigma^*\}$ of the strings with suffix v is described by the regular expression

$$re_3 = (a_1 + \dots a_n + b_1 + \dots + b_m)^* a_1 \dots a_n.$$

It holds that: $L(re_3) = L_v$.

Now we will show that a more general statement holds:

3.2 **Theorem (**Kleene**)**: A language is regular if and only if it is finitely acceptable.

Proof : We consider a language $L \subseteq \Sigma^*$.

"$\Rightarrow$": For a regular expression re over Σ it holds that $L = L(re)$. We show using induction over the structure of re that $L(re)$ is finitely acceptable.

Basis: $L(\emptyset)$, $L(\varepsilon)$ and $L(a)$ are obviously finitely acceptable for $a \in \Sigma$.

Induction step: Let $L(re)$, $L(re_1)$ and $L(re_2)$ be finitely acceptable. Then $L(re_1{+}re_2)$, $L(re_1{\cdot}re_2)$ and $L(re^*)$ are also finitely acceptable, because the class of finitely acceptable languages is closed under union, concatenation and iteration.

"$\Leftarrow$": It holds that $L = L(\mathcal{A})$ for a DFA $\mathcal{A}$ with n states. W.l.o.g. $\mathcal{A} = (\Sigma, Q, \to, 1, F)$, where $Q = \{1, \dots, n\}$. For $i, j \in \{1, \dots, n\}$ and $k \in \{0, 1, \dots, n\}$ we define

$$L^k_{i,j} \;=\; \{w \in \Sigma^* \mid \; i \xrightarrow{w} j \text{ and } \forall\, u \in \Sigma^*, \forall\, l \in Q : \\ \text{from } (\exists\, v \;:\; v \neq \varepsilon,\; v \neq w \text{ and } uv = w) \text{ and } i \xrightarrow{u} l \text{ holds } l \leq k\}.$$

$L^k_{i,j}$ consists of all strings w such that automaton $\mathcal{A}$ moves from state i to state j while reading the string w. Furthermore, while reading the string w, only states with the number less or equal k occur.

Now we show by induction on k that the languages $L^k_{i,j}$ are all regular.

$k = 0$: For strings from $L^0_{i,j}$ no intermediate state can be used. Thus it holds that:

$$L^0_{i,j} \;=\; \begin{cases} \{a \in \Sigma \mid i \xrightarrow{a} j\} & \text{if } i \neq j \\ \{\varepsilon\} \cup \{a \in \Sigma \mid i \xrightarrow{a} j\} & \text{if } i = j \end{cases}$$

Therefore, $L^0_{i,j}$ is regular as a finite subset of $\Sigma \cup \{\varepsilon\}$.

$k \to k{+}1$: Let the languages $L^k_{i,j}$ be regular for all $i, j \in \{1, \dots, n\}$. Then for all $i, j \in \{1, \dots, n\}$ it holds that:

$$L^{k+1}_{i,j} = L^k_{i,j} \cup L^k_{i,k+1} \cdot (L^k_{k+1,k+1})^* \cdot L^k_{k+1,j},$$

In order to go from the state i to the state j, the intermediate state $k+1$ is either unnecessary, then $L^k_{i,j}$ is sufficient for the description; or the state $k+1$ is used as an intermediate state one or several times, then $L^k_{i,k+1} \cdot (L^k_{k+1,k+1})^* \cdot L^k_{k+1,j}$ is used for description. Therefore, by induction on k we get:

$$L^{k+1}_{i,j} \text{ is regular.}$$

From the regularity of languages $L_{i,j}^k$ we conclude that L itself is regular, because it holds that

$$L = L(\mathcal{A}) = \bigcup_{j \in F} L_{1,j}^n.$$

Thus we have proved the statement of KLEENE. □

Remark : As an alternative to the above construction of a regular expression from a given finite automaton we can solve guarded regular systems of equations.

§4 Structural properties of regular languages

As regular languages are exactly the finitely acceptable languages, we can derive important characteristics of regular languages from the finiteness of the sets of states of the accepting automata. First we consider the so called pumping lemma, which gives us a necessary condition for a language to be regular. Let Σ be an arbitrary alphabet.

4.1 **Theorem (the pumping lemma for regular languages)**: For every regular language $L \subseteq \Sigma^*$ there exists a number $n \in \mathbb{N}$, so that for all strings $z \in L$ with $|z| \geq n$ there is a decomposition $z = uvw$ with $v \neq \varepsilon$ and $|uv| \leq n$ and for all $i \in \mathbb{N}$ it holds that $uv^iw \in L$, i. e., we can "pump" the substring v and the resulting string will be in the regular language L.

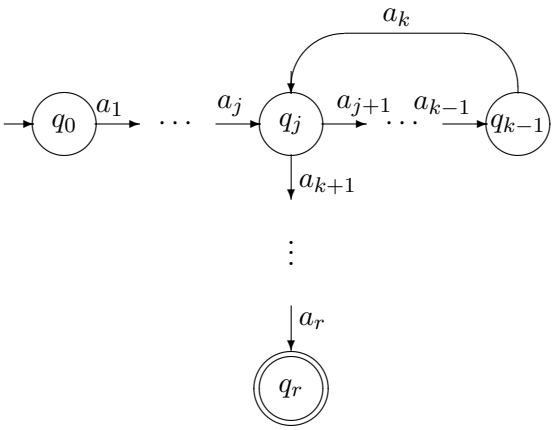
In quantifier notation:

$\forall$ regular $L \subseteq \Sigma^* \; \exists \; n \in \mathbb{N} \; \forall \; z \in L$ with $|z| \geq n$
$\exists \; u, v, w \in \Sigma^* : \; z = uvw$ and $v \neq \varepsilon$ and $|uv| \leq n$ and $\forall \; i \in \mathbb{N} : \; uv^iw \in L$

Proof : Let $L \subseteq \Sigma^*$ be regular.

According to the KLEENE theorem there is a DFA $\mathcal{A} = (\Sigma, Q, \to, q_0, F)$ with $L = L(\mathcal{A})$. We choose $n = |Q|$ and consider a string $z \in L$ with $|z| \geq n$. Then $\mathcal{A}$ must go through at least one state twice while reading z. More precisely it holds that:

Let $z = a_1 \dots a_r$ with $r = |z| \geq n$ and $a_i \in \Sigma$ for $i = 1, \dots, r$ and let $q_1, \dots, q_r \in Q$ be defined as follows: $q_{i-1} \xrightarrow{a_i} q_i$ for $i = 1, \dots, r$. Then there is $j, k \in \{1, \dots, n\}$ with $0 \leq j < k \leq n \leq r$, so that $q_j = q_k$ holds. In graphical representation:



Let $u = a_1 \dots a_j$, $v = a_{j+1} \dots a_k$, $w = a_{k+1} \dots a_r$. Then according to the properties of j and k it holds that $v \neq \varepsilon$ and $|uv| \leq n$. Besides, it is clear that the automaton $\mathcal{A}$, while recognizing, can go through the q_j-loop any number of times, i. e., for all $i \in \mathbb{N}$ it holds that: $uv^i w \in L$. $\square$

We consider a typical application of the pumping lemma, where we prove that a certain language is *not* regular.

Example : The language $L = \{a^n b^n | \ n \in \mathbb{N}\}$ is not regular. We provide a proof by contradiction.

Hypothesis: L is regular. According to the pumping lemma, there is an $n \in \mathbb{N}$ with given properties. Now consider $z = a^n b^n$. Since $|z| \geq n$ holds, we can break z into $z = uvw$ with $v \neq \varepsilon$ and $|uv| \leq n$, so that for all $i \in \mathbb{N}$ it holds: $uv^i w \in L$. However, v consists only of symbols a, so that in particular $uw = a^{n-|v|} b^n \in L$ should hold. *Contradiction*

The above example shows that finite automata cannot count unlimitedly. We shall get acquainted with other applications of the pumping lemma in the section devoted to decidability properties.

NERODE **relation**

If we consider the so called NERODE relation, we get a characteristic, i. e., necessary and sufficient condition for the regularity of languages $L \subseteq \Sigma^*$.

4.2 **Definition** :

Let $L \subseteq \Sigma^*$ be some language. The NERODE *relation* of L is a 2-place relation $\equiv_L$ on Σ^*, i. e., $\equiv_L \subseteq \Sigma^* \times \Sigma^*$, which is defined as follows for $u, v \in \Sigma^*$:

$u \equiv_L v$ if and only if for all $w \in \Sigma^*$ it holds that $uw \in L \Leftrightarrow vw \in L$.

Therefore, it holds that $u \equiv_L v$ if u and v can be extended in the same way to strings from L. In particular, $u \in L \Leftrightarrow v \in L$ (with $w = \varepsilon$).

Remark : The NERODE relation is a right congruence, i. e., it has the following properties:

1. $\equiv_L$ is an equivalence relation on Σ^*, i. e., reflexive, symmetric and transitive.

2. $\equiv_L$ is right compatible with the concatenation, i. e., from $u \equiv_L v$ it follows $uw \equiv_L vw$ for all $w \in \Sigma^*$

Given that $\equiv_L$ is an equivalence relation, we can determine the *index of* $\equiv_L$, i. e., the number of equivalence classes of $\equiv_L$.

4.3 **Theorem** (Myhill and Nerode):

A language $L \subseteq \Sigma^*$ is regular if and only if $\equiv_L \subseteq \Sigma^* \times \Sigma^*$ has a finite index.

Proof : "⇒": Let L be regular, i. e., $L = L(\mathcal{A})$ for a DFA $\mathcal{A} = (\Sigma, Q, \to, q_0, F)$. We introduce the following 2-place relation $\equiv_\mathcal{A}$ on Σ^*. For $u, v \in \Sigma^*$ it holds that:

$u \equiv_\mathcal{A} v$ if and only if we have a $q \in Q$, with $q_0 \xrightarrow{u} q$ and $q_0 \xrightarrow{v} q$,

i. e., if the automaton $\mathcal{A}$ moves from q_0 to the same state q after having read the inputs u and v. Note that q is uniquely defined, because $\mathcal{A}$ is deterministic. The relation $\equiv_\mathcal{A}$ is an equivalence relation on Σ^*.

We show: $\equiv_\mathcal{A}$ is a *refinement* of $\equiv_L$, i. e., for all $u, v \in \Sigma^*$ it holds that:

$$\text{For } u \equiv_\mathcal{A} v \text{ it follows that } u \equiv_L v.$$

Let $u \equiv_\mathcal{A} v$ and $w \in \Sigma^*$. Then it holds that:

$$\begin{aligned} uw \in L &\Leftrightarrow \exists\, q \in Q\ \exists\, q' \in F : q_0 \xrightarrow{u} q \xrightarrow{w} q' \\ &\Leftrightarrow \{u \equiv_\mathcal{A} v\} \\ &\quad \exists\, q \in Q, q' \in F : q_0 \xrightarrow{v} q \xrightarrow{w} q' \\ &\Leftrightarrow vw \in L. \end{aligned}$$

Thus there are at least as many equivalence classes of $\equiv_\mathcal{A}$ as of $\equiv_L$. Therefore, it holds that

$$\begin{aligned} & Index(\equiv_L) \\ \leq\ & Index(\equiv_\mathcal{A}) \\ =\ & \text{number of states reachable from } q_0 \\ \leq\ & |Q|, \end{aligned}$$

thus $\equiv_L$ has a finite index.

"⇐": Let $L \subseteq \Sigma^*$ be a language and $k \in \mathbb{N}$ the finite index of $\equiv_L$. We choose k strings $u_1, \ldots, u_k \in \Sigma^*$ with $u_1 = \varepsilon$ as representatives of the equivalence class of $\equiv_L$. Then Σ^* can be represented as a disjoint union of these equivalence classes:

$$\Sigma^* = [u_1] \,\dot\cup \ldots \dot\cup\, [u_k].$$

In particular, for every string $u \in \Sigma^*$ there is an $i \in \{1, \ldots, k\}$ with $[u] = [u_i]$.

Now we construct the following *equivalence-class automaton*

$\mathcal{A}_L = (\Sigma, Q_L, \to_L, q_L, F_L)$:

$$\begin{aligned} Q_L &= \{[u_1], \ldots, [u_k]\}, \\ q_L &= [u_1] = [\varepsilon] \\ F_L &= \{[u_j] \mid u_j \in L\}, \end{aligned}$$

and for $i, j \in \{1, \ldots, k\}$ and $a \in \Sigma$ let

$$[u_i] \xrightarrow{a}_L [u_j] \text{ if and only if } [u_j] = [u_i a].$$

Then $\mathcal{A}_L$ is a DFA and for all strings $w \in \Sigma^*$ it holds that:

$$[\varepsilon] \xrightarrow{w}_L [u_j] \text{ if and only if } [u_j] = [w],$$

more precisely for $w = a_1 \ldots a_n$

$$[\varepsilon] \xrightarrow{w}_L [u_j] \text{ if and only if } [\varepsilon] \xrightarrow{a_1}_L [a_1] \ldots \xrightarrow{a_n}_L [a_1 \ldots a_n] = [u_j],$$

and thus

$$\begin{aligned} & w \in L(\mathcal{A}_L) = \\ \Leftrightarrow \quad & \exists\, [u_j] \in F_L : [\varepsilon] \xrightarrow{w}_L [u_j] \\ \Leftrightarrow \quad & \exists\, u_j \in L : [u_j] = [w] \\ \Leftrightarrow \quad & w \in L \end{aligned}$$

So $\mathcal{A}_L$ accepts the language L. Therefore, L is regular. □

We use the method of proof of the MYHILL-NERODE theorem in order to minimize the number of states of a DFA. We mainly refer to the deterministic equivalence-class automaton $\mathcal{A}_L$ from the proof.

4.4 **Corollary** : Let $L \subseteq \Sigma^*$ be regular and $k = Index(\equiv_L)$. Then every DFA that accepts L has at least k states. The minimal number k is reached by the DFA $\mathcal{A}_L$. However, there may exist NFAs with less than k states accepting L.

Proof : In the proof of the MYHILL-NERODE theorem we have shown in "⇒" that every DFA $\mathcal{A} = (\Sigma, Q, \to, q_0, F)$ accepting L has at least k states. At the same time $k = Index(\equiv_L) \leq |Q|$. In "⇐" we have constructed the DFA $\mathcal{A}_L$ with k states accepting L. □

The equivalence-class automaton $\mathcal{A}_L$ is the prototype of all DFAs accepting L with minimal number of states k. We can show that every other DFA accepting L and that has k states is isomorphic to $\mathcal{A}_L$, i. e., we can get it from $\mathcal{A}_L$ by a bijective renaming of the states.

4.5 **Definition** : Two DFAs or NFAs $\mathcal{A}_i = (\Sigma, Q_i, \to_i, q_{0i}, F_i), i = 1, 2$, are called *isomorphic* if there is a bijection $\beta : Q_1 \to Q_2$ with the following properties:

- $\beta(q_{01}) = q_{02}$,

- $\beta(F_1) = \{\beta(q) \mid q \in F_1\} = F_2$,
- $\forall\, q, q' \in Q_1\ \forall\, a \in \Sigma : q \xrightarrow{a}_1 q' \Leftrightarrow \beta(q) \xrightarrow{a}_2 \beta(q')$.

The bijection β is called the *isomorphism* from $\mathcal{A}_1$ to $\mathcal{A}_2$.

Note that isomorphism is an equivalence relation on finite automata. Now we will show the announced statement.

4.6 **Theorem** : Let $L \subseteq \Sigma^*$ be regular and $k = Index(\equiv_L)$. Then every DFA $\mathcal{A}$ that accepts L and that has k states is isomorphic to $\mathcal{A}_L$.

Proof : We have $\mathcal{A} = (\Sigma, Q, \rightarrow, q_1, F)$ with $L(\mathcal{A}) = L$ and $|Q| = k$ and the equivalence-class automaton from the Myhill-Nerode theorem with $Q_L = \{[u_1], \ldots, [u_k]\}$ and $u_1 = \varepsilon$. For every $i \in \{1, \ldots, k\}$ we define the state $q_i \in Q$ by transition $q_1 \xrightarrow{u_i} q_i$.

Note that q_i is uniquely defined, because the transition relation $\rightarrow$ is deterministic.

Now we define the mapping $\beta : Q_L \rightarrow Q$ by

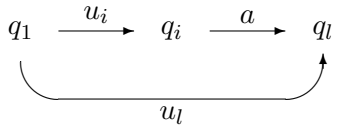
$$\beta([u_i]) = q_i$$

for $i \in 1, \ldots, k$ and show that β is an isomorphism from $\mathcal{A}_L$ to $\mathcal{A}$.

1. β is injective: Let $q_i = q_j$. Then it holds that $q_1 \xrightarrow{u_i} q_i$ and $q_1 \xrightarrow{u_j} q_i$. Therefore, for all $w \in \Sigma^*$ it holds that $u_i w \in L \Leftrightarrow u_j w \in L$. Therefore, $u_i \equiv_L u_j$ and $[u_i] = [u_j]$.
2. β is surjective: this property follows from (1) and the fact that $k = |Q|$. Thus it holds in particular that $Q = \{q_1, \ldots, q_k\}$.
3. $\beta([q_L]) = \beta([u_1]) = \beta([\varepsilon]) = q_1$
4. $\beta(F_L) = F : [u_j] \in F_L \Leftrightarrow u_j \in L \Leftrightarrow q_j \in F$
5. For all $i, j \in \{1, \ldots, k\}$ and $a \in \Sigma$ it holds that:

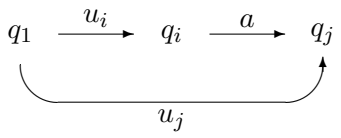
$$[u_i] \xrightarrow{a}_L [u_j] \Leftrightarrow q_i \xrightarrow{a} q_j.$$

- Proof of "⇒": Let $[u_i] \xrightarrow{a}_L [u_j]$. Then by definition of $\rightarrow_L$: $[u_i a] = [u_j]$. There is an $l \in \{1, \ldots, k\}$ with $q_i \xrightarrow{a} q_l$. By definition of q_i and q_l we have the following image:



  Given that the strings $u_i a$ and u_l lead to the same state q_l, it follows that $u_i a \equiv_L u_l$. Therefore, it holds that $[u_j] = [u_i a] = [u_l]$. According to the choice of $u_1, \ldots, u_k$ in A_L it follows that $u_j = u_l$, even $j = l$. Thus $q_i \xrightarrow{a} q_j$ follows as desired.

- Proof of "⇐": Let $q_i \xrightarrow{a} q_j$. Thus we have the following image similar to the one above:



  Hence we make a conclusion as above: $u_i a \equiv_L u_j$. Therefore, it holds that $[u_i a] = [u_j]$; by definition of $\rightarrow_L$ we can derive that $[u_i] \xrightarrow{a}_L [u_j]$.

Thus $\mathcal{A}_L$ and $\mathcal{A}$ are isomorphic. □

For every regular language $L \subseteq \Sigma^*$ with k as index of $\equiv_L$ there is a DFA which is unique up to isomorphism and with minimal number of states k accepting L.

4.7 **Definition** : The *minimal automaton* for a regular language $L \subseteq \Sigma^*$ is the DFA which accepts L and has the number of states equal to the index of the NERODE relation $\equiv_L$. This minimal automaton is unique up to isomorphism.

The minimal automaton for a regular language $L \subseteq \Sigma^*$ from every DFA $\mathcal{A} = (\Sigma, Q, \rightarrow, q_0, F)$ accepting L by *reduction* is algorithmically computable. The reduction includes the following steps:

1. Eliminate unreachable states.

   A state $q \in Q$ is called reachable if there is a string $w \in \Sigma^*$ with $q_0 \xrightarrow{w} q$. The subset of reachable states of Q is computable, because we can consider only those strings w with $q_0 \xrightarrow{w} q$ which have the length $\leq |Q|$ (compare with the proof of the pumping lemma).

2. Combine equivalent states.

   For $q \in Q, w \in \Sigma^*$ and $S \subseteq Q$ we write $q \xrightarrow{w} S$ if there is a $q' \in S$ with $q \xrightarrow{w} q'$. Two states $q_1, q_2 \in Q$ are called *equivalent*, in short $q_1 \sim q_2$, if for all $w \in \Sigma^*$ it holds:

   $$q_1 \xrightarrow{w} F \Leftrightarrow q_2 \xrightarrow{w} F,$$

   i. e., the same strings lead from q_1 and q_2 to final states.

   There is a close relation between equivalence and the NERODE relation $\equiv_L$. Let $q_0 \xrightarrow{u} q_1$ and $q_0 \xrightarrow{v} q_2$. Then it holds that:

   $$q_1 \sim q_2 \Leftrightarrow u \equiv_L v \Leftrightarrow [u] = [v].$$

   Comparing with the equivalence-class automaton $\mathcal{A}_L$ we see that equivalent states must coincide in the minimal automaton. q_0, q_1 and q_2 in $\mathcal{A}_L$ are represented by the equivalence classes $[\varepsilon]$, $[u]$ and $[v]$ such that from $[\varepsilon] \xrightarrow{u} [u]$ and $[\varepsilon] \xrightarrow{v} [v]$ it follows that:

   $$[u] \sim [v] \Leftrightarrow u \equiv_L v \Leftrightarrow [u] = [v].$$

§5 Decidability questions

First of all, we determine that the following constructions are algorithmically computable:

- ε-NFA $\rightarrow$ NFA $\rightarrow$ DFA
- DFA $\rightarrow$ minimal automaton
- ε-NFAs for the following operations on finitely acceptable languages:
  union, complement, intersection, difference, concatenation and iteration
- regular expression $\rightarrow$ NFA $\rightarrow$ DFA $\rightarrow$ regular expression

After that we can move on to decidability questions for the languages represented by finite automata or regular expressions. Due to the constructions mentioned above we consider only the languages represented by DFAs. We consider the following problems for regular languages.

acceptance problem	Given:	DFA $\mathcal{A}$ and a string w
	Question:	Does $w \in L(\mathcal{A})$ hold ?
emptiness problem	Given:	DFA $\mathcal{A}$
	Question:	Does $L(\mathcal{A}) = \emptyset$ hold ?
finiteness problem	Given:	DFA $\mathcal{A}$
	Question:	Is $L(\mathcal{A})$ finite ?
equivalence problem	Given:	DFA 's $\mathcal{A}_1$ and $\mathcal{A}_2$
	Question:	Does $L(\mathcal{A}_1) = L(\mathcal{A}_2)$ hold ?
inclusion problem	Given:	DFA 's $\mathcal{A}_1$ and $\mathcal{A}_2$
	Question:	Does $L(\mathcal{A}_1) \subseteq L(\mathcal{A}_2)$ hold ?
intersection problem	Given:	DFA 's $\mathcal{A}_1$ and $\mathcal{A}_2$
	Question:	Does $L(\mathcal{A}_1) \cap L(\mathcal{A}_2) = \emptyset$ hold ?

5.1 **Theorem (decidability)**: For regular languages

- the acceptance problem,
- the emptiness problem,
- the finiteness problem,
- the equivalence problem,
- the inclusion problem,
- the intersection problem

are all decidable.

Proof : *Acceptance problem*: We apply $\mathcal{A}$ to the given string w and decide whether a final state of $\mathcal{A}$ is reached. Therefore, $\mathcal{A}$ itself gives us the decision procedure.

Emptiness problem: Let n be the number we get by applying the pumping lemma to the regular language $L(\mathcal{A})$. Then it holds that:

$$L(\mathcal{A}) = \emptyset \Leftrightarrow \neg\exists\ w \in L(\mathcal{A}) : |w| < n \qquad (*)$$

The proof of $(*)$: "$\Rightarrow$" is obvious. "$\Leftarrow$" by contraposition: Let $L(\mathcal{A}) \neq \emptyset$. If there is a string $w \in L(\mathcal{A})$ with $|w| < n$, then we have nothing to show. Otherwise there is a string $w \in L(\mathcal{A})$ with $|w| \geq n$. By successive application of the pumping lemma with $i = 0$ we get a string $w_0 \in L(\mathcal{A})$ with $|w_0| < n$. This completes the proof.

The decision procedure for $L(\mathcal{A}) = \emptyset$ now solves the acceptance problem "$w \in L(\mathcal{A})$?" for every string over the input alphabet of $\mathcal{A}$ with $|w| < n$. If the answer is "no", then we get "$L(\mathcal{A}) = \emptyset$". Otherwise we get "$L(\mathcal{A}) \neq \emptyset$".

Finiteness problem: Let n be defined as above. Then it holds that:

$$L(\mathcal{A}) \text{ is finite } \Leftrightarrow \neg\exists\ w \in L(\mathcal{A}) : n \leq |w| < 2n \qquad (**)$$

The proof of $(**)$: "$\Rightarrow$": If there was a string $w \in L(\mathcal{A})$ with $|w| \geq n$, then $L(\mathcal{A})$ would be infinite by the pumping lemma. "$\Leftarrow$" by contraposition: Let $L(\mathcal{A})$ be infinite. Then there are strings of arbitrary length in $L(\mathcal{A})$, in particular a string w with $w \geq 2n$. By applying the pumping lemma successively with $i = 0$ we get a string w_0 with $n \leq |w_0| < 2n$, because with $i = 0$ the given string is shortened maximally by n letters.

The finiteness problem can be decided by $(**)$ by solving of the acceptance problem a finite number of times.

equivalence problem: First we construct a DFA $\mathcal{A}$ with the following property:

$$L(\mathcal{A}) = (L(\mathcal{A}_1) \cap \overline{L(\mathcal{A}_2)}) \cup (L(\mathcal{A}_2) \cap \overline{L(\mathcal{A}_1)}).$$

Obviously it holds that:

$$L(\mathcal{A}_1) = L(\mathcal{A}_2) \Leftrightarrow L(\mathcal{A}) = \emptyset \qquad (***)$$

Hence the equivalence problem for automata is reduced to the emptiness problem for $\mathcal{A}$. The construction of $\mathcal{A}$ described above is hard to implement. Alternatively, we can use the following *product construction*, which is similar to the last remarks in section 2 of this chapter. Let $\mathcal{A}_i = (\Sigma, Q_i, \rightarrow_i, q_{0i}, F_i), i = 1, 2$. Then consider $Q = Q_1 \times Q_2$ with the following transition relation $\rightarrow \subseteq Q \times \Sigma \times Q$

for all $q_1, q_1' \in Q_1$ and $q_2, q_2' \in Q_2$ and $a \in \Sigma$ it holds that

$(q_1.q_2) \xrightarrow{a} (q_1', q_2')$ if and only if $q_1 \xrightarrow{a} q_1'$ and $q_2 \xrightarrow{a} q_2'$. Then define $\mathcal{A} = (\Sigma, Q, \rightarrow, (q_{01}, q_{02}), F)$ with $F = \{(q_1, q_2) \in Q \mid q_1 \in F_1 \Leftrightarrow q_2 \notin F_2\}$.

Then (***) holds for this DFA $\mathcal{A}$:

$$\begin{aligned} L(\mathcal{A}_1) = L(\mathcal{A}_2) \Leftrightarrow & \quad \forall\, w \in \Sigma^* :\ (w \in L(\mathcal{A}_1) \Leftrightarrow w \in L(\mathcal{A}_2)) \\ \Leftrightarrow & \quad \forall\, w \in \Sigma^* :\ ((\exists\, q_1 \in F_1 :\ q_0 \xrightarrow{w} q_1) \Leftrightarrow (\exists\, q_2 \in F_2 :\ q_0 \xrightarrow{w} q_2)) \\ \Leftrightarrow & \quad \forall\, w \in \Sigma^*\ \forall\, (q_1, q_2) \in Q :\ (q_{01}, q_{02}) \xrightarrow{w} (q_1, q_2) \Rightarrow (q_1, q_2) \notin F \\ \Leftrightarrow & \quad L(\mathcal{A}) = \emptyset \end{aligned}$$

Inclusion problem: We construct a DFA $\mathcal{A}$ such that

$$L(\mathcal{A}) = L(\mathcal{A}_1) \cap \overline{L(\mathcal{A}_2)}$$

holds. Due to

$$L(\mathcal{A}_1) \subseteq L(\mathcal{A}_2) \Leftrightarrow L(\mathcal{A}) = \emptyset$$

the inclusion problem for $\mathcal{A}_1$ and $\mathcal{A}_2$ can be reduced to the emptiness problem of $\mathcal{A}$.

Intersection problem: We construct a DFA $\mathcal{A}$ with

$$L(\mathcal{A}) = L(\mathcal{A}_1) \cap L(\mathcal{A}_2),$$

so that the intersection problem of $\mathcal{A}_1$ and $\mathcal{A}_2$ can be reduced to the emptiness problem of $\mathcal{A}$. For $\mathcal{A}$ we can use the product construction $\mathcal{A}_\cap$ from the last remark in section 2. □

§6 Automatic verification

The automatic verification can be implemented for programs represented by finite automata. In general, the *verification problem* can be formulated as follows:

Given	:	program P and specification S
Question	:	Does P fulfil the specification S ?

This problem is also called *Model Checking*, because in terms of logic the question is whether the program P is a model of specification S. Here we consider the following special case of this problem:

- Program P $\quad\hat{=}\quad$ finite automaton (DFA, NFA or ε-NFA) $\mathcal{A}$
- Specification S $\quad\hat{=}\quad$ an *extended* regular expression re, i. e., extended by the operators $re_1 \cap re_2$ (intersection), $\overline{re}$ (complement) and Σ as an abbreviation for $a_1 + \ldots + a_n$ if $\Sigma = \{a_1, \ldots, a_n\}$ holds.
- P satisfies S $\quad\hat{=}\quad$ $L(\mathcal{A}) \subseteq L(re)$. This test can be conducted automatically, because the inclusion problem for regular languages is decidable.

Example : Consider once again the example from the field of operating systems, where two programs share a printer by means of the operations b_1 and b_2, and report the end of use by e_1 and e_2. We put $\Sigma = \{b_1, b_2, e_1, e_2\}$ and describe the set of strings over Σ, which define the allowed executions by the following extended regular expression:

$$re = \overline{\Sigma^* b_1 b_2 \Sigma^* + \Sigma^* b_2 b_1 \Sigma^*}.$$

The application of the operators $\overline{re}$ and Σ makes the regular expression re simpler. Due to closure properties of regular languages, we certainly do not leave the class of regular languages. Hence we can also decide for every given finite automaton $\mathcal{A}$ for synchronization of printer usage by both programs whether $L(\mathcal{A}) \subseteq L(re)$ holds.

One of the possibilities to define the satisfiability is as follows:

- P satisfies $S \stackrel{\wedge}{=} L(\mathcal{A}) = L(re)$. This test can be automatically conducted, because the equivalence problem for languages is decidable.

Even the following *synthesis problem* can be solved automatically for program P represented as a finite automaton and specification S represented as extended regular expression:

Given	:	a specification S
Searched	:	a program P that satisfies S.

Chapter III

Context-free languages and push-down automata

In the previous chapter we have seen that regular languages have multiple applications in computer science (e.g. lexical analysis and substring recognition) and are particularly easy to use (representability by finite automata and regular expressions, good closure and decidability properties). However these are not enough to carry out an important task of computer science, namely the *syntax description of programming languages*.

The reason is that programming languages accept bracket structures of arbitrary nesting-depth, such as for example

- arithmetic expressions of the form $3 * (4 - (x + 1))$,
- lists of the form (`CAR`(`CONS` x y)) or
- statements of the form

```
while(b1) {
  x = e1;
  while(b1) {
    y = e2;
    z = e3;
  }
}
```

In section 4 we have shown with the help of the pumping lemma that the simplest example of such a bracket structure, namely the language

$$L = \{a^n b^n \mid n \in \mathbb{N}\}$$

is no longer regular. Context-free languages are used for the syntax description of programming languages.

§1 Context-free grammars

First we consider the corresponding grammars.

1.1 **Definition** : A *context-free grammar* is a 4-tuple $G = (N, T, P, S)$, where the following holds:

(i) N is an alphabet of *non-terminal symbols*,

(ii) T is an alphabet of *terminal symbols* with $N \cap T = \emptyset$,

(iii) $S \in N$ is the *start symbol*,

(iv) $P \subseteq N \times (N \cup T)^*$ is a finite set of *productions* or *rules*.

We use the following formal definitions:

- $A, B, C, \ldots$ stand for non-terminal symbols,
- $a, b, c, \ldots$ stand for terminal symbols,
- $u, v, w, \ldots$ stand for strings with terminal and non-terminal symbols.

We often write down a production $(A, u) \in P$ as an arrow notation $A \to u$. If several productions have the same left side, such as

$$A \to u_1, A \to u_2, \ldots, A \to u_k,$$

then we write it down shorter as a unique "metarule"

$$A \to u_1 \,|\, u_2 \,|\, \ldots \,|\, u_k$$

or also

$$A ::= u_1 \,|\, u_2 \,|\, \ldots \,|\, u_k. \qquad (*)$$

Thus $|$ is a "metasymbol" for the disjunction of alternatives $u_1, \ldots, u_k$, which may not occur in $N \cup T$.

If the productions of a context-free grammar are represented in the form $(*)$, then we deal with *Backus-Naur form* or shortly BNF-notation. This notation was introduced in 1960 by John Backus and Peter Naur to define the programming language ALGOL 60. The extended BNF-notation, also called EBNF, allows us to make further abbreviations. The EBNF-notation can be translated 1–1 into the syntax diagram introduced in 1970 by Niklaus Wirth to define the programming language PASCAL.

Every context-free grammar G has the two-place *derivation relation* $\vdash_G$ on $(N \cup T)^*$:

$$x \vdash_G y \quad \text{if and only if} \quad \exists A \to u \in P \; \exists w_1, w_2 \in (N \cup T)^* : \\ x = w_1 \boxed{A} w_2 \text{ and } y = w_1 \boxed{u} w_2.$$

By $\vdash_G^*$ we denote the reflexive and transitive closure of $\vdash_G$. We read $x \vdash^* y$ as "y can be derived from x". The *language generated* by G is

$$L(G) = \{w \in T^* \mid S \vdash_G^* w\}.$$

Two context-free grammars G_1 and G_2 are called *equivalent* if $L(G_1) = L(G_2)$.

1.2 **Definition** : A *language* $L \subseteq T^*$ is called *context-free* if there is a context-free grammar G with $L = L(G)$.

Example :

(1) The language $L = \{a^n b^n \mid n \in \mathbb{N}\}$ is generated by the grammar $G_1 = (\{S\}, \{a, b\}, P_1, S)$, where P_1 is given as follows:

$$S \to \varepsilon \mid aSb.$$

For example, it holds that $a^2b^2 \in L(G_1)$, because

$$S \vdash_{G_1} aSb \vdash_{G_1} aaSbb \vdash_{G_1} aabb.$$

(2) The arithmetic expressions with variables a, b, c and operators $+$ and $*$ are generated by the grammar $G_2 = (\{S\}, \{a, b, c, +, *, (,)\}, P_2, S)$ with the following P_2:

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

For example $(a + b) * c \in L(G_2)$, because

$$S \vdash_{G_2} S * S \vdash_{G_2} (S) * S \vdash_{G_2} (S + S) * S \\ \vdash_{G_2} (a + S) * S \vdash_{G_2} (a + b) * S \vdash_{G_2} (a + b) * c.$$

Now we will consider the derivation of strings in a context-free grammar in more detail.

1.3 **Definition** : A *derivation* from A to w in G with the length $n \geq 0$ is a sequence of derivation steps

$$A = z_0 \vdash_G z_1 \vdash_G \cdots \vdash_G z_n = w. \qquad (**)$$

This derivation is called *leftmost derivation* if we replace the leftmost non-terminal symbol in every derivation step $z_i \vdash_G z_{i+1}$, i.e. if z_i and z_{i+1} always have the form

$$z_i = w_1 A w_2 \text{ and } z_{i+1} = w_1 u w_2, \text{ where } w_1 \in T^*.$$

The *rightmost derivations* are defined respectively (then it holds that: $w_2 \in T^*$).

Every derivation can be graphically represented as a tree.

1.4 **Definition** : A *derivation tree* from A to w in G is a tree with the following properties:

(i) Every node is labelled with a symbol from $N \cup T \cup \{\varepsilon\}$. The root is labelled with A and every internal node is labelled with a symbol from N.

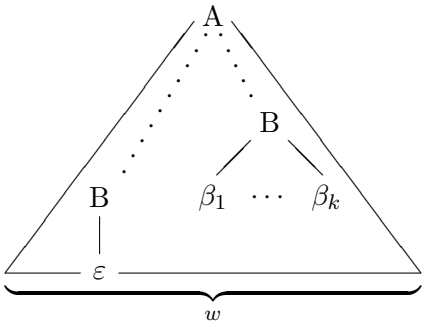
(ii) If an internal node labelled with B has k children nodes, which are labelled with the symbols $\beta_1, \ldots, \beta_k$ from left to right, then it holds that

a) $k = 1$ and $\beta_1 = \varepsilon$ and $B \to \varepsilon \in P$
or

b) $k \geq 1$ and $\beta_1, \ldots, \beta_k \in N \cup T$ and $B \to \beta_1 \ldots \beta_k \in P$.

(iii) The string w is generated by concatenating the symbols on the leaves from left to right.

Illustration



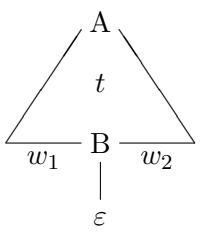
We construct the derivation tree from A to w corresponding to a derivation from A to w of the form $(**)$ by induction on the length n of the derivation.

$n = 0$: The trivial derivation A belongs to the trivial derivation A.

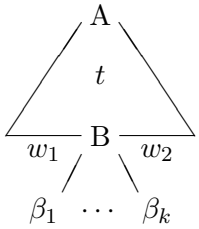
$n \to n + 1$: Consider a derivation

$$A = z_0 \vdash_G \ldots \vdash_G z_n = w_1 B w_2 \vdash_G w_1 u w_2 = z_{n+1}.$$

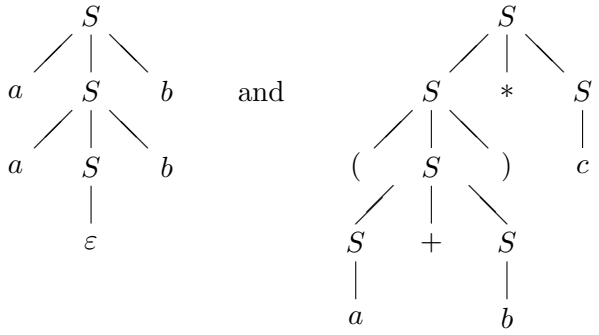
Let t be the derivation tree corresponding to the derivation $A = z_0 \vdash_G \ldots \vdash_G z_n$.
If $u = \varepsilon$, then the entire derivation tree is as follows:



If $u = \beta_1 \ldots \beta_k$, where $\beta_1, \ldots, \beta_k \in N \cup T$, then the entire derivation tree is as follows:



Example : Derivation trees for the derivations mentioned in the previous example are



Remark : There exists the following relationship between derivations and derivation trees:

(i) $A \vdash_G^* w \quad \Leftrightarrow \quad$ There is a derivation tree from A to w in G.

(ii) In general, several derivations from A to w correspond to the given derivation tree from A to w. However, this holds for only one leftmost derivation and one rightmost derivation.

Proof :

(i) "$\Rightarrow$" is obvious due to the above construction.
We show "$\Leftarrow$" inductively on the depth of the derivation tree.

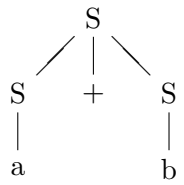
(ii) Derivation trees abstract from the unimportant order of the rule application if several non-terminal symbols occur simultaneously. For example, both derivations

$$S \vdash_{G_2} S + S \vdash_{G_2} a + S \vdash_{G_2} a + b$$

and

$$S \vdash_{G_2} S + S \vdash_{G_2} S + b \vdash_{G_2} a + b$$

result in the same derivation tree:



If we have decided to use leftmost or rightmost derivation respectively, then such alternatives are not possible.

□

Now let the syntax of a programming language PL be given by a context-free grammar G. A compiler for PL generates for every given PL-program a derivation tree in G in the syntax analysis phase. The meaning or semantics of the PL-program depends on the structure of the generated derivation tree. The compiler generates the machine code of the PL-program on the basis of the derivation tree.

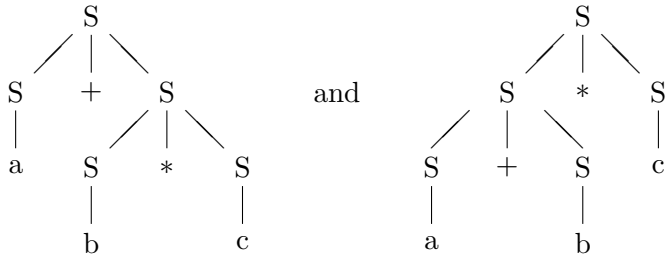
For using the programming language PL it is important that every PL-program has an unambiguous semantics. Therefore, for every PL-program there should exist exactly one derivation tree.

1.5 **Definition** :

(i) A context-free *grammar* $G = (N, T, P, S)$ is called *unambiguous* if for every string $w \in T^*$ there is at most one derivation tree or a leftmost derivation from S to w in G respectively. Otherwise G is *ambiguous*.

(ii) A context-free *language* $L \subseteq T^*$ is called *unambiguous* if there is an unambiguous context-free grammar G with $L = L(G)$. Otherwise L is called *(inherently) ambiguous*.

Example : The above given grammar G_2 for arithmetic expressions is ambiguous. For the string $a + b * c \in L(G_2)$ there exist the following two derivation trees:



These correspond to semantically different bracketings $a + (b * c)$ and $(a + b) * c$.

Therefore, in programming languages we choose a grammar G_3 for arithmetic expressions, where the following evaluation strategy is chosen:

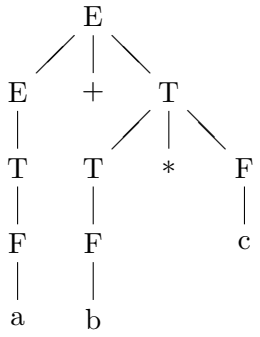
- The evaluation takes place from left to right. Thus, for example, $a + b + c$ is evaluated as $(a + b) + c$.
- Multiplication $*$ has a higher priority than $+$. Thus, for example, $a + b * c$ is evaluated as $a + (b * c)$.

If we want to have another evaluation sequence, we must explicitly put brackets (and).

Consider $G_3 = (\{E, T, F\}, \{a, b, c, +, *, (,)\}, P_3, E)$ with the following P_3:

$$\begin{aligned} E &\rightarrow T \mid E + T \\ T &\rightarrow F \mid T * F \\ F &\rightarrow (E) \mid a \mid b \mid c \end{aligned}$$

G_3 is unambiguous, and it holds that $L(G_3) = L(G_2)$. For example, $a + b * c$ has the derivation tree in G_3



Example : (Chomsky, 1964) An inherent ambiguous context-free language is

$$L = \{a^i b^j c^k \mid i, j, k \geq 0 \text{ and } (i = j \text{ or } j = k)\}.$$

For the proof see the literature.

§2 Pumping Lemma

There also exists a pumping lemma for context-free languages. It provides a necessary condition for a given language to be context-free.

2.1 **Theorem (Pumping lemma for context-free languages or $uvwxy$-lemma)**: For every context-free language $L \subseteq T^*$ there exists a number $n \in \mathbb{N}$ such that for all strings $z \in L$ with $|z| \geq n$ there exists a decomposition $z = uvwxy$ with the following properties:

(i) $vx \neq \varepsilon$,

(ii) $|vwx| \leq n$,

(iii) for all $i \in \mathbb{N}$ it holds that: $uv^i wx^i y \in L$.

Hence we can "pump" the substrings v and x an arbitrary number of times without leaving the context-free language L.

To prove the pumping lemma we need the following general lemma for trees, which we can then apply to derivation trees. For finite trees t we define:

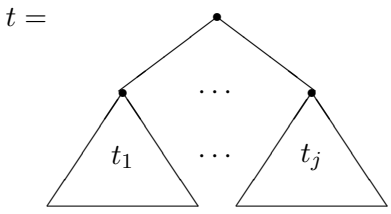
- Branching factor of t = maximum number of children nodes of any node in t.
- A path of length m in t is a sequence of edges from the root to a leaf of t with m edges. The trivial case $m = 0$ is allowed.

2.2 **Lemma** : Let t be a finite tree with the branching factor $\leq k$, where every path has the length $\leq m$. Then the number of leaves in t is $\leq k^m$.

Proof : Induction on $m \in \mathbb{N}$:

$m = 0$**:** t consists only of $k^0 = 1$ nodes.

$m \to m + 1$**:** t has j subtrees $t_1, \ldots, t_j$ with $j \leq k$, where the paths have the length $\leq m$:



By induction hypothesis the number of leaves in each of the subtrees $t_1, \ldots, t_j$ is $\leq k^m$. Therefore, it holds for t that:

$$\text{number of leaves } \leq j \cdot k^m \leq k \cdot k^m = k^{m+1}.$$

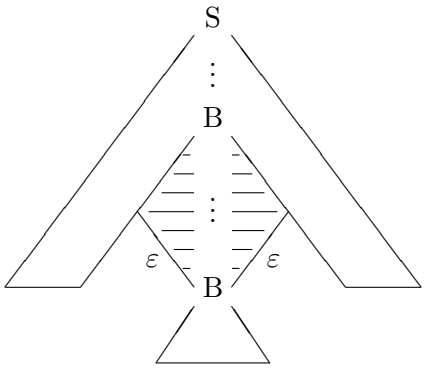
□

Now we are ready for the

Proof of the pumping lemma: Let $G = (N, T, P, S)$ be a context-free grammar with $L(G) = L$. Let:

- $k =$ the length of the longest right side of a production from P, however at least 2
- $m = |N|$,
- $n = k^{m+1}$.

Now let $z \in L$ with $|z| \geq n$. Then there is a derivation tree t from S to z in G with no part corresponding to a derivation of the form $B \vdash_G^* B$:

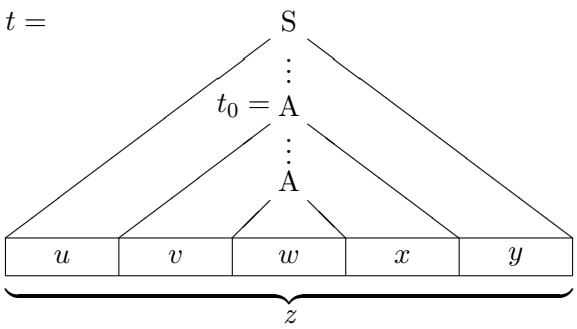


Every part like this could actually be removed from t without changing the derived string z.

Having chosen k and $|z|$ as above, we can see that t has a branching factor $\leq k$ and $\geq k^{m+1}$ leaves. Therefore, according to the previously considered lemma there is a path with the length $\geq m + 1$ in t. There are $\geq m + 1$ internal nodes on this path, so that a non-terminal symbol is repeated, while labeling these nodes. We need this repetition in a special case.

By a *repetition tree* in t we denote a subtree of t, where the labeling of the root is repeated in some other node. Now we choose a minimal repetition tree t_0 in t, i.e. such a tree that contains no other repetition tree (as a real subtree). In t_0 every path has a length $\leq m + 1$.

Let A be the root labeling of t_0. Then t has the following structure:



From this structure we get a decomposition $z = uvwxy$ with

$$S \vdash_G^* uAy \vdash_G^* uvAxy \vdash_G^* uvwxy. \qquad (*)$$

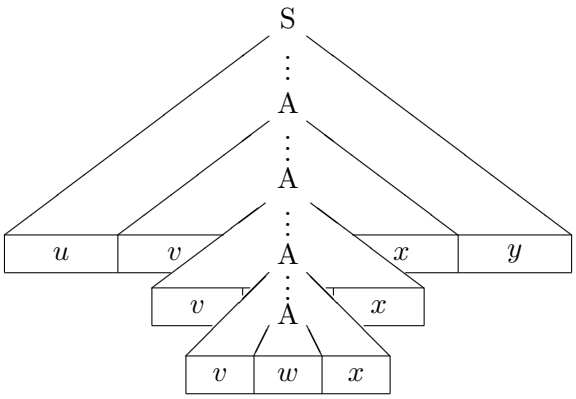
We show that this decomposition of z satisfies the conditions of the pumping lemma:

(i) By the choice of t it holds that $vx \neq \varepsilon$.

(ii) By the choice of t_0 and the preceding pumping lemma it holds that $|vwx| \leq k^{m+1} = n$.

(iii) From $(*)$ it follows immediately that for all $i \in \mathbb{N}$ it holds that: $uv^iwx^iy \in L(G)$.

The derivation tree from S to uv^iwx^iy in G for $i = 3$ looks as follows:



□

Just as in regular languages, the above pumping lemma can be used to prove that a certain language is *not* context-free.

Example : The language $L = \{a^nb^nc^n \mid n \in \mathbb{N}\}$ is not context-free. We provide a proof by contradiction.

Assumption: L is context-free. Then according to the pumping lemma there is an $n \in \mathbb{N}$ with given properties. Now consider $z = a^nb^nc^n$. Because $|z| \geq n$ holds, we can decompose z into $z = uvwxy$ with $vx \neq \varepsilon$ and $|vwx| \leq n$ such that for all $i \in \mathbb{N}$ it holds that $uv^iwx^iy \in L$. Because $|vwx| \leq n$, no a's *or* no c's occur in the substring vwx. Therefore, while pumping up to uv^iwx^iy, at most *two* of the characters a, b, c will be considered. However, such strings are not in L. *Contradiction.* □

The pumping lemma allows us to show that context-free grammars are not enough to provide the *full* description of the syntax of more advanced programming languages such as Java. Though we describe the basic structure of the syntactically correct programs with context-free grammars (in BNF or EBNF notation), there exist side conditions, which are context-sensitive.

Example : The programming language Java, which consists of all syntactically correct Java programs, is not context-free. We make a proof by contradiction.

Hypothesis: Let Java be context-free. Then there is an $n \in \mathbb{N}$ with the properties mentioned in the pumping lemma. Now let us consider the following syntactically correct Java-class:

$$
\begin{array}{l}
\texttt{class } C \ \{ \\
\quad \texttt{int } X \underbrace{1 \ldots 1}_{n}; \\
\quad \texttt{void } m() \ \{ \\
\qquad X \underbrace{1 \ldots 1}_{n} = X \underbrace{1 \ldots 1}_{n} \\
\quad \} \\
\}
\end{array}
$$

By every $uvwxy$-decomposition of this program, the vwx-part influences at most two out of three occurrences of the variables $X \underbrace{1 \ldots 1}_{n}$, because $|vwx| \leq n$. Therefore, while pumping to $uv^i wx^i y$, there appear either character strings which do not fulfill the requirements of the Java syntax diagram, or character strings of the form

$$
\begin{array}{l}
\texttt{class } C \ \{ \\
\quad \texttt{int } X \underbrace{1 \ldots 1}_{k}; \\
\quad \texttt{void } m() \ \{ \\
\qquad X \underbrace{1 \ldots 1}_{l} = X \underbrace{1 \ldots 1}_{m} \\
\quad \} \\
\}
\end{array}
$$

where k, l, m are not all equal. These character strings violate the following condition for syntactically correct Java programs:

"Every variable must be declared before use." $(**)$

Here either $X \underbrace{1 \ldots 1}_{l}$ or $X \underbrace{1 \ldots 1}_{m}$ (or both of them) is not declared.

Contradiction. □

Such conditions as $(**)$ are context-sensitive and thus are mentioned while defining programming languages *together* with the context-free basic structure of programs. A compiler checks these context-sensitive conditions by applying appropriate tables for saving declared variables or general identifiers.

§3 Push-down automata

So far we defined context-free languages by the fact that they can be generated by grammars. Now we want to consider *recognizability* of such languages by automata. Our goal is to extend the model of the finite automaton in such a way that it can recognize the context-free languages.

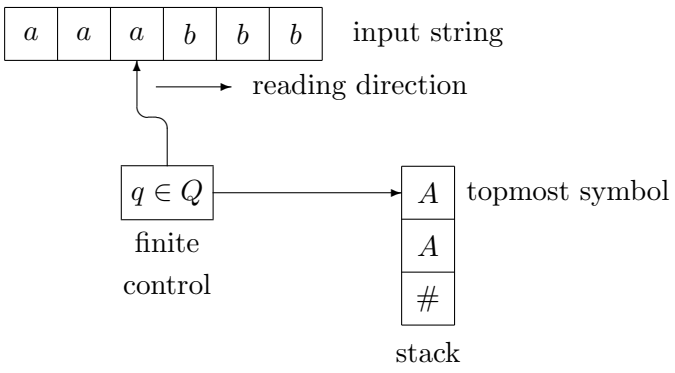
The weak point of finite automata is that they lack memory to store an unlimited amount of data. Obviously, a finite automaton cannot recognize such a language as

$$L = \{a^n b^n \mid n \in \mathbb{N}\}$$

because at the moment the first b is read, it does not know any more how many a's have been read. The only information saved is the current state, i.e. an element of a finite set of states.

Now we consider the so called *push-down automata*. These are nondeterministic finite automata with ε-transitions (ε-NFAs), extended with memory which can save infinitely long strings but can be accessed very limitedly. The memory is organized as a *stack* or a *pushdown-list* with access only to the topmost symbol. The transitions of a push-down automaton may depend on the current state, on the symbol of the input string which has been read, and on the topmost symbol of the stack; they change the state and the contents of the stack.

Sketch



3.1 **Definition (push-down automaton)**: A (*nondeterministic*) *push-down automaton*, shortly PDA, is a 7-tuple

$$\mathcal{K} = (\Sigma, Q, \Gamma, \to, q_0, Z_0, F)$$

with the following properties:

(i) Σ is the *input alphabet*,

(ii) Q is a finite set of *states*,

(iii) Γ is the *stack alphabet*,

(iv) $\to \subseteq Q \times \Gamma \times (\Sigma \cup \{\varepsilon\}) \times Q \times \Gamma^*$ is the *transition relation*,

(v) $q_0 \in Q$ is the *start state*,

(vi) $Z_0 \in \Gamma$ is the *start symbol of the stack*,

(vii) $F \subseteq Q$ is the set of *final states*.

While speaking about PDAs we typically use the following characters: $a, b, c \in \Sigma$, $u, v, w \in \Sigma^*$, $\alpha \in \Sigma \cup \{\varepsilon\}$, $q \in Q$, $Z \in \Gamma$, $\gamma \in \Gamma^*$. These characters can also be "decorated" with indices and primes. The elements $(q, Z, \alpha, q', \gamma') \in \to$ are called *transitions*. Instead of $(q, Z, \alpha, q', \gamma') \in \to$ we often write $(q, Z) \overset{\alpha}{\to} (q', \gamma')$.

We can illustrate it as follows. A transition $(q, Z) \overset{a}{\to} (q', \gamma')$ means: if q is the current state and Z is the topmost stack symbol, then the PDA can read the input symbol a, can move to the state q' and replace the symbol Z by the string γ'. Similarly a transition $(q, Z) \overset{\varepsilon}{\to} (q', \gamma')$ means: in the state q and Z as a topmost stack symbol PDA can make a transition into the state q' and replace Z by γ' on the top of the stack. In this case no input symbol is read.

If $\gamma' = \varepsilon$, then in a transition $(q, Z) \overset{\alpha}{\to} (q', \varepsilon)$ we speak about a *pop step*, because the topmost stack symbol is removed. If $\gamma' = \gamma Z$, then we speak about a *push step* in a transition $(q, Z) \overset{\alpha}{\to} (q', \gamma Z)$, because the string γ is added to the top of the stack.

In order to define the acceptance behaviour of PDAs, we must – just as in ε-NFAs – first of all extend the transition relation. For this purpose we need the definition of the configuration of a PDA.

3.2 **Definition** : Let $\mathcal{K} = (\Sigma, Q, \Gamma, \to, q_0, Z_0, F)$ be a PDA.

(i) By a *configuration* of $\mathcal{K}$ we understand a pair $(q, \gamma) \in Q \times \Gamma^*$ which describes the current state q and the current stack contents γ of $\mathcal{K}$.

(ii) For every $\alpha \in \Sigma \cup \{\varepsilon\}$, $\overset{\alpha}{\to}$ is a 2-place relation on configurations of $\mathcal{K}$ defined as follows:

$$(q, \gamma) \overset{\alpha}{\to} (q', \gamma') \text{ if } \exists Z \in \Gamma, \quad \exists \gamma_0, \gamma_1 \in \Gamma^* :$$

$$\gamma = Z\gamma_0 \text{ and } (q, Z, \alpha, q', \gamma_1) \in \to \text{ and } \gamma' = \gamma_1\gamma_0$$

By $\overset{\alpha}{\to}$ we denote the *α-transition relation* on configurations.

(iii) For every string $w \in \Sigma^*$, $\overset{w}{\Rightarrow}$ is a 2-place relation on configurations of $\mathcal{K}$ defined inductively:

- $(q, \gamma) \overset{\varepsilon}{\Rightarrow} (q', \gamma')$ if $\exists n \geq 0 : (q, \gamma) \underbrace{\overset{\varepsilon}{\to} \circ \ldots \circ \overset{\varepsilon}{\to}}_{n\text{-times}} (q', \gamma')$
- $(q, \gamma) \overset{aw}{\Rightarrow} (q', \gamma')$ if $(q, \gamma) \overset{\varepsilon}{\Rightarrow} \circ \overset{a}{\to} \circ \overset{w}{\Rightarrow} (q', \gamma')$, for all $a \in \Sigma$.

By $\overset{w}{\Rightarrow}$ we denote the *extended w - transition relation* on configurations.

Remark : For all $\alpha \in \Sigma \cup \{\varepsilon\}, a_1, \ldots, a_n \in \Sigma$ and $w_1, w_2 \in \Sigma^*$ it holds that:

(i) $(q, \gamma) \overset{\alpha}{\to} (q', \gamma')$ implies $(q, \gamma) \overset{\alpha}{\Rightarrow} (q', \gamma')$.

(ii) $(q,\gamma) \overset{a_1 \ldots a_n}{\Longrightarrow} (q',\gamma')$ if and only if $(q,\gamma) \overset{\varepsilon}{\Rightarrow} \circ \overset{a_1}{\to} \circ \overset{\varepsilon}{\Rightarrow} \ldots \overset{\varepsilon}{\Rightarrow} \circ \overset{a_n}{\to} \circ \overset{\varepsilon}{\Rightarrow} (q',\gamma')$
if and only if $(q,\gamma) \overset{a_1}{\Rightarrow} \circ \ldots \circ \overset{a_n}{\Rightarrow} (q',\gamma')$

(iii) $(q,\gamma) \overset{w_1 w_2}{\Longrightarrow} (q',\gamma')$ if and only if $(q,\gamma) \overset{w_1}{\Rightarrow} \circ \overset{w_2}{\Rightarrow} (q',\gamma')$

There exist two variants of language acceptance.

3.3 **Definition (acceptance)**:
Let $\mathcal{K} = (\Sigma, Q, \Gamma, \to, q_0, Z_0, F)$ be a PDA and $w \in \Sigma^*$.

(i) $\mathcal{K}$ *accepts* w if $\exists q \in F \quad \exists \gamma \in \Gamma^* : \quad (q_0, Z_0) \overset{w}{\Longrightarrow} (q,\gamma)$.
The *language accepted* (or *recognized*) by $\mathcal{K}$ is

$$L(\mathcal{K}) = \{w \in \Sigma^* \mid \mathcal{K} \text{ accepts } w\}.$$

(ii) $\mathcal{K}$ *accepts* w *with the empty stack*,
if

$$\exists q \in Q : \quad (q_0, Z_0) \overset{w}{\Longrightarrow} (q,\varepsilon).$$

The language accepted (or recognized) by $\mathcal{K}$ with an empty stack is

$$L_\varepsilon(\mathcal{K}) = \{w \in \Sigma^* \mid \mathcal{K} \text{ accepts } w \text{ with an empty stack}\}.$$

Example : We construct a push-down automaton which accepts the mentioned language $L = \{a^n b^n \mid n \in \mathbb{N}\}$. Let

$$\mathcal{K} = (\{a,b\}, \{q_0,q_1,q_2\}, \{a,Z\}, \to, q_0, Z, \{q_0\}),$$

where the transition relation $\to$ consists of the following transitions:

(1)	(q_0, Z)	$\overset{a}{\to}$	(q_1, aZ)
(2)	(q_1, a)	$\overset{a}{\to}$	(q_1, aa)
(3)	(q_1, a)	$\overset{b}{\to}$	(q_2, ε)
(4)	(q_2, a)	$\overset{b}{\to}$	(q_2, ε)
(5)	(q_2, Z)	$\overset{\varepsilon}{\to}$	(q_0, ε)

$\mathcal{K}$ accepts $a^n b^n$ for $n \geq 1$ by using the following $2n+1$ transitions, which we have labelled for the sake of clarity with numbers from 1 to 5:

$$\begin{aligned}(q_0, Z) &\overset{a}{\to}_1 (q_1, aZ) \overset{a}{\to}_2 (q_1, aaZ) \ldots \overset{a}{\to}_2 (q_1, a^n Z) \\ &\overset{b}{\to}_3 (q_2, a^{n-1} Z) \overset{b}{\to}_4 (q_2, a^{n-2} Z) \ldots \overset{b}{\to}_4 (q_2, Z) \\ &\overset{\varepsilon}{\to}_5 (q_0, \varepsilon)\end{aligned}$$

Thus the relation $(q_0, Z) \overset{a^n b^n}{\Longrightarrow} (q_0, \varepsilon)$ holds for $n \geq 1$. For $n = 0$ it trivially holds that $(q_0, Z) \overset{\varepsilon}{\Rightarrow} (q_0, Z)$. Because q_0 is the final state of $\mathcal{K}$, it follows that $L \subseteq L(\mathcal{K})$.

For the inclusion $L(\mathcal{K}) \subseteq L$ we must analyse the transition behaviour of $\mathcal{K}$. For this purpose we index the transitions as above. We can see that $\mathcal{K}$ is *deterministic*, i.e. while sequentially reading

characters of a given input string, only one transition can be applied on each step. Namely the following transition sequences are possible, where $n \geq m \geq 0$ holds:

- $(q_0, Z) \overset{a}{\rightarrow}_1 \circ \left(\overset{a}{\rightarrow}_2\right)^n (q_1, a^{n+1}Z)$
- $(q_0, Z) \overset{a}{\rightarrow}_1 \circ \left(\overset{a}{\rightarrow}_2\right)^n \circ \overset{b}{\rightarrow}_3 \circ \left(\overset{b}{\rightarrow}_4\right)^m (q_2, a^{n-m}Z)$
- $(q_0, Z) \overset{a}{\rightarrow}_1 \circ \left(\overset{a}{\rightarrow}_2\right)^n \circ \overset{b}{\rightarrow}_3 \circ \left(\overset{b}{\rightarrow}_4\right)^n \circ \overset{\varepsilon}{\rightarrow}_5 (q_0, \varepsilon)$

Therefore, $\mathcal{K}$ accepts only those strings which have the form $a^n b^n$. Altogether, it holds that $L(\mathcal{K}) = L$.

Furthermore, note that the automaton $\mathcal{K}$ accepts all strings $a^n b^n$ with the empty stack, except for the case $n = 0$: $L_\varepsilon(\mathcal{K}) = \{a^n b^n \mid n \geq 1\}$.

Now we want to derive general properties of push-down automata. For this purpose we often need the following Top Lemma. Intuitively, it points out that changes on the top of the stack do not depend on the rest of the stack.

3.4 **Lemma (top of the stack)**: Let $\mathcal{K} = (\Sigma, Q, \Gamma, \rightarrow, q_0, Z_0, F)$ be a push-down automaton. Then for all $w \in \Sigma^*$, $q, q' \in Q$, $Z \in \Gamma$ and $\gamma \in \Gamma^*$ it holds that: if

$$(q, Z) \overset{w}{\Longrightarrow} (q', \varepsilon),$$

then also

$$(q, Z\gamma) \overset{w}{\Longrightarrow} (q', \gamma).$$

Proof : Exercise □

Now we show that both variants of language acceptance are equivalent.

3.5 **Theorem (acceptance)**:

(1) For every PDA $\mathcal{A}$ we can construct a PDA B with $L(\mathcal{A}) = L_\varepsilon(B)$.

(2) For every PDA $\mathcal{A}$ we can construct a PDA B with $L_\varepsilon(\mathcal{A}) = L(B)$.

Proof : Let $\mathcal{A} = (\Sigma, Q, \Gamma, \rightarrow_{\mathcal{A}}, q_0, Z_0, F)$.
(1): The idea of the proof is simple: B works like $\mathcal{A}$ and empties the stack in final states. However, we should take care of the fact that B should not have an empty stack after having read input strings which $\mathcal{A}$ does not accept. Therefore, B uses an additional symbol # to label the bottom of the stack. To be more precise, we construct:

$$B = (\Sigma, Q \cup \{q_B, q_\varepsilon\}, \Gamma \cup \{\#\}, \rightarrow_B, q_B, \#, \emptyset)$$

with $q_B, q_\varepsilon \notin Q$ and $\# \notin \Gamma$ and the following transition relation:

$$\begin{array}{lll} \to_B & = \{(q_B, \#, \varepsilon, q_0, Z_0\#)\} & \text{“starting } \mathcal{A}\text{”} \\ & \cup \to_{\mathcal{A}} & \text{“working like } \mathcal{A}\text{”} \\ & \left.\begin{array}{l} \cup \{(q, Z, \varepsilon, q_\varepsilon, \varepsilon) \mid q \in F, Z \in \Gamma \cup \{\#\}\} \\ \cup \{(q_\varepsilon, Z, \varepsilon, q_\varepsilon, \varepsilon) \mid Z \in \Gamma \cup \{\#\}\} \end{array}\right\} & \text{“emptying the stack”} \end{array}$$

Then it holds for all $w \in \Sigma^*, q \in F$ and $\gamma \in \Gamma^*$:

$$(q_0, Z_0) \overset{w}{\Longrightarrow}_{\mathcal{A}} (q, \gamma)$$

if and only if

$$(q_B, \#) \overset{\varepsilon}{\to}_B (q_0, Z_0\#) \overset{w}{\Longrightarrow}_{\mathcal{A}} (q, \gamma\#) \overset{\varepsilon}{\to}_B (q_\varepsilon, \varepsilon).$$

(For the “if-then” direction we use the Top Lemma.) Analysing the applicability of the new transitions in B we get $L(\mathcal{A}) = L_\varepsilon(B)$.

(2): Idea of the proof: B works like $\mathcal{A}$, but uses an additional symbol # to label the bottom of the stack. As soon as $\mathcal{A}$ has emptied its stack, B reads the symbol # and moves to a final state. Making the exact construction of B should be done as an exercise. □

Now we want to show that the nondeterministic push-down automata accept context-free languages (with empty stack). First for a given context-free grammar G we construct a push-down automaton which represents a nondeterministic “top-down” parser of the language $L(G)$.

3.6 **Theorem** : For every context-free grammar G we can construct a nondeterminstic push-down automaton $\mathcal{K}$ with $L_\varepsilon(\mathcal{K}) = L(G)$.

Proof : Let $G = (N, T, P, S)$. We construct $\mathcal{K}$ in such a way that it simulates the *leftmost derivation* in G:

$$\mathcal{K} = (T, \{q\}, N \cup T, \to, q, S, \emptyset),$$

where the transition relation $\to$ consists of the following types of transitions:

(1) $(q, A) \overset{\varepsilon}{\to} (q, u)$, if $A \to u \in P$,
(2) $(q, a) \overset{a}{\to} (q, \varepsilon)$, if $a \in T$.

$\mathcal{K}$ works as follows: first S is on the stack. A rule application $A \to u$ of the grammar is executed on the stack by replacing the topmost stack symbol A by u. If a terminal symbol is at the top of the stack, we compare it with the next symbol of the input string and, if they are equal, we remove it from the stack. This way we produce stepwise by $\mathcal{K}$ a leftmost derivation of the input string. If this derivation is successful, then $\mathcal{K}$ accepts the input string with the empty stack. The application of transitions of type (1) is *nondeterministic* if there are several rules with the same non-terminal symbol A in P. The only state q is not important for the transition behavior of $\mathcal{K}$, but must be indicated in order for the definition of $\mathcal{K}$ to be complete.

To show that $L(G) = L_\varepsilon(\mathcal{K})$ holds, we consider more carefully the relationship between leftmost derivations in G and transition sequences in $\mathcal{K}$. For this purpose we use the following abbreviations of strings $w \in (N \cup T)^*$:

- w_T is the longest prefix of w with $w_T \in T^*$,
- w_R is the remainder of w, defined by $w = w_T w_R$.

Hypothesis 1 For all $A \in N, w \in (N \cup T)^*, n \geq 0$ and leftmost derivations

$$A \underbrace{\vdash_G \ldots \vdash_G}_{n\text{-times}} w$$

of the length n holds

$$(q, A) \stackrel{w_T}{\Longrightarrow} (q, w_R).$$

Proof by induction on n:

$\underline{n = 0:}$ Then $w = A$, thus $w_T = \varepsilon$ and $w_R = A$. Trivially $(q, A) \stackrel{\varepsilon}{\Rightarrow} (q, A)$ holds.

$\underline{n \to n + 1:}$ We analyse the last step of a leftmost derivation with the length $n + 1$:

$$A \underbrace{\vdash_G \ldots \vdash_G}_{n\text{-times}} \tilde{w} = \tilde{w}_T B v \vdash_G \tilde{w}_T u v = w$$

for $B \in N$ and $u, v \in (N \cup T)^*$ with $B \to u \in P$. By the induction hypothesis it holds that

$$(q, A) \stackrel{\tilde{w}_T}{\Longrightarrow} (q, Bv).$$

The transition type (1) implies that

$$(q, Bv) \stackrel{\varepsilon}{\Rightarrow} (q, uv).$$

The transition type (2) also implies that

$$(q, uv) \stackrel{(uv)_T}{\Longrightarrow} (q, (uv)_R).$$

Because $w_T = (\tilde{w}_T u v)_T = \tilde{w}_T (uv)_T$ and $w_R = (\tilde{w}_T u v)_R = (uv)_R$ hold, then in general we get

$$(q, A) \stackrel{w_T}{\Longrightarrow} (q, w_R).$$

Thus we have proved statement 1.

Hypothesis 2 For all $A \in N, m \geq 0, \alpha_1, \ldots, \alpha_m \in T \cup \{\varepsilon\}, \gamma_0, \ldots, \gamma_m \in (N \cup T)^*$ and all transition sequences

$$(q, A) = (q, \gamma_0) \xrightarrow{\alpha_1} (q, \gamma_1) \cdots \xrightarrow{\alpha_m} (q, \gamma_m)$$

of the length m holds

$$A \vdash_G^* \alpha_1 \ldots \alpha_m \gamma_m.$$

Proof by induction on m:

$\underline{m = 0:}$ Then $\gamma_0 = A$. Trivially $A \vdash_G^* A$ holds.

$\underline{m \to m + 1:}$ We analyse the last transition

$$(q, \gamma_m) \stackrel{\alpha_{m+1}}{\longrightarrow} (q, \gamma_{m+1}).$$

By the induction hypothesis $A \vdash_G^* \alpha_1 \dots \alpha_m \gamma_m$.

Case $\alpha_{m+1} = \varepsilon$
Then we used transition type (1) and the transition has the form

$$(q, \gamma_m) = (q, Bv) \xrightarrow{\varepsilon} (q, uv) = (q, \gamma_{m+1})$$

for some $B \in N$ and $u, v \in (N \cup T)^*$, where $B \to u \in P$. Thus it holds

$$A \vdash_G^* \alpha_1 \dots \alpha_m Bv \vdash_G \alpha_1 \dots \alpha_m uv = \alpha_1 \dots \alpha_m \alpha_{m+1} \gamma_{m+1}.$$

Case $\alpha_{m+1} = a \in T$
Then we used transition type (2) and the transition has the form

$$(q, \gamma_m) = (q, av) \xrightarrow{a} (q, v) = (q, \gamma_{m+1})$$

for some $v \in (N \cup T)^*$. Then it holds that

$$A \vdash_G^* \alpha_1 \dots \alpha_m av = \alpha_1 \dots \alpha_m \alpha_{m+1} \gamma_{m+1}.$$

Thus we have also proved statement 2.

Particularly from the statements 1 and 2 it follows that for all strings $w \in T^*$ it holds that:

$$\begin{aligned} S \vdash_G^* w \quad & \text{if and only if } (q, S) \overset{w}{\Rightarrow} (q, \varepsilon) \\ & \text{if and only if } \mathcal{K} \text{ accepts } w \text{ with the empty stack.} \end{aligned}$$

Thus $L(G) = L_\varepsilon(\mathcal{K})$ holds as required. □

Example : We consider again the language $L = \{a^n b^n \mid n \in \mathbb{N}\}$. We have already seen that the language is generated by the context-free grammar $G_1 = (\{S\}, \{a, b\}, P_1, S)$, where P_1 consists of the productions

$$S \to \varepsilon \mid aSb,$$

i.e. $L(G_1) = L$. The construction used in the proof above gives us the push-down automaton

$$\mathcal{K}_1 = (\{a, b\}, \{q\}, \{S, a, b\}, \to, q, S, \emptyset),$$

where the transition relation $\to$ consists of the following transitions:

$$\begin{array}{lcl} (q, S) & \xrightarrow{\varepsilon} & (q, \varepsilon) \\ (q, S) & \xrightarrow{\varepsilon} & (q, aSb) \\ (q, a) & \xrightarrow{a} & (q, \varepsilon) \\ (q, b) & \xrightarrow{b} & (q, \varepsilon) \end{array}$$

It follows from the proof that $L_\varepsilon(\mathcal{K}_1) = L(G_1)$. To illustrate this, let us consider the transition sequence of $\mathcal{K}_1$ while accepting a^2b^2:

$$\begin{array}{lcllcl} (q, S) & \xrightarrow{\varepsilon} & (q, aSb) & \xrightarrow{a} & (q, Sb) \\ & \xrightarrow{\varepsilon} & (q, aSbb) & \xrightarrow{a} & (q, Sbb) \\ & \xrightarrow{\varepsilon} & (q, bb) & \xrightarrow{b} & (q, b) \xrightarrow{b} (q, \varepsilon). \end{array}$$

Now for every given push-down automaton we construct a respective context-free grammar.

3.7 **Theorem** : For every push-down automaton $\mathcal{K}$ we can construct a context-free grammar G with $L(G) = L_\varepsilon(\mathcal{K})$.

Proof : Let $\mathcal{K} = (\Sigma, Q, \Gamma, \to, q_0, Z_0, F)$. We construct $G = (N, T, P, S)$ with $T = \Sigma$ and

$$N = \{S\} \cup \{[q, Z, q'] \,|\, q, q' \in Q \text{ and } Z \in \Gamma\}.$$

The idea of non-terminal symbols $[q, Z, q']$ is as follows:

(1) All strings $w \in \Sigma^*$ which $\mathcal{K}$ can accept from the configuration (q, Z) with empty stack and the state q' should be generated in G from $[q, Z, q']$: $(q, Z) \overset{w}{\Rightarrow} (q', \varepsilon)$.

(2) Thus a transition $(q, Z) \overset{\alpha}{\to} (r_0, Z_1 \dots Z_k)$ from $\mathcal{K}$ is simulated in G by the following productions:

$$[q, Z, r_k] \to \alpha[r_0, Z_1, r_1][r_1, Z_2, r_2] \dots [r_{k-1}, Z_k, r_k]$$

for each $r_1, \dots, r_k \in Q$. The strings which are accepted by $\mathcal{K}$ up to the reduction of the symbol Z_1 are generated from $[r_0, Z_1, r_1]$; the strings accepted by $\mathcal{K}$ up to the reduction of the symbol Z_2 are generated from $[r_1, Z_2, r_2]$, and so on. The intermediate states $r_1, \dots, r_{k-1}$ are those states which $\mathcal{K}$ reaches directly after the reduction of the symbols $Z_1, \dots, Z_{k-1}$.

To make it more precise, P consists of the following transitions:

- **Type (1):** $S \to [q_0, Z_0, r] \in P$ for all $r \in Q$,
- **Type (2):** For every transition $(q, Z) \overset{\alpha}{\to} (r_0, Z_1 \dots Z_k)$ with $\alpha \in \Sigma \cup \{\varepsilon\}$ and $k \geq 1$ in $\mathcal{K}$: $[q, Z, r_k] \to \alpha[r_0, Z_1, r_1] \dots [r_{k-1}, Z_k, r_k] \in P$ for all $r_1, ..., r_k \in Q$.
- **Type (3):** (Special case (2) for $k = 0$.) For every transition $(q, Z) \overset{\alpha}{\to} (r_0, \varepsilon)$ in $\mathcal{K}$: $[q, Z, r_0] \to \alpha \in P$.

To show that $L(G) = L_\varepsilon(\mathcal{K})$ holds, consider the relationship between derivations in G and transition sequences in $\mathcal{K}$.

Hypothesis 1 For all $q, q' \in Q,\ Z \in \Gamma,\ w \in \Sigma^*,\ n \geq 1$ and derivations in G

$$[q, Z, q'] \underbrace{\vdash_G \dots \vdash_G}_{\leq n\text{-times}} w$$

with the length $\leq n$ holds for $\mathcal{K}$

$$(q, Z) \overset{w}{\Rightarrow} (q', \varepsilon).$$

Proof by induction on n:

$\underline{n = 1:}$ From $[q, Z, q'] \vdash_G w$ it follows (because $w \in \Sigma^*$) that we deal with the production type (3) in G. Therefore, it holds that $w = \alpha \in \Sigma \cup \{\varepsilon\}$ and $(q, Z) \overset{\alpha}{\to} (q', \varepsilon)$. Thus $(q, Z) \overset{\alpha}{\Rightarrow} (q', \varepsilon)$.

$\underline{n \to n+1:}$ We analyse the first step of a derivation with the length $n + 1$, which should take place according to the production type (2):

$$[q, Z, r_k] \vdash_G \alpha\, [r_0, Z_1, r_1] \dots [r_{k-1}, Z_k, r_k] \underbrace{\vdash_G \dots \vdash_G}_{n\text{-times}} \alpha\, w_1 \dots w_k = w,$$

where $(q, Z) \overset{\alpha}{\to} (r_0, Z_1 \dots Z_k)$ in $\mathcal{K}$, $r_k = q',\ \alpha \in \Sigma \cup \{\varepsilon\},\ w_1, \dots, w_k \in \Sigma^*$ and

$$[r_{i-1}, Z_i, r_i] \underbrace{\vdash_G \dots \vdash_G}_{\leq n\text{-times}} w_i$$

for $i = 1, \dots, k$ holds. Due to induction it holds in $\mathcal{K}$ that

$$(r_{i-1}, Z_i) \overset{w_i}{\Rightarrow} (r_i, \varepsilon)$$

for $i = 1, \dots, k$ and thus according to the Top Lemma

$$\begin{aligned} (q, Z) \overset{\alpha}{\to} \quad (r_0, Z_1 \dots Z_k) &\overset{w_1}{\Rightarrow} \quad (r_1, Z_2 \dots Z_k) \\ &\vdots \\ (r_{k-1}, Z_k) &\overset{w_k}{\Rightarrow} \quad (r_k, \varepsilon). \end{aligned}$$

So in conclusion $(q, Z) \overset{w}{\Rightarrow} (q', \varepsilon)$ for $\mathcal{K}$ as required.

Hypothesis 2 For all $q, q' \in Q,\ Z \in \Gamma,\ n \geq 1,\ \alpha_1, \dots, \alpha_n \in \Sigma \cup \{\varepsilon\}$ and all transition sequences

$$(q, Z) \overset{\alpha_1}{\to} \circ \dots \circ \overset{\alpha_n}{\to} (q', \varepsilon)$$

in $\mathcal{K}$ with the length n it holds in G that

$$[q, Z, q'] \vdash_G^* \alpha_1 \dots \alpha_n.$$

Proof by induction on n:

$\underline{n = 1:}$ Then $(q, Z) \overset{\alpha_1}{\to} (q', \varepsilon)$ holds. By definition of P in G — see production type (3) — it follows that $[q, Z, q'] \vdash_G \alpha_1$.

$\underline{n \to n+1:}$ We analyse the first step of a transition sequence in $\mathcal{K}$ with the length $n + 1$:

$$(q, Z) \overset{\alpha_1}{\to} (r_0, Z_1 \dots Z_k) \overset{\alpha_2}{\to} \circ \dots \circ \overset{\alpha_{n+1}}{\longrightarrow} (q', \varepsilon),$$

where $k \geq 1$ holds. By definition of P there is a production of type (2) in G

$$[q, Z, q'] \to \alpha_1 [r_0, Z_1, r_1] \dots [r_{k-1}, Z_k, r_k],$$

where $r_k = q'$. We consider in more detail the successive reduction of the stack contents $Z_1 \dots Z_k$ of $\mathcal{K}$. Furthermore, there exist transition sequences in $\mathcal{K}$

$$\begin{aligned} &(r_0, Z_1) \overset{\alpha_{11}}{\longrightarrow} \circ \dots \circ \overset{\alpha_{1m_1}}{\longrightarrow} (r_1, \varepsilon) \\ &\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots \\ &(r_{k-1}, Z_k) \overset{\alpha_{k1}}{\longrightarrow} \circ \dots \circ \overset{\alpha_{km_k}}{\longrightarrow} (r_k, \varepsilon) = (q', \varepsilon) \end{aligned}$$

with $\alpha_2 \dots \alpha_{n+1} = \alpha_{11} \dots \alpha_{1m_1} \dots \alpha_{k1} \dots \alpha_{km_k}$ and $m_1, \dots, m_k \leq n$. By condition of induction it holds in G that

$$[r_{i-1}, Z_i, r_i] \vdash_G^* \alpha_{i1} \dots \alpha_{im_i}$$

for $i = 1, \dots, k$. In conclusion it results in G that

$$[q, Z, q'] \vdash_G^* \alpha_1 \alpha_2 \dots \alpha_{n+1}$$

as required.

The hypotheses 1 and 2 imply: for all $q \in Q$ and $w \in \Sigma^*$ it holds in G

$$S \vdash_G [q_0, Z_0, q] \vdash_G^* w$$

if and only if in $\mathcal{K}$

$$(q_0, Z_0) \overset{w}{\Rightarrow} (q, \varepsilon)$$

holds. Thus we have shown that $L(G) = L_\varepsilon(\mathcal{K})$. □

§4 Closure properties

Now we consider, under which operations the class of context-free languages is closed. In contrast to the regular (i.e. finitely acceptable) languages we have the following results.

4.1 **Theorem** : The class of context-free languages is closed under the following operations

(i) union,

(ii) concatenation,

(iii) iteration,

(iv) intersection with regular languages.

However, the class of context-free languages is *not* closed under the operations

(v) intersection,

(vi) complement.

Proof : Let $L_1, L_2 \subseteq T^*$ be context-free. Then there are context-free grammars $G_i = (N_i, T, P_i, S_i)$ with $L(G_i) = L_i$, where $i = 1, 2$ and $N_1 \cap N_2 = \emptyset$. First we show that $L_1 \cup L_2$, $L_1 \cdot L_2$ and L_1^* are context-free. After that we consider the operations intersection and complement. Let $S \notin N_1 \cup N_2$ be a new start symbol.

(i) $L_1 \cup L_2$: Consider the context-free grammar $G = (\{S\} \cup N_1 \cup N_2, T, P, S)$ with $P = \{S \to S_1, S \to S_2\} \cup P_1 \cup P_2$. Obviously $L(G) = L_1 \cup L_2$ holds.

(ii) $L_1 \cdot L_2$: Consider the context-free grammar $G = (\{S\} \cup N_1 \cup N_2, T, P, S)$ with $P = \{S \to S_1 S_2\} \cup P_1 \cup P_2$. Obviously $L(G) = L_1 \cdot L_2$ holds.

(iii) L_1^*: Consider the context-free grammar $G = (\{S\} \cup N_1, T, P, S)$ with $P = \{S \to \varepsilon, S \to S_1 S\} \cup P_1$. Then $S \vdash_G^* S_1^n$ holds for all $n \geq 0$ and thus $L(G) = L_1^*$.

(iv) $L_1 \cap \mathit{regSpr}$: For the intersection with *regular* languages we use the representation of context-free and regular languages by push-down automata and finite automata, respectively. Let $L_1 = L(\mathcal{K}_1)$ for the (nondeterministic) PDA $\mathcal{K}_1 = (T, Q_1, \Gamma, \to_1, q_{01}, Z_0, F_1)$ and $L_2 = L(\mathcal{A}_2)$ for the DFA $\mathcal{A}_2 = (T, Q_2, \to_2, q_{02}, F_2)$. We construct from $\mathcal{K}_1$ and $\mathcal{A}_2$ the (nondeterministic) PDA

$$\mathcal{K} = (T, Q_1 \times Q_2, \Gamma, \to, (q_{01}, q_{02}), F_1 \times F_2),$$

where the transition relation $\to$ for $q_1, q_1' \in Q_1, q_2, q_2' \in Q_2, Z \in \Gamma, \alpha \in T \cup \{\varepsilon\}$ and $\gamma' \in \Gamma^*$ is defined as follows:

$$((q_1, q_2), Z) \xrightarrow{\alpha} ((q_1', q_2'), \gamma') \text{ in } \mathcal{K}$$

if and only if

$$(q_1, Z) \overset{\alpha}{\to}_1 (q_1', \gamma') \text{ in } \mathcal{K}_1 \text{ and } q_2 \overset{\alpha}{\to}_2 q_2' \text{ in } \mathcal{A}_2.$$

Note that in the special case $\alpha = \varepsilon$ the notation $q_2 \overset{\varepsilon}{\to}_2 q_2'$ for the DFA $\mathcal{A}_2$ simply means $q_2 = q_2'$ (compare with the definition of the extended transition relation $q \overset{w}{\to}_2 q'$ for DFA's in section 1). Thus the relation $\overset{\alpha}{\to}$ of $\mathcal{K}$ models the *synchronous parallel progress* of the automata $\mathcal{K}_1$ and $\mathcal{A}_2$. However, in the special case $\alpha = \varepsilon$ only $\mathcal{K}_1$ makes a spontaneous ε-transition, while the DFA $\mathcal{A}_2$ remains in the current state.

We show that for the acceptance by final states it holds that: $L(\mathcal{K}) = L(\mathcal{K}_1) \cap L(\mathcal{A}_2) = L_1 \cap L_2$. Let $w = a_1 \dots a_n \in T^*$, where $n \geq 0$ and $a_i \in T$ for $i = 1, \dots, n$. Then it holds that:

$$\begin{aligned}
w \in L(\mathcal{K}) \quad &\Leftrightarrow \quad \exists (q_1, q_2) \in F, \gamma \in \Gamma^* : && ((q_{01}, q_{02}), Z_0) \overset{a_1}{\Rightarrow} \circ \dots \circ \overset{a_n}{\Rightarrow} ((q_1, q_2), \gamma) \\
&\Leftrightarrow \quad \exists q_1 \in F_1, q_2 \in F_2, \gamma \in \Gamma^* : && (q_{01}, Z_0) \overset{a_1}{\Rightarrow}_1 \circ \dots \circ \overset{a_n}{\Rightarrow}_1 (q_1, \gamma) \\
& && \text{and } q_{02} \overset{a_1}{\to}_2 \circ \dots \circ \overset{a_n}{\to}_2 q_2 \\
&\Leftrightarrow \quad w \in L(\mathcal{K}_1) \cap L(\mathcal{A}_2). &&
\end{aligned}$$

(v) *not* $L_1 \cap L_2$: However, the context-free languages are *not* closed under intersection with other context-free languages. Consider

$$L_1 = \{a^m b^n c^n \mid m, n \geq 0\}$$

and

$$L_2 = \{a^m b^m c^n \mid m, n \geq 0\}.$$

It is easy to see that L_1 and L_2 are context-free. For example, L_1 can be generated by the context-free grammar $G = (\{S, A, B\}, \{a, b, c\}, P, S\}$ with the following P:

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

The intersection of L_1 and L_2 is the following language

$$L_1 \cap L_2 = \{a^n b^n c^n \mid n \geq 0\},$$

which is not context-free (as we have shown with the help of the pumping lemma).

(vi) *not* $\overline{L_i}$: The context-free languages are also *not* closed under the complement. This statement follows directly from (i) and (v) due to the De Morgan's laws. If context-free languages were closed against the complement, then they would also be closed under intersection, because $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$ holds.

□

§5 Transformation in normal forms

While considering context-free languages, it is beneficial when the rules of the underlying context-free grammars are as simple as possible. Therefore, in this section we introduce such transformations, which transform the given context-free grammars into equivalent grammars with the rules satisfying additional conditions.

5.1 **Definition** : An *ε-production* is a rule of the form $A \to \varepsilon$. A context-free grammar $G = (N, T, P, S)$ is called *ε-free* if in G there is

(i) either no ε-production at all

(ii) or only the ε-production $S \to \varepsilon$ and then S does not occur on the right-hand side of any production in G.

In case (i) it holds that $\varepsilon \notin L(G)$ and in case (ii) it holds that $\varepsilon \in L(G)$.

5.2 **Theorem** : Every context-free grammar can be transformed into an equivalent ε-free grammar.

Proof : Let $G = (N, T, P, S)$ be context-free. G may contain ε-productions, otherwise we have nothing to do. A symbol $A \in N$ is called *erasable* if $A \vdash_G^* \varepsilon$ holds.

Step 1 First we compute all erasable $A \in N$. For this purpose we inductively compute the sets $N_1, N_2, N_3, \ldots$ of erasable non-terminal symbols:

$$\begin{aligned} N_1 &= \{A \in N \mid A \to \varepsilon \in P\} \\ N_{k+1} &= N_k \cup \{A \in N \mid A \to B_1 \ldots B_n \in P \text{ mit } B_1, \ldots, B_n \in N_k\} \end{aligned}$$

These sets represent an ascending sequence bounded from above:

$$N_1 \subseteq N_2 \subseteq N_3 \subseteq \ldots \subseteq N.$$

Because N is finite, there is a minimal k_0 with $N_{k_0} = N_{k_0+1}$.

Statement: $A \in N_{k_0} \iff A$ is erasable.
"$\Rightarrow$" is obvious from the definition of N_{k_0}.
"$\Leftarrow$" is shown by induction on the depth of the derivation tree from A to ε.

Step 2 We construct a grammar $G' = (N', T, P', S')$ equivalent to G. Let S' be a new start symbol $N' = \{S'\} \cup N$. The set of productions P' is defined in two steps. First we introduce the set P_0 generated from P by replacing every production

$$A \to \beta_1 \ldots \beta_n \in P$$

with $\beta_1, \ldots, \beta_n \in N \cup T$ and $n \geq 1$ by the production of the form

$$A \to \alpha_1 \ldots \alpha_n,$$

where the following holds:

- If $\beta_i \in N$ is erasable, then $\alpha_i = \varepsilon$ or $\alpha_i = \beta_i$.
- If $\beta_i \in T$ or $\beta_i \in N$ are non-erasable, then $\alpha_i = \beta_i$.
- Not all α_i's are ε.

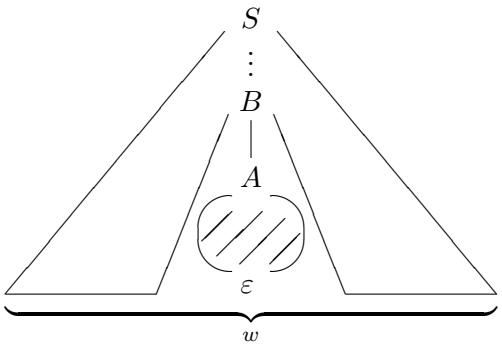
Then it holds that: P_0 contains no ε-productions and $P - \{A \to \varepsilon \mid A \in N\} \subseteq P_0$. Then we get P' from P_0 as follows:

$$P' = \{S' \to \varepsilon \mid S \text{ is erasable } \} \cup \{S' \to u \mid S \to u \in P_0\} \cup P_0$$

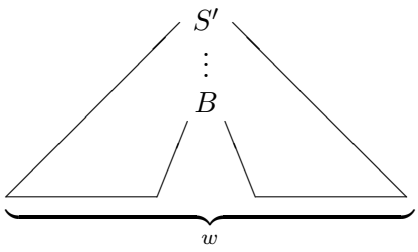
Thus G' is defined completely. It remains to show that: $L(G') = L(G)$.

"$\subseteq$": This inclusion is obvious, because $S' \to u \in P'$ implies $S \vdash_G^* u$, and $A \to u \in P'$ with $A \in N$ implies $A \vdash_G^* u$.

"$\supseteq$": If $\varepsilon \in L(G)$ holds, then S is erasable and thus $S' \to \varepsilon \in P'$. Therefore, $\varepsilon \in L(G')$ holds as well. Now let $w \in L(G) - \{\varepsilon\}$. Consider a derivation tree from S to w in G:



We get a derivation tree from S' to w in G' by replacing S by S' and by removing all maximal subtrees which represent the derivations of the form $A \vdash_G^* \varepsilon$:



Thus $w \in L(G')$ holds. □

5.3 **Definition** : A context-free grammar $G = (N, T, P, S)$ is in *Chomsky normal form* if the following holds:

- G is ε-free (so that at the most $S \to \varepsilon \in P$ is allowed),
- every production in P different from $S \to \varepsilon$ has the form

$$A \to a \quad \text{or} \quad A \to BC,$$

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

5.4 **Theorem** : Every context-free grammar can be transformed into an equivalent grammar in Chomsky normal form.

Proof : see literature. □

5.5 **Definition** : A context-free grammar $G = (N, T, P, S)$ is in *Greibach normal form* if the following holds:

- G is ε-free (so that at the most $S \to \varepsilon \in P$ is allowed),
- every production in P different from $S \to \varepsilon$ has the form

$$A \to aB_1 \dots B_k,$$

  where $k \geq 0, A, B_1, \dots, B_k \in N$ and $a \in T$.

5.6 **Theorem** : Every context-free grammar can be transformed into an equivalent grammar in Greibach normal form.

Proof : see literature. □

§6 Deterministic context-free languages

In section 3 we have shown that in general every context-free language can be recognized by a nondeterministic push-down automaton. The question is, whether we can eliminate the nondeterminism as in the case of finite automata and whether we are always able to construct equivalent deterministic push-down automata. This question is of practical importance for the construction of a parser for a given context-free language. First we define the notion of determinism for push-down automata and context-free languages.

6.1 **Definition** :

(i) A *push-down automaton* $\mathcal{K} = (\Sigma, Q, \Gamma, \to, q_0, Z_0, F)$ is called *deterministic* if the transition relation $\to$ fulfills the following conditions:

$\forall q \in Q,\ Z \in \Gamma,\ a \in \Sigma$:

(number of transitions of the form $(q, Z) \xrightarrow{a} \cdots$

+ number of transitions of the form $(q, Z) \xrightarrow{\varepsilon} \cdots$) ≤ 1

(ii) A context-free *language* L is called *deterministic* if there is a deterministic push-down automaton $\mathcal{K}$ with $L = L(\mathcal{K})$ (acceptance by final states).

Example : The language $L = \{a^n b^n \mid n \in \mathbb{N}\}$ is deterministic context-free, because in section 3 we have introduced a deterministic push-down automaton $\mathcal{K}$ with $L(\mathcal{K}) = L$.

Example : The language $PAL_c = \{wcw^R \mid w \in \{a, b\}^*\}$ of palindromes with c as symbol in the middle is also deterministic context-free. The notation w^R means that the string w should be read backwards.

Theorem 3.5 does not hold for deterministic push-down automata. Thus we cannot replace $L(\mathcal{K})$ (acceptance by final states) by $L_\varepsilon(\mathcal{K})$ (acceptance by empty stack).

Now we show that not all context-free languages are deterministic. For this purpose we use the following theorem.

6.2 **Theorem** : Deterministic context-free languages are closed under complementation.

Proof sketch: We could use the same approach as in the case of finite automata and consider the deterministic PDA $\mathcal{K}' = (\Sigma, Q, \Gamma, \to, q_0, Z_0, Q - F)$ for a deterministic PDA $\mathcal{K} = (\Sigma, Q, \Gamma, \to, q_0, Z_0, F)$. Unfortunately, it generally holds that $L(\mathcal{K}') \subsetneq \Sigma^* - L(\mathcal{K})$. The reason for the fact that not all strings from the complement of $L(\mathcal{K})$ are accepted is the existence of non-terminating computations. For example, if a transition

$$(q, A) \xrightarrow{\varepsilon} (q, AA)$$

is used once, then it must be used repeatedly; then infinitely many elements will be added to the stack and $\mathcal{K}$ will not terminate. If this is the case for some input string $w \in \Sigma^*$, then $w \notin L(\mathcal{K})$. However the same holds for $\mathcal{K}'$, i.e. $w \notin L(\mathcal{K}')$.

Thus we must first transform $\mathcal{K}$ into an equivalent deterministic PDA which terminates after finitely many steps for *every* input string. Such a construction is actually possible for push-down

automata, because the set

$$\{(q, A) \,|\, \exists \gamma \in \Gamma^* \text{ with } (q, A) \stackrel{\varepsilon}{\Rightarrow} (q, A\gamma)\}$$

can be effectively constructed for $\mathcal{K}$ and the respective transitions $(q, A) \stackrel{\varepsilon}{\to} (q', \gamma')$ can be deleted or replaced by appropriate transitions.

Details: think yourself or refer to literature. □

6.3 **Corollary** : There exist context-free languages that are not deterministic.

Proof : If all context-free languages were deterministic, then the context-free languages would be closed under complementation. Contradiction to the theorem from section 4.1 □

6.4 **Lemma** : Deterministic context-free languages are

(i) closed under intersection with regular languages,

(ii) not closed under union, intersection, concatenation and iteration.

Proof : We prove (i) using the same construction for the nondeterministic case (see section 4.1); the PDA $\mathcal{K}$ generated from $\mathcal{K}_1$ and $\mathcal{A}_2$ is deterministic if $\mathcal{K}_1$ is deterministic.

(ii) The context-free languages $L_1 = \{a^m b^n c^n \,|\, m, n \geq 0\}$ and $L_2 = \{a^m b^m c^n \,|\, m, n \geq 0\}$ are both deterministic, however, their intersection is not even context-free. Because $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$ holds, the deterministic context-free languages are not closed under union. For concatenation and iteration refer to literature. □

As an example for a nondeterministic context-free language, we consider

$$PAL = \{ww^R \,|\, w \in \{a, b\}^* \wedge w \neq \varepsilon\},$$

the language of all non-empty *palindromes with even length*. The notation w^R means that the string w should be read backwards. PAL is obviously context-free: to generate it we need the following rules

$$S \to aa \,|\, bb \,|\, aSa \,|\, bSb.$$

In order to show that PAL is nondeterministic we use an auxiliary operator *Min*.

6.5 **Definition** : For a language $L \subseteq \Sigma^*$ let

$$Min(L) = \{w \in L \mid \text{ there is no strict prefix } v \text{ of } w \text{ with } v \in L\},$$

where v is a *strict prefix* of w if $v \neq w$ and $\exists u \in \Sigma^* : w = v \cdot u$.

6.6 **Lemma** : If L with $\varepsilon \notin L$ is a deterministic context-free language, then so is $Min(L)$ as well.

Proof : Consider a deterministic PDA $\mathcal{K} = (\Sigma, Q, \Gamma, \rightarrow, q_0, Z_0, F)$ with $L(\mathcal{K}) = L$. Because $\varepsilon \notin L$, it holds as well $q_0 \notin F$. We transform $\mathcal{K}$ into PDA $\mathcal{K}_1$ that functions as $\mathcal{K}$, but in every transition sequence it reaches one of the final states from F *at most once* and then stops immediately. Let us consider a new state $q_1 \notin Q$ and define $\mathcal{K}_1 = (\Sigma, Q \cup \{q_1\}, \Gamma, \rightarrow_1, q_0, Z_0, \{q_1\})$ with

$$\begin{aligned} \rightarrow_1 &= \{(q, Z, \alpha, q', \gamma') \mid q \in Q - F \text{ and } (q, Z, \alpha, q', \gamma') \in \rightarrow\} \\ &\cup \{(q, Z, \varepsilon, q_1, Z) \mid q \in F \text{ and } Z \in \Gamma\} \end{aligned}$$

$\mathcal{K}_1$ is deterministic and $L(\mathcal{K}_1) = Min(L(\mathcal{K}))$ holds, because $\mathcal{K}$ is deterministic. □

Now we show:

6.7 **Theorem** : The context-free language PAL is nondeterministic.

Proof : *Hypothesis: PAL* is deterministic. Then according to both lemmas considered above the language

$$L_0 = Min(PAL) \cap L((ab)^+(ba)^+(ab)^+(ba)^+)$$

is context-free and deterministic. $(ab)^+$ stands for the regular expression $ab(ab)^*$ and $(ba)^+$ similarly for the regular expression $ba(ba)^*$. Because all strings in L_0 are palindromes with even length without strict prefix, it holds that

$$L_0 = \{(ab)^i(ba)^j(ab)^j(ba)^i \mid i > j > 0\}.$$

According to the pumping lemma there exists a number $n \in \mathbb{N}$ for L_0 with the properties given in the pumping lemma. Then we can decompose the string

$$z = (ab)^{n+1}(ba)^n(ab)^n(ba)^{n+1} \in L_0$$

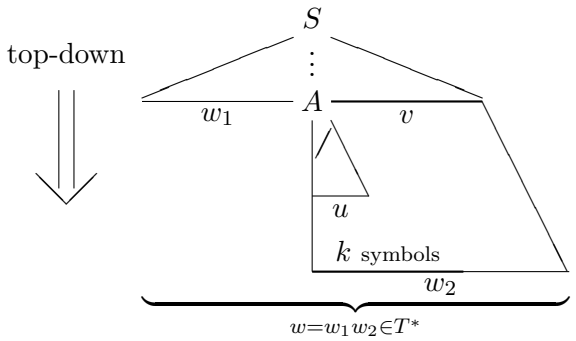
into $z = uvwxy$. Because the intermediate part vwx fulfills the conditions $|vwx| \leq n$ and $vx \neq \varepsilon$, we can show that not all strings of the form uv^iwx^iy with $i \in \mathbb{N}$ can be in L_0. Therefore, L_0 is not even context-free, not to mention deterministic context-free. *Contradiction* □

Remark: Because L_0 is not context-free, the closure properties imply that context-free languages are not closed under the operator *Min*.

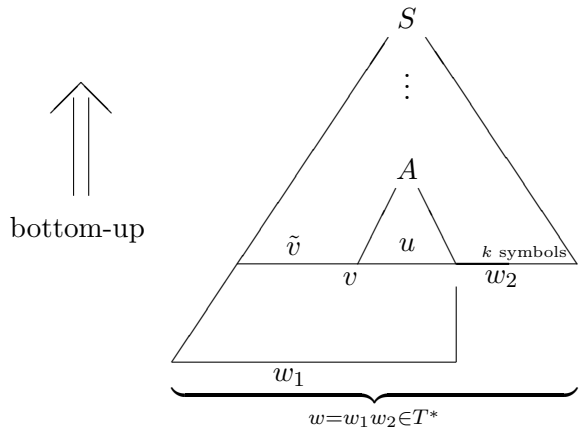
For the practical syntax analysis of programming languages we will use the deterministic context-free languages. There are two different methods of syntax analysis: in the top-down method

we construct the derivation trees from the start symbol of the grammar and in the bottom-up method we reconstruct from the given string. We want to be able to determine unambiguously the next derivation step by looking k symbols ahead for some $k \geq 0$.

For the top-down method we can use the so called $LL(k)$-*grammars*. These are context-free grammars $G = (N, T, P, S)$ where for every intermediate string w_1Av in a leftmost derivation $S \vdash_G^* w_1Av \vdash_G^* w_1w_2 = w$ of a string $w \in T^*$ from S the part Av and the first k symbols of the remainder w_2 of w define *unambiguously* the next left derivation step $w_1Av \vdash_G w_1uv$. This $LL(k)$-condition can be graphically represented as follows:



For the bottom-up method we can use the so called $LR(k)$-*grammars*. These are context-free grammars $G = (N, T, P, S)$ with the following property. For every intermediate string vw_2 in the bottom-up reconstruction of a right derivation $S \vdash_G^* vw_2 \vdash_G^* w_1w_2 = w$ of a string $w \in T^*$ from S, the part v and the first k symbols of the rest w_2 define unambiguously the *previous* right derivation step $\tilde{v}Aw_2 \vdash_G \tilde{v}uw_2$ with $\tilde{v}u = v$ and $A \to u \in P$. This $LR(k)$-condition can be graphically represented as follows:



$LL(k)$ grammars were introduced in 1968 by P.M. Stearns and R.E. Stearns and $LR(k)$ grammars in 1965 by D.E. Knuth. The following relations hold for languages $LL(k)$ and $LR(k)$ generated by these grammars:

- $\left(\bigcup\limits_{k\geq 0} LL(k) \text{ languages}\right) \subsetneq \left(\bigcup\limits_{k\geq 0} LR(k) \text{ languages}\right) =$ deterministic context-free languages

- It even holds that: $LR(1)$ languages = deterministic context-free languages

§7 Questions of decidability

The following constructions are algorithmically computable:

- PDA acceptable by final states $\leftrightarrow$ PDA acceptable by empty stack
- PDA $\leftrightarrow$ context-free grammar
- context-free grammar $\mapsto$ ε-free grammar
- context-free grammar $\mapsto$ Chomsky or Greibach normal form

The questions of decidability concerning context-free languages can be answered using both representation by context-free grammars (in normal form) or by push-down automata. We consider the same problems for regular languages (compare with chapter II, section 5).

7.1 **Theorem (decidability)**: For context-free languages

- the membership problem,
- the emptiness problem,
- the finiteness problem

are decidable.

Proof :
Membership problem: Consider a context-free grammar $G = (N, T, P, S)$ in Greibach normal form and a string $w \in T^*$. The question is: Does $w \in L(G)$ hold? The case $w = \varepsilon$ can be decided immediately, because G is ε-free: $\varepsilon \in L(G) \iff S \to \varepsilon \in P$. Now let $w \neq \varepsilon$. Then the following holds:

$$\begin{aligned} w \in L(G) \quad &\Leftrightarrow \quad \exists n \geq 1 : S \underbrace{\vdash_G \ldots \vdash_G}_{n\text{-times}} w \\ &\Leftrightarrow \quad \{ \text{ In Greibach normal form every derivation step produces exactly one character of } w. \ \} \\ & \qquad S \underbrace{\vdash_G \ldots \vdash_G}_{|w|\text{-times}} w \end{aligned}$$

In order to decide $w \in L(G)$ it is enough to check all derivation sequences with the length $|w|$ in G. This implies the decidability of the decision problem.

Emptiness problem: Consider a context-free grammar $G = (N, T, P, S)$. The question is: Does $L(G) = \emptyset$ hold? Let n be the number which belongs to the context-free language $L(G)$

according to the pumping lemma. The same way as in the case of regular languages we show that:

$$L(G) = \emptyset \iff \neg\exists w \in L(G) : |w| < n.$$

Thus we have solved the emptiness problem by solving the decision problem for all strings $w \in T^*$, where $|w| < n$.

Finiteness problem: Consider a context-free grammar $G = (N, T, P, S)$. The question is: Is $L(G)$ finite? Let n be the same as above. Then we show the same way as in the case of regular languages:

$$L(G) \text{ is finite} \iff \neg\exists w \in L(G) : n \leq |w| < 2 \cdot n$$

Therefore, the finiteness problem can be decided by solving the membership problem a finite number of times. □

However, unlike the regular languages the following result holds.

7.2 **Theorem (undecidability)**: For context-free languages

- the intersection problem,
- the equivalence problem,
- the inclusion problem

are undecidable.

We can prove this theorem only after formalizing the notion of algorithm.

Another result of undecidability concerns the ambiguity of context-free grammars. For the practical application of context-free grammars for describing the syntax of programming languages, it would be beneficial to have an algorithmical test to check the ambiguity of context-free grammars. However, we will show later that such a test does not exist.

7.3 **Theorem** : It is undecidable whether a given context-free grammar is ambiguous.

In practice we can easily avoid the problem of having to test the ambiguity by restricting oneself to $LR(1)$ grammars or its subclasses. The $LR(1)$-property is algorithmically decidable and because LR-grammars are always unambiguous (because the last rightmost derivation step must always be defined unambiguously and thus every string can have only *one* right derivation) the problem of ambiguity does not exist.

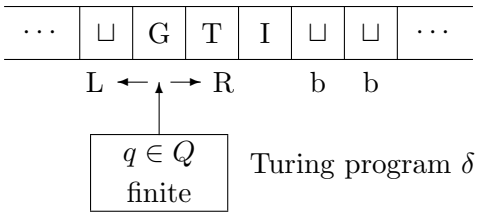
Chapter IV

The notion of algorithm: What can be computed using machines?

§1 Turing machines

Turing machines were introduced in 1936 by A.M. Turing (1912–1954). We consider an elementary model for calculating with pencil and paper. For this purpose we need a tape, where we can write and change the characters, and a finite program.

Sketch



L: means: move one square left on tape.
R: similarly to the right.
S: no movement.

1.1 **Definition** : A *Turing machine*, shortly TM, is a 6-tuple $\tau = (Q, \Sigma, \Gamma, \delta, q_0, \sqcup)$ with the following properties:

(i) Q is a finite non-empty set of *states*,
(ii) $q_0 \in Q$ is the start state,
(iii) Γ is a finite non-empty set, the *tape alphabet*, with $Q \cap \Gamma = \emptyset$,
(iv) $\Sigma \subseteq \Gamma$ is the input alphabet,
(v) $\sqcup \in \Gamma - \Sigma$ is the *blank symbol* or *blank*,

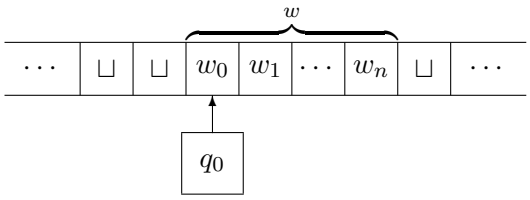
(vi) $\delta : Q \times \Gamma \xrightarrow{part} Q \times \Gamma \times \{R, L, S\}$ is the *transition function*.
It can be represented as a *Turing table* or *Turing program*:

$$\delta : \begin{array}{c|c} q_1 a_1 & q_1' a_1' P_1 \\ \dots & \dots \\ \dots & \dots \\ \dots & \dots \\ \dots & \dots \\ q_n a_n & q_n' a_n' P_n \end{array} \quad \text{with} \quad \begin{array}{l} q_i, q_i' \in Q, \\ a_i, a_i' \in \Gamma, \\ P_i \in \{R, L, S\} \end{array}$$

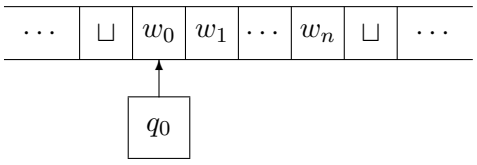
Operating principles of a TM: informal

A configuration of the TM describes the current state, the contents of the tape and the considered cell.

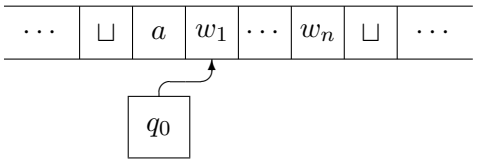
- Initial configuration:



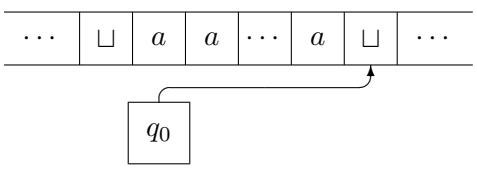
- Making a step:



e.g. $\delta(q_0, w_0) = (q_0, a, R)$ leads to



- Repetition of this step gives us the following result:



where $\delta(q_0, \sqcup)$ is undefined.
Then as the result of the computation we get a string $aa \ldots a$, i.e. the contents of the tape without blanks.

Example : Let us compute the function
even: $\{\,|\,\}^* \longrightarrow \{0, 1\}$
with

$$even(\,|^n\,) \quad = \quad \begin{cases} 1 & \text{if } n \text{ can be divided by 2} \\ 0 & \text{otherwise} \end{cases}$$

for $(n \geq 0)$. Choose

$$\tau = (\{q_0, q_1, q_e\}, \{\,|\,\}, \{\,|, 0, 1, \sqcup\}, \delta, q_0, \sqcup)$$

using the following Turing table:

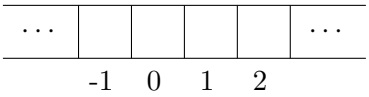
$$\delta: \quad \begin{array}{cc|ccc} q_0 & | & q_1 & \sqcup & R \\ q_0 & \sqcup & q_e & 1 & S \\ q_1 & | & q_0 & \sqcup & R \\ q_1 & \sqcup & q_e & 0 & S \end{array}$$

We can easily check that τ computes the function *even*.

Notion of the configuration

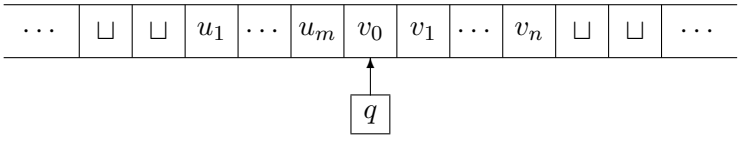
We consider the current state q, the current contents of the tape and the current position of the writing/reading head. Usually two kinds of notations are used:

(1) We number the squares of the infinite tape sequentially:



Contents of the tape:	$f : \mathbb{Z} \longrightarrow \Gamma$ ($\mathbb{Z}$ = Set of integers)
Configuration:	triple (q, f, i) with $i \in \mathbb{Z}$
Disadvantage:	complicated manipulation

(2) Only a finite part of the tape differs from $\sqcup$. Abstraction of blanks and cells' numbers. The configuration



can be unambiguously represented as a string $\overbrace{u_1 \dots u_m}^{u} q \overbrace{v_0 v_1 \dots v_n}^{v}$ with $u_i \in \Gamma, v_j \in \Gamma, m, n \geq 0$.
Note: $Q \cap \Gamma = \emptyset$

1.2 **Definition** : The set $\mathcal{K}_\tau$ of the *configurations* of a $\tau = (Q, \Sigma, \Gamma, \delta, q_0, \sqcup)$ is given by

$$\mathcal{K}_\tau = \Gamma^* \cdot Q \cdot \Gamma^+.$$

A configuration uqv means that the TM is in state q, the contents of the tape is $\sqcup^\infty uv \sqcup^\infty$ and the first (leftmost) symbol of the string v is read.

1.3 **Definition (Operating principles of a TM $\tau = (Q, \Sigma, \Gamma, \delta, q_0, \sqcup)$)**:

(1) The *initial configuration* $\alpha(v)$ for a string $v \in \Sigma^*$ is

$$\alpha(v) = \begin{cases} q_0 v & \text{if } v \neq \varepsilon \\ q_0 \sqcup & \text{otherwise} \end{cases}$$

(2) The *transition relation* $\vdash_\tau \subseteq \mathcal{K}_\tau \times \mathcal{K}_\tau$ is defined as follows:

$$K \vdash_\tau K' \qquad (K' \text{ is a } \textit{successor configuration} \text{ of } K)$$

if $\exists u, v \in \Gamma^* \; \exists a, b \in \Gamma \; \exists q, q' \in Q$:

$$\begin{aligned} &(K = uqav \;\wedge \delta(q,a) = (q', a', S) \wedge K' = uq'a'v \;) \\ \vee\; &(K = uqabv \wedge \delta(q,a) = (q', a', R) \wedge K' = ua'q'bv) \\ \vee\; &(K = uqa \;\;\wedge \delta(q,a) = (q', a', R) \wedge K' = ua'q'\sqcup \;) \\ \vee\; &(K = ubqav \wedge \delta(q,a) = (q', a', L) \wedge K' = uq'ba'v) \\ \vee\; &(K = qav \;\;\wedge \delta(q,a) = (q', a', L) \wedge K' = q'\sqcup a'v \;) \end{aligned}$$

By $\vdash_\tau{}^*$ we denote the *reflexive transitive closure* of $\vdash_\tau$, i.e. it holds

$$\begin{aligned} K \vdash_\tau{}^* K' \text{ if } \; &\exists K_0, \dots, K_n, n \geq 0 : \\ &K = K_0 \vdash_\tau \dots \vdash_\tau K_n = K' \end{aligned}$$

Further on by $\vdash_\tau{}^+$ we will denote the *transitive closure* of $\vdash_\tau$, i.e. it holds that

$$\begin{aligned} K \vdash_\tau{}^+ K' \text{ if } \; &\exists K_0, \dots, K_n, n \geq 1 : \\ &K = K_0 \vdash_\tau \dots \vdash_\tau K_n = K' \end{aligned}$$

(3) A *final configuration* is a configuration $K \in \mathcal{K}_\tau$, which has no successor configurations.

(4) The *result* (or the visible output) of a configuration uqv is

$$\omega(uqv) = \overline{u}\overline{v},$$

where $\overline{u}$ is the shortest string with $u = \sqcup \dots \sqcup \overline{u}$ and $\overline{v}$ is the shortest string with $v = \overline{v}\sqcup \dots \sqcup$. Thus we eliminate q, as well the blanks in the beginning and in the end of the string; however, the blanks located between characters are not removed.

Remark : Because δ is a (partial) function, $\vdash_\tau$ is right-unique, i.e. from $K \vdash_\tau K_1 \wedge K \vdash_\tau K_2$ it follows that $K_1 = K_2$.

1.4 **Definition** : The function computed by TM $\tau = (Q, \Sigma, \Gamma, \delta, q_0, \sqcup)$ is

$$h_\tau : \quad \Sigma^* \stackrel{part}{\longrightarrow} \Gamma^*$$

with

$$h_\tau(v) = \begin{cases} w & \text{if} \quad \exists \text{ final configuration } K \in \mathcal{K}_\tau : \\ & \quad \alpha(v) \vdash_\tau{}^* K \wedge w = \omega(K) \\ \text{undef.} & \text{otherwise} \end{cases}$$

We also write Res_τ for h_τ ("result function" of τ).

Remark : Because $\vdash_\tau$ is right-unique, h_τ is a partial function.

Illustration of the result function:

$$\begin{array}{ccccccccc} \Sigma^* & \ni & v & & \xrightarrow{\qquad h_\tau \qquad} & & w & \in & \Gamma^* \\ & & \alpha \downarrow & & & & \uparrow \omega & & \\ \mathcal{K}_\tau & \ni & \alpha(v) & \vdash_\tau & \cdots & \vdash_\tau & K & \in & \mathcal{K}_\tau \end{array}$$

1.5 **Definition** : A set $M \in \Sigma^*$ is called a *domain* of τ if the following holds:

$$M = \{v \in \Sigma^* \mid h_\tau(v) \text{ is defined}\}$$

A set $N \in \Gamma^*$ is called *range of values* of τ if the following holds:

$$N = \{w \in \Gamma^* \mid \exists v \in \Sigma^* : h_\tau(v) = w\}.$$

1.6 **Definition (Turing-computability)**:
Let A, B be alphabets.

(i) A partially defined *function* $h : A^* \stackrel{part}{\longrightarrow} B^*$ is called *Turing-computable* if there exists a TM $\tau = (Q, \Sigma, \Gamma, \delta, q_0, \sqcup)$ with $A = \Sigma$, $B \subseteq \Gamma$ and $h = h_\tau$, i.e. $h(v) = h_\tau(v)$ for all $v \in A^*$.

(ii) $\mathcal{T}_{A,B} =_{def} \{h : A^* \stackrel{part}{\longrightarrow} B^* \mid \ h \text{ is Turing-computable}\}$

(iii) Let $\mathcal{T}$ be the class of all Turing-computable functions (for arbitrary alphabets A, B).

1.7 **Definition (Turing-decidability)**:
Let A be an alphabet.

(i) A *set* $L \subseteq A^*$ is *Turing-decidable* if the *characteristic function of* L

$$\chi_L : \quad A^* \longrightarrow \{0,1\}$$

is Turing-computable. Here χ_L is the following total function:

$$\chi_L(v) = \begin{cases} 1 & \text{if } v \in L \\ 0 & \text{otherwise} \end{cases}$$

(ii) A *property* $E : A^* \longrightarrow \{$ *true, false* $\}$ is Turing-decidable if the set $\{v \in A^* \mid E(v) = \textit{true}\}$ of the strings with the property E is Turing-decidable.

Remarks on computability

(1) *Computability of functions with several arguments:*
k-tuples as input can be described by a function $h : (A^*)^k \xrightarrow{part} B^*$. For the computability of such functions we simply modify the definition of the initial configuration using *separator* #:

$$\alpha(v_1, \ldots, v_k) = q_0 v_1 \# v_2 \# \ldots \# v_k$$

If $v_1 = \varepsilon$, then q_0 observes the first separator. Except for this modification we can use the previous definition of computability.

(2) *Computability of number theoretic functions:*
$f : \mathbb{N} \xrightarrow{part} \mathbb{N}$
We use the *bar representation* of natural numbers:

$$n \mathrel{\widehat{=}} |^n = \underbrace{|\ldots|}_{n \text{ times}}$$

Then f is Turing-computable if the function

$$h_f : \{\,|\,\}^* \xrightarrow{part} \{\,|\,\}^*$$

with

$$h_f : (\,|^n\,) = |^{f(n)}$$

is Turing-computable.

Constructing Turing machines: the flowchart notation

First we define useful elementary TMs. Let $\Gamma = \{a_0, \ldots, a_n\}$ be the tape alphabet with $a_0 = \sqcup$.

- Small right-machine r:
  makes one step to the right and halts. Turing table:

r					
	q_0	a_0	q_e	a_0	R
	...	...	...	...	...
	q_0	a_n	q_e	a_n	R

- Small left-machine l:
  makes one step to the left and the halts. Turing table:

l					
	q_0	a_0	q_e	a_0	L
	...	...	...	...	...
	q_0	a_n	q_e	a_n	L

- Printer-machine a for $a \in \Gamma$:
  prints the symbol a and then halts. Turing table:

a					
	q_0	a_0	q_e	a	S
	...	...	...	...	...
	q_0	a_n	q_e	a	S

Furthermore we assume that all constructed TMs have exactly one *final state*, i.e. there exists one state q_e so that for all final configurations uqv it holds that:

$$q = q_e.$$

It is obvious that TMs r, l, a fulfill this condition. Such TMs can be combined as follows:

$\tau_1 \xrightarrow{a} \tau_2$ means that first τ_1 works. If τ_1 terminates on a cell with symbol a, τ_2 starts.

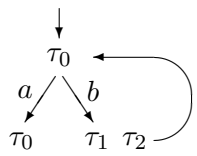
$\tau_1 \longrightarrow \tau_2$ means that first τ_1 works. If τ_1 terminates, τ_2 starts.

$\tau_1\tau_2$ is an abbreviation for $\tau_1 \longrightarrow \tau_2$.

$\tau_1 \xrightarrow{\neq a} \tau_2$ means that first τ_1 works. If τ_1 terminates on a cell with a symbol different from a, τ_2 starts.

On the basis of the given TMs we can construct flowcharts. The nodes of this flowchart are labelled with the names of TMs. The edges are labelled with arrows of the form $\xrightarrow{a}$, $\longrightarrow$ or $\xrightarrow{\neq a}$. Loops are allowed. One TM τ in the flowchart is labelled with an arrow $\longrightarrow \tau$ as a start-TM.

Illustration:



A flowchart describes a "large" TM. We can get its Turing table as follows:

Step 1: For every occurrence of a TM τ_i in the flowchart construct the respective Turing table.

Step 2: Make the states in different tables disjoint.

Step 3: Generate the table of TM by writing all tables (in any order) one below each other.

Step 4: *Combining*: For each $\tau_1 \xrightarrow{a} \tau_2$ in the flowchart add to the table of TM line

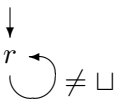
$$q_{e\tau_1} a q_{0\tau_2} a S.$$

Let $q_{e\tau_1}$ be the final state (renamed according to Step 2) of τ_1 and $q_{0\tau_2}$ the start state (renamed according to Step 2) of τ_2. Similarly for $\tau_1 \to \tau_2$ and $\tau_1 \overset{\neq a}{\to} \tau_2$.

Example :

- Large right-machine $\mathcal{R}$:
  first makes a step to the right. Then $\mathcal{R}$ moves to the right until a blank $a_0 = \sqcup$ is observed.

$\mathcal{R}$



Construction of the Turing table of $\mathcal{R}$:

$\mathcal{R}$

q_0	a_0	q_e	a_0	R
...	...	...	...	...
q_0	a_n	q_e	a_n	R

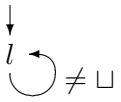
r

q_e	a_1	q_0	a_1	S
...	...	...	...	...
q_e	a_n	q_0	a_n	S

$\neq a_0$

- Large left-machine $\mathcal{L}$: similarly to the large right-machine.

$\mathcal{L}$



Application: Consider a TM for the computation of the "minus" function:

$$f : \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$$

with

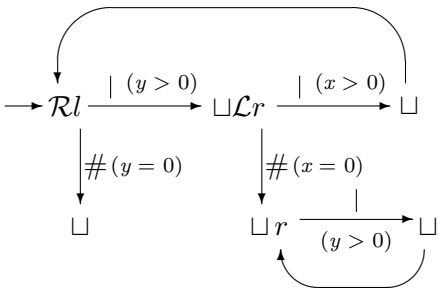
$$f(x,y) = x \dot{-} y = \begin{cases} x - y & \text{for } x \geq y \\ 0 & \text{otherwise} \end{cases}$$

The initial configuration of the TM is:

$$\sqcup q_0 \underbrace{\| \ldots \|}_{x\text{-times}} \# \underbrace{\| \ldots \|}_{y\text{-times}} \sqcup$$

Idea: erase x-dashes as long as y-dashes are there. Then erase the remaining y-dashes and #.

Construction of the TM using a flowchart:



As an exercise we recommend to construct the whole Turing table.

Variations of Turing machines

There exist many variations of the TM definition. These variations are often more convenient if we want to prove that a certain function is computable. We can show that these variations do not increase the power of a TM, which was defined in Definition 1.1. First we consider TM with *several tapes*.

1.8 **Definition (k-tape Turing machine)**: A *k-tape TM* $\tau = (Q, \Sigma, \Gamma, \delta, k, q_0, \sqcup)$ is a 7-tuple with the following properties:

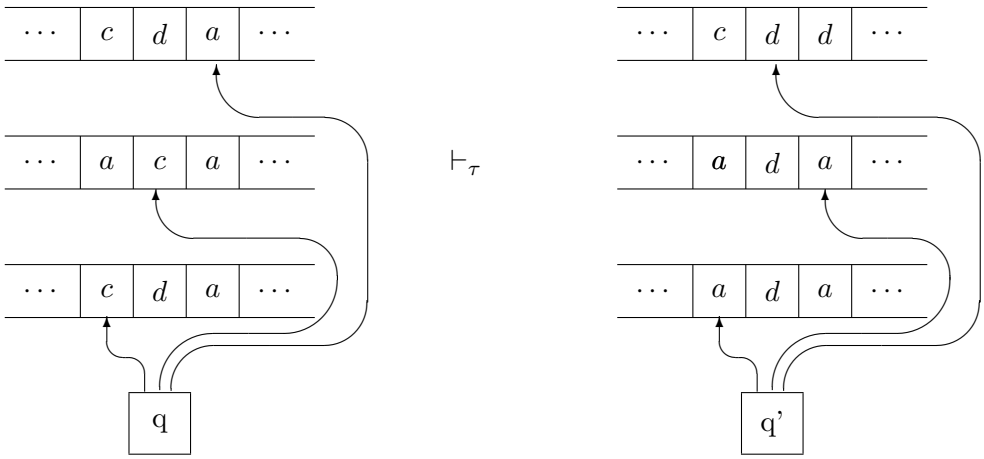
(i)-(v) $Q, \Sigma, \Gamma, q_0, \sqcup$ as in Definition 1.1.

(vi) $k \in \mathbb{N}$ is the number of tapes

(vii) The transition function δ is now given as follows:

$$\delta : Q \times \Gamma^k \xrightarrow{part} Q \times \Gamma^k \times \{R, L, S\}^k$$

Illustration of δ for $k = 3$:



This transition occurs for

$$\delta(q, (a, c, c)) = (q', (d, d, a), (L, R, S))$$

Configuration of a k-tape TM τ:

$$K = (u_1 q v_1, \ldots, u_k q v_k) \in \mathcal{K}_\tau$$

with $u_1, \ldots, u_k \in \Gamma^*$, $q \in Q$, $v_1, \ldots, v_k \in \Gamma^+$.

Operating principles of a k-tape TM τ:

(1) Initial configuration for a string $v \in \Sigma^*$ is

$$\alpha_k(v) = (\alpha(v), \underbrace{q_0\sqcup, \ldots, q_0\sqcup}_{(k-1)\text{-times}}),$$

i.e. v is written on the first tape.

(2) Transition relation $\vdash_\tau \subseteq \mathcal{K}_\tau \times \mathcal{K}_\tau$: similarly

(3) Final configuration: no successor configuration (as before)

(4) The result ω_k of a k-tape configuration is

$$\omega_k(u_1 q v_1, \ldots, u_k q v_k) = \omega(u_1 q v_1),$$

i.e. we take the result of the first tape; we consider the remaining tapes only as auxiliary tapes for computation.

The *computed function* of a k-tape TM is $h_\tau : \Sigma^* \xrightarrow{part} \Gamma^*$ with

$$h_\tau(v) = \begin{cases} w & \text{if } \exists \text{ final config. } K \in \mathcal{K}_\tau : \\ & \quad \alpha_k(v) \vdash_\tau{}^* K \wedge w = \omega_k(K) \\ \text{undef.} & \text{otherwise} \end{cases}$$

Remark: Several tapes can be used for computing functions with several arguments; in this case we distribute the input strings on the tapes appropriately.

Goal: We want to prove that every function computed by a k-tape TM can be computed by a normal 1-tape TM.

Idea of the proof: Every k-tape TM can be "simulated" using a constructed 1-tape TM. Therefore, first we want to specify the notion of simulation.

Notion of simulation

The notion of simulation is essential for proving that different machines have the same power.

1.9 **Definition** : We consider (1- or k-tape) TM τ and τ' over the same input alphabet Σ, with the sets of configurations $\mathcal{K}_\tau$ and $\mathcal{K}_{\tau'}$, with transition relations $\vdash_\tau, \vdash_{\tau'}$, initial configurations α, α' and result functions ω, ω'. Informally, τ is the "higher" TM and τ' the "lower" TM.

A *simulation of* τ *by* τ' is a totally defined function

$$sim: \mathcal{K}_\tau \longrightarrow \mathcal{K}_{\tau'}$$

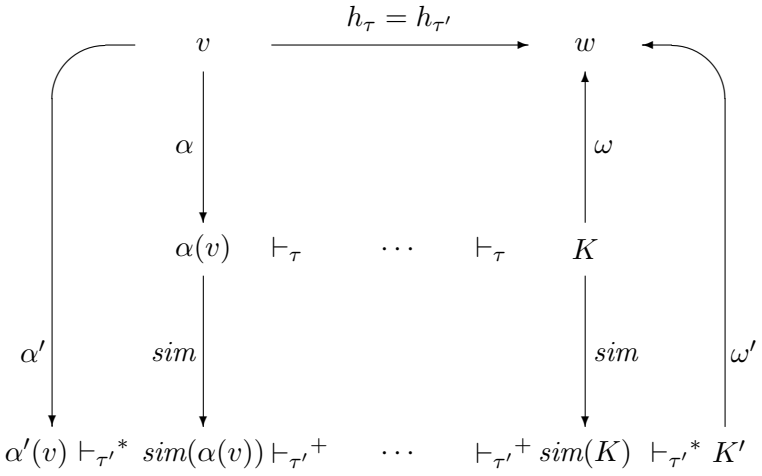
with the following properties:

(1) $\forall v \in \Sigma^*: \quad \alpha'(v) \vdash_{\tau'}{}^* sim(\alpha(v))$

(2) $\forall K_1, K_2 \in \mathcal{K}_\tau : K_1 \vdash_\tau K_2$ implies $sim(K_1) \vdash_{\tau'}{}^+ sim(K_2)$,
i.e. every step of τ can be "simulated" by several steps of τ'.

(3) $\forall$ final config. $K \in \mathcal{K}_\tau \quad \exists$ final config. $K' \in \mathcal{K}_{\tau'}$:
$sim(K) \vdash_{\tau'}{}^* K'$ and $\omega(K) = \omega'(K')$.

Illustration:



1.10 **Theorem (Simulation theorem)**: Let τ and τ' (1- or k-tape) be TMs over the same input alphabet Σ. Let there be a simulation *sim* of τ by τ'. Then the functions h_τ and $h_{\tau'}$ computed by τ and τ', respectively, coincide, i.e.

$$\forall v \in \Sigma^*: \quad h_\tau(v) = h_{\tau'}(v)$$

(The equality here means that either both sides are undefined or they have the same value).

Proof : Let $v \in \Sigma^*$.
Case 1: $h_\tau(v)$ is undefined.
Then there is an infinite sequence of configurations

$$\alpha(v) = K_1 \vdash_\tau K_2 \vdash_\tau K_3 \vdash_\tau \ldots$$

of τ. Therefore, due to properties (1) and (2) of *sim* , we have an infinite sequence of configurations

$$\alpha'(v) \vdash_{\tau'}{}^* sim(K_1) \vdash_{\tau'}{}^+ sim(K_2) \vdash_{\tau'}{}^+ sim(K_3) \vdash_{\tau'}{}^+ \ldots$$

of τ'. Thus $h_{\tau'}(v)$ is also undefined.
Case 2: $h_\tau(v) = w$ is defined.
Then there is a finite sequence of configurations

$$\alpha(v) = K_1 \vdash_\tau \ldots \vdash_\tau K_n, \quad n \geq 1,$$

of τ, where K_n is a final configuration and $w = \omega(K_n)$ holds. Due to the properties (1) - (3) of *sim* , we have a finite sequence of configurations

$$\alpha'(v) \vdash_{\tau'}{}^* sim(K_1) \vdash_{\tau'}{}^+ \ldots \vdash_{\tau'}{}^+ sim(K_n) \vdash_{\tau'}{}^* K'_n$$

of τ', where K'_n is a final configuration with $\omega(K_n) = \omega'(K'_n)$. Thus it holds that

$$h_\tau(v) = w = \omega(K_n) = \omega'(K'_n) = h_{\tau'}(v).$$

□

Now we can show:
1.11 **Theorem** : For every k-tape TM τ there is a 1-tape TM τ' with $h_\tau = h_{\tau'}$.

Proof : Let $\tau = (Q, \Sigma, \Gamma, \delta, k, q_0, \sqcup)$. We construct $\tau' = (Q', \Sigma', \Gamma', \delta', q'_0, \sqcup)$ so that there is a simulation *sim* of τ by τ'. Definition of *sim*:

K	$= ($	$u_1 q a_1 v_1$,	is	$sim(K) =$
		$\ldots$,	simulated	$u_1 q \widetilde{a_1} v_1 \# \ldots \# u_k \widetilde{a_k} v_k$
		$u_k q a_k v_k$ $)$	by	
	$\in$	$\mathcal{K}_\tau$		$\in \mathcal{K}_{\tau'}$

That is we mark the symbols $a_1, \ldots, a_k$ read by k heads using new symbols $\widetilde{a_1}, \ldots, \widetilde{a_k}$, we write down the contents of k tapes one after each other on the tape of τ', separated by #, and we set the state q in such a way that $\widetilde{a_1}$ is observed.

Thus we choose: $\Gamma' = \Gamma \cup \{\tilde{a} \,|\, a \in \Gamma\} \cup \{\#\}$. We sketch the choice of Q' and δ' according to the following idea of simulation. A step of τ of the form

$$\begin{array}{rcl} K_1 = (u_1 q_1 a_1 v_1, & \vdash_\tau & K_2 = (u_1 q_2 b_1 v_1, \\ \ldots \quad , & & \ldots \quad , \\ u_k q_1 a_k v_k) & & u_k q_2 b_k v_k) \end{array}$$

generated by

$$\delta(q_1, (a_1, \ldots, a_k)) = (q_2, (b_1, \ldots, b_k), (S, \ldots, S))$$

is simulated by τ' in 2 phases of steps.

- *Reading phase:*

$$sim(K_1) = u_1 q_1 \widetilde{a_1} v_1 \# \ldots \# u_k \widetilde{a_k} v_k$$
$$\top_{\tau'}$$
$$u_1 [read, q_1] \widetilde{a_1} v_1 \# \ldots \# u_k \widetilde{a_k} v_k$$
$$\top^*_{\tau'}$$
$$u_1 \widetilde{a_1} v_1 \# \ldots \# u_k [read, q_1, a_1, \ldots, a_k] \widetilde{a_k} v_k$$

- *Changing phase:*
  Now we will change the configuration according to $\delta(q, (a_1, \ldots, a_k))$.

$$u_1\widetilde{a_1}v_1\# \ldots \#u_k[read, q_1, a_1, \ldots, a_k]\widetilde{a_k}v_k$$
$$\top_{\tau'}$$
$$u_1\widetilde{a_1}v_1\# \ldots \#u_k[change, q_2, b_1, \ldots, b_k, S, \ldots, S]\widetilde{a_k}v_k$$
$$\top^*_{\tau'}$$
$$u_1[change, q_2]\widetilde{b_1}v_1\# \ldots \#u_k\widetilde{b_k}v_k$$
$$\top_{\tau'}$$
$$sim(K_2) = u_1q_2\widetilde{b_1}v_1\# \ldots \#u_k\widetilde{b_k}v_k$$

Thus a step of τ is simulated by at most as many steps of τ' as the double length of all strings written on the k tapes.

Similarly we deal with other δ-transitions. However, if in R- or L-steps we must "stick" an empty field $\sqcup$ to one of the k tapes, then we must *shift* the contents of the neighboring #-parts on the tape of τ'.

Now we consider the initial and final configurations.

- *Initial configuration:* Let $v \in \Sigma^*$ and $v \neq \varepsilon$.
  Then for τ
  $\alpha(v) = (q_0v\,,$ and $sim(\alpha(v)) = q_0v\#\sqcup\# \ldots \#\sqcup$
  $q_0\sqcup,$
  $\ldots$
  $q_0\sqcup)$
  For τ': $\alpha'(v) = q'_0v$. Of course we can program τ' in such a way that
  $$\alpha'(v) \vdash_{\tau'}{}^* sim(\alpha(v))$$
  holds.

- *Final configuration:* If $K \in \mathcal{K}_\tau$ is a final configuration of the form
  $$K = (u_1q_ea_1v_1\,,$$
  $$\ldots \quad ,$$
  $$u_kq_ea_kv_k),$$
  then we can understand that
  $$sim(K) = u_1q_e\widetilde{a_1}v_1\# \ldots \#u_k\widetilde{a_k}v_k$$
  holds only in the reading phase. Then we must bring $sim(K)$ to the result form of a 1-tape TM by erasing all symbols starting from the first #. Thus we can program τ' in such a way that
  $$sim(K) \vdash_{\tau'}{}^* u_1q'_ea_1v_1$$
  holds and $K' = u_1q'_ea_1v_1$ is the final configuration of τ' with $\omega(K) = \omega'(K')$.

Thus we have:

$$\begin{aligned} Q' = Q \cup \{ & q'_0, q'_e & \} \\ \cup \{ & [\mathit{read}, q, a_1, \ldots, a_j] \,|\, q \in Q, a_1, \ldots, a_j \in \Gamma, 0 \leq j \leq kff & \} \\ \cup \{ & [\mathit{change}, q, b_1, \ldots, b_j, P_1, \ldots, P_j] \,|\, q \in Q, 0 \leq j \leq k, b_1, \ldots, b_j \in \Gamma, \\ & P_1, \ldots, P_j \in \{R, L, S\} & \} \\ \cup \{ & \ldots \text{ further states } \ldots & \} \end{aligned}$$

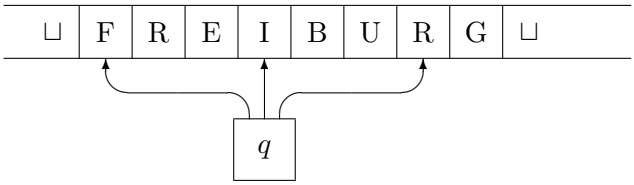
Altogether we have now shown that *sim* is a simulation of τ by τ'. Thus according to the simulation theorem it follows that $h_\tau = h_{\tau'}$.

□

Other variations of Turing machines

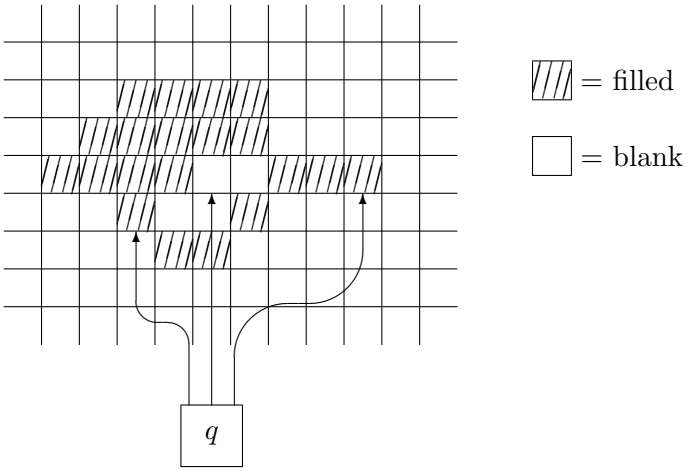
Below we will consider other variations of Turing machines:

- Turing machines with k *heads* on one tape:

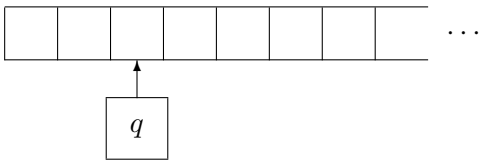


  Definition of the transition function δ: as in Definition 1.8 of k-tape TM, but the definition of a configuration is different, because k cells on the tape should be marked.

- Turing machines with a *two-dimensional* computational space divided into cells ("computing on graph paper"), possibly with several heads:



- Turing machine with *one-sided* infinite tape:



  A normal TM (with two-sided infinite tape) can be simulated as follows by a TM with one-sided infinite tape:

  2-sided configuration $K = a_m \dots a_1 q b_1 \dots b_n$ with $m \geq 0, n \geq 1$
  is (for $m \leq n$) simulated by the

  1-sided configuration $sim(K) =$

q	b_1	$\cdots$	b_m	b_{m+1}	$\cdots$	b_n
	a_1		a_m	$\sqcup$		$\sqcup$

  i.e. pairs of symbols of the 2-sided tape are the symbols of the 1-sided tape.

- Turing machines with *final states*:

$$\tau = (Q, \Sigma, \Gamma, \delta, q_0, \sqcup, F),$$

where $F \subseteq Q$ is the set of final states and all remaining components are defined as before.

We will use TM with final states in order to "accept" languages $L \subseteq \Sigma^*$.

1.12 **Definition** :

(1) A configuration $K = uqv$ of τ is called *accepting* if $q \in F$.

(2) Let $v \in \Sigma^*$. The TM τ *accepts* v if $\exists$ accepting final configuration $K : \alpha(v) \vdash_\tau{}^* K$.

(3) The *language accepted* by τ is

$$L(\tau) = \{v \in \Sigma^* \mid \tau \text{ accepts } v\}.$$

A language $L \subseteq \Sigma^*$ is called *Turing-acceptable* if there is a TM τ with final states for which $L = L(\tau)$ holds.

(4) If we speak about sets, then by $\mathcal{T}$ we denote the languages accepted by Turing machines.

1.13 **Theorem** : Let $L \subseteq \Sigma^*$ and $\overline{L} = \Sigma^* - L$.
L and $\overline{L}$ are Turing-acceptable $\Leftrightarrow$ L is Turing-decidable.

Proof : "$\Leftarrow$": Let L be decidable by τ. By adding final state transitions we get an accepting TM for L and $\overline{L}$.
"$\Rightarrow$": L is accepted by τ_1 and $\overline{L}$ by τ_2. Now let us construct τ with 2 tapes in such a way that *a step of* τ_1 *on tape 1 and simultaneously a step of* τ_2 *on tape 2* is made. If τ_1 accepts the string, then τ produces the value 1. Otherwise τ_2 accepts the string, and τ produces the value 0. Thus τ decides the language L. □

Remark : If only L is Turing-acceptable, then it does not imply that L is also Turing-decidable. The accepting TM might *not terminate* for strings from $\overline{L}$.

- *Non-deterministic Turing machines:*

We consider 1-tape TM with final states. The transition function δ is extended as follows:

$$\delta : \quad Q \times \Gamma \longrightarrow \mathcal{P}(Q \times \Gamma \times \{R, L, S\}).$$

A tuple

$$(q', b, R) \in \delta(q, a)$$

means that the TM, in case if the symbol a is read in state q, can do the following:

go to state q', write b, move to the right.

If there is another tuple

$$(q'', c, S) \in \delta(q, a),$$

then the TM can do the following instead:

go to state q'', write c, stay.

The choice between these two possible steps of the TM is *non-deterministic*, i.e. the TM can arbitrarily choose one of the two possible steps. The transition relation $\vdash_\tau$ of a non-deterministic TM τ is not right-unique any more. Apart from that, the language $L(\tau)$ accepted by τ is defined as for a deterministic TM.

1.14 **Theorem** : Every language accepted by a non-deterministic TM is also acceptable by a deterministic TM.

Proof : Consider a non-deterministic 1-tape TM τ. Then there exists a maximal number r of non-deterministic choices which τ has according to δ in state q and symbol a, i.e.

$$\begin{aligned} &\forall q \in Q \quad \forall a \in \Gamma : \quad |\delta(q, a)| \leq r, \\ &\exists q \in Q \quad \exists a \in \Gamma : \quad |\delta(q, a)| = r \end{aligned}$$

By r we denote the *degree of non-determinism* of τ. For each pair (q, a) we number the possible choices from 1 to (at most) r consequently according to $\delta(q, a)$. Then we can represent every finite sequence of non-deterministic choices as strings over the alphabet $\{1, \ldots, r\}$.

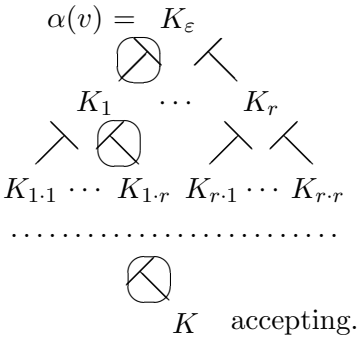
Example: For $r = 5$, for example, $1 \cdot 3 \cdot 5$ means:

- in the 1st step make the 1st choice,
- in the 2nd step make the 3rd choice,
- in the 3rd step make the 5th choice.

Now let $v \in \Sigma^*$. Then τ accepts the string v if there is an accepting computation of τ:

$$\alpha(v) \vdash_\tau{}^* K \quad \text{accepting.}$$

The set of all computations of τ starting in $\alpha(v)$ can be represented as a tree with nodes labelled with configurations:



By ⊢ we mark the choices which have lead to the accepting configuration K.

Now we "construct" a deterministic TM τ' so that τ' generates and traverses every computation tree of τ using *breadth-first search*:

0. $\rightarrow$
1. $\longrightarrow$
2. $\longrightarrow$
3. $\longrightarrow$

$\ldots$

For this purpose τ' needs 3 tapes.

- Tape 1 stores the input string v.
- Tape 2 is used for systematic generation of all strings over $\{1, \ldots, r\}$. In order to model the breadth-first search the strings are generated as follows:
  (i) according to the length, the shorter first,
  (ii) when the length is equal, then we use lexicographic order. Thus we get the following sequence:
  $\varepsilon, 1, 2, \ldots, r, 1 \cdot 1, 1 \cdot 2, \ldots, 1 \cdot r, \ldots, r \cdot 1, \ldots, r \cdot r, \ldots$
- Tape 3 is used for simulation of τ by τ'. First the input string v is copied on the tape 3, then a sequence of transitions of τ is simulated according to the string encoding choices from $\{1, \ldots, r\}$, which is written on tape 2.

If τ has an accepting computation v, then its string of choices is generated by τ' on tape 2, and then τ' accepts v. If there is no accepting computation v of τ, then τ' will run infinitely long and generate all strings over $\{1, \ldots, r\}$ on tape 2.
Thus $L(\tau) = L(\tau')$ holds. □

Remark : The above simulation of τ by τ' has *exponential complexity* in the number of steps: in order to find an accepting computation of τ with n steps, τ' must generate and traverse a computation tree with the breadth r and depth n, i.e. with r^n nodes.

The variations of Turing machines considered here

- k-tape TM
- several heads
- 2-dimensional TM
- Final states: for acceptance
- non-deterministic

§2 Grammars

In the previous chapter we got acquainted with context-free grammars. They are a special case of CHOMSKY-Grammars introduced in 1959 by American linguist NOAM CHOMSKY. There exist several types of such grammars (Types 0–3). Here we consider the most general Type 0.

2.1 **Definition (Grammar)**:
A (CHOMSKY-0- or shortly *CH0-*)*grammar* is a 4-tuple $G = (N, T, P, S)$ with

(i) N is an alphabet of *non-terminal symbols*,

(ii) T is an alphabet of *terminal symbols* with $N \cap T = \emptyset$,

(iii) $S \in N$ is the *start symbol*,

(iv) $P \subseteq (N \cup T)^* \times (N \cup T)^*$ is a finite set of *productions* or *rules*, where for $(u, v) \in P$ u must contain at least *one non-terminal symbol*.

Productions $(u, v) \in P$ are written down using arrow notation $u \to v$, as in context-free grammars. Every grammar G has a *derivation relation* $\vdash_G$ on $(N \cup T)^*$:

$$x \vdash_G y \quad \text{if and only if} \quad \exists u \to v \in P \; \exists w_1, w_2 \in (N \cup T)^* : \\ x = w_1 \boxed{u} w_2 \text{ and } y = w_1 \boxed{v} w_2.$$

By $\vdash_G^*$ we again denote the reflexive and transitive closure of $\vdash_G$. We read $x \vdash^* y$ as "y can be derived from x". It holds

$$x \vdash^* y \text{ if } \exists z_0, \ldots, z_n \in \Sigma^*, n \geq 0 : \; x = z_0 \vdash_G \ldots \vdash_G z_n = y$$

The sequence $x = z_0 \vdash_G \ldots \vdash_G z_n = y$ is also called a *derivation* of y from x (or from x to y) of the *length* n. Particularly, it holds: $x \vdash_G^* x$ with a derivation of the length 0.

2.2 **Definition** :

(i) The *language generated from* G is

$$L(G) = \{w \in T^* \mid S \vdash_G^* w\},$$

i.e. we are interested only in strings over the terminal symbols; the non-terminal symbols are used as auxiliary symbols within derivations.

(ii) $L \subseteq T^*$ is called CHOMSKY-0- (or shortly *CH0-*) *language* if there is a CHOMSKY-0-grammar G with $L = L(G)$. The class of all CH0-languages is also shortly denoted by *CH0*.

2.3 **Theorem ($\mathcal{T} \subseteq$ *CH0*)**:
Every Turing-acceptable language $L \subseteq T^*$ is a CHOMSKY-0-language.

Proof : Let L be accepted by a TM $\tau = (Q, \Sigma, \Gamma, \delta, q_0, \sqcup, F)$ with $F = \{q_e\}$ so that for all configurations $uqv \in K_\tau$ it holds:

$$uqv \text{ is a final configuration} \iff u = \varepsilon \text{ and } q = q_e.$$

We can easily see that for every TM we can provide a TM with this property, which accepts the same language. Now we construct a CHOMSKY-0-grammar $G = (N, T, P, S)$ with $L(G) = L$ in 4 steps.

Step 1: (Generate double start configurations)
Let us define a partial grammar $G_1 = (N_1, T, P_1, S)$ with $S, \not\in, \$, q_0, \sqcup \in N_1$ so that for all $w \in T^*$

$$S \vdash_{G_1}^* w\not\in\alpha(w)\$$$

holds and even for all $v \in (N \cup T)^*$:

$$\begin{array}{ll} & S \vdash_{G_1}^* v \text{ and } v \text{ contains } q_0 \\ \Rightarrow & \exists w \in T^* : v = w\not\in\alpha(w)\$. \end{array}$$

The start configuration $\alpha(w)$ is defined as usual:

$$\alpha(w) = \begin{cases} q_0 w & \text{if } w \neq \varepsilon \\ q_0\sqcup & \text{otherwise.} \end{cases}$$

Choose $N_1 = \{S, \not\in, \$, q_0, \sqcup, A, B\} \cup \{C_a \mid a \in T\}$ and P_1 as follows:

$$\begin{array}{lll} P_1 = \{ \quad S & \to \not\in q_0\sqcup\$, & \\ S & \to aA\not\in C_a\$ & \text{for all } a \in T, \\ C_a b & \to bC_a & \text{for all } a, b \in T, \\ C_a\$ & \to Ba\$ & \text{for all } a \in T, \\ bB & \to Bb & \text{for all } b \in T, \\ A\not\in B & \to \not\in q_0, & \\ A\not\in B & \to aA\not\in C_a & \text{for all } a \in T \qquad \} \end{array}$$

Operating principle of P_1:

$$S \vdash_{G_1} \not\in q_0\sqcup\$ = \varepsilon\not\in\alpha(\varepsilon)\$$$

or

$$\begin{array}{l} S \vdash_{G_1}^* aA\not\in C_a\$ \\ S \vdash_{G_1}^* w\ \boxed{A\not\in B}\ w\$ \\ S \vdash_{G_1}^* w\ aA\not\in C_a w\$ \\ S \vdash_{G_1}^* w\ aA\not\in wC_a\$ \\ S \vdash_{G_1}^* w\ aA\not\in wBa\$ \\ S \vdash_{G_1}^* w\ a\boxed{A\not\in B}wa\$ \\ S \vdash_{G_1}^* w\ a\not\in q_0 wa\$ \end{array}$$

Step 2: (Simulate transition relation $\vdash_\tau$)
Let us define $G_2 = (N_2, T, P_2, S)$ with $N_2 = \{S, \not\in, \$\} \cup Q \cup \Gamma$ so that for all $w \in T^*, v \in \Gamma^*$ it holds that:

$$\alpha(w) \vdash_\tau{}^* q_e v \quad \iff \quad w\not\in\alpha(w)\$ \vdash_{G_2}^* w\not\in q_e v\$.$$

We can not simply choose $P_2 = \vdash_\tau$, because $\vdash_\tau$ is infinite. For example, $\delta(q,a) = (q',a',S)$ implies that for all $u,v \in \Gamma^* : uqav \vdash_\tau uq'a'v$. We can focus on the finite δ and choose a rule of the form

$$qa \to q'a'.$$

More precisely we define:

$$\begin{aligned}
P_2 = \quad & \{qa \to q'a' && \mid q,q' \in Q \text{ and } a,a' \in \Gamma \text{ and } \delta(q,a) = (q',a',S) && \} \\
\cup\ & \{qab \to a'q'b && \mid q,q' \in Q \text{ and } a,a',b \in \Gamma \text{ and } \delta(q,a) = (q',a',R) && \} \\
\cup\ & \{ \underbrace{qa\$}_{\text{TM is at the right end of the tape}} \to a'q'\sqcup\$ && \mid q,q' \in Q \text{ and } a,a' \in \Gamma \text{ and } \delta(q,a) = (q',a',R) && \} \\
\cup\ & \{bqa \to q'ba' && \mid q,q' \in Q \text{ and } a,a',b \in \Gamma \text{ and } \delta(q,a) = (q',a',L) && \} \\
\cup\ & \{ \underbrace{\text{¢}qa}_{\text{TM is at the left end of the tape}} \to \text{¢}q'\sqcup a' && \mid q,q' \in Q \text{ and } a,a' \in \Gamma \text{ and } \delta(q,a) = (q',a',L) && \}
\end{aligned}$$

Step 3: (Erase final configuration)
Define $G_3 = (N_3, T, P_3, S)$ with $N_3 = \{S, \text{¢}, \$, q_e, \sqcup, D\}$ so that for all $w \in T^*, v \in \Gamma^*$ it holds that:

$$w\text{¢}q_e v\$ \vdash_{G_3}^* w.$$

Choose P_3 as follows:

$$\begin{aligned}
P_3 = \{ \quad & \text{¢}q_e \to D, \\
& Da \to D \quad \text{for all } a \in \Gamma, \\
& D\$ \to \varepsilon \quad \}
\end{aligned}$$

Operating principle:

$$w\text{¢}q_e v\$ \vdash_{G_3} wDv\$ \vdash_{G_3}^* wD\$ \vdash_{G_3} w.$$

Step 4: (Compute) G from G_1, G_2, G_3)
Now let us define $G = (N, T, P, S)$ as follows:

$$\begin{aligned}
& N = N_1 \cup N_2 \cup N_3, \\
& P = P_1 \dot\cup P_2 \dot\cup P_3 \qquad \text{(disjoint union)}.
\end{aligned}$$

Then for all $w \in T^*, v \in \Gamma^*$ it holds that:

$$\begin{aligned}
\alpha(w) \vdash_\tau{}^* q_e v \quad \iff \quad & S \vdash_G^* w\text{¢}\alpha(w)\$ && \text{(rules } P_1\text{)} \\
& \vdash_G^* w\text{¢}q_e v\$ && \text{(rules } P_2\text{)} \\
& \vdash_G^* w && \text{(rules } P_3\text{)}
\end{aligned}$$

For "$\Leftarrow$" note that the rules from at most one set of rules P_1, P_2 or P_3 can be applied to a given string over $N \cup T$.

Therefore we get $L(G) = L$. □

2.4 **Corollary** : Let the function $h : T^* \xrightarrow{part} T^*$ be computed by a Turing machine. Then the *graph of h*, i.e. the set

$$L = \{w\#h(w) \,|\, w \in T^* \text{ and } h(w) \text{ is defined } \},$$

is a CHOMSKY-0-language.

Proof : If h is computed by a TM τ, then there is a 2-tape TM τ' which accepts L. The TM τ' operates as follows:

(1) τ' leaves a given input string of the form $w\#v$ unchanged on the 1st tape.

(2) τ' copies the part w on the initially empty 2nd tape and then simulates τ on this tape.

(3) If τ terminates, then the result $h(w)$ of the final configuration is compared to the part v of the 1st tape. If $h(w) = v$ holds, then τ' accepts the input $w\#v$ on the 1st tape. Otherwise, τ' does not accept the input $w\#v$ on the 1st tape.

Thus according to the above theorem L is a CHOMSKY-0-language. □

2.5 **Theorem** (*CH0* $\subseteq \mathcal{T}$):
Every CHOMSKY-0-language $L \subseteq T^*$ is Turing-acceptable.

Proof : L is generated by a CHOMSKY-0-grammar $G = (N, T, P, S)$: $L(G) = L$.

We construct a *non-deterministic 2-tape TM* τ which accepts L. The TM τ operates as follows:

(1) τ leaves a given input string $w \in T^*$ unchanged on the 1st tape.

(2) On the initially empty 2nd tape, τ iteratively generates strings over $N \cup T$ according to the rules from P, starting with the start string S. In every step τ chooses non-deterministically a substring u from the last generated string and a rule $u \to v$ from P and then replaces u by v. If the input string w occurs in some step, then w is accepted.

Thus it holds: τ accepts L. □

In addition to general CHOMSKY-0-grammars $G = (N, T, P, S)$, where for rules $p \to q \in P$ it holds that $p, q \in (N \cup T)^*$ and p contains at least one non-terminal symbol, there also exist other classes of grammars.

2.6 **Definition** : A CHOMSKY-0-grammar (or shortly *CH0-grammar*) $G = (N, T, P, S)$ is called (for ε see below)

(i) *context-sensitive* (CHOMSKY-1- or shortly *CH1-grammar*) if and only if
$\forall\, p \to q \in P\ \exists\, \alpha \in N, u, v, w \in (N \cup T)^*, w \neq \varepsilon :\ p = u\alpha v \wedge q = uwv$
(i.e. a non-terminal symbol $\alpha \in N$ is replaced by the non-empty string w "in context" $u \ldots v$).

(ii) *context-free* (CHOMSKY-2- or shortly *CH2-grammar*) if and only if
$\forall\, p \to q \in P :\ p \in N \quad (q = \varepsilon$ is allowed).

(iii) *right-linear* (CHOMSKY-3- or shortly *CH3-grammar*) if and only if
$\forall\, p \to q \in P :\ p \in N \wedge q \in T^* \cdot N \cup T^*$

For context-sensitive grammars $G = (N, T, P, S)$ there exists the following special rule, so that $\varepsilon \in L(G)$ is possible: ε-production may only be of the form $S \to \varepsilon$. Then all other derivations begin with a new non-terminal symbol S':
$S \to S'$
$S' \to \ldots$

2.7 **Definition** : A language L is called *context-sensitive, context-free, right-linear*, if there exists a grammar of corresponding type which generates L.

Let the classes of languages be defined for some alphabet T:

$$\begin{aligned} CH0 &= \text{Class of languages, generated by Chomsky-0- grammars},\\ CS &= \text{" " " , " " context-sensitive "},\\ CF &= \text{" " " , " " context-free "},\\ RLIN &= \text{" " " , " " right-linear "} \end{aligned}$$

By definition it already holds that (however, the inclusion $CF \subseteq CS$ can be proved only on the basis of the later theorems):

2.8 **Corollary** : $RLIN \subseteq CF \subseteq CS \subseteq CH0$

Now we want to show that the CHOMSKY-3- (i.e. right-linear) languages are exactly the finitely acceptable (i.e. regular) languages from Chapter II.

2.9 **Lemma** : Every finitely acceptable language is a CHOMSKY-3-language.

Proof : Consider a DFA $\mathcal{A} = (\Sigma, Q, \to, q_0, F)$. Let us construct the following CHOMSKY-3-grammar G:

$$G = (Q, \Sigma, P, q_0),$$

where for $q, q' \in Q$ and $a \in \Sigma$ it holds that

$$\begin{aligned} P &= \{q \to aq' \mid q \xrightarrow{a} q'\}\\ &\cup \{q \to \varepsilon \mid q \in F\}. \end{aligned}$$

It is easy to show that $L(\mathcal{A}) = L(G)$ holds. □

For the reverse direction we use the following lemma.

Remark : Every CHOMSKY-3-language can be generated by a grammar $G = (N, T, P, S)$ with $P \subseteq N \times (T \cdot N \cup \{\varepsilon\})$.

Proof : Exercise. □

2.10 **Lemma** : Every CHOMSKY-3-language is finitely acceptable.

Proof : Consider a CHOMSKY-3-language $L \subseteq T^*$. According to the remark L can be generated by a grammar $G = (N, T, P, S)$ with $P \subseteq N \times (T \cdot N \cup \{\varepsilon\})$: $L = L(G)$. Now we construct the following ε-NFA $\mathcal{B}$:

$$\mathcal{B} = (T, N \cup \{q_e\}, \rightarrow, S, \{q_e\}),$$

where $q_e \notin N$ holds and the transition relation $\rightarrow$ for $\beta, \gamma \in N$ and $a \in T$ is defined as follows:

- $\beta \xrightarrow{a} \gamma$ if and only if $\beta \rightarrow a\gamma \in P$
- $\beta \xrightarrow{\varepsilon} q_e$ if and only if $\beta \rightarrow \varepsilon \in P$

Again it is easy to show that $L(G) = L(\mathcal{B})$ holds. □

From these two lemmas we get:

2.11 **Theorem (CHOMSKY-3 = DFA)**: A language is CHOMSKY-3 (right-linear), if and only if it is finitely acceptable (regular).

Summary of the chapter

We have proved the following result:

2.12 **Main theorem** : For function $f : A^* \xrightarrow{part} B^*$ the following statements are equivalent

- f is Turing-computable.
- The graph of f is a CHOMSKY-0-language.

In the literature it is often spoken about "recursion".

2.13 **Definition** : A *function* $f : A^* \xrightarrow{part} B^*$ is called *recursive* if f is Turing-computable or the graph of f is a CHOMSKY-0-language, respectively.

A set $L \subseteq A^*$ is called *recursive* if $\chi_L : A^* \to \{0, 1\}$ is recursive, i.e. if L is Turing-decidable.

Remark : Sometimes when dealing with partially defined recursive functions, we speak about "partially recursive" functions; by "recursive" we mean only total recursive functions. That is why we should make sure that we understand what exactly "recursive" means.

Church's thesis (1936) says:
The functions which are intuitively computable using algorithms are exactly the recursive functions, i.e. "recursion" is the mathematic formalization of the notion "algorithm".

Church's thesis cannot be formally proved, because "intuitively computable" is not a mathematically precise notion, but can only be confirmed by observations.

However, these observations are so grave that Church's thesis is universally accepted. Particularly it holds that:

- In more than 70 years of experience with computable functions no counterexamples have been found.
- Every further formalization of the notion of algorithm has proved to be equivalent to recursion: see below further examples for such formalization.
- Recursive functions have properties, for example, closure under the μ-operator (while loop), which should also hold for the algorithms.

Therefore, we can use the following informal but universally accepted notions for functions f and sets L:

- f is "computable" means that f is partially recursive.
- L is "decidable" means that L is recursive.

In addition to the already considered formalizations of the notion "computable", there exist other equivalent formalizations:

- **μ-recursive functions:**
  According to the works of GÖDEL, HERBRAND, KLEENE and ACKERMANN the set of all computable funcions $f : \mathbb{N}^n \xrightarrow{part} \mathbb{N}$ is defined inductively. For this purpose we first of all introduce very simple basic functions (such as successor function, projection function, constant function). Then we define how we can get new functions from the functions we already know (by the principles of composition, primitive recursion and μ-recursion).

- **Register machines with $k \geq 2$ registers:**
  Every register contains one natural number. The machine can test each of these registers, whether its contents is equal to or larger than 0. Depending on this test and the current state the machine can do the following operations with the values of the register

  – leave unchanged

  – increase by 1

  – reduce by 1, if we get no negative number.

  Note that 2 registers are already enough for the computation of all computable functions $f : \mathbb{N} \xrightarrow{part} \mathbb{N}$ (except for encoding). 2-register machines are also called 2-*counter machines* or MINSKY-*machines*.

- **λ-calculus:**
  This calculus introduced by CHURCH describes the computation using higher-order functions, i.e. functions, which take other functions as parameters. Today the λ-calculus is used to describe the semantics of functional languages.

- **First-order predicate calculus:**
  The notion of satisfiability of formulae turns out to be equivalent to the notion of computability.

Chapter V

Non-computable functions — undecidable problems

§1 Existence of non-computable functions

In this section we consider totally defined functions $f : \mathbb{N} \to \mathbb{N}$.

1.1 **The question of** Gödel, Turing, Church **1936** :

Is every function $f : \mathbb{N} \to \mathbb{N}$ algorithmically computable?
More precise: Is there a Turing machine which computes f or is the graph of f a Chomsky-0-language, respectively? In other words: Is f recursive?

The answer is "no". Below we will show that there exist non-computable functions $f : \mathbb{N} \to \mathbb{N}$.
More precisely:

1.2 **Theorem (Existence of non-computable functions)**:
There exist functions $f : \mathbb{N} \to \mathbb{N}$ which are not Turing-computable.

Idea of the proof : The cardinality of the set of *all* functions is *greater* (namely uncountable) than the cardinality of the set of Turing-*computable* functions (which is countable).

In order to prove this statement we need to make some preparations. Therefore, we put together definitions and results from the set theory.

1.3 **Definition** : Let M and N be sets. Then we define

1. $M \sim N$ (M and N *have the same cardinality*) if $\exists$ bijection $\beta : M \to N$
2. $M \preceq N$ (the cardinality of M is *less than or equal* to the cardinality of N) if $\exists N' \subseteq N$: $M \sim N'$, i.e. M has the same cardinality as N'.
3. $M \prec N$ (the cardinality of N is *greater* than the cardinality of M) if $M \preceq N$ and $M \not\sim N$.
4. M is (at most) *countable* if $M \preceq \mathbb{N}$.
5. M is *uncountable* if $\mathbb{N} \prec M$.
6. M is *finite* if $M \prec \mathbb{N}$.
7. M is *infinite* if $\mathbb{N} \preceq M$

Remark :

- $M \preceq N \Leftrightarrow \exists$ injection $\beta : M \to N$.
- Every finite set is countable.
- Every uncountable set is infinite.

1.4 **Theorem (**Schröder-Bernstein**)**: $M \preceq N$ and $N \preceq M$ imply $M \sim N$.

Proof : See, for example, P.R. Halmos, Naive Set Theory. □

1.5 **Corollary** : An infinite set M is countable if and only if $\mathbb{N} \sim M$, i.e. it holds that $M = \{\beta(0), \beta(1), \beta(2), \ldots\}$ for a bijection $\beta : \mathbb{N} \to M$.

Examples : The following sets are countable:

- $\mathbb{N}$,
- $\{n \in \mathbb{N} \mid n \text{ is even}\}$ and
- $\mathbb{N} \times \mathbb{N}$ (compare with the proof of Lemma 1.6).

However the following sets are over uncountable:

- $\mathcal{P}(\mathbb{N})$,
- $\mathbb{N} \to \mathbb{N}$ (set of functions from $\mathbb{N}$ to $\mathbb{N}$: see Lemma 1.7).

First we show:

1.6 **Lemma** :

The set of TURING-computable functions $f : \mathbb{N} \to \mathbb{N}$ is countable.

Proof : Let us assume that we have a countable set $\{q_0, q_1, q_2, \ldots\}$ of states and a countable set $\{\gamma_0, \gamma_1, \gamma_2, \ldots\}$ of tape symbols, where $\gamma_0 = \sqcup$ and $\gamma_1 = |$. Let t he sets $\{\gamma_0, \gamma_1, \gamma_2, \ldots\}$ and $\{q_0, q_1, q_2, \ldots\}$ be disjoint.

Every TURING-computable function $f : \mathbb{N} \to \mathbb{N}$ can be computed by a Turing machine of the form

$$\begin{aligned} \tau &= (Q, \{|\}, \Gamma, \delta, q_0, \sqcup) \text{ with} \\ Q &= \{q_0, \ldots, q_k\} \text{ for some } k \geq 0, \\ \Gamma &= \{\gamma_0, \ldots, \gamma_l\} \text{ for some } l \geq 1 \end{aligned}$$

Let $\mathcal{T}$ be the set of all those Turing machines.

It is sufficient to show: $\mathcal{T}$ is countable, because the set of TURING-computable functions can not be larger than the set of respective Turing machines.

Let us define for $k \geq 0, l \geq 1$:

$\mathcal{T}_{k,l} = \{\tau \in \mathcal{T} \mid \tau \text{ has } k+1 \text{ states and } l+1 \text{ tape symbols}\}$.

(Remark: we add 1 to k and l, because the states and tape symbols are counted from 0.)

Thus for every $\tau \in \mathcal{T}_{k,l}$ it holds that

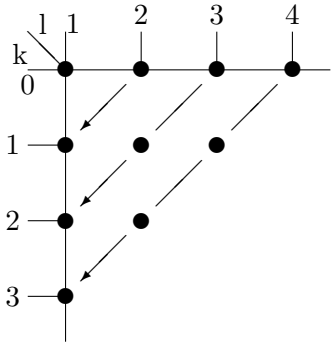
$$\tau = (\underbrace{\{q_0, \ldots, q_k\}}_{Q}, \{|\}, \underbrace{\{\gamma_0, \ldots, \gamma_l\}}_{\Gamma}, \delta, q_0, \sqcup),$$

where δ is one of the finitely many functions

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{R, L, S\}.$$

Thus it holds that: $\mathcal{T}_{k,l}$ is finite.

Because $\mathcal{T} = \bigcup_{\substack{k \geq 0 \\ l \geq 1}} \mathcal{T}_{k,l}$ holds, $\mathcal{T}$ is countable due to a bijection $\beta : \mathbb{N} \to \mathcal{T}$, which is defined according to the following *diagonal scheme* to arrange sets $\mathcal{T}_{k,l}$:



β counts the Turing machines in $\mathcal{T}$ in the following order:

1st diagonal: all Turing machines in $\mathcal{T}_{0,1}$;
2nd diagonal: all Turing machines in $\mathcal{T}_{0,2}$, then in $\mathcal{T}_{1,1}$;
3rd diagonal: all Turing machines in $\mathcal{T}_{0,3}$, then in $\mathcal{T}_{1,2}$,
then in $\mathcal{T}_{2,1}$;
and so on.

Thus we have proved Lemma 1.6. □

Remark : Therefore, the basic idea for the countability of $\mathcal{T}$ is the countability of sets of all pairs $\{(k,l) \mid k \geq 0, l \geq 1\}$ according to the diagonal scheme. This scheme goes back to G. Cantor's proof for countability of $\mathbb{N} \times \mathbb{N}$.

However, it holds:

1.7 **Lemma** :

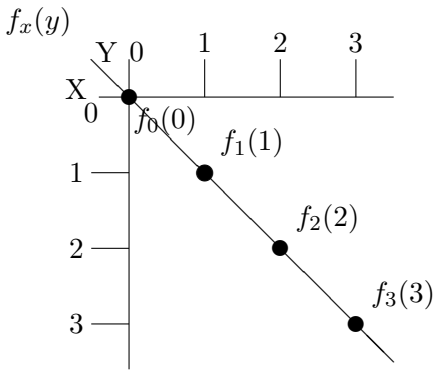
The set of *all* functions $f : \mathbb{N} \to \mathbb{N}$ is uncountable.

Proof :

Let $\mathcal{F} = \{f \mid f : \mathbb{N} \to \mathbb{N}\}$.
Hypothesis : $\mathcal{F}$ is countable, i.e. there is a bijection $\beta : \mathbb{N} \to \mathcal{F}$, so that $\mathcal{F} = \{f_0, f_1, f_2, \ldots\}$ with $f_n = \beta(n)$ for $n \in \mathbb{N}$ holds. Now let us define $g \in \mathcal{F}$ by $g(n) = f_n(n) + 1$ for all $n \in \mathbb{N}$. According to the hypothesis there is an index $m \in \mathbb{N}$ with $g = f_m$. However, then it holds that $f_m(m) = g(m) = f_m(m) + 1$. Contradiction! □

Remark :

Here we use *Cantor's diagonal method*, i.e. the values of $f_x(y)$ are considered on the "diagonals" $x = y$; then the contradiction arises at some point m :



The Lemmas 1.6 and 1.7 imply the theorem about the existence of non-computable functions.

This raises the following question:

Do there exist functions important for computer science which are non-computable?

§2 Concrete undecidable problem: halting for Turing machines

Below we consider the "binary" alphabet $B = \{0, 1\}$. Now we consider sets (languages) $L \subseteq B^*$ which are undecidable and for which the problem:

Given : $w \in B^*$
Question : Does $w \in L$ hold ?

is not algorithmically decidable.

First of all we want to consider the *halting problem for Turing machines* with the input alphabet B:

Given : Turing machine τ and string $w \in B^*$
Question : Does τ applied to w halt, i.e. is $h_\tau(w)$ defined?

The relevance of this problem for computer science: is it decidable whether a given program terminates for given input values?

First we translate the halting problem into an appropriate language $L \subseteq B^*$. In order to do it we need a binary encoding of Turing machines. Let us assume that we have countable sets

- $\{q_0, q_1, q_2, \ldots\}$ of states and
- $\{\gamma_0, \gamma_1, \gamma_2, \ldots\}$ of tape symbols, now with $\gamma_0 = 0, \gamma_1 = 1, \gamma_2 = \sqcup$.

We consider only Turing machines $\tau = (Q, B, \Gamma, \delta, q_0, \sqcup)$ with $Q = \{q_0, \ldots, q_k\}$ for $k \geq 0$, $\Gamma = \{\gamma_0, \ldots, \gamma_l\}$ for $l \geq 2$.

2.1 **Definition** : The *standard encoding* of such a Turing machine τ is the following string w_τ over $\mathbb{N} \cup \{\sharp\}$:

$w_\tau \;=\; k \,\sharp\, l$
.
.
.
$\boxed{\sharp\ i\ \sharp\ j\ \sharp\ s\ \sharp\ t\ \sharp\ nr(P)}$ for $\delta(q_i, \gamma_j) = (q_s, \gamma_t, P)$
.
.
.

where $P \in \{R, L, S\}$ and $nr(R) = 0, nr(L) = 1, nr(S) = 2$ holds and the substrings of the form $\boxed{\ldots}$ of w_τ are written down in lexicographical order of pairs (i, j).

Thus the *binary encoding* of τ is the string $bw_\tau \in B^*$ generated from w_τ by making the following substitution:

$\sharp \to \varepsilon$
$n \to 01^{n+1}$ for $n \in \mathbb{N}$

Example :

Consider the small right-machine $r = (\{q_0, q_1\}, B, B \cup \{\sqcup\}, \delta, q_0, \sqcup)$ with the TURING-table:

δ:				
q_0	0	q_1	0	R
q_0	1	q_1	1	R
q_0	$\sqcup$	q_1	$\sqcup$	R

Then

$w_\tau = 1\ \sharp\ 2$
$\sharp\ 0\ \sharp\ 0\ \sharp\ 1\ \sharp\ 0\ \sharp\ 0$
$\sharp\ 0\ \sharp\ 1\ \sharp\ 1\ \sharp\ 1\ \sharp\ 0$
$\sharp\ 0\ \sharp\ 2\ \sharp\ 1\ \sharp\ 2\ \sharp\ 0$

and

$bw_\tau = 011\ 0111$
$01\ 01\ 011\ 01\ 01$
$01\ 011\ 011\ 011\ 01$
$01\ 0111\ 011\ 0111\ 01$

Remark :

1. Not every string $w \in B^*$ is a binary encoding of a Turing machine, for example $w = \varepsilon$ is not.
2. It is decidable whether a given string $w \in B^*$ is a binary encoding of a Turing machine, i.e. the language $BW = \{w \in B^* \mid \exists\, Turing\ machine; \tau : w = bw_\tau\}$ is decidable.
3. The binary encoding is injective, i.e. $bw_{\tau_1} = bw_{\tau_2}$ implies $\tau_1 = \tau_2$.

2.2 **Definition (Special halting problem or self-application problem for Turing machines)**: The *special halting problem* or *self-application problem for Turing machines* is the language

$$K = \{bw_\tau \in B^* \mid \tau \text{ applied to } bw_\tau \text{ halts}\}.$$

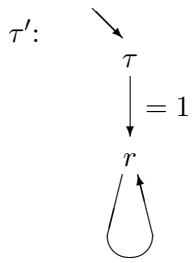
2.3 **Theorem** : $K \subseteq B^*$ is undecidable.

Proof : Again we use Cantor's diagonal method in a proof by contradiction.

Hypothesis: $K \subseteq B^*$ is decidable. Then there exists a Turing machine τ which computes $\chi_K : B^* \to \{0,1\}$. Recall that

$$\chi_K(w) = \begin{cases} 1 & \text{if } w \in K \\ 0 & \text{otherwise} \end{cases}$$

We change τ into a Turing machine τ' as follows. In a flow chart notation



Thus it holds: τ' gets into an infinite loop if τ halts with 1, and τ' halts (with 0) if τ halts with 0.

Then it holds:

$$\begin{aligned} \tau' \text{ applied to } bw_{\tau'} \text{ halts} &\Leftrightarrow \tau \text{ applied to } bw_{\tau'} \text{ halts with } 0 \\ &\Leftrightarrow \chi_K(bw_{\tau'}) = 0 \\ &\Leftrightarrow bw_{\tau'} \notin K \\ &\Leftrightarrow \tau' \text{ applied to } bw_{\tau'} \text{ does not halt.} \\ &\quad \text{Contradiction!} \end{aligned}$$

□

Now we will show a general method, which lets us prove that some problems are undecidable by using some other problems (e.g. K) which are known to be undecidable. This method is the so-called *reduction*.

2.4 **Definition (Reduction)**: Let $L_1 \subseteq \Sigma_1^*$ and $L_2 \subseteq \Sigma_2^*$ be languages. Then L_1 is *reducible* to L_2, shortly $L_1 \leq L_2$, if there is a total computable function $f : \Sigma_1^* \to \Sigma_2^*$ so that for all $w \in \Sigma_1^*$ it holds that: $w \in L_1 \Leftrightarrow f(w) \in L_2$. We also write: $L_1 \leq L_2$ *using* f.

Idea: L_1 is a *special case* of L_2.

2.5 **Lemma (Reduction)**: Let $L_1 \leq L_2$. Then it holds that:

(i) If L_2 is decidable, then L_1 is also decidable.

(ii) If L_1 is undecidable, then L_2 is also undecidable.

Proof : Because (ii) is the contraposition of (i), it is enough to show (i). Thus let $L_1 \leq L_2$ using total computable function f and let L_2 be decidable, i.e. χ_{L_2} is computable. Then the *composition* $\chi_{L_2}(f) = f \circ \chi_{L_2}$ (first apply f, then χ_{L_2}) is computable. It holds

$$\chi_{L_1} = f \circ \chi_{L_2},$$

because for all $w \in \Sigma_1$

$$\chi_{L_1}(w) = \left\{ \begin{array}{lll} 1 & \text{if} & w \in L_1 \\ 0 & \text{otherwise} & \end{array} \right\} = \left\{ \begin{array}{lll} 1 & \text{if} & f(w) \in L_2 \\ 0 & \text{otherwise} & \end{array} \right\} = \chi_{L_2}(f(w))$$

Thus χ_{L_1} is computable, i.e. L_1 is decidable. □

We apply reduction to the general halting problem for Turing machines.

2.6 **Definition (general halting problem for TM)**: The *(general) halting problem* for Turing machines is the language

$$H = \{bw_\tau 00u \in B^* \mid \tau \text{ applied to } u \text{ halts}\}.$$

2.7 **Theorem** : $H \subseteq B^*$ is undecidable.

Proof : According to the reduction lemma, it is enough to show $K \leq H$, i.e. K is a special case of H. Here it is trivial: choose $f : B^* \to B^*$ with $f(w) = w00w$. Then f is computable and it holds:

$$bw_\tau \in K \Leftrightarrow f(bw_\tau) \in H.$$

□

2.8 **Definition** : The *blank tape halting problem for Turing machines* is the language

$$H_0 = \{bw_\tau \in B^* \mid \tau \text{ applied to the blank tape halts}\}.$$

2.9 **Theorem** : H_0 is undecidable.

Proof : We show $H \leq H_0$. First we describe for a given Turing machine τ and a string $u \in B^*$ a Turing machine τ_u, which operates as follows:

- Applied to the blank tape τ_u first writes u on the tape. Then τ_u operates as τ (applied to u).
- It is not important how τ_u operates if the tape is not blank at the beginning.

We can construct τ_u from τ and u. Thus there exists a computable function $f : B^* \to B^*$ with

$$f(bw_\tau 00u) = bw_{(\tau_u)}.$$

Then it holds

$$\begin{aligned} & bw_\tau 00u \in H \\ \Leftrightarrow \quad & \tau \text{ applied to } u \text{ halts.} \\ \Leftrightarrow \quad & f(bw_\tau 00u) \in H_0. \end{aligned}$$

Thus $H \leq H_0$ using f holds. □

We got acquainted with three variants of halting problems, namely K, H and H_0. All three variants are undecidable.

For K we have shown it directly using a *diagonal method*, and for H and H_0 we have shown it by *reduction*: $K \leq H \leq H_0$.

In all three variants we dealt with the question whether there exists a decision method which decides halting for *every* given Turing machine.

In all cases the answer is "no".

Perhaps we can at least construct a decision procedure for a given Turing machine τ which decides whether τ halts.

2.10 **Definition** : Consider a Turing machine τ. The *halting problem for* τ is the language

$$H_\tau = \{w \in B^* \mid \tau \text{ applied to } w \text{ halts}\}.$$

For certain Turing machines H_τ is decidable, for example, for the small right-machine r it holds that

$$H_r = B^*$$

and for the Turing machine τ with :

τ

$$H_\tau = \emptyset.$$

However, there also exist Turing machines τ for which H_τ is undecidable. These are the "programmable" or *universal Turing machines*.

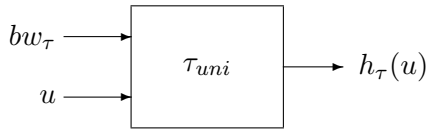
2.11 **Definition** : A Turing machine τ_{uni} with the input alphabet B is called *universal* if for the function $h_{\tau_{uni}}$ computed by τ_{uni} the following holds:

$$h_{\tau_{uni}}(bw_\tau 00u) = h_\tau(u),$$

i.e. τ_{uni} can simulate every other Turing machine τ applied to input string $u \in B^*$.

Relevance for computer science :

The Turing machine τ_{uni} is an *interpreter* of Turing machines, which itself is written as a Turing machine:



Thus the construction of τ_{uni} corresponds to writing an interpreter of a simple programming language in this language.

2.12 **Lemma** : Universal Turing machines τ_{uni} can be constructed effectively.

Proof : Obviously τ_{uni} applied to a string $w \in B^*$ works as follows :

- τ_{uni} determines whether w has the form $w = bw_\tau 00u$ for a Turing machine τ (with input alphabet B) and a string $u \in B^*$.
- If no, then τ_{uni} goes into an infinite loop.
- If yes, then τ_{uni} simulates the Turing machine τ applied to u.

The last can be described in more detail as follows:

1. τ_{uni} changes the tape to
$$bw_\tau ¢ q_0 u \$,$$
where q_0 is the initial state of τ.

2. In general τ_{uni} sees on the tape
$$bw_\tau ¢ v_l q a v_r \$$$
where $v_l q a v_r$ represents the current configuration of the Turing machine τ. τ_{uni} remembers the pair (q, a) and determines the value $\delta(q, a)$ of the transition function δ (which is binary encoded in bw_τ) of τ.

3. If $\delta(q, a)$ is defined, then τ_{uni} returns to the configuration $v_l q a v_r$ of τ and then makes the necessary transition. Usually τ_{uni} must run back and forth several times, because it can store only a finite amount of information in its memory Q. As a result we have a new label of the tape of the form
$$bw_\tau ¢ v_l' q' a' v_r' \$,$$
and τ_{uni} operates further as described in (2).

4. If $\delta(q, a)$ is undefined, i.e. $v_l q a v_r$ is a final configuration of τ, then τ_{uni} erases the substrings $bw_\tau ¢$ and \$ and halts in a final configuration of the form
$$v_l q_e a v_r.$$

It is obvious that the results of τ_{uni} and τ are equal:

$$\omega(v_l q a v_r) = \omega(v_l q_e a v_r).$$

Thus it holds that:

$$h_{\tau_{uni}}(bw_\tau 00u) = h_\tau(u)$$

as desired. □

2.13 **Theorem** : $H_{\tau_{uni}} \subseteq B^*$ is undecidable.

Proof : According to the definition of τ_{uni} it holds that $H \leq H_{\tau_{uni}}$ using $f = id_{B^*}$. For the Turing machine τ_{uni} constructed in the proof it even holds that $H = H_{\tau_{uni}}$, because τ_{uni} gets into an infinite loop if a given string $w \in B^*$ is not from H, i.e. does not have the form $w = bw_\tau 00u$. □

Thus we have shown: $K \leq H = H_{\tau_{uni}} \leq H_0$

§3 Recursive enumerability

In this section we deal with a weakening of the notion of decidability (recursion) for languages L over an alphabet A.

3.1 **Definition** : A language $L \subseteq A^*$ is called *recursively enumerable*, shortly *r.e.*, if $L = \emptyset$ or there exists a total computable function $\beta : \mathbb{N} \to A^*$ with

$$L = \beta(\mathbb{N}) = \{\beta(0), \beta(1), \beta(2), \ldots\},$$

i.e. L is the range of values of β.

Let us revise the notion "countable" for comparison. $L \subseteq A^*$ is *countable*, if $L \preceq \mathbb{N}$ or equivalent: if there is a total function $\beta : \mathbb{N} \to A^*$ with

$$L = \emptyset \text{ or } L = \beta(\mathbb{N}) = \{\beta(0), \beta(1), \beta(2), \ldots\}.$$

Thus the difference is that by "countable" we mean that the function β may not necessarily be computable.

We want to consider the new notion of recursive enumerability in more detail.

3.2 **Definition (Semi-decidability)**:

A language $L \subseteq A^*$ is called *semi-decidable* if the *partial characteristic function of L*

$$\psi_L : A^* \xrightarrow{part} \{1\}$$

is computable. The partial function ψ_L is defined as follows:

$$\psi_L(v) = \begin{cases} 1 & \text{if } v \in L \\ \text{undef.} & \text{otherwise} \end{cases}$$

Thus a semi-decision procedure for a set $L \subseteq A^*$ is a "yes-procedure", while a decision procedure is a "yes-no-procedure".

Remark : For all languages $L \subseteq A^*$ it holds that:

1. L is semi-decidable $\Leftrightarrow$ L is TURING-acceptable.
2. L is decidable $\Leftrightarrow$ L and $\overline{L} = A^* - L$ are TURING-acceptable or semi-decidable.

Proof : (1) follows from the definition of "semi-decidable". (2) follows from the respective theorem about TURING-acceptability. □

3.3 **Lemma** : For all languages $L \subseteq A^*$ it holds that: L is recursively enumerable $\Leftrightarrow$ L is semi-decidable.

Proof : "⇒": Let L be recursively enumerable using the function $\beta : \mathbb{N} \to A^*$. Let $f : A^* \xrightarrow{part} \{1\}$ be computable by the following algorithm:

- Input: $w \in A^*$
- Apply β to $n = 0, 1, 2, \ldots$ successively.
- If at some point $\beta(n) = w$ holds, halt with output 1. (Otherwise the algorithm does not halt.)

$f = \psi_L$ holds. Thus L is semi-decidable.

"⇐" : (dovetailing)

Let L be semi-decidable by the Turing machine τ. If $L \neq \emptyset$, then we must provide a total computable function $\beta : \mathbb{N} \to A^*$ with $\beta(\mathbb{N}) = L$.

Let w_0 be some fixed string from L. β is computed by the following algorithm:

- Input : $n \in \mathbb{N}$
- Determine whether n is a prime number encoding of a pair $(w, k) \in A^* \times \mathbb{N}$. Every such pair can be uniquely encoded by a multiplication of prime numbers.

  Details: think yourself.
- If no, the output is w_0.
- Otherwise, determine whether τ applied to w halts in at most k steps (which means the value 1, i.e. "$w \in L$", is produced).
  If yes, the output is w.
  Otherwise, the output is w_0.

We show: $\beta(\mathbb{N}) = L$.
"$\subseteq$" : The above algorithm produces only strings from L.
"$\supseteq$" : Let $w \in L$. Then there is a number of steps $k \in \mathbb{N}$, such that τ applied to w halts in k steps and thus produces 1 ("$w \in L$"). Let n be the prime number encoding of (w, k). Then $w = \beta(n) \in \beta(\mathbb{N})$ holds. □

3.4 **Theorem (Characterizing recursive enumerability)**: For all languages $L \subseteq A^*$ the following statements are equivalent:

1. L is recursively enumerable.
2. L is the range of results of a Turing machine τ, i.e. $L = \{v \in A^* \mid \exists w \in \Sigma^* \text{ with } h_\tau(w) = v\}$.
3. L is semi-decidable.
4. L is the halting range of a Turing machine τ, i.e. $L = \{v \in A^* \mid h_\tau(v) \text{ exists}\}$.
5. L is TURING-acceptable.
6. L is Chomsky-0.

3.5 **Corollary** : For all languages $L \subseteq A^*$ it holds that:

L is decidable (recursive) $\Leftrightarrow$ L and $\overline{L} = A^* - L$ are recursively enumerable.

The following extension of the reduction-lemma holds:

3.6 **Lemma** : Let $L_1 \leq L_2$. Then it holds : If L_2 is recursively enumerable, then L_1 is also recursively enumerable.

Proof : Let $L_1 \leq L_2$ using f. Then $\psi_{L_1} = f \circ \psi_{L_2}$ holds (first apply f, then ψ_{L_2}). □

Now we show that the halting problems for Turing machines are recursively enumerable.

3.7 **Theorem** : $H_0 \subseteq B^*$ is recursively enumerable.

Proof : H_0 is semi-decidable by the Turing machine τ_0, applied to $w \in B^*$, which operates as follows:

- τ_0 determines whether w is of the form $w = bw_\tau$ for some Turing machine τ.
- If no, then τ_0 goes into an infinite loop.
- If yes, then τ_0 lets the Turing machine τ run on the blank tape. If τ halts, then τ_0 produces the value 1. Otherwise, τ_0 runs further indefinitely.

□

Thus we get the following main result about the halting of Turing machines.

3.8 **Main theorem (Halting of Turing machines)**:

1. The halting problems K, H, H_0 and $H_{\tau_{uni}}$ are recursively enumerable, but not decidable.
2. The complementary problems $\overline{K}, \overline{H}, \overline{H_0}$ and $\overline{H_{\tau_{uni}}}$ are countable, but not recursively enumerable.

Proof :

"(1)": It holds that $K \leq H = H_{\tau_{uni}} \leq H_0$.
"(2)": If the complementary problems were recursively enumerable, then the halting problems would be decidable according to "(1) r.e." and the above corollary. Contradiction! □

§4 Automatic program verification

Can the computer verify programs, i.e. determine whether a program $\mathcal{P}$ satisfies the given specification $\mathcal{S}$? We consider the following verification problem in more detail:

Given : Program $\mathcal{P}$ and specification $\mathcal{S}$
Question : Does $\mathcal{P}$ satisfy the specification $\mathcal{S}$?

We formalize this problem as follows:

- Program $\mathcal{P}$ $\stackrel{\wedge}{=}$ Turing machine τ with input alphabet $B = \{0, 1\}$
- Specification $\mathcal{S}$ $\stackrel{\wedge}{=}$ subset $\mathcal{S}$ of $\mathcal{T}_{B,B}$, the set of all TURING-computable functions $h : B^* \stackrel{part}{\longrightarrow} B^*$
- $\mathcal{P}$ satisfies $\mathcal{S}$ $\stackrel{\wedge}{=}$ $h_\tau \in \mathcal{S}$

The answer is given in RICE's theorem (1953, 1956):

The verification problem is undecidable except for two trivial exceptions:

- $\mathcal{S} = \emptyset$: answer always "no".
- $\mathcal{S} = \mathcal{T}_{B,B}$: answer always "yes"

4.1 **Theorem (RICE's Theorem)**: Let $\mathcal{S}$ be an arbitrary non-trivial subset of $\mathcal{T}_{B,B}$, i.e. it holds that $\emptyset \subset \mathcal{S} \subset \mathcal{T}_{B,B}$. Then the language

$$BW(\mathcal{S}) = \{bw_\tau \mid h_\tau \in \mathcal{S}\} \subseteq B^*$$

of the binary encodings of all Turing machines τ, whose computed function h_τ is in the set of functions $\mathcal{S}$, is undecidable.

Proof : In order to show the undecidability of $BW(\mathcal{S})$, we reduce the undecidable problem H_0 or its complement $\overline{H_0} = B^* - H_0$, respectively, to $BW(\mathcal{S})$.

Therefore, first we consider an arbitrary function $g \in \mathcal{T}_{B,B}$ and an arbitrary Turing machine τ. Let τ_g be a TM computing g. Furthermore, let $\tau' = \tau'(\tau, g)$ be the following TM depending on τ and g. When applied to a string $v \in B^*$, $\tau'(\tau, g)$ operates as follows:

1. First v is ignored and τ' operates as τ applied to the blank tape.
2. If τ halts, then τ' operates as τ_g applied to v.

Let $\Omega \in \mathcal{T}_{B,B}$ be the totally undefined function. Then the function $\tau'(\tau, g)$ computed by $h_{\tau'(\tau,g)}$ is computable by the following case distinction:

$$h_{\tau'(\tau,g)} = \begin{cases} g & \text{if } \tau \text{ halts on the blank tape} \\ \Omega & \text{otherwise} \end{cases}$$

For a given g there exists a total computable function $f_g : B^* \to B^*$ with

$$f_g(bw_\tau) = bw_{\tau'(\tau,g)},$$

i.e. f_g computes the binary encoding of a Turing machine $\tau'(\tau, g)$ from a given binary encoding of a Turing machine τ.

We use this function f_g for reduction. We distinguish between two cases.

Case 1 : $\Omega \notin \mathcal{S}$
Choose some function $g \in \mathcal{S}$. It is possible, because $\mathcal{S} \neq \emptyset$.
We show: $H_0 \leq BW(\mathcal{S})$ using f_g.

$$\begin{aligned} bw_\tau \in H_0 &\Leftrightarrow \tau \text{ halts applied to the blank tape} \\ &\Leftrightarrow h_{\tau'(\tau,g)} = g \\ &\Leftrightarrow \{\text{It holds } h_{\tau'(\tau,g)} \in \{g, \Omega\},\ \Omega \notin \mathcal{S} \text{ and } g \in \mathcal{S}.\} \\ &\quad\ h_{\tau'(\tau,g)} \in \mathcal{S} \\ &\Leftrightarrow bw_{\tau'(\tau,g)} \in BW(\mathcal{S}) \\ &\Leftrightarrow f_g(bw_\tau) \in BW(\mathcal{S}) \end{aligned}$$

Case 2 : $\Omega \in \mathcal{S}$
Choose any function $g \notin \mathcal{S}$. It is possible, because $\mathcal{S} \neq \mathcal{T}_{B,B}$.
We show: $\overline{H_0} \leq BW(\mathcal{S})$ using f_g.

$$\begin{aligned} bw_\tau \notin H_0 &\Leftrightarrow \tau \text{ does not halt applied to the blank tape} \\ &\Leftrightarrow h_{\tau'(\tau,g)} = \Omega \\ &\Leftrightarrow \{\text{It holds } h_{\tau'(\tau,g)} \in \{g, \Omega\},\ \Omega \in \mathcal{S} \text{ and } g \notin \mathcal{S}.\} \\ &\quad\ h_{\tau'(\tau,g)} \in \mathcal{S} \\ &\Leftrightarrow bw_{\tau'(\tau,g)} \in BW(\mathcal{S}) \\ &\Leftrightarrow f_g(bw_\tau) \in BW(\mathcal{S}) \end{aligned}$$

Thus we have proved the Rice's theorem. □

It is clear that Rice's theorem proves that it is not worth trying to algorithmically verify the semantics of Turing machines or programs on the basis of their syntactic form, i.e. their input / output behavior.

Nevertheless, automatic program verification (also called :"*model checking*", where "model" is equivalent to "program") is a very active field of research these days. For example, it is possible to analyse programs whose behavior can be described by *finite automata*.

§5 Grammar problems and Post correspondence problem

First we consider two problems for CHOMSKY-0-grammars.

5.1 **Definition (decision problem for CHOMSKY-0)**:
The *decision problem for CHOMSKY-0-grammars* is:

Given: CHOMSKY-0-grammar $G = (N, T, P, S)$ and a string $w \in T^*$
Question: Does $w \in L(G)$ hold?

5.2 **Theorem** : The decision problem for CHOMSKY-0-grammars is undecidable.

Idea of the proof : With appropriate binary encoding of grammar G and string w the halting problem for Turing machines can be reduced to the following:
$H \leq$ *decision problem.*

For the direct proof compare also the proofs of Theorem 1.2 and Lemma 1.6.

5.3 **Definition** : The *derivation problem for CHOMSKY-0-grammars* is as follows:

Given: CHOMSKY-0-grammar $G = (N, T, P, S)$ and
two strings $u, v \in (N \cup T)^*$
Question: Does $u \vdash_G^* v$ hold?

5.4 **Theorem** : The derivation problem for CHOMSKY-0-grammars is undecidable.

Proof : Obviously the reduction *decision problem* $\leq$ *derivation problem* holds, because for all grammars $G = (N, T, P, S)$ and strings $w \in T^*$ it holds that:

$$w \in L(G) \Leftrightarrow S \vdash_G^* w.$$

□

Next we consider some sort of a puzzle game, the POST correspondence problem. It was introduced by E. POST (1946) and is abbreviated as *PCP* (from "Post Correspondence Problem"). The point is to construct a string in two different ways.

For subsequent encoding purposes let us assume that we have a countable infinite set of symbols.

$$SYM = \{a_0, a_1, a_2, \ldots\}.$$

We always consider strings over SYM. Let $X \subseteq SYM$ be an alphabet.

5.5 **Definition** : An *input/instance of the PCP* is a finite sequence Y of pairs of strings, i.e.

$$Y = ((u_1, v_1), \ldots, (u_n, v_n))$$

with $n \geq 1$ and $u_i, v_i \in SYM^*$ for $i = 1, \ldots, n$. If $u_i, v_i \in X^*$ holds, then Y is called an *input of the PCP over* X.

A *correspondence* or *solution* of Y is a finite sequence of indices

$$(i_1, \ldots, i_m)$$

with $m \geq 1$ and $i_1, \ldots, i_m \in \{1, \ldots, n\}$, such that

$$u_{i_1} u_{i_2} \ldots u_{i_m} = v_{i_1} v_{i_2} \ldots v_{i_m}$$

holds. Y is called *solvable* if there exists a solution of Y.

5.6 **Definition** :

1. The POST *correspondence problem (PCP)* is the following problem:

   Given: Input Y of PCP
   Question: Does Y have a correspondence ?

2. The *PCP over* X is the following problem:

   Given: Input Y of PCP over X
   Question: Does Y have a correspondence ?

3. The *modified PCP (shortly MPCP)* is the following problem:

   Given: Input $Y = ((u_1, v_1), \ldots, (u_n, v_n))$ of PCP,
   where $u_i \neq \varepsilon$ for $i = 1, \ldots, n$
   Question: Does Y have a correspondence $(i_1, \ldots, i_m)$ with $i_1 = 1$?
   Thus the correspondence should begin with the first pair of strings,
   such that $u_{i_1} u_{i_2} \ldots u_{i_m} = v_{i_1} v_{i_2} \ldots v_{i_m}$ holds.

Example : First we consider

$$Y_1 = ((10, 00), (1, 101), (011, 11)),$$

i.e.

$$\begin{array}{lllllll} u_1 & = & 10 & , v_1 & = & 00 \\ u_2 & = & 1 & , v_2 & = & 101 \\ u_3 & = & 011 & , v_3 & = & 11 \end{array}$$

Then $(2, 3, 1, 3)$ is a correspondence of Y_1, because it holds that:

$$u_2u_3u_1u_3 = 101110011 = v_2v_3v_1v_3$$

(2) Next we consider

$$Y_2 = ((00, 10), (1, 0), (101, 0)).$$

Y_2 has no correspondence, because for every pair of strings $(u, v) \in Y_2$ it holds that: neither u is the initial string of v, nor v is the initial string of u.

Even simple inputs of PCP can have a high degree of complexity. For example, the shortest solution for

$$Y = ((001, 0), (01, 011), (01, 101), (10, 001))$$

already has **66** indices (compare with Schöning, 4th edition, p. 132). Thus we consider the question whether the PCP is decidable.

Remark : For a one-element alphabet X, the PCP is decidable over X.

Proof : Let $X = \{I\}$. Every string I^n over X can be identified with a natural number n. Thus every input Y of PCP over X can be considered as a sequence of pairs of natural numbers:

$$Y = ((u_1, v_1), \ldots, (u_n, v_n))$$

with $n \geq 1$ and $u_i, v_i \in \mathbb{N}$ for $i = 1, \ldots, n$. Y is solvable if and only if there is a set of indices $(i_1, \ldots, i_m)$ with $m \geq 1$ and $i_j \in \{1, \ldots, n\}$ for $j = 1, \ldots, m$, such that

$$\sum_{j=1}^{m} u_{i_j} = \sum_{j=1}^{m} v_{i_j}$$

holds. Therefore, we deal with solvability of a problem for natural numbers.

We show:

Y is solvable $\Leftrightarrow$ (a) $\exists j \in \{1, \ldots, n\} : u_j = v_j$
or
(b) $\exists k, j \in \{1, \ldots, n\}$:
$u_k < v_k$ and $u_j > v_j$

"$\Rightarrow$": Proof by contradiction: If neither (a) nor (b) hold, then all pairs $(u, v) \in Y$ have the property $u < v$ or all pairs $(u, v) \in Y$ have the property $u > v$, respectively. Thus the strings composed of u-pairs would be shorter or longer than the strings composed of v-pairs. Thus Y is not solvable.

"$\Leftarrow$": If $u_j = v_j$ holds, the trivial sequence of indices (j) is the solution of Y. If $u_k < v_k$ and $u_l > v_l$ hold, then the sequence

$$(\underbrace{k, \ldots, k}_{(u_l - v_l) \text{ times}} , \underbrace{l, \ldots, l}_{(v_k - u_k) \text{ times}})$$

is the solution of Y, then it holds that

$$u_k(u_l - v_l) + u_l(v_k - u_k) = v_k(u_l - v_l) + v_l(v_k - u_k).$$

Obviously the properties (a) and (b) are decidable. Thus the PCP is decidable over a one-element alphabet. □

Now we consider the general case of the PCP. Our goal is the following theorem:

5.7 **Theorem** : The PCP over X is undecidable for every alphabet X with $|X| \geq 2$.

We show the theorem by reduction of the derivation problem for Chomsky-0-grammars on the PCP over $\{0, 1\}$. We consider the following three reductions:

Derivation problem $\leq$ MPCP $\leq$ PCP $\leq$ PCP over $\{0, 1\}$.

We start with the first reduction.

5.8 **Lemma** : *Derivation problem* $\leq MPCP$

Proof : We introduce an algorithm which constructs an input $Y_{G,u,v}$ for MPCP for a given CHOMSKY-0-grammar $G = (N, T, P, S)$ with $(N \cup T \subseteq SYM)$ and given strings $u, v \in (N \cup T)^*$ such that the following holds:

(1) $u \vdash_G^* v \Leftrightarrow Y_{G,u,v}$ has a correspondence starting with the first pair of strings.

We use a symbol $\sharp$, which does not occur in $N \cup T$. Then $Y_{G,u,v}$ consists of the following pairs of strings:

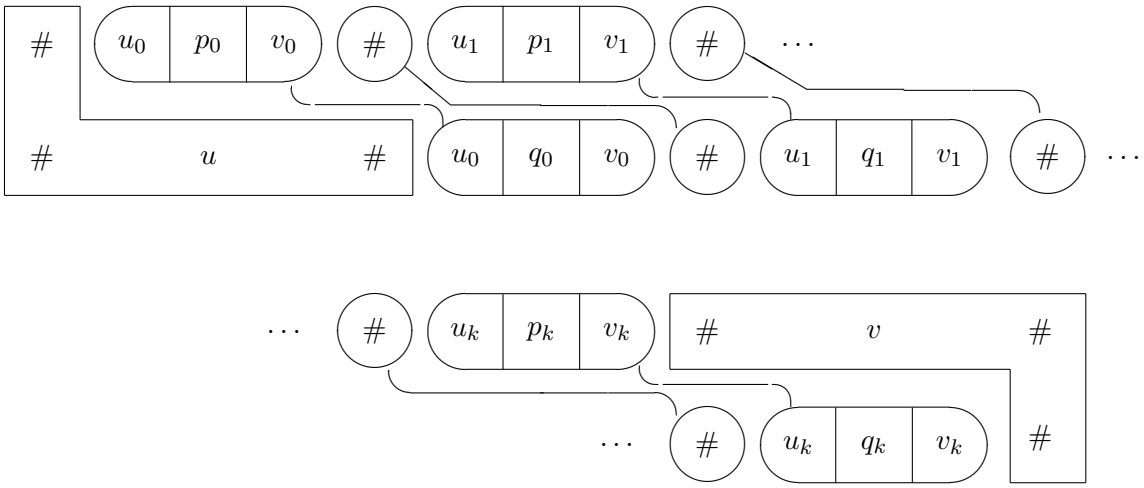
- The first pair of strings: $(\sharp, \sharp u \sharp)$
- Productions: (p, q) for all $p \to q \in P$
- Copy: (a, a) for all $a \in N \cup T \cup \{\sharp\}$
- Last pair: $(\sharp v \sharp, \sharp)$

The exact sequence of pairs of strings in $Y_{G,u,v}$ is irrelevant, except for the first pairs of strings. Now we show that (1) holds.

"⇒": It holds that $u \vdash_G^* v$. Then there is a derivation from u to v of the form

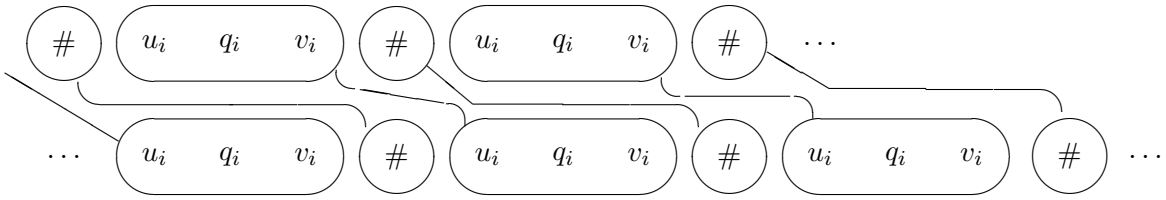
$$\begin{aligned} (2)\ u = u_0 p_0 v_0 &\vdash_G u_0 q_0 v_0 = u_1 p_1 v_1 \\ &\vdash_G u_1 q_1 v_1 = u_2 p_2 v_2 \\ &\dots \\ &\vdash_G u_k q_k v_k = v \end{aligned}$$

where $p_0 \to q_0, \dots, p_k \to q_k \in P$ holds. There exists the following correspondence of $Y_{G,u,v}$, where we write down both components of pairs of strings one after another in two rows :



The "angle pieces" with u and v represent the first pair of strings and the last pair, respectively. The substrings $u_i, v_i, \sharp$ in round frames are constructed character-by-character by copy-pairs (a, a) for $a \in N \cup T \cup \{\sharp\}$. The substrings p_i, q_i in angled frames are placed by the respective pair of productions (p_i, q_i). The connecting lines mean that both strings belong to the same pair of strings.

"$\Leftarrow$" : Now consider a correspondence of $Y_{G,u,v}$ which begins with the first pair of strings. The form of the pairs of strings in $Y_{G,u,v}$ implies that the correspondence must be constructed as shown under "$\Rightarrow$" with the exception that the substrings u and $u_i q_i v_i$ respectively can repeat infinitely many times by applying only copy-pairs (a, a) :



In order to make the correspondence complete, we must be able to place the final pair. For this purpose we must insert pairs of productions (p_i, q_i) such that the correspondence describes a derivation (2) from u to v.

Note that "$\Leftarrow$" holds only because we begin with the *first* pair of strings in the sense of $MPCP$. Without this restriction, for example, the copy-pair $(\sharp, \sharp)$ gives us a trivial correspondence which does not imply $u \vdash_G^* v$. □

5.9 **Lemma** : $MPCP \leq PCP$

Proof : We introduce an algorithm which constructs an input Y_{PCP} of the PCP for the given input $Y = ((u_1, v_1), \ldots, (u_n, v_n))$ of the $MPCP$, i.e. with $u_i \neq \varepsilon$ for $i = 1, \ldots, n$, such that the following holds:

(3) Y has a correspondence which starts with the first pair of strings
$\Leftrightarrow Y_{PCP}$ has a correspondence.

Idea : Let us construct Y_{PCP} in such a way that every correspondence necessarily starts with the first pair of strings.

For this purpose we use new symbols β and γ which may not appear in any of the strings of Y. We define two mappings

$\rho : SYM^* \to SYM^*$,
$\lambda : SYM^* \to SYM^*$,

where for $w \in SYM^*$

- $\rho(w)$ is generated from w by inserting the symbol β to the *right* of every character of w,
- $\lambda(w)$ is generated from w by inserting the symbol β to the *left* of every character of w.

For example it holds that

$$\rho(GTI) = G\beta T\beta I\beta \quad \text{and} \quad \lambda(GTI) = \beta G\beta T\beta I.$$

For the mappings ρ and λ the following holds: for all $u, v \in SYM^*$

1. $\rho(\varepsilon) = \varepsilon$ and $\lambda(\varepsilon) = \varepsilon$
2. $\rho(uv) = \rho(u)\rho(v)$ and $\lambda(uv) = \lambda(u)\lambda(v)$
3. $u = v \Leftrightarrow \beta\rho(u) = \lambda(v)\beta$

The statements 1 and 2 mean that ρ and λ are *string homomorphisms*, i.e. ε and the concatenation are preserved.

From Y we construct the following Y_{PCP} with pairs of strings which are numbered from 0 to $n+1$:

$$\begin{aligned} Y_{PCP} = (\ & (\beta\rho(u_1), \lambda(v_1)), && \text{0 pair of strings} \\ & (\rho(u_1), \lambda(v_1)) && \text{, 1st pair of strings} \\ & \quad \dots && \text{, } \dots \\ & (\rho(u_n), \lambda(v_n)) && \text{, } n\text{-th pair of strings} \\ & (\gamma, \beta\gamma)) && \text{, } n+1\text{-th pair of strings} \end{aligned}$$

Now we show that (3) holds:

"$\Rightarrow$": Let $(1, i_1, \dots, i_m)$ be a correspondence of Y, i.e. $u_1 u_{i_2} \dots u_{i_m} = v_1 v_{i_2} \dots v_{i_m}$. According to the statement 3 it holds that :

$$\beta\rho(u_1 u_{i_2} \dots u_{i_m})\gamma = \lambda(v_1 v_{i_2} \dots v_{i_m})\beta\gamma$$

By applying statement 2 several times, we get :

$=$

$\beta\rho(u_1)$ $\lambda(v_1)$	$\rho(u_{i_2})$ $\lambda(v_{i_2})$	$\dots$ $\dots$	$\rho(u_{i_m})$ $\lambda(v_{i_m})$	γ $\beta\gamma$

By using frames we have put together strings which appear in a pair of strings of Y_{PCP}. We see that:

$$(1, i_2, \dots, i_m, n+1)$$

is a correspondence of Y_{PCP}.

"$\Leftarrow$": We show this direction only for the case $v_i \neq \varepsilon$ for $i = 1, \dots, n$:
Let $(i_1, \dots, i_m)$ be some correspondence of Y_{PCP}. Then $i_1 = 0$ and $i_m = n+1$ hold, because only the pairs of strings in the 1st pair of strings $(\beta\rho(u_1), \lambda(v_1))$ start with the same symbol and only the strings in the $(n+1)$-th pair of strings $(\gamma, \beta\gamma)$ end with the same symbol. Let $k \in \{2, \dots, m\}$ be the smallest index with $i_k = n+1$. Then $(i_1, \dots, i_k)$ is also a correspondence of Y_{PCP}, because γ appears only as the last symbol in the appropriate correspondence string. The form of the pairs of the strings implies that:

$$i_j \neq 1 \text{ for } j = 2, \dots, k-1.$$

Otherwise, there would be two consequent β's in the substring $\rho(u_{i_{j-1}})\beta\rho(u_1)$ which could not be reproduced by placing $\lambda(v_i)$'s, because $v_i \neq \varepsilon$.

Thus the correspondence string has the following structure

$=$

$\beta\rho(u_1)$ $\lambda(v_1)$	$\rho(u_{i_2})$ $\lambda(v_{i_2})$	$\dots$ $\dots$	$\rho(u_{i_{k-1}})$ $\lambda(u_{i_{k-1}})$	γ $\beta\gamma$

By applying the statements (ii) and (iii) we conclude that

$$u_1 u_{i_2} \dots u_{i_{k-1}} = v_1 v_{i_2} \dots v_{i_{k-1}}.$$

Thus $(1, i_2, \dots, i_{k-1})$ is a correspondence of Y. In the case that there is a $v_i = \varepsilon$, the reasoning is more difficult.

□

5.10 **Lemma** : $PCP \leq PCP$ over $\{0, 1\}$

Proof : For reduction we use a binary encoding over SYM, i.e. an injective computable function

$$bw : SYM^* \to \{0,1\}^*,$$

with which we have got acquainted while considering binary encoding of Turing machines.

Now consider an input $Y = ((u_1, v_1), \ldots, (u_n, v_n))$ of the PCP. Then we define

$$bw(Y) = ((bw(u_1), bw(v_1)), \ldots, (bw(u_n), bw(v_n)))$$

as input of PCP over $\{0,1\}$. Obviously it holds that:

$$Y \text{ is solvable} \Leftrightarrow bw(Y) \text{ is solvable.}$$

□

From the three lemmas above we get the undecidability of the following problems:

- $MPCP$,
- PCP,
- PCP over $\{0,1\}$ and
- PCP over X with $|X| \geq 2$.

In particular we have proved the above theorem.

§6 Results on undecidability of context-free languages

Now we can prove the results on undecidability of context-free languages mentioned in Chapter III. In contrast to the regular languages the following holds:

6.1 **Theorem** : For context-free (i.e. CHOMSKY-2-) languages

- the intersection problem,
- the equivalence problem,
- the inclusion problem

are undecidable.

Proof :
Intersection problem: Consider two context-free grammars G_1 and G_2. The question is: Does $L(G_1) \cap L(G_2) = \emptyset$ hold? We show that the Post correspondence problem can be reduced to the intersection problem:

$$PCP \leq \text{ intersection problem}$$

This implies the undecidability of the intersection problem.

Now consider an arbitrary input $Y = ((u_1, v_1), \ldots, (u_n, v_n))$ of the PCP with $u_i, v_i \in X^*$ for an alphabet X. We provide an algorithm which for every input Y constructs two context-free grammars G_1 and G_2, such that the following holds

$$Y \text{ has a correspondence } \iff L(G_1) \cap L(G_2) \neq \emptyset. \qquad (*)$$

The idea is that G_1 generates all strings which can be produced by putting u_i's one after another, and G_2 generates all strings which can be produced by putting v_i's one after another. In order for the necessary relation $(*)$ to hold, G_1 and G_2 must also record the indices i of the placed u_i and v_i. For this purpose we use n new symbols $a_1, \ldots, a_n \notin X$ and choose them as the set of terminal symbols of G_1 and G_2:

$$T = \{a_1, \ldots, a_n\} \cup X$$

Then put $G_i = (\{S\}, T, P_i, S), i = 1, 2$, where P_1 is given by the following productions:

$$S \to a_1 u_1 \,|\, a_1 S u_1 \,|\, \ldots \,|\, a_n u_n \,|\, a_n S u_n$$

P_2 is given by the following productions:

$$S \to a_1 v_1 \,|\, a_1 S v_1 \,|\, \ldots \,|\, a_n v_n \,|\, a_n S v_n$$

Obviously it holds that

$$L(G_1) = \{a_{i_m} \ldots a_{i_1} u_{i_1} \ldots u_{i_m} \,|\, m \geq 1 \text{ and } i_1, \ldots, i_m \in \{1, \ldots, n\}\}$$

and

$$L(G_2) = \{a_{i_m} \dots a_{i_1} v_{i_1} \dots v_{i_m} \mid m \geq 1 \text{ and } i_1, \dots, i_m \in \{1, \dots, n\}\}.$$

This implies that:

$$\begin{aligned} & Y \text{ has the correspondence } (i_1, \dots, i_m) \\ \Leftrightarrow \quad & a_{i_m} \dots a_{i_1} u_{i_1} \dots u_{i_m} = a_{i_m} \dots a_{i_1} v_{i_1} \dots v_{i_m} \in L(G_1) \cap L(G_2). \end{aligned}$$

Therefore, $(*)$ holds and so does the undecidability of the intersection problem. We have proved a stronger result: the undecidability of the intersection problem for *deterministic* context-free languages. As it is easy to check, the languages $L(G_1)$ and $L(G_2)$ are deterministic. □

Equivalence problem: Consider two context-free grammars G_1 and G_2. The question is: Does $L(G_1) = L(G_2)$ hold? We show the undecidability of this problem using the following reduction:

$$\begin{aligned} & \text{Intersection problem for deterministic context-free languages} \\ & \leq \text{ equivalence problem.} \end{aligned}$$

Consider two deterministic push-down automata $\mathcal{K}_1$ and $\mathcal{K}_2$. We show that we can construct from them two (not necessarily deterministic) context-free grammars G_1 and G_2 such that the following holds:

$$L(\mathcal{K}_1) \cap L(\mathcal{K}_2) = \emptyset \iff L(G_1) = L(G_2).$$

We use the fact that for $\mathcal{K}_2$ we can construct the complement push-down automaton $\overline{\mathcal{K}_2}$ with $L(\overline{\mathcal{K}_2}) = \overline{L(\mathcal{K}_2)}$ (compare with Chapter III, section 6). Then from $\mathcal{K}_1$ and $\overline{\mathcal{K}_2}$ we can algorithmically construct context-free grammars G_1 and G_2 with

$$L(G_1) = L(\mathcal{K}_1) \cup L(\overline{\mathcal{K}_2}) \text{ and } L(G_2) = L(\overline{\mathcal{K}_2}).$$

Thus it holds that

$$\begin{aligned} L(\mathcal{K}_1) \cap L(\mathcal{K}_2) = \emptyset \quad & \Leftrightarrow \quad L(\mathcal{K}_1) \subseteq \overline{L(\mathcal{K}_2)} \\ & \Leftrightarrow \quad L(\mathcal{K}_1) \subseteq L(\overline{\mathcal{K}_2}) \\ & \Leftrightarrow \quad L(\mathcal{K}_1) \cup L(\overline{\mathcal{K}_2}) = L(\overline{\mathcal{K}_2}) \\ & \Leftrightarrow \quad L(G_1) = L(G_2) \end{aligned}$$

as required.

Inclusion problem: Consider two context-free grammars G_1 and G_2. The question is: Does $L(G_1) \subseteq L(G_2)$ hold? Obviously the reduction

$$\text{equivalence problem} \leq \text{inclusion problem}$$

holds. Therefore, the inclusion problem is also undecidable. □

Another result of undecidability concerns the ambiguity of context-free grammars. For the practical use of context-free grammars for syntax description of programming languages it would

be beneficial to have an algorithmic ambiguity test. However, we show that such a test does not exist.

6.2 **Theorem** : It is undecidable whether a given context-free grammar is ambiguous.

Proof : We show the following reduction

$$PCP \leq \text{ ambiguity problem.} \qquad (*)$$

Consider an input $Y = ((u_1, v_1), \ldots, (u_n, v_n))$ of the PCP. First we construct the context-free grammar $G_i = (\{S_i\}, T, P_i, S_i), i = 1, 2$, as in the case of reduction of the PCP to the intersection problem. Afterwards we construct a context-free grammar $G = (\{S, S_1, S_2\}, T, P, S)$ from it with

$$P = \{S \to S_1, S \to S_2\} \cup P_1 \cup P_2.$$

Because G_1 and G_2 are unambiguous, the only possible ambiguity of G, while constructing a derivation tree of G for a string $w \in T^*$, is the use of productions $S \to S_1$ or $S \to S_2$ respectively. Therefore, we have the following result:

$$\begin{aligned} G \text{ is ambiguous} &\Leftrightarrow L(G_1) \cap L(G_2) \neq \emptyset \\ &\Leftrightarrow Y \text{ has a correspondence.} \end{aligned}$$

Thus we have shown the reduction $(*)$, and it implies the undecidability of the ambiguity problem. □

Chapter VI

Complexity

§1 Computational complexity

So far we have considered the *computability* of problems, i.e. the question whether the given problems are algorithmically solvable at all. We have also considered a kind of *structural complexity*, i.e. determining which type of machine is necessary for solving problems using algorithms. We have become acquainted with the following hierarchy:

(1) regular languages $\leftrightarrow$ finite automata

(2) context-free languages $\leftrightarrow$ push-down automata

(3) CHOMSKY-0-languages $\leftrightarrow$ Turing machines

Now we will consider the *efficiency* or *computational complexity*, i.e. the question: how much *computing time* and how much *space* (memory) do we need in order to solve the problem algorithmically. We will study time and space depending on the *size of the input*. There exist two working directions:

a) Provide *most efficient algorithms* for concrete problems:

- important for practical problem solving
- Theoretically interesting is the proof of an *upper bound* for the problem: for example, an existing n^3-algorithm proves that the problem is at most n^3 "difficult".

b) Find a *lower bound* for a problem so that *every* algorithm solving this problem has at least this complexity. The size of the input, n, is a trivial lower bound for the time complexity.

Statements about complexity depend on the *machine model*. In theory we mostly consider deterministic or also non-deterministic *Turing machines with several tapes*. By using several

tapes we get more realistic statements than using 1-tape-TM, because the computation time for merely moving back and forth on the tape of TM can be avoided.

1.1 **Definition** : Let $f : \mathbb{N} \to \mathbb{R}$ be a function and τ be a non-deterministic TM with several tapes and input alphabet Σ.

(i) τ has the *time complexity* $f(n)$ if for every string $w \in \Sigma^*$ of length n it holds: τ applied to the input w terminates for every possible computation in at most $f(n)$ steps.

(ii) τ has the *space complexity* $f(n)$ if for every string $w \in \Sigma^*$ of length n it holds: τ applied to the input w uses for every possible computation on every tape at most $f(n)$ cells.

This definition can be also applied to deterministic TMs with several tapes. For deterministic TMs there exists exactly one computation for every input w.

If we deal with TMs, we usually represent problems as languages which should be accepted. We have already used such representation in Chapter "Non-computable functions and undecidable problems". For example, the halting problem for TMs was represented as a language

$$H = \{bw_\tau 00u \in B^* \,|\, \tau \text{ applied to } u \text{ halts}\}.$$

Now we put together problems, i.e. languages, with the same complexity into the so called *complexity classes*.

1.2 **Definition** : Let $f : \mathbb{N} \to \mathbb{R}$.

$\texttt{DTIME}(f(n)) = \{L \,|$ there exists a *deterministic* TM with several tapes which has time complexity $f(n)$ and accepts $L\}$

$\texttt{NTIME}(f(n)) = \{L \,|$ there is a *non-deterministic* TM with several tapes which has time complexity $f(n)$ and accepts $L\}$

$\texttt{DSPACE}(f(n)) = \{L \,|$ there exists a *deterministic* TM with several tapes which has space complexity $f(n)$, and accepts $L\}$

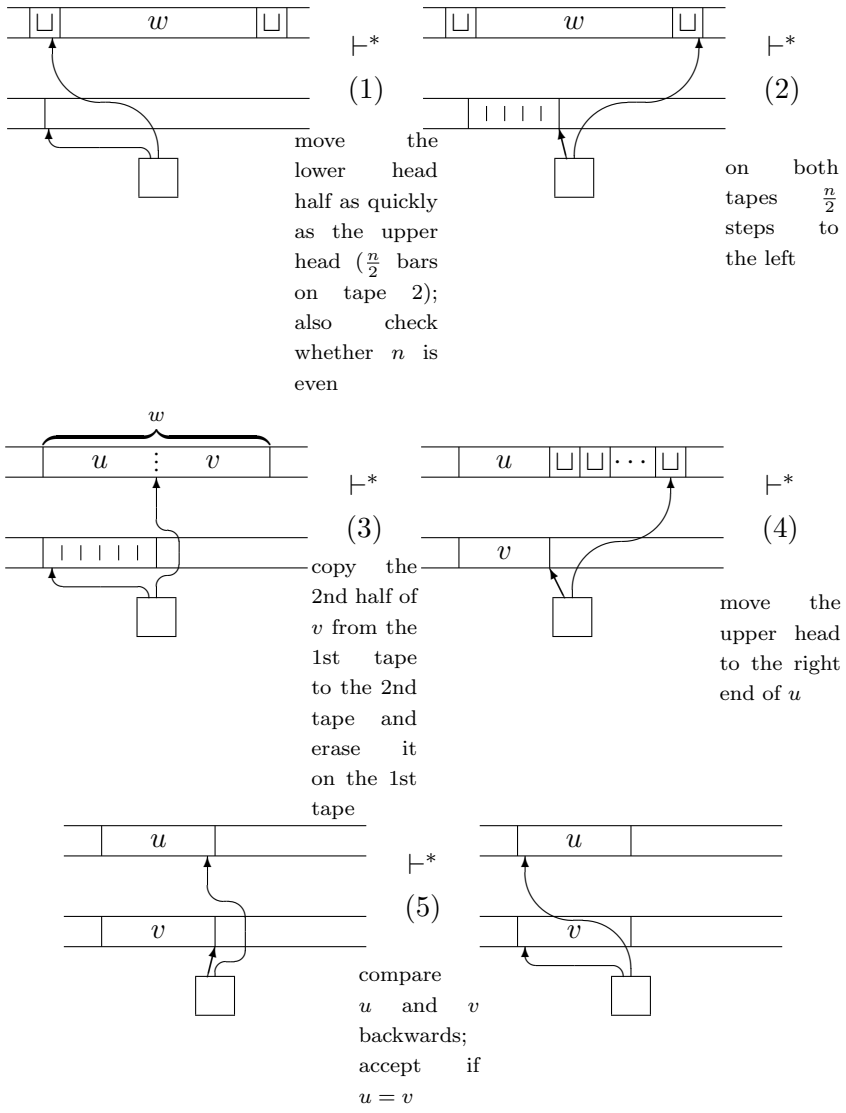
$\texttt{NSPACE}(f(n)) = \{L \,|$ there is a *non-deterministic* TM with several tapes which has space complexity $f(n)$ and accepts $L\}$

Example : Consider $L = \{uu \,|\, u \in \{a, b\}^*\}$. We want to construct a most efficient TM with several tapes which accepts L.

Solution idea:

- For a given string $w \in \{a, b\}^*$ first determine the center of w and then compare both halves.
- In order to determine the center we use the 2nd tape, i.e. we need a deterministic 2-tape TM.

Operating phases: Consider $w \in \{a, b\}^*$ with the length n.



Let us compute time complexity of 5 phases:

$$(n+1) + \left(\frac{n}{2}+1\right) + \left(\frac{n}{2}+1\right) + \left(\frac{n}{2}+1\right) + \left(\frac{n}{2}+1\right) = 3n+5$$

Space complexity: $n+2$

Thus it holds that: $L \in \texttt{DTIME}\ (3n+5)$,
$L \in \texttt{DSPACE}\ (n+2)$.

We can proceed non-determinstically as follows:

Part 1: Copy character by character from tape 1 to tape 2. Stop this process non-deterministically at any time.

Part 2: Return to the beginning on tape 2.

Part 3: Now compare character by character starting from the current position and make sure that the first tape has the same contents as the second one. If it is the case and both heads point to the blank field, then accept.

In the best case this process needs

$$\frac{n}{2} + \left(\frac{n}{2} + 1\right) + \left(\frac{n}{2} + 1\right) = \frac{3n}{2} + 2 \quad \text{steps.}$$

In the worst case, when the non-deterministic decision is made only at the last symbol of the input string, we need

$$n + (n + 1) + 1 = 2n + 2 \quad \text{steps.}$$

Thus it holds that: $L \in \texttt{NTIME}(2n + 2)$

In complexity theory we compare the *asymptotic behavior* of time and space complexity, i.e. the behavior for n which is "large enough". Thus we can omit constant factors. For this purpose we use the *O-notation* from the number theory.

1.3 **Definition** : Let $g : \mathbb{N} \to \mathbb{R}$. Then

$$O(g(n)) = \{f : \mathbb{N} \to \mathbb{R} \mid \exists n_0, k \in \mathbb{N} \, \forall n \geq n_0 : f(n) \leq k \cdot g(n)\}$$

i.e. $O(g(n))$ is the class of all functions f which are *bounded* by a constant multiplied by g for sufficiently large values of n.

Example : $(n \mapsto 3n + 4), (n \mapsto n + 2) \in O(n)$. Therefore, we also say: the language L is of time and space complexity $O(n)$ or L is of *linear* time and space complexity, respectively.

Because a TM can visit at most one new field on its tapes in each computational step, it follows that:

$$\begin{array}{ccccc} & \subseteq & \texttt{DSPACE}(f(n)) & \subseteq & \\ \texttt{DTIME}(f(n)) & & & & \texttt{NSPACE}(f(n)) \\ & \subseteq & \texttt{NTIME}(f(n)) & \subseteq & \end{array}$$

§2 The classes P and NP

The complexity for algorithms which can be applied in practice should be a *polynomial* $p(n)$ of the k-th degree, i.e. of the form

$$p(n) = a_k n^k + \cdots + a_1 n + a_0$$

with $a_i \in \mathbb{N}$ for $i = 0, 1, \ldots, k$, $k \in \mathbb{N}$ and $a_k \neq 0$.

2.1 **Definition (Cobham, 1964)**:

$$\begin{aligned} P &= \bigcup_{p \text{ polynomial in } n} \texttt{DTIME}(p(n)) \\ NP &= \bigcup_{p \text{ polynomial in } n} \texttt{NTIME}(p(n)) \\ PSPACE &= \bigcup_{p \text{ polynomial in } n} \texttt{DSPACE}(p(n)) \\ NPSPACE &= \bigcup_{p \text{ polynomial in } n} \texttt{NSPACE}(p(n)) \end{aligned}$$

2.2 **Theorem (Savitch, 1970)**: For all polynomials p in n it holds that:

$$NSPACE(p(n)) = DSPACE(p^2(n))$$

and thus

$$NPSPACE = PSPACE.$$

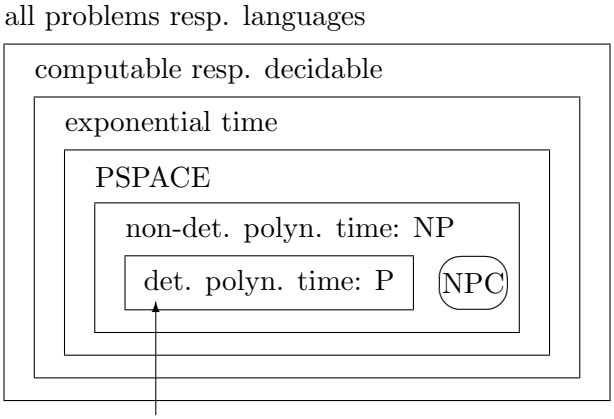
Furthermore, it holds that

$$P \subseteq NP \subseteq PSPACE, \qquad (*)$$

because, as we have already mentioned, a TM can visit at most one new cell in every computational step.

Open problem in computer science: Are the inclusions in $(*)$ strict or does the equality hold ?

The classes P and NP are very important, because they mark the transition from computability or decidability questions which are of practical interest, to the questions which are purely of theoretical interest. This transition can be illustrated by the following diagram:



Practically solvable, i.e. practically computable or decidable are those problems in the class P, which can be characterized as follows:

P: *Construct* the right solution deterministically and in polynomial time.

The polynomials which bound the computation time should have small degree, such as n, n^2 or n^3.

However, practically unsolvable are all problems for which it can be proved that the computation time grows exponentially with the input size n. Between these two extremes there is a large class of practically important problems for which at the moment we know only exponential deterministic algorithms. However, these problems can be solved using *non-deterministic* algorithms in polynomial time. This is the class NP, which in comparison to P can be characterized as follows:

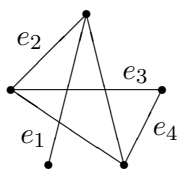
NP: *Guess* a solution proposal *non-deterministically* and then *verify / check* deterministically and in polynomial time whether this proposal is right.

In full generality, these problems are also still practically unsolvable. In practice we make use of so called *heuristics*, which strongly restricts the non-deterministic search space of the possible solution proposals. Using these heuristics we try to approximate an "ideal solution".

Examples of problems from the class NP

(1) **Problem of Hamiltonian path**
Given: A finite graph with n vertices.
Question: Does the graph contain a Hamiltonian path, i.e. a path which visits each vertex exactly once?

Consider, for example, the following graph:



Then the sequence of vertices e_1-e_2-e_3-e_4 is a Hamiltonian path. It is easy to see that the problem of the Hamiltonian path lies in NP: First let us guess a path and then check whether each vertex is visited exactly once. Because there are $n!$ paths in the graph, this method could be used deterministically only with exponential complexity.

(2) **Traveling salesman problem**
Given: A finite graph with n vertices and length $\in \mathbb{N}$ of every edge, as well as a number $k \in \mathbb{N}$.
Question: Is there a round-trip for the traveling salesman with the length $\leq k$, or more formally: Is there a cycle in the graph with the length $\leq k$ which visits each vertex at least once?

This problem also lies in NP: First let us guess a cycle and then compute its length. The traveling salesman problem is of practical importance, for example, for designing telephone networks or integrated circuits.

(3) **Satisfiability problem for Boolean expressions (shortly SAT)**
Given: A Boolean expression B, i.e. an expression which consists only of variables $x_1, x_2, \ldots, x_n$ connected by operators $\neg$ (*not*), $\wedge$ (*and*) and $\vee$ (*or*), as well as by brackets.
Question: Is B satisfiable, i.e. is there an assignment of 0 and 1 to the Boolean variables $x_1, x_2, \ldots, x_n$ in B such that B evaluates to 1?

For example, $B = (x_1 \wedge x_2) \vee \neg x_3$ is satisfiable with the values $x_1 = x_2 = 1$ or $x_3 = 0$. In Section 3 we will consider the SAT problem in more detail. □

While considering the question (which still remains open) whether P = NP holds, a subclass of NP was found, namely the class NPC of the so called NP-*complete* problems. It holds that:

If *one* problem of NPC lies in P, then *all* problems of NP are already in P, i.e. P = NP holds.

The class NPC was introduced in 1971 by S.A. Cook. Cook was the first to prove that the SAT problem, which we have just introduced, is NP-complete. Since 1972 R. Karp has proved that many other problems are also NP-complete. Today we know more than 1000 examples of problems from the class NPC.

Below we want to define the notion of NP-completeness. For this purpose we need the notion of *polynomial-time reduction*, which was introduced by Karp in 1972 as a technique to prove NP-completeness. Therefore, we tighten the notion of reduction $L_1 \leq L_2$, which was introduced in Section 2.

2.3 **Definition** : Let $L_1 \subseteq \Sigma_1^*$ and $L_2 \subseteq \Sigma_2^*$ be languages. Then L_1 is called *polynomially-time reducible to* L_2, shortly

$$L_1 \leq_p L_2,$$

if there is a function $f : \Sigma_1^* \to \Sigma_2^*$ which is total and computable with a polynomial time complexity, such that for all $w \in \Sigma_1^*$ it holds that:

$$w \in L_1 \Leftrightarrow f(w) \in L_2.$$

We also say: $L_1 \leq_p L_2$ *using* f.

Intuitively, $L_1 \leq_p L_2$ indicates that L_1 is not more complex than L_2. We can easily recognize that $\leq_p$ is a reflexive and transitive relation on languages, because with two polynomials $p_1(n)$ and $p_2(n)$, $p_1(p_2(n))$ is also a polynomial.

2.4 **Definition (Cook, 1971)**: A language L_0 is called *NP-complete* if $L_0 \in NP$ holds and $\forall L \in NP : L \leq_p L_0$.

2.5 **Lemma (Polynomial-time reduction)**: Let $L_1 \leq_p L_2$. Then it holds that:

(i) If $L_2 \in P$ holds, then $L_1 \in P$ holds as well.

(ii) If $L_2 \in NP$ holds, then $L_1 \in NP$ holds as well.

(iii) If L_1 is NP-complete and $L_2 \in NP$ holds, then L_2 is also NP-complete.

Proof : *for (i) and (ii):* Let $L_1 \leq_p L_2$ using a function f, which is computed by a Turing machine τ_1. Let the polynomial p_1 bound the computing time of τ_1. Because $L_2 \in P$ (or $L_2 \in NP$, respectively), there is a (non-deterministic) Turing machine τ_2 which is bounded by a polynomial p_2 and computes the characteristic function χ_{L_2}.

Similar to normal reduction, the characteristic function χ_{L_1} for all $w \in \Sigma_1^*$ can be computed as follows:

$$\chi_{L_1}(w) = \chi_{L_2}(f(w)).$$

We do it by sequentially connecting the Turing machines τ_1 and τ_2 . Now let $|w| = n$. Then τ_1 computes the string $f(w)$ in $p_1(n)$ steps. This time bound also limits the length of $f(w)$, i.e. $|f(w)| \leq p_1(n)$. Therefore, the computation of $\chi_{L_2}(f(w))$ is carried out in

$$p_1(n) + p_2(p_1(n))$$

steps. Thus also $L_1 \in P$ (and $L_1 \in NP$, respectively).

for (iii): Let $L \in NP$. Because L_1 is NP-complete, $L \leq_p L_1$ holds. Moreover $L_1 \leq_p L_2$ holds. The transitivity of $\leq_p$ implies $L \leq_p L_2$, what we wanted to show. □

2.6 **Corollary (Karp)**: Let L_0 be an NP-complete language. Then it holds that: $L_0 \in P \Leftrightarrow P = NP$.

Proof : "⇒": Statement (i) of the lemma.
"⇐": is obvious. □

Example : We show the following polynomial-time reduction:

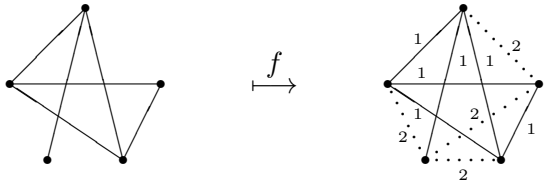
$$\text{Hamiltonian path } \leq_p \text{ traveling salesman.}$$

Consider a graph G with n vertices. As a reduction function we consider the following construction $f : G \mapsto G'$ of a new graph G' as follows:

- G' has n vertices as G.
- Every edge of G is the edge of G' of the length 1.
- Every "non-edge" of G is the edge of G' of the length 2.

Thus G' is a complete graph, i.e. every two vertices are connected by one edge.

E.g. G: $n = 5$ $\xrightarrow{f}$ G': and appropriate bound $k = n + 1$



The construction f takes place in polynomial time (in order to find the non-edges: consider an incidence matrix, i.e. $O(n^2)$). Now we show the reduction property of f, i.e.

$$G \text{ has a Hamiltonian path} \Leftrightarrow G' \text{ has a round-trip of the length} \leq n + 1.$$

Proof of "$\Rightarrow$": We need $n - 1$ edges of the length 1 (i.e. "from G") to connect n different vertices. In order to close this Hamiltonian path, we need another edge of the length ≤ 2. All in all the constructed round-trip has the length $\leq n + 1$.

Proof of "$\Leftarrow$": The round-trip has at least n edges (to connect n vertices) and at most $n + 1$ edges (due to the edge length ≥ 1).

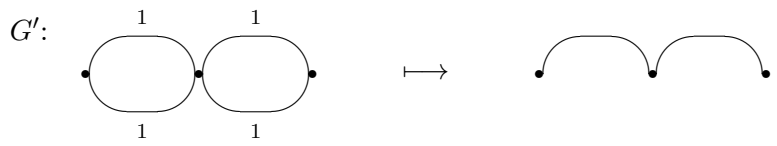
During the round-trip at least 1 vertex is visited twice (start = end). Therefore, we must *definitely remove one edge*, in order to get a path with different vertices. We check whether it is enough.

Case 1: After removing one edge, the remaining path has the length $n - 1$.

Thus the vertices which can be visited are all different. Therefore, we have found a Hamiltonian path.
(Example for case 1: see above)

Case 2: Otherwise, after removing one edge, the remaining path has the length n.

Then this path has n edges and one vertex is visited twice on a remaining cycle. Then we get a Hamiltonian path after having removed one more edge.
Example for case 2: $n = 3$



Round-trip of the length 4 Hamiltonian path

§3 The satisfiability problem for Boolean expressions

3.1 **Definition SAT**:

(i) A *Boolean expression* is a term over {0,1} and variables with the operations `not, and, or`. In particular: Let $V = \{x_1, x_2, \ldots\}$ be an infinite set of variables. Let $B = \{0, 1\}$ be the set of logical values ("false" and "true").

(1) Every $x_i \in V$ is a Boolean expression.

(2) Every $a \in B$ is a Boolean expression.

(3) If E is a Boolean expression, then so is $\neg E$ ("`not` E").

(4) If E_1, E_2 are Boolean expressions, then so are $(E_1 \vee E_2)$ and $(E_1 \wedge E_2)$.

If $x \in V$, then we mostly write $\overline{x}$ instead of $\neg x$.
If $x \in V$, then we call x and $\overline{x}$ *literals*.

$\left(\begin{array}{l}\text{Priorities:}\\ \neg \text{ over } \wedge \text{ over } \vee\end{array}\right)$

(ii) Conjunctive norman form:

(1) If $y_1, y_2, \ldots, y_k$ are literals, then $(y_1 \vee y_2 \vee \ldots \vee y_k)$ is called *clause* (of the order k, i.e. k literals/alternatives)

(2) If $c_1, c_2, \ldots, c_r$ are clauses (of the order $\leq k$), then $c_1 \wedge c_2 \wedge \ldots \wedge c_r$ is called a Boolean expression in *conjunctive normal form* ("CNF") (of the order $\leq k$). If at least one clause contains k literals, then this expression is called a CNF of the order k. Similarly, we can define "disjoint normal form".

(iii) An *assignment* β is a function $\beta : V \to \{0, 1\}$ which can be extended to Boolean expressions (see (i),(2)-(4)):

$\beta(0) = 0, \beta(1) = 1,$
$\beta(\neg E) = 1 - \beta(E),$
$\beta(E_1 \wedge E_2) = \mathit{Min}(\beta(E_1), \beta(E_2)),$
$\beta(E_1 \vee E_2) = \mathit{Max}(\beta(E_1), \beta(E_2)).$

(iv) A Boolean expression E is called *satisfiable* if there is an assignment β with $\beta(E) = 1$.

(v) The satisfiability problem is described using the language

$$\text{GSAT} = \{E \mid E \text{ is Boolean expression, } E \text{ is satisfiable}\}.$$

In this case the alphabet is $V \cup \{0, 1, \neg, \vee, \wedge,), (\ \}$. It is infinite. Thus we encode V using:

$$x_i \mapsto xj, \quad \text{where } j \text{ is the binary representation of the number } i,$$

i.e. $x_1 \mapsto x1, x_2 \mapsto x10, x_3 \mapsto x11, x_4 \mapsto x100$ etc. The elements from B can be easily removed from E using calculation rules, and we can bring E to CNF. Thus we get:

$$\begin{aligned}\text{SAT} := \{E \mid\ & E \text{ is a Boolean expression (without elements from } B) \text{ in CNF;}\\ & \text{if } E \text{ contains } m \text{ different variables, then these are}\\ & \text{the variables } x_1, \ldots, x_m \text{ (in encoded form);}\\ & E \text{ is satisfiable}\}\\ \subset\ & \{x, 0, 1, \neg, \wedge, \vee, (,)\ \}^*.\end{aligned}$$

$\text{SAT}(k) := \{E \mid E \in \text{SAT} \text{ and } E \text{ is of the order } k\}$.

3.2 **Theorem (Cook, 1971)**: SAT is NP-complete.

Proof :

- First of all it is easy to see that SAT lies in NP:
  Let us *guess* an assignment $\beta : \{x_1, \ldots, x_m\} \to \{0, 1\}$ for a Boolean expression E in CNF, which contains exactly m variables. Then to every variable x_i we assign the value $\beta(x_i)$ and calculate $\beta(E)$ according to the standard calculation rules (Def. 3.1(iii)). If $|E| = n$ held, then E would have less than n variables, i.e. $m \leq n$; the guessing of an assignment β is carried out in linear time (create a table, m steps), the substitution in E takes $|E| \cdot const$ steps as well as the evaluation, i.e. $\text{SAT} \in \text{NTIME}\ (c_1 n + c_2)$ for appropriate constants c_1 and c_2. In particular $\text{SAT} \in \text{NP}$ (think yourself how we can compute $\beta(E)$ in linear time).

- The difficult part is to show that for every $L \in \text{NP}$ it holds that $L \leq \text{SAT}$. Thus consider an arbitrary $L \in \text{NP}$. Then there is a non-deterministic Turing machine $\tau = (Q, X, \Gamma, \delta, q_0, \sqcup, F)$ with: Q = set of states; X = input alphabet; Γ = tape alphabet, which contains X; $q_0 \in Q$ initial state; $\sqcup \in \Gamma \setminus X$ blank symbol; $F \subset Q$ set of final states (compare with the definition in chapter V, supplemented with the final states, because τ must halt for all inputs; only those strings $w \in X^*$ belong to L which bring τ into a final state). τ accepts L in polynomial time, i.e. there exists a polynomial p such that τ halts for every $w \in X^*$ and every computation sequence after at most $p(n)$ (with $n = |w|$) steps, and $w \in L$ if and only if there is a computation sequence of τ which transfers the initial configuration $q_0 w$ into a final configuration $u_1 q u_2$ with $q \in F$.

  We assume without loss of generality that τ has only *one tape*. Now for every $w \in X^*$ we construct a Boolean expression $g(w) \in \Sigma^*$ (with $\Sigma = \{x, 0, 1, \neg, \wedge, \vee, (,)\ \}$) in CNF such that it holds that $w \in L \iff g(w) \in \text{SAT}$. If we have shown that this function $g : X^* \to \Sigma^*$ is a polynomial reduction, then it follows that SAT is NP-complete, because L from NP was arbitrary.

  Construction of $g(w)$ from w: (We follow [Mehlhorn, "Data Structures and Algorithms" 2, Section VI.4]) Let $Q = \{q_1, \ldots, q_s\}, q_1$ = initial state, $F = \{q_r, q_{r+1}, \ldots, q_s\}$. Let $\Gamma = \{c_1, \ldots, c_\gamma\}$ with $c_1 = \sqcup$ = blank symbol.
  Without loss of generality let $\delta : Q \times \Gamma \to 2^{Q \times \Gamma \times \{\underset{\text{L}}{-1}, \underset{\text{R}}{+1}, \underset{\text{S}}{0}\}}$ with $\delta(q, c) = \emptyset$ for $q \in F$.

  "Left" "Right" "Stay"

  We make δ complete by changing every $\delta(q, c) = \emptyset$ into $\delta(q, c) = \{(q, c, 0)\}$. Thus the empty

set $\emptyset$ never appears as an image of δ. Because the end of the computation was determined by $\delta(q, c) = \emptyset$, by the modification we define a non-terminating Turing machine τ', which, however, exactly reflects the operation of τ: If τ halts in state q, then τ' after making $p(|w|)$ steps will reach a configuration $u_1 q u_2$ with exactly this state q and τ' will stay in this configuration indefinitely; the opposite direction also holds. Thus:

$w \in L \iff \tau$ applied to w is in a configuration $u_1 q u_2$ after $p(|w|)$ steps with $q \in F$
$\iff \tau'$ runs through a sequence $K_1, K_2, \ldots, K_{p(n)}$ of configurations with

(i) $K_1 = q_1 w$ initial configuration.
(ii) K_{i+1} is a successor configuration of K_i for all $i \geq 1$.
(iii) $n = |w|$ and $K_{p(n)}$ contains a final state $q \in F$.

By δ we usually mean the subset of $Q \times \Gamma \times Q \times \Gamma \times \{-1, +1, 0\}$.
Let $\delta = \{tuple_1, tuple_2, \ldots, tuple_m\}$ be numbered from 1 to m.

Let $w \in X^*$ be given, $|w| = n, w = c_{j_1} c_{j_2} \ldots c_{j_n}$. The formula $g(w)$ is constructed by the following variables:

		meaning of the variables:	
$z_{t,k}$	$1 \leq t \leq p(n)$ $1 \leq k \leq s$	$z_{t,k} = 1$	$\iff$ τ' is in state q_k at the time point t
$a_{t,i,j}$	$1 \leq t \leq p(n)$ $-p(n) \leq i \leq p(n)$ $1 \leq j \leq \gamma$	$a_{t,i,j} = 1$	$\iff$ c_j is the content of the cell i at the time point t
$s_{t,i}$	$1 \leq t \leq p(n)$ $-p(n) \leq i \leq p(n)$	$s_{t,i} = 1$	$\iff$ τ' is located at the cell i at the time point t
$b_{t,l}$	$1 \leq t \leq p(n)$ $1 \leq l \leq m$	$b_{t,l} = 1$	$\iff$ the l-th tuple of δ is used to transfer from the time point t to the time point $t+1$.

Because τ makes at most $p(n)$ steps, we can always assume that $|i| \leq p(n)$ and $t \leq p(n)$. The Boolean expression $g(w)$ should exactly describe the above sequence of configurations $K_1, K_2, \ldots, K_{p(n)}$ (or all such sequences of configurations, respectively). This requires fulfilling the following conditions:

(1) Initial configuration: τ' in state $q_1, \sqcup^{p(n)+1} w \sqcup^{p(n)-n}$ is on the tape, the head is at the cell 1. Time point $t = 1$.

(2) Final configuration (provided that w is accepted): τ' in state q_j with $r \leq j \leq s$ at the time point $t = p(n)$.

(3) Transition condition: τ' at every time point $1 \leq t \leq p(n)$ is in exactly one state, every cell from $-p(n)$ to $+p(n)$ contains exactly one symbol from Γ, the head is exactly at one of these cells, and exactly one tuple of δ is used for the transition.

(4) Successor configuration: The next configuration follows from the previous configuration from the transition determined by the tuple of δ described in (3).

Let $g(w) = A_1 \wedge A_2 \wedge A_3 \wedge A_4$ with:

for(1): $A_1 = a_{1,-p(n),1} \wedge a_{1,-p(n)+1,1} \wedge \ldots \wedge a_{1,0,1}$ $\left(\begin{array}{c}\sqcup^{p(n)+1}\\ \text{to the left of } w\end{array}\right)$

$\wedge\ a_{1,1,j_1} \wedge a_{1,2,j_2} \wedge \ldots \wedge a_{1,n,j_n}$ $(w = c_{j_1} \ldots c_{j_n})$

$\wedge\ a_{1,n+1,1} \wedge a_{1,n+2,1} \wedge \ldots \wedge a_{1,p(n),1}$ $\left(\begin{array}{c}\sqcup^{p(n)-n}\\ \text{to the right of } w\end{array}\right)$

$\wedge\ z_{1,1} \wedge s_{1,1}$

This formula describes exactly the initial configuration with $2 \cdot p(n) + 3$ variables.

for(2): $A_2 = z_{p(n),r} \vee z_{p(n),r+1} \vee \ldots \vee z_{p(n),s}$ ($|F|$ variables).

for(3): First we describe an auxiliary expression: For variables $x_1, \ldots, x_k$ let

$$\mathit{exactlyone}(x_1, \ldots, x_k) := \mathit{atleastone}(x_1, \ldots, x_k) \wedge \mathit{atmostone}(x_1, \ldots, x_k)$$

with

$$\mathit{atleastone}(x_1, \ldots, x_k) := (x_1 \vee x_2 \vee \ldots \vee x_k)$$

and

$$\begin{aligned}\mathit{atmostone}(x_1, \ldots, x_k) &:= \neg\, \mathit{atleasttwo}(x_1, \ldots, x_k)\\ &= \neg \bigvee_{1 \le i < j \le k} (x_i \wedge x_j)\\ &= \bigwedge_{1 \le i < j \le k} (\overline{x_i} \vee \overline{x_j}) \qquad \left(\begin{array}{c}\text{apply}\\ \text{de Morgan's law!}\end{array}\right)\end{aligned}$$

Thus the expression $\mathit{exactlyone}(x_1, \ldots, x_k)$ is in CNF; it has $k + \frac{1}{2} \cdot k \cdot (k-1) \cdot 2 = k^2$ variables. It will be 1 if exactly one x_i has the value 1.

Now put

$$A_3 = \bigwedge_{1 \le t \le p(n)} \left(A_3^{\mathit{state}}(t) \wedge A_3^{\mathit{place}}(t) \wedge A_3^{\mathit{cell}}(t) \wedge A_3^{\mathit{next}}(t) \right)$$

with

$$A_3^{\mathit{state}}(t) = \mathit{exactlyone}(z_{t,1}, \ldots, z_{t,s})$$

$$A_3^{\mathit{place}}(t) = \mathit{exactlyone}(s_{t,-p(n)}, s_{t,-p(n)+1}, \ldots, s_{t,p(n)})$$

$$A_3^{\mathit{cell}}(t) = \bigwedge_{-p(n) \le i \le p(n)} \mathit{exactlyone}(a_{t,i,1}, \ldots, a_{t,i,\gamma})$$

$$A_3^{\mathit{next}}(t) = \mathit{exactlyone}(b_{t,1}, \ldots, b_{t,m})$$

Again, this formula is in CNF.

A_3 has $p(n) \cdot (s^2 + (2 \cdot p(n) + 1)^2 + (2 \cdot p(n) + 1)^2 \cdot \gamma^2 + m^2)$ variables.

A_3 describes exactly the transition condition (3).

for(4): $A_4 = \bigwedge_{1 \le t < p(n)} A_4(t)$

Now we must consider the tuple of δ in more detail. For $l = 1, 2, \ldots, m$ let the l-th tuple of δ be given by

$$\begin{array}{lccccccc} tupel_l = (& q_{k_l}, & c_{j_l}, & q_{\tilde{k}_l}, & c_{\tilde{j}_l}, & & d_l &) \\ & \in & \in & \in & \in & & \in & \\ & Q & \Gamma & Q & \Gamma & & \{+1, -1, 0\} & \end{array}$$

Put:

$$A_4(t) = \bigwedge_{-p(n) \le i \le p(n)} \left[\bigwedge_{1 \le j \le \gamma} (s_{t,i} \vee \overline{a}_{t,i,j} \vee a_{t+1,i,j}) \right.$$

$$\wedge \bigwedge_{1 \le l \le m} \Big((\overline{s}_{t,i} \vee \overline{b}_{t,l} \vee z_{t,k_l}) \wedge (\overline{s}_{t,i} \vee \overline{b}_{t,l} \vee a_{t,i,j_l})$$
$$\wedge (\overline{s}_{t,i} \vee \overline{b}_{t,l} \vee z_{t+1,\tilde{k}_l}) \wedge (\overline{s}_{t,i} \vee \overline{b}_{t,l} \vee a_{t+1,i,\tilde{j}_l})$$
$$\left. \wedge (\overline{s}_{t,i} \vee \overline{b}_{t,l} \vee s_{t+1,i+d_l}) \Big) \right]$$

Comment: $(s_{t,i} \vee \overline{a}_{t,i,j} \vee a_{t+1,i,j})$ is 1

$\iff$ $s_{t,i}$ is 1 (i.e. τ' is at the time point t at the cell i)
or $s_{t,i} = 0$ and c_j is not at the cell i at the time point t
or $s_{t,i} = 0$ and $a_{t,i,j} = 1$ and $a_{t+1,i,j} = 1$

$\iff$ $s_{t,i} = 0$ and $a_{t,i,j} = 1$ implies that $a_{t+1,i,j} = 1$
(i.e. the cells which are not considered are not changed by τ'!)

In a similar way we read the remaining clauses:

$(\overline{s}_{t,i} \vee \overline{b}_{t,l} \vee z_{t,k_l})$ is 1 $\iff$

if τ' was at the time point t at the cell i and the l-th tuple was chosen for transition, then τ' must have been in state q_{k_l} (at the time point t).

A_4 is in CNF. A_4 contains $p(n) \cdot (2 \cdot p(n) + 1) \cdot (3 \cdot \gamma + 15 \cdot m)$ variables. Now it is easy to show: ($g(w)$ is in CNF!)

Statement 1: g is polynomial.

Obvious; the above construction gives us the formula with

$$\begin{aligned} (2 \cdot p(n) + 3) + (s - r + 1) + p(n) \cdot (s^2 + (2p(n) + 1)^2 \cdot (\gamma^2 + 1) + m^2) \\ + p(n) \cdot (2 \cdot p(n) + 1) \cdot (3\gamma + 15m) = p'(n) \end{aligned}$$

variables (for this polynomial p'). It is obvious that the generation of $g(w)$ from w is proportional to $p'(n)$, i.e. should be computed on a deterministic 1-tape Turing machine in at most $const \cdot (p'(n))^2$ steps, i.e. in polynomial time.

Statement 2: g is a reduction, i.e.

$$\forall w \in X^* \text{ it holds: } (w \in L \iff g(w) \in SAT).$$

This expression follows from the construction, where "$\Leftarrow$" has to be proved in a more complex way.

Thus g is a polynomial reduction. $\Rightarrow$ SAT is NP-complete.

□

(End of the proof of theorem 3.2) □

3.3 **Theorem** : SAT(3) is NP-complete.

Proof : Because SAT $\in$ NP, it also holds that SAT(3) $\in$ NP. Let us reduce SAT to SAT(3). Let us replace every clause $(x_1 \vee x_2 \vee \ldots \vee x_r)$ by

$$(x_1 \vee \overline{y_1}) \wedge (y_1 \vee x_2 \vee \overline{y_2}) \wedge (y_2 \vee x_3 \vee \overline{y_3}) \wedge \ldots \wedge (y_{r-1} \vee x_r \vee \overline{y_r}) \wedge y_r$$

with new variables $y_1, y_2, \ldots, y_r$. It is easy to see that $(x_1 \vee \ldots \vee x_r)$ is satisfiable if and only if the longer formula (of the order 3) is satisfiable. The process is of polynomial size $\Rightarrow$ SAT can be polynomially-time reduced to SAT(3). □

For further NP-complete problems: see literature (rucksack problem, shortest round-trip, Hamiltonian paths in graphs, clique problem, coloring problem of graphs, integer programming, timetable problem, allocation problem, graph embedding problem, etc.).