Automata and formal languages

Jarkko Kari

Fall semester 2024

University of Turku

Contents

1 Preliminaries

1.1 Introduction

Theoretical computer science studies the mathematical foundation of computation. It investigates the power and limitations of computing devices. In order to be able to use rigorous mathematical proofs, abstract mathematical models of computers are needed. Models should be as simple as possible so that they can be easily analyzed; yet they have to be powerful enough to be able to perform relevant computation processes.

The reason behind the success of theoretical computer science is the fact that real computers can be modeled with very simple abstract machines. The models ignore the implementation details of individual computers and concentrate on the actual computation process. In the last part of the course we investigate one such model — called a Turing machine — and using it we are even able to prove that some computational problems can not be algorithmically solved, i.e., they are undecidable.

But before reaching the concept of Turing machines we investigate some weaker models of computation. We start with the most restricted models, called finite automata. Then we move up to intermediate level by introducing pushdown automata. We analyze the computation power of different models, and study their limitations.

All our abstract machines manipulate strings of symbols. **Formal languages** are basic mathematical objects used throughout this course. The power of any computation model will be determined by analyzing how complex formal languages it can describe. Finite automata are able to define only very simple languages, called regular languages. Pushdown automata describe more complicated context-free languages. Full-powered computers like Turing machines define recursive and recursively enumerable languages. All these language types will be extensively studied in this course.

1.2 Alphabets, words and languages

Let us start with basic notions. An **alphabet** is a finite, non-empty set. The elements of the alphabet are called **letters**. The choice of the alphabet depends on the application in mind. In our examples we often use letters of the English alphabet, or digits, or other characters found on computer keyboards. But any other symbols could be used as well. Here are some examples of alphabets:

$$\begin{aligned} \Sigma_1 &= \{a, b\}, \\ \Sigma_2 &= \{0, 1, 2\}, \\ \Sigma_3 &= \{\clubsuit, \diamondsuit, \heartsuit, \spadesuit\} \end{aligned}$$

A **word** (or **string**) is a finite sequence of letters. For example, $abaab$, aaa and b are words over the alphabet $\Sigma_1 = \{a, b\}$. Variables with names u, v, w, x, y, z will typically be used to represent words.

If w is a word then $|w|$ denotes its **length**, i.e. the number of symbols in it. Note that the length of a word can be 0. Such word is called the **empty word**, and it is denoted by ε. (We cannot just write nothing: no one would know that the empty word is there!)

For example,

$$\begin{aligned} |abaab| &= 5 \\ |\clubsuit\heartsuit\heartsuit| &= 3 \\ |\varepsilon| &= 0 \end{aligned}$$

The **concatenation** of two words is the word obtained by writing the first word followed by the second one as a single word. For example, the concatenation of *data* and *base* is the word *database*. Notation for concatenation is similar to normal multiplication: For example,

$$ab \cdot aab = abaab.$$

The multiplication sign does not need to be written if the meaning is clear, i.e., uv is the concatenation of words u and v. So, for example, if $v = a$ and $w = ab$, then $vw = v \cdot w = aab$.

The empty word ε is the identity element of concatenation, much the same way as number 1 is the identity element of multiplication: For any word w

$$w\varepsilon = \varepsilon w = w.$$

A concatenation of a word with itself is denoted the same way as the multiplication of a number with itself: For any integer n and word w the word w^n is the concatenation of n copies of w. For example, if $w = abba$ then

$$\begin{aligned} w^2 &= abbaabba, & a^5 &= aaaaa, & \varepsilon^3 &= \varepsilon, \\ ab^2a^3b &= abbaaab, & w^0 &= \varepsilon, & \varepsilon^0 &= \varepsilon. \end{aligned}$$

We may use parentheses to group and to indicate the order of operations, exactly as we do when multiplying numbers:

$$a[(ab)^3aa]^2b = aababababaaababababaab.$$

An important difference between concatenation and multiplication is that concatenation is not commutative. (For example, $ab \neq ba$.) However, concatenation is **associative**: for all words u, v, w holds

$$(uv)w = u(vw).$$

A **prefix** of a word is any sequence of leading symbols of the word. For example, word *abaab* has 6 prefixes:

$$\varepsilon, a, ab, aba, abaa \text{ and } abaab.$$

A **suffix** of a word is any sequence of trailing symbols of the word.

A **subword** of a word is any sequence of consecutive symbols that appears in the word. Subwords of *abaab* are:

$$\varepsilon, a, b, ab, ba, aa, aba, baa, aab, abaa, baab, abaab.$$

Subwords of a word are also called its **factors**.

A prefix, suffix or subword of a word is called **proper** if it is not the word itself. Each word w has $|w|$ different proper prefixes and suffixes.

The **mirror image** of a word is the word obtained by reversing the order of its letters. The mirror image of word w is denoted by w^R. For example,

$$\begin{aligned}(abaab)^R &= baaba \\ (saippuakauppias)^R &= saippuakauppias \\ \varepsilon^R &= \varepsilon.\end{aligned}$$

A word w whose mirror image is the word itself is called a **palindrome**. In other words, word w is a palindrome iff $w = w^R$.

A formal **language** is a set of words over a fixed alphabet. The language is **finite** if it contains only a finite number of words. We are mainly interested in **infinite** languages. Here are a few examples of languages over the alphabet $\Sigma_1 = \{a, b\}$:

$\{a, ab, abb\}$
$\{a, aa, aaa, aaaa, \ldots\} = \{a^n \mid n \geq 1\}$
$\{a^n b^n \mid n \geq 0\}$
$\{w \mid w = w^R\} = \{w \mid \ w \text{ is a palindrome } \}$
$\{a^p \mid \ p \text{ is a prime number } \}$
$\{\varepsilon\}$
$\emptyset$

Note that ε is a word, $\{\varepsilon\}$ is a language containing one element (the empty word), and $\emptyset$ is a language containing no words. They are all different. (Later, in Section 2.4, we also introduce the regular expression ε.)

The language of all words over alphabet Σ is denoted by Σ^*. For example,

$$\begin{aligned}\{a, b\}^* &= \{\varepsilon, a, b, aa, ab, ba, bb, aaa, \ldots\} \\ \{\heartsuit\}^* &= \{\varepsilon, \heartsuit, \heartsuit^2, \heartsuit^3, \heartsuit^4, \ldots\}\end{aligned}$$

The operation "$\cdot$" (concatenation of words) is associative and $\varepsilon \in \Sigma^*$ is its identity element, so $(\Sigma^*, \cdot)$ is a mathematical structure called a **monoid**. More precisely, it is the **free monoid** generated by Σ. Because $|uv| = |u| + |v|$, the length function $w \mapsto |w|$ is a monoid homomorphism from $(\Sigma^*, \cdot)$ to the additive monoid $(\mathbb{N}, +)$ of natural numbers.

The language of all non-empty words over Σ is denoted by Σ^+, so

$$\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}.$$

Note that $(\Sigma^+, \cdot)$ is the free **semigroup** generated by Σ.

A problem with infinite languages is how to describe them. We cannot just list the words as we do in the case of finite languages. Formal language theory develops techniques for specifying (infinite) languages. How simple or complex a language is depends on the "simplicity" of its description.

2 Regular languages

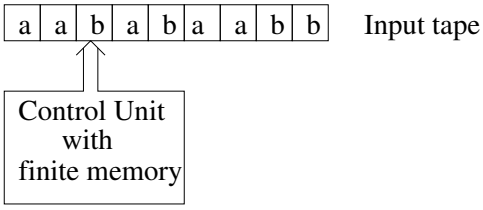
Regular (or rational) languages form a simple but important family of formal languages. They can be specified in several equivalent ways: using deterministic or non-deterministic finite automata, left (or right) linear grammars, or regular expressions. We start with deterministic finite automata.

A linguistic note: *automaton* and *automata* are the singular and the plural form of the same word, respectively.

2.1 Deterministic Finite Automata (DFA)

DFA provide a simple way of describing languages. DFA are accepting devices: one gives a word as input and after a while the DFA tells whether the input word is in the language (DFA **accepts** the word) or whether it is not in the language (DFA **rejects** it).

To decide whether to accept or reject the input word the DFA scans the letters of the word from left to right. The DFA has a finite internal memory available. At each input letter the state of the internal memory is changed depending on the letter scanned. The previous memory state and the input letter together determine what the next state of the memory is. The word is accepted if the internal memory is in an accepting state after scanning the entire word.



Let us be precise: A DFA $A = (Q, \Sigma, \delta, q_0, F)$ is specified by 5 items:

- Finite **state set** Q. At all times the internal memory is in some state $q \in Q$.
- **Input alphabet** Σ. The machine only operates on words over the alphabet Σ.
- **Transition function** δ. The transition function describes how the machine changes its internal state. It is a function
$$\delta : Q \times \Sigma \longrightarrow Q$$
from (state, input letter) -pairs to states. If the machine is in state q and the present input letter is a then the machine changes its internal state to $\delta(q, a)$ and moves to the next input letter.
- **Initial state** $q_0 \in Q$ is the internal state of the machine before any letters have been read.
- Set $F \subseteq Q$ of **final states** specifies which states are accepting and which are rejecting. If the internal state of the machine, after reading the whole input, is some state of F then the word is accepted, otherwise rejected.

The language **recognized** (or **accepted**) by a DFA A consists of all words that A accepts. This language is denoted by $L(A)$.

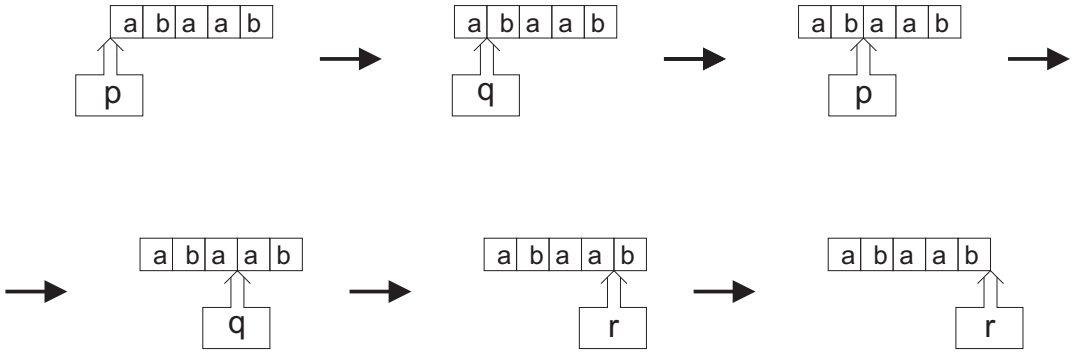
Example 1. For example, consider the DFA

$$A = (\{p, q, r\}, \{a, b\}, \delta, p, \{r\})$$

where the transition function δ is given by the table

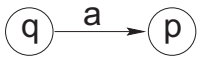
	a	b
p	q	p
q	r	p
r	r	r

The operation of the machine on input word $w = abaab$ is as follows:



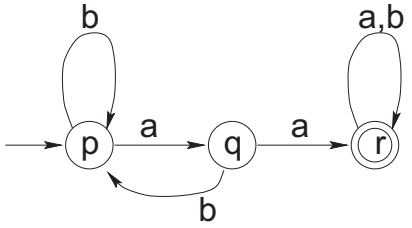
The word *abaab* is accepted because r is a final state. □

A convenient way of displaying DFA is to use a **transition diagram**. It is a labeled directed graph whose vertices represent different states of Q, and whose edges indicate the transitions with different input symbols. The edges are labeled with the input letters and the vertices are labeled with states. Transition $\delta(q, a) = p$ is represented by an arc labeled a going from vertex q into vertex p:



Final states are indicated as double circles, and the initial state is indicated by a short incoming arrow.

Example 2. Here's the transition diagram for the DFA A of Example 1



To determine whether a given word is accepted by A one just follows the path labeled with the input letters, starting from the initial state. If the state where the path ends is a final state, the word is accepted. Otherwise it is rejected. In our example, path labeled with input word $w = abaab$ leads to state r so the word is accepted. Input word *abba* is rejected since it leads to q which is not a final state. □

A DFA A is a finite description of the language $L(A)$. For example, the DFA in Examples 1 and 2 represents the language of all words over the alphabet $\{a, b\}$ that contain aa as a subword.

Example 3. Let us draw transition diagrams for DFA that recognize the following languages over the alphabet $\{a, b\}$. Note that the DFA has to recognize the language exactly: All words of the language have to be accepted; all words not in the language have to be rejected.

1. Words that end in *ab*.

2. Words with an odd number of a's.

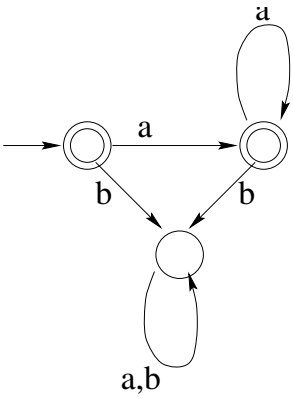
3. Words that contain aba as a subword.

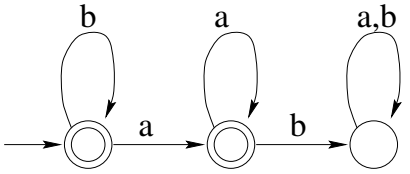
4. Words that start with a and end in a.

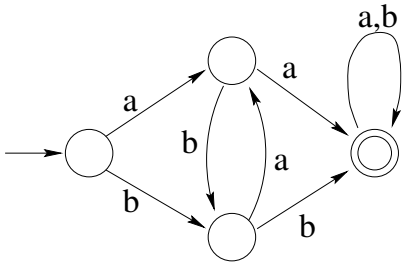
5. The finite language $\{\varepsilon, a, b\}$.

6. All words over $\{a, b\}$, i.e. $\{a, b\}^*$.

Conversely, let us determine (and describe in English) the languages recognized by the following DFA:







□

Not all languages can be defined by a DFA. For example, it is impossible to build a DFA that would accept the language

$$\{a^p \mid \ p \text{ is a prime number } \}.$$

We'll prove later that even simple languages such as

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

cannot be recognized by any DFA. Languages that can be recognized by DFA are called **regular**.

To enable exact mathematical notations, we extend the meaning of the transition function δ. The basic transition function δ gives the new state of the machine after a single letter is read. The extended function (that we will first denote by $\hat{\delta}$) gives the new state after an arbitrary string of letters is read. In other words, $\hat{\delta}$ is a function

$$\hat{\delta} : Q \times \Sigma^* \longrightarrow Q.$$

For every state q and word w the value of $\hat{\delta}(q, w)$ is the state that the DFA reaches if in state q it reads the input word w.

Formally the extended function is defined recursively as follows:

1. $\hat{\delta}(q, \varepsilon) = q$ for every state q. In other words, the machine does not change its state if no input letters are consumed.

2. For all words w and letters a
$$\hat{\delta}(q, wa) = \delta(\hat{\delta}(q, w), a). \tag{1}$$
In other words, if $p = \hat{\delta}(q, w)$ is the state after reading input w then $\delta(p, a)$ is the new state after reading input wa.

Note that $\hat{\delta}$ is an extension of δ. They give same value for input words $w = a$ that contain only one letter:

$$\hat{\delta}(q, a) = \delta(q, a).$$

Therefore there is no danger of confusion if we simplify notations by removing the hat and indicate simply δ instead of $\hat{\delta}$ from now on.

Example 4. Consider our sample three state DFA from Examples 1 and 2. In this DFA

$$\begin{aligned} \delta(r, ab) &= r, \\ \delta(q, bb) &= p, \\ \delta(p, abaab) &= r. \end{aligned}$$

□

The language recognized by DFA $A = (Q, \Sigma, \delta, q_0, F)$ can now be precisely formulated as follows:

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

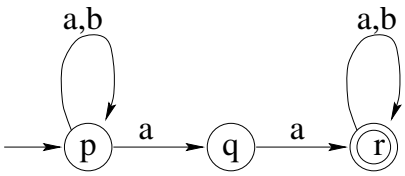
This is a mathematical shorthand for the set of words w over the alphabet Σ such that, if the machine reads input w in the initial state q_0, then the state it reaches is a final state.

Remark: DFA according to our definition are often called a **complete DFA** because each state must have an outgoing transition with each letter in the alphabet. A commonly used alternative is to allow partial transition functions δ, that do not need to be defined for all pairs of states and input letters. In such cases, if there is no transitions from the current state with the current input letter the machine halts and the word is rejected. However, exactly the same languages can be recognized using complete or partial transition functions, so we stick to the terminology that the transition function δ of a DFA is a complete function.

2.2 Nondeterministic finite automata (NFA)

Nondeterministic finite automata are generalizations of DFA. Instead of exactly one outgoing transition from each state by every input letter, NFA allow several outgoing transitions at the same time. A word is accepted by the NFA if some choice of transitions takes the machine to a final state. Some other choices may lead to a non-final state, but the word is accepted as long as there exists at least one accepting computation path in the automaton.

Example 5. Here is an example of a transition diagram of an NFA:



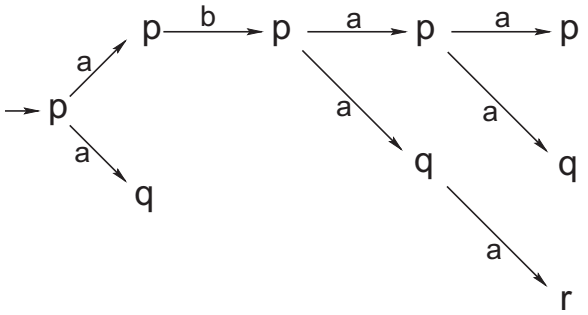
Note that there are two transitions from state p with input letter a, into states p and q. Note also that there are no transitions from state q with input letter b. The number of transitions may be zero, one or more.

NFA may have several different computations for the same input word. Take, for example, the input word $w = abaa$. Already the first input letter a gives two choices in the automaton above: we may go either to state p or state q. Let's say we enter state q. But there is no transition with the next letter b, so this choice does not lead to a final state.

Let us try then the other choice: after the first a the machine is in state p. The second input letter b keeps the machine in state p. The last two letters a and a may take the machine to state q and then state r. So there exists a computation path that takes the machine to a final state. The input $abaa$ is hence accepted, and

$$abaa \in L(A).$$

The following computation tree summarizes all possible computations with input $abaa$:



It is easy to see that the sample NFA accepts exactly those words that contain aa as a subword, so the NFA is **equivalent** to the DFA of Example 1. Two automata are called equivalent if they recognize the same language. □

Let us give a precise definition: An NFA $A = (Q, \Sigma, \delta, q_0, F)$ is specified by 5 items: State set Q, input alphabet Σ, initial state q_0 and the final state set F have the same meaning as for a DFA. The transition function δ is defined differently. It gives for each state q and input letter a a set $\delta(q, a) \subseteq Q$ of possible next states. Using the power set notation

$$2^Q = \{S \mid S \subseteq Q\}$$

we can write

$$\delta : Q \times \Sigma \longrightarrow 2^Q.$$

For example, the transition function δ of the NFA in Example 5 is given by the table

	a	b
p	$\{p, q\}$	$\{p\}$
q	$\{r\}$	$\emptyset$
r	$\{r\}$	$\{r\}$

Let us extend the meaning of the transition function δ the same way we did in DFA. Let us define

$$\hat{\delta} : Q \times \Sigma^* \longrightarrow 2^Q$$

such that $\hat{\delta}(q, w)$ is the set of all states the machine can reach from state q reading input word w. An exact recursive definition goes as follows:

1. For every state q
$$\hat{\delta}(q, \varepsilon) = \{q\}.$$
(No state is changed if no input is read.)

2. For every state q, word w and letter a
$$\hat{\delta}(q, wa) = \{p \in Q \mid \exists r \in \hat{\delta}(q, w) : \ p \in \delta(r, a)\} = \bigcup_{r \in \hat{\delta}(q,w)} \delta(r, a). \tag{2}$$

On single input letters functions δ and $\hat{\delta}$ have identical values:

$$\delta(q, a) = \hat{\delta}(q, a).$$

Therefore there is no risk of confusion if we drop the hat and write simply δ instead of $\hat{\delta}$.

Example 6. In the NFA of Example 5

$$\begin{array}{lcl} \delta(p,a) & = & \{p,q\}, \\ \delta(p,ab) & = & \{p\}, \\ \delta(p,aba) & = & \{p,q\}, \\ \delta(p,abaa) & = & \{p,q,r\}. \end{array}$$

□

The language recognized by NFA $A = (Q, \Sigma, \delta, q_0, F)$ is

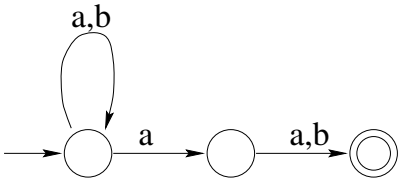
$$L(A) = \{w \in \Sigma^* \mid \delta(q_0, w) \cap F \neq \emptyset\},$$

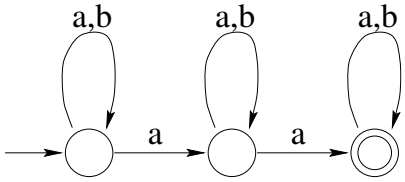
i.e., the language $L(A)$ consists of words w such that there is a final state among the states $\delta(q_0, w)$ reachable from the initial state q_0 on input w.

Example 7. Let us construct small NFA over the input alphabet $\Sigma = \{a, b\}$ that recognize the following languages:

1. Words that end in ab.
2. Words that contain aba as a subword.
3. Words that start with ab and end in ba
4. Words that contain two b's separated by an even number of a's.

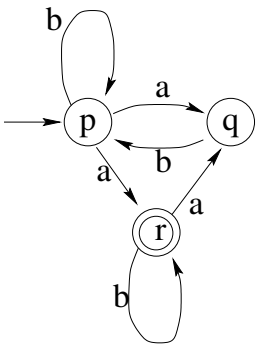
Conversely, let us determine (and describe in English) the languages recognized by the following NFA:





□

Consider the following question: How would one go about checking whether a given NFA A accepts a given input word w ? For example, how would one test if the NFA



accepts the input $w = abbaabb$? One alternative is to try all possible computation paths for w and see if any of them ends in an accepting state. But the number of paths may be very large, and grow exponentially with the length of the input!

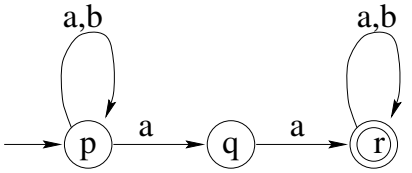
A better way is to scan the input only once, and keep track of the set of possible states. For example, with the NFA above and the input $w = abbaabb$ we have

$$\{p\} \xrightarrow{a} \{q,r\} \xrightarrow{b} \{p,r\} \xrightarrow{b} \{p,r\} \xrightarrow{a} \{q,r\} \xrightarrow{a} \{q\} \xrightarrow{b} \{p\} \xrightarrow{b} \{p\}$$

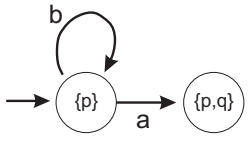
so $\delta(p,w) = \{p\}$, and word w is not accepted because p is not a final state. Notice how the testing is done deterministically while reading the input letter-by-letter, only keeping track of which states can be reached by the prefix of w consumed so far. The testing is, in effect, "DFA-like", and in the following we use this same idea to show how any NFA can be converted into an equivalent DFA. This proves then that NFA only recognize regular languages.

As above, the idea of the proof is to keep track of all possible states that the NFA can be in after reading input symbols. So we construct a DFA whose states are subsets of the state set of the original NFA. The transition function will be constructed in such a way that the state of the DFA after input w is $\delta(q_0, w)$, that is, the set of states that can be reached in the NFA with input w.

Example 8. Consider the NFA

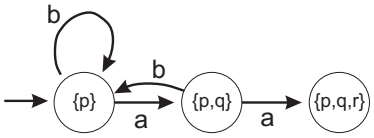


of Example 5. Initially the NFA is in state p so the corresponding DFA is initially in state $\{p\}$. With input letter a the NFA may move to either state p or q, so after input a the DFA will be in state $\{p,q\}$. With input b the NFA remains in state p:

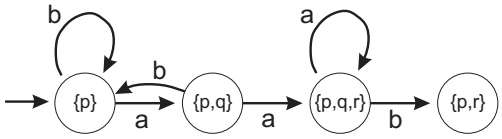


Next we have to figure out the transitions from state $\{p,q\}$. If the DFA is in state $\{p,q\}$ it means that the NFA can be in either state p or q. With input letter a the NFA can move to p or q (if it

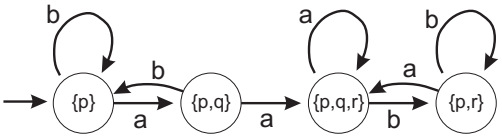
was in state p) or to r (if it was in state q). Therefore, with input letter a the machine can move to any state p or q or r, and so the DFA must make a transition to state $\{p, q, r\}$. With input b the only transition from states p and q is into p. That is why DFA has transition from $\{p, q\}$ back into $\{p\}$:



Consider next the state $\{p, q, r\}$. With input a the NFA can reach any state so the DFA has a loop back to $\{p, q, r\}$. With input b the NFA can move to state p (if it was in state p) or to state r (if it was in state r) so the DFA has a transition from $\{p, q, r\}$ into $\{p, r\}$:

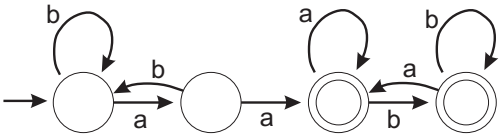


We again have a new state to be processed, state $\{p, r\}$. From p and r the NFA can go to any state with input a, and into states p and r with input b:



No new states were introduced, and all necessary transitions are in place.

We still have to figure out which states should be final states. The NFA accepts word w if it leads to at least one final state. So the DFA should accept w iff it ends into a state S such that S contains at least one final state of the NFA. Our sample NFA has only one final state r, so every state of the new DFA that represents a set containing r is final:



The construction is complete. The result is a DFA that accepts the same exact language as the original NFA. The construction is called a **powerset** construction.

Note that there are also other DFA that accept the same language, some of which may be simpler than ours. We do not care: we only wanted to demonstrate that there exists at least one such DFA. □

Let us prove the general case.

Theorem 9 (Powerset construction) *Given an NFA A one can effectively construct a DFA A' such that $L(A) = L(A')$.*

Proof. Let

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

be any NFA. We construct a DFA

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

as follows. The states of the new DFA A' represent the subsets of the original state set Q, i.e.,

$$Q' = 2^Q.$$

The interpretation of the states is the same as in the example above: If the state of the new DFA machine A' after reading input word w is

$$\{q_1, q_2, \ldots, q_k\}$$

it means that after input w the original nondeterministic machine A can be in state q_1 or q_2 or ... or q_k.

The initial state of A' is

$$q_0' = \{q_0\}.$$

Final states are the elements of Q' that contain at least one final state of A:

$$F' = \{S \subseteq Q \mid S \cap F \neq \emptyset\}.$$

The deterministic transition function δ' is formally defined as follows: For all $S \subseteq Q$ and all $a \in \Sigma$

$$\delta'(S, a) = \bigcup_{q \in S} \delta(q, a). \tag{3}$$

Interpretation: if the nondeterministic machine can be in any state $q \in S$ then after reading the next input letter a it can be in any state that belongs to any of the sets $\delta(q, a)$.

Let us prove that the construction works. Using mathematical induction on the length of the input word w we show that in DFA A'

$$\delta'(q_0', w) = \delta(q_0, w).$$

Note that $\delta(q_0, w)$ on the right is the set of states reachable in the NFA A by input w, while $\delta'(q_0', w)$ is the unique state in the DFA A' reached after reading w. Both are subsets of Q.

1° Case $|w| = 0$, that is, $w = \varepsilon$. This is trivial since

$$\delta'(q_0', \varepsilon) = q_0' = \{q_0\} \quad \text{and} \quad \delta(q_0, \varepsilon) = \{q_0\}.$$

2° Suppose the claim is true for all inputs of length l. Let $|w| = l + 1$. Then $w = ua$ for some word u of length l and letter $a \in \Sigma$. The claim is true for input u, so

$$\delta'(q_0', u) = \delta(q_0, u).$$

Denote $S = \delta'(q_0', u) = \delta(q_0, u)$. We have

$$\delta'(q_0', ua) = \delta'(S, a) = \bigcup_{q \in S} \delta(q, a) = \delta(q_0, ua).$$

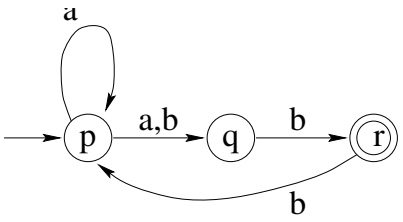
Here the three equalities follow from (1), (3) and (2), respectively.

Our proof is almost done. Word w is accepted by A' if an only if $\delta'(q_0', w)$ contains an element of F. But $\delta(q_0, w)$ is the same set, and A accepts w if and only if there is a final state in $\delta(q_0, w)$. So $L(A) = L(A')$. □

Corollary 10 *Languages accepted by NFA are regular.* □

Remark: In the powerset automaton one only needs to include as states those subsets of Q that are reachable from the initial set $\{q_0\}$. So in practice one adds new states to the DFA during the construction only when they are needed, as we did in the Example 8.

Example 11. Let us use the powerset construction to construct a DFA that is equivalent to

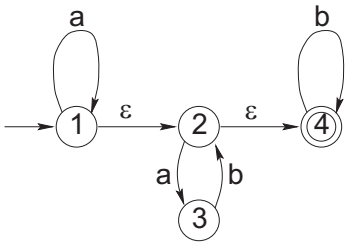


□

2.3 NFA with ε-moves

In forthcoming constructions it turns out to be useful to extend the notion of NFA by allowing spontaneous transitions. When an NFA executes a spontaneous transition, known as an ε-move, it changes its state without reading any input letter. Any number of ε-moves are allowed.

Example 12. Here's an example of an NFA with ε-moves:



For example, word $w = aabbb$ is accepted as follows: The first a keeps the machine in state 1. Then an ε-move to state 2 is executed, without reading any input. Next, ab is consumed through states 3 and 2, followed by another ε-move to state 4. The last bb keeps the automaton in the accepting state 4.

The automaton of this example accepts any sequence of a's followed by any repetition of ab's followed by any number of b's:

$$L(A) = \{a^i(ab)^j b^k \mid i, j, k \geq 0\}.$$

□

Formally, an NFA with ε-moves (or **ε-NFA** for short) is $A = (Q, \Sigma, \delta, q_0, F)$ where Q, Σ, q_0 and F are as before, and δ is a function

$$Q \times (\Sigma \cup \{\varepsilon\}) \longrightarrow 2^Q$$

that specifies for each state q the transitions with all input letters a and the empty word ε.

- $\delta(q, a)$ is the set of all states p such that there is a move from q to p with input a, and
- $\delta(q, \varepsilon)$ is the set of states p such that there is a spontaneous move from q to p.

Example 13. The transition function δ for the ε-NFA of Example 12 is

	a	b	ε
1	$\{1\}$	$\emptyset$	$\{2\}$
2	$\{3\}$	$\emptyset$	$\{4\}$
3	$\emptyset$	$\{2\}$	$\emptyset$
4	$\emptyset$	$\{4\}$	$\emptyset$

□

As in the previous sections, we extend δ into a function $\hat{\delta}$ that gives possible states for all input words:

$$\hat{\delta} : Q \times \Sigma^* \longrightarrow 2^Q,$$

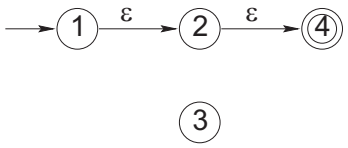
where $\hat{\delta}(q, w)$ is the set of all possible states after the automaton reads input word w, starting at state q. When processing w any number of ε-moves may be used.

Before the recursive definition of $\hat{\delta}$ we need to know all states the machine can enter from q without reading any input letters, i.e. using only ε-moves. Let us call this set the **ε-CLOSURE** of state q.

Example 14. In the ε-NFA of Example 12 we have

$$\begin{aligned} \varepsilon\text{-CLOSURE } (1) &= \{1, 2, 4\}, \\ \varepsilon\text{-CLOSURE } (2) &= \{2, 4\}, \\ \varepsilon\text{-CLOSURE } (3) &= \{3\}, \\ \varepsilon\text{-CLOSURE } (4) &= \{4\}. \end{aligned}$$

The ε-CLOSURE(q) can be easily found by analyzing which nodes can be reached from state q using only ε-moves. This is a simple reachability condition in the directed graph formed by the ε-edges, and can be solved effectively by a standard graph algorithm:



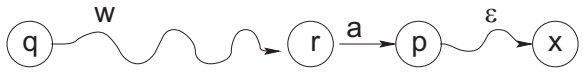
If S is any set of states let us denote by ε-CLOSURE (S) the set of states reachable from any element of S using only ε-moves, i.e.

$$\varepsilon\text{-CLOSURE } (S) = \bigcup_{q \in S} \varepsilon\text{-CLOSURE } (q).$$

Now we are ready to give a recursive definition of $\hat{\delta}(q, w)$:

1. For every state q
$$\hat{\delta}(q, \varepsilon) = \varepsilon\text{-CLOSURE } (q).$$
(By definition, the ε-CLOSURE consists of all states reachable without consuming any input letter.)

2. For every state q, word w and letter a
$$\hat{\delta}(q, wa) = \varepsilon\text{-CLOSURE } \left(\bigcup_{r \in \hat{\delta}(q,w)} \delta(r, a) \right).$$



The language accepted by ε-NFA A is

$$L(A) = \{w \in \Sigma^* \mid \hat{\delta}(q_0, w) \cap F \neq \emptyset\}.$$

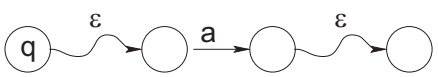
Example 15. In the ε-NFA of Example 12

$$\begin{aligned} \hat{\delta}(1, a) &= \varepsilon\text{-CLOSURE } (\delta(1, a) \cup \delta(2, a) \cup \delta(4, a)) = \varepsilon\text{-CLOSURE } (\{1, 3\}) = \{1, 2, 3, 4\}, \\ \hat{\delta}(1, ab) &= \varepsilon\text{-CLOSURE } (\delta(1, b) \cup \delta(2, b) \cup \delta(3, b) \cup \delta(4, b)) = \varepsilon\text{-CLOSURE } (\{2, 4\}) = \{2, 4\}. \end{aligned}$$

□

The values of $\hat{\delta}(q, a)$ for single letters a are of special interest to us. Note, however, that they are not necessarily identical to $\delta(q, a)$, so we cannot remove the "hat" as we did with DFA and NFA:

- $\delta(q, a)$ contains only states you can reach with one transition labeled by a. It does not allow using ε-moves.
- In contrast, $\hat{\delta}(q, a)$ contains all states you can reach from q by doing any number of ε-moves and one a-transition, in any order:



Introducing spontaneous transitions does not increase the power of NFA:

Theorem 16 *Given an ε-NFA A, one can effectively construct an NFA A' such that $L(A) = L(A')$.*

Proof. Let

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

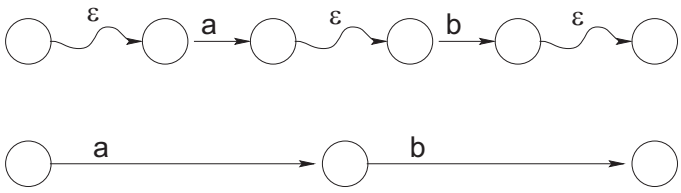
be any ε-NFA. We construct the NFA

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

that has the same state set Q and the initial state q_0 as A, and whose transition function δ' anticipates all possible ε-moves in A before and after each letter transition: In A', there is a transition from q to p with letter a if and only if in A there is a computation path from q to p that reads a from the input. This means that δ' is the same as $\hat{\delta}$ we discussed earlier:

$$\delta'(q, a) = \hat{\delta}(q, a)$$

for all $q \in Q$ and $a \in \Sigma$. Now any computation in A corresponds to a "leaner" computation in A' that skips over all ε-moves:



Let us prove, using mathematical induction on $|w|$, that

$$\delta'(q, w) = \hat{\delta}(q, w)$$

for every $q \in Q$ and every non-empty word w. (Note that if $w = \varepsilon$ the claim may not be true: $\delta'(q, \varepsilon)$ contains only state q whereas $\hat{\delta}(q, \varepsilon)$ contains everything in ε-CLOSURE(q).)

1° Let $|w| = 1$. Then $w = a$ is a letter and

$$\delta'(q, a) = \hat{\delta}(q, a)$$

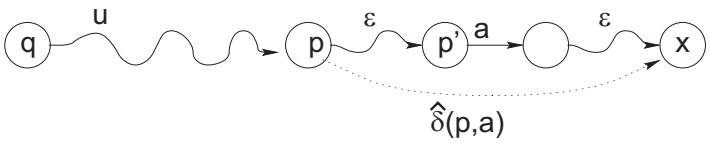
by the definition of δ'.

2° Assume the claim has been proved for u and consider $w = ua$. Let us denote

$$S = \hat{\delta}(q, u) = \delta'(q, u).$$

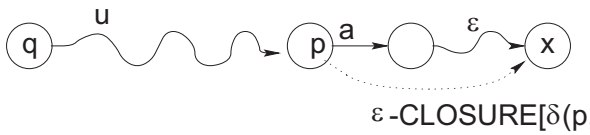
Then in the NFA A'

$$\delta'(q, ua) = \bigcup_{p \in S} \delta'(p, a) = \bigcup_{p \in S} \hat{\delta}(p, a) \ :$$



In the ε-NFA A

$$\hat{\delta}(q, ua) = \varepsilon - \text{CLOSURE}\left[\bigcup_{p \in S} \delta(p, a)\right] :$$



These two sets are identical: Any x that can be reached from p through an $a\varepsilon$ path is trivially reachable through an $\varepsilon a\varepsilon$ path. (Just use $p' = p$.)

On the other hand, state p' along the $\varepsilon a\varepsilon$ path is an element of S. State x that can be reached from p using $\varepsilon a\varepsilon$, can be reached from p' using $a\varepsilon$.

We have shown that $\delta'(q, w) = \hat{\delta}(q, w)$ for all non-empty words w. Next we have to determine which states should be made final states in A'. Assume first that $\varepsilon \notin L(A)$. Then we can simply set

$$F' = F.$$

Because $\delta'(q_0, w) = \hat{\delta}(q_0, w)$, exactly same non-empty inputs w are accepted. And the empty word ε is not accepted by either automaton.

Assume then that $\varepsilon \in L(A)$. Then we have to include the initial state q_0 among the final states in A':

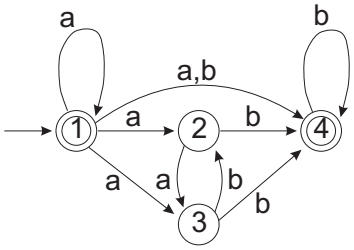
$$F' = F \cup \{q_0\}.$$

Now both automata accept ε. If A accepts a non-empty w then A' accepts it with a computation to the same final state. If A' accepts non-empty w then it either has a computation leading to some element of F (in which case A accepts it) or it has a computation leading to q_0. But there is an ε-path in A from q_0 to some final state so A accepts w as well.

We have proved that $L(A) = L(A')$. □

Corollary 17 *Languages accepted by ε-NFA are regular.* □

Example 18. If we apply the conversion process to the ε-NFA of Example 12 we obtain the equivalent NFA



Note that since ε is in the language, state 1 is made final. □

Now we know how to convert any ε-NFA into an equivalent NFA, and how to convert any NFA into an equivalent DFA, so we can convert any ε-NFA into a DFA. All three automata models accept same languages.

2.4 Regular expressions

Regular expressions provide a completely different technique to define languages. Instead of being accepting devices such as finite automata they are descriptions of how a language is build from simple atomic languages using some basic language operations. As we will see later, the important Kleene theorem states that regular expressions define exactly the same family of languages as finite automata.

Let us first define some operations on languages. Let Σ be an alphabet, and let L, L_1 and L_2 be languages over Σ.

- The **concatenation** L_1L_2 of languages L_1 and L_2 is defined to be the language containing all words obtained by concatenating a word from L_1 and a word from L_2. In other words,

  $$L_1L_2 = \{uv \mid u \in L_1 \text{ and } v \in L_2\}.$$

  For example,

  $$\{ab, b\}\{aa, ba\} = \{abaa, abba, baa, bba\}.$$

- For every $n \geq 0$ we define L^n to be the set of words obtained by concatenating n words from language L. In other words,

  $$\begin{aligned} L^0 &= \{\varepsilon\} \\ L^n &= L^{n-1}L \text{ for every } n \geq 1. \end{aligned}$$

  For example,

  $$\{ab, b\}^3 = \{ababab, ababb, abbab, abbb, babab, babb, bbab, bbb\}.$$

- The **Kleene closure** L^* of language L is defined to be the set of words obtained by concatenating any number of words from L together:

  $$L^* = \bigcup_{i=0}^{\infty} L^i.$$

  For example,

  $$\{ab, b\}^* = \{\varepsilon, ab, b, abab, abb, bab, bb, ababab, ababb, \ldots\}.$$

  Note that this notation is consistent with our use of Σ^* to denote the set of all words over the alphabet Σ.

- The positive closure L^+ of L is

  $$L^+ = \bigcup_{i=1}^{\infty} L^i,$$

  i.e. words that are concatenations of one or more words from L. For example,

  $$\{ab, b\}^+ = \{ab, b, abab, abb, bab, bb, ababab, ababb, \ldots\}.$$

  We have always

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

  Also, we have

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

Note that $L^* = L^+$ if and only if $\varepsilon \in L$. Note also that $\emptyset^* = \{\varepsilon\}$ while $\emptyset^+ = \emptyset$.

Next we define **regular expressions** over the alphabet Σ. They are syntactic expressions that represent certain languages. If r is a regular expression then we denote the language it represent by $L(r)$. Regular expressions are defined recursively as follows:

1. $\emptyset$ is a regular expression representing the empty language.
2. ε is a regular expression and it represents the singleton language $\{\varepsilon\}$.
3. Every letter a of Σ is a regular expression representing the singleton language $\{a\}$.
4. If r and s are arbitrary regular expressions then $(r+s)$, (rs) and (r^*) are regular expressions. If $L(r) = R$ and $L(s) = S$ then

$$\begin{array}{lcl} L(r+s) & = & R \cup S, \\ L(rs) & = & RS, \\ L(r^*) & = & R^*. \end{array}$$

Two regular expressions r and s are **equivalent**, denoted by $r = s$, if $L(r) = L(s)$.

In practice we remove parentheses from regular expressions using the following precedence rules: The Kleene star * has highest precedence, concatenation has the second highest precedence, and union $+$ has the lowest precedence. Because concatenation and union are associative, we can simplify $r(st)$ and $(rs)t$ into rst, and $r + (s + t)$ and $(r + s) + t$ into $r + s + t$. For example,

$$(((ab)^*) + (a(ba)))((a^*)b) = ((ab)^* + aba)a^*b.$$

Often we do not distinguish between a regular expression r and its language $L(r)$. We simplify notations by talking about language r. An expression rr^* will be frequently denoted as r^+, and the expression $\overbrace{rr\ldots r}^{n}$ will be frequently abbreviated as r^n.

Example 19. Let us construct regular expressions for the following languages over the alphabet $\Sigma = \{a, b\}$:

1. $L = \{ab, ba\}$.
2. All words of Σ.
3. All words that start with a and end in b.
4. All words that contain aba as a subword.
5. Words that start with ab and end in ba.
6. Words that contain two b's separated by an even number of a's.

Conversely, let us determine (and express in English) the following languages:

1. $a^*(ab)^*b^*$.
2. $(ab + b)^*$.

3. $(\varepsilon + b)(ab)^*(\varepsilon + a)$.

□

Next we want to show that regular expressions can define exactly the same languages as DFA, NFA and ε-NFA recognize, i.e., regular languages.

Theorem 20 (Kleene 1956) *Language L is regular if and only if there exists a regular expression for it.*

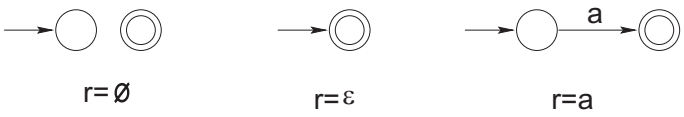
Proof. To prove the theorem we need to show two directions:

(A) We show how to construct for any given regular expression an equivalent ε-NFA.

(B) We show how to construct for any given DFA an equivalent regular expression.

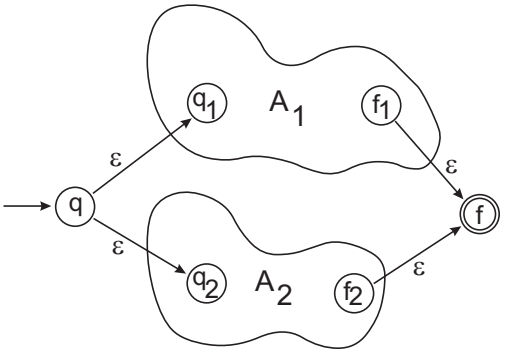
Part(A): Let r be any regular expression. Then r can be $\emptyset$, ε or a, or it can be formed from two smaller regular expression s and t by union $r = s + t$, concatenation $r = st$ or the Kleene closure $r = s^*$. The construction is done inductively: we show how to construct an ε-NFA for $\emptyset$, ε and a. Then we show how to construct an ε-NFA for $s + t$, st and s^* if we already have ε-NFA for s and t. Using these constructions one can build an ε-NFA for any regular expression. We prove even more: we prove inductively that everything can be build using only one final state.

1. Below are diagrams for automata that recognize $\emptyset$, $\{\varepsilon\}$ and $\{a\}$. Every machine has exactly one final state:

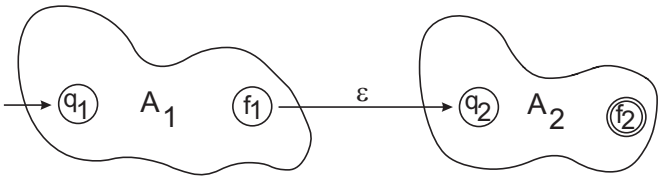


2. Assume we have ε-NFA A_1 and A_2 for regular expressions s and t, respectively. Assume A_1 and A_2 have only one final state each:

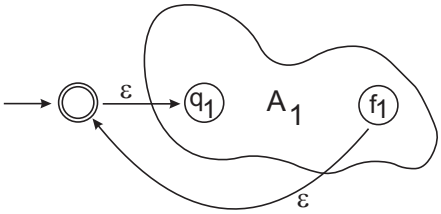
The following ε-NFA recognizes the union $s + t$. Two new states are added that become the new initial and final states, and four ε-transitions allow spontaneous moves to either A_1 or A_2.



This is an ε-NFA for the concatenation st:



Finally, this is an ε-NFA for the Kleene star s^*:



All constructions above work, and the resulting ε-NFA have exactly one final state. Using these constructions we can build an ε-NFA for any given regular expression. This completes the proof of (A).

Part (B): Let

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

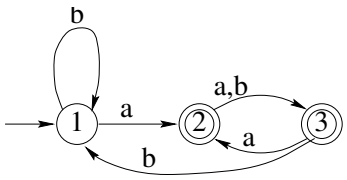
be any given DFA. We want to construct a regular expression for language $L(A)$. Let us number the states from 1 to n:

$$Q = \{1, 2, \ldots, n\}.$$

In this construction we build regular expressions for the following languages R_{ij}^k:

$$R_{ij}^k = \{w \mid \quad w \text{ takes the automaton from state } i \text{ to state } j \\ \text{without going through any states greater than } k.\}.$$

"Going through" a state means that the state is along the computation path, excluding the starting and ending of the computation. For example, in the DFA



word *bbab* belongs to R^2_{33} since the computation

$$3 \longrightarrow 1 \longrightarrow 1 \longrightarrow 2 \longrightarrow 3$$

does not go through any state greater than 2, but word *baaab* does not belong to R^2_{33} since its computation path goes through state 3:

$$3 \longrightarrow 1 \longrightarrow 2 \longrightarrow 3 \longrightarrow 2 \longrightarrow 3,$$

and word *bbaba* is not in R^2_{33} since its computation path does not end in state 3.

Let us construct regular expressions r^k_{ij} for the languages R^k_{ij}. We start with $k = 0$ and then move up to larger values of k.

1. $k = 0$. Now R^k_{ij} is the language of all words that take the automaton from state i into state j without going through any states at all. Only possible input words are ε (if $i = j$) and single letters a (if $\delta(i, a) = j$). If $a_1, a_2, \ldots, a_p$ are the input letters with transitions from state i to j then R^0_{ij} has regular expression

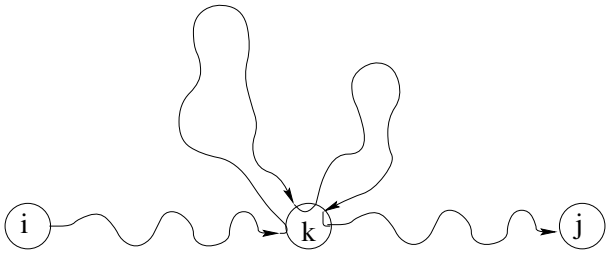
   $$a_1 + a_2 + \ldots + a_p$$

   (in the case $i \neq j$) or

   $$\varepsilon + a_1 + a_2 + \ldots + a_p$$

   (in the case $i = j$). If there are no transitions between the two states then regular expression $\emptyset$ is used.

2. Let $k > 0$, and assume we have constructed regular expressions for all R^{k-1}_{ij}. Consider an arbitrary computation path from state i into state j that only goes through states $\{1, 2, \ldots, k\}$. Let us cut the path into segments at points where it goes through state k:

   All segments only go through states $\{1, 2, \ldots, k-1\}$. The first segment is in the set R^{k-1}_{ik}, and the last segment belongs to R^{k-1}_{kj}. All middle segments start and end in state k, so they belong to R^{k-1}_{kk}. Number of middle segments can be arbitrary. So all words that visit k at least once are in the set

   $$R^{k-1}_{ik} \left(R^{k-1}_{kk}\right)^* R^{k-1}_{kj}.$$

We also must include words that do not visit state k even once: They belong to set R_{ij}^{k-1}.

We have derived the regular expression

$$r_{ik}^{k-1}\ (r_{kk}^{k-1})^*\ r_{kj}^{k-1}\ +\ r_{ij}^{k-1}$$

for the language R_{ij}^k.

Regular expression r_{ij}^n represents all strings that take the automaton from state i to state j through any states. If $i = 1$ is the initial state then r_{1j}^n represents strings whose computation paths finish in state j. If states $j_1, j_2, \ldots, j_f$ are the final states of the DFA then the language recognized by the automaton is represented by the expression

$$r_{1j_1}^n + r_{1j_2}^n + \ldots + r_{1j_f}^n.$$

These are all words that take the machine from the initial state to some final state, using any states whatsoever on the way.

This completes the proof of (B): The construction provides an equivalent regular expression for any given DFA. □

In practice the construction in part (B) provides huge regular expressions even for small automata. It is a good idea to simplify the expressions r_{ij}^k as you go along. The following simplifications are especially useful: For any regular expressions r, s and t, both sides of following simplification rules describe the same language:

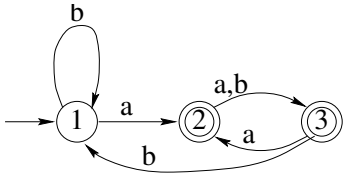
ε^*	$\longrightarrow$	ε
$\varepsilon r, r\varepsilon$	$\longrightarrow$	r
$(r+\varepsilon)^*$	$\longrightarrow$	r^*
$\emptyset r, r\emptyset$	$\longrightarrow$	$\emptyset$
$\emptyset + r, r + \emptyset$	$\longrightarrow$	r
$r(s+t)$	$\longrightarrow$	$rs + rt$
$(s+t)r$	$\longrightarrow$	$sr + tr$
$(r+\varepsilon)r^*, r^*(r+\varepsilon)$	$\longrightarrow$	r^*

Another way to save time is to construct only those expressions r_{ij}^k that are actually needed.

Example 21. As an example of the construction in part (A) of the proof, let us build an ε-NFA for the regular expression

$$(abb^* + a)^*.$$

As an example of part (B), let us form a regular expression for the language recognized by the DFA



□

We finish this section by showing another, language equation based method for constructing a regular expression from a DFA. As in the part (B) above, we number the states from 1 to n:

$$Q = \{1, 2, \ldots, n\},$$

and assume that 1 is the initial state. For every $i, j \in Q$, let us denote

$$L_i = \{w \in \Sigma^* \mid \delta(1, w) = i\}$$

and

$$K_{ij} = \{a \in \Sigma \mid \delta(i, a) = j\}.$$

Then the following equalities of languages hold:

$$\begin{array}{lcl} L_1 & = & L_1K_{11} \cup L_2K_{21} \cup \ldots \cup L_nK_{n1} \cup \{\varepsilon\}, \\ L_2 & = & L_1K_{12} \cup L_2K_{22} \cup \ldots \cup L_nK_{n2}, \\ & \vdots & \\ L_n & = & L_1K_{1n} \cup L_2K_{2n} \cup \ldots \cup L_nK_{nn}. \end{array} \tag{4}$$

In each equation the language on the left and on the right are identical. The languages K_{ij} can be read directly from the given DFA – they are a representation of the transitions. Note that the languages K_{ij} do not contain the empty word ε. It turns out for such K_{ij} the languages L_i are uniquely determined from the system of equations above. The following lemma states this for the case $n = 1$:

Lemma 22 *Let $K \subseteq \Sigma^+$ and $L \subseteq \Sigma^*$ be regular. Then $X = LK^*$ is the unique language for which the equality $X = XK \cup L$ holds.*

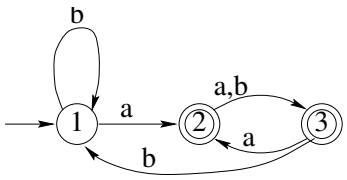
Proof. If $X = LK^*$ then

$$XK \cup L = LK^*K \cup L = LK^+ \cup L = LK^* = X,$$

so the equality holds. Let us prove the uniqueness. Suppose X_1 and X_2 are two different languages such that $X_1 = X_1K \cup L$ and $X_2 = X_2K \cup L$. Let w be a shortest word that belongs to one but not both of the languages. Without loss of generality, assume that $w \in X_1$ but $w \notin X_2$. Because $w \in X_1 = X_1K \cup L$ we have either $w \in X_1K$ or $w \in L$. But because $L \subseteq X_2$ and $w \notin X_2$, we have $w \notin L$. So $w \in X_1K$. We have $w = uv$ for some $u \in X_1$ and $v \in K$. Because $\varepsilon \notin K$, we have $|u| < |w|$. By the minimality of w we then have $u \in X_2$. Hence $w = uv \in X_2K \subseteq X_2$, a contradiction. □

Using the Lemma on the equations in (4) one-by-one (and substituting the solutions obtained) allows one to uniquely solve $L_1, L_2, \ldots, L_n$.

Example 23. Let us use the method of language equations to find a regular expression for the language recognized by



The system of language equations read from the automaton is

$$\begin{aligned} L_1 &= L_1b + L_3b + \varepsilon, \\ L_2 &= L_1a + L_3a, \\ L_3 &= L_2(a + b). \end{aligned}$$

Substituting L_3 from the third equation to the first two gives

$$\begin{aligned} L_1 &= L_1b + L_2(ab + bb) + \varepsilon, \\ L_2 &= L_1a + L_2(aa + ba). \end{aligned}$$

Solving L_2 from the second equation, using Lemma 22, gives

$$L_2 = L_1a(aa + ba)^*,$$

which we substitute to the first equation:

$$L_1 = L_1b + L_1a(aa + ba)^*(ab + bb) + \varepsilon = L_1(b + a(aa + ba)^*(ab + bb)) + \varepsilon.$$

Now we can use Lemma 22 to solve L_1:

$$L_1 = (b + a(aa + ba)^*(ab + bb))^*.$$

Substituting back we obtain L_2 and L_3, and finally

$$L(A) = L_2 + L_3 = (b + a(aa + ba)^*(ab + bb))^*a(aa + ba)^*(\varepsilon + a + b).$$

□

2.5 The pumping lemma

We have learned several types of devices to define formal languages, and we have proved that all of them are able to describe exactly same languages, called **regular languages**. We presented procedures for converting any device into an equivalent device of any other type. The conversion procedures are mechanical algorithms in the sense that one can write a computer program that performs any conversion. We say the devices are **effectively** equivalent.

Let us consider the following question: We are given a language L (described in English, for example) and we want to prove that L is regular. This is fairly straightforward: all we have to do is to design a finite automaton (DFA, NFA or ε-NFA) or a regular expression, and to show that our design defines language L.

Consider then the converse situation: If language L is not regular how do we prove it ? How can we show that there does not exist any finite state device that defines L ? We cannot try all

DFA one-by-one because there are infinitely many of them. In general, proving that something cannot be done is harder than proving that it can be done, and that makes the negative question more interesting.

Example 24. Consider the following example: Language

$$L = \{a^i b^i \mid i \geq 0\}$$

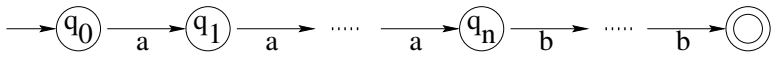
contains all words that begin with any number of a's, followed by equally many b's. Let us show that there does not exist any DFA that recognizes L.

A DFA that recognizes L should first count the number of a's in the beginning of the word. The problem is that a DFA has only finitely many states, so it is bound to get confused: Some a^i and a^j take the machine to the same state, and the machine can no longer remember whether it saw i or j letters a. Since the machine accepts input word $a^i b^i$, it also accepts input word $a^j b^i$, which is not in the language. So the machine works incorrectly.

Let us be more precise: Assume that there exists a DFA

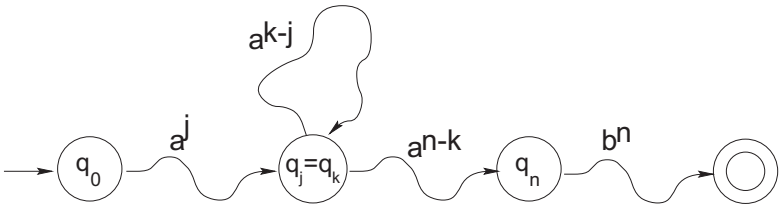
$$A = (Q, \Sigma, \delta, q_0, F)$$

such that $L(A) = L = \{a^i b^i \mid i \geq 0\}$. Let n be the number of states in Q. Consider the accepting computation path for the input word $a^n b^n$.



Let q_j be the state of A after reading the first j input letters a.

There are $n + 1$ states q_0, q_1, ..., q_n but the machine has only n different states, so two of the states must be identical. (This is known as the pigeonhole principal: if you have more pigeons than holes then two or more pigeons have to share a hole.) Let q_j and q_k be two identical states, where $j < k$:



Input a^{k-j} loops the automaton at state $q_j = q_k$. The loop can be repeated arbitrarily many times, always getting an accepting computation. For example, if we repeat the loop twice we have an accepting computation for the word

$$a^j a^{k-j} a^{k-j} a^{n-k} b^n = a^{n+(k-j)} b^n.$$

This word is not in language L because $k - j \neq 0$. Therefore automaton A is not correct: the language it recognizes is not L. □

Similar "confused in counting" argument works with many other languages as well. Instead of always repeating the above argument in each case, we formulate the argumentation as a theorem

known as the pumping lemma. The word "pumping" refers to the fact that the loop can be repeated, or "pumped", arbitrarily many times. The pumping lemma states a property that every regular language satisfies. If a language does not have this property then we know that the language is not regular.

Theorem 25 (Pumping lemma) *Let L be a regular language. Then there exists some positive constant n such that, every word $z \in L$ of length at least n can be divided into three segments*

$$z = uvw$$

in such a way that

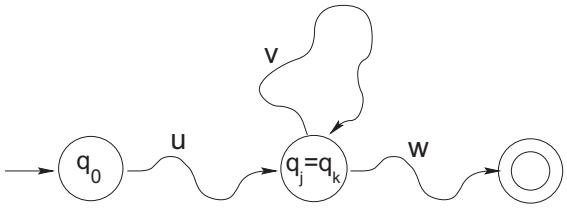
$$\begin{cases} |uv| \leq n, \text{ and} \\ v \neq \varepsilon, \end{cases} \tag{5}$$

and for all $i \geq 0$ the word uv^iw is in language L.

(In other words, every long enough word of L contains a non-empty subword that can be pumped arbitrarily many times, and the result always belongs to L.)

Proof. Let L be a regular language. Then it is recognized by some DFA A. Let n be the number of states in A. Consider an arbitrary word $z \in L$ such that $|z| \geq n$. We have to show how to divide z into three segments in such a way that (5) is satisfied.

We argue as follows: Let q_j be the state of the machine after the first j input letters of z have been read. The machine has n states so the pigeonhole principal tells that there must exist two identical states among $q_0, q_1, \ldots, q_n$, say $q_j = q_k$ for some $j < k \leq n$:

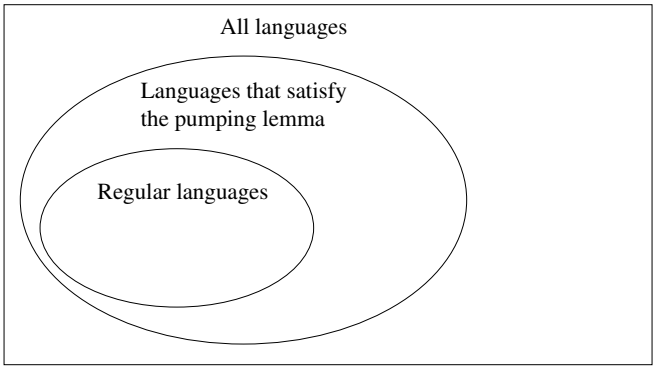


We divide the input word z into three segments in a natural way: u consists of the first j letters, v of the following $k - j$ letters, and w is the remaining tail. This division satisfies (5):

$$\begin{aligned} &|uv| = k \leq n, \text{ and} \\ &v \neq \varepsilon \text{ because } |v| = k - j > 0. \end{aligned}$$

The loop reading input v can be repeated any number of times, which implies that $uv^iw \in L$ for all $i \geq 0$. □

The pumping lemma gives a property satisfied by every regular language. Therefore, if language L does not satisfy the pumping lemma then L is not regular. We can use the pumping lemma to show that certain languages are not regular. But note the following: One can not use the pumping lemma to prove that some language L is regular. The property is not "if and only if": There are some non-regular languages that satisfy the pumping lemma.



Hence we apply the negated formulation of the pumping lemma. The pumping lemma says that if L is regular then

$$(\exists n)(\forall z \ldots)(\exists u, v, w \ldots)(\forall i) uv^i w \in L.$$

Therefore if L satisfies the opposite statement

$$(\forall n)(\exists z \ldots)(\forall u, v, w \ldots)(\exists i) \quad uv^i w \notin L$$

then L is not regular. So to use the pumping lemma to show that L is not regular

(1) for **every** n, select a suitable word $z \in L$, $|z| \geq n$, and

(2) show that for **every** division $z = uvw$ where $|uv| \leq n$ and $v \neq \varepsilon$ **there exists** a number i such that

$$uv^i w$$

is **not** in the language L.

If it is possible to do (1) and (2) then L does not satisfy the pumping lemma, and L is not regular.

Example 26. Consider the language

$$L = \{a^i b^i \mid i \geq 0\}$$

from Example 24.

(1) For any given n, select $z = a^n b^n$. The choice is good since $z \in L$ and $|z| \geq n$.

(2) Consider an arbitrary division of z into three part $z = uvw$ where

$$|uv| \leq n \text{ and } v \neq \varepsilon.$$

Then necessarily $u = a^j$ and $uv = a^k$ for some j and k, and $j < k$. Choosing $i = 2$ gives

$$uv^i w = uvvw = a^{n+(k-j)} b^n$$

which does not belong to L.

□

Example 27. Let us prove that

$$L = \{a^{m^2} \mid m \geq 1\} = \{a, a^4, a^9, a^{16}, \ldots\}$$

is not regular.

(1) For any given n, let us choose

$$z = a^{n^2}.$$

The choice is good since $z \in L$ and $|z| = n^2 \geq n$.

(2) Consider an arbitrary division $z = uvw$ that satisfies (5). Let k denote the length of v. It follows from (5) that $0 < k \leq n$. Let us choose $i = 2$. Then

$$uv^i w = uvvw = a^{n^2+k}$$

Because

$$n^2 + k > n^2 + 0 = n^2$$

and

$$n^2 + k \leq n^2 + n < (n+1)^2,$$

number $n^2 + k$ is not a square of any integer. (It falls between two consecutive squares.) Therefore uv^2w does not belong to L.

□

Example 28. Let us use the pumping lemma to show that the following two languages are not regular:

$$\begin{aligned} L &= \{ww \mid w \in \{a, b\}^*\}, \\ L &= \{a^p \mid p \text{ is a prime number}\}. \end{aligned}$$

2.6 Closure properties

It follows from the definition of regular expressions that the union $L_1 \cup L_2$ of any two regular languages L_1 and L_2 is regular. Namely, if r_1 and r_2 are regular expressions for L_1 and L_2 then $r_1 + r_2$ is a regular expression for the union $L_1 \cup L_2$. In the same way, the concatenation of two regular languages is always regular: $r_1 r_2$ is a regular expression for the concatenation $L_1 L_2$. We say that the family of regular languages is **closed** under union and concatenation.

In general, let Op denote some **language operation**, that is, an operation whose operands and result are all formal languages. We say that the family of regular languages is closed under operation Op if the result after applying Op on regular languages is always a regular language.

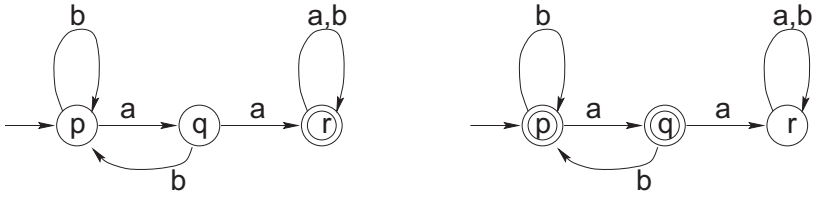
In this section we investigate the closure properties of regular languages under some common operations. Let us first look into the familiar boolean operations: union, intersection and complement. We already know the closure under union. Consider then the complement. Let $L \subseteq \Sigma^*$ be a regular language over the alphabet Σ. To prove that the complement

$$\overline{L} = \Sigma^* \setminus L = \{w \in \Sigma^* \mid w \notin L\}$$

is also regular, consider a DFA A that recognizes L. Without loss of generality we may assume that A uses alphabet Σ, the same alphabet relative to which the complement is defined. (We can remove all transitions, if any, that use letters not in Σ, and we can add transitions to a sink state with all letters of Σ that are not in the original alphabet of the automaton.)

Every input word w takes the automaton to a unique state q. The word w is accepted if and only if state q is among the final states of the machine. Let us change A by making every final state a non-final state, and vice versa. The new machine accepts word w if and only if the original machine did not accept w. In other words, the new machine recognizes the complement of language L.

Example 29. For example, the languages recognized by the deterministic automata



are complements of each other with respect to $\{a, b\}^*$. □

Consider then the intersection of two languages. Elementary set theory (de Morgan's laws) tells us how to use union and complement to do intersection:

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

for any sets L_1 and L_2. If L_1 and L_2 are regular, so are $\overline{L}_1$ and $\overline{L}_2$. Therefore also $\overline{L}_1 \cup \overline{L}_2$ and $\overline{\overline{L}_1 \cup \overline{L}_2}$ are regular. This proves that the family of regular languages is closed under intersection as well.

Here is an important point to note: All closures above are effective in the sense that we can mechanically construct the result of the operation on any given regular languages. For example the union can be constructed by just inserting a plus sign between the regular expressions of the input languages. It is irrelevant which description we use for the operands because DFA, NFA, ε-NFA and regular expressions are effectively equivalent with each other.

We say that the family of regular languages is **effectively closed** under operation Op if there exists an algorithm (=mechanical procedure) that produces the result of the operation for any given regular input languages. The format of the inputs and outputs can be any of the discussed devices (finite automaton or regular expression) since we have algorithms for converting from one format to any other.

We have proved the following theorem:

Theorem 30 *The family of regular languages is effectively closed under union, concatenation, Kleene star, complementation and intersection.* □

Operations union, concatenation and Kleene star are called the **rational** operations.

The closure properties can be used to prove languages regular. It is enough to show how to build the language from known regular languages using above operations.

Example 31. The language L containing all words over the English alphabet

$$\Sigma = \{a, b, c, \ldots, z\}$$

that do not contain the word *matematiikka* as a subword is regular. Namely, it is the complement of the language

$$\Sigma^* matematiikka \Sigma^*.$$

□

We can also use closure properties to show that some languages are not regular:

Example 32. Let

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

We could use pumping lemma to prove that L is not regular. But we can also reason as follows: Assume that L is regular. Then also the language

$$L \cap a^* b^*$$

is regular because the family of regular languages is closed under intersection and a^*b^* is regular. But we know from the previous section that the language

$$L \cap a^* b^* = \{a^n b^n \mid n \geq 0\}.$$

is not regular. Therefore our assumption that L is regular has to be false. □

Example 33. Here's another example: Let

$$L = \{a^{p-1} \mid p \text{ is a prime number }\}.$$

If L were regular, so would be

$$aL = \{a^p \mid p \text{ is a prime number }\}.$$

But it was shown in Example 28 that this is not regular, a contradiction. □

So if you start with L, apply operations to L and known regular languages, and end up with a language that is known to be non-regular, then L is non-regular. (Be careful not do it in the wrong direction: If you start with L, apply operations on L and regular languages and end up with a known regular language, then you can not conclude anything about the regularity of L.)

Next we introduce some new language operations. Let Σ and Δ be two alphabets. A **homomorphism** from Σ to Δ is a function

$$h : \Sigma^* \longrightarrow \Delta^*,$$

that assigns to each letter $a \in \Sigma$ a word $h(a) \in \Delta^*$. For example,

$$\begin{aligned} h(a) &= 0, \\ h(b) &= 01. \end{aligned}$$

gives a homomorphism from $\Sigma = \{a, b\}$ to $\Delta = \{0, 1\}$.

Homomorphisms are applied to words by coding each letter separately and concatenating the results together:

$$h(a_1 a_2 \ldots a_n) = h(a_1)h(a_2)\ldots h(a_n).$$

For example, using the homomorphism h above we have

$$\begin{array}{lcl} h(ba) & = & 010, \\ h(babba) & = & 01001010, \\ h(\varepsilon) & = & \varepsilon. \end{array}$$

Homomorphisms are applied to languages as well: The homomorphic image $h(L)$ of a language L consists of all homomorphic images of all L's words:

$$h(L) = \{h(w) \mid w \in L\} = \bigcup_{w \in L} \{h(w)\}$$

Our sample homomorphism h gives

$$\begin{array}{lcl} h(b + ab + bbb) & = & h(\{b, ab, bbb\}) = \{01, 001, 010101\}, \\ h(b^*) & = & h(\{\varepsilon, b, bb, bbb, \ldots\}) = \{\varepsilon, 01, 0101, \ldots\} = (01)^*, \\ h(a^*b + bb) & = & 0^*01 + 0101. \end{array}$$

A **substitution** is a generalization of the notion of homomorphisms. A substitution assigns a language to each letter. For example,

$$\begin{aligned} s(a) &= 0 + 11, \\ s(b) &= 0^*10^* \end{aligned}$$

is a substitution from $\Sigma = \{a, b\}$ to $\Delta = \{0, 1\}$. The substitution is called **regular** if $s(a)$ is a regular language for each $a \in \Sigma$, and the substitution is called **finite** if $s(a)$ is a finite language for each $a \in \Sigma$. The substitution s above is regular but not finite, and substitution f given by

$$\begin{aligned} f(a) &= 000 + 11, \\ f(b) &= 01 + \varepsilon \end{aligned}$$

is finite. Every homomorphism is a special type of (finite) substitution where each $s(a)$ consists of a single word.

Substitution s is also applied to words by applying it to each letter separately and concatenating the results (which are languages) together:

$$s(a_1 a_2 \dots a_n) = s(a_1)s(a_2)\dots s(a_n).$$

For example, using our sample substitutions s and f we have

$$\begin{array}{lcl} f(ba) & = & (01+\varepsilon)(000+11) = 01000+0111+000+11, \\ s(aba) & = & (0+11)0^*10^*(0+11), \\ s(\varepsilon) & = & \{\varepsilon\}. \end{array}$$

The result of applying substitution s on a language L is the language consisting of all words that can be obtained by applying the substitution on L's words:

$$s(L) = \{u \mid u \in s(w) \text{ for some } w \in L\ \} = \bigcup_{w \in L} s(w).$$

For example,

$$\begin{array}{lcl} s(ba+aba) & = & s(ba) \cup s(aba) = 0^*10^*(0+11) + (0+11)0^*10^*(0+11), \\ s(ba^*) & = & 0^*10^*(0+11)^*. \end{array}$$

Theorem 34 *If L is a regular language and s a regular substitution then $s(L)$ is a regular language. The family of regular languages is (effectively) closed under regular substitutions.*

Proof. Let L be a regular language and s a regular substitution. Let r be a regular expression for L, and let r_a be a regular expression for $s(a)$ for every letter $a \in \Sigma$.

To get a regular expression r' for $s(L)$ one simply replaces every letter in the expression r by the corresponding regular expression r_a. To prove formally that the resulting regular expression r' represents the language $s(L)$, one uses mathematical induction on the size of expression r. The construction works because

$$\begin{array}{rcl} s(\{\varepsilon\}) & = & \{\varepsilon\}, \\ s(\{a\}) & = & r_a, \\ s(L_1 \cup L_2) & = & s(L_1) \cup s(L_2), \\ s(L_1 L_2) & = & s(L_1)s(L_2), \\ s(L^*) & = & s(L)^* \end{array}$$

for all $a \in \Sigma$, all languages L_1, L_2, L and all substitutions s. □

Example 35.

$$s(a^*b+bb) = s(a^*b) + s(bb) = s(a^*)s(b) + s(b)s(b) = s(a)^*s(b) + s(b)s(b) = r_a^* r_b + r_b r_b.$$

□

Corollary 36 *The family of regular languages is effectively closed under homomorphisms.*

Proof. Every homomorphism is a regular substitution. □

Example 37. Using our earlier substitutions, we have

$$\begin{array}{lcl} h((ab^*aa+b)^*+aba) & = & (0(01)^*00+01)^*+0010, \\ s(ab^*+bba) & = & (0+11)(0^*10^*)^*+0^*10^*0^*10^*(0+11). \end{array}$$

(These expressions can be simplified.) □

The closure under homomorphisms and regular substitutions can now be used to prove non-regularity of languages.

Example 38. Let us prove that

$$L = \{(ab)^p \mid p \text{ is a prime number } \}$$

is not regular. We already know that

$$L_p = \{a^p \mid p \text{ is a prime number } \}$$

is not regular, so let us reduce L into L_p. Assume that L is regular. Then also $h(L)$ is regular where h is the homomorphism

$$\begin{array}{l} h(a) = a, \\ h(b) = \varepsilon. \end{array}$$

But $h(L) = L_p$, which is not regular. Therefore L is not regular either. □

Let us move on to our next language operation, the **inverse homomorphism**. Let h be a homomorphism from Σ to Δ. The inverse homomorphism h^{-1} is defined on languages of alphabet Δ. If $L \subseteq \Delta^*$ then language $h^{-1}(L)$ consists of all words over Σ that are mapped by homomorphism h into L, i.e.,

$$h^{-1}(L) = \{w \mid h(w) \in L\}.$$

If L contains only one word we may simply write $h^{-1}(w)$ instead of $h^{-1}(\{w\})$.

Example 39. For example, using homomorphism

$$\begin{array}{l} g(a) = 01, \\ g(b) = 011, \\ g(c) = 101, \end{array}$$

we have

$$\begin{array}{lcl} g^{-1}(011101) & = & \{bc\}, \\ g^{-1}(01101) & = & \{ac, ba\}, \\ g^{-1}(010) & = & \emptyset, \\ g^{-1}((10)^*1) & = & ca^*. \end{array}$$

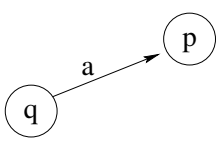
□

Theorem 40 *The family of regular languages is (effectively) closed under inverse homomorphism.*

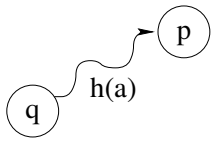
Proof. Let L be a regular language, and h a homomorphism. Let A be a DFA recognizing L. We construct a new DFA A' that recognizes the language $h^{-1}(L)$.

DFA A' has the same states as A. Also the initial state and final states are identical. Only the transitions are different: In A' input symbol a takes the automaton from state q into the same state that input string $h(a)$ takes the original automaton A from the same state q:

$$\delta'(q, a) = \delta(q, h(a)).$$



In A'



In A

Computations by A' simulate computations by A: After reading input

$$w = a_1 a_2 \dots a_n$$

the new automaton is in the same state as the original automaton is after reading input

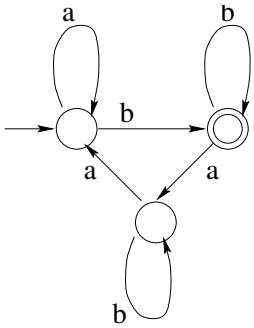
$$h(w) = h(a_1)h(a_2) \dots h(a_n).$$

Since the same states are final states in both automata, DFA A' accepts w if and only if DFA A accepts $h(w)$. In other words, the language recognized by A' is $h^{-1}(L)$. □

Example 41. Let

$$\begin{aligned} h(0) &= ab, \\ h(1) &= abb, \\ h(2) &= bab, \end{aligned}$$

and let the language L be defined by DFA



Let us construct a DFA for the language $h^{-1}(L)$. □

Constructs of type

$$h^{-1}(L) \cap R$$

are very useful, where h is a homomorphism and R is a regular language. They contain all words of R that are mapped to L by homomorphism h.

Example 42. Let us prove that the language

$$L = \{0^n 10^{2n} \mid n \geq 0\}$$

is not regular. Define homomorphism h:

$$\begin{aligned} h(a) &= 0, \\ h(b) &= 1, \\ h(c) &= 00. \end{aligned}$$

Then

$$h^{-1}(L) \cap a^*bc^* = \{a^n bc^n \mid n \geq 0\}.$$

Let us denote this language by L_1. If L is regular then also L_1 is regular. Define another homomorphism g:

$$\begin{aligned} g(a) &= a, \\ g(b) &= \varepsilon, \\ g(c) &= b. \end{aligned}$$

Then

$$g(L_1) = \{a^n b^n \mid n \geq 0\}.$$

But this language is not regular, so L_1 cannot be regular, which means that L cannot be regular either. We only used operations that preserve regularity (inverse homomorphism, intersection with a regular language, homomorphism). □

Our next language operation is called **quotient**. Let L_1 and L_2 be two languages. Their quotient L_1/L_2 is the language containing all words obtained by removing from the end of L_1's words a suffix that belongs to L_2.

$$L_1/L_2 = \{w \mid wu \in L_1 \text{ for some } u \in L_2\}.$$

Example 43. Let

$$\begin{aligned} L_1 &= abaa + aaa, \\ L_2 &= a + baa, \\ L_3 &= a^*ba^*. \end{aligned}$$

Then

$$\begin{aligned} L_1/L_2 &= aba + aa + a, \\ L_2/L_2 &= \varepsilon + ba, \\ L_3/L_1 &= a^* + a^*ba^*, \\ L_3/L_3 &= a^*. \end{aligned}$$

□

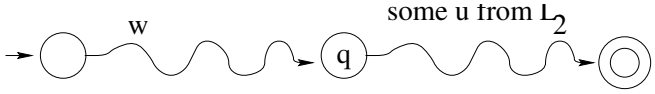
Theorem 44 *The family of regular languages is closed under quotient with arbitrary languages. In other words, if L_1 is a regular language, and L_2 any language (not necessarily even regular) then L_1/L_2 is regular.*

Proof. Let L_1 be a regular language, and A a DFA recognizing it. Let L_2 be an arbitrary language. We show that there is a DFA A' that recognizes the quotient $L' = L_1/L_2$.

DFA A' has exactly same states and transitions as A has. Also the initial state is the same. Only the final state sets F' and F are different:

$$F' = \{q \mid \exists u \in L_2 : \quad \delta(q, u) \in F\},$$

i.e., state q is made final in A' if and only if some word $u \in L_2$ takes A from state q to some original final state.

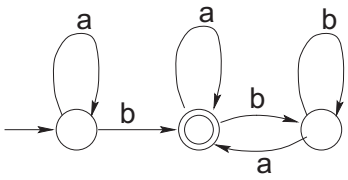


Clearly word w is accepted by A' if and only if some wu is accepted by A for some $u \in L_2$. But this is equivalent to saying that w belongs to L_1/L_2: it is obtained from word $wu \in L_1$ by deleting word $u \in L_2$ from the end. So

$$L(A') = L_1/L_2.$$

□

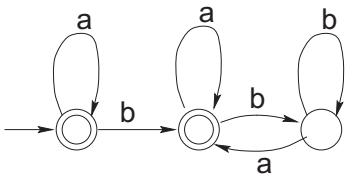
Example 45. Let L_1 be the regular language recognized by DFA



and let

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

Language L_1/L_2 is recognized by a DFA that has the same transitions as A above. We only have to figure out which states are final:



□

Remark: closure under quotient is not effective. If L_2 is some complicated language we may not have any way of determining which states to make final. We'll see later that quotient can be effectively constructed if both L_1 and L_2 are regular.

2.7 Decision algorithms

In this section we present algorithms for determining whether a given regular language is empty, finite or infinite. We also present an algorithm for determining whether two given regular languages are identical, i.e. whether they contain exactly the same words. First we make the same observation as before: it does not matter in which form the input language is represented: as a regular expression, DFA, NFA or ε-NFA. All representations are effectively equivalent.

Theorem 46 *There is an algorithm to determine if a given regular language is empty.*

Proof. Consider any regular expression r. We want to determine whether the language $L(r)$ it represents is empty or not. Observe that

- If $r = \varepsilon$ or $r = a \in \Sigma$ then $L(r)$ is not empty.
- If $r = \emptyset$ then $L(r)$ is empty.
- If $r = r_1 + r_2$ then $L(r)$ is empty if and only if both $L(r_1)$ and $L(r_2)$ are empty.
- If $r = r_1 r_2$ then $L(r)$ is empty if and only if $L(r_1)$ or $L(r_2)$ is empty.
- If $r = (r_1)^*$ then $L(r)$ is not empty.

Based on these facts one can easily write a recursive algorithm for determining whether $L(r)$ is empty:

```
Empty(r)
Begin
   if r = ε or r = a for some letter a then return( False )
   if r = ∅ then return( True )
   if r = r1 + r2 then return( Empty(r1) and Empty(r2) )
   if r = r1 r2 then return( Empty(r1) or Empty(r2) )
   if r = r1* then return( False )
End
```

□

Remark: Emptiness can be also easily determined directly from a finite automaton representation of a regular language. One just determines (using basic graph algorithms) whether there is a path in the transition diagram from the initial state to some final state.

Consider then the equivalence problem. Some regular expressions stand for the same language. For example, $a(a + ba)^*$ and $(ab + a)^*a$ both define the same language. How can we determine for any given two regular expressions correctly if their languages are identical ?

Theorem 47 *There is an algorithm to determine if two regular languages are the same.*

Proof. Let L_1 and L_2 be two regular languages (represented by finite automata or regular expressions.) Using the effective closure properties proved earlier we can construct a regular expression for their symmetric difference

$$L = (L_1 \setminus L_2) \cup (L_2 \setminus L_1) = (L_1 \cap \overline{L}_2) \cup (L_2 \cap \overline{L}_1).$$

Since $L_1 = L_2$ if and only if L is empty, we can use the algorithm for emptyness to find out whether L_1 and L_2 are the same language. □

Our last decision algorithm determines whether a given regular language contains finitely or infinitely many words. We know that every finite language is regular:

$$\{w_1, w_2, \ldots, w_n\} = L(w_1 + w_2 + \ldots + w_n).$$

Some regular languages are infinite, for example a^*.

Theorem 48 *There is an algorithm to determine if a given regular language is infinite.*

Proof. Let r be a regular expression.

- If $r = \varepsilon$, $r = a \in \Sigma$ or $r = \emptyset$ then $L(r)$ is not infinite.
- If $r = r_1 + r_2$ then $L(r)$ is infinite if and only if $L(r_1)$ or $L(r_2)$ is infinite.
- If $r = r_1 r_2$ then $L(r)$ is infinite if and only if $L(r_1)$ and $L(r_2)$ are both non-empty, and at least one of them is infinite.
- If $r = (r_1)^*$ then $L(r)$ is infinite unless $L(r_1)$ is $\emptyset$ or $\{\varepsilon\}$.

Based on these facts one can easily write a recursive algorithm for determining whether $L(r)$ is infinite. The programs for emptyness and equivalence are used as subroutines. □

Note that all effective constructions presented earlier may be used as subroutines in new algorithms.

Example 49. To test whether $L_1 \subseteq L_2$ for given regular languages L_1 and L_2, we simply test whether $L_1 \setminus L_2 = L_1 \cap \overline{L}_2 = \emptyset$. □

2.8 Myhill-Nerode theorem

Recall that a relation R on a set S is an **equivalence** relation if it is reflexive, symmetric and transitive, i.e. if

- aRa for all $a \in S$,
- aRb implies bRa, and
- aRb and bRc imply aRc.

An equivalence relation R partitions S into disjoint equivalence classes such that aRb if and only if a and b are in the same equivalence class. Let us denote the equivalence class containing a by $[a]$:

$$[a] = \{b \in S \mid aRb\}.$$

The **index** of an equivalence relation is the number of equivalence classes. The index can be finite or infinite.

Every positive integer n defines the equivalence relation $a \equiv b \pmod{n}$ on the set of integers. In an (somewhat) analogous way every language $L \subseteq \Sigma^*$ defines an equivalence relation R_L on words of Σ^*:

$$w \; R_L \; u \quad \Longleftrightarrow \quad (\forall x \in \Sigma^*) \; wx \in L \text{ if and only if } ux \in L$$

Words w and u are in the relation R_L iff exactly same extensions take them to L.

To find out whether given words w and u are in relation R_L for given language L, we have to find out for which extensions x words wx and ux belong to L. Let us denote the language of good extensions x by

$$\mathrm{Ext}(w, L) = \{x \mid wx \in L\}.$$

Then $w \; R_L \; u$ if and only if

$$\mathrm{Ext}(w, L) = \mathrm{Ext}(u, L).$$

Example 50. Let $L = aa + baa$. Then

$$\begin{aligned} \mathrm{Ext}(a, L) &= a, \\ \mathrm{Ext}(b, L) &= aa, \\ \mathrm{Ext}(ba, L) &= a, \\ \mathrm{Ext}(\varepsilon, L) &= aa + baa, \\ \mathrm{Ext}(aa, L) &= \varepsilon, \\ \mathrm{Ext}(aba, L) &= \emptyset. \end{aligned}$$

We see that words a and ba are in relation R_L with each other. In fact,

$$[a] = a + ba.$$

The relation R_L has five equivalence classes

$$\varepsilon, \quad b, \quad a + ba, \quad aa + baa \quad \text{and} \quad \overline{\varepsilon + a + b + aa + ba + baa},$$

corresponding to proper extensions

$$aa + baa, \quad aa, \quad a, \quad \varepsilon \quad \text{and} \quad \emptyset,$$

respectively. □

Example 51. Here's another example. Let $L = a^*b^*$. Then

$$\begin{aligned} \mathrm{Ext}(w, L) &= a^*b^*, && \text{for all } w \in a^*, \\ \mathrm{Ext}(w, L) &= b^*, && \text{for all } w \in a^*b^+, \\ \mathrm{Ext}(w, L) &= \emptyset, && \text{for all } w \notin a^*b^*. \end{aligned}$$

The index of R_L is three. □

Example 52. Let us find the equivalence classes of R_L for the language

$$L = \{w \in \{a,b\}^* \mid w \text{ contains equally many } a\text{'s and } b\text{'s }\}.$$

Let us denote

$$|w|_a = \text{ number of letters } a \text{ in word } w.$$

Consider languages

$$L_n = \{w \in \{a,b\}^* \mid |w|_a - |w|_b = n\}$$

for different $n \in \mathbb{Z}$. Clearly $L = L_0$. Every word belongs to exactly one L_n. Observe also that

$$L_n L_m \subseteq L_{n+m}.$$

It follows from the considerations above that if $w \in L_n$ then

$$\mathrm{Ext}(w, L) = L_{-n}.$$

Therefore $\mathrm{Ext}(w, L)$ and $\mathrm{Ext}(u, L)$ are identical if and only if w and u belong to the same set L_n. In other words, the equivalence classes of R_L are languages L_n for all $n \in \mathbb{Z}$.

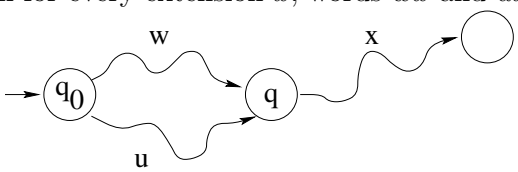
Note that in this case there are infinitely many equivalence classes. □

Theorem 53 *Language L is regular if and only if the index of R_L is finite.*

Proof. ($\Longrightarrow$) Let L be a regular language recognized by DFA A. If w and u are two words that take the automaton to the same state q, i.e.,

$$\delta(q_0, w) = \delta(q_0, u),$$

then for every extension x, words wx and ux lead to same state.



This means that wR_Lu.

Since all words leading to the same state are in relation R_L, the number of equivalence classes is at most the number of states, and therefore finite.

Note that we have in fact shown that the number of states in any DFA recognizing L is at least the index of R_L. In fact, we can make the following observations: Let us assign to every DFA A its own equivalence relation R_A by

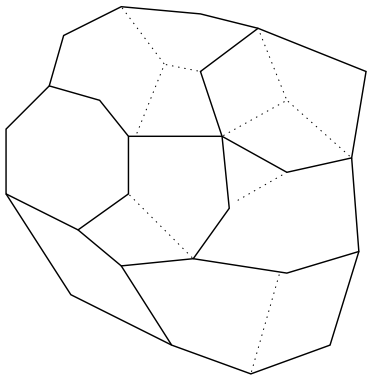
$$w \; R_A \; u \Longleftrightarrow \delta(q_0, w) = \delta(q_0, u).$$

That is, words w and u are in relation R_A if and only if they take the machine to the same state.

We proved above that if A recognizes language L then the relation R_A is a **refinement** of relation R_L. In other words,

$$w \; R_A \; u \Longrightarrow w \; R_L \; u.$$

The equivalence classes of a refinement define a finer partition of the set :



We still need to prove the other direction of the "if and only if" statement:

($\Longleftarrow$) Assume that R_L has finite index. Let L_1, L_2, ..., L_n be its equivalence classes. We construct a DFA A with n states that recognizes L. The state set of A is labeled with the equivalence classes $[w]$:

$$Q = \{L_1, L_2, \ldots, L_n\}.$$

The idea of the construction is that every input word w will lead to state $[w]$, the equivalence class containing w. The transitions are defined as follows:

$$\delta([w], a) = [wa].$$

The transitions are well defined, that is, the definition does not depend on which word w we selected from the equivalence class: If $[w] = [u]$ then $[wa] = [ua]$. (If we would have $[wa] \neq [ua]$ then for some x, $wax \in L$ and $uax \notin L$ or vice versa. But then also $[w] \neq [u]$ since the extension ax separates w and u.)

Let the initial state q_0 be $[\varepsilon]$. Then

$$\delta(q_0, w) = [w]$$

for every w. The transitions were designed that way:

$$\begin{aligned} \delta([\varepsilon], a_1a_2 \ldots a_n) &= \delta([a_1], a_2a_3 \ldots a_n) = \delta([a_1a_2], a_3 \ldots a_n) \\ &= \ldots = [a_1a_2 \ldots a_n]. \end{aligned}$$

Finally, the final states of our automaton are

$$\{[w] \mid w \in L\}.$$

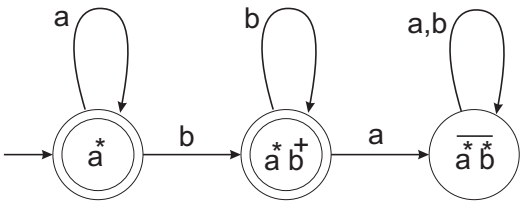
Again, the choice of w does not matter since if $[w] = [u]$ and $w \in L$ then also $u \in L$. (Otherwise extension ε would separate them.)

The automaton we constructed recognizes language L because

$$\delta(q_0, w) = [w]$$

and $[w]$ is a final state if and only if $w \in L$. This completes the proof. □

Example 54. Let us construct a DFA for $L = a^*b^*$ based on our knowledge about its equivalence relation R_L:



□

Note that the number of states of the DFA constructed in the proof is the same as the index of R_L. On the other hand, we observed before that every DFA recognizing L has at least that many states, so the DFA we constructed has the minimal number of states among all DFA recognizing L. This number of states is the index of the equivalence relation R_L.

The DFA we constructed is the **minimum state DFA** for language L. No other DFA for L has fewer states. In fact, the minimum state DFA is unique: every other DFA for L has more states:

Theorem 55 *The minimum state DFA of language L is unique (up to renaming states).*

Proof. Let A be a DFA for language L with as few states as possible, and let A_0 be the automaton constructed in the proof of Theorem 53. We want to prove that A is the same automaton as A_0, except that its states may be named differently.

The number of states in both machines is the same as the index of R_L, and therefore the automata relations R_A and R_{A_0} are identical to R_L. (They are refinements of R_L with the same number of equivalence classes as R_L.) Since the relations are the same we do not need to specify which relation R_L, R_A or R_{A_0} we mean when we talk about the equivalence class $[w]$ of word w: all three relations define identical equivalence classes.

Recall the definition of R_A: two words belong to the same equivalence class if and only if they take machine A to the same state:

$$w \; R_A \; u \iff \delta(q_0, w) = \delta(q_0, u)$$

Let us rename the states of A using the equivalence classes: For every $w \in \Sigma^*$, rename the state $\delta(q_0, w)$ as $[w]$. The naming is well defined: if $\delta(q_0, w) = \delta(q_0, u)$ then $[w] = [u]$, so the name is independent of the choice of w.

Then trivially

$$\delta([w], a) = [wa]$$

for every $w \in \Sigma^*$ and $a \in \Sigma$. But this exactly how the transitions of A_0 were defined in the proof of Theorem 53. This means the transitions of A are identical to the transitions of A_0, and the machines are identical. □

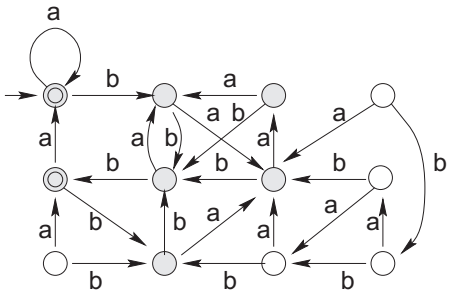
In the rest of this section we discuss the minimization process, i.e., we give an algorithm to construct the minimum state DFA that is equivalent to a given DFA.

Theorem 56 *There is an algorithm to construct the minimum state DFA for given regular language.* □

Let A be a DFA that recognizes language L. In the following we construct a minimum state DFA that is equivalent to A.

1. The first step is to remove all non-reachable states from A. State q is non-reachable if there does not exist any input word that takes automaton A to state q.

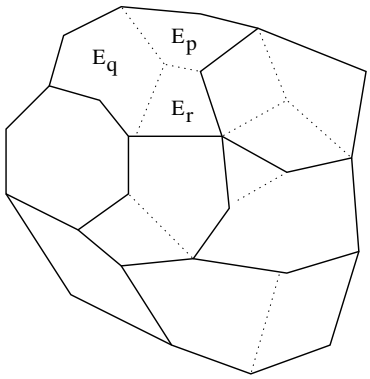
Reachable states can be found by a simple marking procedure: First mark the initial state only. Then mark all states that there is a transition into from some marked state. Continue this until no new states can be marked. All non-marked states are non-reachable and can be deleted. For example, the shaded states are reachable in the following automaton:



2. Now assume DFA A only contains reachable states. Every state q defines an equivalence class E_q of the automaton relation R_A, consisting of all words that take the machine from initial state q_0 to state q:

$$E_q = \{w \mid \delta(q_0, w) = q\}.$$

We know that R_A is a refinement of the relation R_L:

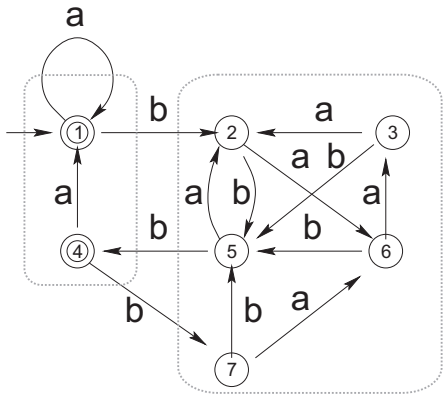


In the minimum state DFA the equivalence classes of the automaton are the same as the equivalence classes of R_L. We can combine states q and p if their classes E_q and E_p belong to the same equivalence class of R_L. We call such states **indistinguishable**. If states q and p are indistinguishable it does not make any difference whether the machine is in state q or p: Exactly same words lead to a final state from q and p. If we find all pairs of indistinguishable states and combine them we end up with a DFA A' whose equivalence relation $R_{A'}$ is identical to R_L. Such A' is the minimum state DFA for language L.

The problem is to find out if two states are indistinguishable. It is easier to find states that are distinguishable: To show that p and q can not be combined you only need to find one word x such that one of the states $\delta(q, x)$ and $\delta(p, x)$ is final and the other one is not.

Let us start classifying the states. We put states that we know are distinguishable into separate classes. Initially we have two classes of states: one contains all final states and the other one all non-final states. The empty word ε distinguishes final and non-final states.

Our sample DFA



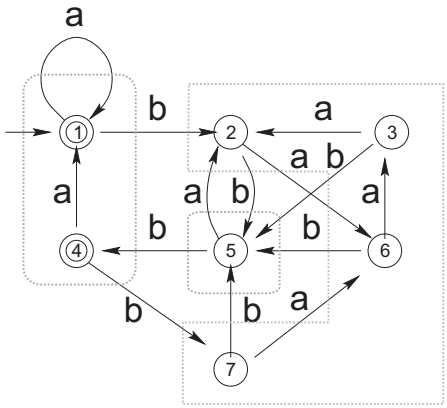
starts with classes

$$C_1 = \{1, 4\} \text{ and } C_2 = \{2, 3, 5, 6, 7\}.$$

Then we try to find an input letter that splits one of the existing classes: An input letter a splits class C_i if there are states p and q in C_i such that $\delta(p, a)$ and $\delta(q, a)$ belong to different classes. Then we namely know that p and q are distinguishable: they are separated by ax where x is the word that separated states $\delta(p, a)$ and $\delta(q, a)$.

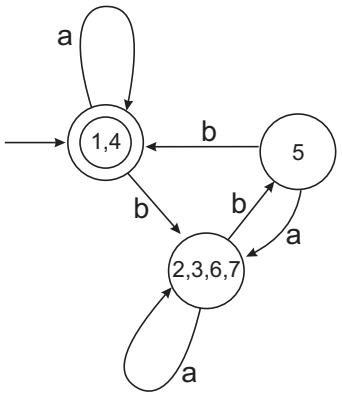
In our example letter b splits class $C_2 = \{2, 3, 5, 6, 7\}$ because $\delta(5, b) = 4 \in C_1$ and all other $\delta(2, b) = 5$, $\delta(3, b) = 5$, $\delta(6, b) = 5$ and $\delta(7, b) = 5$, are in C_2. So we split C_2 into two classes: one class contains all states that go to C_1 with input b; the states in the other class go to C_2 with input b. We obtain new classes

$$D_1 = \{1, 4\}, D_2 = \{5\} \text{ and } D_3 = \{2, 3, 6, 7\}.$$



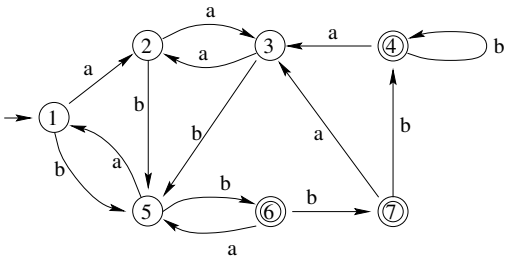
We repeat the process until no class be split any more. Then we can construct an automaton by combining all states that belong to same class. The transitions are uniquely defined: otherwise we could split the classes further.

In our example the three classes D_1, D_2 and D_3 cannot be split any further, so we know that the equivalent minimum state DFA has three states:



Classes that contain final states are made final; class that contains the original initial state is the initial state of the new DFA.

Example 57. Let us construct the minimum state DFA that is equivalent to



□

Remark: The minimization process and all theorems of this section only apply to deterministic automata. There can be several different minimal state non-deterministic automata for the same language.

3 Context-free languages

Superficially thinking, one might view modern computers as deterministic finite automata: Computers have finite memory, so there are only finitely many states the computer can be at, and the previous state of the computer determines the next state, so the machine has deterministic state transitions. This approach is, however, not reasonable. First of all, the number of states is indeed finite but that finite number is astronomical. And even worse: the memory of the computer is extendible, so the approach is not even theoretically correct. A more accurate mathematical model is needed.

Our next step into the direction of "correcting" the notion of computation is to overcome the limitations of finite memory by introducing new models that allow arbitrarily large memory. The memory can be organized and accessed in different ways. Our next model, context-free languages, assumes an infinite memory that is organized as a stack (=last-in-first-out memory). As we are going to see, this does not yet capture the full power of computation, but it provides a natural generalization of regular languages. Every regular language is a context-free language, but also some non-regular languages are included. For example $\{a^n b^n \mid n \geq 0\}$ is a context-free language. On the

other hand there are many simple languages which are not context-free, for example $\{a^n b^n c^n \mid n \geq 0\}$.

Context-free languages have applications in compiler design (parsers). The syntax of programming languages is often given in the form of context-free grammar, or equivalent Backus-Naur form (BN-form).

3.1 Context-free grammars

Analogously to regular languages, we have different ways of representing context-free languages. We first consider generative devices, so-called **context-free grammars**.

Example 58. Let us begin with an example of a context-free grammar. It uses **variables** A and B, and **terminal** symbols a and b. We have following **productions** (also called **rewriting rules**):

$$\begin{array}{lcl} A & \longrightarrow & AbA \\ A & \longrightarrow & B \\ B & \longrightarrow & aBa \\ B & \longrightarrow & b \end{array}$$

In a word containing variables, and possibly terminals, one can replace any variable with the word on the right hand side of a production for that variable. For example, in the word

$$aBabAbA$$

one can, for example, replace the first occurrence of variable A by AbA, obtaining the word

$$aBab\ AbA\ bA.$$

Then one can decide, for example, to replace the variable B by b, which gives the word

$$a\ b\ abAbAbA.$$

Rewriting one variable is called a **derivation step** and we denote

$$aBabAbA \Rightarrow aBabAbAbA \Rightarrow ababAbAbA.$$

Derivation is non-deterministic: usually one has many choices of how to proceed. We can continue the derivation as long as there exist variables in the word. Once the word contains only terminal symbols the derivation terminates.

One variable is called the **start symbol**. In the case of this example, let A be the start symbol. All derivations start with the start symbol A, i.e., initially the word is A. The language defined by the grammar contains all strings of terminal symbols that can be obtained from the start symbol by applying the productions on the variables. For example, word $aabaababa$ is in the language defined by our grammar since it is derived by

$$A \Rightarrow AbA \Rightarrow BbA \Rightarrow aBabA \Rightarrow aaBaabA \Rightarrow aabaabA \Rightarrow aabaabB \Rightarrow aabaabaBa \Rightarrow aabaababa.$$

On the other hand, word aaa is not in the language because the only way to produce letter a is to use the production $B \longrightarrow aBa$, and it creates two a's, so the number of a's has to be even. □

Here is a precise definition of context-free grammars. A context-free grammar $G = (V, T, P, S)$ consists of

- two disjoint finite alphabets, V and T, containing variables (=nonterminals) and terminals, respectively,
- a finite set P of productions of the form

$$A \longrightarrow \alpha$$

  where $A \in V$ is a variable, and $\alpha \in (V \cup T)^*$ is a word of terminals and variables,
- a start symbol $S \in V$.

For any $\alpha, \beta \in (T \cup V)^*$ we denote

$$\alpha \Rightarrow \beta$$

if β is obtained from α by rewriting one variable in α using some production from P, that is, if $\alpha = uAv$ and $\beta = uwv$ for some $u, v, w \in (V \cup T)^*$ and $A \in V$, and $A \longrightarrow w$ is in P. This is called a derivation step. The reflective, transitive closure of $\Rightarrow$ is denoted by $\Rightarrow^*$, so $\alpha \Rightarrow^* \beta$ iff there is a sequence of derivation steps

$$\alpha = \alpha_0 \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n = \beta$$

that starts with α and leads to β. The number n of derivation steps can be zero, so always $\alpha \Rightarrow^* \alpha$. We say that α **derives** β. If we want to emphasize that a derivation takes n steps, we may use the notation $\Rightarrow^n$.

A word $\alpha \in (V \cup T)^*$ is called a **sentential form** if it can be derived from the start symbol S, i.e., if

$$S \Rightarrow^* \alpha.$$

The language $L(G)$ generated by grammar G consists of all sentential forms that contain only terminals. In other words,

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

A language is called a **context-free language** if it is $L(G)$ for some context-free grammar G.

Example 59. Consider the grammar $G = (V, T, P, S)$ where $V = \{S\}$, $T = \{a, b\}$, and P contains the productions

$$\begin{array}{rcl} S & \longrightarrow & aSb, \\ S & \longrightarrow & \varepsilon. \end{array}$$

To make notations shorter we may use the following convention: productions of the same variable may be combined on one line, separated by symbol |. So we may write

$$S \longrightarrow aSb \mid \varepsilon$$

We easily see that $L(G) = \{a^n b^n \mid n \geq 0\}$. □

Example 60. Consider the grammar $G = (V, T, P, E)$ where $V = \{E, N\}$, $T = \{+, *, (,), 0, 1\}$, and P contains the following productions:

$$\begin{array}{rcl} E & \longrightarrow & E + E \mid E * E \mid (E) \mid N \\ N & \longrightarrow & 0N \mid 1N \mid 0 \mid 1 \end{array}$$

For example, all the following words are in the language $L(G)$:

$$0$$
$$0 * 1 + 111$$
$$(1 + 1) * 0$$
$$(1 * 1) + (((0000)) * 1111)$$

For instance, $(1 + 1) * 0$ is derived by

$$E \Rightarrow E * E \Rightarrow (E) * E \Rightarrow (E + E) * E \Rightarrow^* (N + N) * N \Rightarrow^* (1 + 1) * 0.$$

- From the variable N all nonempty strings of symbols 0 and 1 can be derived.
- From the variable E one can derive all well-formed arithmetic expressions containing operators '*' and '+', parentheses, and strings of 0's and 1's derived from variable N.

Here are some words that are **not** in the language $L(G)$:

$$1(1$$
$$()$$
$$1 * 1 * 1 * 1*$$

□

Example 61. Grammar $G = (V, T, P, S)$ where $V = \{S\}$, $T = \{a, b\}$ and P contains productions

$$S \longrightarrow aSbS \mid bSaS \mid \varepsilon.$$

Let us determine $L(G)$. Clearly every word that belongs to $L(G)$ must contain equally many a's and b's. (Every production adds the same number of a's and b's to the sentential form.) Let us next prove that $L(G)$ contains all words $w \in \{a, b\}^*$ with equally many a's and b's. Let

$$L = \{w \in \{a, b\}^* \mid \ w \text{ has equally many } a\text{'s and } b\text{'s } \}.$$

We observe first that any non-empty word $w \in L$ can be written as

$$w = a\ u\ b\ v \qquad \text{or} \qquad w = b\ u\ a\ v$$

for some $u, v \in L$. Let us prove this fact. Assume $w \in L$ and w starts with letter a (if it start with letter b the proof is symmetric.) Let au be the longest prefix of w with more a's than b's. The next letter coming after au has to be b because aua has more a's than b's and is longer than au. So w starts $a\ u\ b \ldots$. In the word au the number of a's is exactly one greater than the number of b's: If the difference would be more than one then also prefix $a\ u\ b$ would contain more a's than b's, and it is longer than au. So u has equally many a's and b's. The suffix v that follows $a\ u\ b$ has also equally many a's and b's. So $w = a\ u\ b\ v$ and $u, v \in L$.

Now we are ready to prove that every word $w \in L$ is in $L(G)$. We use induction on the length of w.

1° (basis) If $|w| = 0$ then $w = \varepsilon \in L(G)$ because $S \longrightarrow \varepsilon$ is in P.

2° (inductive step) Assume $|w| > 0$, and we know that every shorter word with equally many a's and b's can be derived from S. We demonstrated above that

$$w = a\ u\ b\ v \qquad \text{or} \qquad w = b\ u\ a\ v$$

where u and v are words that contain equally many a's and b's. According to the inductive hypothesis,

$$S \Rightarrow^* u \qquad \text{and} \qquad S \Rightarrow^* v.$$

If $w = a\ u\ b\ v$ then we can derive w as follows:

$$S \Rightarrow a\ S\ b\ S \Rightarrow^* a\ u\ b\ v.$$

The case $w = b\ u\ a\ v$ is similar. In either case, we have a derivation for w so $w \in L(G)$.

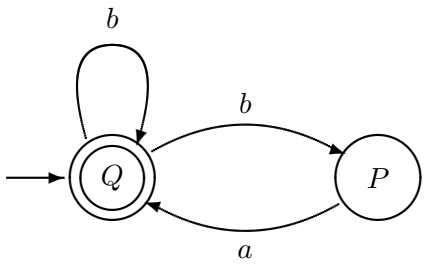
For example, *ababba* has the following derivation:

$$S \Rightarrow aSbS \Rightarrow abSaSbS \Rightarrow abaSbS \Rightarrow ababS \Rightarrow ababbSaS \Rightarrow ababbaS \Rightarrow ababba.$$

□

Let us prove next that every regular language can be generated by a context-free grammar.

Example 62. Let L be the regular language recognized by the NFA A



Let us construct a grammar G that generates the same language L. The variables of G are the states $\{Q, P\}$ and the terminals are $\{a, b\}$. The start symbol is the initial state Q. Every transition of A is simulated by a production:

$$\begin{array}{lcl} Q & \longrightarrow & bQ \\ Q & \longrightarrow & bP \\ P & \longrightarrow & aQ \end{array}$$

Whenever the automaton reads a letter and goes to state $q = Q$ or P, the grammar produces the same letter and changes the variable to indicate the current state q.

Each sentential form consists of a terminal word w followed by one variable. Clearly, word wq is a sentential form if and only if the automaton can go to state q reading input w. For example, $bbaQ$ is a sentential form because the machine can read bba ending up in state Q:

$$Q \Rightarrow bQ \Rightarrow bbP \Rightarrow bbaQ.$$

To terminate the simulation we add the transition

$$Q \longrightarrow \varepsilon.$$

This is correct because Q is a final state: the generation is terminated iff the corresponding computation by A is accepting. For example, when accepting the word bab, the automaton is in states Q, P, Q, Q, in this order. The corresponding derivation by the grammar is

$$Q \Rightarrow bP \Rightarrow baQ \Rightarrow babQ \Rightarrow bab.$$

It is easy to prove (using induction on the length of the terminal word) that $L(G) = L$. □

The analogous construction can be done on any NFA, proving the following theorem.

Theorem 63 *Every regular language is (effectively) context-free.*

Proof. Let L be a regular language and let $A = (Q, \Sigma, \delta, q_0, F)$ be an NFA such that $L = L(A)$. An equivalent grammar is $G = (V, T, P, S)$ where

$$\begin{aligned} V &= Q, \\ T &= \Sigma, \\ S &= q_0, \end{aligned}$$

and P contains a production $q \longrightarrow ap$ for all $q, p \in Q$ and $a \in \Sigma$ such that $p \in \delta(q, a)$, and a production $q \longrightarrow \varepsilon$ for every $q \in F$. This grammar only derives sentential forms that contain at most one variable, and the variable must appear as the last symbol of the form. Moreover, using a straightforward induction on the length of w one sees that $S \Rightarrow^* wq$ for $w \in \Sigma^*$ and $q \in Q$ if and only if $q \in \delta(q_0, w)$. Hence $S \Rightarrow^* w$ for $w \in \Sigma^*$ if and only if $w \in L(A)$. □

A grammar is called **right linear** if all productions are of the forms

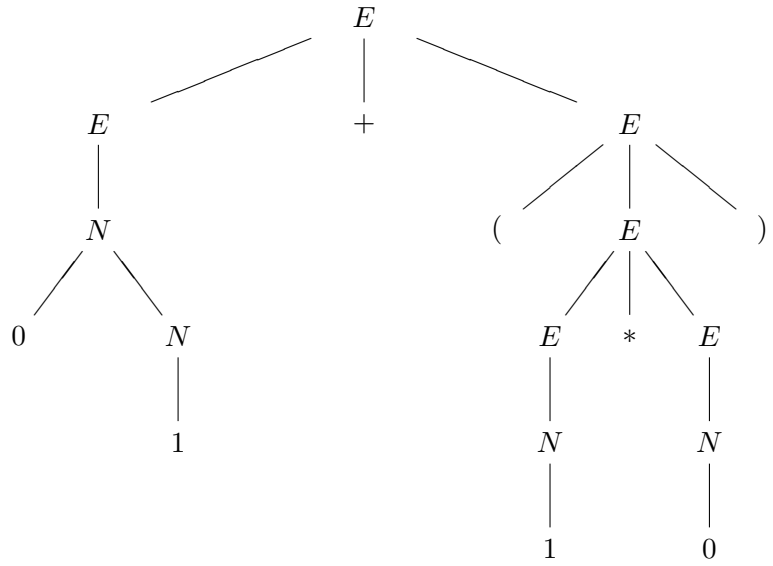
$$\begin{aligned} A &\longrightarrow wB, \text{ and} \\ A &\longrightarrow w \end{aligned}$$

for strings of terminals $w \in T^*$ and variables $A, B \in V$. In other words, the right hand side of a production may contain only one variable, and it has to be the last symbol.

The construction in the proof of Theorem 63 always produces a right linear grammar, so every regular language is generated by a right linear grammar. The converse is also easily seen true: a language generated by a right linear grammar is always regular. This observation provides yet another equivalent way to define regular languages, in terms of right-linear grammars.

3.2 Derivation trees

Derivations by context-free grammars can be visualized using **derivation trees**, also called **parse trees**. Here is a derivation tree for the grammar of Example 60:



All the interior nodes of the tree are labeled using variables. The label of the root is the start symbol E of the grammar. The children of a node are the symbols on the right hand side of some production for the parent's variable, in the correct order. The tree is ordered: the order of the children matters. For example, the labels of the children from left to right of the root are E, $+$ and E, corresponding to the production

$$E \longrightarrow E + E$$

in the grammar. Note that if you read the symbols on the leaves from left to right you get the word $01 + (1 * 0)$ which is a valid sentential form for the grammar. This is called the yield of the tree.

More generally, a derivation tree for a grammar $G = (V, T, P, S)$ is a directed, ordered tree whose nodes are labeled by symbols from the set

$$V \cup T \cup \{\varepsilon\}$$

in such a way that

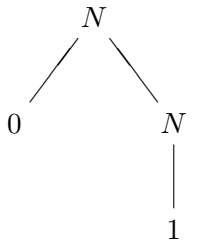
- the interior nodes are labeled by variables, i.e. elements of V,
- the root is labeled by the start symbol S,
- if $X_1, X_2, \ldots, X_n$ are the labels of the children of a node labeled by variable A, ordered from left to right, then
$$A \longrightarrow X_1 X_2 \ldots X_n$$
is a production belonging to P. If $n \geq 2$ then no X_i may be ε.

Note that the label ε is needed to allow productions $A \longrightarrow \varepsilon$ in the derivation tree.

If one reads the labels of the **leaves** from left to right, one obtains a word over $(V \cup T)^*$. This word is called the **yield** of the derivation tree. More precisely, the yield is obtained by traversing the tree in the depth-first-order, always processing the children of each node from left-to-right, and recording the leaves encountered during the traversal.

A **subtree** of a derivation tree is the tree formed by some node and all its descendants. A subtree is like a derivation tree except that its root may have a different label than the start symbol. In general, a tree that is like a derivation tree except that the root is labeled by a variable A instead of the start symbol S, is called an **A-tree**.

In our example,



is a subtree, an N-tree.

It is clear that any derivation $S \Rightarrow^* \alpha$ in the grammar has a corresponding derivation tree whose yield is the sentential form α. Conversely, every derivation tree represents derivations according to the grammar. Note, however, that the tree does not specify in which order parallel variables are rewritten, so the derivation obtained from a given tree is not unique. For example, the derivation tree in the beginning of the section represents the derivation

$$\begin{aligned}\underline{E} &\Rightarrow \underline{E}+E \Rightarrow \underline{N}+E \Rightarrow 0\underline{N}+E \Rightarrow 01+\underline{E} \Rightarrow 01+(\underline{E}) \Rightarrow 01+(\underline{E}*E)\\ &\Rightarrow 01+(\underline{N}*E) \Rightarrow 01+(1*\underline{E}) \Rightarrow 01+(1*\underline{N}) \Rightarrow 01+(1*0)\end{aligned}$$

where we have underlined in every sentential form the variable being rewritten in the next derivation step. But the same tree also represents the derivation

$$\begin{aligned}\underline{E} &\Rightarrow E+\underline{E} \Rightarrow E+(\underline{E}) \Rightarrow E+(E*\underline{E}) \Rightarrow E+(E*\underline{N}) \Rightarrow E+(\underline{E}*0) \Rightarrow E+(\underline{N}*0)\\ &\Rightarrow \underline{E}+(1*0) \Rightarrow \underline{N}+(1*0) \Rightarrow 0\underline{N}+(1*0) \Rightarrow 01+(1*0).\end{aligned}$$

Different derivations that correspond to the same tree only differ in the order in which variables are rewritten. A derivation is called **leftmost** if at every derivation step the leftmost variable of the sentential form is rewritten. The first derivation above is leftmost. The second derivation above is **rightmost**: One always rewrites the rightmost variable.

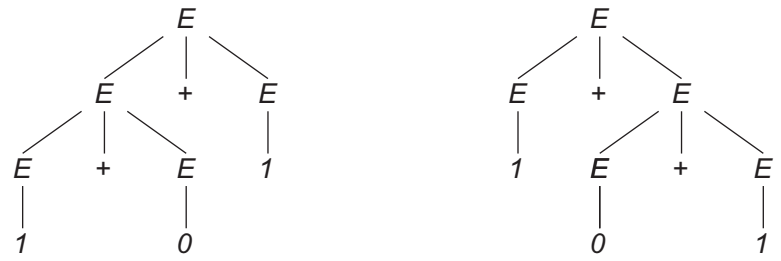
Every derivation tree defines a unique leftmost derivation, and a unique rightmost derivation. The leftmost derivation rewrites the variables in the order in which the depth first traversal of the tree from left to right encounters them. The rightmost derivation corresponds to the depth first traversal from right to left. So we have a one-to-one correspondence between derivation trees and leftmost (and rightmost) derivations.

Finding a derivation for $w \in L(G)$ (or equivalently, finding a derivation tree with yield α) is called **parsing**. Later, in Section 3.7, we discuss an efficient parsing algorithm.

Note that even though every derivation tree defines a unique leftmost derivation, some words may have several leftmost or rightmost derivations. This happens if a word is the yield of more than one derivation tree. For example, in our sample grammar the word $1+0+1$ has the following two leftmost derivations

$$\begin{aligned}E \Rightarrow E+E &\Rightarrow E+E+E \Rightarrow 1+E+E \Rightarrow 1+0+E \Rightarrow 1+0+1, \text{ and}\\ E \Rightarrow E+E &\Rightarrow 1+E \Rightarrow 1+E+E \Rightarrow 1+0+E \Rightarrow 1+0+1,\end{aligned}$$

corresponding to the derivation trees



A context-free grammar G is called **ambiguous** if some word has more than one leftmost derivation (equivalently: more than one derivation tree). Otherwise the grammar is **unambiguous**. In unambiguous grammars there is essentially just one way to parse each word in $L(G)$.

Our sample grammar is ambiguous. However, one can find an equivalent grammar that is not ambiguous:

$$\begin{array}{rcl} E & \longrightarrow & P \mid E + P \\ P & \longrightarrow & A \mid P * A \\ A & \longrightarrow & N \mid (E) \\ N & \longrightarrow & 0N \mid 1N \mid 0 \mid 1 \end{array}$$

(Both non-ambiguity and equivalence to the original grammar can be proved using mathematical induction on the length of the word.)

For some context-free languages there exist only ambiguous grammars. Such languages are called **inherently ambiguous**. Note that ambiguity is a property of a grammar, while inherent ambiguity is a property of a language.

There does not exist an algorithm that would determine if a given context-free grammar G is ambiguous or unambiguous. (This will be proved in the last part of these notes.) For individual grammars one can come up with ad hoc proofs of unambiguity.

Example 64. The grammar $G_1 = (\{S\}, \{a, b\}, P_1, S)$ where P_1 contains the productions

$$S \longrightarrow aSb \mid aaS \mid \varepsilon$$

is ambiguous because the word $aaab$ has two different leftmost derivations

$$S \Rightarrow aaS \Rightarrow aaaSb \Rightarrow aaab \qquad \text{and} \qquad S \Rightarrow aSb \Rightarrow aaaSb \Rightarrow aaab.$$

The language $\{a^{2k+n}b^n \mid k, n \geq 0\}$ it generates is not inherently ambiguous because it is generated by the equivalent unambiguous grammar $(\{S, A\}, \{a, b\}, P_1', S)$ with productions

$$\begin{array}{rcl} S & \longrightarrow & aSb \mid A, \\ A & \longrightarrow & aaA \mid \varepsilon. \end{array}$$

□

Example 65. The grammar $G_2 = (\{S\}, \{a, b\}, P_2, S)$ with the productions

$$S \longrightarrow aSSb \mid ab$$

is unambiguous. Consider two different leftmost derivations D_1 and D_2. Let us prove that the words they derive must be different.

Let $wS\alpha$ be the sentential form right before the first derivation step where D_1 and D_2 differ. Here, w is a terminal word and α may contain both terminal and non-terminal symbols.

Derivations D_1 and D_2 apply different productions to S so after the next derivation step we obtain the sentential forms $waSSb\alpha$ and $wab\alpha$. The words that can be derived from these two sentential forms begin $waa\ldots$ and $wab\ldots$, respectively, and are therefore different. □

Example 66. Let us determine whether the grammar $G_3 = (\{S\}, \{a, b\}, P_3, S)$ with productions

$$S \longrightarrow aSb \mid bS \mid \varepsilon$$

is ambiguous or unambiguous. □

Example 67. The grammar $G_4 = (\{S\}, \{a, b\}, P_4, S)$ with productions

$$S \longrightarrow aSbS \mid bSaS \mid \varepsilon$$

is ambiguous. For example, the word *abab* has two different leftmost derivations. The corresponding language

$$L(G_4) = \{w \in \{a, b\}^* \mid \ w \text{ contains equally many } a\text{'s and } b\text{'s } \}$$

is not inherently ambiguous because it is generated by the unambiguous grammar $(\{S, P, N\}, \{a, b\}, P, S)$ with productions

$$\begin{array}{lcl} S & \longrightarrow & aPbS \mid bNaS \mid \varepsilon, \\ P & \longrightarrow & aPbP \mid \varepsilon, \\ N & \longrightarrow & bNaN \mid \varepsilon. \end{array}$$

□

Also inherent ambiguity is an undecidable property. In other words, there does not exist an algorithm for determining if the language $L(G)$ generated by given context-free grammar G is inherently ambiguous or not. Here is an example of an inherently ambiguous CF language (we skip the proof):

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

3.3 Simplifying context-free grammars

Grammars G and G' are called **equivalent** if $L(G) = L(G')$. Certain types of productions are undesirable in a grammar, and we would like to find an equivalent grammar that does not contain such productions. In the following we want to remove productions of the forms

$$\begin{array}{lcll} A & \longrightarrow & \varepsilon & (\text{"}\varepsilon\text{-production"}), \text{ and} \\ A & \longrightarrow & B & (\text{"unit production"}), \end{array}$$

where A and B are variables. We also want to remove symbols that are unnecessary in the sense that they are not used in any terminating derivations. The simplification is done in the following order:

1. Remove ε -productions.

2. Remove unit productions.

3. Remove variables that do not derive any terminal strings.

4. Remove symbols that cannot be reached from the start symbol.

All four steps are effective, i.e., doable by an algorithm. We use the following grammar as a running example:

$$\begin{array}{lcl} S & \longrightarrow & AC \mid aB \mid AD \\ A & \longrightarrow & \varepsilon \mid ab \mid S \\ B & \longrightarrow & Aa \mid AB \\ C & \longrightarrow & AAa \mid \varepsilon \\ D & \longrightarrow & EbD \\ E & \longrightarrow & bb \end{array}$$

Step 1. Remove ε-productions.

- If $\varepsilon \notin L(G)$ we can construct an equivalent grammar G' that does not have any ε-productions.
- If $\varepsilon \in L(G)$ we can construct a grammar G' that does not have any ε-productions, and $L(G') = L(G) \setminus \{\varepsilon\}$.

Let us call a variable X **nullable** if $X \Rightarrow^* \varepsilon$.

(a) Find all nullable variables using a marking procedure: First, mark nullable all variables X that have a production $X \longrightarrow \varepsilon$. Then mark variables Y that have productions $Y \longrightarrow \alpha$ where α is a string of variables that have all been already marked nullable. Repeat this until no new variables can be marked. It is clear that all nullable variables are found. Note that $\varepsilon \in L(G)$ if and only if the start symbol S is nullable.

In the sample grammar above, variables A, C and S are nullable.

(b) Construct a new set P' of productions as follows: For each original production

$$A \longrightarrow X_1 X_2 \ldots X_n$$

in P, where all $X_i \in V \cup T$, we include in P' all productions

$$A \longrightarrow \alpha_1 \alpha_2 \ldots \alpha_n$$

where

- $\alpha_i = X_i$ if X_i is not a nullable variable,
- $\alpha_i = \varepsilon$ or $\alpha_i = X_i$ if X_i is a nullable variable,
- $\alpha_1 \alpha_2 \ldots \alpha_n \neq \varepsilon$.

If several X_i are nullable, all combinations of $\alpha_i = \varepsilon$ and X_i are used. One original production can result in up to 2^k new productions if the righthand-side contains k nullable variables. In our example, the new productions will be

$$\begin{array}{lcl} S & \longrightarrow & AC \mid C \mid A \mid aB \mid AD \mid D \\ A & \longrightarrow & ab \mid S \\ B & \longrightarrow & Aa \mid a \mid AB \mid B \\ C & \longrightarrow & AAa \mid Aa \mid a \\ D & \longrightarrow & EbD \\ E & \longrightarrow & bb \end{array}$$

There are no ε-productions. The language generated by the new grammar is $L(G) \setminus \{\varepsilon\}$.

The construction works because the empty string ε is directly "plugged in" the righthand-side of the production, whenever there is a variable that can produce ε. When we use that production during a derivation, we decide at that point whether the nullable variable on the right hand side will be used to produce an empty string, or a non-empty string. For example, an original derivation

$$S \Rightarrow AC \Rightarrow abC \Rightarrow ab$$

is replaced by the more direct derivation

$$S \Rightarrow A \Rightarrow ab.$$

Since C derived ε in the old derivation, we used $S \longrightarrow A$ instead of $S \longrightarrow AC$.

Step 2.

Remove unit productions. Assume we have removed all ε-productions from P. The righthand side of every production in P is some non-empty word. Let us find an equivalent grammar that does not contain ε-productions or unit productions. The idea is similar to Step 1: We anticipate all possible unit derivations, and plug them in directly into the productions.

(a) For every variable A, find all variables B, such that

$$A \Rightarrow^* B.$$

All such variable-to-variable derivations $A \Rightarrow^* B$ can be found using a reachability algorithm in the directed graph whose vertices are the variables and there is an edge from A to B iff $A \longrightarrow B$ is in P. Clearly, $A \Rightarrow^* B$ iff there is a path in the graph from A to B. In our example, after the simplification step 1, the unit productions are

$$\begin{array}{lcl} S & \longrightarrow & C \mid A \mid D \\ A & \longrightarrow & S \\ B & \longrightarrow & B \end{array}$$

so we have the following letter-to-letter derivations:

$$\begin{array}{lcl} S & \Rightarrow^* & S \mid C \mid A \mid D \\ A & \Rightarrow^* & A \mid S \mid C \mid D \\ B & \Rightarrow^* & B \\ C & \Rightarrow^* & C \\ D & \Rightarrow^* & D \\ E & \Rightarrow^* & E \end{array}$$

(b) For every pair of variables X and Y, $X \neq Y$, such that $X \Rightarrow^* Y$ and $Y \Rightarrow^* X$, we can remove one of the variables: They both derive exactly the same words. So we trim P by replacing Y by X everywhere, both left- and righthand sides of all productions.

In our example we can remove A since it derives same words as S. This leaves us with a smaller number of variables:

$$\begin{array}{lcl} S & \longrightarrow & SC \mid C \mid S \mid aB \mid SD \mid D \mid ab \\ B & \longrightarrow & Sa \mid a \mid SB \mid B \\ C & \longrightarrow & SSa \mid Sa \mid a \\ D & \longrightarrow & EbD \\ E & \longrightarrow & bb \end{array}$$

(c) Now we construct P' from the trimmed set P. For every non-unit production

$$Y \longrightarrow \alpha$$

in P and every variable X such that $X \Rightarrow^* Y$, we include in P' the non-unit production

$$X \longrightarrow \alpha.$$

We obtain an equivalent grammar whose derivations may be shorter: An original derivation

$$\beta X \gamma \Rightarrow^* \beta Y \gamma \Rightarrow \beta \alpha \gamma$$

that used unit productions $X \Rightarrow^* Y$ is replaced by a single derivation step

$$\beta X \gamma \Rightarrow \beta \alpha \gamma.$$

In our example we end up with the following productions:

$$\begin{array}{lcl} S & \longrightarrow & SC \mid aB \mid SD \mid ab \mid SSa \mid Sa \mid a \mid EbD \\ B & \longrightarrow & Sa \mid a \mid SB \\ C & \longrightarrow & SSa \mid Sa \mid a \\ D & \longrightarrow & EbD \\ E & \longrightarrow & bb \end{array}$$

Step 3. Remove variables that do not derive any terminal strings.

Such variables can be simply removed together with all productions containing the variable on either side of the production. This does not effect the generated language since the variable is not used in any terminating derivation.

To find variables that do generate some terminal string, we apply similar marking procedure as with ε-productions: First we mark all variables X such that there is a production

$$X \longrightarrow w$$

in P, where $w \in T^*$ contains only terminals. Then we mark variables Y such that there is a production

$$Y \longrightarrow \alpha$$

in P where α contains only terminals, and variables that have already been marked. This is repeated until no more variables can be marked. The variables that have **not** been marked at the end can be removed.

In our sample grammar, variable D does not derive any terminal string and can be removed. We obtain the following grammar

$$\begin{array}{lcl} S & \longrightarrow & SC \mid aB \mid ab \mid SSa \mid Sa \mid a \\ B & \longrightarrow & Sa \mid a \mid SB \\ C & \longrightarrow & SSa \mid Sa \mid a \\ E & \longrightarrow & bb \end{array}$$

Step 4. Remove symbols (variables and terminals) that are not reachable from the start symbol S.

We use a similar marking procedure: Initially mark the start symbol S. Then mark all symbols appearing on the right hand side of a production for a variable that has been already marked. Repeat until no new symbols can be marked. All symbols that have **not** been marked can not be reached from the initial symbol, and they can be removed.

Note that it is important that step 3 is executed before step 4, not the other way around: Step 3 may introduce new unreachable symbols!

In our example, variable E is unreachable and can be removed. (But it became unreachable only after we removed D!) We end up with the grammar

$$\begin{array}{lcl} S & \longrightarrow & SC \mid aB \mid ab \mid SSa \mid Sa \mid a \\ B & \longrightarrow & Sa \mid a \mid SB \\ C & \longrightarrow & SSa \mid Sa \mid a \end{array}$$

After steps 1,2,3 and 4 we have found an equivalent (except the possible loss of the the empty word ε) grammar that does not have ε-productions, unit productions, or useless symbols.

If the empty word was in the original language and we want to include it in the new grammar we may introduce a **nonrecursive start symbol**. That is, we add a new variable S', make it the start symbol, and add the productions

$$\begin{array}{lcl} S' & \longrightarrow & \alpha, \text{ for every production } S \longrightarrow \alpha \text{ from the original start symbol } S, \\ S' & \longrightarrow & \varepsilon \end{array}$$

where S is the old start symbol. The new grammar is equivalent to the original one, and it has only one ε-production that can be used only once.

Example 68. Let us simplify the following grammar by removing ε-productions, unit productions and useless symbols:

$$\begin{array}{lcl} S & \longrightarrow & Aa \mid CbDS \\ A & \longrightarrow & \varepsilon \mid BA \\ B & \longrightarrow & AA \\ C & \longrightarrow & a \\ D & \longrightarrow & aAD \end{array}$$

□

In many proofs it is useful to be able to assume that the productions in the grammar are of some restricted, simple form. Several such simplified types of grammars are known to be equivalent to general context-free grammars. They are called **normal forms** of grammars.

A grammar G is said to be in the **Chomsky normal form** (or CNF) if all productions are of the forms

$$\begin{aligned} A &\longrightarrow BC, \text{ or} \\ A &\longrightarrow a \end{aligned}$$

where A, B and C are variables, and a is a terminal. Righthand-sides of productions consist of two variables, or one terminal.

Theorem 69 *Every context-free language L without ε is generated by a grammar that is in the Chomsky normal form. An equivalent CNF grammar can be constructed effectively.*

Proof. Let G be a grammar that generates L. First we use the simplification steps 1, 2, 3 and 4 above to find an equivalent grammar G' that does not contain ε-productions or unit productions. Because there are no unit productions, all productions with one symbol on the right hand side are terminal productions $A \longrightarrow a$, i.e., they are already in the Chomsky normal form.

We only need to work on productions

$$A \longrightarrow X_1 X_2 \dots X_n$$

where $n \geq 2$, and $X_i \in V \cup T$. For every terminal symbol a that appears on the right hand side we introduce a new variable V_a. If $X_i = a$ we replace X_i by V_a. After this is done for all terminals, the right hand side of the production contains only variables. For example, the production

$$S \longrightarrow cAbbS$$

will be replace by the production

$$S \longrightarrow V_c A V_b V_b S$$

where V_b and V_c are new variables.

Then we add new terminal productions

$$V_a \longrightarrow a$$

for all terminals a. The only word derivable from V_a is a. Note that these productions are in the Chomsky normal form.

Clearly the new grammar is equivalent to the original one: Every application of the original production

$$S \longrightarrow cAbbS$$

is replaced by a sequence of derivation steps that first uses production

$$S \longrightarrow V_c A V_b V_b S$$

and then applies productions

$$V_c \longrightarrow c \text{ and } V_b \longrightarrow b$$

to replace all occurrences of variables V_c and V_b by terminals c and b.

So far we have constructed an equivalent grammar whose productions are of forms

$$\begin{array}{lcl} A & \longrightarrow & B_1B_2\ldots B_n, \text{ and} \\ A & \longrightarrow & a \end{array}$$

where $n \geq 2$, and $A, B_1, B_2, \ldots, B_n$ are variables, and a denotes a terminal symbol.

The only productions that are not in CNF are

$$A \longrightarrow B_1B_2\ldots B_n$$

where $n \geq 3$. For each such production we introduce $n-2$ new variables $D_1, D_2, \ldots, D_{n-2}$, and replace the production by the productions

$$\begin{array}{rcl} A & \longrightarrow & B_1D_1 \\ D_1 & \longrightarrow & B_2D_2 \\ D_2 & \longrightarrow & B_3D_3 \\ & \ldots & \\ D_{n-3} & \longrightarrow & B_{n-2}D_{n-2} \\ D_{n-2} & \longrightarrow & B_{n-1}B_n \end{array}$$

(Note that the new variables D_i have to be different for different productions.) One application of the original production gives the same result as applying the new productions sequentially, one after the other. The new grammar is clearly equivalent to the original one. □

Example 70. Let us find a Chomsky normal form grammar that is equivalent to

$$\begin{array}{lcl} S & \longrightarrow & SSaA \mid bc \mid c \\ A & \longrightarrow & AAA \mid b \end{array}$$

□

Another normal form is so-called **Greibach normal form** (or GNF). A grammar is in Greibach normal form if all productions are of the form

$$A \longrightarrow aB_1B_2\ldots B_n$$

where $n \geq 0$ and $A, B_1, B_2, \ldots, B_n$ are variables, and a is a terminal symbol. In other words, all productions contain exactly one terminal symbol and it is the first symbol on the right hand side of the production. Without proof we state the following:

Theorem 71 *Every context-free language L without ε is generated by a grammar that is in the Greibach normal form.* □

3.4 Pushdown automata

A pushdown automaton (PDA) is a non-deterministic finite state automaton that has access to an infinite memory, organized as a stack. We'll see that the family of languages recognized by PDA is exactly the family of context-free languages. A PDA consists of the following:

- **Stack**. The stack is a string of symbols. The PDA has access only to the leftmost symbol of the stack. This is called the top of the stack. During one move of the PDA, the leftmost symbol may be removed ("popped" from the stack) and new symbols may be added ("pushed") on the top of the stack.
- **Input tape**. Similar to finite automata: the input string is normally scanned one symbol at a time. But also ε-moves are possible.
- **Finite state control unit**. The control unit is a non-deterministic finite state machine. Transitions may depend on the next input symbol and the topmost stack symbol.

There are two types of moves: normal moves, and ε-moves.

1. Normal moves: **Depending on**

   (a) the current state of the control unit,
   (b) the next input letter, and
   (c) the topmost symbol on the stack

   the PDA may

   (A) change the state,
   (B) pop the topmost element from the stack,
   (C) push new symbols to the stack, and
   (D) move to the next input symbol.

2. Spontaneous ε-moves don't have (b) and (D), i.e. they are done without using the input tape.

Example 72. As a first example, let us consider the following PDA that recognizes the language $L = \{a^n b^n \mid n \geq 1\}$. The PDA has two states, S_a and S_b, and the stack alphabet contains two symbols, A and Z_0. In the beginning, the machine is in the initial state S_a, and the stack contains only one symbol Z_0, called the start symbol of the stack. Possible moves are summarized in the following table:

State	Top of stack	Input Symbol a	Input Symbol b	Input Symbol ε
S_a	Z_0	Add one A to the stack, stay in state S_a	—	—
S_a	A	Add one A to the stack, stay in state S_a	Remove one A from the stack, go to state S_b	—
S_b	A	—	Remove one A from the stack, stay in state S_b	—
S_b	Z_0	—	—	Remove Z_0 from the stack, stay in state S_b

An input word is accepted if the PDA can reach the empty stack after reading all input symbols. □

An **instantaneous description** (ID) records the configuration of a PDA at any given time. It is a triple

$$(q, w, \gamma)$$

where

- q is the state of the PDA,
- w is the **remaining** input, i.e. the suffix of the original input that has not been used yet, and
- γ is the content of the stack.

The ID contains all relevant information that is needed in subsequent steps of the computation. We denote

$$(q_1, w_1, \gamma_1) \vdash (q_2, w_2, \gamma_2)$$

if there exists a move that takes the first ID into the second ID, and we denote

$$(q_1, w_1, \gamma_1) \vdash^* (q_2, w_2, \gamma_2)$$

if there is a sequence of moves (possibly empty) from the first ID to the second ID. Finally, we denote

$$(q_1, w_1, \gamma_1) \vdash^n (q_2, w_2, \gamma_2)$$

if the first ID becomes the second ID in exactly n moves.

For example, when accepting input string *aaabbb* our sample PDA enters the following ID's:

$$\begin{aligned}(S_a, aaabbb, Z_0) &\vdash (S_a, aabbb, AZ_0) \vdash (S_a, abbb, AAZ_0) \vdash (S_a, bbb, AAAZ_0) \vdash (S_b, bb, AAZ_0) \vdash \\ &\vdash (S_b, b, AZ_0) \vdash (S_b, \varepsilon, Z_0) \vdash (S_b, \varepsilon, \varepsilon)\end{aligned}$$

The word is accepted iff all input letters were consumed, and in the end the stack was empty. This is referred to as acceptance by **empty stack**.

There is another way of defining which words are accepted. Some states may be called final states, and a word is accepted iff after reading all input letters the PDA is in an accepting state. This is called acceptance by **final state**. We see later that the two possible modes of acceptance are equivalent: If there is a PDA that recognizes language L using empty stack then there (effectively) exists another PDA that recognizes L using final states, and vice versa.

Formally, a pushdown automaton M consists of

$$(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

where

- Q is a finite **state set**,
- Σ is the finite **input alphabet**,

- Γ is the finite **stack alphabet**,
- $q_0 \in Q$ is the **initial state**,
- $Z_0 \in \Gamma$ is the **start symbol** of the stack,
- $F \subseteq Q$ is the set of **final states**,
- δ is the **transition function** from

$$Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma$$

to finite subsets of

$$Q \times \Gamma^*.$$

Note that $\delta(q, a, Z)$ is a **set** of possible outcomes; PDA are non-deterministic.

1. The interpretation of executing a transition

$$(p, \gamma) \in \delta(q, a, Z)$$

(where $p, q \in Q$, $a \in \Sigma$, $Z \in \Gamma$, $\gamma \in \Gamma^*$) is that in state q, reading input letter a, and Z on the top of the stack, the PDA goes to state p, moves to the next input symbol, and **replaces** Z by γ on top of the stack. The leftmost symbol of γ will be the new top of the stack (if $\gamma \neq \varepsilon$).

In terms of ID's, this transition means that the move

$$(q, aw, Z\alpha) \vdash (p, w, \gamma\alpha)$$

is allowed for all $w \in \Sigma^*$ and $\alpha \in \Gamma^*$.

2. The interpretation of executing a transition

$$(p, \gamma) \in \delta(q, \varepsilon, Z)$$

is that in state q and Z on the top of the stack, the PDA goes to state p and replaces Z by γ on top of the stack. No input symbol is consumed, and the transition can be used regardless of the current input symbol. In terms of ID's, this transition means that the move

$$(q, w, Z\alpha) \vdash (p, w, \gamma\alpha)$$

is allowed for all $w \in \Sigma^*$ and $\alpha \in \Gamma^*$.

Let us formally define the two modes of acceptance:

- A word $w \in \Sigma^*$ is accepted by PDA M by empty stack, iff

$$(q_0, w, Z_0) \vdash^* (q, \varepsilon, \varepsilon)$$

for some $q \in Q$. Note that q can be any state, final or non-final.

The language recognized by PDA M using empty stack is denoted by $N(M)$:

$$N(M) = \{w \in \Sigma^* \mid \exists q \in Q : \ (q_0, w, Z_0) \vdash^* (q, \varepsilon, \varepsilon)\}.$$

- A word $w \in \Sigma^*$ is accepted by PDA M by final state, iff

$$(q_0, w, Z_0) \vdash^* (q, \varepsilon, \gamma)$$

for some $q \in F$, and $\gamma \in \Gamma^*$. Now q has to be a final state, but the stack does not need to be empty.

The language recognized by PDA M using final state is denoted by $L(M)$:

$$L(M) = \{w \in \Sigma^* \mid \exists q \in F, \gamma \in \Gamma^* : \ (q_0, w, Z_0) \vdash^* (q, \varepsilon, \gamma)\}.$$

Example 73. Our sample PDA is

$$M = (\{S_a, S_b\}, \{a, b\}, \{Z_0, A\}, \delta, S_a, Z_0, \emptyset)$$

where

$$\begin{array}{lcl} \delta(S_a, a, Z_0) & = & \{(S_a, AZ_0)\} \\ \delta(S_a, a, A) & = & \{(S_a, AA)\} \\ \delta(S_a, b, A) & = & \{(S_b, \varepsilon)\} \\ \delta(S_b, b, A) & = & \{(S_b, \varepsilon)\} \\ \delta(S_b, \varepsilon, Z_0) & = & \{(S_b, \varepsilon)\} \end{array}$$

and all other sets $\delta(q, a, Z)$ are empty. It does not matter which set we choose as the set of final states, since we use acceptance by empty stack. (Choose, for example $F = \emptyset$.) We have

$$N(M) = \{a^n b^n \mid n \geq 1\}.$$

□

Example 74. Let us construct a PDA M such that

$$N(M) = \{w \mid w \in \{a, b\}^* \text{ and } w \text{ is a palindrome }\}.$$

The PDA will read symbols from the input and push them into the stack. At some point it guesses that it is in the middle of the input word, and starts popping letters from the stack and comparing them against the following input letters. If all letters match, and the stack and the input string become empty at the same time, the word was a palindrome. We define,

$$M = (\{q_1, q_2\}, \{a, b\}, \{Z_0, A, B\}, \delta, q_1, Z_0, \emptyset)$$

where δ is

$$\begin{array}{lcl} \delta(q_1, \varepsilon, Z_0) & = & \{(q_1, \varepsilon)\} \\ \delta(q_1, a, Z_0) & = & \{(q_1, AZ_0), (q_2, AZ_0), (q_2, Z_0)\} \\ \delta(q_1, b, Z_0) & = & \{(q_1, BZ_0), (q_2, BZ_0), (q_2, Z_0)\} \\ \delta(q_1, a, A) & = & \{(q_1, AA), (q_2, AA), (q_2, A)\} \\ \delta(q_1, b, A) & = & \{(q_1, BA), (q_2, BA), (q_2, A)\} \\ \delta(q_1, a, B) & = & \{(q_1, AB), (q_2, AB), (q_2, B)\} \\ \delta(q_1, b, B) & = & \{(q_1, BB), (q_2, BB), (q_2, B)\} \\ \delta(q_2, b, B) & = & \{(q_2, \varepsilon)\} \\ \delta(q_2, a, A) & = & \{(q_2, \varepsilon)\} \\ \delta(q_2, \varepsilon, Z_0) & = & \{(q_2, \varepsilon)\} \end{array}$$

State q_1 is used in the first half of the input word, state q_2 in the second half. The three possible outcomes of some transitions have the following roles in accepting computations:

- use the first transition, if the input letter is before the end of the first half of the input word.
- use the second transition, if the input word has even length and the current input letter is the last letter of the first half, and
- use the third transition, if the input word has odd length and the current input letter is exactly in the middle of the input word.

For example, the word *bab* is accepted because

$$(q_1, bab, Z_0) \vdash (q_1, ab, BZ_0) \vdash (q_2, b, BZ_0) \vdash (q_2, \varepsilon, Z_0) \vdash (q_2, \varepsilon, \varepsilon),$$

and the word *abba* is accepted, as shown by

$$(q_1, abba, Z_0) \vdash (q_1, bba, AZ_0) \vdash (q_2, ba, BAZ_0) \vdash (q_2, a, AZ_0) \vdash (q_2, \varepsilon, Z_0) \vdash (q_2, \varepsilon, \varepsilon)$$

□

A PDA is called **deterministic** if every ID has at most one possible move. This means that

- if $\delta(q, \varepsilon, Z)$ is non-empty then $\delta(q, a, Z)$ is empty for every input letter a, and
- All $\delta(q, a, Z)$ and $\delta(q, \varepsilon, Z)$ contain at most one element.

The first condition states that there is no choice between ε-move and non-ε-move. If one can make a move without reading an input letter, then that is the only possible move. Our first example was deterministic, while the PDA in Example 74 is non-deterministic.

Important remark: There exist languages that are recognized by non-deterministic PDA but not by any deterministic PDA. Language

$$\{a^n b^m c^k \mid n = m \text{ or } n = k\}$$

is an example of such a language. The situation is different from finite automata where determinism was equivalent to non-determinism. Languages that can be recognized by deterministic PDA (using final states) are called **deterministic context-free languages**.

The following two theorems establish that the empty stack and final state based modes of acceptance are equivalent for PDA.

Theorem 75 *If $L = L(M)$ for some PDA M then there (effectively) exists a PDA M' such that $L = N(M')$.*

Proof. PDA M recognizes L using final states. We want to construct a PDA M' that simulates M, with the additional option that when M enters a final state, M' may enter a special 'erase the stack' -state q_e and remove all symbols from the stack. Them M' accepts w by empty stack if M entered a final state.

We have to be careful to make sure that M' does not accept a word by mistake if M empties the stack without entering an accepting state. We do this by inserting a new bottom of the stack symbol X_0 below the old start symbol Z_0. Even if M empties the stack, M' will have the symbol X_0 in

the stack. We have a new initial state q_0' that simply places Z_0 above X_0 and starts the simulation of M. If M empties the stack without entering the final state then it is stuck. Corresponding computation in M' gets stuck with X_0 in the stack.

More precisely, let

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F).$$

We construct

$$M' = (Q \cup \{q_0', q_e\}, \Sigma, \Gamma \cup \{X_0\}, \delta', q_0', X_0, \emptyset)$$

where δ' contains all transitions of δ and, in addition, the following transitions:

- $\delta'(q_0', \varepsilon, X_0) = \{(q_0, Z_0X_0)\}$,
- $\delta'(q, \varepsilon, Z)$ contains (q_e, Z) for all final states $q \in F$ and stack symbols $Z \in \Gamma \cup \{X_0\}$,
- $\delta'(q_e, \varepsilon, Z) = \{(q_e, \varepsilon)\}$, for all stack symbols $Z \in \Gamma \cup \{X_0\}$.

The first transition initiates the simulation of M' by placing the original 'bottom of the stack' -symbol Z_0 on the stack, and entering the old initial state q_0. The second transitions enable the move to the 'empty the stack' -state q_e whenever a final state is entered. And the third transitions simply remove symbols from the stack if the machine is in state q_e.

To prove that $L(M) = N(M')$ we observe the following: If

$$(q_0, w, Z_0) \vdash^* (q, \varepsilon, \gamma)$$

in the original automaton M, then in M' we have the computation

$$(q_0', w, X_0) \vdash (q_0, w, Z_0X_0) \vdash^* (q, \varepsilon, \gamma X_0)$$

If q is a final state then M' can continue the computation:

$$(q, \varepsilon, \gamma X_0) \vdash (q_e, \varepsilon, \gamma X_0) \vdash^* (q_e, \varepsilon, \varepsilon),$$

i.e. M' accepts w using empty stack.

Conversely, assume that M' accepts word w by empty stack. The only valid computations of M' that empty the stack follow the order:

$$(q_0', w, X_0) \vdash (q_0, w, Z_0X_0) \vdash^* (q, \varepsilon, \gamma X_0) \vdash (q_e, \varepsilon, \gamma X_0) \vdash^* (q_e, \varepsilon, \varepsilon).$$

But the part $(q_0, w, Z_0X_0) \vdash^* (q, \varepsilon, \gamma X_0)$ means that there is a computation

$$(q_0, w, Z_0) \vdash^* (q, \varepsilon, \gamma)$$

in the original PDA M, and the move $(q, \varepsilon, \gamma X_0) \vdash (q_e, \varepsilon, \gamma X_0)$ in M' implies that q is a final state of M. This means that M accepts w by final state. □

Theorem 76 *If $L = N(M)$ for some PDA M then there (effectively) exists a PDA M' such that $L = L(M')$.*

Proof. Now M recognizes L by empty stack. We construct a PDA M' that simulates M and detects when the stack is empty. When that happens the PDA enters a final state. In order to be able to detect the empty stack, we again introduce a new 'bottom of the stack' -symbol X_0. As soon as X_0 is revealed the new PDA enters a new final state q_f.

More precisely, Let

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, \emptyset).$$

We construct

$$M' = (Q \cup \{q_0', q_f\}, \Sigma, \Gamma \cup \{X_0\}, \delta', q_0', X_0, \{q_f\})$$

where δ' contains all transitions of δ plus the following transitions:

- $\delta'(q_0', \varepsilon, X_0) = \{(q_0, Z_0X_0)\}$,
- $\delta'(q, \varepsilon, X_0) = \{(q_f, \varepsilon)\}$, for all states $q \in Q$.

If in the original PDA M

$$(q_0, w, Z_0) \vdash^* (q, \varepsilon, \varepsilon)$$

then in M' we have the accepting computation

$$(q_0', w, X_0) \vdash (q_0, w, Z_0X_0) \vdash^* (q, \varepsilon, X_0) \vdash (q_f, \varepsilon, \varepsilon)$$

that finishes in the final state q_f.

Conversely, every accepting calculation by M' has to use the productions in the correct order. To get to the final state q_f we must have symbol X_0 exposed on top of the stack, i.e. the original PDA M must empty its stack. □

Example 77. Let us modify our first PDA so that it recognizes the language

$$L = \{a^n b^n \mid n \geq 1\}$$

using final state instead of the empty stack. □

Next we prove that PDA recognize all context-free languages. In fact, all context-free languages are recognized by PDA having only one state. We prove this by showing how any context-free grammar can be simulated using only the stack.

Consider for example the grammar $G = (V, T, P, S)$ where $V = \{S\}$, $T = \{a, b\}$ and P contains productions

$$S \longrightarrow aSbS \mid bSaS \mid \varepsilon.$$

Let us construct a PDA M that recognizes $L(G)$ using empty stack. The stack symbols of M are the terminals and variables of G. At all times, the content of the stack is a suffix of a sentential form by G. Initially the stack contains the start symbol of the grammar.

- If the topmost symbol of the stack is a terminal symbol it is compared against next input letter. If they are identical, the symbol is popped from the stack and the machine moves to the next input letter. If they are different the simulation halts.

- If the topmost symbol of the stack is a variable the machine rewrites it by a righthand side of a production using an ε-move.

In our example, we get the machine

$$M = (\{q\}, \{a, b\}, \{a, b, S\}, \delta, q, S, \emptyset)$$

with δ defined as follows: Transitions for terminals on the top of the stack are

$$\begin{array}{lcl} \delta(q, a, a) & = & \{(q, \varepsilon)\}, \\ \delta(q, a, b) & = & \emptyset, \\ \delta(q, b, b) & = & \{(q, \varepsilon)\}, \\ \delta(q, b, a) & = & \emptyset, \end{array}$$

and the transitions for the variable S on top of the stack are

$$\delta(q, \varepsilon, S) = \{(q, aSbS), (q, bSaS), (q, \varepsilon)\}.$$

There are no other transitions. The machine simulates leftmost derivations of G on the stack. Terminals that are generated on the left end have to match the input, while a variable at the left extreme is rewritten in the stack using a production of the grammar.

Compare for instance the leftmost derivation

$$S \Longrightarrow aSbS \Longrightarrow aaSbSbS \Longrightarrow aabSbS \Longrightarrow aabaSbSbS \Longrightarrow aababSbS \Longrightarrow aababbS \Longrightarrow aababb$$

of *aababb* to the accepting computation

$$\begin{array}{l} (q, aababb, S) \vdash (q, aababb, aSbS) \vdash (q, ababb, SbS) \vdash (q, ababb, aSbSbS) \vdash (q, babb, SbSbS) \vdash \\ \quad \vdash (q, babb, bSbS) \vdash (q, abb, SbS) \vdash (q, abb, aSbSbS) \vdash (q, bb, SbSbS) \vdash \\ \quad \vdash (q, bb, bSbS) \vdash (q, b, SbS) \vdash (q, b, bS) \vdash (q, \varepsilon, S) \vdash (q, \varepsilon, \varepsilon). \end{array}$$

Theorem 78 *If $L = L(G)$ for some context-free grammar G then there (effectively) exists a PDA M such that $L = N(M)$.*

Proof. Let $G = (V, T, P, S)$ be a context-free grammar generating L. We construct a PDA

$$M = (\{q\}, T, V \cup T, \delta, q, S, \emptyset)$$

where δ is defined as follows:

- for all $A \longrightarrow \gamma$ in P, we include $(q, \gamma) \in \delta(q, \varepsilon, A)$, and
- for all $a \in T$, we set $\delta(q, a, a) = \{(q, \varepsilon)\}$.

To prove that $N(M) = L(G)$ observe following two facts:

1. The concatenation of the consumed input and the content of the stack in M is always a valid sentential form in G: If
$$(q, wu, S) \vdash^* (q, u, \gamma)$$
in M then
$$S \Rightarrow^* w\gamma$$
in G.

   This can be proved using mathematical induction on the number of moves: It is true in the beginning because the consumed input is ε and the stack is S and $\varepsilon S = S$ is a sentential form.

   If the above is true before a move by M then it true one move later because one move in M means either

   - replacing the variable on top of the stack γ by the righthand side of a production, which corresponds to one derivation step by G, or
   - removing one terminal from the stack γ and adding it to the consumed input w, which keeps $w\gamma$ invariant.

   This proves that $N(M) \subseteq L(G)$: If $w \in N(M)$ then
$$(q, w, S) \vdash^* (q, \varepsilon, \varepsilon)$$
which means that $w\gamma = w$ is a sentential form in G, i.e. $w \in L(G)$.

2. If
$$S \Rightarrow^* w\gamma$$
by a leftmost derivation, where w is terminal word and γ starts with a variable or is empty, then
$$(q, wu, S) \vdash^* (q, u, \gamma)$$
in M, for any word u.

   This can be proved using mathematical induction on the number of derivation steps, using the fact that a leftmost derivation step in G can be simulated by the PDA M by applying the production to the variable that is on top of the stack. Using the terminal matching transitions, one can then move all terminals that appear on top of the stack to the consumed input, until the top of the stack is again a variable (or empty, in the end of an accepting computation).

   This proves that $L(G) \subseteq N(M)$: If $w \in L(G)$ then
$$S \Rightarrow^* w$$
which means that in M (we have $\gamma = \varepsilon$)
$$(q, w, S) \vdash^* (q, \varepsilon, \varepsilon).$$

□

Example 79. Consider the grammar

$$G = (\{S\}, \{a, b\}, P, S)$$

with productions

$$S \longrightarrow \varepsilon \mid aSb$$

that generates $L(G) = \{a^n b^n \mid n \geq 0\}$. Let us use the construction of the proof to build a PDA such that $N(M) = L(G)$, and let us compare the accepting computation on input $aabb$ to the derivation of $aabb$ by G. □

Next we prove that PDA recognize **only** context-free languages: For a given PDA M we construct an equivalent context-free grammar G.

Theorem 80 *If $L = N(M)$ for some PDA M then there (effectively) exists a context-free grammar G such that $L = L(G)$.*

This construction is made complicated by the fact that the PDA may have more than one state. Let us divide the proof into the following two steps:

Lemma 81 *If $L = N(M_1)$ for some PDA M_1 then there effectively exists a PDA M_2 that has only one state such that $L = N(M_2)$.*

Lemma 82 *If $L = N(M_2)$ for some PDA M_2 with one state, then there effectively exists a context-free grammar G such that $L = L(G)$.*

Proof of Lemma 81. Let $L = N(M_1)$ for PDA

$$M_1 = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, \emptyset).$$

Let us construct a PDA M_2 with only one state that recognizes the same language. The computations of M_1 will be simulated in such a way that the state of M_1 will be stored in the topmost element of the stack in M_2. So at all times the topmost element of the stack contains both the stack symbol of M_1 and the current state of M_1.

Then every move of M_1 can be simulated by M_2 since it knows the state and the topmost stack symbol of M_1. After each step, the next state is written into the new topmost element.

This is fine as long as at least one new symbol is written into the stack. However, there is one big problem: Where is the next state stored when the stack is only popped and nothing new is written ? The new top of the stack is the symbol that used to be the second highest on the stack, and it is not accessible for writing. The new state must have been stored in the stack already when the second highest symbol was pushed into the stack — and this was possibly long before it becomes the topmost symbol.

How do we know long before what is going to be the state of M_1 when a particular stack symbol is revealed to the top of the stack ? Answer: we guess it using non-determinism of M_2. At the time when the symbol becomes the topmost element of the stack we only need to verify that the earlier guess was correct.

In order to be able to verify the guess it has to be stored also on the stack symbol above. Therefore stack symbols of M_2 need to contain two states of M_1: one indicates the state of the machine when the symbol is the topmost element of the stack, and the other state indicates the state of M_1 when the the element below is the topmost element of the stack.

Therefore the stack alphabet of M_2 is

$$\Gamma_2 = (Q \times \Gamma \times Q) \cup \{X_0\}$$

where X_0 is a new bottom of the stack symbol. Symbol $[q, Z, p]$ on top of the stack indicates that that M_1 is in state q with Z is on top of the stack and p will be the state of M_1 when the element below becomes the topmost symbol of the stack. PDA M_2 only has one state # which is completely irrelevant to computations.

At all times the stack is kept consistent in the sense that if $[q, Z, p]$ is immediately above $[q', Z', p']$ in the stack then $p = q'$. We call this the consistency requirement of the stack. Consistency means that the content of the stack will always look like

$$[q_1, Z_1, q_2][q_2, Z_2, q_3] \dots [q_m, Z_m, q_{m+1}],$$

for some $q_i \in Q$ and $Z_i \in \Gamma$. The values $q_2, q_3, \dots, q_{m+1}$ are the non-deterministic 'guesses' done by M_2 about the state of M_1 when the corresponding stack elements will get exposed on top of the stack. As the stack shrinks the correctness of the guesses gets verified.

The construction is done in such a way that M_2 can reach ID

$$(\#, u, [q_1, Z_1, q_2][q_2, Z_2, q_3] \dots [q_m, Z_m, q_{m+1}])$$

if and only if M_1 can reach ID

$$(q_1, u, Z_1 Z_2 \dots Z_m),$$

and $q_2, q_3, \dots, q_{m+1}$ are arbitrary states. Let us describe the transitions of M_2 in δ_2.

1. The simulation is initialized by

   $$(\#, [q_0, Z_0, q]) \in \delta_2(\#, \varepsilon, X_0)$$

   for all $q \in Q$. State q is the 'guess' for the state of M_1 after the stack becomes empty. Therefore the first move by M_2 is forced to be

   $$(\#, w, X_0) \vdash (\#, w, [q_0, Z_0, q])$$

   for some (any) $q \in Q$. The result after the move corresponds to the initial ID

   $$(q_0, w, Z_0)$$

   of M_1.

2. For every M_1's transition

   $$(p, \varepsilon) \in \delta(q, a, Z)$$

   that shrinks the stack, we have the transition

   $$(\#, \varepsilon) \in \delta_2(\#, a, [q, Z, p])$$

in M_2. Here $a \in \Sigma \cup \{\varepsilon\}$. Interpretation: If M_1 goes from state q to p and removes Z from the stack, M_2 can remove $[q, Z, p]$ from the stack. We know that the stack element below will be $[p, \ldots]$.

Corresponding computation steps by M_1 and M_2 are

$$(q, au, ZZ_1Z_2 \ldots Z_m) \vdash (p, u, Z_1Z_2 \ldots Z_m)$$

and

$$(\#, au, [q, Z, p][p, Z_1, q_2][q_2, Z_2, q_3] \ldots [q_m, Z_m, q_{m+1}])$$

$$\vdash (\#, u, [p, Z_1, q_2][q_2, Z_2, q_3] \ldots [q_m, Z_m, q_{m+1}]),$$

respectively.

3. For M_1's transitions

$$(p, X_1X_2 \ldots X_k) \in \delta(q, a, Z)$$

where $k \geq 1$, we add, for every $q_1 \in Q$, the transitions

$$(\#, [p, X_1, p_2][p_2, X_2, p_3] \ldots [p_k, X_k, q_1]) \in \delta_2(\#, a, [q, Z, q_1])$$

to M_2, for all combinations of

$$p_2, p_3, \ldots, p_k \in Q.$$

Interpretation: If M_1 goes from state q to p and changes Z on top of the stack into $X_1X_2 \ldots X_k$, then M_2 can replace the topmost element of the stack by the corresponding stack elements. The states $p_2, p_3, \ldots, p_k$ inserted in the stack are non-deterministic guesses, so we must have a transition for all choices of them. The bottom most state inserted is q_1, which was the second state component of the stack element $[q, Z, q_1]$ removed. We know it matches the state stored in the stack element below. And the topmost state is the new state p of the PDA M_1.

The corresponding computation steps by M_1 and M_2 are

$$(q, au, ZZ_1Z_2 \ldots Z_m) \vdash (p, u, X_1X_2 \ldots X_k \ Z_1Z_2 \ldots Z_m)$$

and

$$(\#, au, [q, Z, q_1][q_1, Z_1, q_2][q_2, Z_2, q_3] \ldots [q_m, Z_m, q_{m+1}])$$

$$\vdash (\#, u, [p, X_1, p_2][p_2, X_2, p_3] \ldots [p_k, X_k, q_1] \ [q_1, Z_1, q_2][q_2, Z_2, q_3] \ldots [q_m, Z_m, q_{m+1}]),$$

respectively.

We have now constructed all transitions in δ_2 of M_2. The PDA M_2 simulates M_1 step-by-step. The transitions are such that at all times the stack of M_2 is consistent: the states on the consecutive stack elements match. An ID can be reached in M_2 if and only if the corresponding ID can be reached in M_1. Therefore M_2 empties the stack exactly when M_1 empties the stack, i.e. they recognize the same languages. □

Example 83. Let us construct a one state PDA that is equivalent to the PDA

$$M = (\{S_a, S_b\}, \{a, b\}, \{Z_0, A\}, \delta, S_a, Z_0, \emptyset)$$

where

$$\begin{array}{lcl} \delta(S_a, a, Z_0) & = & \{(S_a, AZ_0)\} \\ \delta(S_a, a, A) & = & \{(S_a, AA)\} \\ \delta(S_a, b, A) & = & \{(S_b, \varepsilon)\} \\ \delta(S_b, b, A) & = & \{(S_b, \varepsilon)\} \\ \delta(S_b, \varepsilon, Z_0) & = & \{(S_b, \varepsilon)\} \end{array}$$

that recognizes the language

$$N(M) = \{a^n b^n \mid n \geq 1\}.$$

□

Proof of Lemma 82. Let

$$M = (\{\#\}, \Sigma, \Gamma, \delta, \#, Z_0, \emptyset)$$

be a PDA with one state. We construct a context-free grammar

$$G = (V, \Sigma, P, S)$$

such that $L(G) = N(M)$.

The variables of G are the stack symbols of M. Let us assume that the stack alphabet Γ and the input alphabet Σ are disjoint. (If they are not we can simply rename stack symbols.) We have

$$\begin{array}{l} V = \Gamma, \\ S = Z_0. \end{array}$$

For every transition

$$(\#, Z_1 Z_2 \ldots Z_m) \in \delta(\#, a, Z)$$

where $a \in \Sigma \cup \{\varepsilon\}$ we introduce a production

$$Z \longrightarrow a Z_1 Z_2 \ldots Z_m$$

in P. All sentential forms derived by leftmost derivations are then of the special form

$$w\gamma,$$

where $w \in \Sigma^*$ and $\gamma \in \Gamma^*$. In the corresponding ID of M the string γ is the content of the stack and w input that has been consumed so far.

The next leftmost derivation step applies a production on the leftmost symbol of γ. This corresponds to one move by PDA M: The corresponding transition changes the topmost symbol of the stack and possibly consumes one more input letter. The derivation step

$$wZ\alpha \Rightarrow waZ_1 Z_2 \ldots Z_m \alpha$$

hence corresponds to the move

$$(\#, au, Z\alpha) \vdash (\#, u, Z_1 Z_2 \ldots Z_m \alpha).$$

Using mathematical induction on the number of derivation steps, one easily proves that $\alpha = w\gamma$ is a sentential form derivable by a leftmost derivation in G if and only if γ is a possible stack content

after consuming w from the input. If α is terminal, it means that the corresponding calculation in M has reached the empty stack, and $\alpha = w$ (=the consumed input) was accepted by M. □

Example 84. Consider the one state PDA

$$M = (\{\#\}, \{a, b\}, \{A, B, Z\}, \delta, \#, Z, \emptyset)$$

where δ contains the transitions

$$\begin{array}{lcl} \delta(\#, \varepsilon, Z) & = & \{(\#, AZZA), (\#, B)\} \\ \delta(\#, a, A) & = & \{(\#, \varepsilon)\} \\ \delta(\#, b, B) & = & \{(\#, \varepsilon)\} \end{array}$$

Our construction gives the following equivalent grammar:

$$G = (\{A, B, Z\}, \{a, b\}, P, Z)$$

where P has the productions

$$\begin{array}{lcl} Z & \longrightarrow & AZZA \mid B \\ A & \longrightarrow & a \\ B & \longrightarrow & b \end{array}$$

The computation

$$\begin{array}{rcl} (\#, abba, Z) & \vdash & (\#, abba, AZZA) \vdash (\#, bba, ZZA) \\ & \vdash & (\#, bba, BZA) \vdash (\#, ba, ZA) \\ & \vdash & (\#, ba, BA) \vdash (\#, a, A) \vdash (\#, \varepsilon, \varepsilon) \end{array}$$

of M has the corresponding derivation

$$Z \Rightarrow AZZA \Rightarrow aZZA \Rightarrow aBZA \Rightarrow abZA \Rightarrow abBA \Rightarrow abbA \Rightarrow abba$$

in G. □

We have proved the equivalence of three devices: context-free grammars and PDA using empty stack and final states acceptance modes. The constructions were all effective which means that when investigating closure properties of context-free languages or decision algorithms we may assume that the input is given in any of the three forms.

3.5 Pumping lemma for CFL

Not all languages are context-free. Our first technique for proving that some language is not context-free is a pumping lemma. Just as in the case of regular languages, pumping lemma states a property that every context-free language satisfies. If a language does not satisfy the pumping lemma then the language is not context-free.

Theorem 85 (Pumping lemma for CFL) *Let L be a context-free language. Then there exists a positive number n such that every word z of length n or greater that belongs to language L can be divided into five segments*

$$z = uvwxy$$

in such a way that

$$\begin{cases} |vwx| \leq n, \ \textit{and} \\ v \neq \varepsilon \quad \textit{or} \quad x \neq \varepsilon, \end{cases} \tag{6}$$

and for all $i \geq 0$ *the word* uv^iwx^iy *is in the language* L.

The main difference to the pumping lemma of regular languages is that now the word z contains two subwords v and x that are pumped. Note that subwords v and x are always pumped the same number of times.

For example, the language $L = \{a^m b^m \mid m \geq 0\}$ satisfies the pumping lemma. Number $n = 2$ can be used. Let $z = a^m b^m$ be an arbitrary word of the language such that $|z| \geq 2$. This means $m \geq 1$. We break z into five parts as follows:

$$z = \underbrace{a^{m-1}}_{u} \ \underbrace{a}_{v} \ \underbrace{\varepsilon}_{w} \ \underbrace{b}_{x} \ \underbrace{b^{m-1}}_{y}$$

This division satisfies (6) since $vx = ab \neq \varepsilon$ and $|vwx| = |ab| \leq 2$. Subwords v and x can be pumped arbitrarily many times: for every $i \geq 0$

$$uv^iwx^iy = a^{m-1} \ a^i \ \varepsilon \ b^i \ b^{m-1} = a^{m+i-1}b^{m+i-1}$$

is in language L.

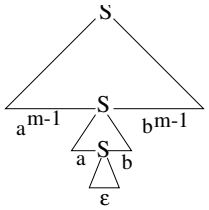
Let us analyze the reason why our L satisfies the pumping lemma: Language L is generated by a grammar with two productions

$$S \longrightarrow aSb \mid \varepsilon.$$

Word $a^m b^m$, $m \geq 1$, has the derivation

$$S \Longrightarrow^* a^{m-1}Sb^{m-1} \Longrightarrow a^{m-1}aSbb^{m-1} \Longrightarrow a^{m-1}a\varepsilon bb^{m-1}$$

corresponding to the derivation tree



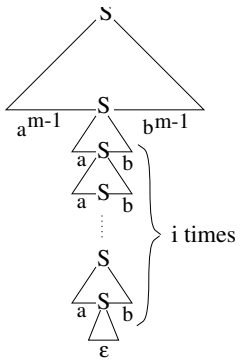
We used three subderivations

$$S \Longrightarrow^* a^{m-1}Sb^{m-1},$$
$$S \Longrightarrow^* aSb,$$
$$S \Longrightarrow^* \varepsilon.$$

The middle subderivation $S \Longrightarrow^* aSb$ derives from the variable S the variable itself. It means that the subderivation can be iterated arbitrarily many times. Repeating it i times gives

$$S \Longrightarrow^* a^{m-1}Sb^{m-1} \Longrightarrow^i a^{m-1}a^iSb^ib^{m-1} \Longrightarrow a^{m-1}a^i\varepsilon b^ib^{m-1}$$

corresponding to the derivation tree



In other words, pumping i times the words $v = a$ and $x = b$ corresponds to iterating the middle subderivaton i times.

Let us prove that the argument above can be repeated for any context-free language.

Proof of the pumping lemma. The proof is based on the fact that during derivations of sufficiently long words, some variable derives a sentential form containing the variable itself. In other words, we have derivations

$$\begin{array}{lll} S & \Rightarrow^* & uAy \\ A & \Rightarrow^* & vAx \\ A & \Rightarrow^* & w \end{array}$$

for some terminal words u, v, w, x, y and variable A. But the derivation $A \Rightarrow^* vAx$ may be repeated arbitrarily many times, say i times:

$$S \Rightarrow^* uAy \Rightarrow^* uvAxy \Rightarrow^* uvvAxxy \Rightarrow^* \ldots \Rightarrow^* uv^iAx^iy \Rightarrow^* uv^iwx^iy$$

This shows that uv^iwx^iy is in language L for every $i \geq 0$.

To make a precise proof including the bounds for the lengths of the subwords, we have to be more careful: Let G be a grammar in the Chomsky normal form such that $L = L(G)$ (or $L = L(G) \cup \{\varepsilon\}$ if $\varepsilon \in L$). Let us define the **depth** of a derivation tree as the length of the longest path from its root to any leaf. We use the following simple lemma:

Lemma 86 *If a derivation tree by a CNF grammar for word w has depth d then $|w| \leq 2^{d-1}$.*

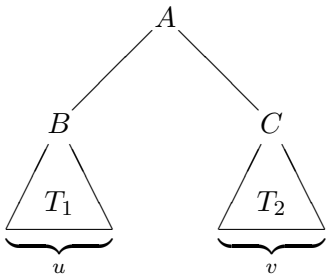
Proof. We use mathematical induction on d:

1° (basis) If $d = 1$ then the tree must be the trivial one:

$$\begin{array}{c} A \\ | \\ a \end{array}$$

and the word it derives has length 2^{d-1}.

2° (inductive step) Let $d > 1$. The tree is



and $w = uv$. Since the depths of trees T_1 and T_2 are at most $d-1$, it follows from the inductive hypothesis that $|u| \leq 2^{d-2}$ and $|v| \leq 2^{d-2}$. Therefore

$$|w| = |u| + |v| \leq 2^{d-2} + 2^{d-2} = 2^{d-1}.$$

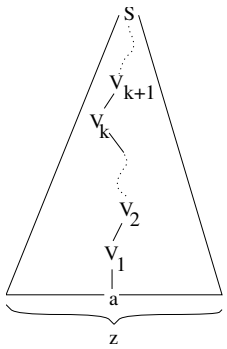
□

Let k be the number of variables in G, a CNF grammar for language L. Let us choose the number n in the pumping lemma as $n = 2^k$. Let $z \in L$ be such that $|z| \geq n$, and let T be a derivation tree for z.

The depth of T has to be at least $k+1$, since we just proved that a tree of smaller depth can yield only shorter words. Pick one maximum length path π from the root to a leaf in T. Reading from the leaf up, let

$$a, V_1, V_2, V_3, \ldots, V_{k+1}$$

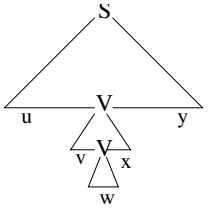
be the first $k+2$ labels along path π:



Since the grammar has only k variables, it follows from the pigeon hole principle that two of the variables V_1, V_2, …, V_{k+1} must be identical, say

$$V_s = V_t = V, \text{ for } s < t.$$

So the derivation tree for z looks like:



We have valid derivations

$$\begin{array}{lll} S & \Rightarrow^* & uVy \\ V & \Rightarrow^* & vVx \\ V & \Rightarrow^* & w. \end{array}$$

This means there are derivations for words uv^iwx^iy for all $i \geq 0$:

$$S \Rightarrow^* uVy \Rightarrow^* uvVxy \Rightarrow^* uv^2Vx^2y \Rightarrow^* \ldots\ldots \Rightarrow^* uv^iVx^iy \Rightarrow^* uv^iwx^iy$$

Clearly $vx \neq \varepsilon$: Since the grammar G is in CNF, node V_s must have a sibling, and the yield of the sibling is part of either v or x.

To complete the proof we have to show that the length of the word vwx is at most $n = 2^k$: But it is the yield of the subtree rooted at V_t, and the depth of that subtree is at most $k + 1$. (If there would be longer path from V_t to a leaf, then that path combined with the path from the root S to V_t would be longer than the longest path π, a contradiction.) We know how long can the yields of trees of depth $\leq k + 1$ be: they can have at most 2^k letters. This completes the proof of the pumping lemma. □

We use the pumping lemma to prove that certain languages are not context-free. It works the same way as the pumping lemma for regular languages. To show that a language is not a CFL we show that it does not satisfy the pumping lemma. So we are more interested in the negation of the pumping lemma.

Here is the mathematical formulation for the pumping lemma:

$$\begin{array}{l} (\exists n) \\ \quad (\forall z \in L, |z| \geq n) \\ \quad\quad (\exists u, v, w, x, y : z = uvwxy, |vwx| \leq n, vx \neq \varepsilon) \\ \quad\quad\quad (\forall i \geq 0) \\ \quad\quad\quad\quad uv^iwx^iy \in L, \end{array}$$

Its negation is the statement:

$$\begin{array}{l} (\forall n) \\ \quad (\exists z \in L, |z| \geq n) \\ \quad\quad (\forall u, v, w, x, y : z = uvwxy, |vwx| \leq n, vx \neq \varepsilon) \\ \quad\quad\quad (\exists i \geq 0) \\ \quad\quad\quad\quad uv^iwx^iy \notin L. \end{array}$$

To use the pumping lemma to prove that a language is not context-free we do the following two steps:

(1) For every n select a word $z \in L$, $|z| \geq n$,

(2) for any division $z = uvwxy$ of z into five segments satisfying $|vwx| \leq n$ and $vx \neq \varepsilon$, find a number $i \geq 0$ such that

$$uv^iwx^iy$$

is **not** in the language L.

Example 87. Consider the language

$$L = \{a^m b^m c^m \mid m \geq 1\}.$$

Let us prove that L is **not** context-free.

(1) For given n choose the word $z = a^n b^n c^n$ from the language L.

(2) Let $z = uvwxy$ be a division of z into 5 segments such that

$$|vwx| \leq n \quad \text{and} \quad vx \neq \varepsilon.$$

We must analyze possible choices of u, v, w, x, y. Since $|vwx| \leq n$, words v and x cannot contain both letters a and c: In z, letters a are namely separated from all letters c by b^n.

So letter a or c does not exist in v and x. Therefore the word

$$uv^2wx^2y$$

contains more some letters than others: By adding one v and x we have increased the number of some letters, while the number of one letter (a or c) has remained equal to n. Conclusion: Choice $i = 2$ gives

$$uv^iwx^iy \notin L.$$

□

Example 88. Another example:

$$L = \left\{a^m b^k c^m d^k \mid m, k \geq 1\right\}.$$

□

3.6 Closure properties of CFL

Recall: We say that the family of context-free languages is closed under language operation *Op* if

$$L_1, L_2, \ldots \text{ are CFL's } \implies Op(L_1, L_2, \ldots) \text{ is a CFL,}$$

that is, if operation *Op* is applied to context-free languages, the result is also context-free. We say that the closure is effective if there is a mechanical procedure (=algorithm) that constructs the result $Op(L_1, L_2, \ldots)$ for any context-free input languages L_1, L_2, ... Inputs and outputs are given in the form of PDA or context-free grammar – it does not matter which format is used since both devices can be algorithmically converted into each other.

Theorem 89 *The family of CFL is effectively closed under the following operations:*

- *Union (if L_1, L_2 are CFL, so is $L_1 \cup L_2$),*
- *Concatenation (if L_1, L_2 are CFL, so is L_1L_2),*

- *Kleene closure (if L is CFL, so is L^*),*
- *Substitutions with CFL (if L is CFL, and f is a substitution such that for every letter $a \in \Sigma$ the language $f(a)$ is CFL, then $f(L)$ is CFL),*
- *Homomorphisms (if L is CFL, and h is a homomorphism then $h(L)$ is CFL),*
- *Inverse homomorphisms (if L is CFL, and h is a homomorphism then $h^{-1}(L)$ is CFL),*
- *Intersections with **regular** languages (if L is CFL and R is a regular language, then $L \cap R$ is CFL).*

*The family of CFL is **not** closed under the following operations:*

- *Intersection (for some CFL L_1 and L_2 the language $L_1 \cap L_2$ is **not** a CFL),*
- *Complementation (for some CFL L the language $\overline{L}$ is **not** a CFL),*
- *Quotient with arbitrary languages (for some CFL L and language K the language L/K is **not** a CFL).*

Proof. 1. **Union and concatenation**. Let L_1 and L_2 be generated by the grammars

$$G_1 = (V_1, T_1, P_1, S_1) \text{ and } G_2 = (V_2, T_2, P_2, S_2),$$

respectively. We may assume that the two variable sets V_1 and V_2 are disjoint, i.e. $V_1 \cap V_2 = \emptyset$. (If necessary we just rename some variables.)

The union $L_1 \cup L_2$ is generated by the grammar

$$G_3 = (V_1 \cup V_2 \cup \{S_3\}, T_1 \cup T_2, P_3, S_3)$$

where S_3 is a new start symbol, and P_3 contains all productions in P_1 and P_2, and the additional "initializing" productions

$$S_3 \longrightarrow S_1 \mid S_2.$$

The first derivation step by G_3 produces either S_1 or S_2, and after that a derivation by G_1 or G_2 is simulated.

The concatenation L_1L_2 is generated by the grammar

$$G_4 = (V_1 \cup V_2 \cup \{S_4\}, T_1 \cup T_2, P_4, S_4)$$

where S_4 is a new start symbol, and P_4 contains all productions in P_1 and P_2, and the additional initializing production

$$S_4 \longrightarrow S_1S_2.$$

After the first derivation step the sentential form is S_1S_2, and from S_1 and S_2 we can derive words of L_1 and L_2, respectively.

2. **Kleene closure**. Let L be generated by the grammar

$$G = (V, T, P, S).$$

Then L^* is generated by the grammar

$$G' = (V \cup \{S'\}, T, P', S')$$

where S' is a new start symbol, and P' contains all productions in P, and the productions

$$S' \longrightarrow \ SS' \mid \varepsilon.$$

From S' one can generate an arbitrarily long sequence $SS \ldots S$

$$S' \Rightarrow SS' \Rightarrow \ldots \Rightarrow SS \ldots SS' \Rightarrow SS \ldots S,$$

and each S produces a word of L.

3. **Substitution**. Let $L \subseteq \Sigma^*$ be generated by the grammar

$$G = (V, \Sigma, P, S),$$

and, for each $a \in \Sigma$, let $f(a)$ be generated by the grammar

$$G_a = (V_a, T_a, P_a, S_a).$$

Assume that all variable sets V and V_a are disjoint. The following grammar generates $f(L)$:

$$G' = (V', T', P', S),$$

where

$$V' = V \cup \left(\bigcup_{a \in \Sigma} V_a \right)$$

contains all variables of all grammars G and G_a,

$$T' = \bigcup_{a \in \Sigma} T_a$$

contains all terminals of grammars G_a. Productions P' include all productions that are in the sets P_a and, in addition, for every production in P there is a production obtained by replacing all occurrences of terminals a by the start symbol S_a of the corresponding grammar G_a.

Example 90. If $L = L(G)$ is generated by the grammar

$$G = (\{X\}, \{a, b\}, P, X)$$

with productions

$$X \longrightarrow aXb \mid bXa \mid \varepsilon$$

and if f is the substitution such that the language $f(a)$ is generated by the grammar

$$G_a = (\{Y\}, \{0, 1\}, P_a, Y)$$

with

$$Y \longrightarrow 0YY0 \mid 1$$

and $f(b)$ is the language generated by the grammar

$$G_b = (\{Z\}, \{0,1\}, P_b, Z)$$

with

$$Z \longrightarrow 0Z1Z \mid \varepsilon,$$

then $f(L)$ is generated by the grammar $G' = (V', T', P', S')$ where

$$\begin{array}{rcl} V' & = & \{X, Y, Z\}, \\ T' & = & \{0,1\}, \\ S' & = & X, \end{array}$$

and P' contains the productions

$$\begin{array}{rcl} X & \longrightarrow & YXZ \mid ZXY \mid \varepsilon, \\ Y & \longrightarrow & 0YY0 \mid 1, \\ Z & \longrightarrow & 0Z1Z \mid \varepsilon. \end{array}$$

□

4. **Homomorphism**. This follows from the previous proof because homomorphism is a special type of substitution. We can also make a direct construction where we replace in the productions all occurrences of terminal letters by their homomorphic images.

Example 91. If L is generated by the grammar

$$\begin{array}{rcl} E & \longrightarrow & E + E \mid N \\ N & \longrightarrow & 0N \mid 1N \mid 0 \mid 1 \end{array}$$

and the homomorphism h is given by $h(+) = \times$, $h(0) = ab$, $h(1) = ba$ then $h(L)$ is generated by

$$\begin{array}{rcl} E & \longrightarrow & E \times E \mid N \\ N & \longrightarrow & abN \mid baN \mid ab \mid ba \end{array}$$

□

5. **Inverse homomorphism**. In this case it is easier to use PDA instead of grammars. Let $L \subseteq \Delta^*$ be recognized by the PDA

$$M = (Q, \Delta, \Gamma, \delta, q_0, Z_0, F)$$

using final states, and let

$$h : \Sigma^* \longrightarrow \Delta^*$$

be a homomorphism. Let us construct a PDA M' that recognizes the language $h^{-1}(L)$ using final states, that is, it accepts all words w such that $h(w) \in L$.

On the input w, machine M' computes $h(w)$ and simulates M with input $h(w)$. If $h(w)$ is accepted by M then w is accepted by M'. The result of h on w is stored on a buffer, or "virtual input tape", inside the control unit. That is where the simulation of M reads its input from. $h(w)$ is computed from w letter-by-letter as needed: as soon as the virtual tape becomes empty the next "real" input letter a is scanned, and $h(a)$ is added on the virtual tape. Let

$$B = \{u \in \Delta^* \mid \exists a \in \Sigma \;\; u \text{ is a suffix of } h(a) \}.$$

Set B consists of all possible contents of the "virtual" input tape. Set B is of course a finite set.

Let us construct the PDA

$$M' = (Q', \Sigma, \Gamma, \delta', S', Z_0, F'),$$

where

$$\begin{aligned} Q' &= Q \times B, \\ S' &= [q_0, \varepsilon], \text{ and} \\ F' &= F \times \{\varepsilon\}. \end{aligned}$$

Function δ' contains two types of productions:

- For every transition of the original machine M we have the simulating transition in M': If

  $$(p, \gamma) \in \delta(q, a, Z)$$

  then

  $$([p, x], \gamma) \in \delta'([q, ax], \varepsilon, Z)$$

  for every $ax \in B$. Here $a \in \Delta \cup \{\varepsilon\}$. Notice that M' reads the input letter a from the virtual tape.

- When the virtual tape becomes empty, we have a transition for reading the next real input letter a, and loading its homomorphic image $h(a)$ to the virtual tape: For all $a \in \Sigma$, $q \in Q$, and $Z \in \Gamma$ we have the transition

  $$([q, h(a)], Z) \in \delta'([q, \varepsilon], a, Z).$$

Initially the virtual tape is empty, so the initial state of M' is

$$[q_0, \varepsilon].$$

The input word is accepted if in the end the virtual input tape is empty (M has consumed the whole input word), and the state of M is a final state. Therefore the final states of M' are the elements of

$$F \times \{\varepsilon\}.$$

Example 92. Consider the PDA

$$M = (\{q\}, \{a, b\}, \{A\}, \delta, q, A, \{q\})$$

where

$$\begin{aligned} \delta(q, a, A) &= \{(q, AA)\} \\ \delta(q, b, A) &= \{(q, \varepsilon)\} \end{aligned}$$

and let h be the homomorphism $h(0) = b$, $h(1) = aa$. Let us construct a PDA that recognizes $h^{-1}(L(M))$. Possible contents of the virtual tape are

$$B = \{b, \varepsilon, aa, a\}$$

so the state set of the new PDA will be

$$Q' = \{[q, b], [q, \varepsilon], [q, aa], [q, a]\}.$$

The initial state is

$$S' = [q, \varepsilon],$$

and there is only one final state

$$F' = \{[q, \varepsilon]\}.$$

From the original transition $(q, AA) \in \delta(q, a, A)$ we get two simulating transitions:

$$\begin{array}{rcl} ([q, a], AA) & \in & \delta'([q, aa], \varepsilon, A), \\ ([q, \varepsilon], AA) & \in & \delta'([q, a], \varepsilon, A), \end{array}$$

and from $(q, \varepsilon) \in \delta(q, b, A)$ one transition:

$$([q, \varepsilon], \varepsilon) \in \delta'([q, b], \varepsilon, A).$$

Finally, we have the following transitions for loading the virtual tape:

$$\begin{array}{rcl} ([q, aa], A) & \in & \delta'([q, \varepsilon], 1, A), \\ ([q, b], A) & \in & \delta'([q, \varepsilon], 0, A). \end{array}$$

□

Example 93. Let us show that the language

$$L = \{a^n b^{2n} c^{3n} \mid n \geq 1\}$$

is not context-free. Define the homomorphism $h(a) = a$, $h(b) = bb$, $h(c) = ccc$. Then

$$h^{-1}(L) = \{a^n b^n c^n \mid n \geq 1\}$$

which we know is not context-free (Example 87). So L cannot be CFL either: if it were, then $h^{-1}(L)$ would be CFL as well. □

6. **Not closed under Intersection**. It is enough to find one counter example. Let

$$L_1 = \{a^n b^n c^m \mid n, m \geq 1\}$$

and

$$L_2 = \{a^m b^n c^n \mid n, m \geq 1\}.$$

Both L_1 and L_2 are context-free: L_1 is the concatenation of context-free languages

$$\{a^n b^n \mid n \geq 1\} \text{ and } c^+,$$

and L_2 is the concatenation of

$$a^+ \text{ and } \{b^n c^n \mid n \geq 1\}.$$

(One can easily also construct grammars for both L_1 and L_2.) But even though L_1 and L_2 are context-free, their intersection

$$L_1 \cap L_2 = \{a^n b^n c^n \mid n \geq 1\}$$

is not. So the family of CFL is not closed under intersection.

7. **Not closed under complementation**. If they were, then the family of CFL would be closed under intersection as well:

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

a contradiction with 6 above.

8. **Not closed under quotient with arbitrary language**. Homework.

9. **Intersection with regular languages**. Even though the intersection of two context-free languages may be non-context-free, as proved in 6 above, the intersection of any context-free language with a regular language is always context-free.

Let L be recognized by the PDA

$$M = (Q_M, \Sigma, \Gamma, \delta_M, q_M, Z_0, F_M)$$

using final states, and let R be recognized by the DFA

$$A = (Q_A, \Sigma, \delta_A, q_A, F_A).$$

Let us construct a PDA M' that runs both M and A in parallel. The state set of M' is

$$Q' = Q_M \times Q_A.$$

The states are used to store the states of both M and A for the input that has been read so far.

All ϵ-transitions of M are simulated as such, without changing the Q_A-component of the state. But whenever M reads next input letter, the Q_A-component is changed according to DFA A. Clearly, if in the end of the input word both Q_M- and Q_A-components are final states, the word is accepted by both M and A.

So we have the PDA

$$M' = (Q_M \times Q_A, \Sigma, \Gamma, \delta', [q_M, q_A], Z_0, F_M \times F_A)$$

where δ' is the following: Each transition

$$(q', \gamma) \in \delta_M(q, a, Z)$$

by the original PDA, and each state $p \in Q_A$ of the DFA provide the transition

$$([q', \delta_A(p, a)], \gamma) \in \delta'([q, p], a, Z)$$

into the new machine M'. The first state component and the stack follow the instruction of the PDA M, while the the second state component simulates the DFA A: On input a, state p is changed to $\delta_A(p, a)$. Note that if $a = \varepsilon$ then $\delta_A(p, a) = p$.

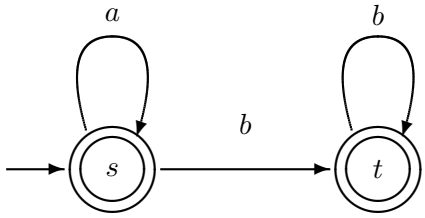
Example 94. Consider the PDA

$$M = (\{q\}, \{a, b\}, \{A\}, \delta, q, A, \{q\})$$

where

$$\begin{aligned} \delta(q, a, A) &= \{(q, AA)\} \\ \delta(q, b, A) &= \{(q, \varepsilon)\} \end{aligned}$$

and let $R = a^*b^*$. Then R is recognized by the DFA



The intersection $L(M) \cap R$ is recognized by the PDA

$$M' = (Q', \{a, b\}, \{A\}, \delta', q_0, A, F)$$

where

$$\begin{array}{lcl} Q' & = & \{[q,s],[q,t]\}, \\ q_0 & = & [q,s], \\ F & = & \{[q,s],[q,t]\}, \end{array}$$

and δ' contains the transitions

$$\begin{array}{lcl} \delta'([q,s],a,A) & = & \{([q,s],AA)\}, \\ \delta'([q,s],b,A) & = & \{([q,t],\varepsilon)\}, \\ \delta'([q,t],a,A) & = & \emptyset, \\ \delta'([q,t],b,A) & = & \{([q,t],\varepsilon)\}. \end{array}$$

□

We have covered all the operations stated in Theorem 89, so the proof of the theorem is complete. □

Example 95. Let us prove that

$$L = \{0^n 1 0^n 1 0^n \mid n \geq 1\}$$

is not context-free. Assume that it would be a CFL. Define the substitution $f(0) = \{a, b, c\}$, $f(1) = \{\#\}$. Since we assumed that L is context-free, so is

$$L_1 = f(L) \cap (a^+ \# b^+ \# c^+) = \{a^n \# b^n \# c^n \mid n \geq 1\}.$$

Define then the erasing homomorphism $h(a) = a$, $h(b) = b$, $h(c) = c$, $h(\#) = \varepsilon$. Then the language

$$L_2 = h(L_1) = \{a^n b^n c^n \mid n \geq 1\}$$

would be context-free. But we know from Example 87 that L_2 is not context-free, a contradiction. Our initial assumption that L is a CFL, must be incorrect. □

Example 96. Let us show that the language

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

is not context-free. If it were CFL, so would be

$$L_1 = L \cap (a^+b^+a^+b^+) = \{a^m b^k a^m b^k \mid m, k \geq 1\}.$$

Define then the substitution $f(a) = \{a, c\}$, $f(b) = \{b, d\}$. If L_1 is a CFL so would be

$$L_2 = f(L_1) \cap (a^+b^+c^+d^+) = \{a^m b^k c^m d^k \mid m, k \geq 1\}.$$

But we used the pumping lemma in Example 88 to prove that L_2 is not context-free, so the original language L is not context-free either. □

Example 97. Let us show that family of context-free languages is closed under quotient with regular languages. Let L be a context-free language, and let R be a regular language.

Let Σ be the union of the alphabets of L and R, and let

$$\Sigma' = \{a' \mid a \in \Sigma\}$$

be a new alphabet obtained by marking symbols of Σ. We may assume Σ and Σ' are disjoint. If w is a word over Σ then w' will denote the corresponding word over Σ' obtained by marking every letter of w.

Consider the finite substitution s defined on alphabet Σ as follows

$$s(a) = \{a, a'\}$$

for every $a \in \Sigma$. It follows from the closure properties that the language

$$L_1 = s(L) \cap (\Sigma'^* R)$$

is context-free.

Let us analyze L_1. It consists of all words $w'u$ where $wu \in L$ and $u \in R$. If we delete non-marked letters from L_1 and remove marks, then we obtain words of the quotient L/R. So let h be the homomorphism $h : \Sigma \cup \Sigma' \longrightarrow \Sigma^*$ defined by

$$\begin{array}{lcll} h(a) & = & \varepsilon & \text{if } a \in \Sigma, \text{ and} \\ h(a') & = & a & \text{if } a' \in \Sigma'. \end{array}$$

Then

$$L/R = h(L_1),$$

which proves that L/R is context-free. □

We proved in the example above that if L is context-free and R is regular then L/R is context-free. We know (see the homework) that there is some (unrestricted) language K such that L/K is not context-free. This raises the following natural question: If L and K are context-free is L/K guaranteed to be context-free ? (Answer is "no" but can you come up with a counter example ?)

3.7 Decision algorithms

Now we give algorithms for deciding if a given CFL is empty, finite or infinite. Also a polynomial time parsing algorithm for deciding if a given word belongs to a given CFL is presented.

Later on, we'll see that there are many questions concerning CFL that do not have algorithms. For example, there do not exist any algorithms for deciding if a given CFG is ambiguous, if given two CFG's are equivalent (i.e. define the same language), if a given CFG generates Σ^*, if the intersection of two given CFL's is empty, etc.

Note that a CFL can be represented in different ways: as a grammar, or as a PDA that recognizes the language. It does not matter which representation we use since we have algorithms for converting a CFG into equivalent PDA and vice versa.

Theorem 98 *There are algorithms to determine if a given CFL is (a) empty, (b) finite, (c) infinite.*

Proof. (a) In Section 3.3 we used a marking procedure to find all variables that derive a terminal word. The language $L(G)$ is non-empty if and only if the start symbol S of the grammar G gets marked.
(b) and (c). To test wether a given grammar generates finitely or infinitely many words, we start by simplifying the grammar, and converting it into Chomsky normal form. After this we only have productions of types

$$\begin{aligned} A &\longrightarrow BC, \\ A &\longrightarrow a, \end{aligned}$$

and we don't have useless symbols, i.e., symbols that do not derive terminal words and/or are not reachable from the start symbol. (We may lose ε but it does not matter since L is finite if and only if $L - \{\varepsilon\}$ is finite.)

So we can assume grammar G is in CNF, and all variables are useful. We'll use the fact that grammar G generates an infinite language if and only if some variable can derive a word containing the variable itself. Let us call a variable A **self-embedded** if

$$A \Rightarrow^+ \alpha A \beta$$

for some words α and β. Since the grammar is in CNF, α and β are not both empty words. Let us prove that $L(G)$ is infinite if and only if G contains a self-embedded variable:

(1) Assume $A \Rightarrow^+ \alpha A \beta$. Then (as in the pumping lemma)

$$A \Rightarrow^+ \alpha^i A \beta^i$$

for every $i \geq 1$. Since the grammar does not contain useless symbols every word of terminals and variables derives some terminal word. If x, y, w are terminal words derivable from α, β and A, respectively, then

$$A \Rightarrow^+ \alpha^i A \beta^i \Rightarrow^* x^i w y^i$$

for every $i \geq 1$. So A derives infinitely many words (because x or y is non-empty), and $L(G)$ must be infinite. (G does not contain useless symbols, so A is reachable from the start symbol.)

(2) Assume that the language $L(G)$ is infinite. Then it contains a word w which is longer than the number n in the pumping lemma. As in the proof of the pumping lemma we know that the derivation of w contains a self-embedded variable. So G has at least one self-embedded variable.

To determine if $L(G)$ is infinite we have to find out whether any variable is self-embedded. We can use a marking procedure to determine if variable A is self-embedded:

```
Self-embedded(G, A)

1. Initially unmark all variables X
2. For every production A ⟶ α of A do
3.   mark every variable X that appears in α
4. Repeat
5.   For every production X ⟶ α in G do
6.     If X is marked then mark every variable of α
7. Until no new variables were marked on line 6
8. If A is marked then return TRUE
9.   else return FALSE
```

□

Next we turn to the parsing problem.

Theorem 99 *There is an algorithm to determine whether a given word w is in a given CFL L.*

Proof. If $w = \varepsilon$ then we simply check whether the start symbol S is nullable in the grammar for L, as done in Section 3.3. Assume then that $|w| \geq 1$. Let G be a grammar in the Chomsky normal form for language L, or $L \setminus \{\varepsilon\}$. Grammar G can be effectively constructed.

In the following we describe (an inefficient) recursive algorithm to test whether a given variable A derives a given non-empty word w. The recursion is based on the fact that $A \Rightarrow^* w$ if and only if either

- $|w| = 1$ and $A \longrightarrow w$ is a production in G, or
- $|w| \geq 2$ and there exists a production $A \longrightarrow BC$ in the grammar, and a division $w = uv$ of w into two non-empty words u and v, such that variables B and C derive u and v, respectively.

```
Derives(G, A, a_1 a_2 ... a_n)
% Does A ⇒* a_1 a_2 ... a_n (where n ≥ 1) in the given CNF grammar G?

1. If n = 1 then if A ⟶ a_1 in G then return TRUE else return FALSE
2. For i ⟵ 2 to n do
3.   For all productions A ⟶ BC in G do
4.     If Derives(G, B, a_1 ... a_{i-1}) and Derives(G, C, a_i ... a_n) then return TRUE
5. return FALSE
```

The call `Derives(`G, S, w`)` with the start symbol S returns `TRUE` if and only if w is in L. The algorithm clearly always halts and returns the correct answer. □

The algorithm presented in the proof is very inefficient: it may require an amount of time which is exponential with respect to the length of w. In this course we are usually only interested in the existence of algorithms — not so much their efficiency — but since parsing context-free grammars is an important problem with applications in compiler design, let us discuss a way to make the algorithm faster.

The standard technique of **dynamic programming** can be used to convert the exponential time recursive algorithm above into an iterative polynomial time algorithm. The idea is based on the observation that during the recursive computation of `Derives(`G, S, w`)` the same recursive calls get repeated over-and-over again, with the same exact call parameters. So rather than recomputing the results of the recursive calls over-and-over again, let us compute them only once and memorize the result in an array. For this purpose we introduce for each variable A of the CNF grammar an $n \times n$ Boolean matrix $A_{n\times n}$, where n is the length of the word $w = a_1a_2 \dots a_n$ we are parsing. The element A_{ij} of the matrix will store value `TRUE` iff variable A derives the subword $a_ia_{i+1} \dots a_{i+j-1}$ of length j that begins in position i, for all $j = 1, 2, \dots, n$ and all $i = 1, 2, \dots, n-j+1$. The matrix can be filled up incrementally for larger and larger values of j:

1° ($j = 1$) We set A_{i1} to TRUE if and only if $A \longrightarrow a_i$ is a production in the grammar, and

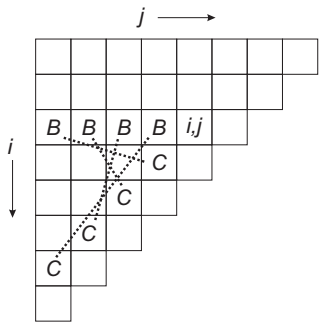
2° ($j > 1$) We set A_{ij} to TRUE if and only if for some $k < j$ there is a production $A \longrightarrow BC$ such that the entries B_{ik} and $C_{i+k,j-k}$ are TRUE, indicating that variables B and C derive the subwords of length k and $j-k$ starting in positions i and $i+k$, respectively.

The resulting algorithm (known as the **CYK-algorithm** for three independent inventors Cocke, Younger and Kasami) runs in time $O(n^3)$.

```
CYK(G, a_1 a_2 ... a_n)
% G is in the Chomsky normal form and n ≥ 1

1. For every variable A and all i,j ≤ n initialize A_{i,j} ⟵ FALSE
2. For i ⟵ 1 to n do
3.   For all productions A ⟶ a_i do
4.     set A_{i,1} ⟵ TRUE
5. For j ⟵ 2 to n do
6.   For i ⟵ 1 to n − j + 1 do
7.     For k ⟵ 1 to j − 1 do
8.       For all productions A ⟶ BC do
9.         If B_{i,k} and C_{i+k,j−k} are TRUE then set A_{i,j} ⟵ TRUE
10.Return S_{1,n}.
```

The following illustration shows which entries of the matrices for B and C are compared when determining A_{ij} on line 9 of the algorithm:

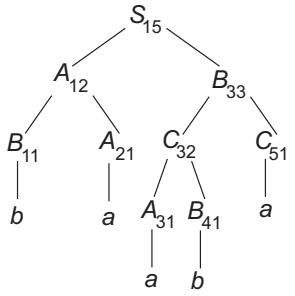


If the input word is generated by the grammar, we usually want to parse it: we want to find a derivation of the word in the grammar. This can be established in the CYK algorithm simply by recording in each `TRUE` matrix element also the reason why the element was set `TRUE`, that is, which two `TRUE` matrix entries were used on line 9. In the end, if the entry S_{1n} is `TRUE`, we can parse the word by following these pointers till the single letters.

Example 100. Consider the CNF grammar $G = (\{S, A, B, C\}, \{a, b\}, P, S)$ where P contains the productions

$$\begin{array}{lcl} S & \longrightarrow & AB \mid BC \\ A & \longrightarrow & BA \mid a \\ B & \longrightarrow & CC \mid b \\ C & \longrightarrow & AB \mid a \end{array}$$

Let us use the CYK algorithm to find out that the word $w = baaba$ is in the language $L(G)$. The parsing discovered by the algorithm is summarized in the derivation tree



□

4 Recursive and recursively enumerable languages

In this last part of the course we want to show that certain **decision problems** can not be algorithmically solved. By a decision problem we mean a computational question with a well defined yes/no-answer. For example, questions like

- "Does a given regular expression define an infinite language ?",
- "Is a given context-free grammar ambiguous ?", or

- "Does a given quadratic equation have a real number solution ?"

are decision problems. A decision problem has an input parameter (identified by the expression "given..." in the examples above), and values of the parameter are called **instances** of the problem. An instance is **positive** or **negative** if the answer to the decision problem is "yes" or "no", respectively. For example, a^*b and $ab + ba$ are a positive and a negative instance of the first problem above.

We say that a decision problem is **decidable** if there exists an algorithm that, when given an instance as input, returns the correct yes/no-answer. The algorithm has to work for all instances correctly. The computation may take any finite amount of time, but for every instance the computation halts after some time and gives the correct answer. A problem is called **undecidable** if no such algorithm exists. We'll see that many important and natural decision problems are undecidable.

Note that it does not make sense to talk about decision problems that have only one instance: such problems are of course decidable since there is a trivial algorithm that simply writes out "yes" or "no" — we just may not know which of the two algorithms is the correct one. For this reason questions like "Is $P = NP$?" or "Is the Riemann hypothesis true ?" are trivial to us: they don't have different input instances. For the same reason, decision problems with only finitely many different instances are decidable since the correct yes/no -answers can be summarized in a finite table. So only problems with infinitely many different instances are considered.

To connect decision problems to formal languages, we encode instances of problems as words over some alphabet. This is reasonable: when typing an instance into a computer in order to run an algorithm we are encoding it, for example, as a word over the binary alphabet $\{0, 1\}$. It is up to us to determine which encoding we use: As long as two encodings can be effectively converted into each other, the problem cannot be decidable in one presentation and undecidable in the other one. It is also reasonable to require that the encoding scheme is such that one can effectively recognize those words that are not encodings of any input instance. In the following, we denote by $\langle I \rangle$ the word that is the encoding of instance I.

We associate to a decision problem $\mathcal{P}$ a language $L_{\mathcal{P}}$ that contains the encodings of all positive instances. (This language, naturally, depends also on the encoding scheme we choose to use.) Then the decision problem $\mathcal{P}$ becomes the problem of determining if a given word w is in the language $L_{\mathcal{P}}$ or not. We define the **membership problem** of language L as the following decision problem:

Membership in $L \subseteq \Sigma^*$
Instance: Word $w \in \Sigma^*$
Problem: Is $w \in L$?

Language L whose membership problem is decidable is called **recursive**. As we have not yet precisely defined the concept of an algorithm (and hence that of decidability), this definition is not yet mathematically stated here. It is an informal description of recursive languages that is useful to keep in mind: if one can design an algorithm (e.g. a computer program) for determining if any given word w is in the language L then L is recursive. In the following sections we provide a precise definition of recursive languages in terms of Turing machines.

A **semi-algorithm** for a decision problem is an algorithm like process that correctly returns value "yes" for positive instances, but on negative instances the semi-algorithm either returns answer "no" or it does not return any value, i.e. it runs for ever without ever halting. Notice that

there is a non-symmetry between the positive and negative instances: A semi-algorithm for positive instances does not imply that there would be also a semi-algorithm for the negative instances. The decision problem obtained by swapping the negative and positive instances is called the **complement** of the original problem. A decision problem is called **semi-decidable** if its positive instances have a semi-algorithm.

A language L is **recursively enumerable** (or simply **r.e.**) if its membership problem is semi-decidable. So the decidability (semi-decidability) of decision problem $\mathcal{P}$ corresponds to the recursiveness (recursive enumerability, respectively) of the corresponding language $L_{\mathcal{P}}$ of positive instances. For this reason, recursive languages are sometimes called decidable languages and r.e. languages are sometimes called semi-decidable languages.

Based on this informal definition we can observe the following apparent facts:

(i) If a problem is decidable then also the complement problem is decidable. (That is, the family of recursive languages is closed under complementation.)

(ii) Every decidable problem is also semi-decidable. (All recursive languages are also recursively enumerable.)

(iii) If a problem and its complement are both semi-decidable then the problem is decidable. (A language L is recursive if and only if both L and its complement are r.e.)

For observation (iii) note that the two semi-algorithms for the problem and its complement can be executed concurrently until one of them returns the answer.

It is not very surprising that there are languages that are not recursive or even recursively enumerable. (We can reason as follows: The number of different r.e. languages is countably infinite. This follows from the fact that the number of different algorithms is countably infinite. But the number of different languages is uncountably infinite, i.e. there exist more languages than there are r.e. languages.) What is more surprising is that we can actually find individual languages that correspond to very natural decision problems and are non-recursive.

4.1 Turing machines

In order to precisely define recursive and recursively enumerable languages we must have a definition of an algorithm. We do this using Turing machines: We say a language is r.e. iff it is recognized by a Turing machine, and it is recursive iff it is recognized by a Turing machine that halts on every input.

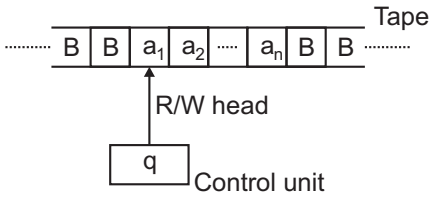
At first, Turing machines may seem like very restricted models of computers, but they are in fact as powerful as any other model people have come up with so far. For example, we'll see that Turing machines can simulate register machines (a model closer to the modern computer architecture).

One cannot, of course, **prove** that nobody ever invents a totally different, more powerful computer architecture. But all evidence indicates that this is not possible, as long as we require that the computer executes mechanically a finitely describable program on discrete steps. The **Church-Turing thesis** states that every effectively computable function can be computed by a Turing machine. This is an unprovable statement since the concept of "effectively computable" has no precise definition – the thesis asserts that Turing machines are a proper meaning of "effective computability".

A Turing machine (TM) has a finite state control similar to PDA. Instead of an input tape and a stack, the machine has only one tape that is both the input tape and the "work" tape. The machine has a read/write head that may move left or right on the tape. Initially, the input is written on the tape. At each time step the machine reads the tape symbol under its read/write head, and depending on the tape symbol and the state of the control unit, the machine may

1. Change its state,
2. Replace the symbol on the tape under the read/write head by another tape symbol, and
3. move the R/W head left or right on the tape.

The tape is infinite to the left and to the right.



Initially the input word is written on the tape in such a way that the read-write head scans the first letter of the input. All other tape locations contain a special blank symbol B. The input is accepted if and only if the machine eventually enters the final state.

Formally, a (deterministic) Turing machine is 7-tuple

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, f)$$

where

- Q is the finite **state set** of the control unit.
- Σ is the **input alphabet**.
- Γ is the **tape alphabet** containing all allowed symbols that may be on the tape. Especially $\Sigma \subset \Gamma$ since initially the input is written on the tape. We assume that $Q \cap \Gamma = \emptyset$ so that there is no confusion between states and tape letters.
- δ is a **transition function**, described below in detail.
- $q_0 \in Q$ is the **initial state**.
- $B \in \Gamma \setminus \Sigma$ is a special **blank symbol**. It is not part of the input alphabet Σ.
- $f \in Q$ is the **final state**.

The transition function δ is a (partial) mapping from the set

$$(Q \setminus \{f\} \times \Gamma$$

into the set

$$Q \times \Gamma \times \{L, R\}.$$

The mapping is partial: it may be undefined for some arguments, in which case the machine has no possible next move, and it halts. In particular, there is no transition from the final state f. Transition

$$\delta(q, X) = (p, Y, L)$$

means that in state q, scanning tape symbol X, the machine changes its state to p, replaces X by Y on the tape, and moves the R/W head one cell to the left on the tape. Transition

$$\delta(q, X) = (p, Y, R)$$

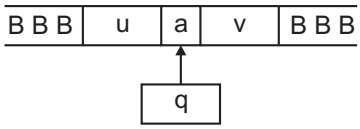
is defined analogously, only the R/W head moves to the right.

Initially, the input word is written on the tape, with all other cells containing the blank symbol B. The machine is in state q_0, and the R/W head is positioned on the leftmost letter of the input. If after some moves the machine enters the final state f the input word is accepted, otherwise it is rejected. (It is rejected if the machine never halts, or if it halts in a non-final state when there is no next move.)

Let us define **instantaneous descriptions** (IDs) of Turing machines that describe its present configuration: An ID is a word

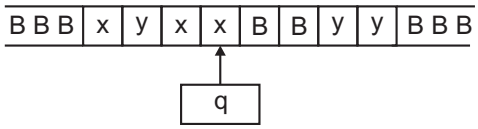
$$u\ q\ a\ v$$

where $q \in Q$ is the state of the control unit, $u, v \in \Gamma^*$ are the content of the tape to the left and to the right of the R/W head, respectively, until the last non-blank symbols, and $a \in \Gamma$ is the symbol currently scanned by the R/W head:

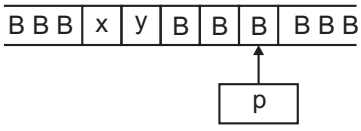


So the first (last) letter of u (or v, respectively) is not B, because the leading and trailing B's are trimmed away from u and v, respectively. Note also that the current state q can be uniquely identified from the word $u\ q\ a\ v$ because we required alphabets Q and Γ to be disjoint.

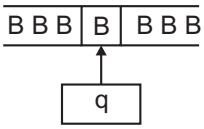
For example, the ID $xyxqxBByy$ (where $q \in Q$) represents the situation



while $xyBBpB$ (where $p \in Q$) represents



and qB represents



On the other hand, BqB and $xyBqBxBB$ are not valid ID's since the surrounding B's have not been removed. Precisely speaking, ID's are words of the set

$$(\{\varepsilon\} \cup (\Gamma \setminus \{B\})\Gamma^*)\ Q\ \Gamma\ (\Gamma^*(\Gamma \setminus \{B\}) \cup \{\varepsilon\}).$$

Let us define **moves** of the Turing machine. Let the current ID be

$$\alpha = u\ q\ a\ v.$$

If β is the next ID we denote $\alpha \vdash \beta$ and say that β results from α by one move. More precisely:

1. Assume that
$$\delta(q, a) = (p, b, L).$$
   - If $u = \varepsilon$ then the next ID is $\beta = pBbv$, except that possible trailing B's are removed from bv.
   - If $u = u'c$ for some $c \in \Gamma$ then the next ID is $\beta = u'pcbv$, except that possible trailing B's are removed from bv.
2. Assume that
$$\delta(q, a) = (p, b, R).$$
   - If $v = \varepsilon$ then the next ID is $\beta = ubpB$ except that possible leading B's are removed from ub.
   - If $v \neq \varepsilon$ then the next ID is $u = ubpv$, except that possible leading B's are removed from ub.
3. If $\delta(q, a)$ is undefined then no move from α is possible, and α is a **halting** ID. If $q = f$ then α is an **accepting** ID.

Our TM model is **deterministic**, which means that for each α there is at most one β such that $\alpha \vdash \beta$. As usual, we write

$$\alpha \vdash^* \beta$$

if the TM changes α into β in any number of moves (including 0 in which case $\alpha = \beta$), we denote

$$\alpha \vdash^+ \beta$$

if the TM changes α into β using at least one move, and we denote

$$\alpha \vdash^i \beta$$

if the TM changes α into β in exactly i moves.

For any $w \in \Sigma^*$ we define the corresponding initial ID

$$\iota_w = \begin{cases} q_0 w, & \text{if } w \neq \varepsilon, \\ q_0 B, & \text{if } w = \varepsilon. \end{cases}$$

The language recognized (or accepted) by the TM M is

$$L(M) = \{w \mid w \in \Sigma^* \text{ and } \iota_w \vdash^* ufv \text{ for some } u, v \in \Gamma^*\ \}.$$

Example 101. Let us design a TM M that recognizes the language

$$L(M) = \{w \in \{a,b\}^* \mid \ w \text{ is a palindrome } \}.$$

We have

$$M = (\{q_0, q_F, q_L, q_a, q'_a, q_b, q'_b\}, \{a,b\}, \{a,b,B\}, \delta, q_0, B, q_F)$$

where δ is detailed below.

The idea is that the machine reads the first input letter from the tape, remembers it in the control unit (using states q_a and q_b), finds the last letter on the tape and compares it with the first letter. If they are identical, they are erased (replaced by B), and the process is repeated. If the whole input word gets erased (or if there remains only one input letter if the original word has odd length) the word was a palindrome, and the machine accepts it.

The transition function δ is defined as follows:

$\delta(q_0, B)$	$=$	(q_F, B, R)	Remaining input is empty
$\delta(q_0, a)$	$=$	(q_a, B, R)	Erase first input letter a
$\delta(q_a, a)$	$=$	(q_a, a, R)	
$\delta(q_a, b)$	$=$	(q_a, b, R)	
$\delta(q_a, B)$	$=$	(q'_a, B, L)	End of input found
$\delta(q'_a, B)$	$=$	(q_F, B, R)	Remaining input was empty
$\delta(q'_a, a)$	$=$	(q_L, B, L)	Erase last input letter a
$\delta(q_0, b)$	$=$	(q_b, B, R)	Erase first input letter b
$\delta(q_b, a)$	$=$	(q_b, a, R)	
$\delta(q_b, b)$	$=$	(q_b, b, R)	
$\delta(q_b, B)$	$=$	(q'_b, B, L)	End of input found
$\delta(q'_b, B)$	$=$	(q_F, B, R)	Remaining input was empty
$\delta(q'_b, b)$	$=$	(q_L, B, L)	Erase last input letter b
$\delta(q_L, a)$	$=$	(q_L, a, L)	move back to beginning
$\delta(q_L, b)$	$=$	(q_L, b, L)	
$\delta(q_L, B)$	$=$	(q_0, B, R)	beginning of input found

Here is the accepting computation for word $abba$:

$$q_0\ abba \vdash q_a\ bba \vdash b\ q_a\ ba \vdash bb\ q_a\ a \vdash bba\ q_a\ B \vdash bb\ q'_a\ a \vdash b\ q_L\ b \vdash q_L\ bb \vdash q_L\ Bbb \vdash$$

$$\vdash q_0\ bb \vdash q_b\ b \vdash b\ q_b\ B \vdash q'_b\ b \vdash q_L\ B \vdash q_0\ B \vdash q_F\ B,$$

and the accepting computation for word aba:

$$q_0\ aba \vdash q_a\ ba \vdash b\ q_a\ a \vdash ba\ q_a\ B \vdash b\ q'_a\ a \vdash q_L\ b \vdash q_L\ Bb \vdash q_0\ b \vdash q_b\ B \vdash q'_b\ B \vdash q_F\ B.$$

Word abb is not accepted because the computation halts in a non-accepting ID:

$$q_0\ abb \vdash q_a\ bb \vdash b\ q_a\ b \vdash bb\ q_a\ B \vdash b\ q'_a\ b.$$

□

A language is **recursively enumerable** (**r.e.**) if it is recognized by a Turing machine. A language is **recursive** if it is recognized by a Turing machine that halts on all inputs. Of course, every recursive language is also r.e. This is a precise mathematical definition of recursive and r.e. languages. In the next sections it becomes clear that it coincides with the informal definition given before.

Our sample TM halts on every input, so the palindrome language L is recursive. In fact, every context-free language is recursive. (We can implement the CYK algorithm on a Turing machine!)

4.2 Programming techniques for Turing machines

Writing down Turing machines for complicated languages can be difficult and boring. But one can use some programming techniques. The goal of this section is to convince the reader that Turing machines are indeed powerful enough to recognize any language that a computer program can recognize.

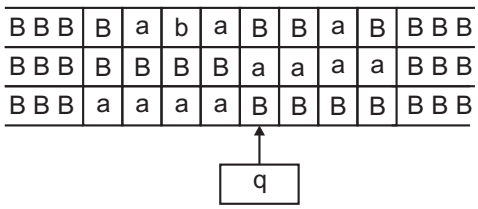
1. **Storing a tape symbol in the finite control**. We can build a TM whose states are pairs $[q, X]$ where q is a state, and X is a tape symbol. The second component can be used in remembering a particular tape symbol. We have already used this technique in Example 101 where states q_a and q_b were used to remember the tape symbol a or b. As another example, consider the following TM that recognizes the language

$$L = ab^* + ba^*$$

The machine reads the first symbol, remembers it in the finite control, and checks that the same symbol does not appear anywhere else in the input word:

$$\begin{array}{lcl}
\delta(q_0, a) & = & ([q, a], a, R) \\
\delta(q_0, b) & = & ([q, b], b, R) \\
\delta([q, a], b) & = & ([q, a], b, R) \\
\delta([q, b], a) & = & ([q, b], a, R) \\
\delta([q, a], B) & = & (q_F, B, R) \\
\delta([q, b], B) & = & (q_F, B, R)
\end{array}$$

2. **Multiple tracks**. Sometimes it is useful to imagine that the tape consists of multiple tracks. We can store different intermediate information on different tracks:



For example, we can construct a TM with 3 track tape that recognizes the language

$$L = \{a^p \mid p \text{ is a prime number } \}$$

as follows. Initially the input is written on the first track and the other two tracks contain B's. (This means we identify a with $[a, B, B]$ and B with $[B, B, B]$.) The machine operates as follows.

It first checks the small cases: If the input is empty or a then the machine halts in a non-final state; if the input is aa it halts in the final state. Otherwise, the machine starts by placing two a's on the second track. Then it repeats the following instructions:

1. Copy the content of the first track to the third track.
2. Subtract the number on the second track from the third track as many times as possible. If the third track becomes empty, halt in a non-final state. (The number on the first track was divisible by the number on the second track.)
3. Increment the number on the second track by one. If the number becomes the same as the number on the first track halt in the final state (no proper factor was found). Else go back to step 1.

3. **Checking off symbols**. This simply means that we introduce a second track where we can place blank B or symbol $\surd$. The tick mark can be conveniently used in remembering which letters of the input have been already processed. It is useful when we have to count or compare letters. For example, consider the language

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

We first use the tick mark to find the center of the input word: Mark alternatively the first and last unmarked letters, one-by-one. The last letter to be marked is in the center. So we know where the second w should start. Using the "**Storing a tape symbol in the finite control**" -technique, one can check one-by-one the letters to verify that the letters in the first half and the second half are identical.

4. **Shifting over**. This means adding an new cell at the current location of the tape. This can be established by shifting all symbols one position to the right by scanning the tape from the current position to the right, remembering the content of the previous cell in the finite control, and writing it to the next cell on the right. Once the rightmost non-blank symbol is reached the machine can return to the new vacant cell that was introduced. (In order to recognize the rightmost non-blank symbol, it is convenient to introduce an end-of-tape symbol that is written in the first cell after the last non-blank symbol.)

5. **Subroutines**. We can use subroutines in TM in an analogous way as they are used in normal programming languages. A subroutine uses its own set of states, including its own "initial state" q_0' and a return state q_r. To call a subroutine, the calling TM simply changes the state to q_0', and makes sure the read-write head is positioned on the leftmost symbol of the "parameter list" to the subroutine.

Constructing TM to perform specific tasks can be quite complicated. Even to recognize some simple languages may require many states and complicated constructions. However, TM are powerful enough to be able to simulate any computer program. The claim that Turing machines can compute everything that is computable using any model of computation is known as **Church-Turing thesis**. Since the thesis talks about *any* model of computation, it can never be proved. But so far TM have been able to simulate all other models of computation that have been proposed.

As an example, let us see how a Turing machine would simulate a register machine, a realistic model of a conventional computer. The tape contains all data the computer has in its memory. The data can be organized for example in such a way that word v_i in memory location i is stored on the tape as the word

$$\#0^i * v_i\#$$

where # and $*$ are special marker symbols. The contents of the registers of the CPU are stored on their own tracks on the tape.

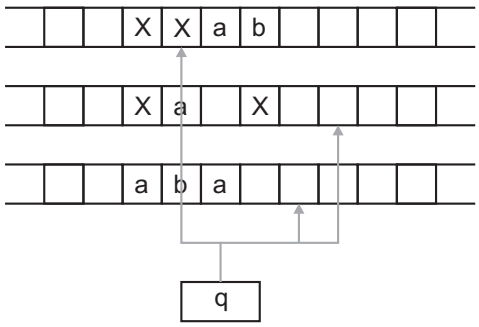
To execute the next instruction, the TM finds the memory location addressed by the specific Program Counter register. In order to do that the TM goes through all memory locations one by one and – using the tick marks – counts if the address i is the same as the content of the Program Counter register. When it finds the correct memory location i, it reads the instruction v_i and memorizes it in the finite control. There are only finitely many different instructions.

To each instruction corresponds its own subroutine. To simulate the instruction, the TM can use the same tick marking to find any required memory locations, and then execute the particular task. The task may be adding the content of a register to another register, for example. Adding two numbers can be easily implemented (especially if we decide to represent all number in the unary format so that number n is represented as the word a^n). Loading a word from the memory to a register is simple as well. To write a word to the memory may require shifting all cells on the right hand side of the memory location, but we know how to do that.

4.3 Modifications of Turing machines

In this section two modifications to our TM model are briefly described. The variations are equivalent: They recognize exactly the same family of r.e. languages as the basic model.

1. **Multiple tape TM**. We can allow the TM to have more than one tape. Each tape has its own independent R/W head. This is different from the one tape TM with multiple tracks since the R/W heads of different tapes can now be at different positions. (For the sake of clarity, in the following illustrations we represent the blank tape symbol by leaving the cell empty):



Depending on the state of the finite control and the current tape symbols on all tapes the machine can

- change the state,
- overwrite the currently scanned symbols on all tapes, and

- move each R/W head to left or right independently of each other.

Formally, the transition function δ is now a (partial) function from

$$(Q \setminus \{f\}) \times \Gamma^n$$

to

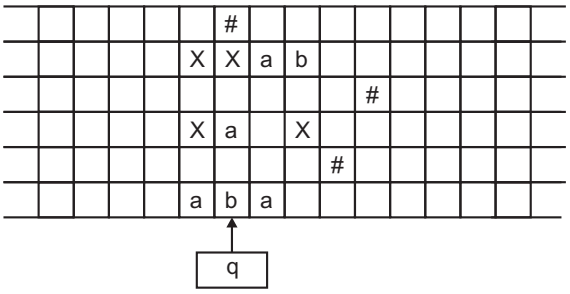
$$Q \times \Gamma^n \times \{L, R\}^n$$

where n is the number of tapes. The transition

$$\delta(q, X_1, X_2, \ldots, X_n) = (p, Y_1, Y_2, \ldots, Y_n, d_1, d_2, \ldots, d_n)$$

(where $d_1, d_2, \ldots, d_n \in \{L, R\}$) means that the machine, in state q, reading symbols X_1, $X_2, \ldots X_n$ on the n tapes, changes its state to p, writes symbols Y_1,$Y_2, \ldots Y_n$ on the tapes, and moves the first, second, third, etc. R/W head to the directions indicated by d_1, $d_2, \ldots d_n$, respectively. Initially, the input is written on tape number one, and all other tapes are blank. A word is accepted iff the machine eventually enters the final state f.

Let us see how a one tape TM can simulate an n-tape TM M. The single tape will have $2n$ tracks — two tracks for every tape of M: One of the tracks contains the data of the corresponding tape in M; The other one contains a single symbol # indicating the position of the R/W head on that tape. The single R/W-head of the one-tape machine is located on the leftmost indicator #. For example, the 3-tape configuration illustrated above would be represented by the following ID with six tracks:

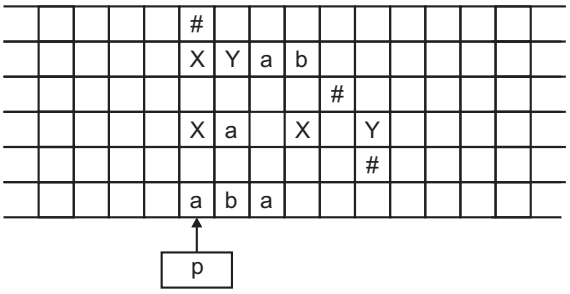


To simulate one move of the multitape machine M, the one-tape machine scans the tape from left to right, remembering in the finite control the tape symbols indicated by the symbols #. Once all #'s have been encountered, the machine can figure out the new state p and the action taken on each tape. During another sweep over the tape, the machine can execute the instruction by writing the required symbols and moving the #'s on the tape left or right.

If, for example, we have

$$\delta(q, X, B, B) = (p, Y, Y, B, L, L, R),$$

after one simulation round the one tape machine will be in the ID



Note that simulating one step of the multitape machine requires scanning through the input twice, so the one-tape machine will be much slower. But all that matters is that the machines accept exactly the same words. It is clear that multitape TM recognize exactly the family of r.e. languages, and multitape TM that halt with all inputs recognize exactly the family of recursive languages.

2. **Non-deterministic Turing machines**. We can also introduce non-determinism. Now δ is a function from

$$(Q \setminus \{f\}) \times \Gamma$$

to subsets of

$$Q \times \Gamma \times \{L, R\}.$$

When scanning tape symbol X in state q, the machine can execute any instruction from the set $\delta(q, X)$. If $\delta(q, X)$ is empty, the machine halts.

A word w is accepted if and only if there exists a computation that takes the initial ID ι_w into an ID where the state is the final state. Note that there may be other computations that halt in a non-final state, or do not halt at all, but the word is accepted as long as there exists at least one accepting computation.

For example, recognizing the language

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

is easy using non-determinism. One does not need to find the center point of the input word first: one may guess where the center point is, and start matching letters one-by-one assuming this center point. If in the end all letters have been matched we know that the guess was correct and the word is in the language.

In an analogous way, it is easy to construct a non-deterministic TM that recognizes the language

$$\{w\#u \mid w, u \in (a + b)^* \text{ and } w \text{ is a subword of } u \}.$$

The machine first makes a non-deterministic guess of the position in u where the subword w begins. Then it verifies the correctness of the guess by erasing symbols in w and u, one by one. If the process ends successfully and the whole w gets erased, the word is in the language.

Non-determinism does not increase the recognizing power of Turing machines. A non-deterministic TM can be simulated by a deterministic one as follows. Let r be the maximum size of the sets $\delta(q, X)$ for any q and X. In other words, there are always at most r possible choices in the non-deterministic machine.

The deterministic TM will use three tapes (which we know can be converted into a one-tape version.) The first tape contains the input, and that tape never changes.

On the second tape we generate words over the alphabet

$$\{1, 2, \ldots, r\}$$

in some predetermined order. We may, for example, start with shortest words, and move up to longer and longer words. Words of equal length are generated in the lexicographical order. (In other words, we are counting integers 1,2,3,... and representing them in r-ary base.)

We use the word generated on the second tape to make a selection among non-deterministic choices for each move. Let x be the word generated on the second tape. We copy the input from the first tape to the third tape, and simulate the non-deterministic TM on the third tape. At each step, we use the next symbol on the second tape to decide which alternative to use among the non-deterministic choices. In any of the following cases we give up, generate the next word on the second tape, and repeat the process:

- If the next r-ary digit on the second tape is greater than the number of choices in the current ID of the machine we are simulating, or
- if we consumed all letters from the second tape.

If on the other hand the machine halts in the final state, the simulating machine accepts the word.

The machine we constructed is deterministic. It is also equivalent to the original non-deterministic machine M because it tries all possible non-deterministic choices one after the other until (if ever) it finds an accepting computation:

- If a word w is accepted by M, there exists a sequence of choices that leads to an accepting calculation. Sooner or later that sequence of choices gets generated on the second tape. Once that happens, we get an accepting computation on the third tape, and the word w is accepted.
- If on the other hand $w \notin L(M)$, there does not exist a sequence of choices that leads to a final state. Therefore, our deterministic machine never halts: it keeps generating longer and longer words on the second tape, never halting.

We conclude that non-deterministic TM recognize exactly the family of recursively enumerable languages. Note that the simulating deterministic machine M' is much slower than the original non-deterministic machine M. If M accepts word w in n moves, it may take more than r^n moves by M' to accept the same word.

It is a famous open problem whether the deterministic machine needs to be so much slower. Let us denote by P the family of languages that are recognized by some deterministic TM in polynomial time. In other words, L is in P if there exists a Turing machine M and a polynomial p such that M recognizes L, and for every $w \in L$ the accepting calculation for w has at most $p(|w|)$ moves.

Analogously, let us denote by NP the family of languages that are recognized by some non-deterministic TM in polynomial time. That means, for every $w \in L$ there exists an accepting computation for w that uses at most $p(|w|)$ moves. Equivalently, every $w \in L$ has a polynomial size certificate string s (=the content of the second tape in the simulation above) such that a deterministic TM can verify in polynomial time that w is in L, when given w and s as input.

It is a celebrated question whether $P = NP$, i.e., whether there are languages that are recognized non-deterministically in polynomial time, but recognizing them deterministically requires super-polynomial time. It is generally assumed that $P \neq NP$. This problem has enormous practical importance because there are many important computational problems that are known to be in NP.

4.4 Closure properties

We can easily prove the following closure properties. The closures are effective when the languages are represented as Turing machines recognizing them.

Theorem 102 *The following hold:*

1. *The family of recursive languages is effectively closed under complementation.*
2. *The families of recursive and recursively enumerable languages are effectively closed under union and intersection.*
3. *A language $L \subseteq \Sigma^*$ is recursive if and only if both L and its complement $\Sigma^* \setminus L$ are recursively enumerable.*

Later – after we learn how to prove that a language is not recursive or r.e. – we see that the family of r.e. languages is not closed under complementation.

Proof.
1. **Complementation of recursive languages**. Let L be a recursive language, recognized by the Turing machine

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, f)$$

that halts on every input. Let us build a Turing machine

$$M' = (Q \cup \{f'\}, \Sigma, \Gamma, \delta', q_0, B, f')$$

that recognizes the complement of L.

M' is identical to M except that M' has a new final state f', and whenever $\delta(q, X)$ is not defined for $q \neq f$ we add the transition $\delta'(q, X) = (f', X, R)$.

Now, if $w \in L(M)$ then both M and M' halt in state f. That state is not the final state of M', so M' does not accept w, and $w \notin L(M')$. Conversely: If $w \notin L(M)$ then M halts in a non-final state q. Therefore M' continues one more move and enters its final state f'. So $w \in L(M')$.

2. **Union and intersection of recursive and r.e. languages** Let L_1 and L_2 be languages recognized by Turing machines M_1 and M_2, respectively. Let us describe a new TM $M_\cap$ for the intersection $L_1 \cap L_2$. The machine $M_\cap$ simply executes M_1 and M_2 one after the other on the same input w: It first simulates M_1 on w. If M_1 halts in the final state, $M_\cap$ clears the tape, copies the input word w on the tape and starts simulating M_2. If also M_2 accepts w then $M_\cap$ accepts. Clearly, $M_\cap$ recognizes $L_1 \cap L_2$, and if M_1 and M_2 halt on all inputs then also $M_\cap$ halts on all inputs.

Consider then the union $L_1 \cup L_2$. Note that the approach we used for the intersection does not work here: simulating first M_1 and then M_2 may not halt on input w even if M_2 would accept w. This happens if M_1 does not halt.

To determine if M_1 or M_2 accepts w we execute both M_1 and M_2 simultaneously, using a two-tape Turing machine $M_\cup$. Machine $M_\cup$ simulates M_1 on the first tape and M_2 on the second tape. If either one enters the final state, the input is accepted.

3. If L is recursive then its complement is recursive by the case 1 of the theorem. Hence both L and its complement are r.e.

Conversely, assume that both L and $\Sigma^* \setminus L$ are r.e. languages. Let M_1 and M_2 be Turing machines that recognize L and $\Sigma^* \setminus L$, respectively. We want to construct a TM M that recognizes L and halts on every input. This is done as in the proof of case 2 above: Machine M uses two tapes and simulates M_1 and M_2 simultaneously on the input word w. Machine M halts when either M_1 or M_2 halts in its final state. It is clear that M eventually halts because every word is accepted by M_1 or M_2. The input is accepted if and only if it was M_1 that halted in the final state. □

4.5 Decision problems and Turing machines

Recall that for each decision problem we have the corresponding language that contains the encodings of positive instances, and that we use the notation $\langle I \rangle$ for the encoding of instance I. Without loss of generality we can assume that encodings are words over the binary alphabet $\{a, b\}$. This is because letters of any larger alphabet can be coded into binary words, and such coding does not change the (semi)decidability of the problem. For this reason, it is enough to consider Turing machines whose input alphabet is $\Sigma = \{a, b\}$.

In this section, we consider decision problems that concern Turing machines, so we need to choose how we encode Turing machines as words. The choice is quite arbitrary; let us fix the following encoding scheme. By renaming the states and tape letters we may assume that the state set is

$$Q = \{q_1, q_2, \ldots, q_n\}$$

and the tape alphabet is

$$\Gamma = \{X_1, X_2, \ldots, X_m\}.$$

The input alphabet $\{a, b\}$ and the blank symbol B must be part of the tape alphabet, so we assume

$$X_1 = a, X_2 = b, X_3 = B.$$

We may also assume that q_1 is the initial state and q_n is the final state of the machine. So we restrict our instances to Turing machines $M = (Q, \Sigma, \Gamma, \delta, q_1, B, q_n)$ where

$$\begin{array}{rcl} Q & = & \{q_1, q_2, \ldots, q_n\}, \\ \Sigma & = & \{a, b\}, \\ \Gamma & = & \{X_1, X_2, \ldots, X_m\} \text{ with} \\ & & \quad X_1 = a, \\ & & \quad X_2 = b, \\ & & \quad X_3 = B. \end{array}$$

It is clear that such Turing machines recognize all recursively enumerable languages over the alphabet $\{a, b\}$, and that halting machines of this kind recognize all recursive languages over $\{a, b\}$.

We also use the notation

$$\begin{array}{rcl} D_1 & = & L, \\ D_2 & = & R \end{array}$$

for the direction of movement.

An arbitrary transition

$$\delta(q_i, X_j) = (q_k, X_l, D_s)$$

by the machine M is encoded as the word

$$a^i\ b\ a^j\ b\ a^k\ b\ a^l\ b\ a^s.$$

Let us encode the machine M as the word

$$\langle M\rangle = bbb\ a^n\ bb\ a^m\ bb\ \text{code}_1\ bb\ \text{code}_2\ bb \ldots bb\ \text{code}_r\ bbb$$

where code_1, code_2, ...,code_r are encodings of all the defined transitions $\delta(q_i, X_j)$, listed in the lexicographic order of i, j.

Example 103. Consider the Turing machine

$$M = (\{q_1, q_2, q_3\}, \{a, b\}, \{a, b, B, X_4\}, \delta, q_1, B, q_3)$$

with transitions

$$\begin{array}{lcl} \delta(q_1, a) & = & (q_1, b, R), \\ \delta(q_1, B) & = & (q_2, B, L), \\ \delta(q_2, b) & = & (q_1, X_4, R), \\ \delta(q_2, B) & = & (q_3, a, R), \\ \delta(q_2, X_4) & = & (q_1, B, L). \end{array}$$

Then

$$\begin{array}{rcl} \langle M\rangle & = & bbbaaabbaaaabb\ abababaabaa\ bb\ abaaabaabaaaba\ bb\ aabaababaaaabaa \\ & & bb\ aabaaabaaababaa\ bb\ aabaaaababaaaba\ bbb. \end{array}$$

□

It is easy to see that the set of valid encodings of Turing machines is recursive: there is an algorithm that checks whether a given word w is an encoding of some Turing machine.

Lemma 104 *The language*

$$L_{enc} = \{\ w \in \{a, b\}^* \mid\ w = \langle M\rangle \text{ for some Turing machine } M\ \ \}$$

of valid encodings of Turing machines is recursive.

Proof. We can construct a TM that verifies the following facts about the input word w:

- Word w has the correct form

  $$bbb\ a^n\ bb\ a^m\ bb\ \text{code}_1\ bb\ \text{code}_2\ bb \ldots bb\ \text{code}_r\ bbb,$$

  for some $n \geq 2$ and $m \geq 3$, where each code_t is a word from $a^+ba^+ba^+ba^+ba^+$. Such words form even a regular language.

- Each $\text{code}_t = a^i\ b\ a^j\ b\ a^k\ b\ a^l\ b\ a^s$ satisfies the conditions

  $$\begin{array}{lcccl} 1 & \leq & i & \leq & n-1, \\ 1 & \leq & k & \leq & n, \\ 1 & \leq & j, l & \leq & m, \\ 1 & \leq & s & \leq & 2. \end{array}$$

  This means each code_t is a valid encoding of some transition $\delta(q_i, X_j) = (q_k, X_l, D_s)$.

- For any consecutive $\text{code}_t = a^i\ b\ a^j\ b\ a^k\ b\ a^l\ b\ a^s$ and $\text{code}_{t+1} = a^{i'}\ b\ a^{j'}\ b\ a^{k'}\ b\ a^{l'}\ b\ a^{s'}$ we have either $i < i'$, or both $i = i'$ and $j < j'$. This guarantees the proper lexicographic ordering of the transitions. This also guarantees that the machine is deterministic.

Input word w represents a Turing machine iff all three conditions are satisfied. □

Now we are ready to prove that a specific language over the alphabet $\{a, b\}$ is not recursively enumerable. The language in question is

$$L_d = \{\langle M\rangle \mid \ M \text{ does not accept word } \langle M\rangle\ \},$$

the encodings of those Turing machines that do not accept their own encoding.

Theorem 105 *The language L_d is not recursively enumerable.*

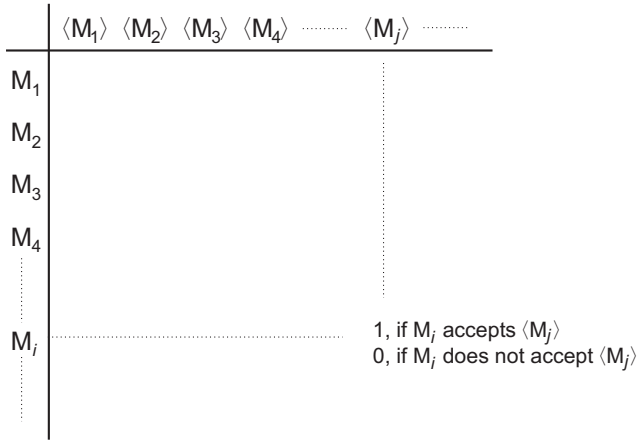
Proof. Suppose the contrary: Turing machine M_d recognizes L_d. Consider the input word $\langle M_d\rangle$ to machine M_d.

- If M_d does not accept $\langle M_d\rangle$ then, by the definition of language L_d, word $\langle M_d\rangle$ is in L_d. But M_d recognizes L_d, so M_d accepts $\langle M_d\rangle$, a contradiction.
- If M_d accepts $\langle M_d\rangle$ then, by the definition of language L_d, word $\langle M_d\rangle$ is not in L_d. But M_d recognizes L_d, so M_d does not accept $\langle M_d\rangle$, a contradiction.

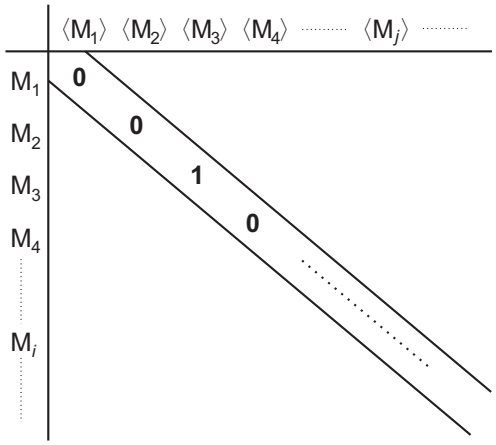
In both cases we reach a contradiction. □

The proof above (due to A.Turing in 1936) is, in fact, the powerful diagonal (or self-reference) argument used in many contexts in mathematics. A similar argumentation was used by G.Cantor in 1891 to show that the set of reals is not countable, and by K.Gödel in 1931 in his first incompleteness theorem. In fact, such self-reference argument was already present in the liar's paradox by Epimenides the Cretan ("*all Cretans are liars*").

The diagonal aspect of Turing's proof can be visualized as follows: Consider an arbitrary enumeration $M_1, M_2, \ldots$ of Turing machines, and an an infinite 0/1-matrix whose rows are indexed by Turing machines M_i and whose columns are indexed by their encodings $\langle M_i\rangle$. The entry $(M_i, \langle M_j\rangle)$ of the table is 1 iff M_i accepts word $\langle M_j\rangle$, so that the row indexed by M_i is the characteristic sequence identifying which words $\langle M_j\rangle$ are accepted by M_i:



The diagonal entries of the table are indexed by $(M_i, \langle M_i \rangle)$. Consider the sequence of bits along the diagonal, and complement this sequence by swapping each bit.



It is clear that the complemented diagonal cannot be identical to any row of the table. So a language whose characteristic sequence is the complemented diagonal is not recognized by any Turing machine. Our L_d is this language.

The theorem means that there is no semi-algorithm that would be able to detect whether a given Turing machine M does not accept the input word $\langle M \rangle$. This decision problem seems rather artificial, but it serves as a "seed" from which we can conclude many other – more natural – problems to be undecidable. The method we use to obtain undecidability results is **reduction**. The following result is the first example of this method. We show that there is no algorithm to determine if a given Turing machine accepts a given input word. More precisely, we show that there is no semi-algorithm to detect the negative instances:

Corollary 106 *The language*

$$L_u = \{ \langle M \rangle \# w \mid M \text{ accepts input word } w \}$$

is not recursive. More specifically, its complement is not recursively enumerable.

Proof. Suppose the contrary: There is a Turing machine $M_{\bar{u}}$ that recognizes the complement of L_u. Using $M_{\bar{u}}$ as a "subroutine" we can then build a Turing machine M_d that recognizes L_d, contradicting Theorem 105. Machine M_d works as follows: On input w it

1. Checks whether w is a valid encoding of some Turing machine. This can be done effectively, as by Lemma 104 the language L_{enc} is recursive. If w is not a valid encoding, machine M_d halts in a non-final state.

2. If $w = \langle M \rangle$ is a valid encoding then M_d writes $w\#w = \langle M \rangle \# \langle M \rangle$ on the tape and starts $M_{\bar{u}}$ on this input. The final state is entered if and only if $M_{\bar{u}}$ accepts this word.

Such Turing machine M_d clearly exists if $M_{\bar{u}}$ exists. Now, word w is accepted by M_d if and only if w is a valid encoding of a Turing machine that does not accept its own encoding. In other words, M_d recognizes L_d, a contradiction. Hence $M_{\bar{u}}$ cannot exist. □

In the reduction above, we described a TM M_d that recognizes a known non-r.e. language L_d, using a hypothetical TM $M_{\bar{u}}$ that recognizes the complement of L_u. We did not provide a detailed transition function of M_d, but a precise enough construction of it so that it is clear that it exists if $M_{\bar{u}}$ exists.

The same reduction can be expressed at a "higher level" by relying on the fact that any semi-algorithm (in our informal, intuitive sense) can be implemented on a Turing machine: Suppose there is a semi-algorithm for recognizing the 0 entries in the infinite table discussed above (i.e., a semi-algorithm for testing whether a given Turing machine does not accept a given word). This semi-algorithm then also identifies entries 1 on the inverted diagonal, contradicting Theorem 105.

The following undecidability result is another example of the reduction technique. This reduction is more complicated as it requires an effective modification of a given Turing machine. The reduction is described at the "high level" using an informal description of an algorithm rather than constructing the corresponding Turing machine. The result is that there is no algorithm to tell if a given Turing machine eventually halts when it is started on the blank tape:

Corollary 107 *The language*

$$L_{halt} = \{ \langle M \rangle \mid \ M \text{ halts when started on the blank tape} \}$$

is not recursive.

Proof. Suppose the contrary: There is an algorithm A to determine if a given Turing machine halts when started on the blank tape. Then the following algorithm $A_{\bar{u}}$ determines whether a given Turing machine M accepts a given word w, contradicting Corollary 106. On input M and w:

1. $A_{\bar{u}}$ builds a new Turing machine M' that halts from the blank initial tape if and only if M accepts w. Such M' can be effectively constructed for any given M and w: When started, M' writes w on its tape. (This may be done using $|w|$ states.) Then M' moves back to the first letter of w and enters the initial state of M. From there on, transitions of machine M are used. For each halting but non-final state of M the machine M' goes into a new looping state that makes the machine move to the right indefinitely without ever halting. Clearly, on the initially empty tape this M' halts if and only if M accepts input w.

2. $A_{\bar{u}}$ does not execute M'. Instead, it just gives M' as an input to the hypothetical algorithm A, and returns "yes" if A returns "yes", and returns "no" if A returns "no".

Clearly, $A_{\bar{u}}$ returns "yes" on input M and w if and only if M accepts w. But such algorithm cannot exist by Corollary 106. Hence, the algorithm A cannot exist either. □

With a more careful analysis of the reduction above we could conclude that the complement of L_{halt} is not recursively enumerable, i.e., non-halting is not semi-decidable.

So, the reduction method is used as follows: To prove that a decision problem P is undecidable we assume that there would exist an algorithm A that solves P. Then we describe an algorithm A' that solves some known undecidable problem P', using A as a subroutine. Since such algorithm A' cannot exist we conclude that algorithm A cannot exist either. Notice that such undecidability proof involves designing an algorithm! – but the algorithm is for a known undecidable problem P' and it uses a hypothetical subroutine A that solves P.

A reduction to show that problem P is not semi-decidable works analogously: Assume that a semi-algorithm for P exists. Build a semi-algorithm A' (using A as a subroutine) for a problem P' that is known not to be semi-decidable. As such A' cannot exist, semi-algorithm A cannot exist either.

Example 108. Yet another example of a reduction: Let us show that there is no algorithm to determine if a given r.e. language is empty, that is, whether a given Turing machine accepts any words. Suppose the contrary: algorithm A determines if a given Turing machine accepts some word. Then we have the following algorithm A' to determine if a given Turing machine M accepts a given word w, contradicting Corollary 106:

1. First A' builds a new Turing machine M' that erases its input, writes w on the tape, moves to the first letter of w and enters the initial state of M. From there on, the transitions of M are used. Such machine M' can be effectively constructed for any given M and w.
2. Next A' gives the machine M' it constructed as an input to the hypothetical algorithm A, and returns the answer that A provides.

Note that if M accepts w then M' accepts all words, and if M does not accept w then M' does not accept any word. So A returns "yes" on input M' if and only if M accepts w. Algorithm A' described above cannot exist (Corollary 106) so the hypothetical algorithm A does not exist. □

4.6 Universal Turing machines

Recall the language

$$L_u = \{ \langle M \rangle \# w \mid \ M \text{ accepts input word } w \ \}.$$

It was shown in Corollary 106 that this language is not recursive. It turns out, however, that L_u is recursively enumerable.

Theorem 109 *The language L_u is recursively enumerable.*

Proof. Rather than constructing an explicit Turing machine that recognizes L_u we only describe informally a semi-algorithm to determine if a given word is in L_u. We know that this semi-algorithm can be implemented by a Turing machine, proving the claim.

The semi-algorithm first checks that the word is of the correct form: an encoding $\langle M \rangle$ of a Turing machine, followed by the marker symbol # and a word w over the alphabet $\{a, b\}$. This can be effectively checked.

Then the semi-algorithm simulates the Turing machine M on the input w until (if ever) M halts. Such step-by-step simulation can clearly be implemented on a semi-algorithm. The semi-algorithm then returns answer "yes" if M halted in its final state. □

Language L_u provides now the following two corollaries, based on Corollary 106 and Theorem 109:

Corollary 110 *There are recursively enumerable languages that are not recursive.* □

Corollary 111 *The family of recursively enumerable languages is not closed under complementation.* □

By Theorem 109 we know that some Turing machine M_u recognizes L_u. Such machine M_u is called a **universal Turing machine** since it can simulate any given Turing machine on any given input, when given the description of the machine to be simulated. So M_u is a programmable computer: rather than building a new TM for each new language, one can use the same TM M_u and only change the "program" $\langle M \rangle$ that describes which Turing machine should be simulated. This is a very important concept — just imagine if we had to build a new computer for each algorithm we want to execute!

4.7 Rice's theorem

We have seen that many questions concerning Turing machines are undecidable. Some questions are clearly decidable (e.g., "Does a given Turing machine have 5 states ?"). But it turns out that any non-trivial question that only concerns the language that a TM recognizes, rather than the machine itself, is undecidable. By a non-trivial question we mean any question that is not always true or always false.

More precisely, let $\mathcal{P}$ be any family of languages. We call $\mathcal{P}$ a **non-trivial property** if there exist Turing machines M_1 and M_2 such that $L(M_1) \in \mathcal{P}$ and $L(M_2) \notin \mathcal{P}$. For any such fixed $\mathcal{P}$ it is undecidable for a given Turing machine M whether $L(M) \in \mathcal{P}$. This result is known as the **Rice's theorem**.

Theorem 112 *Let $\mathcal{P}$ be a non-trivial property. There is no algorithm to determine if a given Turing machine M has $L(M) \in \mathcal{P}$.*

Proof. We use a reduction from L_u. The reduction is very similar to the one in Example 108.

Without loss of generality, assume that $\emptyset \notin \mathcal{P}$. (If this does not hold, we simply consider the complement of $\mathcal{P}$ instead of $\mathcal{P}$.) Because $\mathcal{P}$ is non-trivial, there exists a Turing machine M_P such that $L(M_P) \in \mathcal{P}$.

Suppose there is an algorithm A to determine if a given Turing machine M has $L(M) \in \mathcal{P}$. Then we have the following algorithm A' to determine if a given Turing machine M accepts a given word w, contradicting Corollary 106:

1. First A' builds a new Turing machine M' that
   - Stores its input u on a separate track for later use.
   - Writes w on the tape and moves to the first letter of w.
   - Enters the initial state of M. From there on, the transitions of M are used, ignoring word u, until M enters its final state f.
   - If M enters state f then word u is written on an otherwise blank tape, and machine M_P is simulated on u. Final state is entered if M_P accepts u.
2. Next A' gives the machine M' it constructed as an input to the hypothetical algorithm A, and returns the answer that A provides.

Note that the machine M' can be effectively constructed for any given M and w, so algorithm A' performing the steps above exists, assuming A exists.

The correctness of algorithm A' follows from the following simple observations:

+ If M accepts w then M' will accept exactly the same words u that M_P accepts. Hence, in this case $L(M') = L(M_P) \in \mathcal{P}$.

- If M does not accept w then M' does not accept any input words u, so in this case $L(M') = \emptyset \notin \mathcal{P}$.

Algorithm A' described above cannot exist (Corollary 106) so the hypothetical algorithm A does not exist. □

Remarks: (1) Rice's theorem covers decision problems covered individually in the previous section. It also shows that, for example, the following questions are undecidable: "Does a given TM accept all input words ?", "Is $L(M)$ regular for given TM M ?", "Does a given Turing machine accept all palindromes ?", and so on.

(2) A more careful analysis of the proof of Rice's theorem above shows that, for a non-trivial property $\mathcal{P}$ such that $\emptyset \in \mathcal{P}$, it is not semi-decidable for a given Turing machine M whether $L(M) \in \mathcal{P}$.

4.8 Turing machines as rewriting systems and grammars

A **semi-Thue system** (or a **word rewriting system**) is a pair $T = (\Delta, R)$ where Δ is a finite alphabet and R is a finite set of productions (or **rewrite rules**) $u \longrightarrow v$ where $u, v \in \Delta^*$. The rule $u \longrightarrow v$ can be applied to a word w if w contains u as a subword. The application means replacing an occurrence of u by v in w. If w' is the new word obtained in this way we write $w \Rightarrow w'$ and say that w derives w' in one step. More precisely,

$$w \Rightarrow w'$$

iff

$$w = xuy \text{ and } w' = xvy \text{ for some } x, y, u, v \in \Delta^*$$

and

$$u \longrightarrow v \in R.$$

Example 113. Let $T = (\{a, b\}, R)$ be a semi-Thue system where R contains three productions

$$\begin{array}{lcl} bb & \longrightarrow & b, \\ aba & \longrightarrow & bab, \\ a & \longrightarrow & aa. \end{array}$$

Then, for example,

$$ababa \Rightarrow babba \Rightarrow baba \Rightarrow bbab \Rightarrow bab.$$

□

The **word problem** of semi-Thue systems T asks for given words x and y whether $x \Rightarrow^* y$ using the rewrite rules of T. As usual, $x \Rightarrow^* y$ means that there is a derivation $x \Rightarrow x_1 \Rightarrow x_2 \Rightarrow \ldots \Rightarrow y$ of arbitrary length. We prove in the following that the word problem of some semi-Thue systems is undecidable. In fact, we even show the undecidability of the so-called **individual word problem** in some T: In this problem, word y is fixed, and we ask whether a given word x derives y.

First we associate to each Turing machine M a semi-Thue system T_M that simulates M. This is not difficult to do since the operations of a Turing machine can be viewed as rewriting its instantaneous descriptions. All rewriting is done locally and can hence be expressed as rewrite rules. The only slight complication is caused by the left and right ends of the ID where the tape may need to be extended of shrank by adding or removing blank tape symbols. Note that a rewrite rule cannot recognize the end of a word so no specific rules can be defined for the left and right end. For this reason we add to the ends of the ID new symbols "[" and "]". A Turing machine ID α will then be represented as the word $[\alpha]$.

The following effective construction provides a semi-Thue system such that for any ID α and β of M there is a derivation step $[\alpha] \Rightarrow [\beta]$ if and only if $\alpha \vdash \beta$ is a valid move by the TM. Note that our simulation of M by T_M is tight in the sense that we make sure to add or erase blanks at both ends of the ID.

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, B, f)$ be an arbitrary Turing machine where $Q \cap \Gamma = \emptyset$. The corresponding semi-Thue system is $T_M = (\Delta, R)$ where

$$\Delta = Q \cup \Gamma \cup \{[,]\},$$

and R contains the following productions:

1. (Left moves) For every $q \in Q$ and $a \in \Gamma$ such that $\delta(q, a) = (p, b, L)$ for some $p \in Q$ and $b \in \Gamma$,

   - if $b \neq B$ we have in R the productions

   $$\begin{array}{rcll} xqa & \longrightarrow & pxb & \text{for all } x \in \Gamma \\ [qa & \longrightarrow & [pBb & \end{array}$$

   - if $b = B$ we have in R the productions

   $$\begin{array}{rcll} xqay & \longrightarrow & pxby & \text{for all } x, y \in \Gamma \\ xqa] & \longrightarrow & px] & \text{for all } x \in \Gamma \\ [qay & \longrightarrow & [pBby & \text{for all } y \in \Gamma \\ [qa] & \longrightarrow & [pB] & \end{array}$$

2. (Right moves) For every $q \in Q$ and $a \in \Gamma$ such that $\delta(q, a) = (p, b, R)$ for some $p \in Q$ and $b \in \Gamma$,

   - if $b \neq B$ we have in R the productions

   $$\begin{array}{rcll} qay & \longrightarrow & bpy & \text{for all } y \in \Gamma \\ qa] & \longrightarrow & bpB] & \end{array}$$

   - if $b = B$ we have in R the productions

   $$\begin{array}{rcll} xqay & \longrightarrow & xbpy & \text{for all } x, y \in \Gamma \\ xqa] & \longrightarrow & xbpB] & \text{for all } x \in \Gamma \\ [qay & \longrightarrow & [py & \text{for all } y \in \Gamma \\ [qa] & \longrightarrow & [pB] & \end{array}$$

In each case we correctly simulate the action of adding or removing blanks when the TM acts at the left or right end of the ID. It is easy to see that if $\alpha \vdash \beta$ in M then $[\alpha] \Rightarrow [\beta]$ in T_M, and this is the only possible derivation step from $[\alpha]$. If α is a halting ID then there are no possible derivation steps from $[\alpha]$. The following Lemma summarizes these facts:

Lemma 114 *Let α be an ID of the Turing machine M and let T_M be the semi-Thue system constructed above. Then, for $w \in \Delta^*$,*

$$[\alpha] \Rightarrow w \qquad \textit{if and only if} \qquad w = [\beta] \textit{ and } \alpha \vdash \beta.$$

□

To prove the undecidability of the word problem in semi-Thue systems we want the simulating Turing machine to erase its tape when the computation is finished. This can be assumed:

Lemma 115 *For every Turing machine M one can effectively construct a Turing machine M' such that $L(M') = L(M)$ and every accepting computation in M' ends in the ID fB.*

Proof. We modify M in the following ways:

- In the beginning of the computation, end-markers are added at the beginning and end of the input word.
- During the computation by M, if the machine encounters an end-marker the end-marker is moved so that the Turing machine always remains between the markers.
- If M accepts the input word, the computation continues as follows: the machine moves to the left end-marker, and starting from there moves to the right, erasing all symbols along the way until meeting the right end-marker. Then the final state f is entered.

It is clear that such TM can be effectively constructed and that it has the required properties. □

Theorem 116 *There exists a semi-Thue system T and a fixed word u such that the individual word problem "Does a given word x derive u in T ?" is undecidable*

Proof. Let M be a Turing machine such that $L(M)$ is not recursive. Such M exists by Corollary 110. By Lemma 115 we may assume that every accepting computation ends in the ID fB. Let T_M be the corresponding semi-Thue system constructed above. By Lemma 114, for any input w

$$[\iota_w] \Rightarrow^* [fB]$$

if and only if $w \in L(M)$. Here ι_w is the initial ID corresponding to input word w. If there were an algorithm to determine if a given word x derives $u = [fB]$ in $T = T_M$, then there would be an algorithm to determine if M accepts a given word w, contradicting the non-recursiveness of $L(M)$. □

Remarks: (1) By adding $[fB] \longrightarrow \varepsilon$ to the production set, we can choose $u = \varepsilon$ in the previous theorem. We obtain a semi-Thue system in which it is undecidable whether a given word derives the empty word.

(2) It is known that there exists a semi-Thue system (Δ, R) with $|\Delta| = 2$ and $|R| = 3$ whose individual word problem is undecidable. The decidability status for semi-Thue systems with two productions is not known.

A **Thue system** is a special kind of semi-Thue system where $u \longrightarrow v$ is a production if and only if $v \longrightarrow u$ is a production. In other words, all rewrite-rules may be applied in both directions. For this reason we write the productions of a Thue system as $u \longleftrightarrow v$, and we denote one step derivations using $\Leftrightarrow$ instead of $\Rightarrow$. It is easy to see that $\Leftrightarrow^*$ is an equivalence relation, and moreover, it is a congruence of the monoid Δ^*:

Lemma 117 *Let $T = (\Delta, R)$ be a Thue system. Then $\Leftrightarrow^*$ is an equivalence relation, and $u_1 \Leftrightarrow^* v_1$, $u_2 \Leftrightarrow^* v_2$ implies that $u_1 u_2 \Leftrightarrow^* v_1 v_2$.* □

The Thue system T is a finite presentation of the quotient monoid $\Delta^* / \Leftrightarrow^*$ whose elements are the equivalence classes. The word problem of T is then the question of whether two given words represent the same element in the quotient monoid.

The following theorem strengthens Theorem 116 for the case of Thue systems:

Theorem 118 *There exists a Thue system T and a fixed word u such that the individual word problem "Does a given word x derive u in T ?" is undecidable*

Proof. We use the following two facts concerning the semi-Thue system T_M associated to TM M:

(i) The rewriting is deterministic: If $w \in \Delta^*$ contains a single occurrence of a letter from Q then there is at most one $z \in \Delta^*$ such that $w \Rightarrow z$.

(ii) In each production $u \longrightarrow v$ both u and v contain a single occurrence of a letter from Q.

Let us interpret T_M as a Thue system, i.e., let us allow the productions to applied in either direction. Let us denote the derivation relation of the Thue system by $\Leftrightarrow$, while we denote $u \Rightarrow v$ and $u \Leftarrow v$ if v is obtained from u by an application of a production in the forward or backward direction, respectively.

As in the proof of Theorem 116, let M be a Turing machine that recognizes a non-recursive language, and assume that every accepting computation ends in the unique ID $[fB]$. Let us prove that $[\iota_w] \Leftrightarrow^* [fB]$ in T_M if and only if $w \in L(M)$.

One direction is clear: If $w \in L(M)$ then $[\iota_w] \Rightarrow^* [fB]$, so also $[\iota_w] \Leftrightarrow^* [fB]$. Consider then the other direction: Suppose that $w \in \Sigma^*$ is such that $[\iota_w] \Leftrightarrow^* [fB]$. Then there exists a sequence of words $w_0, w_1, w_2, \ldots w_n$ such that

$$[\iota_w] = w_0 \Leftrightarrow w_1 \Leftrightarrow w_2 \Leftrightarrow \ldots \Leftrightarrow w_n = [fB].$$

Let n be as small as possible. If all derivation steps use the productions in the forward direction, that is, if $w_{i-1} \Rightarrow w_i$ for all $i = 1, 2, \ldots, n$, then the derivation simulates the computation of the TM from initial ID ι_w, so $w \in L(M)$.

Suppose then that for some i we have $w_{i-1} \Leftarrow w_i$. Let i be the largest such index. Because no production can be applied in the forward direction to $[fB]$ we must have $i < n$. By the maximality of i we then have $w_i \Rightarrow w_{i+1}$. But the property (ii) above guarantees that all w_i contain exactly one occurrence of a letter from Q, and property (i) then implies that the forward rewriting is

deterministic. We conclude that $w_{i-1} = w_{i+1}$, and the derivation can be made shorter by removing the unnecessary loop $w_{i-1} \Leftrightarrow w_i \Leftrightarrow w_{i+1}$. This contradicts the minimality of n. □

Remark: (1) The theorem means that the word problem is undecidable among finitely presented monoids. The result can be strengthened further: It is known that the word problem is undecidable even among finitely presented groups. On the other hand, the word problem is decidable among finitely presented abelian (i.e. commutative) monoids and groups.

(2) By adding $[fB] \longleftrightarrow \varepsilon$ to the production set, we can choose $u = \varepsilon$ in the theorem. In this case, it is undecidable if a given word represents the identity element of the monoid. (A small proof is needed to show that shortest derivations $[\iota_w] \Leftrightarrow^* [fB]$ do not use the new production $[fB] \longleftrightarrow \varepsilon$. This can be done because [and] isolate the rewriting into disjoint regions of the word.)

(3) There exists a Thue system (Δ, R) with $|\Delta| = 2$ and $|R| = 3$ whose word problem is undecidable.

A **grammar** is a quadruple $G = (V, T, P, S)$ where V and T are disjoint finite alphabets of variables (=non-terminals) and terminals, respectively, $S \in V$ is the start symbol, and P is a finite set of productions $u \longrightarrow v$ where u and v are words over $V \cup T$ and u contains at least one variable. The pair $(V \cup T, P)$ acts as a semi-Thue system: we denote $w \Rightarrow w'$ if w' is obtained from w by replacing an occurrence of u by v for some $u \longrightarrow v \in P$. Note that if $w \in T^*$ is terminal then no derivation is possible from w. The language

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

generated by the grammar consists of all the terminal words that can be derived from the start symbol S. Such grammars are called **type-0 grammars**.

Example 119. The grammar $G = (\{S, X, Y, Z\}, \{a, b, c\}, P, S)$ with productions

$$\begin{array}{rcl} S & \longrightarrow & aXbc \mid \varepsilon \\ X & \longrightarrow & \varepsilon \mid aYb \\ Yb & \longrightarrow & bY \\ Yc & \longrightarrow & Zcc \\ bZ & \longrightarrow & Zb \\ aZ & \longrightarrow & aX \end{array}$$

generates the language

$$L(G) = \{a^n b^n c^n \mid n \geq 0\}.$$

Indeed, the start symbol S generates directly ε. Otherwise, word $aXbc$ is derived. From the word $a^n X b^n c^n$, for any $n \geq 1$, one can either derive $a^n b^n c^n$ using $X \longrightarrow \varepsilon$, or the following derivation steps are forced:

$$a^n X b^n c^n \Rightarrow a^{n+1} Y b^{n+1} c^n \Rightarrow^* a^{n+1} b^{n+1} Y c^n \Rightarrow a^{n+1} b^{n+1} Z c^{n+1} \Rightarrow^* a^{n+1} Z b^{n+1} c^{n+1} \Rightarrow a^{n+1} X b^{n+1} c^{n+1}.$$

One sees (using induction on n) that $L(G)$ consists of the words $a^n b^n c^n$ for $n \geq 0$. □

In the following we show that type-0 grammars generate exactly the family of recursively enumerable languages. While Turing machines are accepting devices, grammars are generative. To convert a Turing machine into an equivalent grammar we therefore need to "reverse" the computation so that we start generating from the accepting ID backwards in time, until an initial ID is reached.

Theorem 120 *Turing machines and type-0 grammars are effectively equivalent.*

Proof. It is clear that the language generated by any grammar is recursively enumerable: There is a semi-algorithm that enumerates all derivations in the order of increasing length. If any of them derives the input word then the word is accepted. A Turing machine executing such semi-algorithm can be effectively constructed.

For the converse direction we show how to construct for any given Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, B, f)$ a type-0 grammar $G = (V, \Sigma, P, S)$ such that $L(G) = L(M)$. By Lemma 115 we can assume that every accepting computation in M ends in the ID fB. Let $T_M = (\Delta, R)$ be the semi-Thue system associated to machine M. Recall that $\Delta = Q \cup \Gamma \cup \{[,]\}$ and that in each production $u \longrightarrow v$ in R both u and v contains exactly one occurrence of a letter from Q. Let R^R be the set of reversed rules: for every $u \longrightarrow v$ in R there is $v \longrightarrow u$ in R^R. From Lemma 114 we have that $[fB] \Rightarrow^* [\iota_w]$ in $T_M^R = (\Delta, R^R)$ if and only if $w \in L(M)$. Here $\iota_w = q_0 w$ if $w \neq \varepsilon$ and $\iota_\varepsilon = q_0 B$. Notice also that all words that can be derived from $[fB]$ are of the form $[zqy]$ where $q \in Q$ and $z, y \in \Gamma^*$ with $|y| \geq 1$. This follows from the facts that the productions do not change the boundary symbols "[" and "]", and between the boundary symbols exactly one occurrence of a letter from Q is maintained, followed by at least one other symbol before the right boundary marker.

We add to Δ two new symbols # and S so the variable set of the grammar is

$$V = Q \cup (\Gamma \setminus \Sigma) \cup \{[,], \#, S\}.$$

We add to the production set R^R the following new productions:

$$\begin{array}{rcll} S & \longrightarrow & [fB] & \\ [q_0 & \longrightarrow & \# & \\ \#a & \longrightarrow & a\# & \text{for all } a \in \Sigma \\ \#] & \longrightarrow & \varepsilon & \\ [q_0 B] & \longrightarrow & \varepsilon & \end{array}$$

The first production is used to initialize the sentential form to $[fB]$. The last production erases $[q_0 B]$ if the empty word is accepted. The other productions erase from any $[q_0 w]$ symbols [, q_0 and], leaving w. In any case, $S \Rightarrow [fB]$ and $[\iota_w] \Rightarrow^* w$ using these productions.

Let P be R^R extended by the productions above. Now the grammar $G = (V, \Sigma, P, S)$ is fully defined. Let us prove that $L(G) = L(M)$.

If $w \in L(M)$ then $\iota_w \vdash^* fB$ in M. Hence, G admits the derivation

$$S \Rightarrow [fB] \Rightarrow^* [\iota_w] \Rightarrow^* w,$$

so $w \in L(G)$.

Conversely, let $w \in L(G)$. All derivations begin $S \Rightarrow [fB]$. Using productions in R^R one can only reach sentential forms of types $[zqy]$ where $q \in Q$. In order to derive a terminal word, production $[q_0 B] \longrightarrow \varepsilon$ or $[q_0 \longrightarrow \#$ needs to be used. The first production can only be applied on $[q_0 B]$, which can be reached only if $\varepsilon \in L(M)$, as required. The second one can be applied only on $[zqy] = [q_0 y]$, deriving $\#y]$ in one step. This derives only the terminal string y so $y = w$. In any case, if $S \Rightarrow^* w$ then $[fB] \Rightarrow^* [\iota_w]$ using productions of R^R only. We conclude that $\iota_w \vdash^* fB$ in M, so $w \in L(M)$. □

4.9 Other undecidable problems

In this section we see undecidable problems encountered in different contexts. Problems considered include the Post correspondence problem, problems associated to context-free grammars, a problem concerning products of integer matrices, and a variant of the tiling problem.

4.9.1 Post correspondence problem

Our next undecidable problem is the **Post correspondence problem** (PCP). An instance to PCP consists of two lists of words over some alphabet Σ:

$$\begin{array}{lcllll} L_1 & : & w_1, & w_2, & \ldots & w_k \\ L_2 & : & x_1, & x_2, & \ldots & x_k \end{array}$$

Both lists contain equally many words. We say that each pair (w_i, x_i) forms a pair of corresponding words. A solution to the instance is any non-empty string $i_1 i_2 \ldots i_m$ of indices from $\{1, 2, \ldots, k\}$ such that

$$w_{i_1} w_{i_2} \ldots w_{i_m} = x_{i_1} x_{i_2} \ldots x_{i_m}.$$

In other words, we concatenate corresponding words w_i and x_i to form two words. We have a solution if the concatenated w_i's form the same word as the corresponding concatenated x_i's. The PCP asks whether a given instance has a solution or not. It turns out that PCP is undecidable.

PCP can also be expressed compactly in terms of homomorphisms: The problem asks for two given homomorphisms $h_1, h_2 : \Delta^* \longrightarrow \Sigma^*$ whether there exists a non-empty word u such that $h_1(u) = h_2(u)$. The connection to the formulation using lists is seen if we choose $\Delta = \{1, 2, \ldots, k\}$ and $h_1(i) = w_i$ and $h_2(i) = x_i$ for all $i \in \Delta$. Then $h_1(u)$ and $h_2(u)$ are the concatenated words of w_i's and x_i's given by the index sequence u.

Example 121. Consider the following two lists:

$$\begin{array}{lcllll} L_1 & : & a^2, & b^2, & ab^2 \\ L_2 & : & a^2b, & ba, & b \end{array}$$

This instance has solution 1213 because

$$\begin{array}{lcl} w_1 w_2 w_1 w_3 & = & aa\ bb\ aa\ abb \\ x_1 x_2 x_1 x_3 & = & aab\ ba\ aab\ b \end{array}$$

are identical. In terms of homomorphisms, the instance is expressed as $h_1, h_2 : \{1, 2, 3\}^* \longrightarrow \{a, b\}^*$ where $h_1(1) = aa$, $h_1(2) = bb$, $h_1(3) = abb$ and $h_2(1) = aab$, $h_2(2) = ba$, $h_2(3) = b$. The solution 1213 simply means that $h_1(1213) = h_2(1213)$. □

Example 122. The PCP instance

$$\begin{array}{lcllll} L_1 & : & a^2b, & a \\ L_2 & : & a^2, & ba^2 \end{array}$$

does not have a solution: If it would have a solution, the solution would need to start with index 1. Since $w_1 = a^2b$ and $x_1 = a^2$, the second list has to catch up the missing b: The second index

has to be 2. Because $w_1w_2 = a^2ba$ and $x_1x_2 = a^2ba^2$ the first list has to catch up. The next index cannot be 1 because $w_1w_2w_1 = a^2baa^2b$ and $x_1x_2x_1 = a^2ba^2a^2$ differ in the 7'th letter. So the third index is 2, yielding $w_1w_2w_2 = a^2baa$ and $x_1x_2x_2 = a^2ba^2ba^2$. Now the first list has to catch up ba^2 which is not possible since neither w_1 nor w_2 starts with letter b. □

Example 123. Consider the following instance of PCP:

$$\begin{array}{lcll} L_1 & : & a, & ba \\ L_2 & : & ab, & ab \end{array}$$

Now a solution has to start with index 1. Since $w_1 = a$, $x_1 = ab$ we need a word from the first list that starts with a b. So the second index has to be a 2. Since $w_1w_2 = aba$, $x_1x_2 = abab$ the first list has to catch up a missing b again, so the third index has to be a 2 as well. This never ends: The first list will always be behind by one b. Therefore this PCP instance does not have a solution. □

Theorem 124 *The Post correspondence problem is undecidable.*

Proof. We reduce the word problem of semi-Thue systems to PCP. Assume there exists an algorithm A that solves PCP. Let us see how A can be used to solve the word problem of semi-Thue systems.

Let $T = (\Sigma, R)$ be a given semi-Thue system, and let x and y be given words over alphabet Σ. We may assume that in every rewrite rule $u \longrightarrow v$ both u and v are non-empty words, and also that x and y are both non-empty. We can make this assumption because the semi-Thue systems constructed in the proof of Theorem 116 has this property, so the word problem is undecidable in such restricted cases.

First we construct an equivalent PCP instance. Let Σ' be a new alphabet containing letters a' for all $a \in \Sigma$. For every word w over Σ we will denote by w' the corresponding word obtained by priming all letters. (More precisely, $w' = h(w)$ where h is the homomorphism that maps $a \mapsto a'$, for all $a \in \Sigma$.) Let $\#$ be a new marker symbol.

The following corresponding pairs of words are placed into the lists L_1 and L_2:

L_1	L_2	
$\#x$	$\#$	
$\#$	$y\#$	
a	a'	for every $a \in \Sigma$
a'	a	for every $a \in \Sigma$
v'	u	for every $u \longrightarrow v \in R$

The only pair in which one of the words is the prefix of the corresponding word in the other list is $(\#x, \#)$. Therefore a solution to the PCP instance must start with this pair. Correspondingly, the only pair in which one word is the suffix of the other one is $(\#, y\#)$ so any solution must end in this pair.

Consider the situation after the first pair:

$$\begin{array}{l} \# \; x \\ \# \end{array}$$

The second word has to catch-up x. The only pairs that can match letters of x are (a', a) for $a \in \Sigma$ and (v', u) for $u \longrightarrow v \in R$. Such pairs produce a new primed word after x in the first catenated word:

$$\begin{array}{l} \# \; x \; x_1' \\ \# \; x \end{array}$$

Note that x_1 can be obtained from x in the semi-Thue system in some number of derivation steps, using non-overlapping applications of productions. (In other words, words x and x_1 can be expressed as concatenations $x = w_0u_1w_1u_2 \dots u_iw_i$ and $x_1 = w_0v_1w_1v_2 \dots v_iw_i$ of words such that all (u_i, v_i) are pairs of corresponding words in the PCP instance. Notice that the priming of the letters in x_1' prevent the u_i from extending over the boundary of x and x_1'.) So we have $x \Rightarrow^* x_1$ in T.

Next the second list has to catch-up the primed word x_1'. The only pairs that contain primed letters in the second components are (a, a') for $a \in \Sigma$. Such pairs create a new unprimed copy of x_1 in the first word:

$$\begin{array}{l} \# \; x \; x_1' \; x_1 \\ \# \; x \; x_1' \end{array}$$

Now the process is repeated on x_1 instead of x, and so on. We are forced to create matching words of type

$$\begin{array}{l} \# \; x \; x_1' \; x_1 \; x_2' \; x_2 \; x_3' \; x_3 \dots \; x_n' \; x_n \\ \# \; x \; x_1' \; x_1 \; x_2' \; x_2 \; x_3' \; x_3 \dots \; x_n' \end{array}$$

such that the semi-Thue system has a derivation

$$x \Rightarrow^* x_1 \Rightarrow^* x_2 \Rightarrow^* x_3 \dots \Rightarrow^* x_n.$$

The second list can catch-up the first list only if for some n we have $x_n = y$. Then the pair $(\#, y\#)$ can be used to close the words:

$$\begin{array}{l} \# \; x \; x_1' \; x_1 \; x_2' \; x_2 \; x_3' \; x_3 \dots \; x_n' \; x_n \; \# \\ \# \; x \; x_1' \; x_1 \; x_2' \; x_2 \; x_3' \; x_3 \dots \; x_n' \; y \; \# \end{array}$$

From the discussion above it is obvious that the PCP instance we constructed has a solution if and only if $x \Rightarrow^* y$ in the given semi-Thue system T. □

Example 125. Consider the semi-Thue system of Example 113 with productions

$$\begin{array}{lcl} bb & \longrightarrow & b, \\ aba & \longrightarrow & bab, \\ a & \longrightarrow & aa, \end{array}$$

and let $x = ababa$ and $y = bab$. The corresponding PCP instance contains 9 pairs of words:

$$\begin{array}{lcccccccccc} L_1 & : & \#ababa & \# & a & b & a' & b' & b' & b'a'b' & a'a' \\ L_2 & : & \# & bab\# & a' & b' & a & b & bb & aba & a \end{array}$$

The word problem instance has a positive solution, so the PCP has one too:

$$\begin{array}{l} \#ababa \; b'a'b' \; b' \; a' \; b \; a \; b \; b \; a \; b' \; a' \; b' \; a' \; b \; a \; b \; a \; b' \; b'a'b' \; b \; b \; a \; b \; b' \; a' \; b' \; b \; a \; b \quad \# \\ \quad \# \quad aba \; b \; a \; b' \; a' \; b' \; b' \; a' \; b \; a \; bb \; a \; b' \; a' \; b' \; a' \; b \quad aba \; b' \; b' \; a' \; b' \; bb \; a \; b \; b' \; a' \; b' \; bab\# \end{array}$$

□

Remarks: (1) The PCP is undecidable even if the words w_i and x_i are over the binary alphabet $\{a, b\}$. This is because any alphabet Σ can be encoded in the binary alphabet using an injective homomorphism $h : \Sigma^* \longrightarrow \{a, b\}^*$. (For example, we can use distinct binary words of length $\lceil \log_2 |\Sigma| \rceil$ to encode the letters of Σ.) Because of the injectivity of h, the instance $h_1, h_2 : \Delta^* \longrightarrow \Sigma^*$ is equivalent to the binary instance $h \circ h_1, h \circ h_2 : \Delta^* \longrightarrow \{a, b\}^*$.

(2) Our construction converts a semi-Thue system (Σ, R) into a PCP instance containing $2|\Sigma| + |R| + 2$ pairs of words. Over the binary alphabet $|\Sigma| = 2$ this means $|R| + 6$ pairs. There is a smarter reduction that provides only $|R|+4$ pairs of words. Since there is a semi-Thue system with $|R| = 3$ rules whose individual word problem is undecidable, we obtain that PCP is undecidable among lists with 7 pairs of words. With a different approach this number can be reduced to 5. On the other hand, PCP is known to be decidable among instances with 2 pairs of words. It is presently not known whether PCP is decidable or undecidable for 3 or 4 pairs of words.

4.9.2 Problems concerning context-free grammars

Next we learn how PCP can be reduced to some questions concerning context-free grammars and languages, proving that those questions are undecidable. Let $w_1, w_2, \ldots, w_k$ be a finite list of words over an alphabet Σ, and assume that $\{1, 2, \ldots k, \#, \$\} \cap \Sigma = \emptyset$. We associate to this list the context-free language

$$L(w_1, \ldots, w_k) = \{\ i_1 i_2 \ldots i_n \# w_{i_n}^R w_{i_{n-1}}^R \ldots w_{i_1}^R \$ \mid n \geq 1\}$$

over the alphabet $\Delta = \Sigma \cup \{1, 2, \ldots, k\} \cup \{\#, \$\}$. Using the homomorphism notation with $h(i) = w_i$ for all $i \in \{1, 2, \ldots, k\}$, the language consists of words $u \# h(u)^R \$$ for all non-empty $u \in \{1, 2, \ldots k\}^+$.

The language is indeed context-free because it is derived by the CFG $G = (\{S, A\}, \Delta, P, S)$ with productions

$$S \longrightarrow A\$$$

to create the end marker \$,

$$A \longrightarrow 1Aw_1^R \mid 2Aw_2^R \mid \ldots \mid kAw_k^R$$

to create matching index/word pairs, and

$$A \longrightarrow 1\#w_1^R \mid 2\#w_2^R \mid \ldots \mid k\#w_k^R$$

to terminate. In fact, we have the following:

Lemma 126 *Language $L(w_1, \ldots, w_k)$ is (effectively) recognized by a deterministic PDA without ε-transitions, using the final state acceptance mode.*

Proof. We construct such a PDA

$$A = (Q, \Delta, \Gamma, \delta, q_0, Z_0, \{q_F\}),$$

where $Q = \{q_0, q_1, q_2, q_F\}$ and $\Gamma = \Sigma \cup \{Z_0\}$. The idea is that A reads in state q_1 indices $i \in \{1, 2, \ldots, k\}$ and pushes, for each index i, the corresponding word w_i^R into the stack. Once the

marker # is encountered the machine changes into state q_2 and starts matching input letters with the symbols in the stack. The word is accepted if and only if the stack contains Z_0 when the last input letter is \$ is being read. The precise transition function is given by

$$\begin{array}{lcll}
\delta(q_0, i, Z_0) & = & \{(q_1, w_i^R Z_0)\} & \text{for all } i \in \{1, 2, \ldots, k\} \\
\delta(q_1, i, Z) & = & \{(q_1, w_i^R Z)\} & \text{for all } Z \in \Gamma \text{ and } i \in \{1, 2, \ldots, k\} \\
\delta(q_1, \#, Z) & = & \{(q_2, Z)\} & \text{for all } Z \in \Gamma \\
\delta(q_2, a, a) & = & \{(q_2, \varepsilon)\} & \text{for all } a \in \Sigma \\
\delta(q_2, \$, Z_0) & = & \{(q_F, Z_0)\} &
\end{array}$$

□

The reason for proving the lemma above is the following corollary

Corollary 127 *The complement of language $L(w_1, \ldots, w_k)$ is (effectively) context-free.*

Proof. Let us add into the PDA A of Lemma 126 a new state f and transitions $\delta(q, a, Z) = (f, Z)$ for all $q \in Q \cup \{f\}$, $a \in \Delta$ and $Z \in \Gamma$ such that $\delta(q, a, Z)$ was undefined in A. The new PDA is still deterministic without ε-transitions, its transition function is complete, and the symbol Z_0 is never removed from the stack so the stack never becomes empty. These properties imply that the PDA does not halt until the entire input word has been read, and that for every input word $w \in \Delta^*$ there corresponds a unique state q and stack content $\alpha \in \Gamma^*$ such that $(q_0, w, Z_0) \vdash^* (q, \varepsilon, \alpha)$. Here, $q = q_F$ if and only if $w \in L(w_1, \ldots, w_k)$. If we swap the final states so that the new final states are all states except q_F then the new PDA recognizes the complement of $L(w_1, \ldots, w_k)$. □

In fact, the family of deterministic context-free languages is closed under complementation. However, possible ε-transitions in a deterministic PDA cause technical difficulties in the proof. In our case, complementation is easy because the PDA has no ε-transitions: it is enough simply to swap the accepting and non-accepting states.

The complement is not needed in our first result:

Theorem 128 *It is undecidable for given context-free languages L_1 and L_2 whether $L_1 \cap L_2 = \emptyset$.*

Proof. We reduce the Post correspondence problem. Let

$$\begin{array}{lcllll}
L_1 & : & w_1, & w_2, & \ldots & w_k \\
L_2 & : & x_1, & x_2, & \ldots & x_k
\end{array}$$

be a given instance to PCP. We effectively construct the context-free languages $L(w_1, \ldots w_k)$ and $L(x_1, \ldots x_k)$ corresponding to the two lists, as discussed above. We clearly have

$$L(w_1, \ldots w_k) \cap L(x_1, \ldots x_k) \neq \emptyset$$

if and only if the PCP instance has a solution. Indeed, word $i_1 i_2 \ldots i_n \# w^R \$$ is in $L(w_1, \ldots w_k)$ and $L(x_1, \ldots x_k)$ if and only if $n \geq 1$ and

$$w_{i_1} w_{i_2} \ldots w_{i_n} = w = x_{i_1} x_{i_2} \ldots x_{i_n}.$$

□

The grammar G constructed in the beginning of the section for the language $L(w_1, \ldots, w_n)$ is of very restricted form: it is a linear grammar (righthand-sides of all productions contain at most one variable) and it is unambiguous. Therefore we have the undecidability of the question "Is $L(G_1) \cap L(G_2) = \emptyset$ for given linear, unambiguous context-free grammars G_1 and G_2 ?" From this remark we immediately obtain the next undecidability result:

Theorem 129 *It is undecidable whether a given linear context-free grammar G is unambiguous.*

Proof. Let $G_1 = (V_1, T_1, P_1, S_1)$ and $G_2 = (V_2, T_2, P_2, S_2)$ be two given linear, unambiguous grammars. Rename the variables so that V_1 and V_2 are disjoint and do not contain variable S. Construct a new linear grammar

$$G = (V_1 \cup V_2 \cup \{S\}, T_1 \cup T_2, P, S)$$

where P contains all productions from P_1 and P_2, and the additional "start-up" productions

$$S \longrightarrow S_1 \mid S_2.$$

Then $L(G_1) \cap L(G_2) = \emptyset$ if and only if G is unambiguous. Indeed, every $u \in L(G_1) \cap L(G_2)$ has two left-most derivations in G starting with the steps $S \Rightarrow S_1$ and $S \Rightarrow S_2$, and continuing with derivations according to G_1 and G_2, respectively. Conversely, if $L(G_1) \cap L(G_2) = \emptyset$ then every generated word has only one leftmost derivation because G_1 and G_2 are unambiguous. □

In the next result we use the fact that the complements of $L(w_1, \ldots, w_k)$ and $L(x_1, \ldots, x_k)$ are effectively context-free.

Theorem 130 *It is undecidable if a given context-free language $L \subseteq \Sigma^*$ satisfies $L = \Sigma^*$.*

Proof. We reduce Post correspondence problem. For any given instance

$$\begin{array}{lcllll} L_1 & : & w_1, & w_2, & \ldots & w_k \\ L_2 & : & x_1, & x_2, & \ldots & x_k \end{array}$$

to PCP we effectively construct the complements L_1 and L_2 of $L(w_1, \ldots, w_k)$ and $L(x_1, \ldots, x_k)$, respectively. Then $L_1 \cup L_2 = \Sigma^*$ if and only if $L(w_1, \ldots, w_k) \cap L(x_1, \ldots, x_k) = \emptyset$, that is, if and only if the PCP instance has no solution. We can effectively construct the union $L_1 \cup L_2$, so the result follows from the undecidability of PCP. □

Corollary 131 *Let L_1 and L_2 be given context-free languages and let R be a given regular language. The following questions are undecidable:*

(a) Is $L_1 = L_2$?

(b) Is $L_2 \subseteq L_1$?

(c) Is $L_1 = R$?

(c) Is $R \subseteq L_1$?

The question whether $L_1 \subseteq R$ is, however, decidable.

Proof. Undecidability of (a)-(d) follows from Theorem 130 by fixing $L_2 = R = \Sigma^*$. To decide whether $L_1 \subseteq R$ one can effectively construct the intersection $L_1 \cap \overline{R}$ and test it for emptyness. □

4.9.3 Mortality of matrix products

Next we prove an undecidability result that concerns product of square matrices with integer entries. Let $\{M_1, M_2, \ldots, M_k\}$ be a finite set of $n \times n$ matrices over integers. We say that the matrix set is **mortal** if some product of the matrices from the set is the zero matrix.

Example 132. Consider the following two 2×2 matrices:

$$M_1 = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \qquad M_2 = \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix}$$

Matrix set $\{M_1, M_2\}$ is not mortal: Because $\det(M_1)$ and $\det(M_2)$ are non-zero, the determinant of every product is non-zero. □

Example 133. Consider

$$M_1 = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \qquad M_2 = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}.$$

The product $M_2 M_1 M_2$ is the zero matrix so this pair is mortal. □

The matrix mortality problem asks whether a given set of $n \times n$ integer matrices is mortal. In the following we show that this problem is undecidable, even among 3×3 integer matrices. We start by associating to each word $w = a_1 a_2 \ldots a_m$ over the alphabet $\{1, 2, 3\}$ the integer

$$\sigma(w) = a_m + 4a_{m-1} + \ldots + 4^{m-1} a_1,$$

that is, the number that w represents in base four. It is clear that σ is injective, and for any words u and v we have

$$\sigma(uv) = \sigma(v) + 4^{|v|}\sigma(u).$$

Next we associate to any pair u, v of words over the alphabet $\{1, 2, 3\}$ the following 3×3 integer matrix:

$$M_{u,v} = \begin{pmatrix} 4^{|u|} & 0 & 0 \\ 0 & 4^{|v|} & 0 \\ \sigma(u) & \sigma(v) & 1 \end{pmatrix}.$$

The following lemma shows that the mapping $(u, v) \mapsto M_{u,v}$ is a monoid homomorphism from $\{1, 2, 3\}^* \times \{1, 2, 3\}^*$ to the multiplicative monoid of 3×3 integer matrices:

Lemma 134 *For all $u, v, x, y \in \{1, 2, 3\}^*$ holds that $M_{u,v} M_{x,y} = M_{ux,vy}$.*

Proof. A direct calculation shows that

$$\begin{pmatrix} 4^{|u|} & 0 & 0 \\ 0 & 4^{|v|} & 0 \\ \sigma(u) & \sigma(v) & 1 \end{pmatrix} \begin{pmatrix} 4^{|x|} & 0 & 0 \\ 0 & 4^{|y|} & 0 \\ \sigma(x) & \sigma(y) & 1 \end{pmatrix} = \begin{pmatrix} 4^{|u|}4^{|x|} & 0 & 0 \\ 0 & 4^{|v|}4^{|y|} & 0 \\ 4^{|x|}\sigma(u)+\sigma(x) & 4^{|y|}\sigma(v)+\sigma(y) & 1 \end{pmatrix}$$

$$= \begin{pmatrix} 4^{|ux|} & 0 & 0 \\ 0 & 4^{|vy|} & 0 \\ \sigma(ux) & \sigma(vy) & 1 \end{pmatrix}$$

□

Let

$$A = \begin{pmatrix} 1 & 0 & 1 \\ -1 & 0 & -1 \\ 0 & 0 & 0 \end{pmatrix}.$$

This matrix is idempotent: $A^2 = A$. One can also easily calculate that

$$AM_{u,v}A = (4^{|u|} + \sigma(u) - \sigma(v))A. \tag{7}$$

We see that $AM_{u,v}A = 0$ if and only if

$$\sigma(v) = 4^{|u|} + \sigma(u) = \sigma(1u),$$

that is, if and only if $v = 1u$.

Theorem 135 *It is undecidable whether a given finite set of* 3×3 *integer matrices is mortal.*

Proof. We reduce the Post correspondence problem over the two letter alphabet $\Delta = \{2, 3\}$. This is undecidable by the remark at the end of Section 4.9.1. Let

$$\begin{array}{lcllll} L_1 & : & w_1, & w_2, & \ldots & w_k \\ L_2 & : & x_1, & x_2, & \ldots & x_k \end{array}$$

be a given instance of PCP. We construct the set of $2k + 1$ matrices that contains the matrix A above, and the matrices

$$M_i = M_{w_i,x_i} \qquad \text{and} \qquad M'_i = M_{w_i,1x_i}$$

for all $i = 1, 2, \ldots, k$. Let us show that this set is mortal if and only if the PCP instance has a solution.

Suppose first that the PCP instance has a solution $i_1 i_2 \ldots i_m$, so that $w_{i_1} w_{i_2} \ldots w_{i_m} = x_{i_1} x_{i_2} \ldots x_{i_m}$. Then

$$M'_{i_1} M_{i_2} M_{i_3} \ldots M_{i_m} = M_{u,v}$$

where $u = w_{i_1} w_{i_2} \ldots w_{i_m}$ and $v = 1x_{i_1} x_{i_2} \ldots x_{i_m}$. We have $v = 1u$, so by (7) holds $AM_{u,v}A = 0$. We see that the matrix set is mortal.

Conversely, suppose that the matrix set is mortal. This means that

$$AW_1AW_2A\dots AW_mA = 0$$

where each W_j is a product of some matrices M_i and M_i'. Each W_j is of the form $M_{u,v}$ for some words u, v over the alphabet $\{1,2,3\}$, so by (7) the matrices AW_jA are scalar multiples of A, say $AW_jA = a_jA$. Then

$$0 = AW_1AW_2A\dots AW_mA = a_1AW_2A\dots AW_mA = a_1a_2AW_3A\dots AW_mA\dots = a_1a_2\dots a_mA.$$

We conclude that some $a_j = 0$, that is, $AW_jA = 0$. Let $W_j = M_{u,v}$. By (7) we have that $v = 1u$. Because word u is over the alphabet $\{2,3\}$ we must have

$$W_j = M_{i_1}'M_{i_2}M_{i_3}\dots M_{i_m}$$

for some indices $i_1, i_2, \dots i_m$, with $m \geq 1$. We see that

$$\begin{aligned} u &= w_{i_1}w_{i_2}\dots w_{i_m} \\ v &= 1x_{i_1}x_{i_2}\dots x_{i_m} \end{aligned}$$

so $v = 1u$ implies that the PCP instance has a solution $i_1i_2\dots i_m$. □

Remarks: (1) Our proof converted a PCP instance of size k into a set of $2k+1$ matrices. Since the PCP is undecidable among $k = 5$ pairs of words, we see that the mortality problem is undecidable among sets of 11 integer matrices of size 3×3. A more careful analysis of the proof above, in fact, yields the undecidability of the mortality problem for sets of $k + 1 = 6$ matrices of size 3×3.

(2) Mortality is undecidable among sets of two 18×18 matrices, and among sets of three 9×9 matrices.

(3) It is not known whether the mortality problem is decidable among sets of 2×2 matrices. For a set of two 2×2 matrices the problem is decidable.

An interesting decision problem concerning a given single square matrix is the **Skolem-Pisot problem**: Given an $n \times n$ integer matrix M, does there exists $k \geq 1$ such that the element in the upper right corner of M^k is zero ? This is known to be decidable for matrices of size $n \leq 5$, but the decidability status is unknown for larger values of n.

4.9.4 Tiling problems

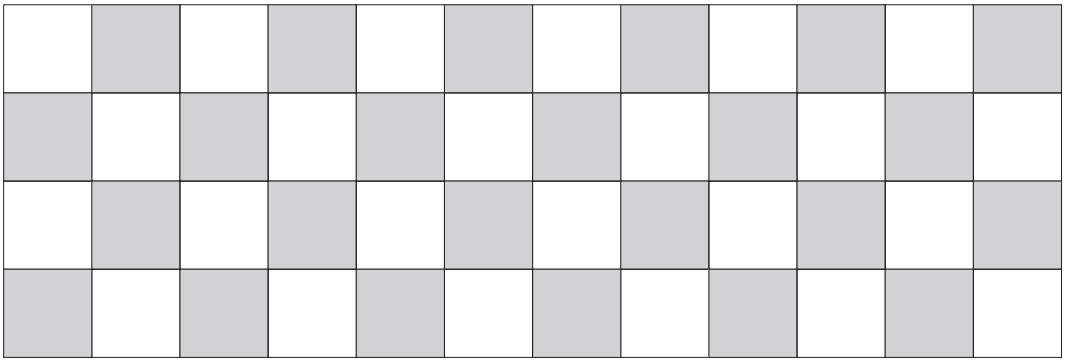
Wang tiles are unit square tiles with colored edges. Each tile can be represented as a 4-tuple (N, E, S, W) where N, E, S and W are the colors of the north, east, south and west sides of the square. Let T be a finite set of such tiles. A tiling is an assignment $t : \mathbb{Z}^2 \longrightarrow T$ of tiles on the plane in such a way that the adjacent edges of neighboring tiles have the same color.

Example 136. The tile set

$$T = \{\text{(Green, Green, Red, Red), (Red, Red, Green, Green)}\}$$

consists of two tiles

that only admit the checkerboard-tilings



□

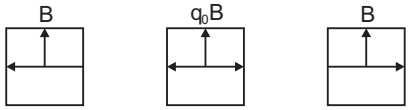
Note that the tiles may not be rotated. The **tiling problem** asks whether a given finite set of tiles admits at least one valid tiling. This question turns out to be undecidable. Here we, however, only prove the undecidability of the following variant, called the **tiling problem with a seed tile**: Given a finite set T of Wang tiles and a seed tile $s \in T$, does there exist a valid tiling of the plane that contains at least one copy s ?

To prove the undecidability of tiling problems we associate to each Turing machine M a set of Wang tiles such that valid tilings "draw" computations according to M. Horizontal rows represent consecutive ID's.

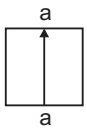
Theorem 137 *The tiling problem with a seed is undecidable.*

Proof. For any given Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, B, f)$ we effectively construct a Wang tile set T with $s \in T$ such that T admits a tiling containing s if and only M does not halt when started on the blank tape. The undecidability then follows from Corollary 107.

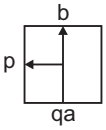
In the following description of the tile set T the colors on the edges are drawn as labeled arrows. The matching rule is that each arrow head must meet an arrow tail with the same label in the neighboring tile. Such presentation can easily be converted into colors by identifying each arrow direction/label -pair by a unique color. Tile set T contains the following three **initialization tiles**:



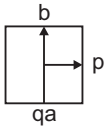
The second initialization tile is chosen as the seed tile s. For each tape symbol $a \in \Gamma$ there is the following **alphabet tile**:



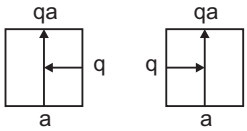
For each $q, p \in Q$ and $a, b \in \Gamma$ such that $\delta(q, a) = (p, b, L)$ we include the **left action tile**



and for each $q, p \in Q$ and $a, b \in \Gamma$ such that $\delta(q, a) = (p, b, R)$ we have the **right action tile**

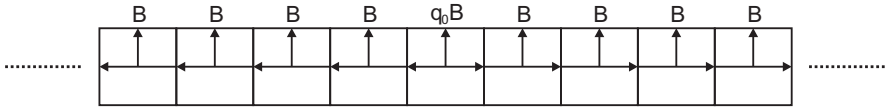


For every $q \in Q$ and $a \in \Gamma$ we have two **merging tiles**

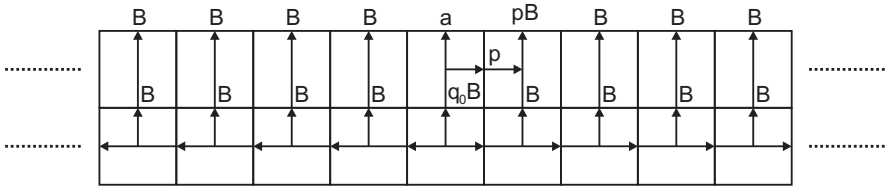


Finally, T contains the **blank tile**

Consider a valid tiling using T that contains a copy of the seed tile s. The horizontal row that contains the seed is necessarily tiled with the initialization tiles:



The labels on the arrows represent the initial configuration of the Turing machine on the initial blank tape. Above the seed tile fits only the action tile corresponding to the transition $\delta(q_0, B)$. The tile next to the action tile is forced to be the merging tile, and the other tiles on this row are necessarily the alphabet tiles. For instance, if $\delta(q_0, B) = (p, a, R)$ then the following row is forced:



As a result, the labels on the arrows on the second row represent the configuration of the TM after one step. Continuing likewise, we see that the rows above the seed row are forced to simulate the TM step-by-step. Because there is no action tile corresponding to qa if $\delta(q, a)$ is undefined, we see that the tiling can be completed to the whole plane if and only if the TM does not halt. The bottom half of the plane can always be filled with the blank tile. □

4.10 Undecidability and incompleteness in arithmetics

In this section we show that it is undecidable whether a given first-order mathematical statement concerning natural numbers is true. As a direct consequence we obtain the incompleteness result of Gödel: there are such statements that are true but cannot be proved to be true.

Recall that $\mathbb{N} = \{0, 1, 2, \ldots\}$ denotes the set of natural numbers (including 0). Arithmetic formula we consider may contain variables, natural number constants, operators $+$ and $\cdot$, relation symbol $=$, logical connectives $\vee, \wedge, \neg$ and quantifiers $\forall, \exists$. All these items have the usual, well-known interpretation in $\mathbb{N}$. An occurrence of variable x in a formula is **free** if it is not within the scope of a quantifier $\forall x$ or $\exists x$. A formula that has no free variables is a **statement**. Each statement is either true or false in $\mathbb{N}$, using the usual interpretation of the quantifiers and connectives. Formula that contains free variables becomes true or false when all free variables are given a value from $\mathbb{N}$.

Other symbols (such as relations $\neq$, $<$ and $\leq$, and connectives $\Rightarrow$ and $\Leftrightarrow$) can be used as shorthand notations as they can be expressed using the basic symbols. For example, $x \leq y$ is a shorthand notation for $\exists z\ (x + z = y)$, and $\varphi \Rightarrow \psi$ represents $\neg\varphi\ \vee\ \psi$.

We skip the formal, precise definitions of formulas and their truth values, as these are well known from other courses. Note that the formulas considered are first-order, meaning that all variables refer to single natural numbers. No second-order variables representing sets of numbers are used. We employ the following commonly used precedence rules: the evaluation is done in the order

$$1.\ \neg \qquad 2.\ \wedge \text{ and } \vee \qquad 3.\ \forall \text{ and } \exists \qquad 4.\ \Rightarrow.$$

The following example illustrates the concepts.

Example 138. The formula

$$\neg(z = 1) \wedge \forall x\ (x = 1) \vee (x = z) \vee (\forall y\ \neg(xy = z))$$

is a formula with one free variable z. The formula is true iff z is a prime number. Here is an equivalent formula using common shorthand notations, also stating the primality of z:

$$(z > 1) \wedge \forall x, y\ (x, y > 1 \Rightarrow xy \neq z).$$

Formula

$$\forall m\ \exists z\ (z > m) \wedge \forall x, y\ (x, y > 1 \Rightarrow xy \neq z)$$

has no free variables, so it is a statement. The statement is true: it states that there are infinitely many prime numbers. The formula

$$\forall m\ \exists z(z > m) \wedge \forall x, y\ [x, y > 1 \Rightarrow (xy \neq z) \wedge (xy \neq z + 2)]$$

is also a statement. It states that there are infinitely many twin primes (=primes p and $p + 2$). It is not known whether this statement is true or false. □

In the following, we build complicated formulas step-by-step, using simpler formulas as building blocks. So we may introduce new relation symbols as shorthand notations. For example, we could define formula Prime(z) to be

$$(z > 1) \wedge \forall x, y\ (x, y > 1 \Rightarrow xy \neq z),$$

and using this formula we can express the twin prime conjecture as the statement

$$\forall m\ \exists z(z > m) \wedge \text{Prime}(z) \wedge \text{Prime}(z+2).$$

It would be very useful if there would exist an algorithm to determine if a given first-order statement over $\mathbb{N}$ is true or false. But it turns out that such an algorithm does not exist. We prove this fact using a reduction from the Post correspondence problem. In the proof we use the notations from Section 4.9.3 on matrix product mortality: We assume a PCP instance over the binary alphabet $\Sigma = \{2, 3\}$, and for any word $w = a_1 a_2 \ldots a_m$ over the alphabet $\{2, 3\}$ we define the natural number

$$\sigma(w) = a_m + 4a_{m-1} + \ldots + 4^{m-1} a_1.$$

(Note: we could equally well use the alphabet $\{1, 2\}$ and base 3 instead of $\{2, 3\}$ and base 4, but for the sake of consistency with Section 4.9.3 we stick to the prior notation.)

Theorem 139 *It is undecidable whether a given first-order arithmetic statement is true.*

Proof. We reduce the Post correspondence problem. Let

$$\begin{array}{lcllll} L_1 & : & w_1, & w_2, & \ldots & w_k \\ L_2 & : & x_1, & x_2, & \ldots & x_k \end{array}$$

be a given instance of PCP, over the binary alphabet $\Sigma = \{2, 3\}$. In the following, we construct an arithmetic statement that is true if and only if the PCP instance has a solution. Let us denote, for all $i = 1, \ldots, k$,

$$\begin{array}{rcl} A_i & = & \sigma(w_i), \\ B_i & = & 4^{|w_i|}, \\ C_i & = & \sigma(x_i), \text{ and} \\ D_i & = & 4^{|x_i|}. \end{array}$$

These are natural number constants that can be computed from the given PCP instance. Notice that these are the elements in the 3×3 matrices

$$M_i = M_{w_i, x_i} = \begin{pmatrix} B_i & 0 & 0 \\ 0 & D_i & 0 \\ A_i & C_i & 1 \end{pmatrix}$$

we defined in the proof of Theorem 135 on matrix product mortality. Let $i_1 i_1 \ldots i_m$ be a sequence of indices, $i_j \in \{1, 2, \ldots, k\}$. For every $j = 0, 1, \ldots, m$ we define the natural numbers

$$\begin{array}{rcl} a_j & = & \sigma(w_{i_1} \ldots w_{i_j}), \\ b_j & = & 4^{|w_{i_1} \ldots w_{i_j}|}, \\ c_j & = & \sigma(x_{i_1} \ldots x_{i_j}), \text{ and} \\ d_j & = & 4^{|x_{i_1} \ldots x_{i_j}|}. \end{array}$$

Notice that in the matrix notation these are the elements of the product

$$M_{i_1} M_{i_1} \ldots M_{i_j} = \begin{pmatrix} b_j & 0 & 0 \\ 0 & d_j & 0 \\ a_j & c_j & 1 \end{pmatrix}.$$

Because

$$\begin{pmatrix} b_{j-1} & 0 & 0 \\ 0 & d_{j-1} & 0 \\ a_{j-1} & c_{j-1} & 1 \end{pmatrix} \begin{pmatrix} B_{i_j} & 0 & 0 \\ 0 & D_{i_j} & 0 \\ A_{i_j} & C_{i_j} & 1 \end{pmatrix} = \begin{pmatrix} B_{i_j} b_{j-1} & 0 & 0 \\ 0 & D_{i_j} d_{j-1} & 0 \\ B_{i_j} a_{j-1} + A_{i_j} & D_{i_j} c_{j-1} + C_{i_j} & 1 \end{pmatrix}$$

the numbers are determined by the equations

$$a_0 = c_0 = 0 \text{ and } b_0 = d_0 = 1, \qquad \text{(Start)},$$

$$\begin{aligned} &\forall j = 1, \dots m && \text{(Follow)}, \\ &\qquad a_j &&= B_{i_j} a_{j-1} + A_{i_j}, \\ &\qquad b_j &&= B_{i_j} b_{j-1}, \\ &\qquad c_j &&= D_{i_j} c_{j-1} + C_{i_j}, \\ &\qquad d_j &&= D_{i_j} d_{j-1}. \end{aligned}$$

Because σ is injective, the sequence $i_1 i_2 \dots i_m$ is a solution to the PCP if and only if the condition

$$a_m = c_m \qquad \text{(End)}.$$

holds. We see that the PCP instance has a solution if and only if numbers a_j, b_j, c_j and d_j satisfying the conditions (First), (Follow) and (End) exist for some choice of $m \geq 1$ and $i_1 i_2 \dots i_m$. In other words, the PCP instance has a solution if and only if the "statement"

$$\begin{aligned} &\exists m \ (m \geq 1) \wedge \exists a_0, \dots, a_m \exists b_0, \dots, b_m \exists c_0, \dots, c_m \exists d_0, \dots, d_m \\ &\qquad \text{Start}(a_0, b_0, c_0, d_0) \wedge \\ &\qquad\qquad \forall j \ \ [(j < m) \Rightarrow \text{Follow}(a_j, b_j, c_j, d_j, a_{j+1}, b_{j+1}, c_{j+1}, d_{j+1})] \wedge \\ &\qquad\qquad\qquad \text{End}(a_m, b_m, c_m, d_m) \end{aligned} \tag{8}$$

is true, where we use the shorthand notation

- Start(a,b,c,d) for $(a = 0) \wedge (b = 1) \wedge (c = 0) \wedge (d = 1)$,
- Follow(a,b,c,d,a',b',c',d') for

$$\begin{aligned} &[(a' = B_1 a + A_1) \wedge (b' = B_1 b) \wedge (c' = D_1 c + C_1) \wedge (d' = D_1 d)] \quad \vee \\ &\qquad [(a' = B_2 a + A_2) \wedge (b' = B_2 b) \wedge (c' = D_2 c + C_2) \wedge (d' = D_2 d)] \quad \vee \\ &\qquad\qquad \dots \\ &\qquad\qquad\qquad [(a' = B_k a + A_k) \wedge (b' = B_k b) \wedge (c' = D_k c + C_k) \wedge (d' = D_k d)] \,, \end{aligned}$$

- End(a,b,c,d) for $a = c$.

The problem with (8) is that it is not a valid first-order statement because the number of quantified variables a_i, b_i, c_i and d_i is not constant but depends on the value of variable m (representing the length of the solution). Fortunately, this problem can be resolved by observing that arbitrary finite sequences of natural numbers can be encoded as single numbers.

For the simplicity of expressing such encoding in the first order logic, we choose to encode sequences as two numbers, using **Gödel's β-function**. For any natural numbers a, b, i we define

$$\beta(a, b, i) = a \bmod [1 + (i + 1)b],$$

where $x \bmod y$ denotes the smallest natural number that is congruent to x modulo y. Note that $n = \beta(a, b, i)$ if and only if

$$(n < 1 + (i + 1)b) \wedge [\exists x \; a = n + x(1 + (i + 1)b)]$$

is true. Let us denote this formula by $\mathrm{Beta}(a, b, i, n)$.

The idea of the β-function is that it can reproduce sequences of numbers from fixed a and b by varying i. Any desired sequence can be obtained, as stated by the following lemma:

Lemma 140 *Let $m \geq 1$, and let $n_0, n_1, \ldots n_m$ be arbitrary natural numbers. Then there exist $a, b \in \mathbb{N}$ such that $n_i = \beta(a, b, i)$ for all $i = 0, 1, \ldots, m$.*

Proof of the lemma. Let us choose $b = m! \cdot \max\{n_0, n_1, \ldots, n_m\}$. Let us first show that numbers $1+(i+1)b$ are pairwise coprime: Let p be a prime number that divides both $1+(i+1)b$ and $1+(j+1)b$ where $0 \leq i < j \leq m$. Then p also divides the difference $(1 + (j + 1)b) - (1 + (i + 1)b) = (j - i)b$. Because $j - i$ divides b we see that p necessarily divides b, which contradicts the fact that it also divides $1 + (i + 1)b$.

As numbers $1+(i+1)b$ are pairwise coprime, the Chinese remainder theorem provides a natural number a such that $a \equiv n_i \pmod{1+(i+1)b}$, for all $i = 0, 1, \ldots, m$. Because $n_i \leq b < 1+(i+1)b$, we have that $n_i = \beta(a, b, i)$. □

Now each of the variable sequences in (8) can be replaced by two variables. As we need four sequences of numbers, we encode them in four pairs: α_1, α_2 provide a_i's, β_1, β_2 provide b_i's, γ_1, γ_2 provide c_i's, and δ_1, δ_2 provide d_i's. We obtain the statement

$$\begin{aligned}
&\exists m \; (m \geq 1) \wedge \exists \alpha_1, \alpha_2, \beta_1, \beta_2, \gamma_1, \gamma_2, \delta_1, \delta_2 \\
&\quad \mathrm{Beta}(\alpha_1, \alpha_2, 0, 0) \wedge \mathrm{Beta}(\beta_1, \beta_2, 0, 1) \wedge \mathrm{Beta}(\gamma_1, \gamma_2, 0, 0) \wedge \mathrm{Beta}(\delta_1, \delta_2, 0, 1) \wedge \\
&\quad \forall j \; [(j < m) \Rightarrow \exists a, b, c, d, a', b', c', d' \;\; \mathrm{Follow}(a, b, c, d, a', b', c', d') \wedge \\
&\qquad\qquad \mathrm{Beta}(\alpha_1, \alpha_2, j, a) \wedge \mathrm{Beta}(\alpha_1, \alpha_2, j + 1, a') \wedge \\
&\qquad\qquad \mathrm{Beta}(\beta_1, \beta_2, j, b) \wedge \mathrm{Beta}(\beta_1, \beta_2, j + 1, b') \wedge \\
&\qquad\qquad \mathrm{Beta}(\gamma_1, \gamma_2, j, c) \wedge \mathrm{Beta}(\gamma_1, \gamma_2, j + 1, c') \wedge \\
&\qquad\qquad \mathrm{Beta}(\delta_1, \delta_2, j, d) \wedge \mathrm{Beta}(\delta_1, \delta_2, j + 1, d')] \wedge \\
&\quad \exists r \; \mathrm{Beta}(\alpha_1, \alpha_2, m, r) \wedge \mathrm{Beta}(\gamma_1, \gamma_2, m, r),
\end{aligned} \tag{9}$$

which is true if and only if the PCP instance has a solution. Note that the (Start) condition

$$\mathrm{Beta}(\alpha_1, \alpha_2, 0, 0) \wedge \mathrm{Beta}(\beta_1, \beta_2, 0, 1) \wedge \mathrm{Beta}(\gamma_1, \gamma_2, 0, 0) \wedge \mathrm{Beta}(\delta_1, \delta_2, 0, 1)$$

just states that

$$(a_0 = 0) \wedge (b_0 = 1) \wedge (c_0 = 0) \wedge (d_0 = 1)$$

and the (End) condition

$$\exists r \; \mathrm{Beta}(\alpha_1, \alpha_2, m, r) \wedge \mathrm{Beta}(\gamma_1, \gamma_2, m, r)$$

states that there exists number r that is equal to both a_m and c_m, that is, $a_m = c_m$. The middle part of the formula extracts $a_j, b_j, c_j, d_j, a_{j+1}, b_{j+1}, c_{j+1}$ and d_{j+1} into variables a, b, c, d, a', b', c' and d', and verifies that the (Follow) condition is satisfied at step j.

The undecidability of determining the truth value of a first order statement now follows from the undecidability of PCP. □

As an immediate corollary we see that that the same problem is not even semi-decidable:

Corollary 141 *There is no semi-algorithm to determine if a given first-order arithmetic statement is true.*

Proof. For any statement φ either φ is true or $\neg\varphi$ is true. If there is a semi-algorithm for true statements then we can execute this semi-algorithm on inputs φ and $\neg\varphi$ in parallel until we receive an answer. This provides an algorithm to determine whether φ is true, contradicting Theorem 139. □

The previous corollary has far reaching consequences on the (un)axiomatizability of arithmetics. The statements that can be deduced from any finite axiom set form a semi-decidable set. Indeed, to check whether a given statement follows from the axioms, one can enumerate longer and longer deductions until (if ever) the given statement is obtained. Hence, by the previous corollary, the sets of provable statements and true statements cannot coincide: One is semi-decidable while the other one is not. There always remain true statements that have no proof from the axioms. This is Gödel's celebrated incompleteness theorem.

More generally, consider any proof system for arithmetics. A reasonable request for such a system is that proofs π are finite sequences of symbols, and that there is an algorithm to verify that a given proof π is a correct proof for a given statement φ. For any such proof system, there is a semi-algorithm to determine if a given statement φ has a correct proof in the system. (Indeed, enumerate longer and longer strings π until a valid proof for φ is found.) Hence, by the previous corollary, the sets of true statements and provable statements cannot be identical.

Remark: An important subfamily of first-order arithmetic formulas are diophantine equations. These are polynomial equations with integer coefficients, and one is interested in finding integer solutions. For example, $x^3+y^3 = z^3$ is a diophantine equation with three variables. It has solutions $x = 0, y = z$ and $y = 0, x = z$ among natural numbers, so the statement $\exists x\ \exists y\ \exists z\ x^3 + y^3 = z^3$ is true. On the other hand, statement $\exists x\ x^2 + 1 = 0$ is clearly false in $\mathbb{N}$.

In 1900, D.Hilbert proposed a list of 23 open mathematical problems, that have greatly influenced the mathematical research in the 20'th century. The 10'th problem in the list asked to "device a process" (=algorithm, in modern terms) to determine if a given diophantine equation has an integer solution. The problem is algorithmically equivalent if natural number solutions are required rather than integer solutions. In 1970, the problem was proved undecidable by Y.Matiyasevich. This means that there is no algorithm to determine if the first-order formula

$$\exists x \exists y \ldots\ P(x, y, \ldots) = Q(x, y, \ldots)$$

over $\mathbb{N}$ is true, where P and Q are given polynomials with natural number coefficients, that is, P and Q are given terms formed using $+$, $\cdot$, variables and constants.

4.11 Computable functions and reducibility

So far we have considered decision problems and associated languages. In many setups it would be useful to have a more complex output from an algorithm than just "yes" or "no". Informally, we say that a (total) function $f : \Sigma^* \longrightarrow \Delta^*$ is **total computable** (or a **total recursive function**) if there is an algorithm that produces from any input $w \in \Sigma^*$ the output $f(w) \in \Delta^*$. A partial function $f : \Sigma^* \longrightarrow \Delta^*$ is **partial computable** (or a **partial recursive function**) if there is a semi-algorithm-like process that produces from input $w \in \Sigma^*$ the output $f(w) \in \Delta^*$ if $f(w)$ is defined, and returns no answer if $f(w)$ is not defined.

In applications, algorithms are applied on objects other than words. However, computation is defined on words and therefore we encode inputs and outputs as words, exactly as we did when we connected decision problems to languages. Using some fixed encodings, we say that a total (or partial) function $f : D \longrightarrow R$ from domain D to range R is total (partial, respectively) computable if there is a total (partial, respectively) word function that maps $\langle d \rangle \mapsto \langle f(d) \rangle$ for all $d \in D$ such that $f(d)$ is defined, and is undefined on $\langle d \rangle$ if f is not defined on d. The behavior of the word function on input words that are not valid encodings is irrelevant. (As we request that the set of valid encodings is a recursive language, the algorithm that computes the word function $\langle d \rangle \mapsto \langle f(d) \rangle$ can first recognize whether the input word is a valid encoding or not.)

All reasonable encodings can be effectively converted into each other, so the choice of the encoding is rather arbitrary. (The fact that encodings $\langle \cdot \rangle_1$ and $\langle \cdot \rangle_2$ of set X can be effectively converted into each other is simply stating that there are total computable word functions f and g that map $f : \langle x \rangle_1 \mapsto \langle x \rangle_2$ and $g : \langle x \rangle_2 \mapsto \langle x \rangle_1$ for all $x \in X$.)

Example 142. If $D = R = \mathbb{N}$ we may choose to use the unary encoding $\langle n \rangle = a^n$ into the alphabet $\{a\}$. In this case, a function $f : \mathbb{N} \longrightarrow \mathbb{N}$ is total computable iff the function $a^n \mapsto a^{f(n)}$ is a total computable function $a^* \longrightarrow a^*$. If we decided to use the usual binary encoding $\text{bin}(n)$ of numbers in the binary alphabet $\{0, 1\}$ then exactly same functions $f : \mathbb{N} \longrightarrow \mathbb{N}$ would be computable because there are total computable conversions $\text{bin}(n) \longleftrightarrow a^n$ between the encodings. □

Example 143. Let us define the **Busy Beaver** -function $BB : \mathbb{N} \longrightarrow \mathbb{N}$ as follows. For any positive integer n, let $TM(n)$ denote the set of all Turing machines that

(i) have n non-final states (and one final state),

(ii) have two tape symbols (one of which is the blank symbol), and

(iii) eventually halt when started on the blank tape.

In other words, Turing machines in $TM(n)$ are of the form $(\{q_1, q_2, \ldots, q_n, f\}, \{a\}, \{a, B\}, \delta, q_1, B, f)$ and they halt when started with the empty input ε.

The value $BB(n)$ is defined as the maximum number of moves that any machine in $TM(n)$ makes before halting when started on the blank tape. (And let us define $BB(0) = 0$ to make the function total.) It is known that $BB(1) = 1$, $BB(2) = 6$, $BB(3) = 21$, $BB(4) = 107$ and $BB(5) = 47176870$. The value of $BB(n)$ is not known for larger values of n, but it is known that $BB(6) \geq 10^{2879}$. The values of the function grow very rapidly.

Let us show that $BB : \mathbb{N} \longrightarrow \mathbb{N}$ is not total computable and that, in fact, for any total computable $g : \mathbb{N} \longrightarrow \mathbb{N}$ there exists n such that $BB(n) > g(n)$. So the function BB is not

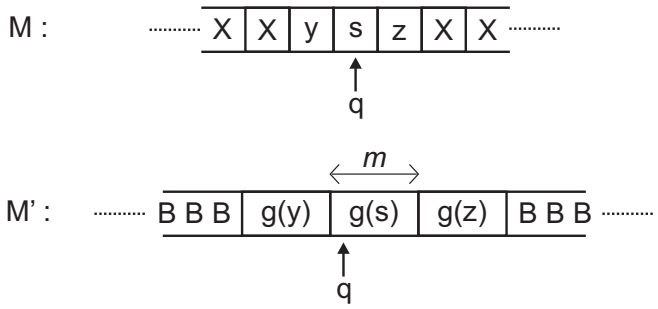
bounded by any total computable function g. This is based on the fact that if there were a total computable g such that $BB(n) \leq g(n)$ for all $n \in \mathbb{N}$, then one could effectively determine if a given TM M that satisfies conditions (i) and (ii) above halts when started on the blank tape: One first computes $g(n)$ and then simulates M for up to $g(n)$ steps. If M does not halt by the $g(n)$'th step, the machine never halts.

This contradicts the undecidability of the halting problem of Turing machines from the empty tape, proved in Corollary 107. Actually, we need to know that this undecidability holds even when the machine considered has only two tape letters:

Lemma 144 *It is undecidable whether a given TM with two tape symbols halts when started on the blank tape.*

Proof. We reduce the halting problem of TM with arbitrary tape alphabet, shown undecidable in Corollary 107. Let $M = (Q, \Sigma, \Gamma, \delta, q_0, X, f)$ be an arbitrary Turing machine, with blank symbol X. Let us choose $m = \lceil \log_2 |\Gamma| \rceil$ so that there is an injective function $g : \Gamma \longrightarrow \{a, B\}^m$. We can choose g in such a way that $g(X) = B^m$. Let us extend g into a homomorphism $\Gamma^* \longrightarrow \{a, B\}^*$.

We can effectively construct a TM M' with tape alphabet $\{a, B\}$ that simulates M on any input w as follows. The corresponding input to M' is $g(w)$, so all tape symbols of M are encoded as blocks of length m. To simulate the next move of M, the simulating machine M' is located on the first letter of the block that encodes the symbol currently scanned by M:



One move of M is simulated by M' as follows:

(1) Move m cells to the right, memorizing the symbols seen. The tape symbol s scanned by M is now known to M', so M' can compute $\delta(q, s) = (p, b, d)$.

(2) Move m cells back to the left, replacing the symbols on the tape by the word $g(b)$.

(3) Move m cells to the left or right depending on whether d is L or R, and change to state p. Now M' is ready to simulate the next move of M.

Because the encoding of the blank tape of M is the blank tape in M', it is clear that M' halts from the empty initial tape if and only if M halts from the empty initial tape. □

More precisely, we use Turing machines to define total and partial computable functions $f : \Sigma^* \longrightarrow \Delta^*$. Let $M = (Q, \Sigma \cup \Delta, \Gamma, \delta, q_0, B, q_F)$ be a Turing machine, with final state q_F. We define the halting ID ζ_w corresponding to word $w \in \Delta^*$ analogously to the initial ID ι_w:

$$\zeta_w = \begin{cases} q_F w, & \text{if } w \neq \varepsilon, \\ q_F B, & \text{if } w = \varepsilon. \end{cases}$$

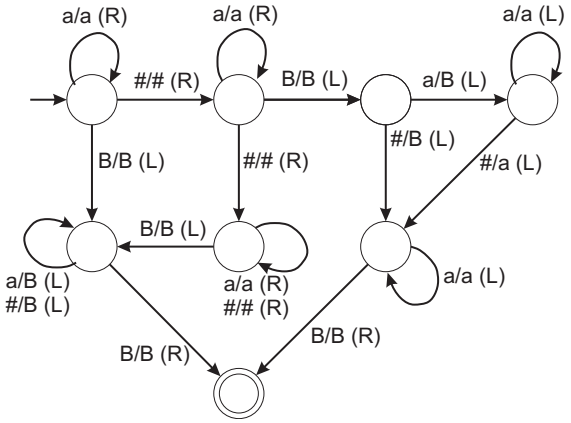
The partial function $f_M : \Sigma^* \longrightarrow \Delta^*$ computed by M is

$$f_M(w) = \begin{cases} u, & \text{if } \iota_w \vdash^* \zeta_u \text{ and } u \in \Delta^* \ , \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

In other words, we require the Turing machine to write the output $f_M(w)$ on the tape, move on the first letter of the output and halt in state q_F. In all other cases (if M does not halt or halts in a different ID) the value $f_M(w)$ is undefined.

A function f is a **partial computable function** if there exists a Turing machine M such that $f = f_M$. If, moreover, f is defined for all $w \in \Sigma^*$ then f is a **total computable function**.

Example 145. The function $a^n \# a^m \mapsto a^{n+m}$ and $w \mapsto \varepsilon$ for $w \notin a^* \# a^*$ is a total recursive function $\{a, \#\}^* \longrightarrow a^*$. It is computed by the TM described by the diagram



where vertices are the states, and for each transition $\delta(q, a) = (p, b, D)$ there is an edge in the diagram from vertex q into vertex p labeled by a/b (D). The initial state q_0 is indicated by a single incoming arrow and the final state q_F is the one with a double circle.

The idea of the machine is very simple: it first scans the input from left-to-right to check that the input contains one occurrence of letter # (if not, the machine erases the input and halts), and then returns to the beginning, replacing the symbol # by the last a of the input.

This example shows that the function $(n, m) \mapsto n + m$ from $\mathbb{N} \times \mathbb{N}$ to $\mathbb{N}$ is total computable when we use the natural encodings $\langle n, m \rangle = a^n \# a^m$ and $\langle k \rangle = a^k$ for all $n, m, k \in \mathbb{N}$. □

The following observation is easy: Language $L \subseteq \Sigma^*$ is recursive if and only if its characteristic function $\chi_L : \Sigma^* \longrightarrow \{0, 1\}^*$ defined by

$$\chi_L(w) = \begin{cases} 0, & \text{if } w \notin L, \\ 1, & \text{if } w \in L \end{cases}$$

is total recursive. Analogously, $L \subseteq \Sigma^*$ is recursively enumerable iff the partial function obtained by changing output 0 of $\chi_L(w)$ to "undefined" is partial recursive.

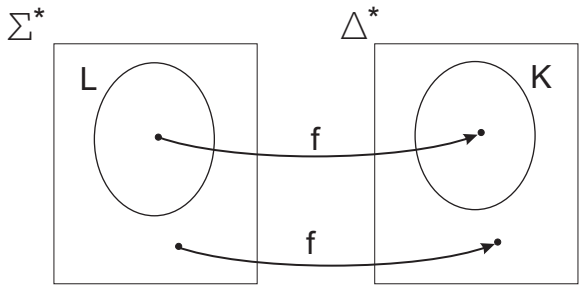
As we know, Turing machines are powerful enough devices to compute anything computable by other types of algorithms (e.g. arbitrary computer programs). Taking advantage of this observation, in order to show that a function f is total (or partial) computable, we simply describe from now on informal algorithms (or semi-algorithms) with outputs, instead of providing exact Turing machines that compute f.

Example 146. Our earlier Example 108 describes a function $f : \langle M\rangle \# w \mapsto \langle M'\rangle$ that associates to a Turing machine M and word w a Turing machine M' such that $L(M') = \{a, b\}^*$ if $w \in L(M)$, and $L(M') = \emptyset$ if $w \notin L(M)$. The construction of M' from given M and w is effective, so the function f is total computable. □

Let $L \subseteq \Sigma^*$ and $K \subseteq \Delta^*$ be two languages. A total computable function $f : \Sigma^* \longrightarrow \Delta^*$ is a **reduction** of L to K if for all $w \in \Sigma^*$

$$w \in L \iff f(w) \in K.$$

If a reduction exists from L to K, we denote $L \leq_m K$ and say that L is **many-one reducible** (or **mapping reducible**) to K.



Transitivity of $\leq_m$ is obvious:

Lemma 147 *If $L_1 \leq_m L_2$ and $L_2 \leq_m L_3$ then $L_1 \leq_m L_3$.*

Proof. Let f_1 and f_2 be the reductions of L_1 to L_2, and L_2 to L_3, respectively. Then the composition $w \mapsto f_2(f_1(w))$ is a reduction of L_1 to L_3. □

Because a reduction is a computable function, if $L \leq_m K$ then any algorithm solving the membership problem of K can be used to solve the membership problem of L. This justifies the reduction method we have used in the undecidability proofs.

Theorem 148 *Let $L \subseteq \Sigma^*$ and $K \subseteq \Delta^*$ be languages such that $L \leq_m K$. Then also $\Sigma^* \setminus L \leq_m \Delta^* \setminus K$. Moreover,*

- *if K is recursive then L is recursive,*
- *if K is recursively enumerable then L is recursively enumerable,*
- *if $\Delta^* \setminus K$ is recursively enumerable then $\Sigma^* \setminus L$ is recursively enumerable.*

Proof. The same reduction f that reduces L to K also reduces the complement $\Sigma^* \setminus L$ to the complement $\Delta^* \setminus K$. If K is recursive (or r.e.) then the following algorithm (semi-algorithm, respectively) determines if given w is in L:

1. Compute $u = f(w)$ where f is the reduction from L to K.

2. Test whether u is in K using the algorithm (semi-algorithm, respectively) for K.

□

More informally, we say that a decision problem A is many-one reducible to a decision problem B if $L_A \leq_m L_B$ where L_A and L_B are the languages containing the encodings of the positive instances of A and B, respectively. In other words, there is an effective conversion that takes positive instances of A into positive instances of B, and negative instances of A into negative instances of B. Then, as shown by the previous theorem, if problem A is known not to be decidable (or semi-decidable) we can conclude that B is not decidable (or semi-decidable) either. All our reductions in prior sections were of this type.

Example 149. Example 108 provides a many-one reduction from $L_u = \{\langle M\rangle \# w \mid w \in L(M)\}$ to $K = \{\langle M\rangle \mid L(M) \neq \emptyset\}$. Because the complement of L_u is not r.e., we concluded that the complement of K is not r.e. either. □

We call an r.e language K **r.e.-complete** if for every r.e. language L we have $L \leq_m K$. In other words, K is an r.e. language such that all r.e. languages can be many-one reduced to K. We already know examples of r.e.-complete languages:

Theorem 150 *The language $L_u = \{\langle M\rangle \# w \mid w \in L(M)\}$ is r.e.-complete.*

Proof. Language L_u is recursively enumerable (Theorem 109). We just need to show that all r.e. languages can be many-one reduced to L_u. We first observe that any r.e. language $L' \subseteq \Sigma^*$ over an arbitrary alphabet Σ is many-one reducible to an r.e. language $L \subseteq \{a, b\}^*$ over the alphabet $\{a, b\}$. Such reduction is provided, for example, by choosing any injective homomorphism $h : \Sigma^* \longrightarrow \{a, b\}^*$ and $L = h(L')$. Function h is clearly total computable and hence a reduction of L' into L. Language L is r.e. because the family of r.e. languages is closed under homomorphisms (homework).

By transitivity of $\leq_m$, is it enough to show that $L \leq_m L_u$. Because L is r.e. there is a TM M such that $L = L(M)$. Define the function f that maps $w \in \{a, b\}^*$ into the word $\langle M\rangle \# w$. Clearly such f is total computable, and it reduces L into L_u. □

Once language L_u is known to be r.e.-complete, many-one reductions can be used to find other r.e.-complete languages:

Theorem 151 *Let L be an r.e.-complete language. If K is an r.e. language such that $L \leq_m K$ then also K is r.e.-complete*

Proof. Let M be an arbitrary r.e. language. Then $M \leq_m L$, so by transitivity of $\leq_m$ we have $M \leq_m K$. □

We say that a decision problem is r.e.-complete if the encodings of its positive instances form an r.e.-complete language, that is, if the decision problem is semi-decidable and every semi-decidable problem can be many-one reduced into it. By the previous theorem, all semi-decidable problems that we proved undecidable using a many-one reduction from L_u are r.e.-complete. Hence we can immediately conclude the following:

- "Does a given TM halt when started on the blank initial tape ?" is an r.e.-complete decision problem.
- There is a semi-Thue system T and a word u such that individual word problem "Does a given x derive u in T ?" is r.e.-complete. There is a Thue system with the same property.
- "Does a given PCP instance have a solution ?" is r.e.-complete.
- Questions "Is $L_1 \cap L_2 \neq \emptyset$ for given CFL L_1 and L_2 ?", "Is given CFG G ambiguous ?" and "Is $L \neq \Sigma^*$ for a given CFL L ?" are r.e.-complete.
- "Does a given Wang tile set not admit a valid tiling that contains a given seed tile ?" is r.e.-complete.

If (not necessarily r.e.) language K has the property that $L \leq_m K$ for all r.e. languages L then we say that K is **r.e.-hard**. Hence a language is r.e.-complete if and only if it is r.e.-hard and it is r.e. Clearly a language is r.e.-hard if some r.e.-complete language K can be many-one reduced into it. An analogous terminology is used on decision problems. The proof of the Rice's theorem shows that all non-trivial questions (or their complements) about r.e. languages are r.e.-hard. We have also shown the r.e.-hardness of determining whether a given first-order arithmetic statement over $\mathbb{N}$ is true.

Remark: Many-one reducibility is only one form of reducibility that can be used to show that a language is not r.e. Language L is said to be **Turing reducible** to language K, denoted by $L \leq_T K$, if there is an algorithm for the membership problem of L that may use as a subroutine a hypothetical algorithm (=oracle) that solves the membership problem of K. The oracle may be invoked several times and the input to the oracle may depend on the answers the oracle has provided on previous queries. Many-one reductions are special types of Turing reductions where the oracle is invoked only once, and this happens at the end of the algorithm and the answer from the oracle is directly relayed as the final answer of the algorithm. Still, if $L \leq_T K$ and K is known to be recursive (or r.e) then L has to be recursive (or r.e., respectively) as well. So Turing reductions can also be used to prove undecidability results.

4.12 Some other universal models of computation

We have used Turing machines as the mathematical model to precisely define the concept of decidability/undecidability. In this last section we briefly describe (without detailed proofs) some other simple devices that are computationally as powerful as Turing machines.

4.12.1 Counter machines

A deterministic **counter machine** (also known as the **Minsky machine**) is a 5-tuple $M = (Q, k, \delta, q_0, f)$ where Q is a finite state set and $k \geq 1$ is the number of counters. We say that M is a k-counter machine. States $q_0, f \in Q$ are the initial state and the final state. The machine is a deterministic finite state automaton, supplied with k counters, each capable of storing a natural number. ID's of the device are tuples $(q, n_1, n_2, \ldots, n_k)$ where $q \in Q$ is the current state and $n_1, n_2, \ldots, n_k \in \mathbb{N}$ are the current contents of the counters. For each $n \in \mathbb{N}$, let us denote

$$\text{sign}(n) = \begin{cases} Z, \text{ if } n = 0, \\ P, \text{ if } n > 0. \end{cases}$$

The transition function δ is a partial function

$$Q \times \{Z, P\}^k \longrightarrow Q \times \{-1, 0, 1\}^k.$$

The idea is that a transition

$$\delta(q, p_1, p_2, \ldots, p_k) = (p, d_1, d_2, \ldots, d_k)$$

can be applied to those ID $(q, n_1, n_2, \ldots, n_k)$ where $\text{sign}(n_j) = p_j$ for every $j = 1, 2, \ldots, k$. In this case, the move

$$(q, n_1, n_2, \ldots, n_k) \vdash (p, n_1 + d_1, n_2 + d_2, \ldots, n_k + d_k)$$

is made. In other words, the next move depends on the current state q and the sign of the counter values in all k counters. The transition function δ provides the new state p and allows each counter to be decremented, incremented or left unchanged, as indicated by the values d_j.

To keep the counter values non-negative, the constraint is imposed that in each transtion

$$\delta(q, p_1, p_2, \ldots, p_k) = (p, d_1, d_2, \ldots, d_k),$$

if $p_j = Z$ then $d_j \neq -1$. Moreover, to prevent further moves from the final state f, we also require that $\delta(f, p_1, p_2, \ldots, p_k)$ are undefined for all choices of p_j.

The counter machine recognizes subsets of $\mathbb{N}$ rather than languages. In the beginning of a computation the machine is in state q_0, and the input number n is written in the first counter, while the other counters contain 0. The input is accepted if the machine eventually enters the final state f. So the set $L(M) \subseteq \mathbb{N}$ recognized by M is

$$L(M) = \{n \in \mathbb{N} \mid (q_0, n, 0, \ldots, 0) \vdash^* (f, n_1, n_2, \ldots, n_k) \text{ for some } n_1, n_2, \ldots, n_k \in \mathbb{N}\}.$$

Example 152. Consider the 2-counter machine $M = (\{q_0, q_1, q_2, f\}, 2, \delta, q_0, f)$ with transitions

$$\begin{array}{lcl}
\delta(q_0, P, *) & = & (q_1, -1, 0) \\
\delta(q_0, Z, *) & = & (q_2, 0, 0) \\
\delta(q_1, Z, Z) & = & (f, 0, 0) \\
\delta(q_1, P, *) & = & (q_0, -1, +1) \\
\delta(q_2, *, P) & = & (q_2, +1, -1) \\
\delta(q_2, *, Z) & = & (q_0, 0, 0)
\end{array}$$

In the transitions, symbol $*$ is used to indicate that both Z and P are allowed. This machine recognizes the set $L(M) = \{2^k \mid k \geq 0\}$. The idea is that the first counter value is divided by two (using a loop in which counter 1 is decremented twice and counter 2 is incremented once), and this is repeated until an odd value is reached. The number is accepted iff this odd value is one. For example,

$$\begin{array}{l}
(q_0, 4, 0) \vdash (q_1, 3, 0) \vdash (q_0, 2, 1) \vdash (q_1, 1, 1) \vdash (q_0, 0, 2) \vdash (q_2, 0, 2) \vdash (q_2, 1, 1) \vdash (q_2, 2, 0) \\
\quad \vdash (q_0, 2, 0) \vdash (q_1, 1, 0) \vdash (q_0, 0, 1) \vdash (q_2, 0, 1) \vdash (q_2, 1, 0) \\
\quad\quad \vdash (q_0, 1, 0) \vdash (q_1, 0, 0) \vdash (f, 0, 0).
\end{array}$$

□

Example 153. The following 4-counter machine recognizes the prime numbers: $M = (Q, 4, \delta, q_0, f)$ where $Q = \{q_0, \ldots, g_7, f\}$ and the transition function δ is

$$\begin{array}{lcl}
\delta(q_0, P, Z, Z, Z) & = & (q_1, -1, +1, 0, 0) \\
\delta(q_1, P, P, Z, Z) & = & (q_2, +1, +1, 0, 0) \\
\delta(q_2, P, *, *, *) & = & (q_2, -1, 0, +1, +1) \\
\delta(q_2, Z, *, *, *) & = & (q_3, 0, 0, 0, 0) \\
\delta(q_3, *, *, *, P) & = & (q_3, +1, 0, 0, -1) \\
\delta(q_3, *, *, *, Z) & = & (q_4, 0, 0, 0, 0) \\
\delta(q_4, *, P, P, *) & = & (q_4, 0, -1, -1, +1)
\end{array}$$

$$\begin{array}{lcl}
\delta(q_4, *, Z, P, *) & = & (q_5, 0, 0, 0, 0) \\
\delta(q_5, *, *, *, P) & = & (q_5, 0, +1, 0, -1) \\
\delta(q_5, *, *, *, Z) & = & (q_4, 0, 0, 0, 0)
\end{array}$$

$$\begin{array}{lcl}
\delta(q_4, *, P, Z, *) & = & (q_6, 0, 0, 0, 0) \\
\delta(q_6, *, *, *, P) & = & (q_6, 0, +1, 0, -1) \\
\delta(q_6, *, *, *, Z) & = & (q_2, 0, +1, 0, 0)
\end{array}$$

$$\begin{array}{lcl}
\delta(q_4, *, Z, Z, *) & = & (q_7, 0, 0, 0, 0) \\
\delta(q_7, P, Z, Z, P) & = & (q_7, -1, 0, 0, -1) \\
\delta(q_7, Z, Z, Z, Z) & = & (f, 0, 0, 0, 0)
\end{array}$$

First, the machine verifies that the input n is at least 2, and it initializes the second counter to value 2:

$$(q_0, n, 0, 0, 0) \vdash (q_1, n-1, 1, 0, 0) \vdash (q_2, n, 2, 0, 0).$$

In state q_2, the value of counter 1 is copied to counter 3:

$$(q_2, n, m, 0, 0) \vdash^* (q_2, 0, m, n, n) \vdash (q_3, 0, m, n, n) \vdash^* (q_3, n, m, n, 0) \vdash (q_4, n, m, n, 0).$$

In state q_4, the content of counter 2 is repeatedly subtracted from counter 3 as many times as possible:

$$(q_4, n, m, k, 0) \vdash^* (q_4, n, 0, k-m, m) \vdash (q_5, n, 0, k-m, m) \vdash^* (q_5, n, m, k-m, 0) \vdash^* (q_4, n, m, k-m, 0).$$

In the last round, when counter 3 becomes zero we have reached some ID $(q_4, n, m-x, 0, x)$. The next action depends on whether counter 2 became zero at the same time:

- If counter 3 becomes zero first, then m does not divide evenly n. The machine increases m by one and repeats everything from state q_2:

  $$(q_4, n, m-x, 0, x) \vdash (q_6, n, m-x, 0, x) \vdash^* (q_6, n, m, 0, 0) \vdash (q_2, n, m+1, 0, 0).$$

- If counters 2 and 3 become zero at the same time then m divides n. Number n is a prime number if and only if $m = n$:

  $$(q_4, n, 0, 0, m) \vdash (q_7, n, 0, 0, m) \vdash^* (q_7, n-m, 0, 0, 0).$$

  This leads to $(f, 0, 0, 0)$ iff $n - m = 0$.

□

Counter machines recognize sets of numbers rather than words. But words can be encoded as numbers (in various ways). The following definition is robust to changes in the encoding used, as long as they remain effectively equivalent: We say that $A \subseteq \mathbb{N}$ is **recursive** (or **recursively enumerable** or **r.e.-complete**) if the language $\{a^n \mid n \in A\}$ is recursive (recursively enumerable or r.e.-complete, respectively). It is clear that all sets recognized by counter machines are recursively enumerable. It turns out that there are counter machines with just two counters that recognize r.e.-complete sets. Two counters are sufficient to simulate any Turing machine.

Theorem 154 *There is a 2-counter machine M that recognizes an r.e.-complete subset of* $\mathbb{N}$.

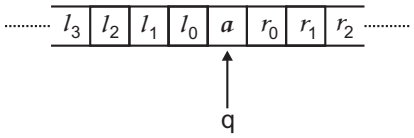
Proof sketch. Due to lack of time we only sketch the proof. The goal is to simulate an arbitrary Turing machine with a counter machine. We first sketch the simulation using three counters, starting with a Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, B, f)$ that recognizes an r.e.-complete language.

1. Any Turing machine can be simulated by a TM with two tape symbols 0 and 1, as shown in the proof of Lemma 144. The original tape alphabet Γ is encoded as fixed length binary words, making sure that the blank symbol B gets mapped into the word $00\ldots0$. We obtain a TM with tape alphabet $\{0,1\}$ such that the question "Does a given ID evolve into an accepting ID ?" is r.e.-complete.

2. Now we assume that the tape alphabet of the TM is $\{0,1\}$. We encode any ID of the TM using one state and two natural numbers n and m. Suppose the tape contains the symbols $\ldots l_2 l_1 l_0$ and $r_0 r_1 r_2 \ldots$ to the left and to the right of the read/write head. These two sequences are viewed as bits in the binary expansion of integers

$$\begin{aligned} n &= l_0 + 2l_1 + 2^2 l_2 + \ldots, \text{ and} \\ m &= r_0 + 2r_1 + 2^2 r_2 + \ldots. \end{aligned}$$

Note that the sums are finite because there are only finitely many non-zero symbols on the tape. Let a be the tape letter currently scanned and let q be the current state:

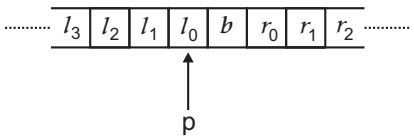


This situation of the TM is encoded as the ID

$$((q,a), n, m, 0)$$

of the simulating 3-counter machine. So two counters represent the left and the right halves of the tape, represented in binary. The third counter is an auxiliary counter used in the simulation. The state (q,a) of the counter machine records the state q of the simulated TM and the currently scanned tape symbol a. If $q = f$ then the counter machine enters its final state and accepts. Otherwise, the pair (q,a) uniquely determines the next move of the TM.

Suppose $\delta(q,a) = (p,b,L)$. (The case of a right move is analogous.) The next Turing machine ID is



In the counter machine, the move of the read/write-head to the left corresponds to dividing and multiplying by two the numbers n and m, respectively. The least significant bit l_0 of n is the new tape symbol to be scanned, and the newly written tape symbol b should be written as the least least significant bit of the new value m. So, the new ID after the move should be

$$((p, l_0), n', m', 0)$$

where $m' = 2m + b$ and $n = 2n' + l_0$. Such an ID is obtained through the following sequence of moves:

(a) Update $m \longrightarrow 2m + b$: repeatedly decrement counter 2 once and increment counter 3 twice, until counter 2 becomes zero. Then, repeatedly decrement counter 3 and increment counter 2, until counter 3 is zero. Now counter 2 contains value $2m$. If $b = 1$, increment counter 2 once more.

(b) Update $n \longrightarrow \lfloor n/2 \rfloor$ and read the least significant bit l_0 of n: Repeatedly decrement counter 1 twice and increment counter 3 once, until counter 1 is zero ($l_0 = 0$) or one ($l_0 = 1$). The symbol l_0 can be read from counter 1, after which we can copy (by decrementing and incrementing counters 3 and 1, respectively) the value $\lfloor n/2 \rfloor$ from counter 3 back to counter 1.

It is clear that the construction above works. Each move by the TM is simulated through a sequence of moves in the 3-counter machine.

3. The final task is to simulate a 3-counter machine by a 2-counter machine. An arbitrary 3-counter ID (q, n, m, k) is represented as the 2-counter ID $(q, j, 0)$ where $j = 2^n 3^m 5^k$. So all three counters are expressed in counter 1 as powers of three co-prime numbers 2, 3 and 5, while counter 2 is used as an auxiliary storage. To simulate one move of the 3-counter machine the new machine has to check which of the three counter values n, m and k are zero. Note that $n > 0$ iff j is even, $m > 0$ iff 3 divides j, and $k > 0$ iff 5 divides j. These three conditions are first checked by repeatedly moving a token from counter 1 to counter 2, and counting modulo 2,3, and 5 the number of tokens moved. Once counter 1 is zero, one checks whether the number was divisible by 2,3 or 5. From this, the next move to be simulated can be inferred.

Each instruction may increment of decrement the counters. Each increment or decrement corresponds to multiplying or dividing j by 2,3, or 5. These can be established, one-by-one, by repeatedly decrementing and incrementing counters 1 and 2, respectively, 2, 3 or 5 times. For example, to divide counter 1 by 3, we repeatedly decrement counter 1 three times and increment counter 2 once, until counter 1 is zero. Then the value $j/3$ can be returned from counter 2 back to counter 1. □

Remarks. The simulation of a Turing machine by a 3-counter machine presented above is quite efficient: The time to simulate one move of the TM is proportional to the binary numbers represented by the two halves of the tape. In contrast, the simulation by a 2-counter machine is exponentially slower.

4.12.2 Fractran

J.Conway's **Fractran** is a fun variant of counter machines. Fractran programs consist of finite lists

$$\frac{n_1}{m_1}, \quad \frac{n_2}{m_2}, \quad \frac{n_3}{m_3}, \quad \ldots, \quad \frac{n_k}{m_k}$$

of rational numbers. Let $n \in \mathbb{N}$ be any natural number. In one step, the number gets multiplied by the first fraction $\frac{n_i}{m_i}$ in the list such that $n\frac{n_i}{m_i}$ is an integer. If no such fraction exists then the system halts.

Example 155. Consider the following Fractran program:

$$\frac{3}{11}, \quad \frac{847}{45}, \quad \frac{143}{6}, \quad \frac{7}{3}, \quad \frac{10}{91}, \quad \frac{3}{7}, \quad \frac{36}{325}, \quad \frac{1}{2}, \quad \frac{36}{5}$$

Starting from $n = 10$, the first multiplication is by $\frac{1}{2}$, giving 5. This is then multiplied by $\frac{36}{5}$, giving 36. Continuing likewise, we get the iteration

$$10 \xrightarrow{1/2} 5 \xrightarrow{36/5} 36 \xrightarrow{143/6} 858 \xrightarrow{3/11} 234 \xrightarrow{143/6} 5577 \xrightarrow{3/11} 1521 \xrightarrow{7/3} 3549 \xrightarrow{7/3} 8281 \xrightarrow{10/91} 910 \xrightarrow{10/91} 100 \longrightarrow \ldots$$

36 steps later we get 1000, then several steps later $10^5, 10^7, 10^{11}, \ldots$. It turns out that the powers of 10 that are generated are exactly the prime powers, in the correct order. □

It is relatively easy to simulate any counter machine using Fractran. The idea is the same that we used when converting a 3-counter machine into a 2-counter machine: We associate separate prime numbers to different counters, and the counter values are represented as powers of those primes. Yet one more prime is used to represent the state of the counter machine. More precisely, to simulate a 2-counter machine with state set $Q = \{s_0, s_1, \ldots, s_k\}$, we choose 6 distinct prime numbers p, q, r and p', q', r'. We assume that $s_0 = f$ is the final state, and that the counter machine halts if and only if its state is s_0. (To have this condition satisfied we can simply add a dummy state that is entered from all halting ID's that are not accepting.)

The ID (s_i, n, m) of the 2-counter machine is represented by the natural number $r^i p^n q^m$. Four Fractran instructions are associated to each non-final state s_i: These have denominators $r^i pq$, $r^i p$, $r^i q$, and r^i, in this order, corresponding to transitions $\delta(s_i, P, P), \delta(s_i, P, Z), \delta(s_i, Z, P)$ and $\delta(s_i, Z, Z)$. To make notations simpler, let us use the "inverse" sign-function and define

$$\begin{aligned} \alpha(Z) &= 0, \\ \alpha(P) &= 1. \end{aligned}$$

Now we can write that the denominator of the Fractran-instruction corresponding to $\delta(s_i, x, y)$ is

$$r^i p^{\alpha(x)} q^{\alpha(y)}.$$

The instructions for different non-final states s_i are ordered from the largest $i = k$ to the smallest $i = 1$. So we get $4k$ instructions whose denominators are

$$r^k pq, \quad r^k p, \quad r^k q, \quad r^k; \qquad r^{k-1} pq, \quad r^{k-1} p, \quad r^{k-1} q, \quad r^{k-1}; \qquad \ldots \qquad rpq, \quad rp, \quad rq, \quad r.$$

It is easy to see that the first denominator in this list that divides $r^i p^n q^m$ is the one that corresponds to transition $\delta(s_i, \text{sign}(n), \text{sign}(m))$.

The numerators are designed so that they give the result of the transition, but we have to use marked versions of the primes because otherwise identical primes would be canceled in the fraction. The fraction that corresponds to an arbitrary transition $\delta(s_i, x, y) = (s_j, d, e)$ is

$$\frac{r'^j \; p'^{\alpha(x)+d} \; q'^{\alpha(y)+e}}{r^i \; p^{\alpha(x)} \; q^{\alpha(y)}}.$$

An application of this instruction changes the state into (the marked version) r'^j and changes the number of p's and q's as indicated by the values d and e in the counter machine instruction.

Finally, as the first three instructions of the Fractran program we set

$$\frac{p}{p'}, \frac{q}{q'}, \frac{r}{r'}.$$

These make sure that the marked versions of the primes get immediately replaced by the unmarked primes. We end up with $3 + 4k$ Fractran instructions that simulate each move of the 2-counter machine using up to $k + 5$ Fractran steps, depending on how many marked primes need to be exchanged into non-marked primes.

Based on the construction above and Theorem 154 we obtain the following:

Theorem 156 *There is a Fractran program such that the question of whether the iteration starting from a given number n eventually halts is r.e.-complete.* □

Remark: The simulation of a k-counter machine by Fractran is analogous, for any k. One just uses two additional primes p_c and p'_c for each counter c. Because 3-counter machines admit an efficient simulation of Turing machine, we also obtain an analogous efficient simulation by Fractran.

Example 157. Consider the 2-counter machine $M = (\{s_0, s_1, s_2\}, 2, \delta, s_1, s_0)$ with transitions

$$\begin{array}{lcllcl} \delta(s_2, P, *) & = & (s_1, -1, +1) & \delta(s_1, P, *) & = & (s_2, -1, 0) \\ \delta(s_2, Z, *) & = & (s_2, 0, 0) & \delta(s_1, Z, *) & = & (s_0, 0, 0) \end{array}$$

Let us choose the primes $r = 2, r' = 3, p = 5, p' = 7, q = 11$ and $q' = 13$. The corresponding Fractran program (as provided by the construction above) is

$$\frac{2}{3}, \quad \frac{5}{7}, \quad \frac{11}{13}, \quad \frac{3 \cdot 13^2}{2^2 \cdot 5 \cdot 11}, \quad \frac{3 \cdot 13}{2^2 \cdot 5}, \quad \frac{3^2 \cdot 13}{2^2 \cdot 11}, \quad \frac{3^2}{2^2}, \quad \frac{3^2 \cdot 13}{2 \cdot 5 \cdot 11}, \quad \frac{3^2}{2 \cdot 5}, \quad \frac{13}{2 \cdot 11}, \quad \frac{1}{2}.$$

The five step computation from the initial ID $(s_1, 4, 0)$ into $(s_0, 0, 2)$ in the counter machine corresponds to the Fractran computation

$$\begin{array}{l} 2 \cdot 5^4 \longrightarrow 3^2 \cdot 5^3 \longrightarrow 2 \cdot 3 \cdot 5^3 \longrightarrow \\ \quad 2^2 \cdot 5^3 \longrightarrow 3 \cdot 13 \cdot 5^2 \longrightarrow 2 \cdot 13 \cdot 5^2 \longrightarrow \\ \quad\quad 2 \cdot 11 \cdot 5^2 \longrightarrow 3^2 \cdot 13 \cdot 5 \longrightarrow 2 \cdot 3 \cdot 13 \cdot 5 \longrightarrow 2^2 \cdot 13 \cdot 5 \longrightarrow \\ \quad\quad\quad 2^2 \cdot 11 \cdot 5 \longrightarrow 3 \cdot 13^2 \longrightarrow 2 \cdot 13^2 \longrightarrow 2 \cdot 11 \cdot 13 \longrightarrow \\ \quad\quad\quad\quad 2 \cdot 11^2 \longrightarrow 11 \cdot 13 \longrightarrow \\ \quad\quad\quad\quad\quad 11^2 \end{array}$$

□

4.12.3 Tag systems

Post's tag systems are yet another class of simple systems capable of simulating arbitrary Turing machines. A **tag system** is a triple $T = (\Sigma, k, g)$ where Σ is a finite alphabet, $k \geq 1$ is the **deletion number** and $g : \Sigma \longrightarrow \Sigma^*$ is a function assigning a word to each letter in Σ. Such system is called a k-tag system. This is a deterministic rewrite system. Any word $u \in \Sigma^*$ whose length is at least k gets rewritten as follows: Let $u = avw$ where $a \in \Sigma$ and $|av| = k$. Then

$$u \Rightarrow wg(a).$$

In other words, the first k letters of u are erased and the word $g(a)$ is appended to the end where a is the first letter of u. The rewriting terminates if a word is reached whose length is less than k.

Example 158. Consider the 3-tag system $(\{0,1\},3,g)$ where $g(0)=00$ and $g(1)=1101$. This is a system studied (and found intractable) by E.Post. For example, starting from word 10010 we obtain

$$
\begin{aligned}
&10010 \Rightarrow \\
&\quad 101101 \Rightarrow \\
&\quad\quad 1011101 \Rightarrow \\
&\quad\quad\quad 11011101 \Rightarrow \\
&\quad\quad\quad\quad 111011101 \Rightarrow \\
&\quad\quad\quad\quad\quad 0111011101 \Rightarrow \\
&\quad\quad\quad\quad\quad\quad 101110100 \Rightarrow \\
&\quad\quad\quad\quad\quad\quad\quad 1101001101 \Rightarrow \\
&\quad\quad\quad\quad\quad\quad\quad\quad 10011011101 \Rightarrow \\
&\quad\quad\quad\quad\quad\quad\quad\quad\quad 110111011101 \Rightarrow \\
&\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad 1110111011101 \Rightarrow \\
&\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad 01110111011101 \Rightarrow \\
&\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad 1011101110100 \Rightarrow \\
&\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad 11011101001101 \Rightarrow \\
&\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad 111010011011101 \Rightarrow \\
&\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad 0100110111011101 \Rightarrow \\
&\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad 011011101110100 \Rightarrow \\
&\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad 01110111010000 \Rightarrow \\
&\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad 1011101000000 \Rightarrow \\
&\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad 11010000001101 \Rightarrow \\
&\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad 100000011011101 \Rightarrow \\
&\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad 0000110111011101 \Rightarrow \\
&\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad 011011101110100
\end{aligned}
$$

The last word in the sequence already appeared 6 steps before, so we enter a loop of length 6. On the other hand, starting from 100100100000 one reaches in 419 steps the halting word 00. □

Let us denote by $L(T)$ the set of words that eventually halt (i.e., evolve into a word of length less than k) in the k-tag system T. Clearly $L(T)$ is recursively enumerable. Without a proof we state the following result:

Theorem 159 *There is a 2-tag system T such that the language $L(T)$ is r.e.-complete.* □

Remarks: (1) Sometimes tag systems are defined slightly differently, as follows: the successor of a non-empty word u is obtained by first appending $g(a)$ to the end, corresponding to the first letter a of u. If the result $ug(a)$ is shorter than the deletion number k the process halts; otherwise the first k letters of $ug(a)$ are erased. In this way, even words shorter than k may have a successor.

(2) Another natural decision problem associated to a tag-system is the **word problem**, asking whether a given word x derives another given word y. Clearly, the halting problem of any tag-system can be Turing reduced to its word problem.

(3) The tag system of Example 158 is still a mystery. It is not known whether its word problem is decidable or not. It is not even known whether there exists any word that neither eventually halts nor enters a cycle. (If one these always happens then the word problem is of course decidable.)