St. Pölten University of Applied Sciences

/informatik & security

/fh/// st. pölten

Formal Languages and Automata 1

Theoretical Computer Science

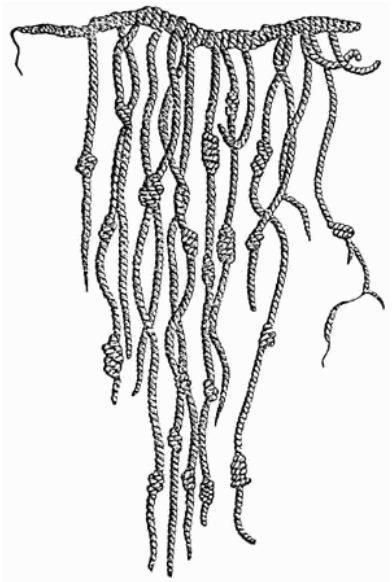
Dipl.-Ing. Hubert Schölnast, BSc
September 12, 2021

St. Pölten University of Applied Sciences, Campus-Platz 1, 3100 St. Pölten, T: +43 (2742) 313 228, F: +43 (2742) 313 228-339, E: csc@fhstp.ac.at, I: www.fhstp.ac.at

Table of Contents

1 Definitions

1.1 Sign



Quipu: A writing system based on knots in strings

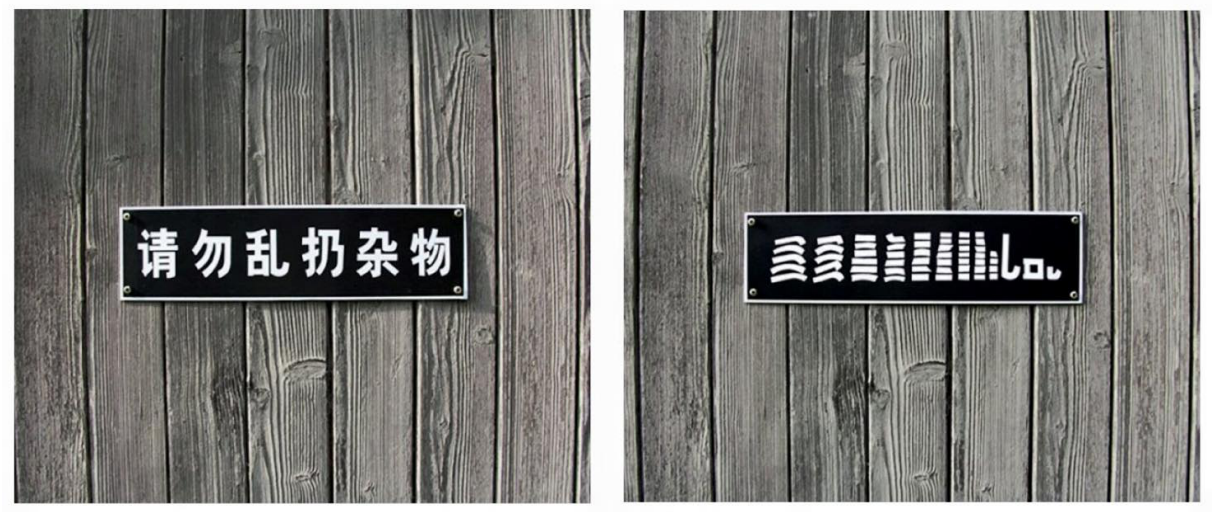
Signs are the basic elements of languages. (At least in the sense in which this term is used in the field of formal languages).

Signs can be letters, digits, sounds, electromagnetic patterns, nodes, and very often groups of such atomic entities.

For example, English grammar is the art of combining words to form valid English sentences. In this context, individual letters do not play the role of signs, but all words that occur in the German vocabulary are signs, in the sense in which we will use this term here.

In fact, anything can be a sign. The only condition is that these things are used to form words and languages, as words and languages will be defined in the next sections.

Thus, signs are the basic building blocks of a language. If you break down these signs into smaller elements (if you break down an English word into letters, a letter into lines, circles and arcs etc.), then you are no longer in the context of that language.



Left: Six Chinese signs. Right: No signs at all.
By breaking down signs into smaller objects, the context of the language is abandoned.

1.2 Alphabet

An alphabet is a set in the mathematical sense of set theory. And the elements of an alphabet are signs, as defined in 1.1.

The cardinality of an alphabet is finite. This means that there is no such thing as an infinite alphabet.

There is no explicit rule that prevents the empty set from being an alphabet, but the only word that a language based on an empty alphabet can have is the empty word (see 1.3.1). However, this is only of academic interest and of no practical value. So for this course, we define that an alphabet contains at least one sign. But even languages with only one sign are a bit boring. Only alphabets with 2 or more signs are really interesting and practically useful.

Note that this definition of the alphabet is a little different from the definition you may intuitively have in your mind. If you define English (or any other language written with Latin letters) on a single letter basis, then the alphabet as defined here includes not only the 26 letters from A to Z. The alphabet used for English also includes the 26 lowercase letters from a to z, 10 digits, the space sign, punctuation, parentheses, currency symbols, special signs like #, @, %, and &, and many, many more signs.

International nautical flag alphabet

Just think of the texts in math books and the many signs that are needed for mathematical formulas. The alphabet needed to write such texts includes far more than 100, maybe even more than 1000 different signs.


Alphabet with signs without any order

Another difference: In the alphabet you learned in school, the letters have a certain order. *E* comes somewhere before *M* and *P* comes after them. But we have defined an alphabet as a set, and we have not defined a mechanism that creates an intrinsic order of the signs in the alphabet.

So there is no intrinsic order of the signs in an alphabet. It's like the bag you know from Scrabble. But this bag contains each sign only once, and when you take a sign out of the bag, you always get a new copy, and the sign is still in this bag.

1.2.1 Standard symbol and examples

The standard symbol for an alphabet is the Greek capital letter Sigma: Σ. The signs contained in this alphabet are written like normal elements of a set:

$$\Sigma = \{0,1\}$$

This example of an alphabet consists of two signs, and these signs are the digits 0 and 1. You probably know that everything that is stored on a computer is stored in bits, and 0 and 1 are the two values that a bit can have. Therefore, any executable program, PDF document, image, music file, video, virtual reality game, and anything else that can exist on a computer can be written using this very simple alphabet.

$$\Sigma = \{\blacksquare\}$$

This is an alphabet that has only one sign. The only way words (see below, 0) formed from this alphabet differ from each other is their length. There is an infinite number of possible lengths, so this results in an infinite number of different words. But as soon as you need more than one word, you have to find a way to separate two words, and then you need a second sign.

In many situations we will use two different alphabets. Remember the example with the big cat and the sleeping guitar? There we had 8 red "words", which in our new terminology are not words but signs, and we had 6 blue "phrases", which in the terminology of formal languages are also nothing but signs. And the red sign set and the blue sign set are two different alphabets with different properties.

The red sign set contained the 8 signs from which the final "sentences" were to be formed, and the blue sign set contained auxiliary signs whose only purpose was to be replaced by other signs (which are either final signs or auxiliary signs). You will get this concept explained in more detail soon, but until then you should know that we will often have two separate alphabets:

- Σ or T = the alphabet that contains the final ("terminal") signs.
- V or N = the alphabet that contains the auxiliary signs, also called "variables" or "non-terminal" signs.

1.3 Word

A word, in the context of formal languages, is a concatenation of signs that are elements of a particular alphabet.

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4 \cdot a \cdot c}}{2 \cdot a}$$

A word written with signs from the language of mathematics

Note that an alphabet is a set, but a word is not. In an alphabet, each sign occurs exactly once. But in a word, each sign can occur not at all, once, twice, three times, or any number of times.

In an alphabet, the signs have no particular order, but the order of the signs in a word is important. If we again take the alphabet $\Sigma = \{0,1\}$, then we can form these words among many others:

- 01
- 10
- 0
- 0000

These are four different words. (Like the English words "on" and "no", which are two different words, or "god" and "good").

Technically, words are mathematical sequences, and the standard notation for sequences is the following (shown for the same 4 words as before):

- $(0,1)$
- $(1,0)$
- (0)
- $(0,0,0,0)$

(S, o) (w, e) $(\mathrm{s}, \mathrm{h}, \mathrm{o}, \mathrm{u}, \mathrm{l}, \mathrm{d})$ $(\mathrm{a}, \mathrm{c}, \mathrm{t}, \mathrm{u}, \mathrm{a}, \mathrm{l}, \mathrm{l}, \mathrm{y})$ $(\mathrm{p}, \mathrm{u}, \mathrm{t})$ $(\mathrm{a}, \mathrm{l}, \mathrm{l})$ $(\mathrm{w}, \mathrm{o}, \mathrm{r}, \mathrm{d}, \mathrm{s})$ (i, n) $(\mathrm{r}, \mathrm{o}, \mathrm{u}, \mathrm{n}, \mathrm{d})$ $(\mathrm{b}, \mathrm{r}, \mathrm{a}, \mathrm{c}, \mathrm{k}, \mathrm{e}, \mathrm{t}, \mathrm{s})$ $(\mathrm{a}, \mathrm{n}, \mathrm{d})$ $(\mathrm{s}, \mathrm{e}, \mathrm{p}, \mathrm{a}, \mathrm{r}, \mathrm{a}, \mathrm{t}, \mathrm{e})$ $(\mathrm{t}, \mathrm{h}, \mathrm{e})$ $(\mathrm{i}, \mathrm{n}, \mathrm{d}, \mathrm{i}, \mathrm{v}, \mathrm{i}, \mathrm{d}, \mathrm{u}, \mathrm{a}, \mathrm{l})$ $(\mathrm{s}, \mathrm{i}, \mathrm{g}, \mathrm{n}, \mathrm{s})$ $(\mathrm{w}, \mathrm{i}, \mathrm{t}, \mathrm{h})$ $(\mathrm{c}, \mathrm{o}, \mathrm{m}, \mathrm{m}, \mathrm{a}, \mathrm{s})$. But this is very annoying, and in natural languages we do not write words this way. So when we are in the context of formal languages and when there is no doubt about what is meant (which is usually always the case), we write all words without brackets and commas. This is a convenient convention, but sometimes it is misleading, especially when we need to distinguish signs from one-sign-words.

The length of a word can be any natural number. This means that a word of 0 signs is allowed (see "the empty word", 1.3.1). But it also means that words cannot be infinitely long (but there is no upper limit to the length of a word).

Grundstücksverkehrsgenehmigungszuständigkeitsübertragungsverordnung

A valid and existing German word (in English: *Ordinance on the delegation of authority concerning land conveyance permissions*)

The full chemical name for the human protein titin is a word that is 189,819 letters long and takes about three-and-a-half hours to pronounce.

1.3.1 The empty word

As mentioned earlier, it is permissible for a word to contain no sign at all. Technically, this is the empty sequence, which is closely related to the empty set. Note that a word of length 0 is an existing thing, so it is not nothing. The empty word is not no word!

The empty word wouldn't actually be a big deal if there wasn't a problem with its notation. It's easy to write it in the standard notation for sequences. Then you just write a pair of round brackets with nothing between them:

$$( )$$

But, as mentioned a few lines above, this is not the way words are written in the context of formal languages. If we were to write the empty word as usual, without brackets (and without commas, which don't exist here anyway), we wouldn't need any signs at all to write it, but that's exactly how we write nothing. But, as said, the empty word is not nothing. The solution is a special symbol for the empty word:

$$\varepsilon := ( )$$

It is common to write the empty word with the Greek lowercase letter Epsilon. Note that this symbol is **not** a sign in an alphabet! It is the symbol for the empty word. Words and signs are different things!

Therefore, even in formal languages, it is customary not to use the lowercase Greek letter epsilon as a sign for an alphabet. If you must use it as a sign, you must explicitly define another symbol, which is not a sign of the alphabet you are using, as the symbol for the empty word.

1.3.2 The term "word" in everyday life and in formal languages

As mentioned in the presentation on semiotics, there is a certain hierarchy in natural languages:

The smallest building blocks of spoken languages are certain sounds, which linguists call "phonemes".

The next level in all languages spoken by humans are syllables, and there are rules about how phonemes can be put together to form syllables. In our new terminology we have:

- a sign = a phoneme (basic sound of articulation) (including: silence).
- the alphabet = all phonemes used in a particular language
- one word = one syllable

The syllables, in turn, are assembled into the smallest meaning-bearing building blocks. These "meaning atoms" are called "morphemes":

- a sign = a syllable
- the alphabet = all syllables within a certain natural language
- one word = one morpheme

When we talk about written languages, we start with letters, digits, punctuation, etc. and form "words":

- a sign = a letter, a digit, a space, etc.
- the alphabet = all the signs needed to write text
- a word = a word (as it is used in normal everyday language)

There are rules that define how these signs can be combined to form words. These rules are called "orthography".

But words can also be put together to form sentences:

- a sign = a word (as used in everyday language) or a morpheme
- the alphabet = all words or morphemes that exist in a certain language
- a word = a sentence

Again, there are rules that determine how words can be connected to form sentences. These rules are called "grammar".

Sentences can subsequently be grouped into paragraphs, paragraphs form chapters, and chapters combine to form whole books. So in natural languages we have a hierarchy in which the words of one level are the signs of the next level.

But we can flatten that hierarchy and analyze, for example, a politician's entire speech as something built out of phonemes, or we can treat an entire book as a sequence of letters, spaces, punctuation marks, and other signs. And this is exactly the way languages are handled in formal languages.

In formal languages, there is no hierarchy. If you analyze a book at the level of single signs, then the whole book is a single word in the sense of formal languages!

It is very important that you understand that the term "word" as used in formal languages can sometimes mean the same thing as "word" in normal everyday life, but much more often means a different linguistic structure. Confusing these two meanings of the term "word" is one of the main reasons why people have difficulty understanding formal languages. So make sure you always use the right meaning in the right context!

1.4 Kleene Closure

This term is easier to understand because it cannot be confused with terms from everyday life, because this term doesn't exist in everyday life.

The Kleene closure of an alphabet is the set of all possible words that can be formed by combining signs from this alphabet. This set is named after the American mathematician and logician Stephen Cole Kleene (1909-1994), who was one of the co-founders of theoretical computer science.

Technically, a Kleene closure is a free monoid over the alphabet, and its elements are all possible sequences that can be formed from the elements of that alphabet. But in the context of formal languages, we will always use the term "Kleene closure". However, the notation for Kleene closures is the same as for free monoids over arbitrary sets:

If Σ is the alphabet, then its Kleene closure is written as follows:

$$\Sigma^*$$

This $*$ is a unary operator and its name is "Kleene star". It takes an alphabet as input and generates all possible words that can be written with the signs in that alphabet.

Example

Let Σ again be the very simple alphabet with only 2 signs that we had before:

$$\Sigma = \{0,1\}$$

Then its Kleene closure is:

$$\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, \cdots\}$$

in the stricter notation:

$$\Sigma^* = \{(), (0), (1), (0,0), (0,1), (1,0), (1,1), (0,0,0), (0,0,1), (0,1,0), (0,1,1), (1,0,0), (1,0,1), (1,1,0), (1,1,1), \cdots\}$$

Only words that do not exceed a length of 3 signs are shown here. But it is easy to understand that the Kleene closure also includes all words with a length of 4 signs as well as all words with a length of 5 signs and all words with an even greater length.

This leads to an important consequence:

Although by definition no alphabet of any language can contain an infinite number of signs, and although it is impossible (again by definition) for any word to contain an infinite number of signs, the Kleene closure of any non-empty alphabet always contains an infinite number of words.

But we can easily enumerate the words in a Kleene closure. All we have to do is invent any order for the signs in the alphabet. This is always possible, because every alphabet has a finite number of signs. In Kleene closure, we sort words by size, shortest words first, starting with the empty word, then all words of length 1, then length 2, and so on. Within a group of words of the same length, we arrange the words lexically, like "real" words in a dictionary, and we use the previously invented order of letters in the alphabet to do this.

Then every single word has its well-defined place in the sequence of all words. So we can assign a uniquely determined natural number to each word, and each natural number is the index of a uniquely determined word.

$$\begin{array}{c} \Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, \cdots\} \\ \updownarrow\updownarrow\updownarrow\ \updownarrow\ \updownarrow\ \updownarrow\ \updownarrow\ \updownarrow\ \updownarrow\ \updownarrow\ \updownarrow\ \updownarrow\ \updownarrow\ \updownarrow\ \updownarrow\ \updownarrow \\ \mathbb{N} = \{0, 1, 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9,\ 10,\ 11,\ 12,\ 13,\ 14, \cdots\} \end{array}$$

This is a bijective mapping between the Kleene closure and the set of natural numbers, and it means that any Kleene closure has exactly the same cardinality (i.e. the same size) as the set of natural numbers.

$$|\Sigma^*| = |\mathbb{N}| = \aleph_0$$

The technical term for creating a bijective mapping between a set and an interval of the set of natural numbers starting at 0 or 1 is "counting". Counting is the oldest mathematical operation known to mankind. Even some animals can perform this bijective mapping on intervals of $\mathbb{N}$ if the intervals involved are small enough. We call sets that can be mapped bijectively to an interval of $\mathbb{N}$ starting at 0 or 1 "countable".

But since the whole set $\mathbb{N}$ is also an interval of itself, in mathematics we also use this term for a bijective mapping of a set to the whole set $\mathbb{N}$. And this is why we also call sets "countable" for which we can create a bijective mapping with all $\mathbb{N}$. But since these sets are infinite, sometimes we are a little more precise and say "countably infinite".

So every Kleene closure is a countably infinite set.

1.4.1 Positive closure

Sometimes we explicitly want to exclude the empty word from the set of all words, and this is exactly what the positive case does. It is the Kleene closure without the empty word and is written with a $+$ instead of a $*$:

$$\Sigma^+ = \Sigma^* \backslash \{\varepsilon\}$$

$$\Sigma^+ \cup \{\varepsilon\} = \Sigma^*$$

$$\Sigma^+ \cap \{\varepsilon\} = \emptyset$$

1.5 Grammar

A grammar is a set with exactly four elements. Three of these elements are sets, the other element is a sign that must be contained in a specific one of the three sets.

Here is the formal definition:

$$G = \{V, \Sigma, P, S\}$$

- G = a grammar
- V = the alphabet of variables (auxiliary signs = non-terminal signs)
- Σ = the alphabet of final (terminal) signs
- P = the set of production rules
- S = the starting sign

Remember the example with the cat and the guitar? That example was a grammar as defined here.

The red set was Σ and it contained these 8 signs: (Don't use the term "words" anymore! You already know the correct terminology!)

$$\Sigma = \{\text{a, big, cat, guitar, plays, sleeps, small, the}\}$$

The blue set was V, the set of variables with this 6 auxiliary signs:

$$V = \{\text{adjective, article, nominal\_group, noun, sentence, verb}\}$$

And if you compare these two alphabets, you will see that there is no sign that appears in both alphabets. The two alphabets are disjoint:

$$\Sigma \cap V = \emptyset$$

This was not a coincidence. It always has to be that way. Why it always has to be this way will become a little clearer when we talk about production rules, and when we finally define the term "language" it will seem inevitable. For now, just note that a sign can never appear in both alphabets of the same grammar.

In the cat-and-guitar example, we also had a special sign with which we started the substitutions:

$$S = \text{sentence}$$

The start sign in the example was a member of V, and that's not a coincidence either. This must also always be the case.

$$S \in V$$

And this also becomes clear when we learn the details of the last element in G, namely P, the set of production rules.

1.5.1 production rules

P is the set of production rules. Each element of P is a rule that looks like this:

$$l \to r$$

Each rule has a left side l and a right side r, and between them we write an arrow pointing from left to right. Both l and r are words:

$$l \in (V \cup \Sigma)^+$$

$$r \in (V \cup \Sigma)^*$$

So you form the union of the two alphabets, creating a new alphabet. And from this united alphabet you form the positive closure and the Kleene closure. Both closures are infinite sets of words. The only difference between the two sets of words is the empty word contained in $(V \cup \Sigma)^*$ but excluded from $(V \cup \Sigma)^+$.

Each production rule means:

If a given word contains the partial word l, then this partial word l can be replaced by r. Everything before or after this manipulated part of the whole word remains as it was before.

Note that the word "can" occurs in the definition. This is necessary because sometimes there are different partial words that correspond to the left-hand sides of rules. Sometimes they overlap or are even identical. And if this is the case, you are free to decide which of the matching rules to use. So in most cases there are different sequences of rules that can be applied to a given word, and they usually lead to different results.

Usually, there is always at least one production rule where $l = S$. You need a word to start with, and that word is S. So you also need at least one rule to replace S with something else, otherwise you can never replace anything, and the grammar will not be able to produce a word. Also the empty word cannot be produced, because, as already mentioned, the empty word is no word!

But wait!

Here we just claimed that S is a word, but a few lines before we claimed that S must be an element of V. But V does not contain words but only signs. How is this to be explained?

Well, it is because of the abbreviated notation of words that we introduced to write words without $(\mathrm{b}, \mathrm{r}, \mathrm{a}, \mathrm{c}, \mathrm{k}, \mathrm{e}, \mathrm{t}, \mathrm{s})$ $(\mathrm{a}, \mathrm{n}, \mathrm{d})$ $(\mathrm{c}, \mathrm{o}, \mathrm{m}, \mathrm{m}, \mathrm{a}, \mathrm{s})$. If we were to write words correctly as mathematical sequences, we would define a grammar as we defined it before, but with these significant differences:

- S = The start **word** that consists of exactly one sign from V

$$S = (x); \qquad x \in V$$

Thus, S is a word (i.e., a sequence of signs), and this sequence consists of only one sign, which is an element of V.

However, in our shorthand notation for words, we no longer have a way to distinguish between signs and one-sign-words.

Consequently, in most books on formal languages, you will find a definition that S must be an element of V, but if you really want to get it right, you should say that S is a word of length 1, and the sign in this word must be an element of V. In computer science, it is already a common convention to treat single signs and one-sign-words as identical entities, and here, in this document, we will follow this convention because it usually does not cause irritation. And when necessary, we will be stricter and pay closer attention to the difference.

1.5.2 Example

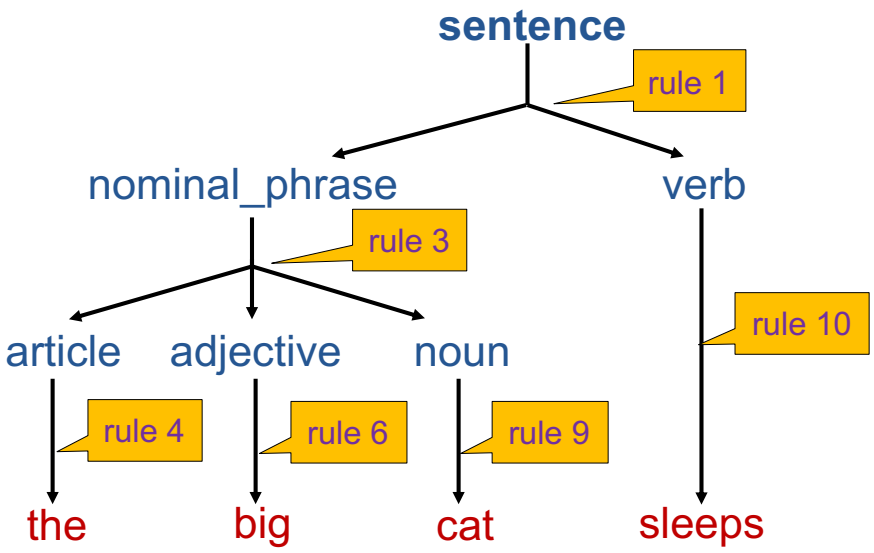
So, how do the rules from the cat-and-guitar-example look like in actual notation? Well, here they are:

```
 1        sentence → nominal_group  verb
 2 nominal_group → article  noun
 3 nominal_group → article  adjective  noun
 4         article → the
 5         article → a
 6       adjective → big
 7       adjective → small
 8            noun → guitar
 9            noun → cat
10            verb → sleeps
11            verb → plays
```

You will notice that in the original example we had only 6 rules, but here there are 11. This is because all the rules except one (the very first one) from the example contained alternative substitutions written with a vertical line between them. In reality, however, they were always two different production rules with identical left-hand sides.

This notation using vertical strokes to separate optional right pages that have identical left pages has become a common convention. Sometimes, if it does not cause confusion, commas or even just spaces are used instead of the vertical strokes.

Here is the example from the presentation again, but now with the new rule numbers:



1.5.3 Derivability

You have probably noticed that there are several different orders in which the rules can be applied to arrive at the result shown. Let's assume that the rules were processed in this order: 1, 3, 4, 6, 9, 10.

Then, the evolution of the word from the starting word S to the final result can be represented as follows:

sentence
↓
nominal_group verb
↓
article adjective noun verb
↓
the adjective noun verb
↓
the big noun verb
↓
the big cat verb
↓
the big cat sleeps

This means for example, that the word the adjective noun verb is directly derived from the word article adjective noun verb and the big cat verb is derived from the word nominal_group verb, but not as a direct derivation, but by a finite sequence of steps.

There is a notation to express such relations between words:

If the word b can be derived from the word a by applying exactly one rule from P to a, then we write it like this:

$$a \Longrightarrow b$$

Say: "b is directly derivable from a."

(Note that the symbol $\Longrightarrow$ has many different meanings in mathematics. We will use it again in the section on logic, but there it means something quite different).

When a word is not directly derivable from another word, we write:

$$x \nRightarrow y$$

Say: "y is not directly derivable from x."

If a word is derivable from another word by applying any finite number of rules we use the Kleene star on the arrow to indicate that there exists a sequence of rules that will bring you from a to b.

$$a \Longrightarrow^* b$$

"b is derivable from a."

And the negation:

$$x \nRightarrow^* y$$

"y is not derivable from x."

Some examples:

$$\text{article adjective noun verb} \Longrightarrow \text{article adjective guitar verb}$$

$$\text{article adjective noun verb} \Longrightarrow^* \text{article adjective guitar verb}$$

$$\text{article adjective noun verb} \nRightarrow \text{a adjective guitar verb}$$

$$\text{article adjective noun verb} \Longrightarrow^* \text{a adjective guitar verb}$$

$$\text{sentence} \nRightarrow \text{sentence}$$

$$\text{sentence} \Longrightarrow^* \text{sentence}$$

but also

$$\text{verb noun verb nominal\_group} \Longrightarrow \text{verb noun verb article noun}$$

$$\text{verb noun verb nominal\_group} \Longrightarrow^* \text{plays cat sleeps the guitar}$$

1.6 Language

At last we have done all preparations to define now also the central term of formal languages: What is a language (in the world of formal languages)?

Let's define this term the mathematical way with an explanation afterwards:

Let $G = \{V, \Sigma, P, S\}$ be a grammar. This grammar produces a language $\mathcal{L}(G)$:

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

The language $\mathcal{L}$ of the grammar G is a set of words where each word is an element of the Kleene closure of Σ and where the word is derivable from S.

Let's look at this with the example:

The Kleene closure of Σ contains infinitely many words, among them are, for example, ε and a a the a a cat cat the a and the big cat sleeps.

The set of all words derivable from S is finite in our example, among them are article noun verb and article small guitar verb and the big cat sleeps.

The words, which belong to the language, are all elements, which occur in both sets, thus the intersection of these two sets. Among them are these words:

- the big cat sleeps
- a guitar plays
- the cat sleeps
- a small cat plays
- the big guitar sleeps
- a guitar sleeps

All in all there are 24 words in this language.

1.6.1 Alternative ways to define languages

Languages also can be defined by ...

- A complete list of all words
  You create a complete list of all words that appear in the language. If a word is on the list, it is part of the language. If it is not on the list, it is not part of the language. This is only possible for finite languages, i.e. languages that do not contain an infinite number of words. There is nothing more to say about this method.
- Some languages (not all) can be defined by "expressions". This is explained in detail in chapter XXX.
- An algorithm or machine that takes as input a word from the set Σ^* and then tells whether the word belongs to the language or not. The name for this class of algorithms or machines is "automaton".

1.7 Automaton

An automaton is a machine (in hardware form or simulated in software or just as a conceptual idea) that operates on a given alphabet Σ and receives an element of Σ^* (the word w) and then performs some actions in which it goes through a sequence of inner states that are elements of the set of all possible states of this automaton. When this sequence of states reaches a halting state, the automaton's work is done and it says "yes" or "no" to the question "Is w a word of the language $\mathcal{L}$ defined by the automaton?"

So when we use automata, we have neither a grammar G nor an auxiliary alphabet V, but an automaton A and a set of inner states Z and sometimes even more components.

There are different kinds of automata, and each class of automata corresponds to a class of grammars defining the same languages. The grammars differ from class to class only in the way their production rules are constrained. However, the automata show considerably greater differences from class to class.

2 Chomsky Hierarchy

Noam Chomsky (born 1928) is a famous linguist who has written more than 150 books on linguistics, politics and mass media. He is still active, especially in the field of politics. In 1956 he described a hierarchy of formal grammars and thus a hierarchy of formal languages, which has been the standard classification of languages ever since. Chomsky defined 4 hierarchies of grammars.

2.1 Overview

2.1.1 Type 0

With this type there is no restriction on the production rules of a grammar. Consequently, this class is the superclass of all other classes. Grammars of this class are called "unrestricted grammars".

The languages that can be produced by unrestricted grammars are called "recursively enumerable languages".

The class of automata that are able to detect whether a word belongs to a recursively enumerable language are called "Turing machines" which are the most powerful automata.

2.1.2 Type 1

The grammars in this class are called "context-sensitive grammars". The production rules of these grammars have some restrictions that involve signs in the neighborhood (in the "context") of other signs. This class is a subclass of type 0 and a superclass of type 2 and type 3.

The languages that can be produced by context-sensitive grammars are named "context-sensitive languages".

The class of automata that are able to tell if a word belongs to a context-sensitive language is called "Linear bounded non-deterministic Turing machines".

2.1.3 Type 2

The grammars in this class are called "context-free grammars". The production rules of these grammars have more restrictions that type-1 grammars. The production rules do not involve signs in the neighborhood of other signs. This class is a subclass of the types 0 and 1 and a superclass of type 3.

The languages that can be produced by context-free grammars are named "context-free languages"

The class of automata that are able to tell if a word belongs to a context-free language is called "Non-deterministic pushdown automaton".

2.1.4 Type 3

This is the most restricted class. Grammars of this class are called "regular grammars". They are a subclass of all other classes. Every regular grammar is either a left-regular grammar or a right-regular grammar, but for every left-regular grammar there is a right-regular grammar that produces exactly the same language, and vice versa.

The languages that can be generated by regular grammars are called "regular languages".

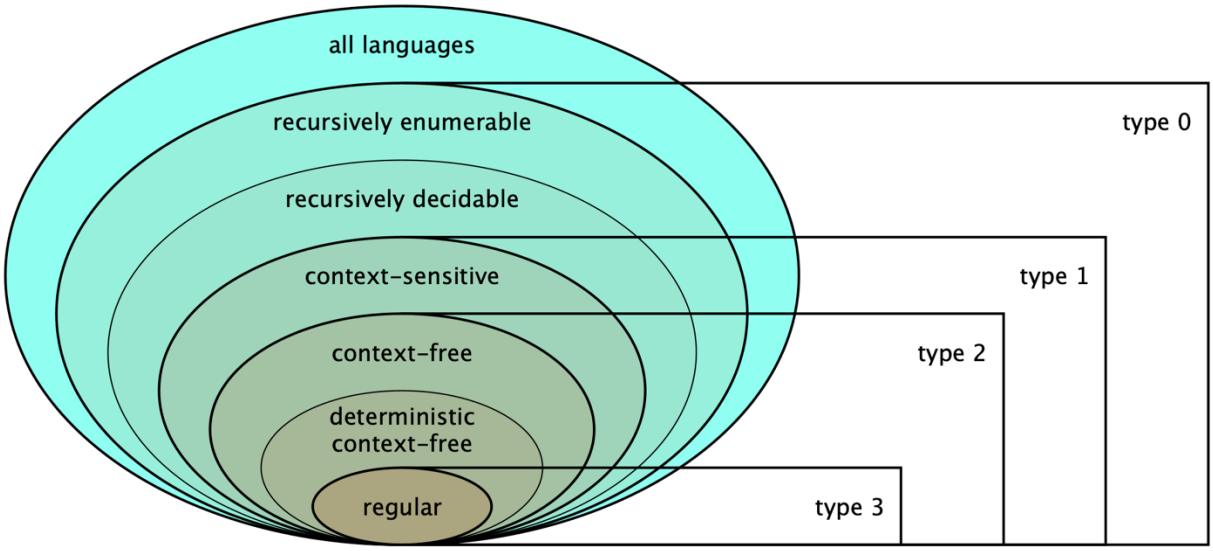
The class of automata capable of detecting whether a word belongs to a regular language is called a "finite state machine".

Closely related to regular languages are regular expressions, which describe regular languages and can be used to determine whether a word belongs to a language. (There are no expressions in the other types of the Chomsky hierarchy).

2.2 Summary

Chomsky hierarchy	Grammar	Language	Automaton	Expression
type 0	unrestricted	recursively enumerable	Turing machine	–
type 1	context-sensitive	context-sensitive	linear bounded Turing machine	–
type 2	context-free	context-free	Pushdown Automaton	–
type 3	regular	regular	Finite state machine	regular

Note, that some of the hierarchies can be subdivided further and that there are also languages outside this hierarchy which cannot be produced by any grammar. This image shows an overview of all language classes:



Language classes and Chomsky types

The class of type-2-languages is sometimes called "context-free languages" and sometimes "non-deterministic context-free languages". Both name mean the same class. "Deterministic context-free languages" is a subclass inside this class but is still is a strict superclass of type-3-languages. ("Strict" means, that there exist deterministic context-free languages that are not regular.) You will learn more about deterministic and nondeterministic automata and their associated languages later in this document.

In other chapters we will also see that type-1 automata can be deterministic and nondeterministic, but it is still an open question of research whether there is a difference between the sets of languages that can be recognized by these different classes of automata.

The difference between recursively decidable and recursively enumerable languages is of a different nature. It has nothing to do with determinism.

3 Type 3 grammars, languages, automata and expressions

As mentioned earlier, there are left-regular and right-regular grammars in this class. In this document, we will focus on right-regular grammars. You will soon see where the name comes from and you will also soon see that they are equivalent to left-regular grammars. Both types of regular grammars produce the same set of languages called "regular languages". There are no left- or right-regular languages, because when you look at a regular language, you can't tell whether it was generated by a left- or right-regular grammar. There are always grammars of both types that can generate any regular language.

There is also an algorithm that can be used to convert a left-regular grammar into a right-regular grammar that produces the same language and vice versa, but that algorithm is not shown here.

3.1 Regular grammars

Remember what a grammar is (see chapter 1.5):

A grammar is a set with 4 elements:

$$G = \{V, \Sigma, P, S\}$$

- G = a grammar
- V = the alphabet of variables (auxiliary signs = non-terminal signs)
- Σ = the alphabet of final (terminal) signs
- P = the set of production rules
- S = the start sign (actually a start word that consists of exactly 1 sign)

V and Σ are disjoint: $V \cap \Sigma = \emptyset$

S is a word consisting of exactly one sign from V (in most books written as $S \in V$; Details of this notation have been discussed in the last few paragraphs in 1.5 immediately before 1.5.1)

Each element of P is a rule, and each rule looks like this:

$$l \to r$$

with

$$l \in (V \cup \Sigma)^+$$
$$r \in (V \cup \Sigma)^*$$

All this is true for all Chomsky hierarchies. The hierarchies merely add some additional constraints to the production rules.

3.1.1 A convention that applies to all types

Before proceeding, we should define a convention that makes life much easier:

Although signs can be almost anything, talking about them and about grammars, production rules, etc., becomes much easier if we define this standard convention, which is used in almost all textbooks on formal languages:

- All elements of Σ (all elements in the alphabet of terminal signs) are lowercase Latin letters ($a, b, c, \cdots$).
- All elements of V (all elements in the alphabet of variables, i.e. all auxiliary sign) are upper case Latin letters. Among them is the sign S: ($A, B, C, \cdots, S, \cdots$).
- The starting word S is always the word containing the sign S. In the strict notation: $S = (S)$. In the convenient notation: $S = S$, where the left S is the name of the starting word (one of the 4 elements of the grammar set) and the right S is a sign from the alphabet V.
- This convention is also used in many examples, but sometimes it is also useful to use digits or other symbols in the set Σ (but not in V). This will be clearly indicated beforehand.

In the cat-guitar-example we used red and blue colors to keep terminal and non-terminal signs apart, which also helped us to handle the two alphabets. Upper and lowercase signs do exactly the same job, just without colors.

We will use this convention throughout the rest of this document (not just in Chapter 3 only).

3.1.2 Restrictions of the production rules for type-3-grammars

Let A and B be auxiliary signs (elements of V) and let a be a terminal sign (an element of Σ)

$$A, B \in V \qquad a \in \Sigma$$

And as always, ε is supposed to be the empty word. Then for right-regular grammars only these two kinds of production rules are allowed:

- $A \to \varepsilon$
  On the left side is exactly 1 auxiliary sign, on the right sign is the empty word
- $A \to aB$
  On the left side there is exactly 1 auxiliary sign, on the right side exactly 1 terminal sign followed by exactly 1 auxiliary sign in exactly this order. The reverse order would define a left-regular grammar.
  Left- and right-regular languages can produce exactly the same languages (regular languages), but as soon as you mix rules of type $A \to aB$ with $A \to Ba$ within the same grammar, the grammar is no longer regular, but context-free. This will be explained in more detail in the chapter on context-free languages.

3.1.3 Alternative definitions

In some textbooks you will find definitions where this is also allowed:

- $A \to a$
  On the left side is exactly 1 auxiliary sign, on the right side is exactly 1 terminal sign. Whenever you have a rule of the form $A \to a$, you can add an extra auxiliary sign (e.g. X) and write two new rules to replace the old rule: $A \to aX,\ X \to \varepsilon$. You need more auxiliary signs and more steps to form the words, but you get exactly the same words, and that's all that matters.
- $A \to abB, \quad A \to abcB, \quad A \to abcdB, \quad \cdots$
  On the left side there is exactly 1 auxiliary sign, on the right side there are two or more terminal signs followed by 1 auxiliary sign.
  $A \to abB$ is the same as $A \to aX,\ X \to bB$. By repeating this process, one can show that any number of terminal signs still yields exactly the same set of languages.

But none of these patterns is really necessary. The two types of rules shown at the beginning are quite sufficient to create all possible regular languages. In the further course of this chapter on regular expressions, we will make no more use of these additional types of rules.

3.1.4 Example

- $G = \{V, \Sigma, P, S\}$
- $\Sigma = \{0, 1\}$
- $V = \{\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{S}\}$
- $\mathrm{P} = \{\mathrm{S} \to 0\mathrm{S},\ \mathrm{S} \to 1\mathrm{S},\ \mathrm{S} \to 0\mathrm{A},\ \mathrm{A} \to 1\mathrm{B},\ \mathrm{B} \to 0\mathrm{C},\ \mathrm{C} \to \varepsilon\}$
- $S = S$ (This line is part of the new convention and will not be shown in future examples)

Note, that with our new convention we also could have written:

- $\mathrm{P} = \{\mathrm{S} \to \mathbf{0S} \mid \mathbf{1S} \mid \mathbf{0A},\ \mathrm{A} \to 1\mathrm{B},\ \mathrm{B} \to 0\mathrm{C},\ \mathrm{C} \to \varepsilon\}$

But then it becomes harder to talk about different rules. Instead we want to have it easy to talk about them, and for this reason we assign numbers to the rules:

$$1\ \ \mathrm{S} \to 0\mathrm{S}$$

$$2\ \ \mathrm{S} \to 1\mathrm{S}$$

$$3\ \ \mathrm{S} \to 0\mathrm{A}$$

$$4\ \ \mathrm{A} \to 1\mathrm{B}$$

$$5\ \ \mathrm{B} \to 0\mathrm{C}$$

$$6\ \ \mathrm{C} \to \varepsilon$$

One of the first things you should notice is this: As you apply more and more rules of the type $A \to aB$ (rules 1 through 5 in the example), the resulting word grows by exactly 1 sign per rule, and it always contains exactly 1 variable, and that variable is always the rightmost sign in the word. So the word never shrinks and never keeps the same length. It grows at a constant rate and it only grows at its **right** end. For this reason, this grammar is called "**right**-regular".

This behavior is caused by rules that all have the form $A \to aB$. So it is not just a property of this example, but of all right-regular grammars.

Then you may also have noticed that the growth stops immediately (the word even shrinks by 1 sign) as soon as you apply rule 6, which is of type $A \to \varepsilon$. After applying a rule of this type, all variables are gone from the word and only signs from the alphabet Σ remain. Thus, it is a word w that is a subset of Σ^*, and it is a word derivable from S:

$$w \in \Sigma^* \quad \wedge \quad w \Longrightarrow^* S$$

And this means, that the resulting word is a word of the language $\mathcal{L}$ that is produced by this grammar G:

$$w \in \mathcal{L}(G)$$

Let's summarize:

- If you apply a rule of type $A \to aB$, the word grows by a new terminal sign at its right end, and a non-terminal sign always remains at the right end. The resulting word is therefore not a word of the language.
- If you apply a rule of the type $A \to \varepsilon$, the non-terminal sign at the right end disappears, and the resulting word is a word the language. After that it is not possible to apply any more rules.

In the concrete example, this is also noticeable: At the beginning, we can apply rules 1 and 2 in any order and as often as we want. Each time we simply add a 0 or a 1 to the right end of the word, immediately before the variable S. As long as this S exists, we cannot apply rules 4 to 6. But as soon as we apply rule 3, from this point on, the exact order of all further rules is exactly predetermined. It reads:

- Rule 3 adds a 0
- Rule 4 adds a 1
- Rule 5 adds a 0
- Rule 6 terminates the process.

So we get words that start with a prefix of any length consisting of the signs 0 and 1, which can appear in this prefix in any arbitrary order. The last 3 signs of the sequence must always be the signs 010.

So this grammar will produce a language that contains these words:

$$\mathcal{L}(G) = \{010, 0010, 1010, 00010, 01010, 10010, 11010, \cdots\}$$

This language contains all words ending with the sequence of signs 010.

3.2 Regular expressions

The language from the previous example can be encoded in this way:

$$\mathcal{L} = (0|1) * 010$$

This expression is a kind of abbreviation for all possible words in the language, and since this only works for regular languages, this kind of notation is called a "regular expression".

What the individual components in this string mean, along with symbols that are not used in this example, is in the following list:

- $\emptyset$ The empty set symbol denotes the empty language (the language that contains no words). It cannot be combined with other symbols shown here.
- $(\cdots)$ Brackets define the scope of other operators.
- ab Concatenation. The invisible binary operator between a and b joins the operand on the left (which is a word) and the operand on the right (also a word) to form a new and longer word.
- $a \mid b$ Option. This binary operator returns either its left or its right operand (never both).
- $\cdot$ Wildcard. This 0-ary operator stands for any singular word that can be formed from any terminal sign in Σ. It is short for $(a|b|c|\cdots)$ where between the brackets are all the one-sign-words that can be created from elements of Σ.
- $a*$ The zero-or-more quantifier. This unary operator returns any number of copies of its left operand. This can be 0 copies (returns ε) or one (returns a) or 2 (returns aa) or 3 (returns aaa) or any other natural number of copies. It works similarly to the Kleene star, which is why the star was chosen as the symbol.
- $a+$ The one-or-more quantifier. Works like the zero-or-more quantifier, but returns at least one copy of a. Works similarly to the positive closure operator, so it is a $+$.
- $a?$ The zero-or-one quantifier. This unary operator returns either its left operand or the empty word. So it is a short form for $a \mid \varepsilon$.

Note that each regular expression by itself is also a word, and there is a language that contains exactly all possible regular expressions, but this language of regular expressions is not a regular language! It is a context-free language, and we will see it there again as an example of context-free languages.

Regular expressions as tools for pattern matching can be found in almost every programming language. But all of these implementations contain elements that go beyond the above description. Some of them serve only to make the notation more convenient. For example, in all implementations there are sign classes written in square brackets:

$$[0123456789] := (0|1|2|3|4|5|6|7|8|9)$$

This notation with square brackets is fully compatible with regular expressions as defined above.

But you can make it even simpler by taking advantage of the fact that real implementations of alphabets on computers always have an inner order. And if there is an inner order, you can use intervals written as follows:

$$[0-9] := (0|1|2|3|4|5|6|7|8|9)$$

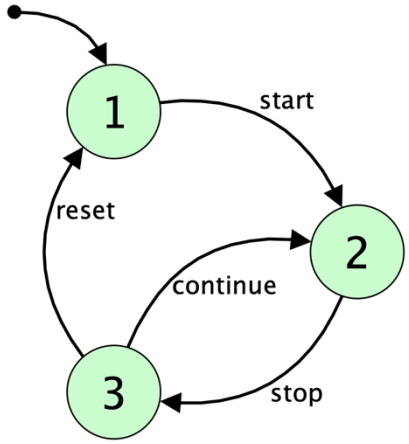
This notation is no longer compatible with our definitions because there is no internal order of the signs in an alphabet in our definitions. But if you add such a definition, then you could also use this notation and would still describe the same class of languages: only regular languages. And for some intervals there are standard names starting with a backslash:

$$\backslash d := (0|1|2|3|4|5|6|7|8|9)$$

This notation does not require any inner ordering, so it is again fully compatible with our definitions.

Real implementations of regular expressions, however, often include extensions such as backreferences that are no longer compatible with our definition because they describe languages that are no longer regular. (A back reference is a kind of memory that allows a piece of variable text to be found again later in the same word).

3.3 Finite State Machines (FSM)



State diagram of a simple stopwatch

This is the state diagram of a simple stopwatch. The green circles indicate three states:

1 The clock is not running, it displays 0:00.

2 The clock is running, the display changes constantly.

3 The clock is not running. The display shows the time when the stop button was pressed.

The arrow coming from the small black dot indicates the initial state, and all other arrows indicate actions performed by a human user.

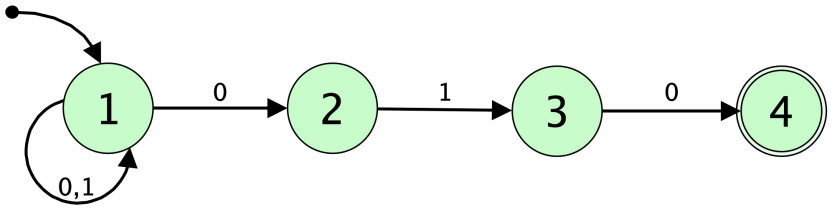
A finite state machine is very similar to such a state diagram. The text on the arrows does not indicate actions performed by a human user, but signs "consumed" by the machine. And there are two types of states:

The border with a single line indicates a non-accepting state

The border with a double line indicates an accepting state

You will soon learn what accepting and non-accepting states are.

3.3.1 An Example

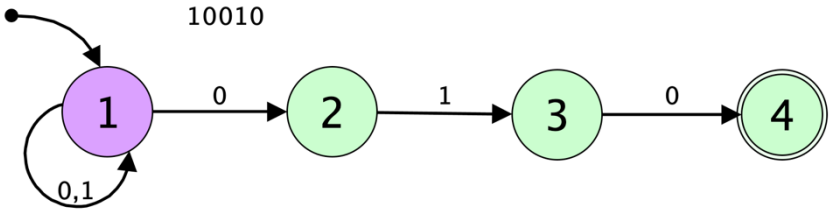


A finite state machine

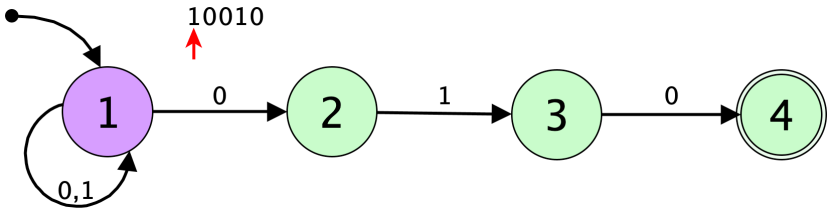
The task of each state machine is to read words sign by sign and then switch to a new state depending on the current state and the sign read. Let us now assume that the word the automaton is to process is the following word:

10010

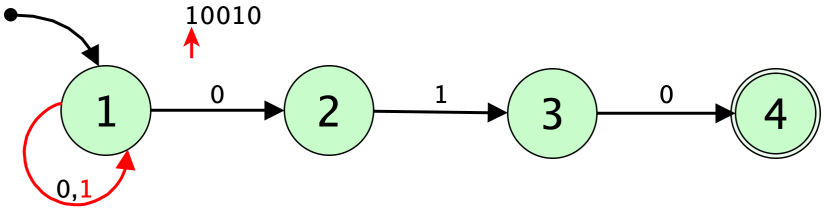
At the beginning (before the first sign is read in), the machine is in the state pointed to by the arrow that starts from the small black dot. In the example, this is state 1, which is therefore displayed in a different color.



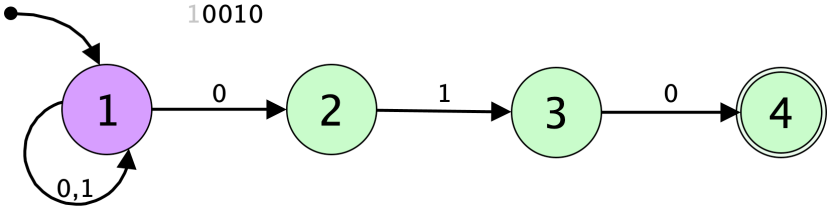
In this state, the machine reads the first (leftmost) sign of the word, which is the sign 1.



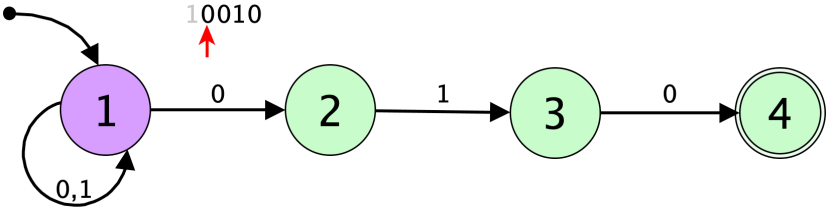
There is an arrow starting from the state 1, next to which there is the sign 1. This arrow indicates to which state the automaton must now change.



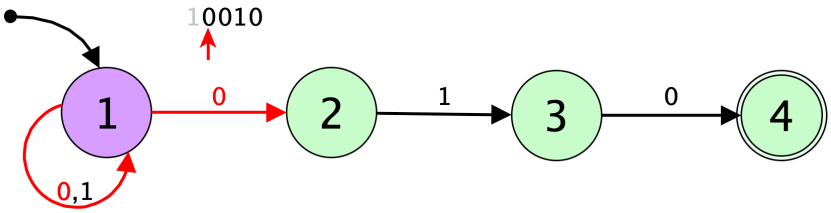
The automaton is then again in state 1, but the first sign of the word is now consumed.



Now the automaton reads the next sign, it is the sign 0:



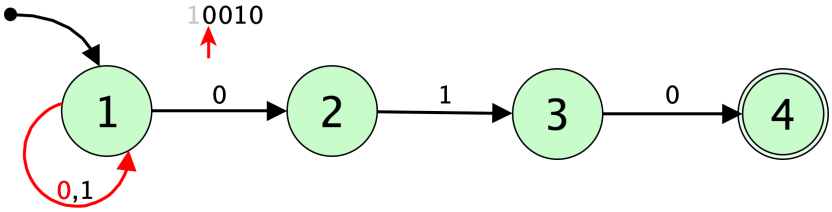
Now we find two arrows which start from the active state and which are labeled with the read sign:



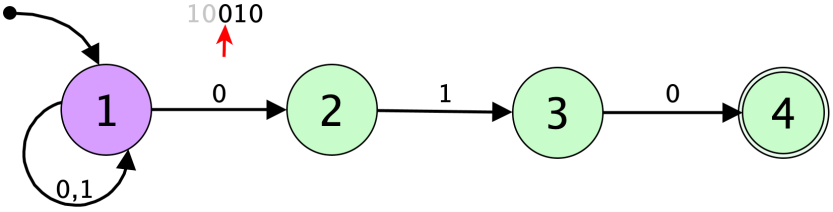
So now there are two different possibilities which lead to two different states. This is typical for this kind of automatons, which are called "non-deterministic" for obvious reasons. What exactly this means will be explained in more detail in the following section.

For now, however, we need someone to help us make the right decision. This should ideally be someone who can see into the future and already knows where which decisions will lead, and who can tell us which of the possible decisions is the right one to end up in an accepting state. This magic being exists in every non-deterministic automaton. It is called "oracle". This will also be explained in more detail in a moment.

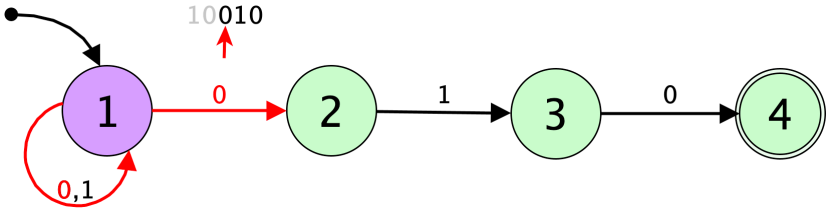
This oracle tells us in this example that it is best to follow the path to state 1, so we do just that:



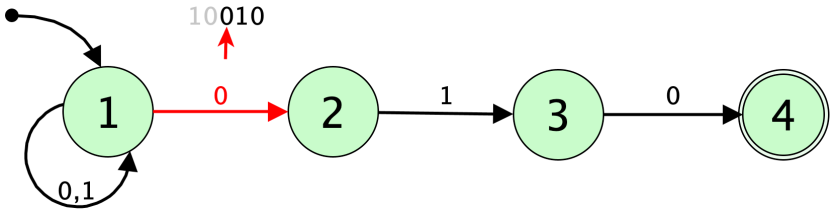
Again we reach state 1, but now we have already consumed the second sign and are about to read in the third sign.



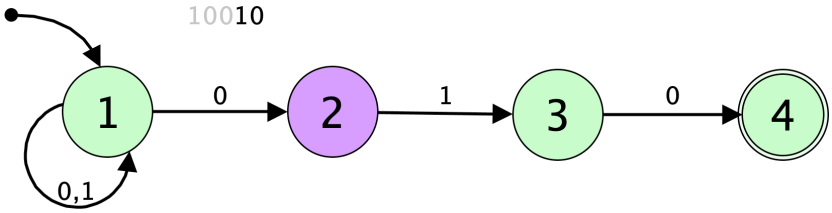
Again we have 2 possibilities:



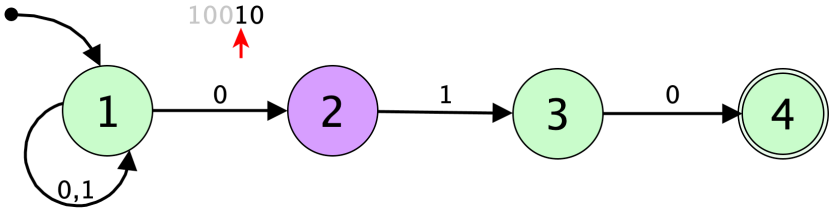
But this time the oracle says: "Go to state 2":



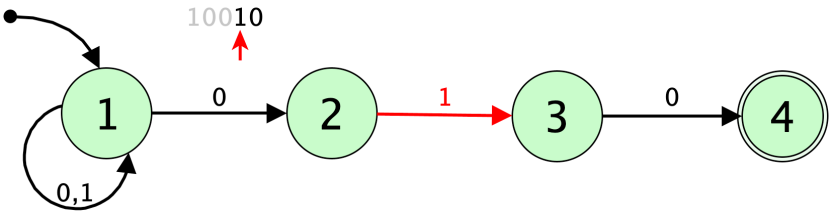
The third sign is now used up

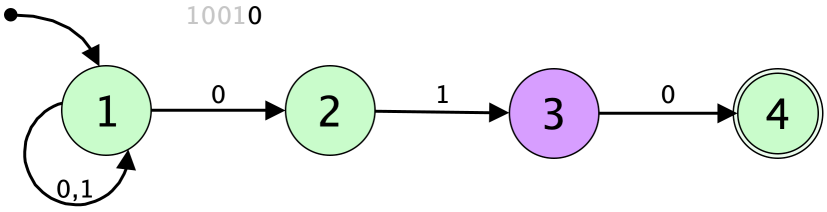


And the automaton reads in the fourth sign

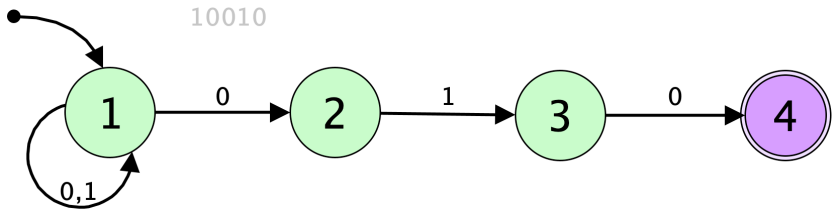


Now we are lucky that the read in sign corresponds to an arrow which starts from the active state and is labeled with the same sign. Because if we had read in the sign 0, there would be no arrow to follow. This is also something that is often found in non-deterministic automatons. This will be discussed in more detail below. For now, we are happy that the signs match. We follow the only option we have:





It should now be clear that the last sign leads us straight to state 4.



At this point there is nothing more to read in. If the automaton is in an accepting state after reading in all the signs, it means that it accepts the consumed word. And this, in turn, means that this word belongs to the language recognized by this automaton.

$$10010 \in \mathcal{L}(A)$$

What could have gone wrong?

Case 1: 1111

For example, we could read in the word 1111. In this case, the automaton can never leave state 1. So when the word is over, the automaton is still in state 1. But this is not an accepting state, so the word 1111 is not accepted by this automaton.

$$1111 \notin \mathcal{L}(A)$$

Case 2: 000000

We could read in the word 000000. As long as we are in state 1, we have to ask the oracle for every sign we read. But the oracle can't say "It's hopeless, give up", that's not in its power. It will always name a way, because it can do nothing but choose one from the available possibilities. It may be that each time it tells us, "Go to state 1." Then we can read the word to the end, but then stand in a non-accepting state.

But the oracle could also send us to state 2 on the first or second sign. But there the automaton reads the sign 0 as the next sign and then has no arrow to follow. In this case, the currently active sign (and all following signs) are considered as not read in. So the automaton can't even finish processing the word until the last sign.

This means that it cannot accept the word. It does not matter whether the automaton is in an accepting or a non-accepting state at that moment. The mere fact that the word cannot be read to the end leads to rejection.

$$000000 \notin \mathcal{L}(A)$$

Case 3: 0100

Again, the oracle could keep us at state 1 until the end, but it could also send us on to state 2 right at the first sign. After the first 3 signs are read, we are at the end of the automaton, at state 4, and this is an accepting state. But the word still contains a fourth sign, and there is no arrow leading away from state 4 at all. No matter if the fourth sign is 0 or 1, it cannot be processed and thus the word is not accepted, although the termination happens in an accepting state.

$$0100 \notin \mathcal{L}(A)$$

3.3.2 Structure of a non-deterministic finite state machine (NFSM)

That the automaton from the example is non-deterministic has already been mentioned. But before we explain what that means exactly, let's analyze the automaton at hand in a little more detail.

We have, of course, a set of states, including a start state, and some of the states are accepting states. The automaton needs to process individual signs. That means it needs an input alphabet. And then, of course, there are the state transitions. At each transition, a state (namely, the currently active state) together with a sign (namely, the currently read sign) determines what the next state should be, and there may well be several alternative subsequent states, or none at all.

Let us write this down formally:

A is a non-deterministic finite state machine (NFSM), and every NFSM is a set with exactly 5 elements:

$$A = \{Z, Z_0, E, \Sigma, \delta\}$$

- Z is the set of all states (represented as nodes in the state graph)
- Z_0 is the initial state $Z_0 \in Z$ (In the state graph marked by an arrow coming from a small dot. This dot is not a state, it only marks the initial state)
- E is the set of all accepting states. $E \subseteq Z$
- Σ is an alphabet. It is the same alphabet that contains the terminal signs in a grammar. So it is the alphabet that makes up the words in the language.
- δ is a relation: $\delta \subseteq Z \times \Sigma \times Z$ It takes as "input" an element from the Cartesian product $Z \times \Sigma$ (a combination of state and sign) and produces as "output" a subset of Z which can also be empty. In the state graph, this relation is the set of all arrows together with their labels.

You can also define nondeterministic automatons with multiple initial states. Then Z_0 is not a single state, but a set containing several states, with $Z_0 \subseteq Z$. However, we will not discuss this possibility further in the following.

What is the mathematical description of the automaton from the example?

- $A = \{Z, Z_0, E, \Sigma, \delta\}$
- $Z = \{①, ②, ③, ④\}$
- $Z_0 = ①$
- $E = \{④\}$
- $\Sigma = \{0,1\}$
- $\delta = \{(①, 0, ①),\ (①, 1, ①),\ (①, 0, ②),\ (②, 1, ③),\ (③, 0, ④)\}$

3.3.3 Connection grammar - automaton

It will have been noticed that in the example shown before, the repeated visit to the same state followed by the final goal-directed journey to the end point bears great resemblance to the fact that in the grammar example we could stay in state S as long as we wanted and always ended up generating the three signs 010.

It is indeed the case that the automaton presented here accepts exactly the words that the grammar from the previous example can generate. And it is also the case that the automaton can exactly not accept those words that the grammar cannot generate. So automaton and grammar belong together, because the automaton recognizes exactly the language that the grammar generates:

$$\mathcal{L}(A) = \mathcal{L}(G)$$

There is actually a simple algorithm that can be used to create the automaton from the grammar, and a similar algorithm allows to make a grammar from an automaton.

Making a finite state machine out of a regular grammar.

Let's look again at the grammar from the example:

- $G = \{V, \Sigma, P, S\}$
- $\Sigma = \{0, 1\}$
- $V = \{\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{S}\}$
- $P = \{\mathrm{S} \to 0\mathrm{S},\ \mathrm{S} \to 1\mathrm{S},\ \mathrm{S} \to 0\mathrm{A},\ \mathrm{A} \to 1\mathrm{B},\ \mathrm{B} \to 0\mathrm{C},\ \mathrm{C} \to \varepsilon\}$
- $S = \mathrm{S}$

It is obvious that the alphabet of the automaton must be the terminal alphabet of the grammar:

$$\Sigma_G \rightarrow \Sigma_A = \{0, 1\}$$

The alphabet of the auxiliary signs becomes the set of states (we just have to change the names of the states later)

$$V \rightarrow Z = \{\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{S}\}$$

The start state of the automaton is the start sign (or the start word) of the grammar

$$S \rightarrow Z_0 = \mathrm{S}$$

The set of accepting states are all auxiliary signs for which there are production rules that transfer them to the empty word

$$E = \{\mathrm{C}\}$$

The transition relation consists of all production rules that do not result in an empty word. They must be noted only other:

$P = \{\mathrm{S} \rightarrow 0\mathrm{S},\ \ \mathrm{S} \rightarrow 1\mathrm{S},\ \ \mathrm{S} \rightarrow 0\mathrm{A}\,,\ \mathrm{A} \rightarrow 1\mathrm{B},\ \ \mathrm{B} \rightarrow 0\mathrm{C},\quad \mathrm{C} \rightarrow \varepsilon\}$

$\delta = \{(\mathrm{S}, 0, \mathrm{S}),\ (\mathrm{S}, 1, \mathrm{S}),\ (\mathrm{S}, 0, \mathrm{A}),\ (\mathrm{A}, 1, \mathrm{B}),\ (\mathrm{B}, 0, \mathrm{C})\}$

This is actually the end of the algorithm, because we now have a complete non-deterministic automaton. If we want to compare it with the automaton discussed in detail before, we only have to name the states differently:

$$\mathrm{S} \rightarrow ①, \qquad A \rightarrow ②, \qquad B \rightarrow ③, \qquad C \rightarrow ④$$

However, the changed names do not change the function of the automaton.

The existence of the algorithm just shown proves that for every right-regular grammar there is a non-deterministic finite state machine.

Making a regular grammar out of a finite state machine.

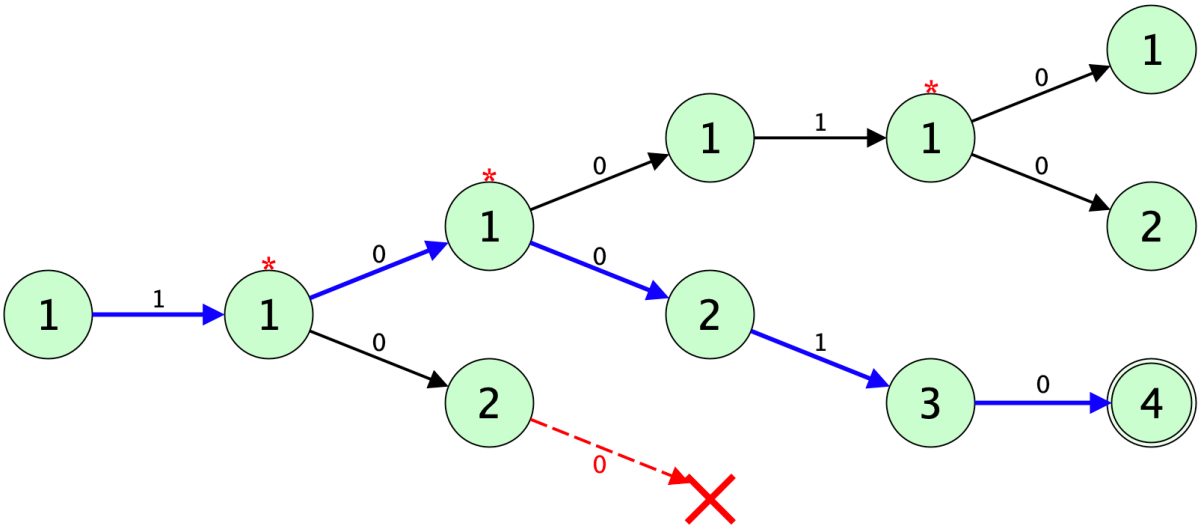
However, the same algorithm can also be run in the reverse order, and in this way one can make a right-regular grammar out of every finite state machine. This works for both nondeterministic and deterministic finite state machines because, as we will see in a moment, the set of all deterministic finite state machines (DFSM) is a subset of the set of nondeterministic finite state machines (NFSM).

A small change in the notation of the relation δ also makes it possible to generate a left-regular grammar from any finite automaton using the same algorithm. This proves the claim made at the beginning that left-regular and right-regular grammars produce the same languages.

3.4 Determinism

Let's talk about the oracle.

Let's stay with the same automaton from before and let it analyze the word 10010 again. But instead of representing the whole automaton step by step again and again, we represent only the sequence of states that the automaton takes. If there are several possibilities at certain points (this is exactly the case when the oracle has to be consulted) then all possible subsequent states are to be displayed, which leads to a branching of the path, i.e. to a tree:



This tree shows all possible processing paths of the automaton for the word 10010. All places where the oracle must be consulted are marked with a small red star above the state symbol. The only path that leads to an accepting state is the blue path. One can interpret the processing that happens in a nondeterministic automaton in two ways:

3.4.1 Oracle

The oracle knows the complete processing tree in advance. In the course of the actual processing, when the automaton comes to a point where the tree branches into several branches, it asks the oracle. The oracle answers with a branch of the tree that contains a path leading to an accepting state. If there are several such branches, the oracle chooses one at random from them. If there is no branch leading to an accepting state at this point, the oracle randomly selects one from the available branches. The automaton follows this selection.

As a result, the automaton is guaranteed to reach an accepting state whenever there is at least one path leading from the initial state to an accepting state. Only if there is no such path at all, the automaton will not accept the word.

3.4.2 Parallel processing

The oracle would have to be able to know a result before the entire algorithm leading to the result is fully processed. Such a clairvoyant being is therefore a nice thought experiment, but can neither be built as hardware nor programmed as software. The now following approach, is at least to some extent realizable by software:

Every time the computer program simulating the automaton comes to a place where the oracle would have to be consulted, it creates as many copies of running process as there are branches at the current place. In each branch, it then continues to work on a different subsequent state.

In the present example, the program thus splits into a total of 4 copies. One of the four processes cannot finish reading the word, two others read it completely but then end up in non-accepting states. Only one copy reaches an accepting state at the end. This is perfectly sufficient. As soon as even a single process copy reaches an accepting state after reading all the signs, the word is considered accepted. Only if this goal is not reached in any single copy, the word is rejected.

However, this approach has a problem: it can only simulate automata with small alphabets, and even that only if they process short words. Automatons with very large alphabets and very many states that process very long words run into limits due to the limited resources of real computers. For any conceivable computer, regardless of how much memory it has, one can construct a regular language whose resources are insufficient to recognize all words.

3.4.3 Structure of a deterministic finite state machine (DFSM)

We would like that from each state of the automaton for each sign from the alphabet exactly one single arrow leads to exactly one subsequent state. Then it would be clear for every combination of state and sign which state comes next. Then there would be no need for an oracle and the automaton could not get stuck in the middle of a word, because it stands at a state, from which there is no progress for the read word.

Such an automaton is called: deterministic.

Does this sound familiar? "Each element from the source set is assigned exactly one element from the target set" - That is the definition of a function! And in fact, the mathematical structure of a deterministic FSM differs from a non-deterministic one only in the fact that a (total) transition function is used instead of the transition relation.

This then looks like this:

A is a deterministic finite state machine (DFSM), and every DFSM is a set with exactly 5 elements:

$$A = \{Z, Z_0, E, \Sigma, \delta\}$$

Z	is the set of all states (represented as nodes in the state graph)
Z_0	is the initial state $Z_0 \in Z$ (In the state graph marked by an arrow coming from a small dot. This dot is not a state, it only marks the initial state)
E	is the set of all accepting states. $E \subseteq Z$
Σ	is an alphabet. It is the same alphabet that contains the terminal signs in a grammar. So it is the alphabet that makes up the words in the language.
δ	is a function: $\delta\colon Z \times \Sigma \to Z$ It takes as "input" an element from the Cartesian product $Z \times \Sigma$ (a combination of state and sign) and produces as "output" exactly one element of Z. In the state graph, this function is the set of all arrows together with their labels.

Anything that has changed from the definition of an NFSM is highlighted in red. Everything in black has remained the same.

This sounds quite nice, but is there any way to turn a given nondeterministic finite automaton into a deterministic one?

3.4.4 Converting a NFSM into a DFSM

Yes, you can!

Powerset construction

Let's look at the same automaton again, but this time only at the transition relation, which we now show as a table. This relation maps elements from the Cartesian product of the state set and the alphabet to subsets of the state set. So let's draw the Cartesian product as a table and enter the resulting subsets into the table. The accepting states are highlighted in red. (In the example, this is only the state ④).

NFSM	0	1
①	{①, ②}	{①}
②	{ }	{③}
③	{④}	{ }
④	{ }	{ }

Now we create a new table for the DFSM, which first contains only that row of the original relation which starts from the initial state (i.e. from ①). We also write the state in the left column as a set:

DFSM	0	1
$\{①\}$	$\{①,②\}$	$\{①\}$

The state ①, which is now called $\{①\}$, has already been completely processed. Now we add a new line for each state that has been added, with the new state on the left. In the example, there is only one new state, which is $\{①,②\}$:

DFSM	0	1
$\{①\}$	$\{①,②\}$	$\{①\}$
$\{①,②\}$		

The new state is a combination of the states ① and ②. Therefore, we look in the original (green) NFSM table to see where we get if we read a 0 at ① (we get to $\{①,②\}$) and if we read a 0 at ② (we get to the empty set $\{\}$).
Unites these two sets: $\{①,②\} \cup \{\} = \{①,②\}$. The resulting set $\{①,②\}$ is then entered into the table:

DFSM	0	1
$\{①\}$	$\{①,②\}$	$\{①\}$
$\{①,②\}$	$\{①,②\}$	

Now you look to see what you get when you read 1 at ① and ② each. That is $\{①\}$ and $\{③\}$. The union is $\{①,③\}$:

DFSM	0	1
$\{①\}$	$\{①,②\}$	$\{①\}$
$\{①,②\}$	$\{①,②\}$	$\{①,③\}$

This means that the state $\{①,②\}$ has been completely processed, and all newly added sets are now written on the left as a new line:

DFSM	0	1
$\{①\}$	$\{①,②\}$	$\{①\}$
$\{①,②\}$	$\{①,②\}$	$\{①,③\}$
$\{①,③\}$		

Same game as before: Check the original NFSM table to see where ① und ③ go when you read a 0.
This gives $\{①,②\}$ and $\{④\}$. The union of the two sets is $\{①,②,④\}$.

DFSM	0	1
{①}	{①, ②}	{①}
{①, ②}	{①, ②}	{①, ③}
{①, ③}	{①, ②, ④}	

① and ③ lead to {①} und { } when reading 1, which united results in {①}:

DFSM	0	1
{①}	{①, ②}	{①}
{①, ②}	{①, ②}	{①, ③}
{①, ③}	{①, ②, ④}	{①}

Again, a new state has been added, leading to a new line:

DFSM	0	1
{①}	{①, ②}	{①}
{①, ②}	{①, ②}	{①, ③}
{①, ③}	{①, ②, ④}	{①}
{①, ②, ④}		

Now the results of the NFSM states ①, ② and ④ must be determined and combined when reading 0: {①, ②} ∪ { } ∪ { } = {①, ②}:

DFSM	0	1
{①}	{①, ②}	{①}
{①, ②}	{①, ②}	{①, ③}
{①, ③}	{①, ②, ④}	{①}
{①, ②, ④}	{①, ②}	

What do ①, ② and ④ result in when reading 1? It is {①} ∪ {③} ∪ { } = {①, ③}

DFSM	0	1
{①}	{①, ②}	{①}
{①, ②}	{①, ②}	{①, ③}
{①, ③}	{①, ②, ④}	{①}
{①, ②, ④}	{①, ②}	{①, ③}

With this you are actually done. However, the names of the new states are a bit bulky, so it is recommended to rename them, for example like this:

$$\{①\} \rightarrow ❶$$
$$\{①, ②\} \rightarrow ❷$$
$$\{①, ③\} \rightarrow ❸$$
$$\{①, ②, ④\} \rightarrow ❹$$

The state ❹ is an accepting state because there is an accepting state in the set $\{①, ②, ④\}$. All other new states are non-accepting. This is generally true: as soon as a set contains an accepting state, the whole set is accepting.

The new table now contains exactly one subsequent state in each cell. The relation shown in it is therefore a function and looks like this:

DFSM	0	1
❶	❷	❶
❷	❷	❸
❸	❹	❶
❹	❷	❸

And the whole deterministic finite state machine turns out like this in the end:

$A = \{Z, Z_0, E, \Sigma, \delta\}$

$Z = \{❶, ❷, ❸, ❹\}$

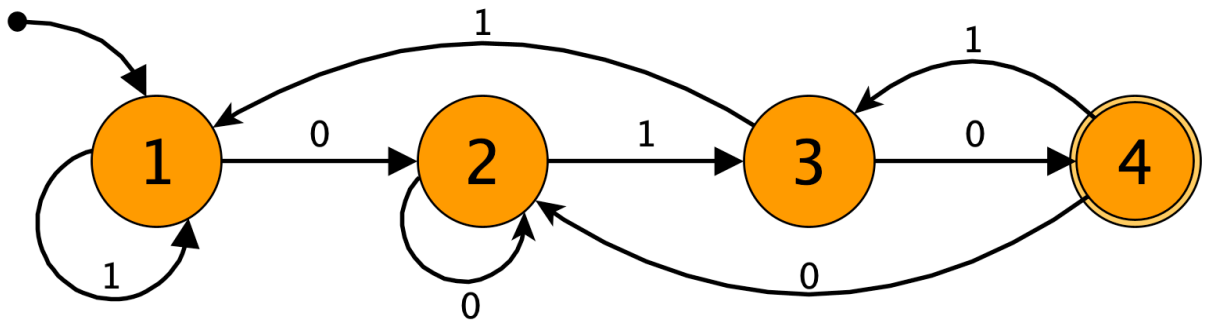
$Z_0 = ❶$

$E = \{❹\}$

$\Sigma = \{0,1\}$

$$\delta = \begin{Bmatrix} (❶, 0) \mapsto ❷, & (❶, 1) \mapsto ❶, \\ (❷, 0) \mapsto ❷, & (❷, 1) \mapsto ❸, \\ (❸, 0) \mapsto ❹, & (❸, 1) \mapsto ❶, \\ (❹, 0) \mapsto ❷, & (❹, 1) \mapsto ❸ \end{Bmatrix}$$

If you draw this information as a graph, the DFSM looks like this:



If you now try to process different words with this automaton, you will find out that this automaton also recognizes exactly the same language as the original automaton. However, this deterministic automaton gets along completely without oracle and never reaches a state where there is no progress for a read sign.

The existence of this conversion algorithm proves that any NFSM can be converted into a DFSM that recognizes the same language. Therefore, the set of all languages recognized by NFSMs must be a subset of the set of all languages recognized by DFSMs.

$$\mathcal{L}(\text{NFSM}) \subseteq \mathcal{L}(\text{DFSM})$$

On the other hand, the set of all DFSMs is a subset of the set of all NFSMs. This is because an NFSM may well contain states which do not require oracles at all, and where there is exactly one arrow to the successor state for each sign read. And every NFSM, which contains exclusively such states, is still a NFSM, but also a DFSM. And for this reason, the set of all languages that can be recognized by DFSMs must be a subset of all languages that are recognized by NFSMs.

$$\mathcal{L}(\text{DFSM}) \subseteq \mathcal{L}(\text{NFSM})$$

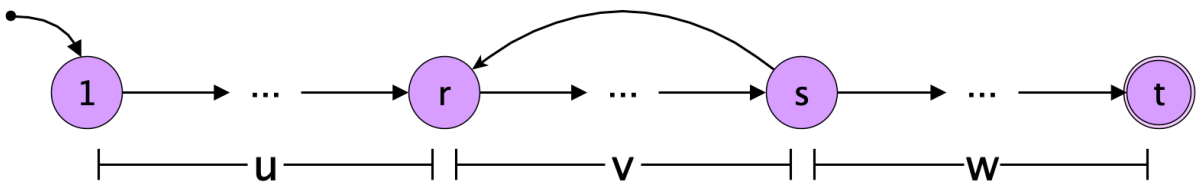
Thus, both sets of languages are subsets of each other, which means that the two sets are identical. The set of deterministic regular languages is equal to the set of non-deterministic regular languages.

$$\mathcal{L}(\text{NFSM}) \subseteq \mathcal{L}(\text{DFSM}) \;\wedge\; \mathcal{L}(\text{DFSM}) \subseteq \mathcal{L}(\text{NFSM}) \quad \Longrightarrow \quad \mathcal{L}(\text{DFSM}) = \mathcal{L}(\text{NFSM})$$

3.5 Pumping lemma

Every regular language has (at least) one finite state machine that recognizes it. Each FSM has only a finite number of states. However, besides finite regular languages, there are also regular languages that contain an infinite number of words. This is only possible if the length of the words is not limited by any upper bound. In particular, in such languages it is inevitable that there are words which contain more signs than the FSM belonging to this language has states.

This in turn makes it imperative that the FSM must pass through a certain sequence of states several times when recognizing such words.



Thus, at the beginning and at the end of a word whose length is greater than the number of states in the automaton, there may be a prefix u and a postfix w, which have a finite length that can also be 0, and in between a part v that must repeat.

For short words, v may be absent or present only once; for longer words, v must necessarily repeat several times.

That is, if there is a word z in the language $\mathcal{L}$ where at least one node of the FSM is traversed more than once, then this word can be decomposed into the three subwords u, v and w ($z = uvw$) such that the cycle of states in the EA corresponds exactly to the subword v.

However, an FSM has no memory, in particular it has no loop counter, and therefore there is no mechanism in an FSM that causes this loop to be exited after a certain number of runs.

Thus, if there is a word with the above properties, then this word can be pumped up with as many copies of v as desired. Because then these words also belong to the language:

$$uvw, uvvw, uvvvw, uvvvvw, uvvvvvw, \cdots$$

or generally uv^iw where v^i is the i-fold concatenation of v with itself, thus $v^1 = v$; $v^2 = vv$; $v^3 = vvv$ etc.

Now if there is a language which contains words longer than the number of states, and if these words do not contain a part v ($v \neq \varepsilon$) which can be pumped up arbitrarily, then this language cannot be regular. And such languages do exist.

A simple example is the language a^nb^n where a^n again means $a, aa, aaa, aaaa, \cdots$.

So this language contains all words of the form $ab, aabb, aaabbb, aaaabbbb, \cdots$

If a longer word of this form (longer than the number of states in the FSM) is broken down into the three parts uvw, then v may contain either exclusively a's or exclusively b's, or it may consist of at least one a followed by at least one b.

In the first two cases, pumping up results in increasing only the number of a's or only the number of b's, respectively, while the number of other signs remains unchanged. The requirement that there be an equal number of signs of each kind in the word is therefore violated.

$$aaaabbbb = a|aa|abbbb \qquad u = a, v = aa, w = abbbb$$

$$uvvw = a|aa|aa|abbbb = a^6b^4 \neq a^nb^n$$

If, on the other hand, v is chosen to contain each of the two signs at least once, then pumping it up causes b's to suddenly appear in front of a's in the word

$$aaaabbbb = aaa|ab|bbb \qquad u = aaa, v = ab, w = bbb$$

$$uvvw = aaa|ab|ab|bbb = a^4bab^4 \neq a^nb^n$$

It follows that the language a^nb^n is not regular. However, the formula a^nb^n, albeit in a modified form, would always be needed if a language is to contain the same number of signs of the type

a as of the type b. However, this is exactly the case when opening and closing parentheses must occur equally often, as is the case with regular expressions, for example.

In a similar way it can be proved that the language of palindromes ww^R is not regular. Here w^R means the reflected form of w. So, for example, if w is $abcde$, then $w^R = edcba$. And the palindrome is then $abcdeedcba$.

3.5.1 Why you can't mix left-regular and right-regular rules

Finally, it should be explained why in regular grammars rules of type $A \to aB$ (right-regular rules) may not be mixed with rules of type $A \to Ba$ (left-regular rules).

Let's assume that this would be allowed in regular grammars, then this grammar would be allowed too:

- $G = \{V, \Sigma, P, S\}$
- $\Sigma = \{\mathrm{a}, \mathrm{b}\}$
- $V = \{A, S\}$
- $P = \{S \to aA,\ A \to Sb,\ S \to \varepsilon\}$

Let's number the rules:

$$1\ \ S \to aA$$

$$2\ \ A \to Sb$$

$$3\ \ S \to \varepsilon$$

We could start with rule 3, in which case we generate the empty word ε and are already done.

Or we start with rule 1, which gives aA, which forces us to apply rule 2, which leads to aSb. Rule 3 aborts the process and we get ab.

Instead of aborting, however, we could apply rule 1 again, then rule 2 again, and then rule 3. Then we get $aabb$.

Or we decide to apply, say, rule 1 eight times (and, forcibly, rule 2 just as many times) before coming to an end with rule 3. Then we get $aaaaaaaabbbbbbbb$ or, using the simplified notation from earlier: a^8b^8. No matter how many times we decide to use rule 1, we always end up with an expression of the form a^nb^n.

However, we just proved in the previous chapter with the pumping lemma that this is not a regular language. And for this reason, one must not use left- and right-regular rules together in regular grammars.

3.6 Practical applications

Tokenizers

Tokenizers or lexical scanners are computer programs that read unformatted text (e.g. the source code of a computer program) and parse it into logically related units called tokens. Tokenizers are part of many compilers, but also of many parsers, which try to recognize and evaluate structures in texts (e.g. html code).

The rules according to which tokenizers work are defined by regular grammars, and the realization is done with the help of deterministic finite state machines, which are automatically generated from these grammars via the intermediate step of a non-deterministic finite state machines.

Examples include JFlex and AnnoFLex, which both tokenize Java programs, and Flex, re2c, and Quex, which do the same for C and C++ programs.

Pattern matching

Almost all high-level programming languages contain regular expressions that can be used to determine whether a string contains a certain pattern. One can also extract the matching digits from the string or replace these digits with other text.

However, many realizations of regular expressions used in practice also contain the possibility to search for a concrete expression of a pattern found at one place at another place by means of backward references. This operation requires a memory that is not present in the theoretical description of regular expressions and makes it possible to recognize languages that are more complex than real regular languages.

But regular expressions are also used outside of programming languages, for example in bioinformatics. There they are used to search for protein molecules in protein databases.

Model Checking

Model checking is an elaborate automaton test procedure in which a mathematical-logical model of a computer program or logic component is compared with a formal specification.

The formal specification is usually in the form of logical expressions whose variables are bound to the model. The model satisfies the specification if all logical expressions that make up the specification are true.

The model in turn can be described by a finite state machine.

In evaluation, Büchi automata are used. Büchi automata are extensions of finite state machines that can also recognize infinitely long words.