1 Basic Notions

1.1 Basic Definitions

A **formal language** is a set of strings over finite sets of symbols. We have different ways of describing such languages:

- Finite automata
- Regular expressions
- Grammars
- Turing machines

Example: The set of all strings of tokens generated by the grammar that describes the Java programming language is a formal language.

An **alphabet** is a finite set of symbols such as letters, digits, etc. Greek letters are (Σ, Γ, etc.) are usually used to denote alphabets.

Examples:

- $\Sigma_1 = \{a, b, c\}$,
- $\Sigma_2 = \{a, b, c, \dots, z\}$,
- $\Gamma = \{a, b, c, \$, \square\}$

A **word** over an alphabet Σ is a string of finite length, in other words a finite sequence of symbols of Σ.

A (formal) **language** over an alphabet Σ is defined as a set of words defined over Σ. The letters w, u and v are usually used to denote words. The letters s, t and r are usually used for single letters. Languages are usually denoted by capital letters.

Examples of languages:

- $L_1 = \{a, b, aab, abbcc\}$
- $L_2 = \{acbb, accbb, acccbb, accccbb, \dots\}$

These languages are defined over the alphabet $\Sigma = \{a, b, c\}$.

- L_1 is finite.
- L_2 is composed of all the words starting with the symbol a, followed by at least one symbol c and ending with bb.

From now on, only finite alphabets are considered. A language can be infinite, but each of its elements are always finite.

1.2 Operations on Words

The **concatenation** operation of two words consists of appending the second word to the end of the first one. Concatenation of two words $w_1 = s_1 s_2 \dots s_m$ and $w_2 = t_1 t_2 \dots t_n$ is defined by

$$w_1 \cdot w_2 = s_1 \dots s_m t_1 \dots t_n.$$

The concatenation operation is associative, but not commutative.

The operation that returns the **length**, i.e. the number of symbols of a word w is denoted by $|w|$.

ϵ denotes a special word called the **empty string** (also denoted by λ). Some properties of ϵ:

- $|\epsilon| = 0$
- For any word w, we have $w \cdot \epsilon = \epsilon \cdot w = w$.

The **mirror** operation associates the reverse of a word $w = s_1 s_2 \dots s_n$, which is denoted by $w^R = s_n \dots s_2 s_1$.

Recursive definition:

1. $\epsilon^R = \epsilon$
2. $s^R = s$, for any symbol s
3. $w^R = v^R \cdot u^R$, for any word $w = u \cdot v$

If $w = w^R$ then w is called a **palindrome**, i.e. a word that reads the same forwards and backwards.

The **concatenation** of the word w with itself n times is denoted by w^n where $n \geq 0$.

Recursive definition:

1. $w^0 = \epsilon$
2. $w^{n+1} = w^n w$

The **set of all words** of length greater or equal to 1 over an alphabet Σ is denoted by Σ^+.

Example:

$$\Sigma^+ = \{a, b, aa, ab, ba, bb, aaa, aab,$$
$$aba, abb, baa, bab, \dots\}, \text{ where } \Sigma = \{a, b\}$$

The **set of all words** of length greater or equal to 0 over an alphabet Σ is denoted by Σ^*. Some remarks:

- Σ^+ and Σ^* are infinite if and only if $\Sigma \neq \emptyset$
- $\Sigma^* = \Sigma^+ \cup \{\epsilon\}$

Definition of Σ^+:

1. any element of Σ belongs to Σ^+
2. for all $a \in \Sigma$ and for all $w \in \Sigma^+$, we have $aw \in \Sigma^+$

1.3 Operations on Languages

Operations on languages L_1, L_2 and L are defined over an alphabet Σ:

- **Union:** $L_1 \cup L_2$
- **Intersection:** $L_1 \cap L_2$
- **Complement:** $\overline{L} = \Sigma^* \backslash L$

Further operations include:

- **Concatenation:** $L_1 \cdot L_2 = \{w_1 \cdot w_2 | w_1 \in L_1 \text{ and } w_2 \in L_2\}$
- **Neutral element:** $L_\epsilon = \{\epsilon\}$
- **Absorbance element:** $\emptyset$

For the neutral element holds

$$L \cdot L_\epsilon = L_\epsilon \cdot L = L.$$

For the absorbance element holds

$$L \cdot \emptyset = \emptyset \cdot L = \emptyset.$$

Note that $L_\epsilon \neq \emptyset$.

Example: let $\Sigma = \{a, b\}, L_1 = \{a, b\}, L_2 = \{bac, b, a\}$.

$$L_1 \cdot L_2 = \{abac, ab, aa, bbac, bb, ba\}$$

The **concatenation** of the language L with itself n times is denoted by L^n where $n \geq 0$.

Recursive definition:

- $L^0 = L_\epsilon = \{\epsilon\}$
- $L^{n+1} = L^n \cdot L$

The **iterative closure** or **Kleene's closure** of a language L, denoted as L^*, is the set of words resulting from the concatenation of a finite number of words of L. Formally

$$L^* = L^0 \cup L^1 \cup L^2 \cup L^3 \cup \dots = \bigcup_{i \geq 0} L_i$$

A variation of Kleene's closure called **Kleene plus** is defined as

$$L^+ = L^1 \cup L^2 \cup L^3 \cup \dots = \bigcup_{i \geq 1} L_i$$
$$= L \cdot L^* = L^* \cdot L$$

Example: let $L = \{aa, bb, c\}$

$$L^* = \{\epsilon, aa, bb, c, aaaa, aabb, aac, bbaa, bbbb,$$
$$bbc, caa, cbb, cc \dots\}$$

2 Regular Languages

2.1 Introduction

How do we specify i.e. describe languages? **Finite languages** Since a language is a set of words, any finite set can be exhaustively enumerated. **Infinite languages** An exhaustive enumeration is not possible (in a finite time). Consequently, more general concepts are required.

Regular languages are the simplest class of languages. Each one of these formalisms define this class of languages:

- **Finite state automata**
- **Regular grammars**
- **Regular expressions**

They all have the same expression power, i.e. they all describe the same things.

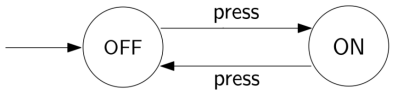
2.2 Finite State Automata

A **finite state automaton (FSA)** consists of the following three components:

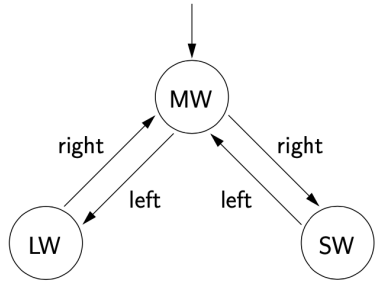
- A **finite set of states** (drawn as circles).
- A **transition function** describing the actions which allow the transition from one state to another (drawn as arrows).
- The **initial state** i.e. the state in which the automaton is in initially (indicated by an arrow coming form nowhere pointing at it).

First, we will consider **deterministic** finite state automata (DFA) and then **non-deterministic** finite state automata (NFA). Both DFA and NFA are FSA.

Example: A switch changing alternately between two states "ON" and "OFF" can be modelled as follows:



Example: Another example is that of a circular selector for radio channels LW, MW and SW. This selector can be turned in any directions from LW to SW passing by MW, but not directly from LW to SW. Such a selector can be modeled as follows:



2.2.1 Deterministic Finite State Automata

Deterministic finite state automata can be formally defined as being a 4-tuple (Q, Σ, δ, q_0), where:

- Q is a finite set of states
- Σ is a finite set of symbols, i.e. an alphabet
- $\delta : Q \times \Sigma \to Q$ is a transition function
- $q_0 \in Q$ is the initial state

The transition function $\delta : Q \times \Sigma \to Q$ expresses which symbol moves the automaton from one state to the next one. Such functions represent the transition with their label. Automata with an infinite number of states do not share the same properties with automata with finite states.

The extension of δ is the function $\delta^* : Q \times \Sigma^* \to Q$ which provides the next state for a sequence of symbols and can be defined as follows: For a state $q \in Q$, a word $w \in \Sigma^*$ and a symbol $s \in \Sigma$, the function $\delta^* : Q \times \Sigma^* \to Q$ is recursively defined as follows:

1. $\delta^*(q, \epsilon) = q$
2. $\delta^*(q, ws) = \delta(\delta^*(q, w), s)$

Example: Using the previous example with the radio, the transition function of the automaton is as follows:

$$\delta(\text{MW, right}) = \text{SW}$$
$$\delta(\text{SW, left}) = \text{MW}$$
$$\delta(\text{MW, left}) = \text{LW}$$
$$\delta(\text{LW, right}) = \text{MW}$$

Further, the following three uses of δ^* are correct:

$$\delta(\text{MW, left} \cdot \text{right}) = \text{MW}$$
$$\delta(\text{LW, right} \cdot \text{right} \cdot \text{left}) = \text{MW}$$
$$\delta(\text{MW, right}) = \text{SW}$$

A **language recognizer** is a FSA that answers whether a given word belongs to the language described by an automaton. An input word is **recognized** or **accepted** by a given recognizer if and only if a final state is reached. The set of all recognized words by a given recognizer forms the language modeled by the automaton and is called the **recognized language** by the automaton.

Formally defined, a DFA language recognizer is a DFA equipped with a set of final states. Final states are drawn as double circles. It is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where

- Q is a finite set of states
- Σ is a finite set of symbols, i.e. an alphabet
- $\delta : Q \times \Sigma \to Q$ is a transition function
- $q_0 \in Q$ is the initial state
- $F \subseteq Q$ is a set of final states

The difference between a DFA and a DFA recognizer lies only in the additional final states, which are included in Q.

Informally, a word is recognized by a DFA if and only if there exists one path from the initial state to one of the final states. Formally, however the following definition is used: A word $w \in \Sigma^*$ is recognized by a DFA $M = (Q, \Sigma, \delta, q_0, F)$ iff $\delta^*(q_0, w) \in F$.

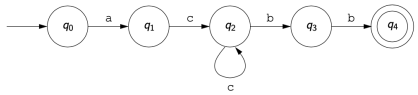
The language recognized (or accepted) by a DFA M is denoted $L(M)$. This is the set of all the words recognized by M.

Finally, we can define regular languages as follows: Every language recognized by a finite state recognizer automaton is called a **regular language**.

Example: The language

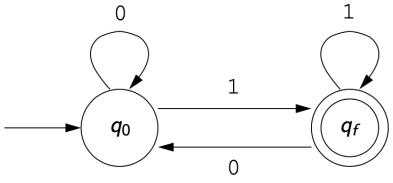
$$L_2 = \{acbb, accbb, acccbb, accccbb, \dots\}$$

is recognized by the following automaton:



Example: Let A be the DFA recognizer formally defined as

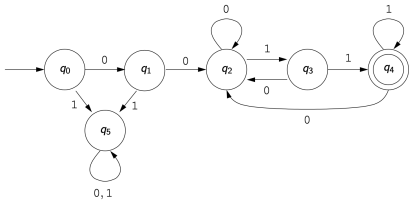
$$A = (Q, \Sigma, \delta, q_0, F)$$
$$Q = q_0, q_f$$
$$\Sigma = \{0, 1\}$$
$$F = q_f$$
$$\delta(q_0, 0) = q_0$$
$$\delta(q_0, 1) = q_f$$
$$\delta(q_f, 0) = q_0$$
$$\delta(q_f, 1) = q_f$$



If we consider that the words over the alphabet $\Sigma = \{0, 1\}$ denote the binary numbers, the recognized language by A is

$$L(A) = \{x \mid x \in \{0,1\}^+ \text{ and } x \text{ is odd}\}$$

Example: Let B be the DFA recognizer graphically represented by:



The language recognized by B is then

$$L(B) = \{0^n x 1^m \mid n \geq 2, m \geq 2, x \in \{0,1\}^*\}$$

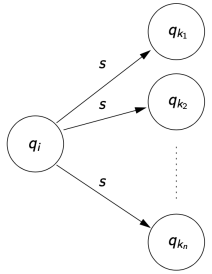
A function is called **totally defined** if it is defined for all possible inputs. In finite state automata, δ is called totally defined if it is defined for all states and symbols.

2.2.2 Non-deterministic Finite State Automata

Up until now, we have seen DFA with the property that form one state, another state can be reached in an unique manner by δ. Non-deterministic finite state automata allow multiple transitions from a given state and a given symbol. Finite state automata in general can be defined as the union of DFA and NFA.

NFA and DFA have the same expression power. Everything that can be represented as a DFA can also be represented as a NFA and vice versa. The usage of non-determinism makes the automata smaller and easier to understand.

Graphically, a non-deterministic choice is represented by several arrows leaving a state, with each arrow labeled with the same symbol.



The co-domain of the transition function is the power set of the set of states Q:

$$\delta : Q \times \Sigma \to \mathcal{P}(Q)$$

where $\mathcal{P}(Q)$ is the power set of Q.
Formally defined, a non-deterministic automaton is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where

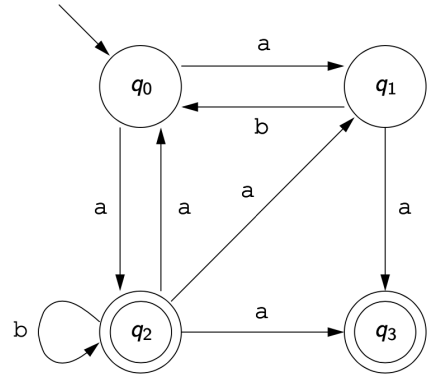
- Q is a finite set of states
- Σ is a finite set of symbols, i.e. an alphabet
- $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ is a transition function
- $q_0 \in Q$ is the initial state
- $F \subseteq Q$ is a set of final states

Similarly to DFA, the extension function δ^* of δ finds the next state for a sequence of symbols and can be defined as follows: For a state $q \in Q$, a word $w \in \Sigma^*$ and a symbol $s \in \Sigma$, the function $\delta^* : Q \times \Sigma^* \to \mathcal{P}(Q)$ is recursively defined as follows:

1. $\delta^*(q, \epsilon) = \{q\}$
2. $\delta^*(q, ws) = \bigcup_{q' \in \delta^*(q,w)} \delta(q', s)$

Similarly to above, there exist NFA language recognizers. A word is recognized by a NFA if and only there exists at least one path from the initial state to one of the final states. More formally: A word $w \in \Sigma^*$ is recognized by a NFA $M = (Q, \Sigma, \delta, q_0, F)$ if and only if $\delta^*(q_0, w) \cap F \neq \emptyset$. The language recognized by a NFA consists of the set of all words recognized by the automaton.

Example: The following NFA accepts the word *abaa* since there exists at least one path from q_0 to q_3:



2.2.3 Equivalence of NFA and DFA

We are going to prove shortly that for any NFA, there exists a DFA which is equivalent, i.e. if we consider the automata as black boxes, we are not able to distinguish them because they recognize the same language. More formally, two finite state automata M_1 and M_2 are equivalent if and only if they recognize the same language, i.e. $L(M_1) = L(M_2)$.

The theorem states: *For any nondeterministic finite state automaton NFA there exists a deterministic finite state automaton DFA, which is equivalent, and vice versa.* From this theorem follows:

- Non-determinism does not provide more expression power than determinism.
- NFA are generally more compact than equivalent DFA.

The first step of the proof is trivial, since any DFA is per definition a NFA (a deterministic FSA is a particular case of a non-deterministic FSA). The second step is not as easy as the first one. In order to show that for any NFA there exists a DFA we are going to construct a DFA from the NFA and prove that both accept the same language.

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a NFA. We construct a DFA $M' = (Q', \Sigma, \delta', q_0', F')$ as follows:

1. $Q' = \mathcal{P}(Q)$. An element of Q' will be denoted $[q_1, q_2, \ldots, q_i]$ where $q_1, \ldots, q_i$ belong to Q. This element represents a single state of the deterministic automaton corresponding to a set of states of the NFA.
2. F' is the set of members of Q' composed of at least one final state of M.
3. $q_0' = [q_0]$
4. $\delta'([q_1, \ldots, q_i], s) = [p_1, \ldots, p_j] \iff \delta(\{q_1, \ldots, q_i\}, s) = \{p_1, \ldots, p_j\}$.

Now we have to show that every word w accepted by M is also accepted by M' and if M does not accept the word w, then M' rejects w.

We prove by induction on the length of w that

$$\delta'^*(q_0', w) = [q_1, \ldots, q_i]$$
$$\iff \delta^*(\{q_0, w\}, s) = \{q_1, \ldots, q_i\}$$

Base case: Trivial for $|w| = 0$ because $q_0' = [q_0]$ and $w = \epsilon$.
Induction: Let us assume that the above is true for some words of length $\leq m$. Let $s \in \Sigma$ and ws be a string of length $m + 1$. By definition of δ'^*,

$$\delta'^*(q_0', ws) = \delta'(\delta'^*(q_0', w), s).$$

By induction hypothesis,

$$\delta'^*(q_0', w) = [p_1, \ldots, p_j] \iff \delta^*(q_0, w) = \{p_1, \ldots, p_j\}.$$

But by definition of δ',

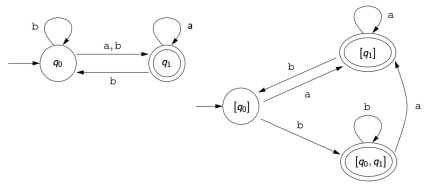
$$\delta'([p_1, \ldots, p_j], s) = [r_1, \ldots, r_k]$$
$$\iff \delta(\{p_1, \ldots, p_j\}, s) = \{r_1, \ldots, r_k\}.$$

Consequently,

$$\delta'^*(q_0', ws) = [r_1, \ldots, r_k]$$
$$\iff \delta^*(q_0, ws) = \{r_1, \ldots, r_k\}.$$

Finally, we only have to add the following $\delta'^*(q_0', w) \in F'$ only when $\delta^*(q_0, w)$ contains a state of Q that belongs to F. Thus, $L(M) = L(M')$.

Example: Here is a non-deterministic automaton. We eliminate the nondeterminism according to the method given in the proof of the theorem:



2.2.4 Automata with ϵ-transitions

Up to now, we have seen FSA where the label of the transition functions belong to the alphabet. We can extend this so that we allow a transition to be caused by the empty string ϵ. Such transitions are called **ϵ-transitions**, **empty transitions** or **spontaneous transitions**. Empty transitions make the design and understanding of NFA easier.

A finite state automaton with ϵ-transitions is defined in the same manner as a classic non-deterministic automata except that the symbol ϵ is in the alphabet.

The concept of **ϵ-closures** can be formally defined as follwing: For a state q in Q of an automaton M, the ϵ-closure(q) is the set of all states of M reachable from q by a sequence of empty transitions:

$$\epsilon\text{-closure}(q) = \{p \in Q | (q, w) \vdash_m^* (p, w)\}$$
$$\epsilon\text{-closure}(P) = \bigcup_{p \in P} \epsilon\text{-closure}(q)$$

where P denotes a set of states.

For a given NFA with empty transitions, we define a new transition function δ^* that uses the ϵ-closure. The new transition function is defined as follows:

1. $\delta(q, \epsilon) = \epsilon\text{-closure}(q)$
2. For every $w \in \Sigma^*$ and $s \in \Sigma$:
$$\delta^*(q, ws) = \epsilon\text{-closure}(P)$$
where $P = \{p | p \in \delta r, s \text{ for an } r \in \delta^*(q, w)\}$

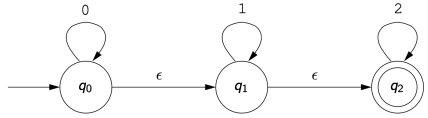
We extend δ and δ^* for a set of states:

$$\delta(R, s) = \bigcup_{q \in R} \delta(q, s)$$
$$\delta^*(R, w) = \bigcup_{q \in R} \delta^*(q, w)$$

The introduction of ϵ-transitions does not provide any extra expression power. This means that for any NFA with ϵ-transitions, there exists a NFA without ϵ-transitions.

The removal method of empty transitions of a non-deterministic automaton consists of computing the ϵ-closure for all states of the automaton, which defines the transition function δ^*. Once the operation is performed, it is necessary to add the initial state to the set of final states when the empty string is recognized by the automaton with empty transitions.

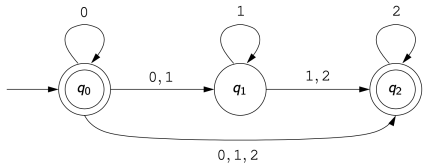
Example: Here is a non-deterministic automaton with some empty transitions:



The computation of the ϵ-closure provides the new transition function δ:

δ	0	1	2
q_0	$\{q_0, q_1, q_2\}$	$\{q_1, q_2\}$	$\{q_2\}$
q_1	$\emptyset$	$\{q_1, q_2\}$	$\{q_2\}$
q_2	$\emptyset$	$\emptyset$	$\{q_2\}$

All that remains now is to add q_0 to the set of final states since the empty string is recognized by the above automaton:



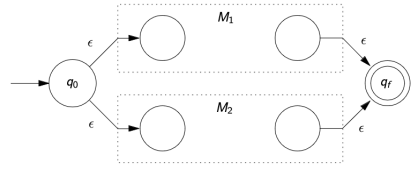
Any language L recognized by a NFA with ϵ-tranistions is also recognized by a NFA without ϵ-transitions.

2.2.5 Minimization of DFA

2.3 Properties of Regular Languages

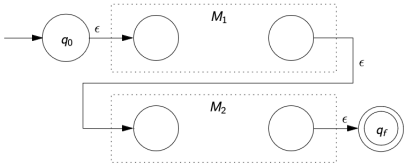
Regular languages have a few properties which are the following:

1. The union $L_1 \cup L_2$ of two regular languages L_1 and L_2 is a regular language. Let M_1 and M_2 be two FSA such that $L(M_1) = L_1$ and $L(M_2) = L_2$. The union of both the languages recognized by the automata is a regular language. Indeed, we can construct an automaton that recognized L_1 and L_2 as follows:



If $w \in L_1$ (recognized by M_1) or $w \in L_2$ (recognized by M_2), then $w \in L_1 \cup L_2$ (recognized by the above FSA).

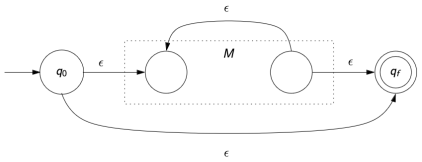
2. The concatenation $L_1 \cdot L_2$ of two regular languages L_1 and L_2 is a regular language.



If $w_1 \in L_1$ and $w_2 \in L_2$, then the concatenation $w_1 \cdot w_2 \in L_1 \cdot L_2$ (recognized by the above FSA).

3. The complementation $\overline{L} = \Sigma^* \backslash L$ of a regular language L is a regular language as well. To construct an automaton accepting $\overline{L}$, it suffices to transform every final state into a non-final state and every non-final state into a final state. The transition function must be total.

4. The iterative closure L^* of a language L is a regular language.



5. The intersection $L_1 \cap L_2$ of two regular languages L_1 and L_2 is a regular language. Indeed, because

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

Previously, we saw that the complement of a regular language is regular (both $\overline{L_1}$ and $\overline{L_2}$ are regular) and the union of two regular languages is also regular as well as the last complementation. It follows that the intersection of two regular languages is regular.

2.4 Grammars

2.4.1 Introduction

Just like the automata, a **grammar** is a formalism used for specifying languages. Each grammar corresponds to a type of machine or automaton:

- Regular grammars ≡ finite state automata
- Context-free grammars ≡ pushdown automata
- Grammars without restrictions ≡ Turing machines

Intuitively, a grammar is a system of rules which allows to "rewrite" a term (or an expression) into another term. Although the grammars and the FSA aare two similar formalisms, there is a difference form an operational standpoint:

- An automaton recognizes a language, i.e. consumes input symbols and determines whether a word belongs to a language.
- A grammar on the other hand generates words.

Grammars are sometimes called rewriting systems and rules are called rewriting rules or productions.

A **rule** is composed of a **left-hand side** α and a **right-hand side** β surrounding an arrow:

$$\alpha \to \beta$$

Both sides are composed of two types of symbols:

- The **terminal** symbols that compose the words generated by the grammar.
- The **non-terminal** symbols that are mainly used in the rewriting process.

The rewriting of a term into another consists of applying a rule to the original term, i.e. to replace the left-hand side which appears in the term with the right-hand side of the rule in order to obtain a new term. Starting the rewriting process from a special rule called the **axiom** and applying repetitively the rules, we obtain (not always) a term composed of a terminal symbol only (no more rules can be applied). Such a term is a word generated by the grammar.

The set of words generated by the rewriting process is called the language generated by the grammar. The rewriting process can be endless and corresponds to a computer program which does not terminate. Since any rules can be selected during the rewriting process, the rewriting process is potentially non-deterministic.

The formal definition of a grammar is as follows: A grammar is a 4-tuple $G = (V_T, V_N, P, S)$, where

- V_T is a set of terminal symbols
- V_N is a set of non terminal symbols such that $V_T \cap V_N = \emptyset$
- P is a set of rules $\alpha \to \beta$, where α and β are sequences of terminals or non-terminals, but α must contain at least non-terminal
- $S \in V_N$ is the axiom or the initial symbol

Let $G = G = (V_T, V_N, P, S)$ be a grammar and $A \to B$ a production of P. The sequence of terminals and non-terminals $\alpha A \beta \in (V_T \cup V_N)^*$ can be derived into a sequence $\alpha B \beta$ by applying the production $A \to B$, i.e. by substituting A for B. We call the move from a sequence α to another sequence β by applying one production of the grammar a **one-step-derivation**. We denote this derivation $\alpha \Rightarrow_G \beta$. We call a **derivation** a sequence of one-step derivations and denote such a derivation $\alpha \Rightarrow_G^* \beta$.

The formal definition of a language generated by a grammar is as follows: The language $L(G)$ generated by a grammar $G = (V_T, V_N, P, S)$ is the set of all words which can be optained or derived by rewriting from the axiom S and do not contain any terminals:

$$L(G) = \{w \mid S \Rightarrow_G^* w, w \in V_T^*\}$$

Examples:

1. The grammar
$$G = (\{a, b\}, \{S, A\}, P, S)$$
in which
$$P = \{S \to bbA, A \to aaA, A \to aa\}$$
generates the language
$$L(G) = \{bb\} \cdot \{aa\}^n \text{ where } n \geq 1.$$

2. The (regular) grammar
$$G = (\{0, 1\}, \{S\}, P, S)$$
where
$$P = \{S \to 1S, S \to 0S, S \to 1\}$$
generates
$$L(G) = \{0, 1\}^* \cdot \{1\}.$$

3. The (context-free) grammar
$$G = (\{a, b\}, \{S\}, P, S)$$
where
$$P = \{S \to aSb, S \to \epsilon\}$$
generates
$$L(G) = \{a^n b^n \mid n \geq 0\}.$$
The empty string ϵ means that a term can be replaced by a string of length zero.

4. The language $L = \{a^n b^n c^n \mid n \geq 1\}$ is generated by the grammar
$$G = (\{a, b, c\}, \{S, A, B, C, D\}, P, S)$$
where $P = \{S \to aACD, A \to aAC, A \to \epsilon, B \to b, CD \to BDc, CB \to BC, D \to \epsilon\}$. This grammar is neither regular nor context-free.

2.4.2 Regular Grammar

We formally define **regular grammars** as follows: A grammar is **right-regular** if all rewriting rules have the form $\alpha \to \beta$ with the following restrictions:

1. $|\alpha| = 1, \alpha \in V_N$
2. $\beta = aB, \beta = a$, or $\beta = \epsilon$ with $B \in V_N$ and $a \in V_T$

A grammar is **left-regular** if all productions have the form:

1. $|\alpha| = 1, \alpha \in V_N$
2. $\beta = Ba, \beta = a$, or $\beta = \epsilon$ with $B \in V_N$ and $a \in V_T$

Any right or left-regular grammar us simply called a regular grammar.

It is important to note that a regular grammar is either left regular or right regular, but not both (except for the trivial cases). If the rules of right and left-regular grammars are mixed, then the language generated by the grammar can be non-regular. Non-regular grammars generate more complex languages than regular ones.

FSA and regular grammars are equivalent: A language L is recognized by a FSA iff L is generated by a regular grammar.

2.4.3 Regular Expressions

Regular expressions are the third formalism after FSA and regular grammars that can be used to specify regular languages. We say that a regular expression **denotes** a regular language (automata recognized and grammars generate). Regular expressions are easy to integrate into software systems and are widely used for searching sub-strings in text, for instance in editors or web search engines. Regular expressions are also intensively used for syntax analysis and consequently for compiler design.

The formal definition of a regular expression is as follows: Let Σ be an alphabet. A regular expression defined over Σ is recursively defined as follows:

1. $\emptyset$ us a regular expression that denotes the empty set
2. ϵ is a regular expression that denotes the set $\{\epsilon\}$
3. For any $a \in \Sigma$, a is a regular expression that denotes the set $\{a\}$
4. If r and s are two regular expressions which denote the sets R and S respectively then $(r + s)$, (rs) and (r^*) are regular expressions that denote the sets $R \cup S$, $R \cdot S$ and R^*.
5. Nothing else is considered a regular expression

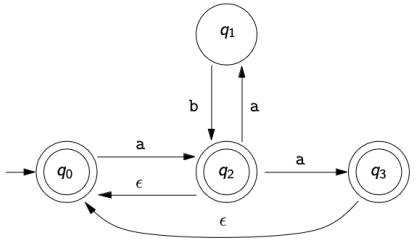
The language denoted by a regular expression r is written $L(r)$.

Examples:

1. Let $\Sigma = \{a, b\}$ and the regular expression
$$r = (a(a + b)^*)$$
then
$$L(r) = \{a\} \cdot \{a, b\}^*.$$

2. The regular expression
$$r = (0^*1)^* + 0$$
denotes the set
$$L(r) = (0^* \cdot \{1\})^* \cup \{0\}$$
or
$$L(r) = \{x \mid x \in \{0, 1\}^*, x \text{ represents an odd binary number } \} \cup \{\epsilon\}.$$

3. The regular expression
$$r = (a(ab) * (a + \epsilon))^*$$
corresponds to the automaton M:



The following equalities hold for regular expressions:

- $r + s = s + r$
- (r + s) + t = r + (s + t)
- $(rs)t = r(st)$
- $r(s + t) = rs + rt$
- $(r + s)t = rt + st$
- $\emptyset^* = \epsilon$
- $(r^*)^* = r^*$
- $(\epsilon + r)^* = r^*$
- $(r^* s^*)^* = (r + s)^*$

Regular expressions are equivalent to FSA:

- Regular expressions to FSA: From any regular expression r, there exists a NFA with ϵ-transitions which recognizes $L(r)$.
- DFA to regular expressions: If a language L is recognized by a DFA, then there exists a regular expression r that denotes L.

3 Context-free Languages

3.1 Introduction

Basic example of a non-regular language: L is composed of all words starting with a's and ending with the same number of b's:

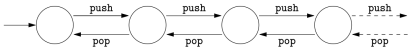
$$L = \{a^n b^n \mid n \geq 0\}$$

This language cannot be recognized by a FSA, nor generated by a regular grammar, nor denoted by a regular expression. The limitation comes from the fact that FSA have a finite set of states. Consequently, FSA cannot memorize and arbitrary large number n of symbols a and verify that the same number of b follows.

Non-regular languages include:

- Nested paranthesis in arithmetic expressions
- Nested `begin`, `end` in programming languages
- Nested { and } in programming languages
- And a lot more...

Example (Stack): FSA cannot be used to model a stack with the operations push and pop, for which the number of push operations is arbitrarily large.



Note that if a language L is non-regular, a language $L \subset L'$ is not necessarily non-regular: *Example:* The left-hand side language is non-regular, whereas the right-hand side language is regular:

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

The distinguishing characteristic is the structure.

3.2 Pushdown Automata

To overcome some limitations of FSA, **pushdown automata** have been devised:

- A pushdown automaton (PDA) is a kind of finite state automaton equipped with a stack that memorizes symbols.
- The transitions of a pushdown depend on an input symbol but also on the symbol at the top of the stack.
- The size of the stack has no limit.

Example: Using the previous language $\{a^n b^n \mid n \geq 1\}$, the PDA behaves as follows:

1. While a symbol can be read, this symbol is pushed onto the stack.
2. When the first input symbol b comes, then one symbol a is removed from the stack.
3. While a symbol b can be read, one symbol a is removed from the stack.
4. The process ends when the stack is empty and all the input symbols have been consumed.

Formal definition: A pushdown automaton PDA is a 6-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, \triangleleft)$, where:

- Q is a finite set of states
- Σ is an input alphabet
- Γ is an auxiliary alphabet (symbols of the stack) such that $\Gamma \cap \Sigma = \emptyset$ (i.e. Γ and Σ share no elements)
- $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \to Q \times \Gamma^*$ is a transition function
- $q_0 \in Q$ is the initial state
- $\triangleleft \in \Gamma$ is a special symbol which indicates that the stack is empty

A PDA that recognizes a language is defined similarly: We add the set of final states. This means that a recognizer is a 7-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, \triangleleft, F)$, where $F \subseteq Q$ is the set of final states.

Example: Let us construct a PDA M which recognizes $a^n b^n \mid n \geq 1$:

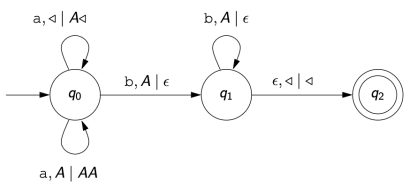
$$M = (\{q_0, q_1, q_2\}, \{a, b\}, \{A, \triangleleft\}, q_0, \triangleleft, \{q_2\})$$

where

$$\begin{aligned} \delta(q_0, a, \triangleleft) &= (q_0, A\triangleleft) \\ \delta(q_0, a, A) &= (q_0, AA) \\ \delta(q_0, b, A) &= (q_1, \epsilon) \\ \delta(q_1, b, A) &= (q_1, \epsilon) \\ \delta(q_0, \epsilon, \triangleleft) &= (q_2, \epsilon) \end{aligned}$$

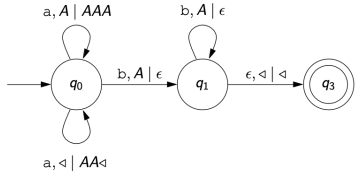
Convention: $(a, Z \mid \alpha)$ means:

- a is the input symbol
- Z is the symbol at the top of the stack before applying the transition function
- α is the content of the stack after the application of the transition function



The empty string ϵ can be used as an input symbol. This is useful to detect that the stack is empty.

Example: A PDA that recognizes $L = \{a^n b^{2n} | n \geq 1\}$ could be:



3.2.1 Limitations of PDA

The language $L = \{a^n b^m | m = n \text{ or } mm = 2n\}$ cannot be recognized by a PDA with a single stack. It would require two stacks, where the first stack would recognize the number n and the second one would recognize the number $2n$.

3.2.2 Non-deterministic PDA

Non-determinism is modeled by means of a transition function whose co-domain is a power set. The formal definitions is as follows: A non-deterministic PDA is a 7-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, \triangleleft)$ with $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \to \mathcal{P}(Q \times \Gamma^*)$ where $\mathcal{P}(Q \times \Gamma^*)$ denotes the power set of $Q \times \Gamma^*$. A word is recognized by a non-deterministic PDA iff there is at least one path between the initial state and a final state.

It is important to observe that the classes of languages recognized by deterministic and non-deterministic PDA are not the same. Non-deterministic PDA recognize a larger class of languages than deterministic PDA, i.e. they are more powerful. This is not the case for FSAs.

3.3 Context-free Grammars

The difference between regular grammars and **context-free** grammars is the restrictions imposed on the rewriting rules. The rules of regular grammars are more restrictive than the ones of context-free grammars, which means that any regular grammar is also a context-free grammar. Context-free grammars have a greater expression power than regular ones.

The formal definition of context-free grammars is as follows: A context-free grammar $G = (V_T, V_N, P, S)$ is a grammar whose productions have the form $\alpha \to \beta$, where

1. $|\alpha| = 1$ and $\alpha \in V_N$
2. β is a sequence of terminal and/or non-terminals

The language generated by a context-free grammar G is called a **context-free** language, denoted by $L(G)$.

Example: Let $L = \{a^n b^n \mid n \geq 1\}$. The grammar that generates L can be defined as:

$$G = (\{a, b\}, \{S\}, P, S)$$

where $P = \{S \to aSb, S \to ab\}$

Example: Let G = ({a,b}, {S,A,B}, P, S) with

$$\begin{aligned} P = \{&S \to aB, \\ &S \to bA, \\ &A \to a, \\ &A \to aS, \\ &A \to bAA, \\ &B \to b, \\ &B \to bS, \\ &B \to aBB\} \end{aligned}$$

G is context-free and generates the language of all words composed of the same number of symbols a and b.

3.4 BNF

The **Backus-Naur Form** (BNF) is a notation used to describe the syntax of programming languages (originally Algol 60). BNF uses ASCII characters only and is equivalent to the notion of context-free grammars.

- Non-terminals are strings, e.g. `type`, `identifier`, etc.
- Terminals are (single or double) quoted strings or symbols, such as `"while"`, `"float"`, `'s'`, `'+'`, etc.
- = corresponds to the arrow of a production $\alpha \to \beta$
- | corresponds to alternative
- , concatenates elements
- ; terminates a production

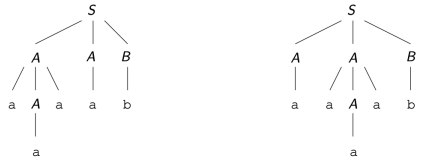
The extended BNF (EBNF) is similar, but introduces some new constructs that make the notation easier to read and more compact. The most immporant aspects of EBNF are:

- Parantheses can be used to group elements
- `[` γ `]` means that γ is optional, i.e. appears zero or one times.
- { γ } means that γ can be repeated zero or more times.

3.5 Parse Trees, Ambiguity

The application of rewriting rules captures the notion of the **parse tree**. For some grammars, a word can be generated in several different ways and have thus several different parse trees. Such words are generated ambiguously and may potentially have several different meanings. Having different meanings is sometimes undesirable, especially in the context of programming languages, where a given program should have unique semantics.

Example: Consider the productions $S \to AAB, A \to aAa, A \to a, B \to b$. There are several ways of generated the string *aaaab*, e.g.



The definition of an ambiguous grammar is as follows: A grammar G is an ambiguous grammar iff there exists a word $w \in L(G)$ such that w has more than one parse tree. A language L is an ambiguous language iff all the grammars that generate L are ambiguous.

The theorem for PDA and context-free languages is as follows: A language L is recognized by a non-deterministic PDA iff L is a context-free language.

4 Parsing

Processing an input text consists generally in, verifying that the structure of the text is correct, and in performing some operations using the text under analysis.

In general, a grammar describes the syntactical structure of the input text. Thus, verifying that the structure of the text is correct consists in making sure that the text is in conformity with the grammar. This is **parsing**.

The parsing operation is divided into two activities:

1. Lexical analysis
2. Syntax analysis

4.1 Lexical Analysis

The **lexical analysis** activity consists in reading a text character by character and in grouping the characters that form terminals of the grammar. Such terminals are usually called **tokens**. E.g.

- Identifiers of programming languages, such as `anItem`, `n`
- Keywords of programming languages, such as `if`, `extends`
- Various numerical constants
- Character strings such as `"This is a comment"`, `"Yes"`
- Specific tokens such as `(`, `=`, etc.

Between terminals lie separators, such as blank spaces, tabs, new lines, and comments, which are all usually ignored. We usually use `lexers` or `scanners`, which are programs that perform the lexical analysis of an input text. The implementation of lexers is generally based on the use of regular expressions and/or FSAs. Some lexer tools include `lex`, `flex`, `jflex`, anf more.

4.2 Syntax Analysis

The second activity of parsing consists in verifying that the input text is well constructed, i.e. the input is in conformity with a given grammar. For the verification of the conformity with respect of a given grammar, we build the parse tree of the input text.

An input text is syntactically correct iff the parse tree can be built.

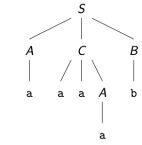
A program that performs this activity is called a **parser**. The main task of a parser is to create the parse tree of its input using a grammar that describes the structure of its input. Note that a parser uses a lexer for the lexical analysis.

Parse trees graphically illustrate how a given word is generated by a grammar, i.e. which productions are used to generate a given word.

4.2.1 Bottom-Up Analysis

One approach to generate a parse tree of an input according to a given grammar is the **bottom-up analysis**.

1) $S \to ACB$
2) $A \to \mathtt{a}$
3) $C \to \mathtt{aa}A$
4) $B \to \mathtt{b}$

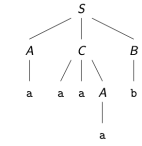


The parse tree of the input *aaaab* can be constructed from the terminals using the rules 2) twice and 4) once, then rule 3) and finally rule 1).

4.2.2 Top-Down Analysis

The second approach called **top-down analysis** consists in applying derivations successively from the first rewriting rule or axiom. This approach is easier but has some limitations.

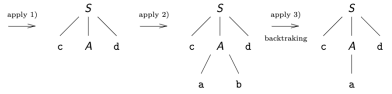
1) $S \to ACB$
2) $A \to \mathtt{a}$
3) $C \to \mathtt{aa}A$
4) $B \to \mathtt{b}$



4.2.3 Recursive Descent

Recursive Descent is a top-down approach in which we execute a programm that applies a set of recursive functions to process the input. This may involve backtracking, but we will try to get rid of it, for efficiency and simplification reasons.

1) $S \rightarrow cAd$
2) $A \rightarrow ab$
3) $\quad\;\; | \; a$



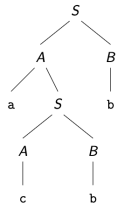
4.3 Classes of Context-free Grammars

The class of LL(1) grammars is a subset of context-free grammars. One way of parsing such grammars is to use recursive descent parsers in which no backtracking is required.

- LL(1) stands for Left to right Left-most derivation with 1 token lookahead.
- LL(1) is one of the easiest (sometimes fastest) techniques.
- A recursive descent parser has a parsing function (or method) for each non terminal that can recognize any sequences of tokens generated by that nonterminals.
- The idea of a recursive descent parser it to use the current input token to decide which alternative to choose.

Left-most derivation means that the left-most rule is used (expanded) first. For example the word *acbb* gets recognized as follows:

$S \rightarrow AB$
$A \rightarrow aS \mid c$
$B \rightarrow b$



Left-most derivation visits the nodes of a parse tree according to the DFS algorithm. The main problem of recursive descent is **left-recursion**, which causes the parser to enter into an infinite recursion.

4.3.1 Left-recursion

Left recursion has **immediate left-recursion** if it has the form

$$X \rightarrow X\alpha \mid \beta$$

or more generally

$$X \rightarrow X\alpha_1 \mid \ldots X\alpha_n \mid \beta_1 \mid \cdots \mid \beta_n$$

General left-recursion can occur through several rules for example:

$$S \rightarrow Aa \mid b$$

$$A \rightarrow Ac \mid Sd \mid \epsilon$$

To remove immediate left-recursion, auxiliary rules are required, e.g.:

$$X \rightarrow X\alpha_1 \mid \ldots X\alpha_n \mid \beta_1 \mid \cdots \mid \beta_n$$

becomes

$$X \rightarrow \beta_1 X' \mid \cdots \mid \beta_n X'$$

$$X' \rightarrow \alpha_1 X' \mid \cdots \mid \alpha_n X' \mid \epsilon$$

4.3.2 Left-factoring

When two or more grammar rule choices share a common prefix, LL(1) parsers cannot determine which choice to use or which part has to be expanded:

$$X \rightarrow \alpha\beta \mid \alpha\gamma$$

To solve this, we can factorize the common prefix:

$$X \rightarrow \alpha X'$$

$$X' \rightarrow \beta \mid \gamma$$

4.4 LL Grammars

A grammar is LL(1) iff there exists a recursive descent parser with 1 token lookahead for this grammar (top-down analysis). A grammar is LL(k) iff there exists a recursive descent parser that uses k tokens lookahead.

The class of LL(k) grammars strictly includes the class LL($k-1$), where ($k > 1$); in other words there are LL(k) grammars for which no LL($k-1$) grammar exists. In other words LL(k) parser are more powerful than LL($k-1$) parsers.

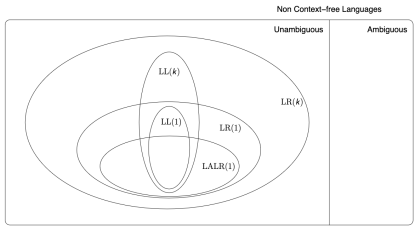
Fortunately most LL(k) grammars can be translated into LL(1) grammar, but not all.

4.5 LR Grammars

LR(1), LR(k) grammars can be analyzed with parsers based on the bottom-up technique with 1 token lookahead, k token lookahead respectively. LR(k) stands for Left to right Right-most derivation with k tokens lookahead. Bottom-up parsers are much more powerful than top-down parsers. It has been shown that any LR(k) grammar ($k > 0$) can be translated into a LR(1) grammar.

4.6 Comparison of LL and LR Parsers

The following shows the relationships between LL and LR:



LR parsers are (disadvantages of LR vs LL):

- Much more complex than LL parsers
- More difficult to use
- Less efficient
- Require much more space than LL parsers
- Only used through parser generators. Due to the size of tables of LR parsers, parser generators only work for a subclass of LR grammars called LALR (lookahead LR)

LR parsers are (advantages of LR vs LL):

- Left-recursion is allowed (grammars are easier to read)
- More complex context-free languages can be analyzed
- Errors are detected as soon as they are encountered
- Almost all programming languages can be parsed by bottom-up parsers