

Tutorial Automata and formal Languages

Notes for to the tutorial in the summer term 2017

Sebastian Küpper, Christine Mika

8. August 2017

1 Introduction: Notations and basic Definitions

At the beginning of the tutorial we repeat some concepts and notations, which can be helpful for the understanding of the lecture.

Notation 1.1. *In the following, we always use the following symbols with a certain meaning*

- Σ alphabet (finite set)
- Z set of states of an automaton (finite set)
- G grammar
- V set of variables of a grammar (finite set)
- S the starting variable of a grammar
- P production rules of a grammar (pairs $(V \cup \Sigma)^+ \rightarrow (V \cup \Sigma)^*$)
- Σ^* the set of all sequences over Σ , so

$$\Sigma^* = \{a_1 a_2, \dots, a_n \mid n \in \mathbb{N}_0, \forall 1 \leq i \leq n : a_i \in \Sigma\}$$
- L Language (subset of Σ^*)
- ε the empty word (an empty sequence over an arbitrary Σ)
- Σ^+ the set of all non-empty sequences over Σ , $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$

One of the most important concepts in the theory of formal languages is, of course, the concept of a language itself.

Definition 1.2 (Language). *Let be Σ a finite set, called the alphabet. Then we call Σ^* the set of all Words (sequences) over Σ , formal:*

$$\Sigma^* = \{a_1 a_2 \dots a_n \mid n \in \mathbb{N}_0, \forall 1 \leq i \leq n : a_i \in \Sigma\}$$

Σ^* contains also the empty sequence ε , which we also call the empty word. To see this, let be $n = 0$ is set in the above definition. A language L over an alphabet Σ is an arbitrary subset of Σ^* , formal: $L \subseteq \Sigma^*$.

Let us consider some examples for languages:

Example 1.3. • \emptyset , the empty language.

- Σ^* , the language of all words.
- $\{\varepsilon\}$, the language including only the empty word.
- $\{aw \mid w \in \{a, b\}^*\}$ the set of all words over the alphabet $\{a, b\}$, which starts with an a .
- $\{a^n b^n \mid n \in \mathbb{N}_0\}$ over the alphabet $\{a, b\}$.
- $\{w \in \{(,)\}^* \mid \#_((w) = \#_)(w) \wedge \forall w' \sqsubseteq w : \#_((w') \geq \#_)(w')\}$, the language of correctly bracketed expressions. \sqsubseteq denotes the prefix relation. $w' \sqsubseteq w$ holds if and only if w could be written as $w = w'w''$ with $w'' \in \{a, b\}^*$.
- $\mathcal{N}_0 = \{n_1 n_2 \dots n_m \mid m \in \mathbb{N}, \forall 1 \leq i \leq m : n_i \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}, m \neq 1 \Rightarrow n_1 \neq 0\}$.
- Primes = $\{n \in \mathcal{N}_0 \mid n \text{ interpreted as a natural number is prime.}\}$.

Remark 1.4. *In the exercises, we will often identify the set \mathcal{N}_0 with the number set \mathbb{N}_0 .*

Languages can be characterized in various ways; in the context of this lecture, we are particularly interested in two types of models:

1. Generative models (grammars)

2. Acceptors (automata / machines)

Both models provide a hierarchy of languages. The more general the model, the lower the index of the language class, but the more difficult it is to answer questions about the respective language class. In the following, we will refer to the Chomsky hierarchy of languages defined by classes of grammars. We will first define what a grammar is and then define the grammar classes.

Definition 1.5. *A grammar is a four-tuple (V, Σ, P, S) , such that*

- V is a finite set, called the set of variables. We usually use for elements $A \in V$ uppercase letters and denote the elements of V as variables.
- Σ is a finite set, called alphabet. We usually use for elements $a \in \Sigma$ lowercase letters and denote the elements of Σ as alphabet symbols or letters.
- P is a set of production rules, which means pairs $(\ell, r) \in (V \cup \Sigma)^+ \times (V \cup \Sigma)^*$. For a set M , the notation $M^+ := M^* \setminus \{\varepsilon\}$ denotes the set of all non-empty sequences over M . We use for P the notation $\ell \rightarrow r$.
- $S \in V$ is called the starting symbol.

A grammar can perform a derivation step on a word $w \in (V \cup \Sigma)^+$ as follows: If a rule $\ell \rightarrow r$ exists, such that w could be written as $w_1 \ell w_2$, $w_1, w_2 \in (V \cup \Sigma)^*$, then w could be derived to $w_1 r w_2$. Thus the occurrence of ℓ is removed from the word and r is inserted in its place. We write

$$w \Rightarrow_G w_1 r w_2$$

(or $w \Rightarrow w_1 r w_2$, if the grammar is clear from the context). The language produced by G is defined as

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

Where \Rightarrow_G^* is the repeated application of \Rightarrow_G , the language $L(G)$ is the set of all words that do not contain variables and can be derived from S by G .

Notation 1.6. *We write abbreviated, if n rules with the same left side ℓ exist, instead of $\ell \rightarrow r_1, \ell \rightarrow r_2, \dots, \ell \rightarrow r_n$ usually $\ell \rightarrow r_1 \mid r_2 \mid \dots \mid r_n$.*

If we formally restrict the grammars, we obtain a hierarchy of grammars named after the linguist Noam Chomsky, who originally set them up.

Definition 1.7 (Chomsky-Hierarchy of grammars). *Let be $G = (V, \Sigma, P, S)$ a grammar, then we denote with G a grammar of type*

- 0 *without imposing restrictions on the grammar.*
- 1 *if for all $\ell \rightarrow r$ it holds, that $|\ell| \leq r$, or $\ell = S$ and for all rules $\ell \rightarrow r$ it holds that $S \notin r$. Therefore, rule applications are never allowed to shorten the word unless they direct S directly to the empty word. Grammars that satisfy this condition are also called context-sensitive.*
- 2 *if G is context-sensitive, and for all rules $\ell \rightarrow r$ holds that $\ell \in V$, so every rule on the left contains only a single variable. We call G context-free in this case.*
- 3 *if G is context-free and in addition to all rules $\ell \rightarrow r$ it holds that $r = aA$ or $r = a$, $a \in \Sigma, A \in V$. We call G regular in this case.*

In this way, grammars can be classified into a strict hierarchy. Each grammar class contains all classes with a higher index, which means, for example, that each type 3 grammar is also type 2, type 1, and type 0. Furthermore, it is obvious that for $i = 1, 2, 3$ there are grammars of type $i - 1$, which are not of type i . We illustrate this via four different grammars for the language $L = \{aw \mid w \in \{a, b\}^*\}$:

Example 1.8. *The following grammars are grammars for the language $L = \{aw \mid w \in \{a, b\}^*\}$ over the alphabet $\{a, b\}$. Hence we have $G_i = (\{A, S\}, \{a, b\}, P_i, S)$ for $i = 0, 1, 2, 3$.*

$i = 0$ P_0 is defined as: $S \rightarrow aA \quad aA \rightarrow a \mid aAa \mid aAb$.

The grammar is of type 0, since every grammar is of type 0. However, the grammar is not of type 1 because $|aA| = 2$, $|a| = 1$ and $2 \not\leq 1$. So the $|\ell| \leq r$ condition is not satisfied for all $\ell \rightarrow r$ rules.

$i = 1$ P_1 is defined according to: $S \rightarrow aA \mid a \quad aA \rightarrow aAa \mid aAb \mid aa \mid ab$.

The grammar is of type 1, as one can easily verify: $|S| = 1 \leq 2 = |aA|$, $|S| = 1 = |a|$, $|aA| = 2 \leq 3 = |aAa| = |aAb|$ and $|aA| = 2 = |aa| = |ab|$. So $|\ell| \leq r$ is satisfied for all rules $\ell \rightarrow r$. As a grammar of type 1, this grammar is automatically also of type 0. However, the grammar is not of type 2 because there are rules where the left-hand side is not from V , i.e. all those rules with the left-hand side aA .

$i = 2$ P_2 is defined as: $S \rightarrow aA \mid a \quad A \rightarrow Aa \mid Ab \mid a \mid b$.

The grammar is of the type 1, as can be verified in the same way as in the previous case, and thus also of the type 0. Moreover, the grammar is of the type 2, since only A and S appear as left-hand side, both of which are variables. However, the grammar is not of type 3 because there is, for example, the rule $A \rightarrow Aa$, which is not allowed in regular grammars.

$i = 3$ P_3 is defined according to: $S \rightarrow aA \mid a \quad A \rightarrow aA \mid bA \mid a \mid b$.

The grammar is of the type 2 (and thus also of the type 1, as well as of the type 0), as can be verified in the same way as in the previous case. Obviously, the grammar is also of type 3, since there are only rules in which on the right side there is either an alphabet letter, or one of the expressions aA or bA each having the form that the first character is an alphabet letter and the second character is a variable.

However, another subject of examination in the lecture are languages. The Chomsky hierarchy of grammars can be naturally extended to a hierarchy of languages as follows:

Definition 1.9. *A language $L \subseteq \Sigma^*$ is of Chomsky-Type i , $i = 0, 1, 2, 3$, if a grammar G exists, which includes the Chomsky-Type i and produces L , therefore $L(G) = L$ holds.*

Note that the hierarchy of languages, similar to the hierarchy of grammars, is based on each other, which means that each language of type i , $i > 0$ is also of type $i - 1$. This is clear by a simple observation: If a language L is of type i , then there is a grammar G of the type i that generates L by definition of the language classes. Since a grammar of the i type is always of type $i - 1$, there is also a grammar – just G – that is of type $i - 1$ and which generates the language.

To show that a language is of a particular Chomsky type, it is sufficient to specify a corresponding grammar. However, to show that a language is *not* of a certain type, further efforts are needed. In the simplest case, the type 3 languages, one can use a suitable automat model. In contrast to grammars, which function as producer systems – that is, they are able to generate words of a language – machines are acceptors. This means that they do not generate words, but

expect an input from $w \in \Sigma^*$, similar to a computer program, and then check that the word w is in the language that the machines generate.

Example 1.10. We now consider some exercises:

- *Exercise 1: Given the following grammar $G = (\{S, A, B\}, \{a, b\}, P, S)$ with P :*

$$\begin{aligned} S &\rightarrow aA|bB \\ A &\rightarrow aA|bA|a \\ B &\rightarrow aB|bB|b \end{aligned}$$

What language is generated by this grammar?

Possible solution steps:

- *First we derive some words:*












$$S \rightarrow aA \rightarrow abA \rightarrow aba \text{ or } S \rightarrow aA \rightarrow abA \rightarrow abaA \rightarrow abaa \text{ etc.}$$

You should try as many different derivations-steps as possible in order to recognize a pattern. This means that you should look at which words can not be derived. Let us try to derive the word abba: $S \rightarrow aA \rightarrow abA \rightarrow aba$ or only $abA \rightarrow abbA$, but we could not generate abb. On closer inspection, it can be seen that there is no way to end with b using $S \rightarrow aA$. \Rightarrow Consider the derivation paths to recognize the constraints of the produced words.

- *Exercise 2: Adventure*








Specify a grammar that allows only paths, so that after a dragon there always follows a jump into a river in the case no swords are available.

- *First consider what this means: Can it be that   occurs in a path?*

- *If you start $S \rightarrow$ you have to jump directly into the water, as soon as you meet a , $S \rightarrow \dots | \text{  } S | \varepsilon$. But on the other hand it can also be that you find a  or you have to cross a . Therefore we have $S \rightarrow XS | \text{  } S | \varepsilon$ and X should exactly produce the symbols or sequences of symbols that do not violate the Dragon rule: $X \rightarrow \text{ |  |  | }$.*

- *Exercise 3: Combination of the simple door rule with the dragon rule. Remember the simple door rule: Passing a door is only allowed if you have a key. You can use a key for every door.*

Starting from S we can pass every symbol possible by X . If we meet a dragon we have to jump into a river. After having found a key, we use K for deriving the paths, in which we are now also able to open every door:

$$\begin{aligned} S &\rightarrow XS | \text{  } S | \text{ } K | \varepsilon \\ K &\rightarrow XK | \text{ } K | \text{  } K | \varepsilon \\ X &\rightarrow \text{ |  | } \end{aligned}$$

- *Exercise 4: Given a language L , we search for a grammar G generating exactly the words of the language. How do we determine a grammar, which is also of the maximal type?*

1. Consider the language $L_1 = \{b^m aac^t \mid m, t \in \mathbb{N} \setminus \{0\}\}$.
 First property of every produced word is, that you have to start with at least one b . So $S \rightarrow bS \mid \dots$ should be the first idea for constructing a grammar, which allows you as many b 's you want at the beginning of a word. Now you should be able to switch to the aa part of the word: $S \rightarrow bS \mid bB$, but still ensure that there is at least one b at the beginning. A sequence of two a is not that difficult: $A \rightarrow aB$ and $B \rightarrow aC$, cause you only want exactly two a 's. Now we only need to enable as many c 's as we want, but at least one: $C \rightarrow Cc \mid c$. Of course you can displace $A \rightarrow aaC$, but that would not be a grammar of type 3. Just try to avoid using rules of lower types, if you have the feeling, that it is a higher Chomsky-type.
2. Consider $L_2 = \{b^m aac^t \mid m, t \in \mathbb{N} \setminus \{0\} \text{ and } m > t\}$. Again every word should start with at least one b , but here we can not produce as many c 's as we want. There must be more b 's than c 's in every word. So this time $S \rightarrow bBX$ could be one idea and $B \rightarrow bB \mid b$ allows to produce arbitrary many b 's you. The variable X should ensure, that you can not produce more c 's than b 's: $X \rightarrow bXc \mid \dots$. But now we somehow need to terminate with aa between the two sequences: $X \rightarrow bXc \mid aac$. For that language we could not restrict to rules of type 3, because we have a relation between the length of the beginning and final sequence of every word.
3. Consider $L_3 = \{a^m bc^{2^m} d^m \mid m, n \geq 1\}$ a language. Again we have a relation between the beginning and end of every word. But in addition we are only allowed to produce twice as many c 's as a 's. So we only consider the type 1 rules here:
 $S \rightarrow aSCd \mid abccd$ seems to be a nice start for our production rules. That way we ensure that only as many d 's as a 's are produced. But what we get is something like that: $S \rightarrow aSCd \rightarrow aaSCdCd \rightarrow aaabccdCdCd$. To shift all the d to the right side of the word and ensure that all C occur in one sequence before the sequence of d 's we need to work with some context: $dC \rightarrow Cd$ and finally $cC \rightarrow ccc$, just to ensure, that we get two c 's for every a .

Finally, a remark on the special rule for Typ 1 grammars: $S \rightarrow \varepsilon$: This rule is allowed if S does not appear on the right side of any rule. This also holds for Typ 2 and Type 3. Through appropriate transformations you can create a grammar that is up to ε -derivations regulare (context-free), into a regulare (Context-free) grammar, which fulfill the specific ε rule. Such a construction generally does not exist for all type 1 grammars.

2 Automata models and concepts

As already mentioned grammars generate words, but there are also concepts for modeling acceptors, which exactly accept the words of a language $L \subseteq \Sigma^*$ for some given alphabet Σ .

2.1 Automata models I (Regular languages)

We start with an acceptance model for languages, which can be generated by Typ-3-grammars.

Definition 2.1. A deterministic finite automaton is a five-tuple $M = (Z, \Sigma, z_0, \delta, E)$ where Z and Σ are finite sets, $Z \cap \Sigma = \emptyset$ and $z_0 \in Z$ is the initial state, $E \subseteq Z$ is the set of accepting states. $\delta : Z \times \Sigma \rightarrow Z$ is the transfer function, which assigns a unique successor state to each pair of state and alphabet symbol.

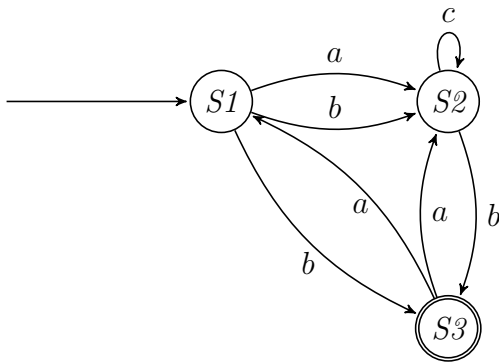
The set of all words accepted by a DFA M is defined as the language $L(M)$. The set of all languages accepted by a DFA is the class of regular languages. Non-deterministic finite automata

(NFA), DFA, and type-3 grammars (in the Chomsky hierarchy) describe the same language class. NFAs can be converted into equivalent DFAs by means of the power set construction.

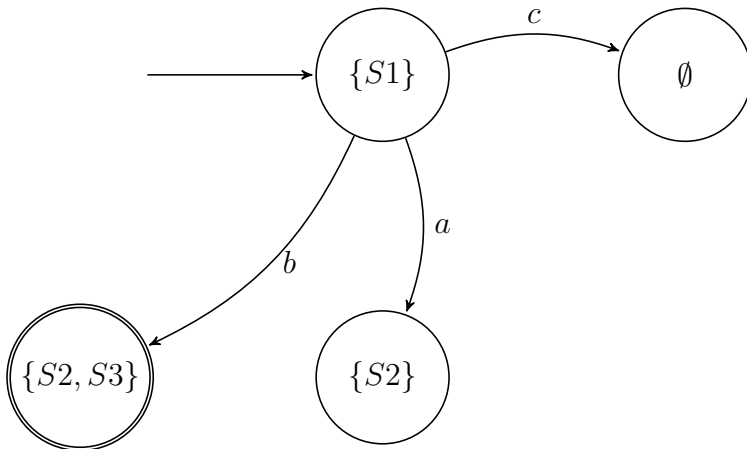
Definition 2.2. A non-deterministic finite automaton is a five-tuple $M = (Z, \Sigma, \delta, S, E)$ where Z is a finite set of states, Σ is a finite set called alphabet, S is a set of initial states, E is the set of final states and $\delta: Z \times \Sigma \rightarrow \mathcal{P}(Z)$ is the transition function.

Both concepts of modelling acceptors capture all regular languages. By the power set construction a NFA can be transformed into a DFA. Imagine an deterministic automaton simulates all possible behaviours of a NFA and remembers all states he is visiting. So the states of this automaton corresponds to set of states. Here is one example to demonstrate that fact:

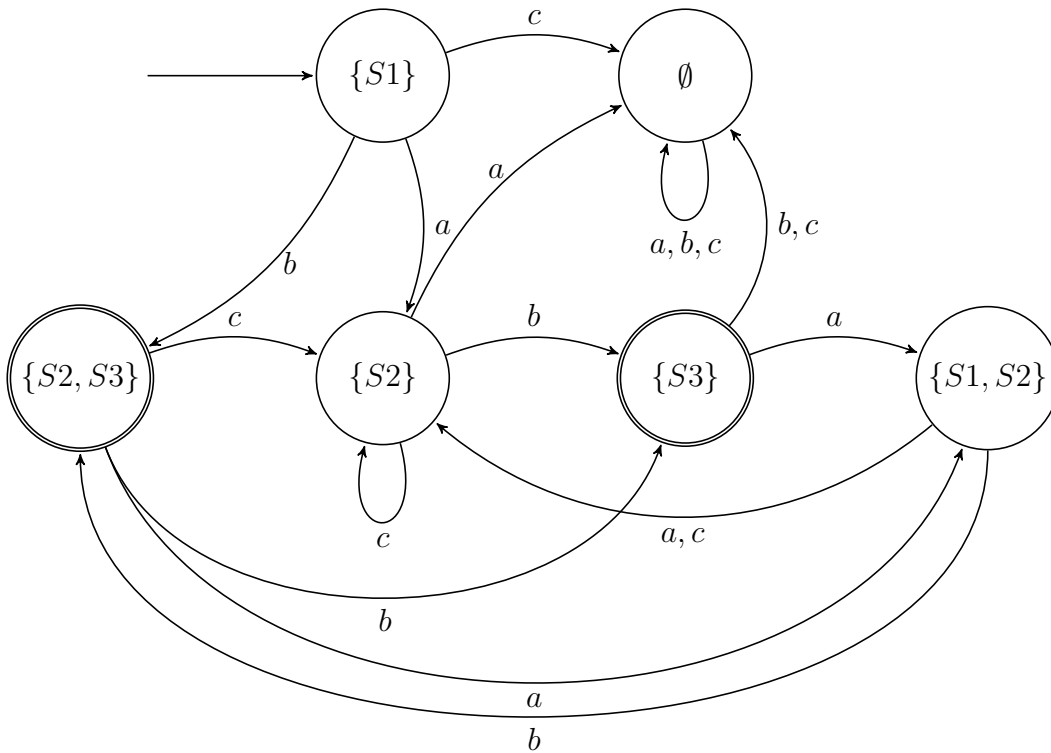
Example 2.3. Powerset construction: NFA into a DFA:



Reading b being in state $S1$ could end in state $S2$ or $S3$ where reading an a can only lead to state $S2$ and for c no transition is defined. This is captured in the set $\{S2, S3\}$, $\{S2\}$ and \emptyset :



Here you see the whole DFA generated by the powerset construction:



Another concept to describe languages, which can be modelled by *DFA* or *NFA* are regular expressions.

Definition 2.4. *Regular expression*

A regular expression α is of the following structure:

- \emptyset
- ε
- a with $a \in \Sigma$. The single symbols of an alphabet serves as the basic building blocks for non trivial expressions.
- $\alpha\beta$ Means you just concatenate two regular expressions, for example $\alpha = a, \beta = cac^*$ so $\alpha\beta = acac^*$.
- $(\alpha|\beta)$ Choice the same example $(\alpha|\beta) = (a|cac^*)$.
- $(\alpha)^*$ and $(\alpha)^* = a^*$, so this is the operator that enables the representation of infinite languages. Or $(\beta)^* = (cac^*)^*$, such that you can construct caccccacaccccccc.

and α, β are regular expressions.

In the next example we will summarise the relation between different grammar types, finite automata and regular expressions:

Example 2.5. Consider the following grammar G :

$$\begin{aligned}
 S &\rightarrow \varepsilon \mid aA \mid bB \mid a \mid b \mid ab \mid AXB \mid AX \mid XB \mid X \\
 A &\rightarrow aA \mid a \quad B \rightarrow bB \mid b \quad X \rightarrow ab \mid abX
 \end{aligned}$$

1. Which type of grammar do we have?

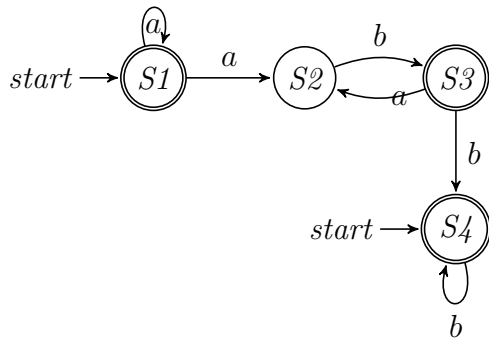
This grammar is of type 2 because the special ε rule is satisfied and $|l| \leq |r|$ for all $l \rightarrow r$.

2. Describe the language using set notation and regular expressions: $L = \{a^n(ab)^l b^m \mid n, m, l \in \mathbb{N}_0\}$ and $L(G) = (a^*(ab)^*b^*)$

3. Please define a grammar of type 3 for the language L from above:

$$\begin{aligned} S &\rightarrow \varepsilon \mid aS_1 \mid aS_4 \mid aS_2 \mid bS_4 \mid a \mid b \\ S_1 &\rightarrow aS_1 \mid aS_2 \mid a \mid aS_4 & S_2 &\rightarrow bS_3 \mid b \\ S_3 &\rightarrow aS_2 \mid b \mid bS_4 & S_4 &\rightarrow bS_4 \mid b \end{aligned}$$

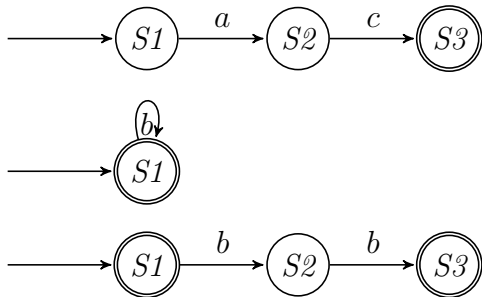
4. Define an automaton, which exactly accepts the language L of G :



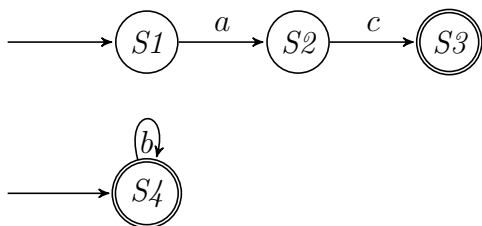
Next we will discuss transformations from NFA to regular expressions considering an example:

Example 2.6. The single steps are based on the rules in the lecture. We start with the regular expression $((ac \mid b^*)bb)^*$ to NFA:

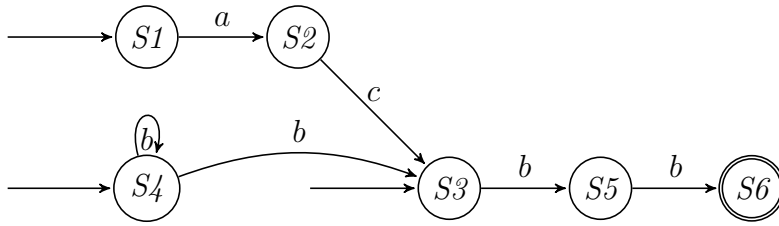
1. Construct an automaton for the different basic blocks: ac, b^* and bb :



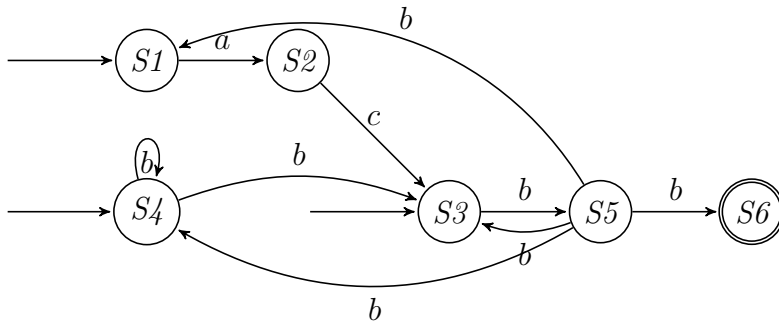
2. First combine the automaton for $(ac \mid b^*)$, just drawing both automata:



3. Next build the automaton for $(ac \mid b^*)bb$ connecting the accepting states of $(ac \mid b^*)$ with the initial states of bb and mark the initial state of bb also as initial state because $\varepsilon \in (ac \mid b^*)$.



4. Next, we need to connect the states connected with an accepting state of the automaton of step 4 with the initial states, such that the star operation is captured by the NFA:

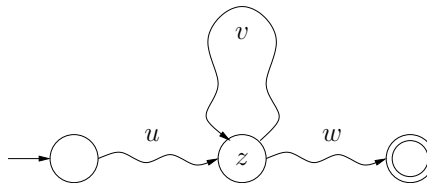


5. Last, we need to add an initial and accepting additional state, to enable the empty word. (no illustration)

Often the questions arises if a language is regular or not. This question could be answered by two different approaches. One approach, the Pumping Lemma, can be used to check if a language is not regular, but not to answer the question if it is regular. The other method is based on equivalence relations. Let us first concentrate on the Pumping Lemma:

2.1.1 Pumping Lemma

The idea behind the Pumping Lemma is to use the property, that a regular languages has to be accepted by an automaton consisting of finitely many states. This means for a word $x = uvw \in L$ which is at least as long as the number of states the automaton has, one of the states z of the automaton must be visited at least twice.



The loop created this way, can than be repeated several times (or not at all), thereby the word $x = uvw$ is **pumped up** and we find that uv^2w , uv^3w , \dots , and uw are also in L . To achieve, that z is visited at least two times and does not contain the empty word we demand the following properties for $x = uvw \in L$:

1. $|v| \geq 1$: The loop is not trivial and was taken at least once.
2. $|uv| \leq n$: After at most n alphabet symbols the state z will be visited second time.

How to use the pumping lemma to show that a language is not regular?

Statement of the pumping lemma with logical operators:

- L regular $\rightarrow \exists n \forall x \in L, |x| \geq n \exists u, v, w, x = uvw \forall i (uv^i w \in L)$

- $\forall n \exists x \in L, |x| \geq n \forall u, v, w, x = uvw \exists i (uv^i w \notin L) \rightarrow L$ is not regular.

This logical statements just mean, that if you find a word $x = uvw \in L$ for this language, that for some n and all uvw satisfying the conditions 1 and 2 the pumping produces a word $uv^i w \notin L$ then L is not regular. Of course sometimes it could be hard to find the right word $x = uvw$ or there is maybe none (see example lecture: $L = \{a^k b^m c^m \mid k, m \geq 0\} \cup \{b^i c^j \mid i, j \geq 0\}$). In the latter case, no conclusion can be drawn about the non regularity of the language. For such difficult cases we recommend to use the Myhill Nerode Theorem. Before we switch to the Myhill Nerode Theorem we first consider some examples for the Pumping Lemma:

Example 2.7. *The following languages are not regular:*

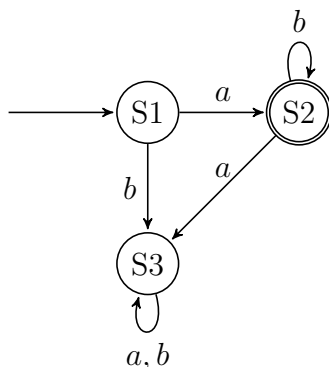
1. $L_1 = \{a^n b^n \mid n \in \mathbb{N}_0\}$ Let $n \in \mathcal{N}_0$ be given, we choose $x = a^n b^n$, obviously the number of symbols in $a^n b^n$ is $2n > n$.
Possible decomposition for $x = a^n b^n$:
 $u = a^k \ v = a^l \ w = a^{n-l-k} b^n$ with $l \geq 1, k \geq 0$ and $l + k \leq n$. For $i = 2$ we get $a^k a^{2l} a^{n-l-k} b^n = a^{n+l} b^n \notin L$. Note that there are no further decompositions with the required properties that are not already contained in this one. Only decompositions which are not already covered must be considered ($u = \varepsilon$ is covered by a^0).
2. $L_2 = \{a^{n!} \mid n \in \mathbb{N}_0\}$. Let $n \in \mathcal{N}_0$ be given, we choose $x = a^{n!}$, obviously the number of symbols in $a^{n!}$ is $n! \geq n$.
Possible decomposition for $x = a^{n!}$:
 $u = a^k \ v = a^l \ w = a^{n!-l-k}$ with $l \geq 1, k \geq 0$ and $l + k \leq n$. For $i = 2$ we get $a^k a^{2l} a^{n!-l-k} = a^{n!+l} \notin L$, since $n! + l < (n+1)!$ with $l \leq n$ for $n > 1$, i.e. $n! + l \neq m!$ for all $m \in \mathcal{N}_0$. But here the case $n = 0$ and $n = 1$ are not captured.
To capture also $n \geq 0$ we choose $x = a^{(n+2)!}$:
 $u = a^k \ v = a^l \ w = a^{(n+2)!-l-k}$. For $i = 2$ we get $a^{(n+2)!+l}$ since $l \leq n : (n+2)! + l \neq m!$ for all $m \in \mathcal{N}_0$, so $a^{(n+2)!+l} \notin L_2$.
3. $L_3 = \{a^k b^l c^m \mid 1 \leq k < l, m \geq 1, k, l, m \in \mathbb{N}\}$. Let $n \in \mathcal{N}_0$ be given, we choose $x = a^k b^n c$, obviously the number of symbols in $a^k b^n c$ is $n + k + 1 > n$.
Possible decomposition for $x = a^k b^n c$ with $k < n$:
1. $u = a^j \ v = a^t \ w = b^n c$ with $j + t < n$. So for $i = n$ it holds that $n \cdot t > n$. hence $uv^n w \notin L_3$ hence $t \geq 1$.
2. $u = a^j \ v = a^t b^s \ w = b^{n-s} c$ with $j + t + s < n$. For $i = 2$ we get $a^j a^{2t} b^{2s} a^t b^{n-s} c \notin L_3$.
Alternativ word $x = a^n b^{n+1} c$. Only one decomposition possible.

Like mentioned before the Pumping Lemma can be used to show that a language is not regular, but cannot be used as decision-making. Also in some cases it fails to show, that a language is not regular. However, in order to be able to make statements about the (non) regularity of a language, we now focus on the Myhill Nerode Theorem.

Remark 2.8. *In the next section we speak about equivalence relations, please have a look at the formal definition in the lecture slides, if you are not familiar with it.*

2.1.2 Myhill Nerode

The idea behind the next theorem is again to use the finite set of states. Consider the following deterministic automaton M over the alphabet $\Sigma = \{a, b\}$:



From state S_1 an a yields an accepting word. Also a word that has the form ab^* yields a word contained in $L(M)$ from state S_1 . Now have a look at the state S_2 , from this reading an a does not end up in a word included in $L(M)$. So both states allow to make different transitions or sequences of transitions to end up in an accepting state. This concept of allowing the same sequences of transitions, which represents words in Σ^* is captured by the Myhill Nerode Theorem.

Definition 2.9. *Myhill-Nerode-Equivalence*

Let be L a language and words $x, y \in \Sigma^*$. We define an equivalence relation R_L with $x R_L y$ if and only if

$$\text{for all } z \in \Sigma^* \text{ it holds that } (xz \in L \iff yz \in L).$$

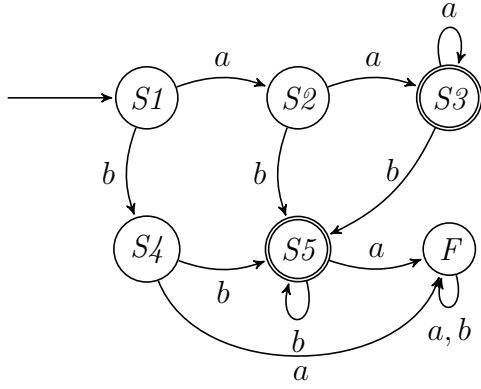
For our example automaton consider $x = ab$ and $y = abbb$. For both words the set of words $z \in \Sigma^*$ which produces $xz \in L$ and $yz \in L$ is the same set. This means that both words belongs to the same equivalence class. For a word like $aa \in \Sigma^*$ every word $z \in \Sigma^*$ does not generate a word $aa z \in L$. So every word, where the set of suffixes, which can continue the word such that the resulting word is in L , is empty belongs to that equivalence class. This does not mean that you have in general the empty set for words $w \notin \Sigma^*$ and no z with $wz \in L$. In our example $\varepsilon \notin L$ but $\varepsilon a \in L$.

In our example we have exactly three equivalence classes: $[\varepsilon] = \{\varepsilon\}$: In this class every $z \in L$ with $z = ab^* \in L$ yields a word in L . This class corresponds to state S_1 and $[a] = \{w \in \Sigma^* | z \in L(b^*) \text{ and } wz \in L.\}$ is represented by state S_2 . For all other words we have a third class $[aa]$ because for them the set of valid suffixes is the empty set, corresponding in our example to state S_3 . So looking at the minimal deterministic automaton of a regular language you have as many equivalence classes as states.

So to check if a language is regular, just check if you have finitely many equivalence classes.

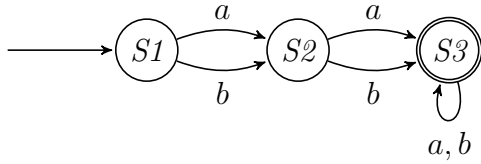
Example 2.10. *To better understand this concept, let us take a look at some examples, two for regular languages and one for non regular languages. (We assume that we only consider the minimal automata for a given language and a set of valid suffixes means: the set of words $z \in \Sigma^*$ such that for all words w of an equivalence class based on a language L $wz \in L$) :*

1. $L_1 = \{a^n b^m \mid n + m \geq 2; n, m \in \mathbb{N}_0\}$ is regular, hence we show that we have finitly many equivalence classes:



- $[\varepsilon] = \{\varepsilon\}$ and the set of suffixes is $\{a^n b^m \mid n + m \geq 2\}$
- $[a] = \{a\}$ and the set of suffixes is $\{a^n b^m \mid n + m \geq 1\}$
- $[aa] = \{aa, aaa, aaaa, \dots\}$ and the set of suffixes is $\{a^n b^m \mid n, m \in \mathbb{N}_0\}$
- $[b] = \{b\}$ and the set of suffixes is $\{b^m \mid m \in \mathbb{N}_1\}$
- $[ab] = \{a^n b^m \mid m \geq 1, n + m \geq 2\}$ and the set of suffixes is $\{b^m \mid m \in \mathbb{N}_0\}$
- $[ba] = \{w \mid w \notin L(a^*b^*)\}$ and the set of suffixes is $\{\}$

2. $L_2 = \{w \in \Sigma^* \mid |w| \geq 2\}$ with $\Sigma = \{a, b\}$ is regular, hence we show that we have finitely many equivalence classes:



- $[\varepsilon] = \{\varepsilon\}$ and the set of suffixes is $\{w \in \Sigma^* \mid |w| \geq 2\}$
- $[a] = \{a, b\}$ and the set of suffixes is $\{w \in \Sigma^* \mid |w| \geq 1\}$
- $[aa] = \Sigma^* \setminus \{\varepsilon, a, b\}$ and the set of suffixes is Σ^* .

3. $L_2 = \{a^n b^n \mid n \in \mathbb{N}_0\}$ is not regular, hence we show that we have infinitely many equivalence classes:

Consider the words $a, aa, aaa, \dots, a^i, \dots$ with $i \in \mathbb{N}_0$.

It holds that $\neg(a^i R_L a^j)$ for $i \neq j$, since $a^i b^i \in L$ and $a^j b^i \notin L$. So we have for example a different class for every $i \in \mathbb{N}_0$ with $\{a^{k+i} b^k \mid k \in \mathbb{N}_0\}$ and the corresponding set of valid suffixes is $\{b^i\}$.

4. $L_3 = \{a^n b^m c^{n+m} \mid n, m \in \mathbb{N}\}$ is not regular, hence we show that we have infinitely many equivalence classes:

It holds that $\neg(a^i R_L a^j)$ for $i \neq j$, since $a^i b^m c^{i+m} \in L$ and $a^j b^m c^{i+m} \notin L$.

In the next section we will have a look at the context-free languages and accepting models.

2.2 Automata models II (Context free languages)

Both models (finite automata or regular grammars) for regular languages do not allow to remember how often an alphabet symbol occurs within a word. As already seen in context-free grammars, it is possible to generate words that contain as many a's as b's. We will now consider an acceptance model that exactly covers the context-free languages:

Definition 2.11. A (non deterministic) pushdown automata M is a 6-tuple $M = (Z, \Sigma, \Gamma, \delta, z_0, \#)$, wher

- Z set of states,
- Σ the input alphabet (with $Z \cap \Sigma = \emptyset$),
- Γ the cellar alphabet,
- $z_0 \in Z$ the initial state,
- $\delta: Z \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \rightarrow \mathcal{P}_e(Z \times \Gamma^*)$ the transfer function and
- $\# \in \Gamma$ the lowest cellar symbol.

Before looking more closely at the relation between context-free grammars and pushdown automata, we first look at a more efficient algorithm for the word problem.

2.2.1 Word problem and context-free grammars

We repeat the definition of the word problem:

Definition 2.12. *Word problem* Given a grammar G (arbitrary grammar type) and a word $w \in \Sigma^*$. Is $w \in L(G)$?

In the lecture and exercise we have already discussed an algorithm for the word problem which applies for grammars from type 1. Here we assume that a context-free grammar is given in Chomsky Normalform (please have a look into the lecture slides for Chomsky Normalform and transformation of a type-2-grammar into a grammar in Chomsky Normalform).

Definition 2.13. *Chomsky Normalform*

Given a context-free grammar $G = (V, \Sigma, P, S)$ all rules in P are of the following two forms:

$$A \rightarrow BC \text{ or } A \rightarrow a$$

Given an arbitrary word $x \in \Sigma^*$ we want to know how this word could be generated by the given grammar in Chomsky Normalform. One case could be, that the word only consists of one single terminal symbol a , so it is easy to check, if any rule in P enables a derivation form $S \rightarrow A_i \cdots \rightarrow a$ with $A_i \in V$. In the other case the word $x = a_1 \dots a_n$ consists of several terminal symbols. In this case first a production of the form $A \rightarrow BC$ must be applied and a part $a_1 \dots a_k$ should be derived from B and $a_{k+1} \dots a_n$ from C . However, it is not clear where the word x must be divided, that is, how big the index k is. To solve this problem the following concept is used: Given a word $x = a_1 \dots a_n$. Check for all k with $1 \leq k < n$:

- Determine all variables V_1 from which $a_1 \dots a_k$ is derivable.
- Determine all variables V_2 from which $a_{k+1} \dots a_n$ is derivable.
- Determine whether there are variables A, B, C with $A \rightarrow BC \in P$ and $B \in V_1, C \in V_2$. In this case x could be generated by A .

This concept is optimized by the following CYK-Algorithm:

- First determine all variables from which all subwords of x of the length 1 could be derived.
- Then determine all variables from which all subwords of x of the length 2 could be derived.

- ...

- Then determine all variables from which x could be derived and if S is an element of this set the word x can be generated by the grammar G .

We also illustrate this concept by an example. The general work flow of this algorithm is demonstrated in the slides (283-286).

Example 2.14. Let the grammar $G = (\{S, U, B, V, X, A, A'\}, \{a, b\}, P, S)$ be given and P is defined as follows:

$$\begin{array}{lll}
 S \rightarrow UB & A \rightarrow UB & X \rightarrow VB \mid A'B \\
 U \rightarrow BX & V \rightarrow A'A & \\
 A' \rightarrow a & B \rightarrow b &
 \end{array}$$

Given the word $x = bababbb \in L(G)$ we demonstrate how this algorithm works:

- We construct the following table with some intermediate steps starting with the subwords of length 1:

	b	a	b	a	b	b	b	b
1	B	A'	B	A'	B	B	B	B

- Next we consider all subwords of length 2, due to this no entry for the subword ba cause no rule of the form $V' \rightarrow BA'$ with $V' \in V$ exists, but $X \rightarrow A'B \in P$ for ab :

	b	a	b	a	b	b	b	b
1	B	A'	B	A'	B	B	B	B
2		X		X				

- Next we consider all subwords of length 3, due to this no entry for the subword aba cause no rule of the form $V' \rightarrow XA'$ with $V' \in V$ exists, but $U \rightarrow BX \in P$ for bab :

	b	a	b	a	b	b	b	b
1	B	A'	B	A'	B	B	B	B
2		X		X				
3	U		U					

- After steps 4,5,6 and 7 we finally get for the word x of length 8:

	b	a	b	a	b	b	b	b
1	B	A'	B	A'	B	B	B	B
2		X		X				
3	U		U					
4			S, A					
5		V						
6		X						
7	U							
8	S, A							

In the next section we discuss the concept of transforming pushdown automata into grammars and vice versa.

2.2.2 Context free Grammar \Leftrightarrow Pushdown automaton

In that section we want to consider how to transform a pushdown automaton into a contextfree grammar and vice versa. Let be a non deterministic pushdown automaton

$$K = (\{z_0, z_1\}, \{a, b\}, \{\#, A\}, \delta, z_0, \#)$$

given where δ is defined as follows:

$$\begin{aligned}\delta(z_0, a, \#) &= \{(z_0, A\#)\} \\ \delta(z_0, a, A) &= \{(z_0, AA)\} \\ \delta(z_0, b, A) &= \{(z_1, \varepsilon)\} \\ \delta(z_0, \varepsilon, \#) &= \{(z_0, \varepsilon)\} \\ \delta(z_1, b, A) &= \{(z_1, \varepsilon)\} \\ \delta(z_1, \varepsilon, \#) &= \{(z_1, \varepsilon)\}\end{aligned}$$

All other $\delta(z, t, V)$ with $z \in \{z_0, z_1\}, t \in \{a, b\}$ and $V \in \{A, \#\}$ are mapped to the emptyset. Before we extract the production rules for the grammar, we first need to establish the variable set V . The alphabet is the same as for the pushdown automaton and the initial Variable is S . And $V = \{S, (z_0, \#, z_0), (z_0, \#, z_1), (z_1, \#, z_0), (z_1, \#, z_1), (z_0, A, z_0), (z_0, A, z_1), (z_1, A, z_0), (z_1, A, z_1)\}$ First we provide a production rule for removing the bottom element of the stack $\#$ with $S \rightarrow (z_0, \#, z_0) \mid (z_0, \#, z_1)$. Next we capture deleting of A from top of the stack if the automaton is in state z_0 reading a b on the input tape with $(z_0, A, z_1) \rightarrow b$ and $(z_1, A, z_1) \rightarrow b$ for being in state z_1 . If the complete word is read and the last symbol on the stack is $\#$ the stack is emptied, This behaviour of the pushdown automaton is transformed into the rule $(z_1, \#, z_1) \rightarrow \varepsilon$.

Now we consider the remaining two parts of δ . Here we will describe the transformation based on $\delta(z_0, a, \#) = \{(z_0, A\#)\}$, which means that the symbol $\#$ is replaced by $A\#$ if the automaton is in the state z_0 and reads an a (for the second part it is similar). This implies that this two symbols on the stack, has to be removed somehow. It is not clear which states simulate the deleting, so we construct the production in the following way:

$$(z_0, \#, ?) \rightarrow a(z_0, A, ?)(?, \#, ?)$$

We construct all possible combinations based on the statespace:

$$\begin{aligned}(z_0, \#, z_0) &\rightarrow a(z_0, A, z_0)(z_0, \#, z_0) \\ (z_0, \#, z_0) &\rightarrow a(z_0, A, z_1)(z_1, \#, z_0) \\ (z_0, \#, z_1) &\rightarrow a(z_0, A, z_0)(z_0, \#, z_1) \\ (z_0, \#, z_1) &\rightarrow a(z_0, A, z_1)(z_1, \#, z_1)\end{aligned}$$

It can be, that some production rules do not produce sequences of terminal symbols, like for example $(z_0, \#, z_0) \rightarrow a(z_0, A, z_1)(z_1, \#, z_0)$, but that has no impact on the language $L = \{a^n b^n \mid n \in \mathbb{N}_0\}$. And finally the rest of the production rules based on $\delta(z_0, a, A) = \{(z_0, AA)\}$:

$$\begin{aligned}(z_0, A, z_0) &\rightarrow a(z_0, A, z_0)(z_0, A, z_0) \\ (z_0, A, z_0) &\rightarrow a(z_0, A, z_1)(z_1, A, z_0) \\ (z_0, A, z_1) &\rightarrow a(z_0, A, z_0)(z_0, A, z_1) \\ (z_0, A, z_1) &\rightarrow a(z_0, A, z_1)(z_1, A, z_1)\end{aligned}$$

Now we consider the opposite direction, where we transform a grammar into an pushdown automaton. Again we consider the language $L = \{a^n b^n \mid n \in \mathbb{N}_0\}$, but this time described by the grammar $G = (\{S, X\}, \{a, b\}, P, S)$ with P defined as follows:

$$S \rightarrow aXb \mid \varepsilon \mid ab \qquad X \rightarrow aXb \mid ab$$

For that transformation we use the stack to derive a word (somehow the word is guessed). After parts of the word are compared with the input they are removed if they match the input. So we construct the pushdown automaton $K = (\{z\}, \{a, b\}, \{S, X\} \cup \{a, b\}, \delta, z, S)$ and δ is defined as follows:

$\delta(z, \varepsilon, S) \rightarrow \{(z, aXb), (z, \varepsilon), (z, ab)\}$ and $\delta(z, \varepsilon, X) \rightarrow \{(z, aXb), (z, ab)\}$ for rating a word on the stack based on P .

$\delta(z, a, a) \rightarrow \{(z, \varepsilon)\}$ and $\delta(z, b, b) \rightarrow \{(z, \varepsilon)\}$ for comparing the rated word with the input word w .

Example 2.15. *Let be the word $aabb$ be given and the write-read head is positioned at the beginning of the word from the left. First we rate $S \rightarrow aXb$ and update the stack. After that we compare the symbol on the input tape with the top of the stack and remove the a . Since a comparison of X and a leads to a mismatch, we again guess a derivation on the stack $X \rightarrow ab$. After that we compare the content of the input tape with the top of the stack and remove the matching parts.*

Input tape:	<u>a</u> abb	<u>a</u> abb	<u>ab</u> b	<u>ab</u> b	<u>bb</u>	<u>b</u>	-
Stack:		a X b	X b	a b b	b b	b	

Just as with the regular languages, sometimes the question arises whether a language is context-free. This can be the case, for example, if one has determined that a language is not regular, but one nevertheless wants to use an efficient method for the word problem. For this we will look at the pumping lemma for context-free languages in the next section.

2.2.3 Context-free languages: Pumping Lemma

For context-free languages a similar concept is used as for regular languages is used. But here it is not considered, how often a state is visited in a run of an automaton, but whether a variable is used twice when deriving the word from a grammar. For a word of sufficient length variable is used inside the syntax tree at least twice. Thinking of the fact, that every context-free language could be represented by a *type 2* grammar in Chomsky normal form we can infer long enough means, that a word $z \in L$ satisfies $|z| \geq 2^k$ where k is the number of variables of the grammar in Chomsky normal form. Then it is obvious that a path in the syntax tree exists with $k + 1$ nodes. In order to capture the effect of visiting a variable twice, we consider a decomposition of z into five parts $z = uvwxy$. To reduce the number of decompositions we demand the following properties for $z = uvwxy \in L$ and $n = 2^k$:

1. $|vx| \geq 1$:
2. $|vwx| \leq n$: and
3. for all $i = 0, 1, 2, \dots$ it holds: $uv^iwx^iy \in L$.

To show that a language is not context-free we need to consider all possible decompositions of z satisfying $|vx| \geq 1$ and $|vwx| \leq n$ and prove that some $i \in \mathbb{N}$ exists such that $uv^iwx^iy \notin L$.

Example 2.16. Consider the language $L = \{a^n b^m a^n \mid n > m\}$. We choose $z = a^{n+1} b^n a^{n+1}$. Because of $|vwx| \leq n$ there are only four different cases how the symbols can be distributed: vwx only consists of:

- a 's
- a 's followed by b 's
- only b 's
- b 's followed by a 's

Now let us discuss what happens for every case, no matter how the decomposition looks in detail if we choose an $i \geq 1$:

- a 's: For $i = 2$ we get more a 's before the b 's than after them. This generates a word, which is not in L . (Or more a 's after the b 's than before the b 's.)
- a 's followed by b 's: Again for $i = 2$ we get more a 's before the b 's than after the b 's. This generates a word, which is not in L .
- only b 's: We get for $i = (n + 1) * 2$ more b 's than a 's at all.
- b 's followed by a 's: We get more a 's after the b 's than before the b 's.