Lecture Formal Languages and Automata

Markus Lohrey

University of Siegen

Sommersemester 2024

Lecture Organization

Under
https://www.eti.uni-siegen.de/ti/lehre/sommer_2024/fsa/ you can find

- updated lecture slides,
- exercise sheets,
- announcements, etc.

Recommended Literature:

- Uwe Schöning, Theoretical Computer Science – Briefly Summarized, Spektrum Akademischer Verlag (5th Edition): The section on computability closely follows this book in content.
- Lutz Priese, Katrin Erk, *Theoretische Informatik: Eine umfassende Einführung*. Springer: Available electronically through the university library.
- Alexander Asteroth, Christel Baier, *Theoretische Informatik*, Pearson Studium: This book is structured somewhat differently from the lecture but still provides a very good supplement.

Set Theory Basics (Review from DMI)

Naive Definition (Sets, Elements, $\in$, $\notin$)

A set is the collection of certain distinct objects (the elements of the set) into a new whole.

We write $x \in M$ if the object x belongs to the set M.
We write $x \notin M$ if the object x does not belong to the set M.

A set that consists of only a finite number of objects (a finite set) can be specified by listing these elements explicitly.

Example: $M = \{2, 3, 5, 7\}$.

The order of listing does not matter: $\{2, 3, 5, 7\} = \{7, 5, 3, 2\}$.

Multiple listings do not matter either: $\{2, 3, 5, 7\} = \{2, 2, 2, 3, 3, 5, 7\}$.

Set Theory Basics (Review from DMI)

A particularly important set is the empty set $\emptyset = \{\}$, which contains no elements.

In mathematics, one often deals with infinite sets (sets consisting of infinitely many objects).

Such sets can be specified by stating a property that characterizes the elements of the set.

Examples:

- $\mathbb{N} = \{0, 1, 2, 3, 4, 5, \ldots\}$ (set of natural numbers)
- $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ (set of integers)
- $P = \{n \in \mathbb{N} \mid n \geq 2, n \text{ is only divisible by 1 and } n\}$
  (set of prime numbers)

Set Theory Basics (Review from DMI)

Definition ($\subseteq$, Power Set, $\cap$, $\cup$, $\setminus$, disjoint)

Let A and B be two sets.

- $A \subseteq B$ means that every element of A also belongs to B (A is a subset of B); formally:

$$\forall a : a \in A \rightarrow a \in B$$

- $2^A = \{B \mid B \subseteq A\}$ (power set of A)
- $A \cap B = \{c \mid c \in A \text{ and } c \in B\}$ (intersection of A and B)
- $A \cup B = \{c \mid c \in A \text{ or } c \in B\}$ (union of A and B)
- $A \setminus B = \{c \in A \mid c \notin B\}$ (difference of A and B)
- Two sets A and B are disjoint if $A \cap B = \emptyset$ holds.

Set Theory Basics (Review from DMI)

Definition (Arbitrary Union and Intersection)

Let I be a set, and for each $i \in I$, let A_i be a set. Then we define:

$$\bigcup_{i \in I} A_i = \{a \mid \exists j \in I : a \in A_j\}$$

$$\bigcap_{i \in I} A_i = \{a \mid \forall j \in I : a \in A_j\}$$

Examples:

$$\bigcup_{a \in A} \{a\} = A \text{ for any set } A$$

$$\bigcap_{n \in \mathbb{N}} \{m \in \mathbb{N} \mid m \geq n\} = \emptyset$$

Set Theory Basics

Definition (Cartesian Product)

For two sets A and B,

$$A \times B = \{(a, b) \mid a \in A \text{ and } b \in B\}$$

is the Cartesian product of A and B (the set of all pairs consisting of an element from A and an element from B).

More generally, for sets $A_1, \ldots, A_n$ ($n \geq 2$), let

$$\begin{aligned} \prod_{i=1}^{n} A_i &= A_1 \times A_2 \times \cdots \times A_n \\ &= \{(a_1, \ldots, a_n) \mid \text{for all } 1 \leq i \leq n,\ a_i \in A_i\} \end{aligned}$$

If $A_1 = A_2 = \cdots = A_n = A$, we also write A^n for this set.

Examples and Some Simple Statements:

- $\{1,2,3\} \times \{4,5\} = \{(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)\}$
- For all sets A, B, and C:

$$
\begin{aligned}
(A \cup B) \times C &= (A \times C) \cup (B \times C) \\
A \times (B \cup C) &= (A \times B) \cup (A \times C) \\
(A \cap B) \times C &= (A \times C) \cap (B \times C) \\
A \times (B \cap C) &= (A \times B) \cap (A \times C)
\end{aligned}
$$

Complete Induction (Review from DMI)

To prove a statement $P(n)$ for each natural number $n \in \mathbb{N}$, it suffices to show the following:

1. $P(0)$ holds (Base Case).
2. For every natural number $n \in \mathbb{N}$, if $P(n)$ holds, then $P(n+1)$ also holds (Inductive Step).

This principle of proof is called the principle of complete induction.

Example: We prove by complete induction that for all natural numbers n:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}.$$

Complete Induction (Review from DMI)

Base Case: We have $\sum_{i=1}^{0} i = 0 = \frac{0 \cdot 1}{2}$.

Inductive Step: Assume that

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}.$$

Then we also have

$$\begin{aligned}
\sum_{i=1}^{n+1} i &= \Big(\sum_{i=1}^{n} i\Big) + n + 1 \\
&= \frac{n(n+1)}{2} + n + 1 \\
&= \frac{n(n+1) + 2(n+1)}{2} \\
&= \frac{(n+1)(n+2)}{2}
\end{aligned}$$

Complete Induction (Review from DMI)

For base case (inductive step), we often write **BC** (**IS**) for short.

The principle of induction can also be used to define objects.

Suppose we want to define an object A_n for each natural number $n \in \mathbb{N}$.

This can be done as follows:

1. Define A_0.
2. Provide a general rule for constructing the object A_{n+1} from the (already constructed) objects $A_0, A_1, \dots, A_n$.

Words: Intuitive Understanding

The content from slides 12–44 can be found in Schöning's book on pages 3–18.

A central data structure in computer science consists of finite sequences of symbols, also known as words or strings.

Examples:

1. A byte is a sequence of 8 bits, e.g., 00110101.
2. A German or English text is a sequence consisting of the symbols $a, b, c, \ldots, z, A, B, C, \ldots, Z, 1, 2, \ldots, 9$, ␣ (blank) and punctuation marks ., ! , ? as well as , .
3. A gene is a sequence of the symbols A, G, T, C (4 DNA bases).

Words: Formal Definition

Definition (Alphabet, Words)

An alphabet is a finite, non-empty set.

A word over the alphabet Σ is a finite sequence of symbols in the form $a_1 a_2 \cdots a_n$ with $a_i \in \Sigma$ for $1 \leq i \leq n$. The length of this word is n.

For a word w, we also write $|w|$ to denote the length of the word w.

For $n = 0$, we obtain the empty word (the word of length 0), denoted by ε.

We use Σ^* to denote the set of all words over the alphabet Σ.

The set of all non-empty words is $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$.

Words

Example 1: Let $\Sigma = \{a, b, c\}$. Then possible words from Σ^* are:

$$\varepsilon, a, b, aa, ab, bc, bbbab, \ldots$$

For the lengths, we have $|\varepsilon| = 0$, $|a| = |b| = 1$, $|aa| = |ab| = |bc| = 2$, and $|bbbab| = 5$.

Example 2: A genome is a word over the alphabet $\{A, G, T, C\}$.

Remark: It is often asked what the empty word ε is used for.

The empty word will prove useful in many contexts. The empty word ε can be compared to the number $0 \in \mathbb{N}$. In fact, it has similar properties to the number 0.

Words

Conventions: Words from Σ^* are denoted with lowercase letters (from the latter half of the alphabet): u, v, w, x, y, z, ...

Definition (Concatenation of Words)

For words $u = a_1 \cdots a_m$ and $v = b_1 \cdots b_n$ with $a_1, \ldots, a_m, b_1, \ldots, b_n \in \Sigma$, the word

$$u \circ v = a_1 \cdots a_m b_1 \cdots b_n$$

is the concatenation (or juxtaposition) of the words u and v.

Instead of $u \circ v$, we usually write just uv.

Words

It is clear that for all words $u, v, w \in \Sigma^*$:

- $(u \circ v) \circ w = u \circ (v \circ w)$ or simply $(uv)w = u(vw)$ (Associativity Law)
- $\varepsilon \circ u = u = u \circ \varepsilon$

We also write $(uv)w = u(vw)$ simply as uvw.

Reminder from DMI: $(\Sigma^*, \circ)$ is a monoid, also called the free monoid generated by Σ. The empty word ε is the identity element.

Note: For words u and v, in general, $uv \neq vu$.

For example, $ab \neq ba$ for $a, b \in \Sigma$ with $a \neq b$.

Concatenation of words is not commutative.

Words

Assume that Σ is an alphabet with n symbols: $|\Sigma| = n$.

Then there are exactly n^k words of length k over the alphabet Σ:

$$|\{w \in \Sigma^* \mid |w| = k\}| = n^k.$$

Justification: For the first symbol in a word, there are exactly n possibilities, for the second symbol there are also n possibilities, and so on. In total, there are

$$\underbrace{n \cdot n \cdot n \cdots n}_{k \text{ times}} = n^k$$

possibilities.

For the set $\{w \in \Sigma^* \mid |w| = k\}$ (the set of all words of length k), we also write Σ^k.

Languages

In the context of natural languages (e.g., German or English), a language can be defined as the set of all words over the alphabet from Example 2, Slide 12, that form a correct sentence.

For example, the string *Der_Hund_jagt_die_Katze.* would be an element of the German language.

Definition (Language)

Let Σ be an alphabet.
A (formal) language L over the alphabet Σ is any subset of Σ^*, i.e. $L \subseteq \Sigma^*$.

Example: Let $\Sigma = \{(,),+,-,*,/,a\}$. We can define the language *EXPR* of correctly parenthesized expressions. For example:

- $(a-a)*a+a/(a+a)-a \;\in\; EXPR$
- $(((a))) \;\in\; EXPR$
- $((a+)-a(\;\notin\; EXPR$

Grammars (Introduction)

Grammars in computer science are similar to grammars for natural languages and serve as a means to generate all syntactically correct sentences (here: words) of a language.

Example: Grammar for generating elements from $EXPR$:

$$
\begin{aligned}
E &\rightarrow a \\
E &\rightarrow E + E \\
E &\rightarrow E - E \\
E &\rightarrow E * E \\
E &\rightarrow E / E \\
E &\rightarrow (E)
\end{aligned}
$$

Grammars (Introduction)

Using this (finite) grammar, it is possible to derive elements from *EXPR*.

Example:

$$E \to E * E \to (E) * E \to (E + E) * E \to (a + a) * a$$

Clearly, with the grammar, one can generate infinitely many words.

This means that the language corresponding to the grammar (also called the language generated by the grammar) is infinite.

Grammars (Definition)

Grammars have productions of the form

$$\textit{left side} \;\rightarrow\; \textit{right side}$$

Both the left and right sides can contain two types of symbols:

- Non-terminals (the variables, from which further word components are to be derived)
- Terminals (the äctualßymbols)

In the previous example: the left side always contains exactly one non-terminal; this is referred to as a context-free grammar.

However, there are also more general grammars.
There are even grammars that work with trees and graphs instead of words. These are not covered in this lecture.

Grammars (Definition)

Definition (Grammar, Sentence Form)

A Grammar G is a 4-tuple $G = (V, \Sigma, P, S)$, which satisfies the following conditions:

- V is an alphabet (set of non-terminals or variables).
- Σ is an alphabet (set of terminal symbols) with $V \cap \Sigma = \emptyset$, i.e., no symbol is both terminal and non-terminal.
- $P \subseteq ((V \cup \Sigma)^+ \setminus \Sigma^*) \times (V \cup \Sigma)^*$ is a finite set of productions.
- $S \in V$ is the start variable (axiom).

A word from $(V \cup \Sigma)^*$ is also called a sentence form.

Grammars (Definition)

A production from P is a pair (ℓ, r) of words over $V \cup \Sigma$, typically written as $\ell \to r$. The following applies:

- Both ℓ and r consist of variables and terminal symbols.
- ℓ must not consist solely of terminals. A rule must always replace at least one non-terminal.

Conventions:

- Variables (elements from V) are denoted by uppercase letters: A, B, C, ..., S, T, ...
- Terminal symbols (elements from Σ) are represented by lowercase letters: a, b, c, ...

Grammars (Example)

Example Grammar

$G = (V, \Sigma, P, S)$ with

- $V = \{S, B, C\}$
- $\Sigma = \{a, b, c\}$
- $P = \{S \to aSBC, S \to aBC, CB \to BC, aB \to ab,$
  $bB \to bb, bC \to bc, cC \to cc\}$

Grammars (Derivations)

How are the productions applied to generate words from the start variable S?

Definition (Derivation Step)

Let $G = (V, \Sigma, P, S)$ be a grammar and let $u, v \in (V \cup \Sigma)^*$. It holds that:

$$u \Rightarrow_G v \text{ (} u \text{ directly goes to } v \text{ under } G\text{)},$$

if there exists a production $(\ell \to r) \in P$ and words $x, y \in (V \cup \Sigma)^*$ such that

$$u = x\ell y \qquad v = xry.$$

One can interpret $\Rightarrow_G$ as a binary relation on $(V \cup \Sigma)^*$, i.e., as a subset of $(V \cup \Sigma)^* \times (V \cup \Sigma)^*$:

$$\Rightarrow_G = \{(u, v) \mid \exists (\ell \to r) \in P \ \exists x, y \in (V \cup \Sigma)^* : u = x\ell y, v = xry\}$$

Grammars (Derivations)

Instead of $u \Rightarrow_G v$, one also writes $u \Rightarrow v$ when it is clear which grammar is being referred to.

Definition (Derivation)

A sequence of words $w_0, w_1, w_2, \dots, w_n$ with $w_0 = S$ and

$$w_0 \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \cdots \Rightarrow w_n$$

is called a derivation of w_n (from S). Here, w_n may contain both terminal symbols and variables, thus it is a sentence form.

Here is a derivation of *aabbcc* from S using the grammar G from Slide 24:

$$S \Rightarrow aSBC \Rightarrow aaBCBC \Rightarrow aaBBCC \Rightarrow aabBCC$$
$$\Rightarrow aabbCC \Rightarrow aabbcC \Rightarrow aabbcc$$

Definition (the language generated by a grammar)

The language generated (represented, defined) by a grammar $G = (V, \Sigma, P, S)$ is

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

Here, $\Rightarrow_G^*$ is the reflexive and transitive closure of $\Rightarrow_G$, i.e., $u \Rightarrow_G^* v$ holds if and only if $n \geq 0$ and sentence forms $u_0, u_1, \ldots, u_n \in (V \cup \Sigma)^*$ exist such that: $u_0 = u$, $u_n = v$, and $u_i \Rightarrow_G u_{i+1}$ for all $0 \leq i \leq n-1$.

In other words: The language generated by G, $L(G)$, consists exactly of the sentence forms that can be derived from S in any number of steps, and that consist solely of terminal symbols.

The previous example grammar G (Slide 24) generates the language

$$L(G) = \{a^n b^n c^n \mid n \geq 1\}.$$

Here, $a^n = \underbrace{a \dots a}_{n \text{ times}}$.

The claim that G indeed generates this language is not immediately obvious.

Remark: Derivation is not a deterministic, but a non-deterministic process. For a $u \in (V \cup \Sigma)^*$, there may be no, one, or multiple v such that $u \Rightarrow_G v$.

In other words: $\Rightarrow_G$ is not a function.

This non-determinism can be caused by two different effects ...

- A rule can be applied in two different places.

  Example grammar:

$$aaaSB\,\textcolor{blue}{CB}\,\textcolor{red}{CB}\,C \Longrightarrow aaaSB\,\textcolor{blue}{BC}\,CBC$$
$$aaaSB\,\textcolor{blue}{CB}\,\textcolor{red}{CB}\,C \Longrightarrow aaaSBCB\,\textcolor{red}{BC}\,C$$

- Two different productions can be applied (either at the same place – as shown below – or at different places):

  Example grammar:

$$S \Longrightarrow aSBC$$
$$S \Longrightarrow aBC$$

Further Remarks:

- There can be arbitrarily long derivations that never lead to a word made up of terminal symbols:

$$S \Rightarrow aSBC \Rightarrow aaSBCBC \Rightarrow aaaSBCBCBC \Rightarrow \dots$$

- Sometimes, derivations may end in a dead end, i.e., although variables still appear in a sentence form, no rule is applicable anymore.

$$S \Rightarrow aSBC \Rightarrow aaBCBC \Rightarrow aabCBC \Rightarrow aabcBC \not\Rightarrow$$

Chomsky Hierarchy

Type 0 – Chomsky-0

Every grammar is of type 0 (no restriction on productions).

Type 1 – Chomsky-1

A grammar $G = (V, \Sigma, P, S)$ is of type 1 (or monotonic, context-sensitive), if $|\ell| \leq |r|$ for all productions $(\ell \to r) \in P$.

Type 2 – Chomsky-2

A grammar $G = (V, \Sigma, P, S)$ is of type 2 (or context-free) if it is (i) of type 1 and (ii) additionally, $\ell \in V$ for every production $(\ell \to r) \in P$.
In particular, it must hold that $|r| \geq |\ell| = 1$.

Chomsky Hierarchy

Type 3 – Chomsky-3

A grammar $G = (V, \Sigma, P, S)$ is of type 3 (or regular) if it is (i) of type 2 and (ii) additionally for all productions $(A \to r) \in P$, it holds that: $r \in \Sigma$ or $r = aB$ with $a \in \Sigma, B \in V$.

That is, the right-hand sides of productions are either individual terminals or a terminal followed by a variable.

Type-i Language

A language $L \subseteq \Sigma^*$ is of type i ($i \in \{0, 1, 2, 3\}$) if there exists a type-i grammar G such that $L(G) = L$.

Such languages are also called semi-decidable or recursively enumerable (type 0), context-sensitive (type 1), context-free (type 2), or regular (type 3).

Remarks:

- Where does the name “context-sensitive” come from?

  In context-free grammars, there are only productions of the form $A \to x$, where $A \in V$ and $x \in (\Sigma \cup V)^*$. This means: A can be replaced by x independently of the context.

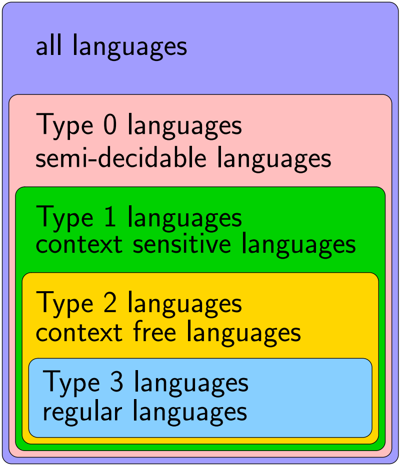
  In the more powerful context-sensitive grammars, however, productions of the form $uAv \to uxv$ are possible, with the meaning: A can only be replaced by x in certain contexts.

- ε-Special Rule: In type-1 grammars (and thus also in regular and context-free grammars), productions of the form $\ell \to \varepsilon$ are initially not allowed, due to $|\ell| > 0$ and $|\ell| \leq |r|$ for all $(\ell \to r) \in P$. This means that the empty word ε cannot be derived!

  Therefore, we slightly modify the grammar definition for type-1 (and type-2, type-3) grammars and allow $S \to \varepsilon$, if S is the start symbol and does not appear on any right-hand side.

Every type-i grammar is a type-$(i-1)$ grammar (for $i \in \{1, 2, 3\}$) $\rightsquigarrow$ the corresponding sets of languages are nested.

Furthermore: the inclusions are strict, i.e., for each i there exists a type-$(i-1)$ language that is not a type-i language (e.g., a context-free language that is not regular). We will show this later.



Word Problem

Definition (Word Problem)

Let $G = (V, \Sigma, P, S)$ be a grammar (of any type). The word problem for $L(G)$ is the following decision problem:
INPUT: A word $w \in \Sigma^*$.
QUESTION: Is it true that $w \in L(G)$?

Theorem (Decidability of the Word Problem for Type 1)

There exists an algorithm that, given as input a type-1 grammar $G = (V, \Sigma, P, S)$ and a word $w \in \Sigma^*$, outputs "Yes" (or "No") in finite time if $w \in L(G)$ (or $w \notin L(G)$) holds.

It is also said: The word problem is decidable for type-1 languages (a more detailed definition will come later in the lecture).

Word Problem

Proof:

If $w = \varepsilon$, we only need to check whether $S \to \varepsilon$ is a production.

If yes, then $w \in L(G)$, otherwise $w \notin L(G)$.

Now, assume $w \neq \varepsilon$ and let $n = |w| \geq 1$.

We define a directed finite graph $\mathcal{G}$ as follows:

- The set of nodes of $\mathcal{G}$ is the set

$$K := \{u \in (V \cup \Sigma)^+ \mid |u| \leq n\}$$

  of all sentence forms of length at most n.

- For $u, v \in K$, there is an edge $u \to v$ if $u \Rightarrow_G v$ holds.

Note: $|K| = \sum_{i=1}^{n} (|V| + |\Sigma|)^i$.

Word Problem

Since G is a Type-1 grammar, we have: $w \in L(G)$ if and only if there is a path in the graph $\mathcal{G}$ from the node $S \in K$ to the node $w \in K$.

Justification: When deriving a word of length $n \geq 1$ from the start symbol using a Type-1 grammar, no sentence form of length greater than n appears in the derivation (this is generally not true for Type-0 grammars).

One constructs the graph $\mathcal{G}$ by iterating through all nodes in K in a for-loop, and for each node $u \in K$, generating the set $\{v \mid u \Rightarrow_G v\}$ of all direct successor nodes of u.

Using Depth-First Search ($\rightsquigarrow$ *Algorithms & Data Structures* lecture), one can then test whether there is a path in the graph $\mathcal{G}$ from S to w. $\square$

Word Problem

Remark: This algorithm is not very efficient, as the size of the constructed graph grows exponentially with the length of the input word w (this is referred to as an exponential-time algorithm).

However, it is believed that this is not avoidable:

The word problem for Type-1 grammars is a so-called PSPACE-complete problem, see the lecture *Complexity Theory I*.

For PSPACE-complete problems, no algorithms with polynomial time complexity are known.

Syntax Trees and Uniqueness

We consider the following example grammar (a Type-2 grammar) for generating correctly parenthesized arithmetic expressions:

$$G = (\{E, T, F\}, \{(,), a, +, *\}, P, E)$$

with the following production set P (in abbreviated Backus-Naur Form):

$$\begin{array}{lcl} E & \to & T \mid E + T \\ T & \to & F \mid T * F \\ F & \to & a \mid (E) \end{array}$$

In Backus-Naur Form for Type-2 grammars, multiple productions are written

$$A \to w_1, A \to w_2, \ldots, A \to w_k \tag{1}$$

in the form

$$A \to w_1 \mid w_2 \mid \cdots \mid w_k.$$

This is just an abbreviation for (1).

Syntax Trees and Uniqueness

For most words of the language generated by G, there are multiple possible derivations:

$$\begin{aligned}
&E \Rightarrow T \Rightarrow T * F \Rightarrow F * F \Rightarrow a * F \Rightarrow a * (E) \\
&\Rightarrow a * (E + T) \Rightarrow a * (T + T) \Rightarrow a * (F + T) \\
&\Rightarrow a * (a + T) \Rightarrow a * (a + F) \Rightarrow a * (a + a)
\end{aligned}$$

$$\begin{aligned}
&E \Rightarrow T \Rightarrow T * F \Rightarrow T * (E) \Rightarrow T * (E + T) \\
&\Rightarrow T * (E + F) \Rightarrow T * (E + a) \Rightarrow T * (T + a) \\
&\Rightarrow T * (F + a) \Rightarrow T * (a + a) \Rightarrow F * (a + a) \Rightarrow a * (a + a)
\end{aligned}$$

The first derivation is a left derivation (in each step, the leftmost non-terminal is replaced), and the second one is a right derivation (in each step, the rightmost non-terminal is replaced).

Syntax Trees and Uniqueness

We now form the syntax tree from both derivations by:

- Labeling the root of the tree with the start variable of the grammar.
- For each application of a production $A \to z$, adding exactly $|z|$ children to A, labeled with the symbols from z.

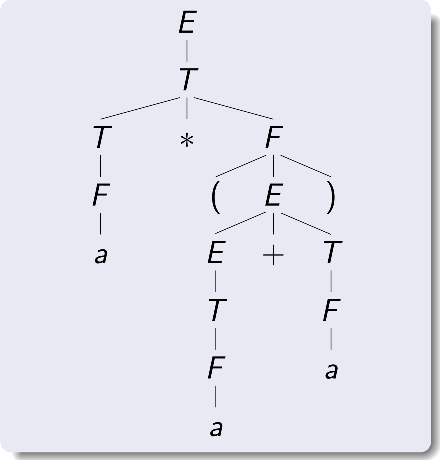
Syntax trees can be constructed for all derivations of context-free grammars.

In both cases, we obtain the same syntax tree.

A grammar is called unambiguous if for every word in the generated language, there is exactly one syntax tree

$\iff$ there is exactly one left derivation for every word

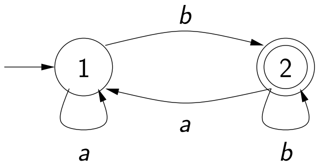
$\iff$ there is exactly one right derivation for every word.



Finite Automata

The content of slides 44–88 can be found in Schöning's book on pages 19–27.

In this section, we focus on regular languages, but from a different perspective. Instead of Type-3 grammars, we consider state-based automaton models, which can also be viewed as "language generators" or "language acceptors."



Deterministic Finite Automata

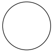
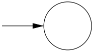
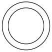
Definition (Deterministic Finite Automaton)

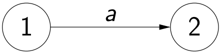
A (deterministic) finite automaton M is a 5-tuple $M = (Z, \Sigma, \delta, z_0, E)$, where:

- Z is a finite set of states,
- Σ is the finite input alphabet (with $Z \cap \Sigma = \emptyset$),
- $z_0 \in Z$ is the start state,
- $E \subseteq Z$ is the set of accepting states,
- $\delta \colon Z \times \Sigma \to Z$ is the transition function.

Abbreviation: DFA (deterministic finite automaton)

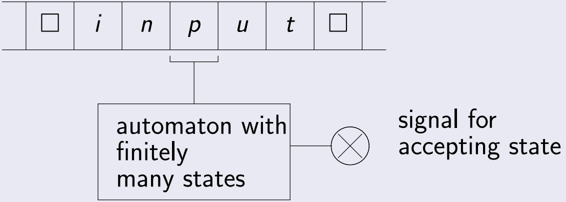
graphical notation:

state:    start state:    accepting state: 

transition $\delta(1, a) = 2$: 

Where does the name finite automatoncome from?

Imagine a machine that can be in a finite number of states, reads an input, and signals when the input is accepted.



Analogy to a Ticket Machine:

A ticket machine can be in the following states:

- No input
- Destination selected
- Money entered
- Ticket issued

Of course, this is only part of the truth, as a ticket machine needs to keep track of how much money has been inserted. Modeling it with only finitely many states is therefore a significant simplification.

Deterministic Finite Automata

From a rather abstract standpoint, any real computer can be considered a DFA:

- The set of states is the set of all possible memory configurations.

  If the entire memory of the computer consists of n bits, then there are 2^n possible memory configurations (you can think of a memory configuration as a word from $\{0,1\}^n$).

  Example: A computer with 8 GB of RAM and 512 GB of hard drive storage can store a total of $8 \cdot 520 \cdot 1000^3 = 4160000000000$ bits and thus corresponds to a DFA with $2^{4160000000000}$ states!

- The initial state is the memory configuration in the factory state.

Deterministic Finite Automata

- The transition function is determined by the behavior of the computer in response to inputs.

  Suppose your computer only receives inputs via the keyboard.

  Then the input alphabet consists of the keys on the keyboard.

  If the computer is in a particular memory state and a specific key is pressed (input), the computer transitions to a new state.

- Final states make less sense for a real computer, as computers are not typically used to accept words.

This perspective is, of course, far too abstract and entirely impractical for practical use, as seen by the $2^{4160000000000}$ states. However, it is still applied in smaller hardware components in the field of so-called hardware verification (see the master's course *Model-Checking* by Prof. Lochau).

Deterministic Finite Automata

The previous transition function δ of a DFA reads only one symbol at a time. We therefore generalize it to a transition function $\widehat{\delta}$ that determines transitions for entire words.

Definition (Multi-Step Transitions of a DFA)

For a given DFA $M = (Z, \Sigma, \delta, z_0, E)$, we define a function $\widehat{\delta}\colon Z \times \Sigma^* \to Z$ inductively as follows, where $z \in Z$, $x \in \Sigma^*$, and $a \in \Sigma$:

$$\begin{aligned} \widehat{\delta}(z, \varepsilon) &= z \\ \widehat{\delta}(z, ax) &= \widehat{\delta}(\delta(z, a), x) \end{aligned}$$

Intuition: $\widehat{\delta}(z, a_1 a_2 \cdots a_n)$ is the state reached from state z by first following the edge labeled with a_1, then following the edge labeled with a_2, and so on:

$$z \xrightarrow{a_1} z_1 \xrightarrow{a_2} z_2 \xrightarrow{a_3} \cdots \xrightarrow{a_n} z_n = \widehat{\delta}(z, a_1 a_2 \cdots a_n).$$

Deterministic Finite Automata

Without always mentioning it explicitly, we often use the following easily proven statement:

Lemma 1

For all words $u, v \in \Sigma^*$ and every state $z \in Z$, it holds that:

$$\widehat{\delta}(z, uv) = \widehat{\delta}(\widehat{\delta}(z, u), v).$$

Definition (Language Accepted by a DFA)

The accepted language of a DFA $M = (Z, \Sigma, \delta, z_0, E)$ is

$$T(M) = \{x \in \Sigma^* \mid \widehat{\delta}(z_0, x) \in E\}.$$

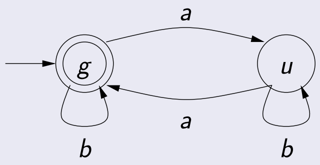
In other words:

The language can be obtained by following all paths from the start state to an end state, collecting all symbols on the transitions.

Example 1: We are looking for a DFA that accepts the following language L:

$$L = \{w \in \{a,b\}^* \mid \#_a(w) \text{ is even}\}.$$

Here, $\#_a(w)$ is the number of a's in w.



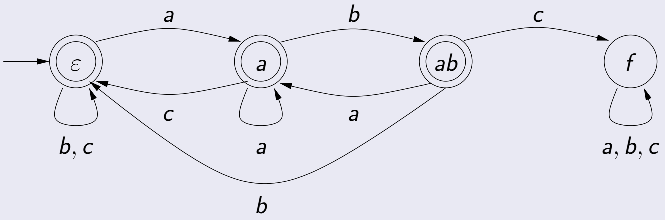
Meaning of the States:

g – even number of a's

u – odd number of a's

Example 2: We are looking for a DFA M such that

$$T(M) = \{w \in \{a, b, c\}^* \mid \text{the substring } abc \text{ does not appear in } w\}.$$



Meaning of the States:

- ε: no prefix of abc read
- a: last read character was an a
- ab: last read characters were ab
- f: abc appeared in the word read so far (trap state, error state)

Deterministic Finite Automata

Theorem (DFAs $\rightarrow$ Regular Grammar)

Every language accepted by a DFA is regular.

Remark: The converse statement also holds: every regular language can be accepted by a DFA (more on this later).

Proof:

Let $M = (Z, \Sigma, \delta, z_0, E)$ be a DFA.

First, we modify M so that no edges lead into the initial state, i.e.,

$$\delta(z, a) \neq z_0$$

for all $z \in Z$ and $a \in \Sigma$.

Idea: We introduce a copy z_0' of the initial state z_0 into the DFA, which has the same outgoing edges as z_0. Then we redirect all edges that lead to state z_0 to z_0'.

Formally: Let $z_0' \notin Z$ be a new state, and let $Z' = Z \cup \{z_0'\}$.

Deterministic Finite Automata

Let $M' = (Z', \Sigma, \delta', z_0, E')$, where:

$$\delta'(z,a) = \begin{cases} \delta(z,a) & \text{if } z \in Z \text{ and } \delta(z,a) \neq z_0 \\ z_0' & \text{if } z \in Z \text{ and } \delta(z,a) = z_0 \end{cases}$$

$$\delta'(z_0',a) = \begin{cases} \delta(z_0,a) & \text{if } \delta(z_0,a) \neq z_0 \\ z_0' & \text{if } \delta(z_0,a) = z_0 \end{cases}$$

$$E' = \begin{cases} E & \text{if } z_0 \notin E \\ E \cup \{z_0'\} & \text{if } z_0 \in E \end{cases}$$

Then:

- $\delta'(z,a) \neq z_0$ for all $z \in Z'$ and $a \in \Sigma$, and
- $T(M') = T(M)$.

Deterministic Finite Automata

We now revert to using Z, δ, E for Z', δ', E'.

We define a Type-3 grammar $G = (V, \Sigma, P, S)$ with $L(G) = T(M)$ as follows:

$$\begin{array}{rcl} V & = & Z \\ S & = & z_0 \\ P & = & \{z \to a\ \delta(z,a) \mid z \in Z, a \in \Sigma\} \cup \\ & & \{z \to a \mid z \in Z, a \in \Sigma, \delta(z,a) \in E\} \cup \\ & & \{z_0 \to \varepsilon\} \text{ if } z_0 \in E \end{array}$$

Note: The ε special condition is fulfilled.

Claim 1: For all $z, z' \in Z$ and $w \in \Sigma^*$, it holds that:

$$z \Rightarrow_G^* wz' \quad \Longleftrightarrow \quad \widehat{\delta}(z,w) = z'.$$

Claim 1 is proven by induction over $|w|$.

Base Case: $|w| = 0$, i.e., $w = \varepsilon$. We have

$$z \Rightarrow_G^* z' \;\Leftrightarrow\; z = z' \;\Leftrightarrow\; \widehat{\delta}(z, \varepsilon) = z'$$

Inductive Step: Now let $|w| = n + 1$.

Then we can write w as $w = av$ with $|v| = n$ and $a \in \Sigma$.

Inductive Hypothesis: Claim 1 holds for v.

It follows that:

$$\begin{aligned}
z \Rightarrow_G^* avz' \quad &\Longleftrightarrow \quad \exists z'' \in Z : (z \to az'') \in P \text{ and } z'' \Rightarrow_G^* vz' \\
&\Longleftrightarrow \quad \delta(z, a) \Rightarrow_G^* vz' \\
&\overset{\text{Ind. Hyp.}}{\Longleftrightarrow} \quad \widehat{\delta}(\delta(z, a), v) = z' \\
&\Longleftrightarrow \quad \widehat{\delta}(z, av) = z'
\end{aligned}$$

Claim 2: For all $w \in \Sigma^*$, we have: $w \in L(G) \iff w \in T(M)$.

Case 1: $w = \varepsilon$.

We have:

$$\varepsilon \in L(G) \iff (z_0 \to \varepsilon) \in P \iff z_0 \in E \iff \varepsilon \in T(M)$$

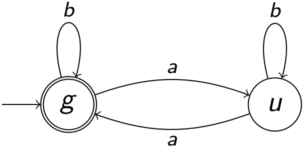
Case 2: $w \neq \varepsilon$.

Let $w = va$ with $a \in \Sigma$ and $v \in \Sigma^*$. Then:

$$\begin{aligned}
va \in L(G) \quad &\iff \quad \exists z \in Z : z_0 \Rightarrow_G^* vz \Rightarrow_G va \\
&\overset{\text{Claim 1}}{\iff} \quad \exists z \in Z : \widehat{\delta}(z_0, v) = z, \widehat{\delta}(z, a) \in E \\
&\iff \quad \widehat{\delta}(z_0, va) \in E \\
&\iff \quad va \in T(M)
\end{aligned}$$

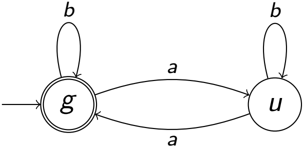
Deterministic Finite Automata

Example: Consider the DFA from Slide 54:

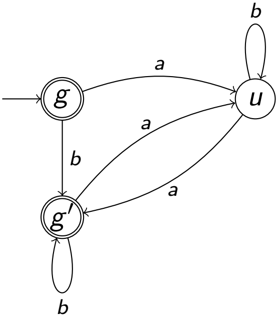


Deterministic Finite Automata

Example: Consider the DFA from Slide 54:



The construction from Slides 57–58 results in the following DFA:



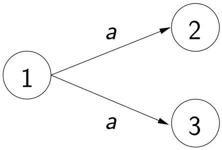
Example (Continuation): The construction from Slide 59 gives the Type-3 grammar $G = (V, \{a, b\}, P, S)$ with:

- $V = \{g, u, g'\}$,
- $S = g$,
- P consists of the following productions:

$$\begin{array}{lll} g \to \varepsilon & u \to a & g' \to b \\ g \to b & u \to bu & g' \to au \\ g \to au & u \to ag' & g' \to bg' \\ g \to bg' & & \end{array}$$

In contrast to grammars, there are no non-deterministic effects in DFAs. That is, once the next symbol is read, the next state is determined.

However: In many cases, it is more natural to allow non-deterministic transitions. This often leads to smaller automata.



Non-deterministic Finite Automata

Definition (Non-deterministic Finite Automaton)

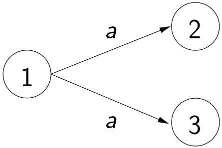
A non-deterministic finite automaton M is a 5-tuple $M = (Z, \Sigma, \delta, S, E)$, where:

- Z is a finite set of states,
- Σ is the finite input alphabet (with $Z \cap \Sigma = \emptyset$),
- $S \subseteq Z$ is the set of start states,
- $E \subseteq Z$ is the set of end states, and
- $\delta \colon Z \times \Sigma \to 2^Z$ is the transition function (or transition function).

Abbreviation: NFA (nondeterministic finite automaton)

To recall: $2^Z = \{A \mid A \subseteq Z\}$ is the power set of Z.

Example: $\delta(1, a) = \{2, 3\}$



The transition function δ can again be extended to a multi-step transition function:

Definition (Multi-step transitions of an NFA)

For a given NFA $M = (Z, \Sigma, \delta, S, E)$, we define a function

$$\widehat{\delta} \colon 2^Z \times \Sigma^* \to 2^Z$$

inductively as follows, where $Y \subseteq Z$, $x \in \Sigma^*$, and $a \in \Sigma$:

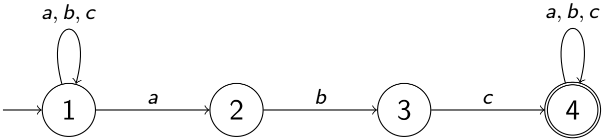
$$\begin{aligned} \widehat{\delta}(Y, \varepsilon) &= Y \\ \widehat{\delta}(Y, ax) &= \widehat{\delta}\left(\bigcup_{z \in Y} \delta(z, a),\ x \right) \end{aligned}$$

Note: The set

$$\bigcup_{z \in Y} \delta(z, a) = \{z' \in Z \mid \exists z \in Y : z' \in \delta(z, a)\}$$

contains all the states reachable from any state in Y by applying a.

Example: For the NFA



It holds that $\widehat{\delta}(\{1\}, abca) = \{1, 2, 4\}$ and $\widehat{\delta}(\{2, 3\}, abca) = \emptyset$.

Non-deterministic Finite Automata

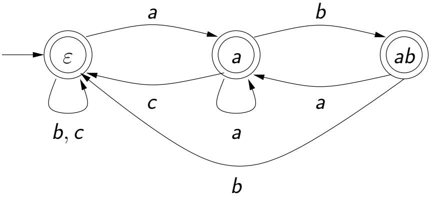
Definition (Language Accepted by an NFA)

The language accepted by an NFA $M = (Z, \Sigma, \delta, S, E)$ is

$$T(M) = \{x \in \Sigma^* \mid \widehat{\delta}(S, x) \cap E \neq \emptyset\}.$$

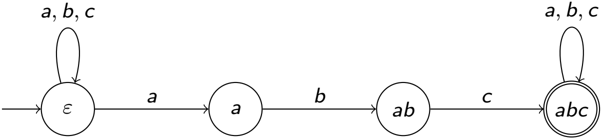
In other words: a word x is accepted if and only if there is a path from a start state to an accepting state, with transitions marked by the symbols of x (there may be multiple such paths).

Example 1: In non-deterministic automata, it is also allowed that $\delta(z, a) = \emptyset$ for some $a \in \Sigma$, meaning that it is not required for each alphabet symbol to always have a transition, and the dead state can be omitted.



Example 2: We seek an NFA that accepts the language

$$L = \{w \in \{a, b, c\}^* \mid \text{the substring } abc \text{ occurs in } w\}.$$



This automaton non-deterministically decides at some point that the substring *abc* is starting.

Non-deterministic Finite Automata

Remark: Real computers are always deterministic: the next state is uniquely determined by the current state and the input.

So why do we need non-determinism at all?

- NFAs allow a smaller representation of regular languages in many cases compared to DFAs. A concrete example will be shown on slides 81–83.
- NFAs can model systems where we do not have complete knowledge.
- Non-deterministic systems often arise through abstraction from real (deterministic) systems.
- Non-determinism also plays an important role in complexity theory, see the lecture *Complexity Theory I*.

Another Interpretation of Non-determinism:

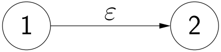
Each time a non-deterministic branch is possible, multiple parallel universesäre created, in which different copies of the machine explore the various possible paths.

The word is accepted if it is accepted in one of these parallel universes.

Non-deterministic Finite Automata

There are also non-deterministic automata with so-called ε-edges (spontaneous transitions where no alphabet symbol is read). These, however, are generally not used in this lecture.

Example of an ε-edge:



New transition function: $\delta\colon Z \times (\Sigma \cup \{\varepsilon\}) \to 2^Z$

In the above example: $\delta(1, \varepsilon) = \{2\}$.

Non-deterministic Finite Automata

New Multi-step Transition Function: $\widehat{\delta}\colon 2^Z \times \Sigma^* \to 2^Z$. Here, between the reading of symbols, arbitrary numbers of ε-transitions are allowed.

$$1 \xrightarrow{\varepsilon} 2 \xrightarrow{a} 3 \xrightarrow{\varepsilon} 4 \xrightarrow{\varepsilon} 5 \xrightarrow{b} 6 \xrightarrow{\varepsilon} 7 \xrightarrow{\varepsilon} 8$$

$$\widehat{\delta}(\{1\}, ab) = \{6, 7, 8\}$$

Equivalence of NFAs with and without ε-Transitions

Every NFA with ε-transitions can be converted into an NFA without ε-transitions, without changing the accepted language or increasing the number of states.

(Without proof.)

Theorem (NFAs $\to$ DFAs; Rabin, Scott)

Every language accepted by an NFA can also be accepted by a DFA.

Proof:

Idea: We simulate the various “parallel universes” of an automaton. It keeps track of the states it is currently in.

This means that the states of this automaton are sets of states from the original NFA. This construction is therefore called the powerset construction.

Let $M = (Z, \Sigma, \delta, S, E)$ be an NFA.

Define the DFA

$$M' = (2^Z, \Sigma, \gamma, S, F)$$

where

$$\begin{aligned} \gamma(Y, a) &= \bigcup_{z \in Y} \delta(z, a) \text{ for } Y \subseteq Z, a \in \Sigma \\ F &= \{Y \subseteq Z \mid Y \cap E \neq \emptyset\} \end{aligned}$$

Intuition: $\gamma(Y, a)$ is the set of all states $z' \in Z$ that can be reached from a state in Y by an a-transition.

By induction on the length of the word $w \in \Sigma^*$, we show for all $Y \subseteq Z$:

$$\widehat{\gamma}(Y, w) = \widehat{\delta}(Y, w)$$

Base Case: $\widehat{\gamma}(Y,\varepsilon) = Y = \widehat{\delta}(Y,\varepsilon)$

Inductive Step: Let $w = ax$ with $a \in \Sigma$ and $x \in \Sigma^*$. Then:

$$\begin{aligned}
\widehat{\gamma}(Y,ax) &= \widehat{\gamma}(\gamma(Y,a),x) \\
&\overset{\text{Ind. Hyp.}}{=} \widehat{\delta}(\gamma(Y,a),x) \\
&= \widehat{\delta}\left(\bigcup_{z\in Y}\delta(z,a),x\right) \\
&= \widehat{\delta}(Y,ax)
\end{aligned}$$

Therefore, for every word $w \in \Sigma^*$:

$$\begin{aligned}
w \in T(M') &\iff \widehat{\gamma}(S,w) \in F \\
&\iff \widehat{\delta}(S,w) \cap E \neq \emptyset \\
&\iff w \in T(M)
\end{aligned}$$

$\square$

Remark:

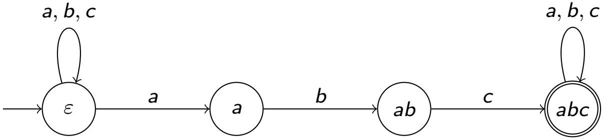
- The power set construction transforms an NFA with n states into an equivalent DFA with 2^n states.
- In many cases, not all of these 2^n states are needed.
- Therefore, it is advisable to only include the subsets of Z that are actually needed in the DFA during the power set construction.

On the next slide, we will construct an equivalent DFA for the NFA from Slide 71 step by step.

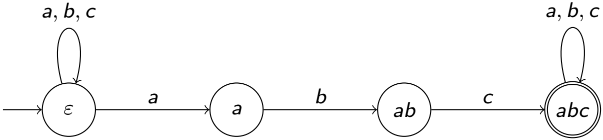
Only 6 of the $2^4 = 16$ possible subsets will be needed.

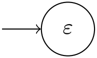
The node ε, ab abc represents the subset $\{\varepsilon, ab, abc\}$, for example.

Example 1:

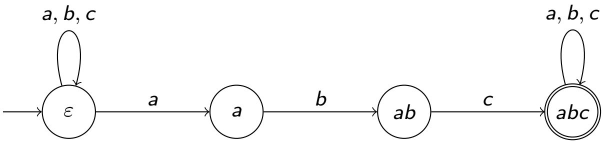


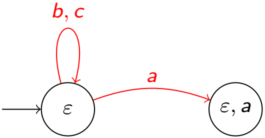
Example 1:



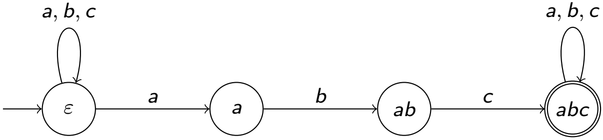


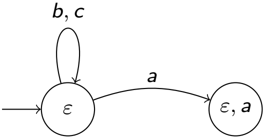
Example 1:



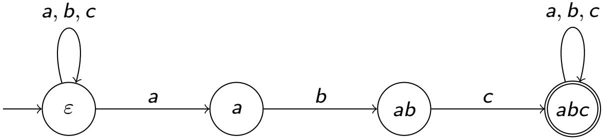


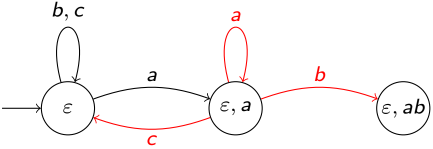
Example 1:



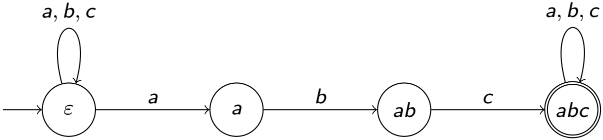


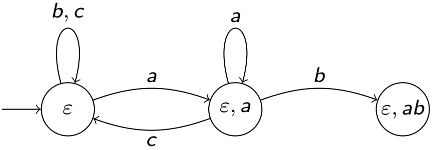
Example 1:



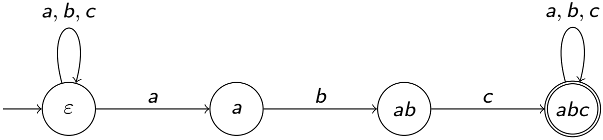


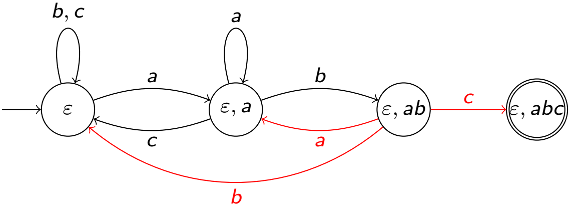
Example 1:



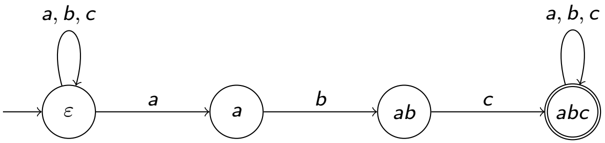


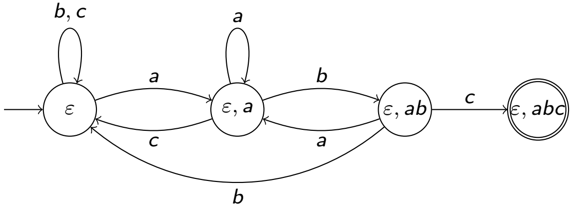
Example 1:



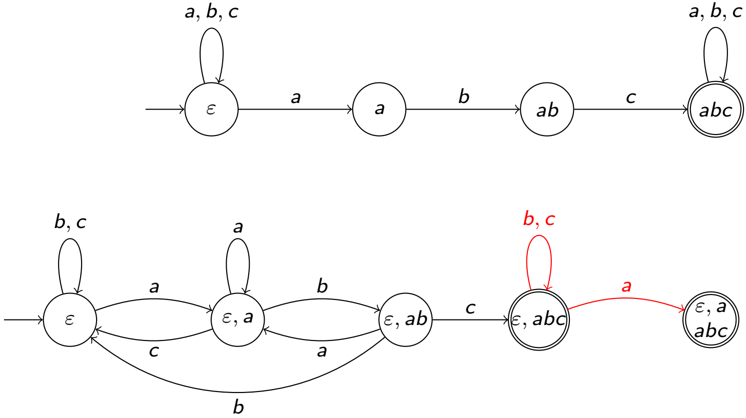


Example 1:

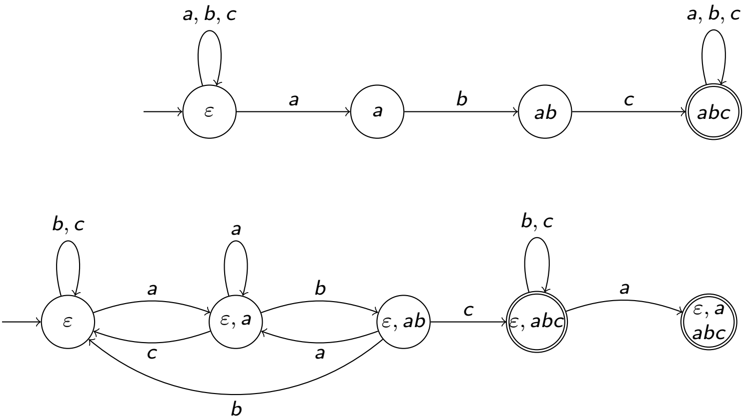




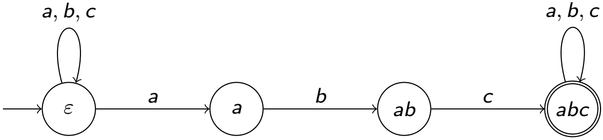
Example 1:

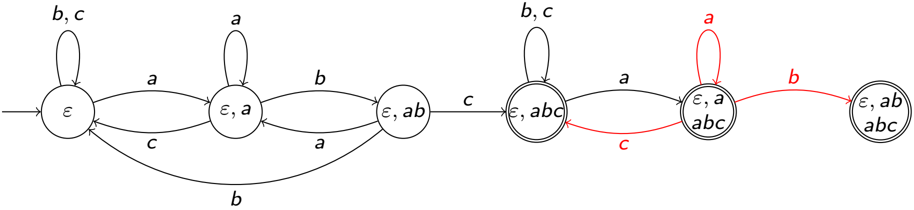


Example 1:

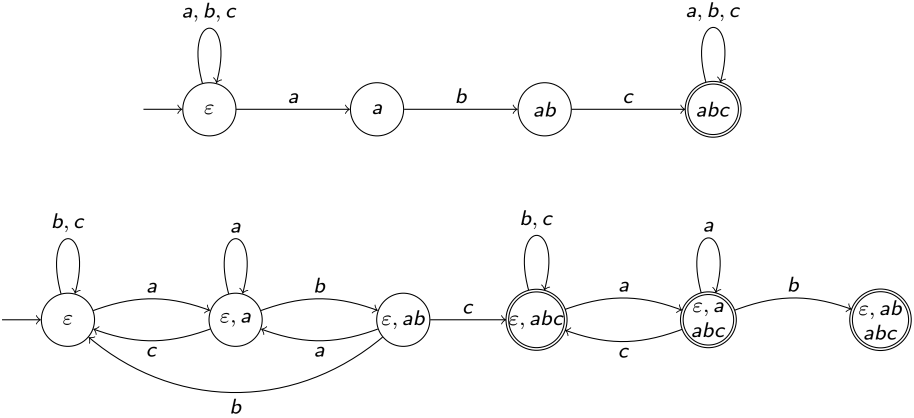


Example 1:


a, b, c
a
b
c
ε
a
ab
abc
a, b, c

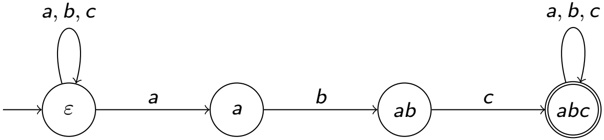

b, c
a
b, c
a
ε
a
ε, a
b
ε, ab
c
ε, abc
a
ε, a abc
b
ε, ab abc
c
a
c
b

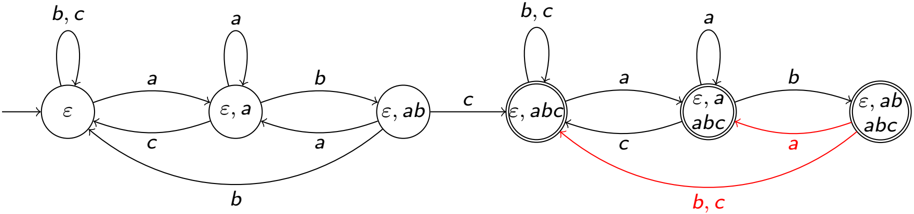
Example 1:


a, b, c
ε
a
a
b
ab
c
abc
a, b, c
b, c
ε
a
c
ε, a
a
b
a
ε, ab
b
c
b, c
ε, abc
a
c
ε, a
abc
a
b
ε, ab
abc

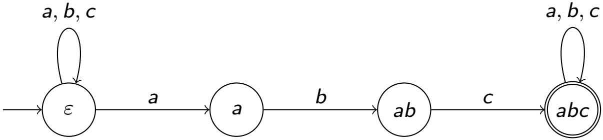
NFAs, DFAs, and Regular Grammars

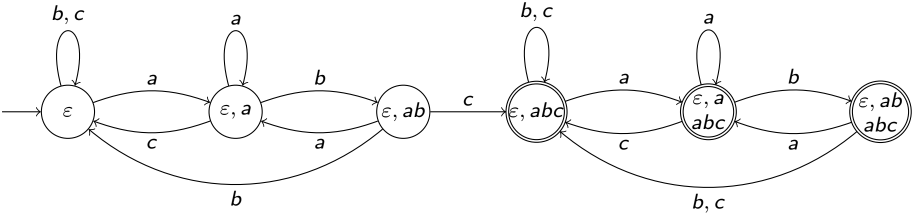
Example 1:





Example 1:

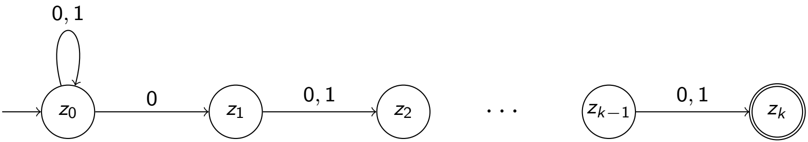

a, b, c
a
b
c
ε
a
ab
abc
a, b, c


b, c
a
a
b
c
ε
ε, a
ε, ab
ε, abc
c
a
b
b, c
a
a
b
ε, a abc
ε, ab abc
c
a
b, c

Example 2: For $k \geq 1$, define

$$L_k = \{w \in \{0,1\}^* \mid \ |w| \geq k, \text{ the } k\text{-th last character of } w \text{ is } 0\}.$$

(A) There exists an NFA M with $k+1$ states such that $T(M) = L_k$:



(B) There is no DFA M with fewer than 2^k states such that $T(M) = L_k$.

Proof of (B):

Assume that $M = (Z, \{0,1\}, \delta, z_0, E)$ is a DFA with fewer than 2^k states and $T(M) = L_k$.

Then, there exist words $w_1, w_2 \in \{0,1\}^k$ with $w_1 \neq w_2$ and $\widehat{\delta}(z_0, w_1) = \widehat{\delta}(z_0, w_2)$ (since there are 2^k possible words in $\{0,1\}^k$).

Let $i \in \{1, \ldots, k\}$ be the first position where w_1 and w_2 differ.

Let $w \in \{0,1\}^{i-1}$ be arbitrary.

Then, there exist words $v, v' \in \{0,1\}^{k-i}$ and $u \in \{0,1\}^{i-1}$ such that (without loss of generality)

$$w_1 w = u0vw \quad \text{and} \quad w_2 w = u1v'w.$$

Since $|vw| = |v'w| = k - i + i - 1 = k - 1$, it follows that

$$w_1 w \in L_k \quad \text{and} \quad w_2 w \notin L_k.$$

But:

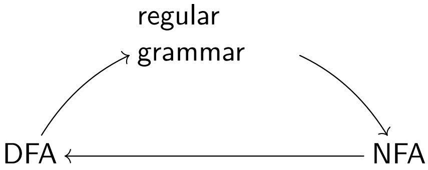
$$\widehat{\delta}(z_0, w_1 w) = \widehat{\delta}(\widehat{\delta}(z_0, w_1), w) = \widehat{\delta}(\widehat{\delta}(z_0, w_2), w) = \widehat{\delta}(z_0, w_2 w),$$

meaning $w_1 w \in L_k \Leftrightarrow w_2 w \in L_k$. Contradiction!

We can now

- convert NFAs into DFAs, and
- convert DFAs into regular grammars.

What remains is the direction "regular grammar $\rightarrow$ NFA", and then we will have shown the equivalence of all these formalisms.



Theorem (Regular Grammars $\to$ NFAs)

For every regular grammar G, there exists an NFA M such that $L(G) = T(M)$.

Proof:

Let $G = (V, \Sigma, P, S)$ be a regular grammar.

We define the NFA $M = (V \cup \{X\}, \Sigma, \delta, \{S\}, E)$, where $X \notin V$ and

$$\delta(A, a) = \{B \mid (A \to aB) \in P\} \cup \{X \mid (A \to a) \in P\} \text{ for } A \in V, a \in \Sigma$$

$$\delta(X, a) = \emptyset \text{ for } a \in \Sigma$$

$$E = \begin{cases} \{S, X\} & \text{if } (S \to \varepsilon) \in P \\ \{X\} & \text{if } (S \to \varepsilon) \notin P \end{cases}$$

Due to the construction, we have

$$\varepsilon \in L(G) \iff (S \to \varepsilon) \in P \iff \{S\} \cap E \neq \emptyset \iff \varepsilon \in T(M).$$

Thus, we still need to show for all words $w \in \Sigma^+$:

$$w \in L(G) \iff w \in T(M).$$

Claim: For all $w \in \Sigma^*$ and all $A, B \in V$, we have:

$$A \Rightarrow_G^* wB \iff B \in \widehat{\delta}(\{A\}, w)$$

We prove this claim by induction on $|w|$.

Base case: $w = \varepsilon$. We have:

$$A \Rightarrow_G^* B \iff A = B \iff B \in \{A\} = \widehat{\delta}(\{A\}, \varepsilon)$$

Inductive step: Let $w = av$ ($a \in \Sigma$, $v \in \Sigma^*$), and assume the claim holds for the word v.

$$\begin{aligned}
A \Rightarrow_G^* avB \quad &\iff \quad \exists C \in V : (A \to aC) \in P \text{ and } C \Rightarrow_G^* vB \\
&\iff \quad \exists C \in V : C \in \delta(A, a) \text{ and } B \in \widehat{\delta}(\{C\}, v) \\
&\iff \quad \exists C \in V \cup \{X\} : C \in \delta(A, a) \text{ and } B \in \widehat{\delta}(\{C\}, v) \\
&\iff \quad B \in \widehat{\delta}(\{A\}, av)
\end{aligned}$$

This proves the claim.

Now let $w \in \Sigma^+$, for example $w = va$ with $a \in \Sigma$. Then we have:

$$\begin{aligned}
va \in L(G) \quad &\iff \quad \exists A \in V : S \Rightarrow_G^* vA \text{ and } (A \to a) \in P \\
&\overset{\text{Claim}}{\iff} \quad \exists A \in V : A \in \widehat{\delta}(\{S\}, v) \text{ and } X \in \delta(A, a) \\
&\iff \quad \exists A \in V \cup \{X\} : A \in \widehat{\delta}(\{S\}, v) \text{ and } X \in \delta(A, a) \\
&\iff \quad X \in \widehat{\delta}(\{S\}, va) \\
&\iff \quad va \in T(M)
\end{aligned}$$

Note for the last equivalence: Either

- X is the only accepting state of M or
- S is the second accepting state.

  Then we have $(S \to \varepsilon) \in P$.

  Due to the ε-special rule, S will not appear on the right-hand side of any production from P.

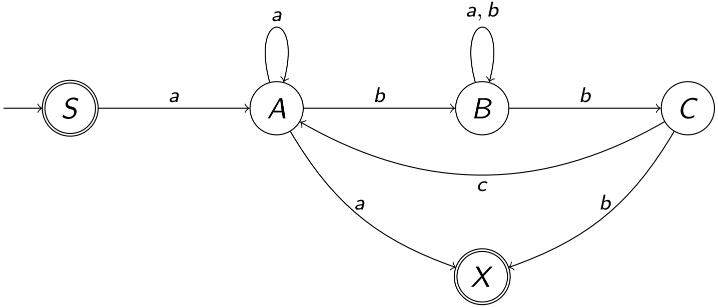
  Therefore, we have $S \notin \delta(A, a)$ for all $A \in V \cup \{X\}$, $a \in \Sigma$.

  This implies $S \notin \widehat{\delta}(\{S\}, va)$. $\square$

Example: Let G be the regular grammar with the following productions (we use Backus-Naur form, see slide 41):

$$S \to \varepsilon \mid aA \qquad A \to aA \mid bB \mid a$$
$$B \to aB \mid bB \mid bC \qquad C \to cA \mid b$$

The construction from slide 85 gives the following NFA:



NFAs, DFAs and Regular Grammars

Summary

We have learned about different models for describing regular languages:

- Regular Grammars: These connect to the Chomsky hierarchy. They are used for generating languages. They are less suited for deciding if a particular word belongs to the language.
- NFAs: These often allow for compact representations of languages. Due to their non-determinism, they are less suitable for solving the word problem compared to grammars. However, they possess an intuitive graphical notation.
- DFAs: These can be exponentially larger than equivalent NFAs. However, once a DFA is available, it allows for an efficient solution to the word problem (simply follow the transitions of the automaton and check if an accepting state is reached).

Regular Expressions

All models, however, require relatively much writing effort and space for notation. Therefore, we are looking for a more compact representation. This is where regular expressions come in.

Definition (Regular Expressions)

The set $\mathrm{Reg}(\Sigma)$ of regular expressions over the alphabet Σ is the smallest set with the following properties:

- $\emptyset \in \mathrm{Reg}(\Sigma)$, $\varepsilon \in \mathrm{Reg}(\Sigma)$, $\Sigma \subseteq \mathrm{Reg}(\Sigma)$.
- If $\alpha, \beta \in \mathrm{Reg}(\Sigma)$, then also $\alpha\beta, (\alpha|\beta), (\alpha)^* \in \mathrm{Reg}(\Sigma)$.

Remarks:

- Instead of $(\alpha|\beta)$, $(\alpha + \beta)$ is often used.
- We often omit unnecessary parentheses.
  For example, $(a|b)^*$ instead of $((a|b))^*$.

To save parentheses, we use so-called operator precedence rules:

- * binds more strongly than concatenation.
- Concatenation binds more strongly than $|$.

Example: $ab^*|c$ is read as $(a(b)^*|c)$.

These are the same operator precedence rules known from arithmetic operations like $+$, $\cdot$, and exponentiation.

$xy^n + z$ is read as $((x \cdot (y)^n) + z)$.

Regular Expressions

After defining the syntax of regular expressions, we must also define their meaning (semantics).

The semantics of a regular expression is a language:

Definition (Language of a regular expression)

- $L(\emptyset) = \emptyset$ (empty language), $L(\varepsilon) = \{\varepsilon\}$, $L(a) = \{a\}$ for $a \in \Sigma$.
- $L(\alpha\beta) = L(\alpha)L(\beta)$, where $L_1 L_2 = \{w_1 w_2 \mid w_1 \in L_1, w_2 \in L_2\}$ for two languages L_1, L_2 (concatenation of L_1 and L_2).
- $L(\alpha|\beta) = L(\alpha) \cup L(\beta)$
- $L((\alpha)^*) = (L(\alpha))^*$, where $L^* = \{w_1 \cdots w_n \mid n \geq 0, w_1, \ldots, w_n \in L\}$ for a language L

Example for concatenation of languages:

$$\{a, b, ab\}\{c, ba\} = \{ac, bc, abc, aba, bba, abba\}.$$

Remarks on the $*$-operator: $L^* = \{w_1 \cdots w_n \mid n \in \mathbb{N}, w_i \in L\}$

- For $n = 0$, we have $w_1 \cdots w_n = \varepsilon$.
- L^* always contains the empty word ε.
  Special case: $\emptyset^* = \{\varepsilon\}$.
- The $*$ operator is often called the Kleene star. It is the only operator capable of generating infinite languages.
  More precisely: L^* is infinite if and only if $L \cap \Sigma^+ \neq \emptyset$.
- Example for the application of the $*$-operator:

  Let $L = \{a, bb, cc\}$. Then

  $$L^* = \{\varepsilon, a, bb, cc, aa, abb, acc, bba, bbbb, bbcc, cca, ccbb, cccc, \dots\}$$

  All combinations of any length are possible.

Further Remarks:

- Note: regular expressions are purely syntactical expressions.
  Only through the definition on Slide 93 is a language assigned to a regular expression.

- The distinction between syntax and semantics can be found in many areas of computer science (programming languages, logic, etc.)

- In programming languages, we first define what syntactically correct programs are. After that, the semantics of a program are defined (what the program does).
  This may be, for example, the function computed by a program.
  Later, we will do the same for very simple programming languages (GOTO-programs, while-programs).

Regular Expressions

- Formally, one should also distinguish between the regular expression $\emptyset$ and the regular language $\emptyset$ (empty language), but we do not want to overdo it.
- The languages $\emptyset$ and $\{\varepsilon\}$ are often confused.
  $\emptyset$ is the empty language (has zero elements).
  $\{\varepsilon\}$ is a language that contains exactly one word (the empty word).

Examples of regular expressions over the alphabet $\Sigma = \{a, b\}$.

Example 1: Language of all words that begin with a and end with bb

$$\alpha = a(a|b)^* bb$$

Example 2: Language of all words that contain the substring aba.

$$\alpha = (a|b)^* aba(a|b)^*$$

Example 3: Language of all words that contain an even number of a's.

$$\alpha = (b^* ab^* a)^* b^* \qquad \text{or} \qquad \alpha = (b|ab^* a)^*$$

Theorem (Regular Expressions $\to$ NFAs)

For every regular expression γ, there is an NFA M such that $L(\gamma) = T(M)$.

Proof: Induction on the structure of γ.

Base Case: For $\gamma = \emptyset$, $\gamma = \varepsilon$, $\gamma = a$ ($a \in \Sigma$), corresponding NFAs clearly exist.

Inductive Step: Suppose $\gamma = \alpha\beta$. Then there are NFAs

$$\begin{aligned} M_\alpha &= (Z_\alpha, \Sigma, \delta_\alpha, S_\alpha, E_\alpha) \\ M_\beta &= (Z_\beta, \Sigma, \delta_\beta, S_\beta, E_\beta) \end{aligned}$$

with $T(M_\alpha) = L(\alpha)$ and $T(M_\beta) = L(\beta)$.

We can assume that $Z_\alpha \cap Z_\beta = \emptyset$.

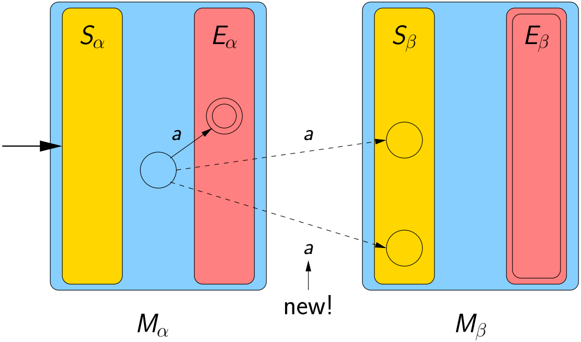
We now combine M_α and M_β sequentially to form an NFA M:

- M has the union of both state sets, the same start states as M_α and the same end states as M_β. If $\varepsilon \in L(\alpha)$, then the start states of M_β are also start states of M.
- All transitions from M_α and M_β are preserved. Any states that have an arrow to an end state of M_α also receive similarly labeled arrows to all start states of M_β.

Formally: $M = (Z_\alpha \cup Z_\beta, \Sigma, \delta, S, E_\beta)$, where

$$S = \begin{cases} S_\alpha & \text{if } \varepsilon \notin L(\alpha) \\ S_\alpha \cup S_\beta & \text{if } \varepsilon \in L(\alpha) \end{cases}$$

$$\delta(z,a) = \begin{cases} \delta_\beta(z,a) & \text{for } z \in Z_\beta \\ \delta_\alpha(z,a) & \text{for } z \in Z_\alpha \text{ with } \delta_\alpha(z,a) \cap E_\alpha = \emptyset \\ \delta_\alpha(z,a) \cup S_\beta & \text{for } z \in Z_\alpha \text{ with } \delta_\alpha(z,a) \cap E_\alpha \neq \emptyset \end{cases}$$



We have $T(M) = T(M_\alpha)T(M_\beta) = L(\alpha)L(\beta) = L(\alpha\beta) = L(\gamma)$

Let $\gamma = (\alpha \mid \beta)$. Then there exist NFAs

$$\begin{aligned} M_\alpha &= (Z_\alpha, \Sigma, \delta_\alpha, S_\alpha, E_\alpha) \\ M_\beta &= (Z_\beta, \Sigma, \delta_\beta, S_\beta, E_\beta) \end{aligned}$$

with $T(M_\alpha) = L(\alpha)$ and $T(M_\beta) = L(\beta)$.

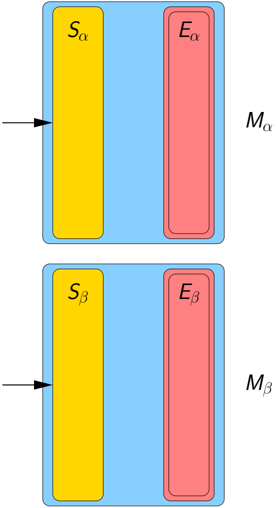
We can assume that $Z_\alpha \cap Z_\beta = \emptyset$.

We now construct a union NFA M from these two NFAs:

- M has as states the union of both state sets. Similarly, the start states are the union of the start state sets, and the end states are the union of the end state sets.
- All transitions from M_α and M_β are preserved.

Formally: $M = (Z_\alpha \cup Z_\beta, \Sigma, \delta, S_\alpha \cup S_\beta, E_\alpha \cup E_\beta)$, where

$$\delta(z, a) = \begin{cases} \delta_\alpha(z, a) & \text{for } z \in Z_\alpha \\ \delta_\beta(z, a) & \text{for } z \in Z_\beta \end{cases}$$



It holds $$\begin{aligned} T(M) &= T(M_\alpha) \cup T(M_\beta) \\ &= L(\alpha) \cup L(\beta) \\ &= L(\alpha \mid \beta) \\ &= L(\gamma) \end{aligned}$$

Let $\gamma = (\alpha)^*$. Then there is an NFA

$$M_\alpha = (Z_\alpha, \Sigma, \delta_\alpha, S_\alpha, E_\alpha)$$

with $T(M_\alpha) = L(\alpha)$.

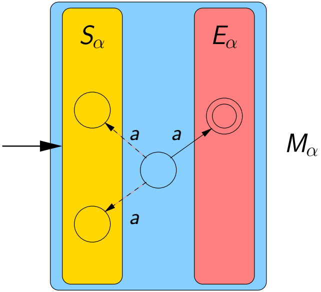
We now construct an NFA M from this NFA as follows:

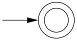
- If $\varepsilon \notin T(M_\alpha)$, then an additional state is added, which is both a start and an end state (so that the empty word is also recognized).
- The other states, start and end states, and transitions are preserved.
- All states that have a transition to an end state of M_α also receive transitions to all start states of M_α (feedback loop).

Formal: $M = (Z, \Sigma, \delta, S, E)$, where:

$$
\begin{aligned}
Z &= \begin{cases} Z_\alpha & \text{if } \varepsilon \in L(\alpha) \\ Z_\alpha \cup \{s_0\} & \text{if } \varepsilon \notin L(\alpha) \end{cases} \\
S &= \begin{cases} S_\alpha & \text{if } \varepsilon \in L(\alpha) \\ S_\alpha \cup \{s_0\} & \text{if } \varepsilon \notin L(\alpha) \end{cases} \\
E &= \begin{cases} E_\alpha & \text{if } \varepsilon \in L(\alpha) \\ E_\alpha \cup \{s_0\} & \text{if } \varepsilon \notin L(\alpha) \end{cases} \\
\delta(z, a) &= \begin{cases} \delta_\alpha(z, a) & \text{for } z \in Z_\alpha \text{ with } \delta_\alpha(z, a) \cap E_\alpha = \emptyset \\ \delta_\alpha(z, a) \cup S_\alpha & \text{for } z \in Z_\alpha \text{ with } \delta_\alpha(z, a) \cap E_\alpha \neq \emptyset \end{cases}
\end{aligned}
$$

Here, $s_0 \notin Z_\alpha$.





possibly additional state

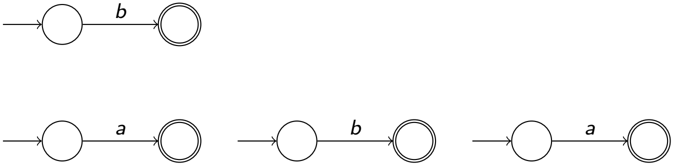
It holds $T(M) = (T(M_\alpha))^* = (L(\alpha))^* = L(\alpha^*) = L(\gamma)$.

Regular Expressions

Example: We will construct step by step an NFA for the regular expression $(b \mid ab^* a)^*$.

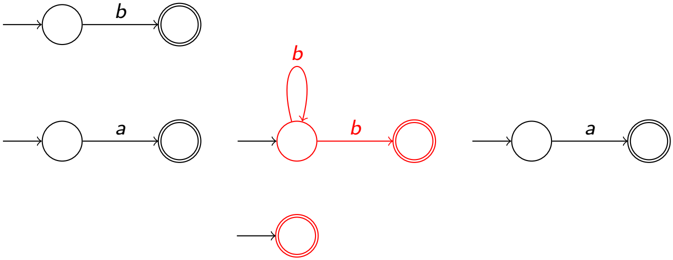
Example: We will construct step by step an NFA for the regular expression $(b \mid ab^* a)^*$.

We begin with the transitions for individual symbols.



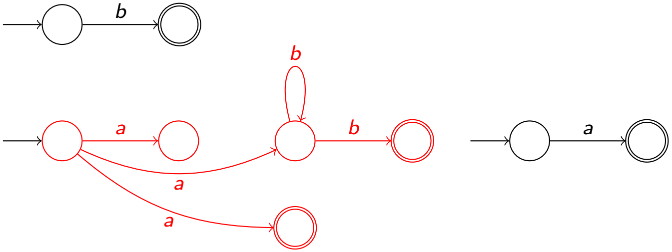
Example: We will construct step by step an NFA for the regular expression $(b \mid ab^* a)^*$.

NFA for b^*



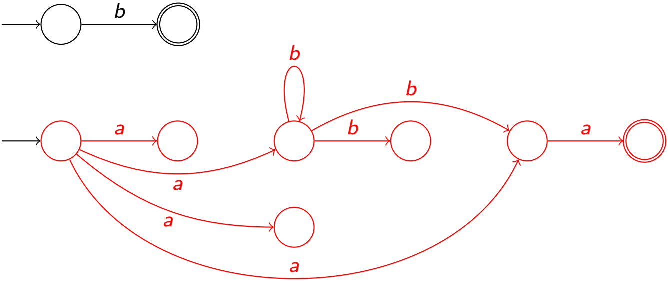
Example: We will construct step by step an NFA for the regular expression $(b \mid ab^*a)^*$.

NFA for ab^*



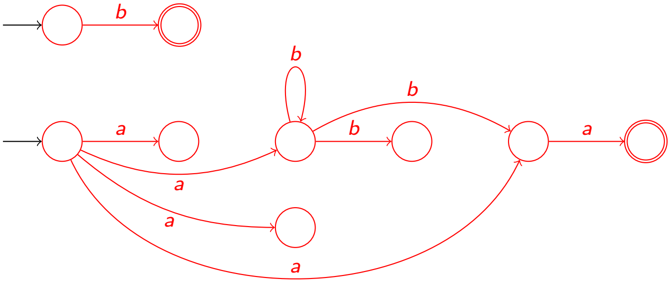
Example: We will construct step by step an NFA for the regular expression $(b \mid ab^*a)^*$.

NFA for ab^*a



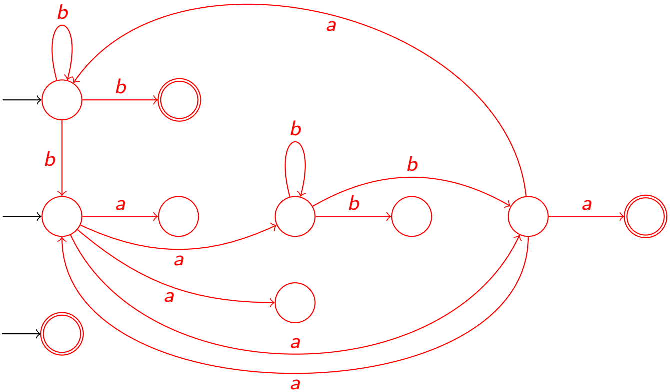
Example: We will construct step by step an NFA for the regular expression $(b \mid ab^*a)^*$.

NFA for $(b \mid ab^*a)$



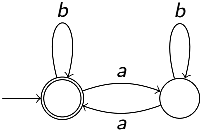
Example: We will construct step by step an NFA for the regular expression $(b \mid ab^*a)^*$.

NFA for $(b \mid ab^*a)^*$



Example (continued): This NFA contains many redundant states and can be simplified.

A much simpler NFA for $(b \mid ab^*a)^*$ is:



Theorem (DFAs $\to$ Regular Expressions)

For every DFA M, there is a regular expression γ such that $T(M) = L(\gamma)$.

Proof: Let $M = (\{z_1, \dots, z_n\}, \Sigma, \delta, z_1, E)$ be a DFA.

We construct a regular expression γ with $T(M) = L(\gamma)$.

For a word $w \in \Sigma^*$, define

$$\mathrm{Pref}(w) = \{u \in \Sigma^* \mid \exists v : w = uv, \varepsilon \neq u \neq w\}$$

as the set of all non-empty proper prefixes of w.

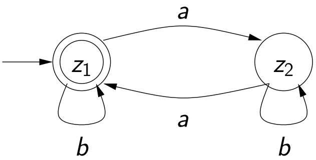
Example: $\mathrm{Pref}(abbca) = \{a, ab, abb, abbc\}$

For $i, j \in \{1, \dots, n\}$ and $k \in \{0, \dots, n\}$, define

$$L_{i,j}^k = \{w \in \Sigma^* \mid \widehat{\delta}(z_i, w) = z_j, \forall u \in \mathrm{Pref}(w) : \widehat{\delta}(z_i, u) \in \{z_1, \dots, z_k\}\}.$$

Intuition: A word w belongs to $L^k_{i,j}$ if and only if w transitions from state z_i to state z_j, and during this transition, no intermediate state (other than the start and end states) is from $\{z_{k+1}, \ldots, z_n\}$.

Example: Consider the following DFA M:



For example, we have:

$$L^0_{1,1} = \{\varepsilon, b\}, \quad L^0_{1,2} = \{a\}, \quad L^1_{2,2} = \{ab^n a \mid n \geq 0\} \cup \{\varepsilon, b\}$$

and $L^2_{1,1} = T(M) = \{w \in \{a, b\}^* \mid w \text{ contains an even number of } a\text{'s}\}$.

We construct regular expressions $\gamma^k_{i,j}$ for all $i, j \in \{1, \dots, n\}$ and $k \in \{0, \dots, n\}$ with $L(\gamma^k_{i,j}) = L^k_{i,j}$.

If $E = \{z_{i_1}, z_{i_2}, \dots, z_{i_m}\}$, then we have:

$$L(\gamma^n_{1,i_1} \mid \gamma^n_{1,i_2} \mid \cdots \mid \gamma^n_{1,i_m}) = T(M).$$

Construction of $\gamma^k_{i,j}$ by induction over $k \in \{0, \dots, n\}$.

Base case: $k = 0$. We have:

$$L^0_{i,j} = \begin{cases} \{\varepsilon\} \cup \{a \in \Sigma \mid \delta(z_i, a) = z_j\} & \text{if } i = j \\ \{a \in \Sigma \mid \delta(z_i, a) = z_j\} & \text{if } i \neq j \end{cases}$$

A regular expression $\gamma^0_{i,j}$ with $L(\gamma^0_{i,j}) = L^0_{i,j}$ can be easily provided.

Inductive step: Let $0 \leq k < n$ and assume the regular expressions $\gamma^k_{p,q}$ have already been constructed for all $p, q \in \{1, \dots, n\}$.

Claim: For all $i, j \in \{1, \ldots, n\}$, the following holds:

$$L_{i,j}^{k+1} = L_{i,j}^{k} \cup L_{i,k+1}^{k}(L_{k+1,k+1}^{k})^* L_{k+1,j}^{k}. \tag{2}$$

Justification:

$\subseteq$: Let $w \in L_{i,j}^{k+1}$ and suppose $\ell \geq 0$ such that the state z_{k+1} appears exactly ℓ times as a genuine intermediate state on the unique path from z_i to z_j labeled by w.

Case 1: $\ell = 0$, i.e., z_{k+1} does not appear as a genuine intermediate state.

Then $w \in L_{i,j}^{k}$, so we have $w \in L_{i,j}^{k} \cup L_{i,k+1}^{k}(L_{k+1,k+1}^{k})^* L_{k+1,j}^{k}$.

Case 2: $\ell > 0$.

Then w can be written as $w = w_0 w_1 \cdots w_{\ell-1} w_\ell$, where:

$$\begin{aligned}
\widehat{\delta}(z_i, w_0) &= z_{k+1} \\
\widehat{\delta}(z_{k+1}, w_p) &= z_{k+1} \text{ for } 1 \leq p \leq \ell - 1 \\
\widehat{\delta}(z_{k+1}, w_\ell) &= z_j
\end{aligned}$$

Regular Expressions

It follows that $w_0 \in L^k_{i,k+1},\ w_1, \ldots, w_{\ell-1} \in L^k_{k+1,k+1},\ w_\ell \in L^k_{k+1,j}$, and thus

$$w = w_0(w_1 \cdots w_{\ell-1})w_\ell \in L^k_{i,k+1}(L^k_{k+1,k+1})^* L^k_{k+1,j}.$$

$\supseteq$: $L^k_{i,j} \subseteq L^{k+1}_{i,j}$ is obvious.

If $w \in L^k_{i,k+1}(L^k_{k+1,k+1})^* L^k_{k+1,j}$, there exists an $\ell \geq 1$ and a factorization $w = w_0 w_1 \cdots w_{\ell-1} w_\ell$ with

$$w_0 \in L^k_{i,k+1},\ w_1, \ldots, w_{\ell-1} \in L^k_{k+1,k+1},\ w_\ell \in L^k_{k+1,j}.$$

This easily shows that $w \in L^{k+1}_{i,j}$. Thus, the claim is proved.

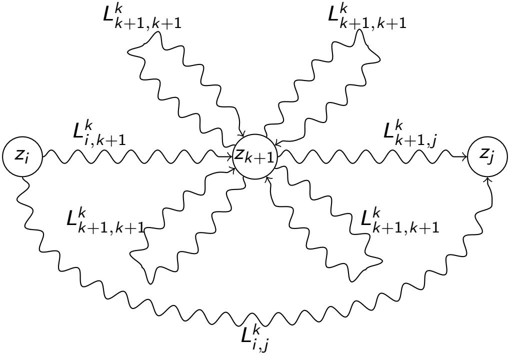
Since the regular expressions $\gamma^k_{i,j}, \gamma^k_{i,k+1}, \gamma^k_{k+1,k+1}, \gamma^k_{k+1,j}$ have already been constructed (inductive hypothesis), we can define the regular expression $\gamma^{k+1}_{i,j}$ as follows:

$$\gamma^{k+1}_{i,j} = \gamma^k_{i,j} \mid \gamma^k_{i,k+1}(\gamma^k_{k+1,k+1})^* \gamma^k_{k+1,j}$$

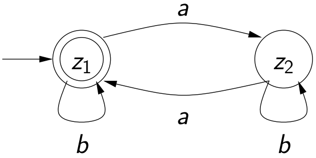
$\square$

Regular Expressions

$$L_{i,j}^{k+1} = L_{i,j}^{k} \cup L_{i,k+1}^{k}(L_{k+1,k+1}^{k})^{*} L_{k+1,j}^{k}$$



Example: Consider the following DFA:



This results in (after performing obvious simplifications):

$$\gamma^0_{1,1} = \varepsilon|b \qquad \gamma^0_{1,2} = a \qquad \gamma^0_{2,1} = a \qquad \gamma^0_{2,2} = \varepsilon|b$$

$$\gamma^1_{1,1} = \gamma^0_{1,1} \,|\, \gamma^0_{1,1}(\gamma^0_{1,1})^*\gamma^0_{1,1} = \varepsilon \,|\, b \,|\, (\varepsilon|b)(\varepsilon|b)^*(\varepsilon|b) = b^*$$

$$\gamma^1_{1,2} = \gamma^0_{1,2} \,|\, \gamma^0_{1,1}(\gamma^0_{1,1})^*\gamma^0_{1,2} = a \,|\, (\varepsilon|b)(\varepsilon|b)^* a = b^* a$$

$$\gamma^1_{2,1} = \gamma^0_{2,1} \,|\, \gamma^0_{2,1}(\gamma^0_{1,1})^*\gamma^0_{1,1} = a \,|\, a(\varepsilon|b)^*(\varepsilon|b) = ab^*$$

$$\gamma^1_{2,2} = \gamma^0_{2,2} \,|\, \gamma^0_{2,1}(\gamma^0_{1,1})^*\gamma^0_{1,2} = \varepsilon \,|\, b \,|\, a(\varepsilon|b)^* a = \varepsilon \,|\, b \,|\, ab^* a$$

$$\gamma^2_{1,1} = \gamma^1_{1,1} \,|\, \gamma^1_{1,2}(\gamma^1_{2,2})^*\gamma^1_{2,1} = b^* \,|\, b^* a(\varepsilon|b|ab^* a)^* ab^*$$

Regular Expressions

What are regular expressions useful for in practice?

- Search and replace in editors (Try with `vi`, `emacs`, …)
- Pattern matching and processing large texts and data sets, e.g., in data mining (Tools: Stream editor `sed`, `awk`, …)
- Translation of programming languages: Lexical analysis – converting a sequence of characters (the program) into a sequence of tokens, where keywords, identifiers, data, etc., are already identified. (Tools: `lex`, `flex`, …), see the lecture on *Compiler Construction* (where a more efficient version of the conversion from regular expressions to NFAs is also discussed).

Closure Properties

Definition (Closure)

Let M be a set and $\otimes\colon M \times M \to M$ be a binary operator.
A set $M' \subseteq M$ is said to be closed under $\otimes$ if for any two elements $m_1, m_2 \in M'$, we have: $m_1 \otimes m_2 \in M'$.

We consider closure properties for the set of regular languages (i.e., we set M as the set of all languages and M' as the set of all regular languages).

The interesting question is:

> If L_1, L_2 are regular, are $L_1 \cup L_2$, $L_1 \cap L_2$, $L_1 L_2$, $\overline{L_1} = \Sigma^* \setminus L_1$ (complement), and L_1^* also regular?

Short answer: The regular languages are closed under all these operations.

Closure Properties

Why are closure properties interesting?

They are particularly interesting when they can be constructed, that is, when one can – given automata for L_1 and L_2 – also construct an automaton for, say, the intersection of L_1 and L_2.

This way, one can have an automaton as a data structure for infinite languages, which can be further processed by a machine.

Theorem (Closure under Union)

If L_1 and L_2 are regular languages, then $L_1 \cup L_2$ is also regular.

Proof:

The automaton for $L_1 \cup L_2$ can be constructed using the same method as the automaton for $L(\alpha|\beta)$ when converting regular expressions to NFAs (see slide 101). □

Closure Properties

Theorem (Closure under Complementation)

If $L \subseteq \Sigma^*$ is a regular language, then $\overline{L} = \Sigma^* \setminus L$ is also regular.

Remark: When taking the complement, it must always be specified with respect to which superset the complement is formed. Here, the superset is Σ^*, the set of all words over the alphabet Σ being considered.

Proof:

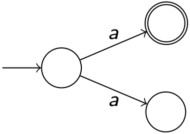
From a DFA $M = (Z, \Sigma, \delta, z_0, E)$ for L, we can easily obtain a DFA M' for $\overline{L}$ by swapping the accepting and non-accepting states. That is,
$M' = (Z, \Sigma, \delta, z_0, Z \setminus E)$.

Then it holds that:
$w \in \overline{L} \iff w \notin T(M) \iff \widehat{\delta}(z_0, w) \notin E \iff \widehat{\delta}(z_0, w) \in Z \setminus E \iff w \in T(M')$. $\square$

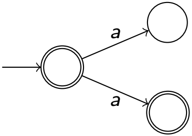
Caution: In the proof on the previous slide, it is important that M is a DFA.

If we swap the accepting and non-accepting states in an NFA, we generally do not obtain an NFA for the complement.

Example: Consider the following NFA for the language $\{a\} \subseteq \{a\}^*$.



By swapping the accepting and non-accepting states, we obtain an NFA for $\{\varepsilon, a\} \neq \{a\}^* \setminus \{a\}$:



If you want to complement an NFA M (i.e., construct an NFA for $\Sigma^* \setminus T(M)$), the essentially best method is as follows:

1. Construct a DFA M' using the powerset construction such that $T(M') = T(M)$.
2. Swapping the accepting and non-accepting states in M' gives a DFA (and thus also an NFA) M'' with $T(M'') = \Sigma^* \setminus T(M') = \Sigma^* \setminus T(M)$.

Theorem (Closure under Product/Concatenation)

If L_1 and L_2 are regular languages, then $L_1 L_2$ is also regular.

Proof:

The automaton for $L_1 L_2$ can be constructed in the same way as the automaton for $L(\alpha\beta)$ when converting regular expressions to NFAs (see slide 100). $\square$

Theorem (Closure under the Star Operation)

If L is a regular language, then L^* is also regular.

Proof:

The automaton for L^* can be constructed in the same way as the automaton for $L((\alpha)^*)$ when converting regular expressions to NFAs (see slide 105). $\square$

Closure Properties

Theorem (Closure under Intersection)

If L_1 and L_2 are regular languages, then $L_1 \cap L_2$ is also regular.

Proof 1:

We have $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$, and we already know that regular languages are closed under complement and union. $\square$

In the above proof, complementing leads to a very large automaton for $L_1 \cap L_2$.

Proof 2:

There is another more direct construction. This involves synchronizing the two automata for L_1 and L_2 and essentially running them "in parallel." This is achieved by forming the cross product.

Let $M_1 = (Z_1, \Sigma, \delta_1, S_1, E_1)$ and $M_2 = (Z_2, \Sigma, \delta_2, S_2, E_2)$ be NFAs with $T(M_1) = L_1$ and $T(M_2) = L_2$. Then the following NFA M accepts the language $L_1 \cap L_2$:

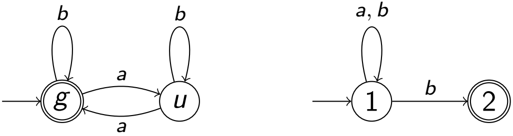
$$M = (Z_1 \times Z_2, \Sigma, \delta, S_1 \times S_2, E_1 \times E_2),$$

where $\delta((z_1, z_2), a) = \{(z_1', z_2') \mid z_1' \in \delta_1(z_1, a), z_2' \in \delta_2(z_2, a)\}$.

M accepts a word w if and only if both M_1 and M_2 accept the word w. $\square$

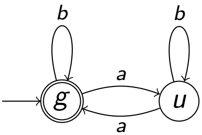
Example of a Cross Product:

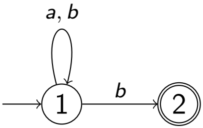
Form the cross product of the following two automata:

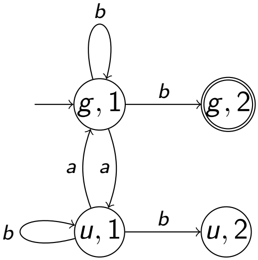


Example of a Cross Product:

Form the cross product of the following two automata:







Outlook

Further Important Questions

- How can one prove that a language is not regular?

  Example: The language $\{a^n b^n c^n \mid n \geq 1\}$, which appeared as an example, seems not to be regular. How can this be demonstrated?

- If a language is regular, how large is the smallest automaton that accepts the language?

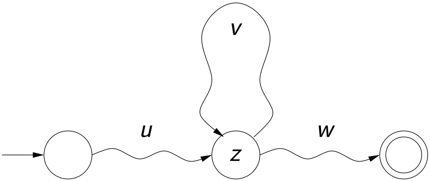
  Does the smallest automaton even exist?

The Pumping Lemma

How can one prove that a language L is not regular?

Idea: The goal is to exploit the fact that a regular language must be accepted by an automaton with a finite number of states.

This also implies: if a word $x \in L$ is sufficiently long, then at least one state z is visited more than once during the traversal of the automaton.



The resulting loop can then be traversed multiple times (or not at all), thus "pumping" the word $x = uvw$. It follows that uw, uv^2w, uv^3w, ... must also belong to L.

Remark: It holds that $v^i = \underbrace{v \ldots v}_{i \text{ times}}$.

The Pumping Lemma

Additionally, for u, v, w, the following properties can be required, where n is the number of states of the automaton.

1. $|v| \geq 1$: The loop is non-trivial, i.e., it contains at least one transition.
2. $|uv| \leq n =$ number of states of the NFA: After at most n alphabet symbols, the state z is reached for the second time.

The Pumping Lemma

Theorem (Pumping Lemma, *uvw*-Theorem)

Let L be a regular language. Then there exists a number n such that all words $x \in L$ with $|x| \geq n$ can be decomposed as $x = uvw$, satisfying the following properties:

1. $|v| \geq 1$,
2. $|uv| \leq n$, and
3. for all $i \geq 0$, $uv^iw \in L$ holds.

Here, n is the number of states of an automaton that recognizes L.

This lemma, however, does not speak about automata but only about the properties of the language. Hence, it is suitable for making statements about non-regularity.

Proof of the Pumping Lemma:

Let L be a regular language.

Let $M = (Z, \Sigma, \delta, S, E)$ be an NFA with $L = T(M)$, and let $n = |Z|$.

Now let x be an arbitrary word with $x \in L = T(M)$ and $|x| \geq n$, i.e., $x = a_1 a_2 \cdots a_m$ with $m \geq n$ and $a_1, a_2, \ldots, a_m \in \Sigma$.

Since $x \in T(M)$, there exist states $z_0, z_1, \ldots, z_m \in Z$ such that

$$z_0 \in S, \quad z_j \in \delta(z_{j-1}, a_j) \text{ for } 1 \leq j \leq m, \quad z_m \in E.$$

Because $|Z| = n$, there exist $0 \leq j < k \leq n$ with $z_j = z_k$ (pigeonhole principle).

Let $u = a_1 \cdots a_j$, $v = a_{j+1} \cdots a_k$, and $w = a_{k+1} \cdots a_m$.

Then the following holds:

- $|v| = k - (j+1) + 1 = k - j > 0$ and $|uv| = k \leq n$
- for all $i \geq 0$: $z_m \in \widehat{\delta}(\{z_0\}, uv^i w)$ and thus $uv^i w \in T(M) = L$,

□

The Pumping Lemma

How can the Pumping Lemma be used to show that L is not regular?

Statement of the Pumping Lemma using logical operators:

L is regular

$\rightarrow$

$\exists n : \forall x \in L$ with $|x| \geq n$:

$\exists u, v, w$ such that $|v| \geq 1, |uv| \leq n, x = uvw$ and $\forall i : uv^i w \in L$

This is logically equivalent to:

$\forall n : \exists x \in L$ with $|x| \geq n$:

$\forall u, v, w$ such that $|v| \geq 1, |uv| \leq n$ and $x = uvw$:

$\exists i : uv^i w \notin L$

$\rightarrow$ L is not regular

Note for this: $A \rightarrow B \equiv \neg B \rightarrow \neg A$ and $\neg \forall x \exists y F \equiv \exists x \forall y \neg F$

"Recipe" for Using the Pumping Lemma

Given a language L.

Example: $\{a^k b^k \mid k \geq 0\}$

We want to show that it is not regular.

1. Take an arbitrary number n. This number must not be chosen specifically (it has to be arbitrary).
2. Choose a suitable word $x \in L$ with $|x| \geq n$. To ensure the word actually has at least length n, it is advisable to include n (for instance, as an exponent) in the word.

   Example: $x = a^n b^n$

"Recipe" for Using the Pumping Lemma

3. Now consider all possible decompositions $x = uvw$ with the restrictions $|v| \geq 1$ and $|uv| \leq n$.

   Example: From $uvw = a^n b^n$, $|v| \geq 1$, and $|uv| \leq n$, it follows that $j \geq 0$ and $\ell \geq 1$ exist with:
   $u = a^j$, $v = a^\ell$, and $w = a^m b^n$ with $j + \ell + m = n$

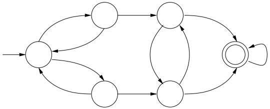
4. Choose for each of these decompositions a value of i (this can differ for each case) such that $uv^i w \notin L$.
   In many cases, $i = 0$ and $i = 2$ are good choices.

   Example: Choose $i = 2$, then $uv^2 w = a^{j+2\ell+m} b^n \notin L$, since $j + 2\ell + m = n + \ell \neq n$ because $\ell \geq 1$.

We now address the following questions:

- Does there always exist the smallest deterministic/non-deterministic automaton for every language?
- Can the number of states of the minimal automaton be directly inferred from the language?
- How can the minimal automaton be determined?

Consider the following DFA M:



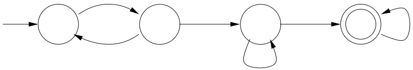
Observation: For states 4 and 5, it holds that:

- With a word containing an a, one always reaches state 6 (final state) from either state.
- With a word containing no a, one always reaches state 4 or 5 (non-final states) respectively.

From this, it follows that states 4 and 5 are recognition-equivalent and can be merged into a single state.

Similarly, states 2 and 3 are recognition-equivalent.

Resulting Automaton M':



Now, no states are recognition-equivalent anymore and therefore cannot be merged further.

$\rightsquigarrow$ The automaton M' is minimal for this language.

Equivalence Relations and Minimal Automaton

Definition (Recognition-Equivalence)

Let $M = (Z, \Sigma, \delta, q_0, E)$ be a DFA.

Two states $z_1, z_2 \in Z$ are called recognition-equivalent if and only if for every word $w \in \Sigma^*$, the following holds:

$$\widehat{\delta}(z_1, w) \in E \iff \widehat{\delta}(z_2, w) \in E.$$

The relation $\{(z_1, z_2) \in Z \times Z \mid z_1 \text{ and } z_2 \text{ are recognition-equivalent}\}$ is an equivalence relation on the state set Z.

Insert: Equivalence Relation

Equivalence relations are discussed in the module *Discrete Mathematics for Computer Scientists*.

A binary relation $R \subseteq A \times A$ is an equivalence relation if the following hold:

- R is reflexive: for all $a \in A$, $(a, a) \in R$.
- R is symmetric: for all $a, b \in A$, if $(a, b) \in R$, then also $(b, a) \in R$.
- R is transitive: for all $a, b, c \in A$, if $(a, b) \in R$ and $(b, c) \in R$, then also $(a, c) \in R$.

Often, $a\,R\,b$ is written instead of $(a, b) \in R$ (infix notation).

Insert: Equivalence Relation

For $x \in A$, $[x] = \{y \in A \mid x \, R \, y\}$ is the equivalence class of x.

Sometimes, $[x]_R$ is written to clarify that it refers to the equivalence class with respect to the equivalence relation R.

However, when it is clear which equivalence relation R is meant, we simply write $[x]$.

Note:

- It always holds that $x \in [x]$.
- $x \, R \, y$ if and only if $[x] = [y]$.

The equivalence classes of R form a partition of A, meaning every element of A belongs to exactly one equivalence class.

Equivalence Relations and Minimal Automaton

Each word $x \in \Sigma^*$ can be assigned a unique state $z = \widehat{\delta}(z_0, x)$ in a DFA. Therefore, the definition of recognition equivalence can be extended to words from Σ^* and languages (instead of automata).

Definition (Myhill-Nerode Equivalence)

Given a language L and words $x, y \in \Sigma^*$, we define an equivalence relation R_L with $x \, R_L \, y$ if and only if

$$\forall w \in \Sigma^* \, (xw \in L \iff yw \in L).$$

For a regular language L, the following relationship holds between the Myhill-Nerode equivalence R_L and the concept of recognition equivalence:

Lemma 2

Let $M = (Z, \Sigma, \delta, z_0, E)$ be a DFA and $L = T(M) \subseteq \Sigma^*$. Then for all words $x, y \in \Sigma^*$, we have:

$$x \, R_L \, y \iff \text{the states } \widehat{\delta}(z_0, x) \text{ and } \widehat{\delta}(z_0, y) \text{ are recognition equivalent.}$$

Proof: It holds that

$$\begin{aligned}
x \, R_L \, y &\iff \forall w \in \Sigma^* (xw \in L \iff yw \in L) \\
&\iff \forall w \in \Sigma^* (xw \in T(M) \iff yw \in T(M)) \\
&\iff \forall w \in \Sigma^* (\widehat{\delta}(z_0, xw) \in E \iff \widehat{\delta}(z_0, yw) \in E) \\
&\iff \forall w \in \Sigma^* (\widehat{\delta}(\widehat{\delta}(z_0, x), w) \in E \iff \widehat{\delta}(\widehat{\delta}(z_0, y), w) \in E) \\
&\iff \text{the states } \widehat{\delta}(z_0, x) \text{ and } \widehat{\delta}(z_0, y) \text{ are recognition equivalent.}
\end{aligned}$$

$\square$

Remarks:

- The Myhill-Nerode equivalence R_L is defined for every language L, not just for regular languages.
- From $x \, R_L \, y$, it follows that: $x \in L \;\Leftrightarrow\; y \in L$.
  For each equivalence class $[x]$, it thus holds that: $[x] \subseteq L$ or $[x] \cap L = \emptyset$.

Common mistake: It is often thought that $x \, R_L \, y$ holds if and only if $\forall w \in \Sigma^* (xw \in L \text{ and } yw \in L)$.

But this is false!

The definition of R_L can also be written as follows:
$x \, R_L \, y$ holds if and only if for all words $w \in \Sigma^*$:

- $(xw \in L$ and $yw \in L)$ or
- $(xw \notin L$ and $yw \notin L)$ holds.

Equivalence Relations and Minimal Automaton

Example 1 for Myhill-Nerode equivalence: Given the language

$$L = \{w \in \{a,b\}^* \mid \#_a(w) \text{ is even}\}.$$

The following equivalence classes for R_L exist:

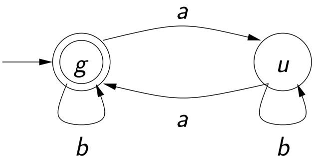
- $[\varepsilon] = \{w \in \{a,b\}^* \mid \#_a(w) \text{ is even}\} = L$
  (Equivalence class of ε)
- $[a] = \{w \in \{a,b\}^* \mid \#_a(w) \text{ is odd}\} = \{a,b\}^* \setminus L$
  (Equivalence class of a)

The words ε and aa are equivalent, because:

- If a word with an even number of a's is appended to both, they stay in the language.
- If a word with an odd number of a's is appended to both, they fall out of the language.

Equivalence Relations and Minimal Automaton

DFA for $\{w \in \{a, b\}^* \mid \#_a(w) \text{ is even}\}$:



Example 2 for Myhill-Nerode equivalence: Given the language

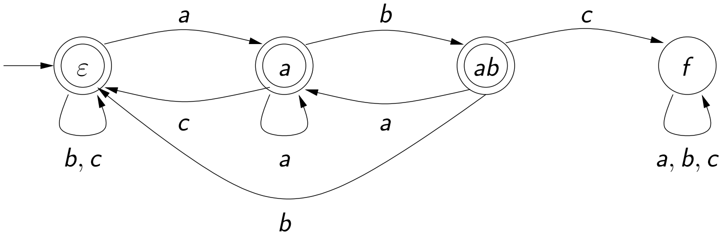
$$L = \{w \in \{a, b, c\}^* \mid \text{the substring } abc \text{ does not appear in } w\}.$$

The following equivalence classes for R_L exist:

- $[\varepsilon] = \{w \in \{a, b, c\}^* \mid w$ does not end with a or ab and does not contain $abc\}$
- $[a] = \{w \in \{a, b, c\}^* \mid w$ ends with a and does not contain $abc\}$
- $[ab] = \{w \in \{a, b, c\}^* \mid w$ ends with ab and does not contain $abc\}$
- $[abc] = \{w \in \{a, b, c\}^* \mid w$ contains $abc\}$ (Trap state)

The words a and ab are not equivalent, because if c is appended to both, ac is still in L, but abc is not.

DFA for $\{w \in \{a, b, c\}^* \mid \text{the substring } abc \text{ does not appear in } w\}$:



Insertion on Equivalence Relations

Insertion on Equivalence Relations: Let $R \subseteq A \times A$ be an equivalence relation on the set A.

The index $\text{index}(R)$ of R is the number of equivalence classes of R (which can be infinite):

$$\text{index}(R) = |\{[x] \mid x \in A\}| \in \mathbb{N} \cup \{\infty\}.$$

Example: On the set of integers $\mathbb{Z}$, for a natural number $k \geq 2$, the equivalence relation $\equiv_k$ is defined by $a \equiv_k b$ (a is congruent to b modulo k) if and only if there exists a $q \in \mathbb{Z}$ such that $a - b = q \cdot k$ (see the DMI lecture). Then, we have $\text{index}(\equiv_k) = k$.

Insertion on Equivalence Relations

Observation: Let R and S be equivalence relations on the same set A. If $R \subseteq S$ (i.e., if $a\,R\,b$ implies $a\,S\,b$), then it follows that $\mathrm{index}(S) \leq \mathrm{index}(R)$ (where $x \leq \infty$ for $x \in \mathbb{N} \cup \{\infty\}$).

Justification: Let $[A]_R$ ($[A]_S$) denote the set of equivalence classes of R (S).

We define a map $f : [A]_R \to [A]_S$ by the rule

$$f([a]_R) = [a]_S.$$

Caution: Is $f([a]_R)$ uniquely defined?

The value $f([a]_R)$ must not depend on which representative we choose for the equivalence class $[a]_R$.

Specifically: We need to show $[a]_R = [b]_R \implies [a]_S = [b]_S$:

$$[a]_R = [b]_R \iff a\,R\,b \implies a\,S\,b \iff [a]_S = [b]_S.$$

Naturally, f is also surjective: Every equivalence class $[a]_S$ is hit: $f([a]_R) = [a]_S$.

Now, for arbitrary sets X and Y: $|X| \geq |Y|$ if and only if there exists a surjective map $f : X \to Y$ (this is, in fact, the definition of $|X| \geq |Y|$).

So in our situation: $\text{index}(R) = |[A]_R| \geq |[A]_S| = \text{index}(S)$.

One of the most famous theorems in automata theory is the following characterization of regular languages:

Myhill-Nerode Theorem

Let L be a language. L is regular if and only if $\text{index}(R_L) < \infty$.

Proof:

$\Longrightarrow$: Let L be regular.

Let $M = (Z, \Sigma, \delta, z_0, E)$ be a DFA with $T(M) = L$.

Define an equivalence relation R_M on Σ^* as follows:

$$x \, R_M \, y \quad \Longleftrightarrow \quad \widehat{\delta}(z_0, x) = \widehat{\delta}(z_0, y).$$

Note:

- R_M is indeed an equivalence relation.
- $\mathrm{index}(R_M) \leq |Z|$.

  More precisely: $\mathrm{index}(R_M)$ is the number of states that can be reached from the initial state, i.e., $\mathrm{index}(R_M) = |\{\widehat{\delta}(z_0, x) \mid x \in \Sigma^*\}|$.

Equivalence Relations and Minimal Automaton

Claim: $\forall x, y \in \Sigma^* (x \, R_M \, y \Longrightarrow x \, R_L \, y)$, i.e., $R_M \subseteq R_L$.

Proof of the claim:

$$\begin{aligned} x \, R_M \, y \quad & \Longleftrightarrow \quad \widehat{\delta}(z_0, x) = \widehat{\delta}(z_0, y) \\ & \Longleftrightarrow \quad \forall w \in \Sigma^* : \widehat{\delta}(z_0, xw) = \widehat{\delta}(z_0, yw) \\ & \Longrightarrow \quad \forall w \in \Sigma^* : xw \in T(M) = L \Leftrightarrow yw \in T(M) = L \\ & \Longleftrightarrow \quad x \, R_L \, y \end{aligned}$$

The remark on slide 150 shows that $\text{index}(R_L) \leq \text{index}(R_M) \leq |Z| < \infty$.

Equivalence Relations and Minimal Automaton

$\Longleftarrow$: Let $\mathrm{index}(R_L) < \infty$.

Let $[x_1], \ldots, [x_n]$ be a listing of all equivalence classes of R_L.

Note:

- $\Sigma^* = [x_1] \cup \cdots \cup [x_n]$.
- If $[x] = [y]$, then $[xa] = [ya]$ for all $a \in \Sigma$:

$$\begin{aligned}
[x] = [y] \quad & \Longleftrightarrow \quad x \, R_L \, y \\
& \Longleftrightarrow \quad \forall w \in \Sigma^* \, (xw \in L \Leftrightarrow yw \in L) \\
& \Longrightarrow \quad \forall w \in \Sigma^+ \, (xw \in L \Leftrightarrow yw \in L) \\
& \Longleftrightarrow \quad \forall a \in \Sigma \; \forall w \in \Sigma^* (xaw \in L \Leftrightarrow yaw \in L) \\
& \Longleftrightarrow \quad \forall a \in \Sigma \; (xa \, R_L \, ya) \\
& \Longleftrightarrow \quad \forall a \in \Sigma \; [xa] = [ya]
\end{aligned}$$

We now define the DFA (the so-called equivalence class automaton for L)

$$M_L = (\{[x_1], \ldots, [x_n]\}, \Sigma, \delta_L, [\varepsilon], \{[w] \mid w \in L\}),$$

where $\delta_L([x_i], a) = [x_i a]$ for all $1 \leq i \leq n$ and $a \in \Sigma$.

Note:

- The set of final states $\{[w] \mid w \in L\}$ is a set of equivalence classes and therefore a subset of the state set $\{[x_1], \ldots, [x_n]\}$ (the set of all equivalence classes).
- The transition function δ_L is well-defined due to the remark on the previous slide.
- For all $x \in \Sigma^*$, we have: $\widehat{\delta}_L([\varepsilon], x) = [x]$.

Equivalence Relations and Minimal Automaton

Claim: $T(M_L) = L$ (this shows that L is regular).

Proof of the claim:

$$\begin{aligned}
x \in T(M_L) &\iff \widehat{\delta}_L([\varepsilon], x) \in \{[w] \mid w \in L\} \\
&\iff [x] \in \{[w] \mid w \in L\} \\
&\iff \exists w \in L : [x] = [w] \\
&\iff \exists w \in L : x \, R_L \, w \\
&\iff x \in L
\end{aligned}$$

□

With the Myhill-Nerode theorem, one can also show that a language L is not regular.

To do this, one needs to find infinitely many words from Σ^* that lie in different R_L equivalence classes.

Example 3 for Myhill-Nerode equivalence:

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

Consider the words a, aa, aaa, $\ldots$, a^i, $\ldots$

It holds: $\neg(a^i \, R_L \, a^j)$ for $i \neq j$, since $a^i b^i \in L$ and $a^j b^i \notin L$.

Therefore, R_L has infinitely many equivalence classes, and L is not regular.

Equivalence Relations and Minimal Automaton

Let M be a DFA with n states. We say that M is a minimal DFA for the regular language L if

- $T(M) = L$, and
- there is no DFA M' with $T(M') = L$ and fewer than n states.

Let's reconsider the DFA M_L constructed on slide 155.

Theorem

Let L be regular.

1. M_L is a minimal DFA for L.
2. Let M be a DFA with $T(M) = L$ and all states being reachable from the initial state. Then:
   M is a minimal DFA for L if and only if $R_L = R_M$.
3. If M is a minimal DFA for L, then M can be obtained from M_L by renaming the states.

Proof:

Let $M = (Z, \Sigma, \delta_M, z_0, E)$ be an arbitrary DFA with $T(M) = L$.

Let $M_L = (\{[x_1], \ldots, [x_n]\}, \Sigma, \delta_L, [\varepsilon], \{[w] \mid w \in L\})$ be the equivalence class automaton.

For (1), we need to show that M_L has at most as many states as M.

From slide 153, we have seen that $\text{index}(R_L) \leq |Z|$.

Furthermore, the number of states of M_L is equal to $\text{index}(R_L)$.

This proves (1).

Assume that all states in M are reachable from the initial state z_0, but M is still not minimal for L.

Then we have $\text{index}(R_L) < |Z| = \text{index}(R_M)$
(see the last remark on slide 152).

Therefore, $R_L \neq R_M$.

On the other hand, if M is minimal for L, then we have
$|Z| =$ number of states of $M_L = \text{index}(R_L)$.

Since $|Z| = \text{index}(R_L) \leq \text{index}(R_M) \leq |Z|$ (see slide 153 below), it follows that $\text{index}(R_L) = \text{index}(R_M) < \infty$.

With $R_M \subseteq R_L$ (see slide 153 above), we obtain $R_M = R_L$.

This proves (2).

For (3), assume that M is minimal for L.

Then we have $R_M = R_L = R_{M_L}$ and $[x_1], \ldots, [x_n]$ are exactly the equivalence classes of $R_M = R_L$.

Define $f : Z \to \{[x_1], \ldots, [x_n]\}$ by $f(z) = \{w \in \Sigma^* \mid \widehat{\delta}_M(z_0, w) = z\}$.

Then f is a bijection.

Furthermore, the following holds:

- $f(z_0) = [\varepsilon]$ is the initial state of M_L.
- Let $z \in Z$ and let $w \in \Sigma^*$ such that $\widehat{\delta}_M(z_0, w) = z$ and hence $f(z) = [w]$. Then we have:

$$f(\delta_M(z, a)) = f(\widehat{\delta}_M(z_0, wa)) = [wa] = \delta_L([w], a) = \delta_L(f(z), a)$$

$$z \in E \iff w \in L \iff f(z) = [w] \text{ is a final state of } M_L$$

Equivalence Relations and Minimal Automaton

This means that we can form M_L from M by renaming each state $z \in Z$ to $f(z)$.

Or conversely: M is formed from the equivalence class automaton M_L by renaming each state $[x_i]$ to $f^{-1}([x_i])$. □

Remark: Thus, for a regular language, there is exactly one minimal DFA up to renaming of states.

The minimal DFA M_L for a regular language is, so to speak, a unique representative for L.

Next Goal: Construct the minimal automaton M_L from a non-minimal DFA $M = (Z, \Sigma, \delta, z_0, E)$ with $T(M) = L$.

First, we can assume that each state $z \in Z$ is reachable from the initial state z_0, i.e., $\exists x \in \Sigma^* : \widehat{\delta}(z_0, x) = z$.

Equivalence Relations and Minimal Automaton

If a state z is not reachable from the initial state, we can remove z from the DFA without changing the accepted language.
Note: If there is an edge from z' to z, then z' is also not reachable from z_0.

It holds:

$$
\begin{array}{cl}
 & M \text{ is not minimal for } L \\
\overset{\text{Slide 158}}{\iff} & R_M \subsetneq R_L \text{ (i.e., } R_M \subseteq R_L \text{ and } R_M \neq R_L) \\
\iff & \exists x, y \in \Sigma^* : (x, y) \in R_L \ \wedge \ (x, y) \notin R_M \\
\overset{\text{Slide 143}}{\iff} & \exists x, y \in \Sigma^* : \widehat{\delta}(z_0, x), \widehat{\delta}(z_0, y) \text{ are recognition-equivalent} \\
 & \qquad\qquad \wedge\ \widehat{\delta}(z_0, x) \neq \widehat{\delta}(z_0, y) \\
\iff & \exists z_1, z_2 \in Z : z_1 \text{ and } z_2 \text{ are recognition-equivalent and } z_1 \neq z_2
\end{array}
$$

For the last equivalence, we use that for every state $z \in Z$, there exists an $x \in \Sigma^*$ with $\widehat{\delta}(z_0, x) = z$.

Solution: In M, we merge all recognition-equivalent states.

To determine which states are recognition-equivalent, we mark all pairs of states $\{z, z'\}$ that are not recognition-equivalent.

We write pairs as 2-element subsets $\{z, z'\}$ because the order does not matter: $\{z, z'\} = \{z', z\}$.

Initially, certainly all pairs $\{z, z'\}$ with $z \in E$ and $z' \notin E$ are not recognition-equivalent, these pairs we mark at the beginning.

Suppose for a pair $\{z, z'\}$, there exists an $a \in \Sigma$ such that $\{\delta(z, a), \delta(z', a)\}$ are not recognition-equivalent.
Then, $\{z, z'\}$ is also not recognition-equivalent.

This observation allows us to mark additional pairs as not recognition-equivalent.

Equivalence Relations and Minimal Automata

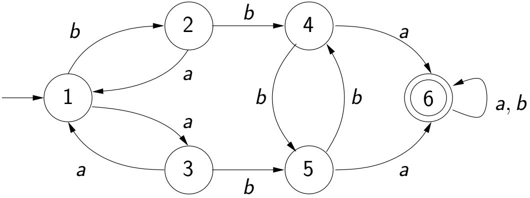
Minimal Automaton Algorithm

Input: DFA M (states that are not reachable from the start state have already been removed.)

Output: Sets of recognition-equivalent states

1. Create a table of all state pairs $\{z, z'\}$ with $z \neq z'$.
2. Mark all pairs $\{z, z'\}$ with $z \in E$ and $z' \notin E$.
3. For each unmarked pair $\{z, z'\}$ and each $a \in \Sigma$, test if $\{\delta(z, a), \delta(z', a)\}$ is already marked. If so, mark $\{z, z'\}$ as well.
4. Repeat the previous step until no changes occur in the table.
5. For all currently unmarked pairs $\{z, z'\}$, the states z and z' are recognition-equivalent.

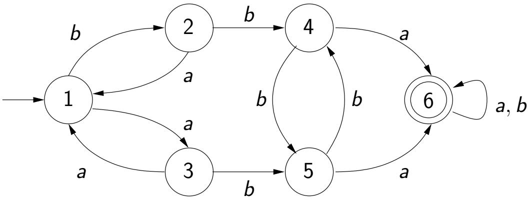
Example for the execution of the minimization algorithm:



2					
3					
4					
5					
6					
	1	2	3	4	5

Create a table of all state pairs.

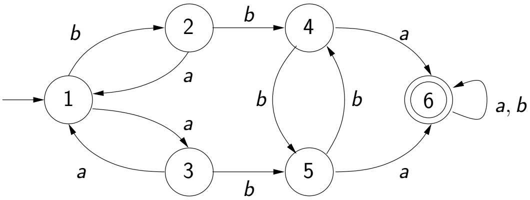
Example for the execution of the minimization algorithm:



	1	2	3	4	5
2					
3					
4					
5					
6	1	1	1	1	1

(1) Mark pairs of accepting and non-accepting states.

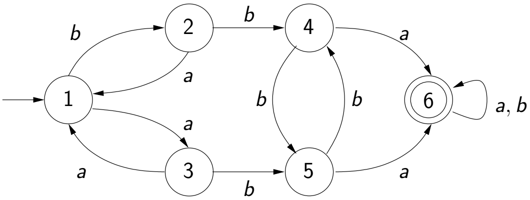
Example for the execution of the minimization algorithm:



	1	2	3	4	5
2					
3					
4		2			
5					
6	1	1	1	1	1

(2) Mark $\{2,4\}$ because $\delta(2,a) = 1$, $\delta(4,a) = 6$, and $\{1,6\}$ is marked.

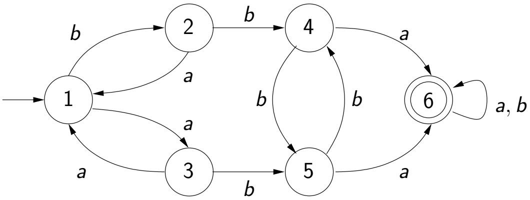
Example for the execution of the minimization algorithm:



2					
3					
4		2			
5			3		
6	1	1	1	1	1
	1	2	3	4	5

(3) Mark $\{3,5\}$ because $\delta(3,a) = 1$, $\delta(5,a) = 6$, and $\{1,6\}$ is marked.

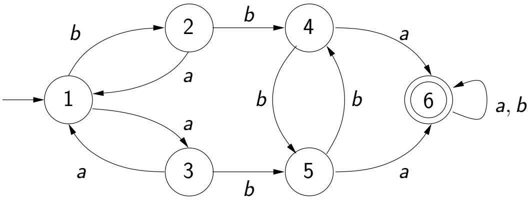
Example for the execution of the minimization algorithm:



2					
3					
4		2			
5		4	3		
6	1	1	1	1	1
	1	2	3	4	5

(4) Mark $\{2, 5\}$ because $\delta(2, a) = 1$, $\delta(5, a) = 6$, and $\{1, 6\}$ is marked.

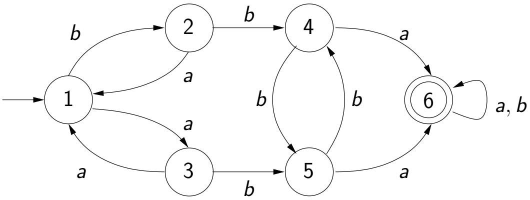
Example for the execution of the minimization algorithm:



	1	2	3	4	5
2					
3					
4		2	5		
5		4	3		
6	1	1	1	1	1

(5) Mark $\{3, 4\}$ because $\delta(3, a) = 1$, $\delta(4, a) = 6$, and $\{1, 6\}$ is marked.

Example for the execution of the minimization algorithm:

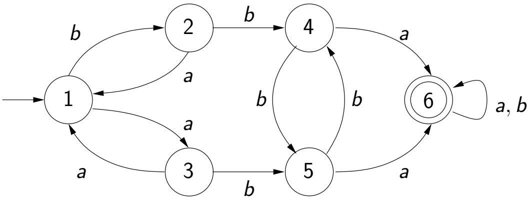


	1	2	3	4	5
2					
3					
4		2	5		
5	6	4	3		
6	1	1	1	1	1

(6) Mark $\{1,5\}$ because $\delta(1,a) = 3$, $\delta(5,a) = 6$, and $\{3,6\}$ is marked.

Equivalence Relations and Minimal Automata

Example for the execution of the minimization algorithm:

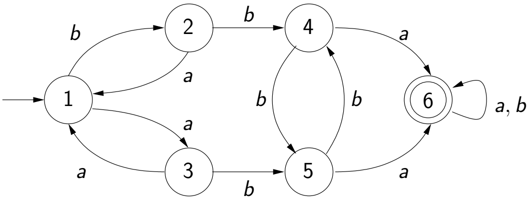


2					
3					
4	7	2	5		
5	6	4	3		
6	1	1	1	1	1
	1	2	3	4	5

(7) Mark $\{1, 4\}$ because $\delta(1, a) = 3$, $\delta(4, a) = 6$, and $\{3, 6\}$ is marked.

Equivalence Relations and Minimal Automata

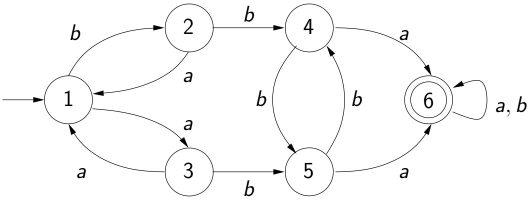
Example for the execution of the minimization algorithm:



	1	2	3	4	5
2					
3	8				
4	7	2	5		
5	6	4	3		
6	1	1	1	1	1

(8) Mark $\{1, 3\}$ because $\delta(1, b) = 2$, $\delta(3, b) = 5$, and $\{2, 5\}$ is marked.

Example for the execution of the minimization algorithm:

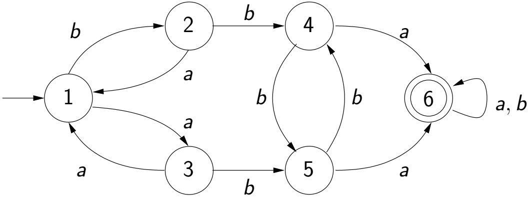


	1	2	3	4	5
2	9				
3	8				
4	7	2	5		
5	6	4	3		
6	1	1	1	1	1

(9) Mark $\{1,2\}$ because $\delta(1,b) = 2$, $\delta(2,b) = 4$, and $\{2,4\}$ is marked.

Equivalence Relations and Minimal Automata

Example for the execution of the minimization algorithm:



2	9				
3	8				
4	7	2	5		
5	6	4	3		
6	1	1	1	1	1
	1	2	3	4	5

The remaining state pairs $\{2, 3\}$ and $\{4, 5\}$ cannot be marked anymore. $\rightsquigarrow$ They are recognition-equivalent.

Equivalence Relations and Minimal Automaton

Theorem (Correctness of the Minimization Algorithm)

For a given DFA $M = (Z, \Sigma, \delta, z_0, E)$, the minimization algorithm marks a pair $\{z, z'\}$ ($z, z' \in Z$, $z \neq z'$) if and only if z and z' are not recognition-equivalent.

Proof:

(A) If $\{z, z'\}$ is marked, then z and z' are not recognition-equivalent.

Proof by induction on the time at which $\{z, z'\}$ is marked.

Base case: $\{z, z'\}$ is marked at the beginning because $z \in E$ and $z' \notin E$.

Then, z and z' are not recognition-equivalent.

Inductive step: $\{z, z'\}$ is eventually marked because there exists an $a \in \Sigma$ such that $\{\delta(z, a), \delta(z', a)\}$ was marked at an earlier time.

By the induction hypothesis, $\delta(z, a)$ and $\delta(z', a)$ are not recognition-equivalent.

Thus, z and z' are also not recognition-equivalent.

(B) If z and z' are not recognition-equivalent, then $\{z, z'\}$ will eventually be marked.

Let z and z' be not recognition-equivalent.

Let $\lambda(z, z')$ be the length of a shortest word w such that $\widehat{\delta}(z, w) \in E$ and $\widehat{\delta}(z', w) \notin E$ (or vice versa).

We will show by induction on $\lambda(z, z')$ that $\{z, z'\}$ will be marked.

Base case: $\lambda(z, z') = 0$

Then, $z \in E$ and $z' \notin E$.

Thus, $\{z, z'\}$ will be marked at the beginning.

Inductive step: Let $\lambda(z, z') > 0$.

Then there is a word au ($a \in \Sigma$ and $u \in \Sigma^*$) with $|au| = \lambda(z, z')$, such that

$$\widehat{\delta}(z, au) = \widehat{\delta}(\delta(z, a), u) \in E, \quad \widehat{\delta}(z', au) = \widehat{\delta}(\delta(z', a), u) \notin E$$

(or vice versa).

Then $\delta(z, a)$ and $\delta(z', a)$ are also not recognition-equivalent, and $\lambda(\delta(z, a), \delta(z', a)) \leq |u| < \lambda(z, z')$.

By the induction hypothesis, $\{\delta(z, a), \delta(z', a)\}$ will eventually be marked.

Thus, $\{z, z'\}$ will also eventually be marked. $\square$

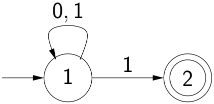
Hints for performing the minimization algorithm:

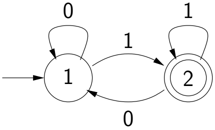
- Set up the table in such a way that each pair appears only once! So, for a state set $\{1, \ldots, n\}$:
  Write $2, \ldots, n$ vertically and $1, \ldots, n-1$ horizontally.
- Please indicate which states were marked in which order and why!
  In Schöning's book, only asterisks $(*)$ are used, but in the correction, the order and reasons for marking are not apparent.

For non-deterministic automata, the following statements can be made:

- There is *not a* minimal NFA, but there can be multiple minimal NFAs.

  The following two minimal NFAs recognize $L = ((0|1)^*1)$ and have two states (it is not possible to recognize L with only one state).

- Given a DFA M, a minimal NFA that recognizes $T(M)$ will always have at most as many states as M, because M itself is already an NFA.

  Furthermore: the minimal NFA can be exponentially smaller than the minimal DFA.

  See $L_k = \{x \in \{0,1\}^* \mid |x| \geq k, \text{the } k\text{-th last symbol of } x \text{ is } 0\}$.

Decidability

We now discuss whether there are methods to decide the following questions or problems for regular languages. Here, we assume that regular languages are given as DFAs, NFAs, grammars, or regular expressions.

Problems

- Word Problem: Does $w \in L$ hold for a given regular language L and $w \in \Sigma^*$?
- Emptiness Problem: Does $L = \emptyset$ hold for a given regular language L?
- Finiteness Problem: Is a given regular language L finite?
- Intersection Problem: Does $L_1 \cap L_2 = \emptyset$ hold for given regular languages L_1, L_2?
- Inclusion Problem: Does $L_1 \subseteq L_2$ hold for given regular languages L_1, L_2?
- Equivalence Problem: Does $L_1 = L_2$ hold for given regular languages L_1, L_2?

Decidability

Word Problem:

Let $L \subseteq \Sigma^*$ be a regular language, given by a DFA $M = (Z, \Sigma, \delta, z_0, E)$ with $T(M) = L$, and let $w \in \Sigma^*$.

Question: Is $w \in L$?

Solution:

Let $w = a_1 a_2 \cdots a_n$ with $a_i \in \Sigma$.

Follow the state transitions of M as determined by the symbols $a_1, \ldots, a_n$:

$z := z_0$
for $i := 1$ **to** n **do**
    $z := \delta(z, a_i)$
endfor
if $z \in E$ **then return**(YES) **else return**(NO)

Emptiness Problem:

Let $M = (Z, \Sigma, \delta, S, E)$ be an NFA.

Question: Is $T(M) \neq \emptyset$?

Solution:

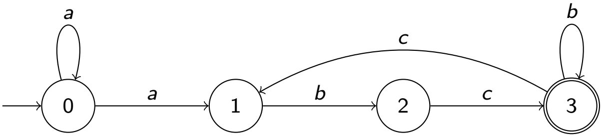
Let $G = (Z, \to)$ be the directed graph with

$$z \to z' \quad \Longleftrightarrow \quad \exists a \in \Sigma : z' \in \delta(z, a).$$

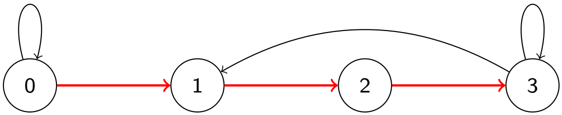
Then, $T(M) \neq \emptyset$ if and only if there exists a (possibly empty) path in the graph G from a node in S to a node in E.

This can be decided, for example, using depth-first or breadth-first search (see the lecture *Algorithms and Data Structures*).

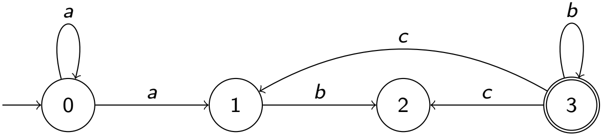
Example 1: Consider the following automaton M.



The automaton recognizes a non-empty language, as demonstrated by the following path in the graph G:



Example 2: The following automaton accepts the empty language because there is no path in the graph G from 0 to 3:



Finiteness Problem:

Let $M = (Z, \Sigma, \delta, S, E)$ be an NFA.

Question: Is $T(M)$ finite?

Solution:

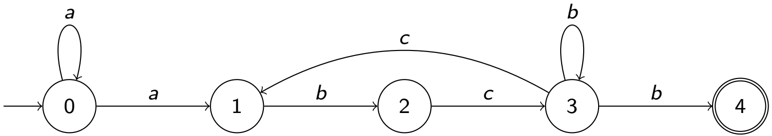
Let G be defined as on the previous slide.

Then: $T(M)$ is infinite if and only if there exist states $z_0 \in S$, $z \in Z$, and $z_1 \in E$ such that:

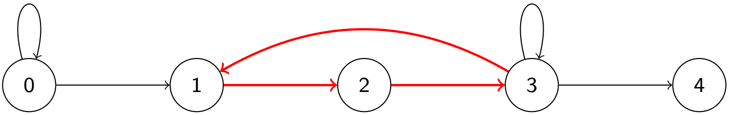
- $z_0 \to^* z$ (z is reachable from the initial state z_0),
- $z \to^+ z$ (there is a path from z back to itself with at least one edge, i.e., z lies on a cycle),
- $z \to^* z_1$ (from z, the final state z_1 can be reached).

This can again be determined using depth-first or breadth-first search.

Example: Consider again the following automaton M.



The automaton recognizes an infinite language: The red circle is reachable from state 0, and from the red circle, one can reach node 4 (the accepting state of the NFA).



Intersection Problem:

Let M_1 and M_2 be NFAs.

Question: Does $T(M_1) \cap T(M_2) = \emptyset$?

Solution:

Construct the product automaton M from M_1 and M_2
($\leadsto T(M) = T(M_1) \cap T(M_2)$), see Slide 125.

Test if $T(M) = \emptyset$.

Inclusion Problem:

Let M_1 and M_2 be NFAs.

Question: Does $T(M_1) \subseteq T(M_2)$?

Solution: From M_1 and M_2, we can construct an NFA M with $T(M) = \overline{T(M_2)} \cap T(M_1)$.

It holds that $T(M_1) \subseteq T(M_2)$ if and only if $T(M) = \emptyset$.

Equivalence Problem:

Let M_1 and M_2 be NFAs.

Question: Does $T(M_1) = T(M_2)$?

Solution 1:

It holds that $T(M_1) = T(M_2)$ if and only if $T(M_1) \subseteq T(M_2)$ and $T(M_2) \subseteq T(M_1)$.

Solution 2:

For each M_i ($i \in \{1, 2\}$), determine an equivalent minimal DFA N_i.

Then we have: $T(M_1) = T(M_2) \Leftrightarrow T(N_1) = T(N_2) \Leftrightarrow N_1$ and N_2 are isomorphic (i.e., they can be transformed into each other by renaming the states).

Decidability

Efficiency Considerations:

Depending on the representation of a regular language L, the runtime of the procedures described above can vary significantly.

Example: Equivalence Problem $L_1 = L_2$:

- L_1, L_2 given as DFAs

  $\rightsquigarrow$ Runtime $O(n^2)$

- L_1, L_2 given as grammars, regular expressions, or NFAs

  $\rightsquigarrow$ Complexity NP-hard

  This means, among other things, that it is not known whether this problem can be solved in polynomial time.

  More on the complexity class NP and related issues $\rightsquigarrow$ Master's course on Complexity Theory.

Context-Free Languages

We now discuss context-free or Type-2-languages.

Review: Productions of Context-Free Grammars

In context-free grammars, all productions are of the form $A \to w$, where $A \in V$ (i.e., A is a variable) and $w \in (V \cup \Sigma)^+$.

Exception (ε-special rule): If $S \to \varepsilon$, the start symbol S must not appear on the right-hand side of any production.

Considered example grammars:

- A grammar that generates correctly parenthesized arithmetic expressions
- A grammar that generates sentences of natural language

Another example: the language $L = \{a^k b^k \mid k \geq 0\}$ is context-free.

Productions: $S \to \varepsilon \mid T$, $T \to ab \mid aTb$

Context-Free Languages

Applications of Context-Free Languages

Main application: Description of the syntax of programming languages

Many of the techniques discussed here are therefore of interest for use in compiler construction.

Remark: A grammar that describes a natural language may not be context-free, despite having some context-free components, because natural language involves many subtle contextual dependencies that need to be taken into account.

To date, no one has succeeded in creating a complete grammar for all correct sentences in natural language.
Question: What exactly constitutes a correct sentence?

Context-Free Languages

Content of the section "Context-Free Languages"

- Normal Forms – To apply certain methods/techniques, it is important to convert a grammar into a specific normal form.
- Pumping Lemma for context-free languages
- Closure Properties – Context-free languages do not behave as well as regular languages in terms of closure properties.
- Word Problem – and the algorithm to solve the word problem (CYK algorithm)
- Pushdown Automata – The automaton model for context-free languages

We will first revisit the "ε special rule":

The definition for context-free grammars (with the ε special rule) requires that S must not appear on the right-hand side if $S \to \varepsilon$ is a production. Moreover, no other productions of the form $A \to \varepsilon$ are allowed.

What happens if these conditions are relaxed and arbitrary rules of the form $A \to \varepsilon$ are allowed? Can this lead to a non-context-free language?

Answer: No

Normal Forms

Theorem (ε-free Grammars)

Given a grammar $G = (V, \Sigma, P, S)$, whose productions are all of the form $A \to w$ for $A \in V$, $w \in (V \cup \Sigma)^*$.

Then there exists a context-free grammar $G' = (V, \Sigma, P', S)$ such that:

- all productions in P' are of the form $A \to w$ with $A \in V$, $w \in (V \cup \Sigma)^+$, and
- $L(G') = L(G) \setminus \{\varepsilon\}$.

Hence, ε-productions can be freely used. They do not alter the expressive power of context-free grammars.

Proof:

Let $V_\varepsilon = \{A \in V \mid A \Rightarrow_G^* \varepsilon\}$ be the set of all variables from which the empty word can be derived.

Normal Forms

The set V_ε can be computed using the following algorithm:

$U := \emptyset$
$V_\varepsilon := \{A \in V \mid (A \to \varepsilon) \in P\}$
while $U \neq V_\varepsilon$ **do**
    $U := V_\varepsilon$
    $V_\varepsilon := U \cup \{A \in V \mid \exists w \in U^+ : (A \to w) \in P\}$
endwhile

Then the following holds:

- If a variable A is eventually added to the set V_ε, then $A \Rightarrow_G^* \varepsilon$.

  This is easily shown by induction on the time t when A is added to the set V_ε.

Normal Forms

- If $A \Rightarrow_G^* \varepsilon$, then eventually A will be added to the set V_ε.

  This can be shown by induction on the length ℓ of the derivation $A \Rightarrow_G^* \varepsilon$:

  If $(A \to \varepsilon) \in P$, then A is added to V_ε at the very beginning.

  Otherwise, there is a production $(A \to A_1 A_2 \cdots A_n) \in P$ with $A_i \Rightarrow_G^* \varepsilon$ for all $1 \leq i \leq n$, where the derivation $A_i \Rightarrow_G^* \varepsilon$ has length $< \ell$.

  By induction, each variable A_i $(1 \leq i \leq n)$ will eventually be added to V_ε.

  Thus, the same holds for A.

For a non-empty word $w \in (V \cup \Sigma)^+$, we define the set of words $F(w) \subseteq (V \cup \Sigma)^+$ as follows:

Let $w = w_0 A_1 w_1 A_2 \cdots w_{n-1} A_n w_n$, where $n \geq 0$, $A_1, \dots, A_n \in V_\varepsilon$ and no variable from V_ε appears in the word $w_0 w_1 \cdots w_n$. Then define

$$F(w) = \{w_0 A_1^{e_1} w_1 A_2^{e_2} \cdots w_{n-1} A_n^{e_n} w_n \mid e_1, \dots e_n \in \{0,1\}\} \setminus \{\varepsilon\},$$

where $A_i^0 = \varepsilon$ and $A_i^1 = A_i$.

Intuitively: All words that can be formed from w by deleting some (but not necessarily all) occurrences of variables from V_ε, excluding the empty word.

We can now define the production set P' of the ε-free grammar G' as:

$$P' = \{A \to w' \mid \exists w : (A \to w) \in P \text{ and } w' \in F(w)\}.$$

Normal Forms

Claim: $L(G') = L(G) \setminus \{\varepsilon\}$

Proof of the Claim:

- $L(G') \subseteq L(G) \setminus \{\varepsilon\}$: By the construction of G', we have $\varepsilon \notin L(G')$.

  Furthermore, for each production $(A \to w') \in P'$ of G':

  $$A \Rightarrow_G^* w'.$$

  This implies $L(G') \subseteq L(G) \setminus \{\varepsilon\}$.

- $L(G) \setminus \{\varepsilon\} \subseteq L(G')$: By induction on the length of derivations, we show for all nonterminals $A \in V$ and words $w \in \Sigma^+$:

  $$A \Rightarrow_G^* w \quad \text{implies} \quad A \Rightarrow_{G'}^* w.$$

  So suppose $A \Rightarrow_G^* w$.

Normal Forms

If $(A \to w) \in P$, then $(A \to w) \in P'$ and thus $A \Rightarrow^*_{G'} w$.

Suppose the derivation $A \Rightarrow^*_G w$ has length at least 2.

There must be a production $(A \to w_0 A_1 w_1 A_2 w_2 \cdots A_n w_n) \in P$ and shorter derivations $A_i \Rightarrow^*_G u_i$ $(1 \le i \le n)$ with $w = w_0 u_1 w_1 u_2 w_2 \cdots u_n w_n$.

Let $J = \{i \mid 1 \le i \le n, u_i = \varepsilon\}$.

Let w' be the word that results from $w_0 A_1 w_1 A_2 w_2 \cdots A_n w_n$ by replacing all A_i with $i \in J$ by ε (note: $A_i \in V_\varepsilon$ for all $i \in J$).

Since $w \neq \varepsilon$, it must also be that $w' \neq \varepsilon$.

By the definition of P', $(A \to w') \in P'$.

Furthermore, by induction: $A_i \Rightarrow^*_{G'} u_i$ for all $i \in \{1, \ldots, n\} \setminus J$.

Altogether, we obtain $A \Rightarrow^*_{G'} w$. □

Normal Forms

The theorem just proven shows in particular:

Theorem

Let $G = (V, \Sigma, P, S)$ be a grammar whose productions are all of the form $A \to w$ for $A \in V$, $w \in (V \cup \Sigma)^*$. Then $L(G)$ is context-free.

Proof: Construct from G a context-free grammar G' such that $L(G') = L(G) \setminus \{\varepsilon\}$ and G' contains no productions of the form $A \to \varepsilon$.

If $\varepsilon \notin L(G)$, then $L(G') = L(G)$.

Now, assume $\varepsilon \in L(G)$.

Take a new start symbol S' and add the productions $S' \to \varepsilon \mid S$ to G'.

The resulting grammar H is context-free (with ε-special rule), and we have $L(G) = L(G') \cup \{\varepsilon\} = L(H)$. □

Example: Consider the grammar G with the following productions:

$$S \to aABC, \quad A \to \varepsilon \mid AA, \quad B \to \varepsilon \mid BbA, \quad C \to \varepsilon \mid CAc$$

We have $V_\varepsilon = \{A, B, C\}$.

For the (non-empty) right-hand sides of the grammar, we get:

- $F(aABC) = \{aABC, aBC, aAC, aAB, aA, aB, aC, a\}$
- $F(AA) = \{AA, A\}$
- $F(BbA) = \{BbA, bA, Bb, b\}$
- $F(CAc) = \{CAc, Ac, Cc, c\}$

Note: $\varepsilon \notin L(G)$. Therefore, the grammar G' with the following productions satisfies $L(G) = L(G')$:

$$
\begin{aligned}
S &\to aABC \mid aBC \mid aAC \mid aAB \mid aA \mid aB \mid aC \mid a \\
A &\to AA \mid A \\
B &\to BbA \mid bA \mid Bb \mid b \\
C &\to CAc \mid Ac \mid Cc \mid c
\end{aligned}
$$

Remark: The set defined on slide 190 can contain up to 2^n words. This can cause the constructed grammar G' to become quite large.

Normal Forms

We now consider another important normal form:

Definition (Chomsky Normal Form)

A context-free grammar G with $\varepsilon \notin L(G)$ is in Chomsky Normal Form (CNF), if all productions have one of the following two forms:

$$A \to BC \qquad A \to a$$

where $A, B, C \in V$ are variables and $a \in \Sigma$ is a terminal symbol.

Theorem (Conversion to Chomsky Normal Form)

For every context-free grammar G with $\varepsilon \notin L(G)$, there exists a grammar G' in Chomsky Normal Form with $L(G) = L(G')$.

Normal Forms

Proof:

Step 1:

Based on the theorem "ε-free grammars" (Slide 187), we can assume that G has no productions of the form $A \to \varepsilon$.

Step 2:

For each terminal symbol $a \in \Sigma$, we introduce a new variable $A_a \notin V$ along with the production $A_a \to a$.

Then, we can replace each occurrence of a in a right-hand side $\neq a$ by A_a.

After this, all productions will be of the form $A \to a$ or $A \to A_1 \cdots A_n$ with $a \in \Sigma$, $n \geq 1$, and variables $A_1, \ldots, A_n$.

Step 3: Elimination of chain rules.

We now eliminate all productions of the form $A \to B$ for variables A, B (chain rules) as follows:

For each variable A, we add the production $A \to \alpha$ if α is not a variable and there exists a variable B such that $A \Rightarrow^* B \to \alpha$.

Afterward, we can remove all chain rules.

All productions now have the form $A \to a$ or $A \to A_1 \cdots A_n$ with $a \in \Sigma$, $n \geq 2$, and variables $A_1, \ldots, A_n$.

Step 4: Elimination of productions of the form $A \to A_1 \cdots A_n$ with $n \geq 3$.

Let $A \to A_1 \cdots A_n$ be a production with $n \geq 3$.

We introduce new variables $B_2, \ldots, B_{n-1}$ and replace the production $A \to A_1 \cdots A_n$ with the following productions:

$$A \to A_1 B_2, \quad B_i \to A_i B_{i+1} \ (2 \leq i \leq n-2), \quad B_{n-1} \to A_{n-1} A_n$$

$\square$

Example: Let

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

with the following production set P:

$$\begin{aligned} S &\to aAb \\ A &\to S \mid aaSc \mid \varepsilon \end{aligned}$$

We transform G into CNF.

Step 1: We make G ε-free.

This results in the following productions:

$$\begin{aligned} S &\to aAb \mid ab \\ A &\to S \mid aaSc \end{aligned}$$

Normal Forms

Step 2: This results in the following productions:

$$\begin{aligned} S &\to A_a A A_b \mid A_a A_b \\ A &\to S \mid A_a A_a S A_c \\ A_a &\to a \\ A_b &\to b \\ A_c &\to c \end{aligned}$$

Step 3: Elimination of Chain Rules.

The only chain rule in our grammar is $A \to S$. Its elimination results in the following productions:

$$\begin{array}{rcl} S & \to & A_a A A_b \mid A_a A_b \\ A & \to & A_a A A_b \mid A_a A_b \mid A_a A_a S A_c \\ A_a & \to & a \\ A_b & \to & b \\ A_c & \to & c \end{array}$$

Step 4: Elimination of rules of the form $A \to A_1 \cdots A_n$ with $n \geq 3$.

$$
\begin{aligned}
S &\to A_a B \mid A_a A_b \\
A &\to A_a B \mid A_a A_b \mid A_a C \\
B &\to A A_b \\
C &\to A_a S A_c \\
A_a &\to a \\
A_b &\to b \\
A_c &\to c
\end{aligned}
$$

Step 4: Elimination of rules of the form $A \to A_1 \cdots A_n$ with $n \geq 3$.

$$
\begin{aligned}
S &\to A_a B \mid A_a A_b \\
A &\to A_a B \mid A_a A_b \mid A_a C \\
B &\to A A_b \\
C &\to A_a D \\
D &\to S A_c \\
A_a &\to a \\
A_b &\to b \\
A_c &\to c
\end{aligned}
$$

Normal Forms

Definition (Greibach Normal Form)

A context-free grammar $G = (V, \Sigma, P, S)$ with $\varepsilon \notin L(G)$ is in Greibach Normal Form, if all productions in P have the following form:

$$A \to aB_1 B_2 \dots B_k \qquad \text{with } k \geq 0$$

Here, $A, B_1, \dots, B_k \in V$ are variables and $a \in \Sigma$ is an alphabet symbol.

The Greibach Normal Form guarantees that at every derivation step exactly one alphabet symbol is produced.
It is useful to show that pushdown automata (i.e., automata for context-free languages) do not require ε-transitions.

Theorem (Conversion to Greibach Normal Form)

For every context-free grammar G with $\varepsilon \notin L(G)$, there exists a grammar G' in Greibach Normal Form such that $L(G) = L(G')$.

Normal Forms

Proof: Let $G = (V, \Sigma, P, S)$ be a context-free grammar with $\varepsilon \notin L(G)$.

Preliminary Consideration:

Suppose there are the following productions for a variable A in P:

$$A \to A\alpha_1 \mid \cdots \mid A\alpha_k \mid \beta_1 \mid \cdots \mid \beta_\ell.$$

Here, $\alpha_1, \ldots, \alpha_k, \beta_1, \ldots, \beta_\ell \in (V \cup \Sigma)^*$ and $\beta_1, \ldots, \beta_\ell$ do not begin with A.

Then, with these productions, the same sentence forms can be generated as with

$$\begin{aligned} A &\to \beta_1 \mid \cdots \mid \beta_\ell \mid \beta_1 B \mid \cdots \mid \beta_\ell B \\ B &\to \alpha_1 \mid \cdots \mid \alpha_k \mid \alpha_1 B \mid \cdots \mid \alpha_k B. \end{aligned}$$

Both rule sets can generate all sentence forms from

$$(\beta_1 \mid \cdots \mid \beta_\ell)(\alpha_1 \mid \cdots \mid \alpha_k)^*$$

.

Normal Forms

Let $A_1, \ldots, A_m$ be an arbitrary enumeration of all the variables of G.

Step 1: Using the algorithm on the next slide, we transform G into an equivalent context-free grammar in which all productions of the form $A_i \to \alpha$ satisfy:

$$\alpha = a\beta \text{ with } a \in \Sigma, \beta \in V^* \text{ or } \alpha = A_j\beta \text{ with } j > i, \beta \in V^*$$

Without loss of generality, we can assume that G is in Chomsky Normal Form.

Normal Forms

for $i := 1$ **to** m **do**
  for $j := 1$ **to** $i - 1$ **do**
    for all $(A_i \to A_j\alpha) \in P$ **do**
      Let $A_j \to \beta_1 \mid \cdots \mid \beta_n$ be all rules with left-hand side $= A_j$.
      $P := (P \cup \{A_i \to \beta_1\alpha \mid \cdots \mid \beta_n\alpha\}) \setminus \{A_i \to A_j\alpha\}$
    endfor
  endfor
  if there are productions of the form $A_i \to A_i\alpha$ **then**
    Apply the transformation from the preliminary consideration to A_i (introducing a new variable B_i).
  endif
endfor

After Step 1, all productions with left-hand side $= A_m$ are of the form $A_m \to a\alpha$ with $a \in \Sigma, \alpha \in V^*$.

Step 2: The following algorithm ensures that all productions with left-hand side A_i begin with a terminal symbol on the right-hand side.

for $i := m - 1$ **downto** 1 **do**
    forall $(A_i \to A_j\alpha) \in P$ with $j > i$ **do**
        Let $A_j \to \beta_1 \mid \cdots \mid \beta_n$ be all rules with left-hand side $= A_j$.
        $P := (P \cup \{A_i \to \beta_1\alpha \mid \cdots \mid \beta_n\alpha\}) \setminus \{A_i \to A_j\alpha\}$
    endfor
endfor

After Step 2, all productions with left-hand side $= A_i$ $(1 \leq i \leq m)$ are in Greibach Normal Form.

However, the productions introduced for the new variables B_i in Step 1 may not be in Greibach Normal Form.

Normal Forms

Let $B_i \to A_j\alpha$ be a rule that violates the Greibach Normal Form.

Let $A_j \to \beta_1 \mid \cdots \mid \beta_k$ be all productions with left-hand side $= A_j$.

Then $\beta_1, \ldots, \beta_k$ begin with terminal symbols.

Replace $B_i \to A_j\alpha$ by $B_i \to \beta_1\alpha \mid \cdots \mid \beta_k\alpha$.

Now the grammar is in Greibach Normal Form. □

Example: Let G be the grammar in CNF with the following productions:

$$\begin{aligned} A_1 &\to A_2A_3 \\ A_2 &\to A_3A_1 \mid b \\ A_3 &\to A_1A_2 \mid a. \end{aligned}$$

In Step 1, only the production $A_3 \to A_1A_2$ in the iteration $i = 3$ is replaced as follows:

- For $j = 1$: $A_3 \to A_2A_3A_2$
- For $j = 2$: $A_3 \to A_3A_1A_3A_2 \mid bA_3A_2$

Now a new variable B_3 is introduced, and the productions

$$A_3 \to A_3A_1A_3A_2 \mid bA_3A_2 \mid a$$

are replaced by

$$\begin{aligned} A_3 &\to bA_3A_2B_3 \mid aB_3 \mid bA_3A_2 \mid a \\ B_3 &\to A_1A_3A_2B_3 \mid A_1A_3A_2. \end{aligned}$$

We now have the following grammar after Step 1:

$$\begin{array}{rcl} A_1 & \to & A_2A_3 \\ A_2 & \to & A_3A_1 \mid b \\ A_3 & \to & bA_3A_2B_3 \mid aB_3 \mid bA_3A_2 \mid a \\ B_3 & \to & A_1A_3A_2B_3 \mid A_1A_3A_2. \end{array}$$

Note: All productions for A_3 indeed begin with a terminal symbol on the right-hand side.

After Step 2, iteration $i = 2$:

$$\begin{array}{rcl} A_1 & \to & A_2A_3 \\ A_2 & \to & bA_3A_2B_3A_1 \mid aB_3A_1 \mid bA_3A_2A_1 \mid aA_1 \mid b \\ A_3 & \to & bA_3A_2B_3 \mid aB_3 \mid bA_3A_2 \mid a \\ B_3 & \to & A_1A_3A_2B_3 \mid A_1A_3A_2 \end{array}$$

After Step 2, iteration $i = 1$:

$$
\begin{aligned}
A_1 &\rightarrow bA_3A_2B_3A_1A_3 \mid aB_3A_1A_3 \mid bA_3A_2A_1A_3 \mid aA_1A_3 \mid bA_3 \\
A_2 &\rightarrow bA_3A_2B_3A_1 \mid aB_3A_1 \mid bA_3A_2A_1 \mid aA_1 \mid b \\
A_3 &\rightarrow bA_3A_2B_3 \mid aB_3 \mid bA_3A_2 \mid a \\
B_3 &\rightarrow A_1A_3A_2B_3 \mid A_1A_3A_2
\end{aligned}
$$

Now, in the right-hand sides of the B_3 productions, A_1 must be replaced by the right-hand sides of A_1:

Normal Forms

$$
\begin{aligned}
A_1 &\to bA_3A_2B_3A_1A_3 \mid aB_3A_1A_3 \mid bA_3A_2A_1A_3 \mid aA_1A_3 \mid bA_3 \\
A_2 &\to bA_3A_2B_3A_1 \mid aB_3A_1 \mid bA_3A_2A_1 \mid aA_1 \mid b \\
A_3 &\to bA_3A_2B_3 \mid aB_3 \mid bA_3A_2 \mid a \\
B_3 &\to bA_3A_2B_3A_1A_3A_3A_2B_3 \mid aB_3A_1A_3A_3A_2B_3 \mid bA_3A_2A_1A_3A_3A_2B_3 \mid \\
&\quad\; aA_1A_3A_3A_2B_3 \mid bA_3A_3A_2B_3 \mid bA_3A_2B_3A_1A_3A_3A_2 \mid \\
&\quad\; aB_3A_1A_3A_3A_2 \mid bA_3A_2A_1A_3A_3A_2 \mid aA_1A_3A_3A_2 \mid bA_3A_3A_2
\end{aligned}
$$

Normal Forms

Remark on the empty word ε: With grammars in Chomsky Normal Form (CNF) or Greibach Normal Form (GNF), only context-free languages L with $\varepsilon \notin L$ can be generated.

Now, if you have a context-free grammar G with $\varepsilon \in L(G)$, you can proceed as follows:

- Construct from G a context-free grammar G' with $L(G') = L(G) \setminus \{\varepsilon\}$ (see the theorem on slide 187).
- Convert G' into a grammar G'' in Chomsky Normal Form or Greibach Normal Form.

  Let S be the start symbol of G'', and let $S \to \alpha_1 \mid \cdots \mid \alpha_n$ be all productions in G'' with left-hand side $= S$.
- Take a new start symbol S' and add the productions $S' \to \varepsilon \mid \alpha_1 \mid \cdots \mid \alpha_n$ to G''.

  For the resulting grammar H, it holds that $L(G) = L(H)$, and all productions in H are in Chomsky Normal Form or Greibach Normal Form, except for the production $S' \to \varepsilon$.

Analogous to regular languages, we can now prove a Pumping Lemma for context-free languages.

The statement valid for regular languages and finite automata

Any sufficiently long word passes through a state of the automaton twice.

is replaced by

On a path of the syntax tree, which represents the derivation of a sufficiently long word by a context-free grammar, a variable appears at least twice.

Pumping Lemma

What does "sufficiently long word" mean here?

The answer to this question depends on the form of the grammar.

We assume that the grammar is in Chomsky Normal Form.

Then, syntax trees are (except for the bottom layer of the leaves) always binary trees (due to productions of the form $A \to BC$).

For binary trees, the following holds:

Lemma (Path length in binary trees)

Let B be a binary tree (i.e., each node in B has either zero or two children) with at least 2^k leaves.
Then, B has a path from the root to a leaf consisting of at least k edges and $k+1$ nodes.

Pumping Lemma

Proof: Induction on k.

Base case: $k = 0$.

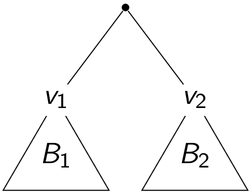
Let B be a binary tree with at least $2^0 = 1$ leaf.

Then, B has a path that consists of at least one node (namely, the root).

Inductive step: $k \geq 0$.

Let B be a binary tree with at least $2^{k+1} = 2^k + 2^k$ leaves.

Let v_1 and v_2 be the two children of the root, and let B_1 and B_2 be the binary trees with roots v_1 and v_2, respectively:



Then, either B_1 or B_2 must have at least 2^k leaves: If both B_1 and B_2 had strictly fewer than 2^k leaves, then the tree B would have strictly fewer than $2^k + 2^k = 2^{k+1}$ leaves.

Without loss of generality, assume B_1 has at least 2^k leaves.

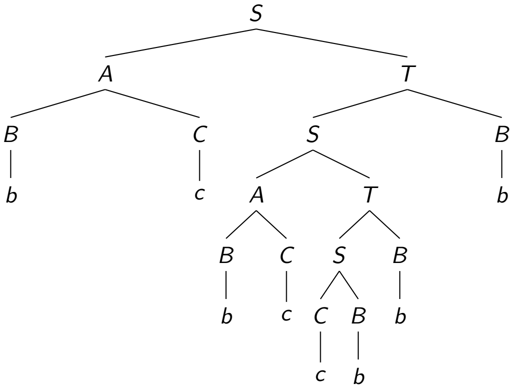
By the inductive hypothesis, there is a path in B_1 from the root v_1 to a leaf with at least k edges and $k+1$ nodes.

By adding the edge from the root to v_1, we obtain a path in B from the root to a leaf with at least $k+1$ edges and $k+2$ nodes. $\square$

Example: Let the context-free grammar G (in CNF) consist of the following productions:

$$S \to AT \mid CB,\ T \to SB,\ A \to BC,\ B \to b,\ C \to c.$$

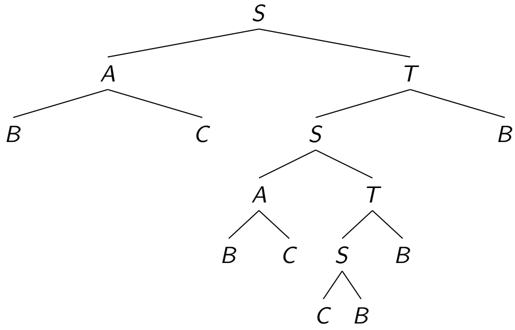
Consider the following syntax tree:



Example: Let the context-free grammar G (in CNF) consist of the following productions:

$$S \to AT \mid CB, \;\; T \to SB, \;\; A \to BC, \;\; B \to b, \;\; C \to c.$$

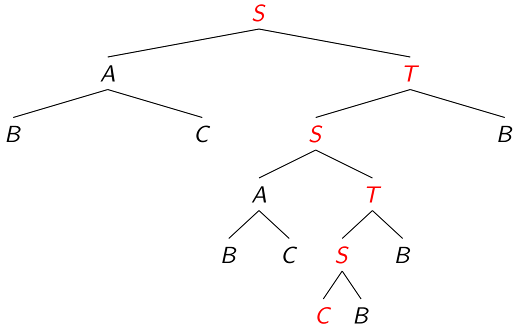
Removing the leaves results in a binary tree:



Example: Let the context-free grammar G (in CNF) consist of the following productions:

$$S \to AT \mid CB,\ \ T \to SB,\ \ A \to BC,\ \ B \to b,\ \ C \to c.$$

Removing the leaves results in a binary tree:



Pumping Lemma

Let $G = (V, \Sigma, P, S)$ be a grammar in Chomsky Normal Form with $k = |V|$ variables.

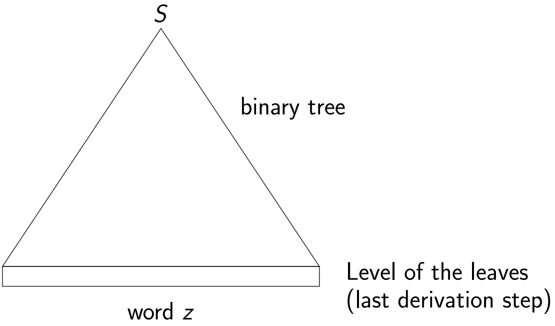
Let $z \in L(G)$.

- If $|z| \geq 2^k$, then every syntax tree for z obviously has at least 2^k leaves.
- Consider a syntax tree for z and remove the leaves labeled with terminal symbols.
  This results in a binary tree T.
- Consider the longest path in T from the root to a leaf.
- The lemma from slide 217 implies that this path has at least $k + 1 > |V|$ nodes.
- Thus, some variable A appears at least twice on the path (we will refer to this as a double occurrence in the following).

Pumping Lemma

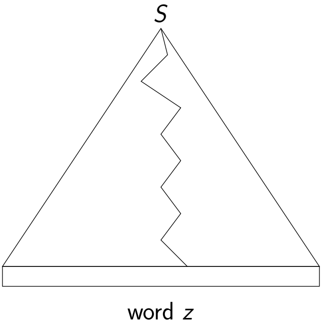
Syntax tree for a word z with $|z| \geq n = 2^k$

Here, n is the "constant of the pumping lemma".



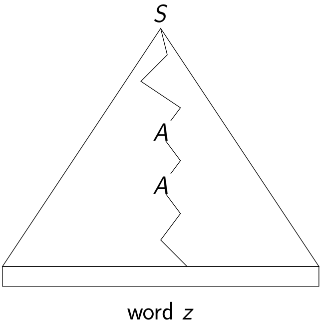
Pumping Lemma

The longest path has at least $k+1$ internal nodes.



Pumping Lemma

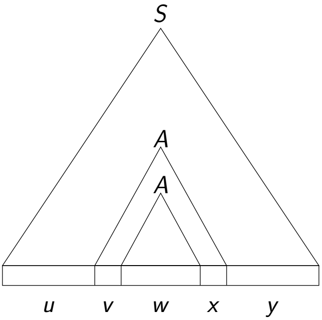
On this path, there is a variable that appears twice, such as A.



Pumping Lemma

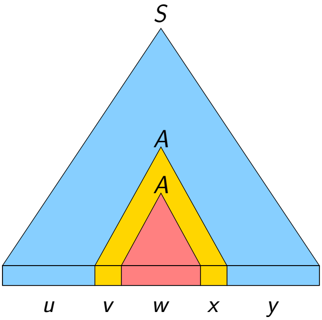
The word z is now split into five substrings u, v, w, x, y:

- w is derived from the lower A: $A \Rightarrow^* w$
- vwx is derived from the upper A: $A \Rightarrow^* vAx \Rightarrow^* vwx$



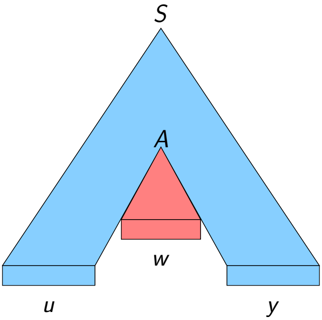
Pumping Lemma

This gives three nested sub-syntax trees, which can be reassembled.



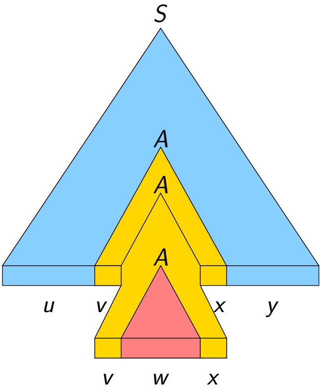
Pumping Lemma

By removing the middle subtree, we get a syntax tree for uwy. Thus, $uwy \in L(G)$.

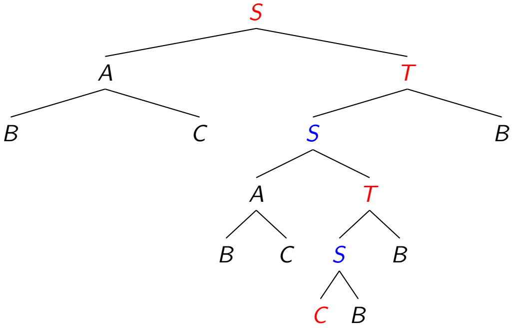


Pumping Lemma

By doubling the middle subtree, we get a syntax tree for uv^2wx^2y. Thus, $uv^2wx^2y \in L(G)$.



Using the concrete example on slide 220:



We get: $u = bc$, $v = bc$, $w = cb$, $x = b$, $y = b$

Pumping Lemma

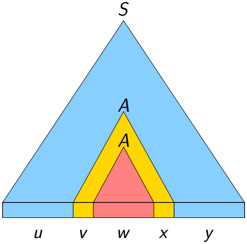
Additionally, the following properties can be required for v, w, and x:

$|vwx| \leq n = 2^k$:

We can assume that we have selected the deepest double occurrence of a variable, i.e., the double occurrence with the greatest depth.

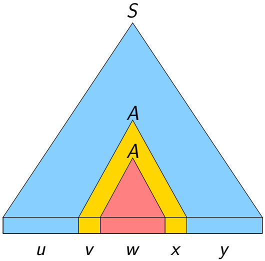
This can be achieved by following a path of maximal length from bottom to top until a double occurrence is found.

Therefore, the distance from the upper A to the leaf level is at most k, and the binary tree below it has at most 2^k leaves.



$|vx| \geq 1$:

Let B, C be the two children of the upper A. Then the lower A either originates from B or C. The other variable must, since the grammar is in Chomsky Normal Form, derive a non-empty word.
And this word is a subword of v or x.



We have thus proven the following theorem:

Theorem (Pumping Lemma, *uvwxy*-Theorem)

Let L be a context-free language. Then there exists a number n such that all words $z \in L$ with $|z| \geq n$ can be decomposed as $z = uvwxy$, such that the following properties hold:

1. $|vx| \geq 1$,
2. $|vwx| \leq n$,
3. for all $i \geq 0$, $uv^iwx^iy \in L$.

Here, $n = 2^k$ is derived from the number k of variables in a context-free grammar in CNF for L.

Pumping Lemma

Application of the Pumping Lemma:

We will show that the language $L = \{a^m b^m c^m \mid m \geq 1\}$ is not context-free.

1. We assume an arbitrary number n.
2. We choose a word $z \in L$ with $|z| \geq n$. In this case, $z = a^n b^n c^n$ is suitable.
3. Now, we consider all possible decompositions of $z = uvwxy$ with the restrictions $|vx| \geq 1$ and $|vwx| \leq n$.

   Since $|vwx| \leq n$, it follows that vx cannot consist of a's, b's, and c's because it cannot span across the entire b-block.
4. We choose $i = 2$ for all these possible decompositions and consider uv^2wx^2y. Due to the above reasoning, one or two alphabet symbols have been pumped, but at least one has not.

   Thus, it is clear that uv^2wx^2y cannot be in L, because every word in L has an equal number of a's, b's, and c's.

Pumping Lemma

One can also show that the following languages are not context-free:

$$\begin{aligned} L_1 &= \{a^p \mid p \text{ is prime}\} \\ L_2 &= \{a^n \mid n \text{ is a perfect square}\} \\ L_3 &= \{a^{2^n} \mid n \geq 0\} \end{aligned}$$

The languages L_1, L_2, L_3 are all unary, meaning they are languages over a one-letter alphabet: $L_1, L_2, L_3 \subseteq \Sigma^*$ with $|\Sigma| = 1$.

For unary languages, the following theorem holds (without proof).

Theorem (Unary Context-Free Languages)

Every context-free language over a one-letter alphabet is already regular.

Closure Properties

Closure

Context-free languages are closed under:

- Union (L_1, L_2 context-free $\Rightarrow L_1 \cup L_2$ context-free)
- Product/Concatenation (L_1, L_2 context-free $\Rightarrow L_1 L_2$ context-free)
- Star operation (L context-free $\Rightarrow L^*$ context-free)

Context-free languages are not closed under:

- Intersection
- Complement

Closure Properties

Closure under Union

If L_1 and L_2 are context-free languages, then $L_1 \cup L_2$ is also context-free.

Reasoning: Let $G_1 = (V_1, \Sigma, P_1, S_1)$ and $G_2 = (V_2, \Sigma, P_2, S_2)$ be context-free grammars.

Without loss of generality, assume that $V_1 \cap V_2 = \emptyset$.

Let $S \notin V_1 \cup V_2$.

Then, $G = (V_1 \cup V_2 \cup \{S\}, \Sigma, P_1 \cup P_2 \cup \{S \to S_1, S \to S_2\}, S)$ is a context-free grammar with $L(G) = L(G_1) \cup L(G_2)$.

Closure Properties

Closure under Product/Concatenation

If L_1 and L_2 are context-free languages, then $L_1 L_2$ is also context-free.

Reasoning: Let

$$G_1 = (V_1, \Sigma, P_1, S_1), \quad G_2 = (V_2, \Sigma, P_2, S_2)$$

be context-free grammars. Without loss of generality, assume that $V_1 \cap V_2 = \emptyset$.

Let $S \notin V_1 \cup V_2$.

Then,

$$G = (V_1 \cup V_2 \cup \{S\}, \Sigma, P_1 \cup P_2 \cup \{S \to S_1 S_2\}, S)$$

is a context-free grammar with $L(G) = L(G_1)L(G_2)$.

Closure under the Star Operation

If L is a context-free language, then L^* is also context-free.

Reasoning: Let

$$G_1 = (V_1, \Sigma, P_1, S_1)$$

be a context-free grammar.

Let $S \notin V_1$.

Then,

$$G = (V_1 \cup \{S\}, \Sigma, P_1 \cup \{S \to \varepsilon, S \to S_1 S\}, S)$$

is a context-free grammar with $L(G) = L(G_1)^*$.

No Closure under Intersection

There are context-free languages L_1 and L_2 such that $L_1 \cap L_2$ is not context-free.

Counterexample: The languages

$$\begin{aligned} L_1 &= \{a^j b^k c^k \mid j \geq 0, k \geq 0\} \\ L_2 &= \{a^k b^k c^j \mid j \geq 0, k \geq 0\} \end{aligned}$$

are both context-free (for example, L_1 is generated by a grammar with productions $S \to aS \mid A$, $A \to \varepsilon \mid bAc$).

However, their intersection is

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

and this language is not context-free, as shown using the Pumping Lemma.

Closure Properties

No Closure under Complement

There exists a context-free language L such that $\overline{L} = \Sigma^* \setminus L$ is not context-free.

Reasoning:

Suppose context-free languages were closed under complement, and let L_1 and L_2 be context-free. By De Morgan's law,

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

which would imply that $L_1 \cap L_2$ is context-free.

This contradicts the counterexample on the previous slide.

The CYK Algorithm

We already know a method for solving the word problem for G, where G can be a Type-1, Type-2, or Type-3 grammar (Slide 37).
Essentially: listing all words up to a certain length.

However, since this method can have exponential runtime (in the length of the word), we consider here a more efficient method for context-free grammars: the CYK Algorithm (developed by Cocke, Younger, Kasami).

Prerequisite: The grammar is in Chomsky Normal Form, so all productions have the form $A \to a$ or $A \to BC$.

The CYK Algorithm

Idea: Given a word $x \in \Sigma^*$. We want to determine which variables it can be derived from.

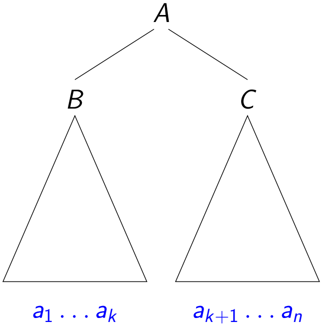
- **Possibility 1:** $x = a \in \Sigma$, i.e., x consists of a single alphabet symbol.

  In this case, x can only be derived from variables A for which there is a production $A \to a$.

- **Possibility 2:** $x = a_1 \cdots a_n$ with $n \geq 2$.

  In this case: First, a production $A \to BC$ must be applied, then one part $a_1 \cdots a_k$ of the word must be derived from B and the other part $a_{k+1} \cdots a_n$ from C $(1 \leq k < n)$.

Possibility 2 can be schematically represented as follows:



However, it is not clear where the word x should be split, i.e., what the position k is!

Therefore: Try all possible k's. This means:

Given a word $x = a_1 \cdots a_n$.

For all k with $1 \leq k < n$, do the following:

- Determine the set V_1 of all variables from which $a_1 \cdots a_k$ can be derived.
- Determine the set V_2 of all variables from which $a_{k+1} \cdots a_n$ can be derived.
- Check if there are variables A, B, C such that $(A \to BC) \in P$, $B \in V_1$ and $C \in V_2$.
  In this case, x can be derived from A.

The CYK Algorithm

To avoid unnecessary work, we use the method of dynamic programming, i.e.:

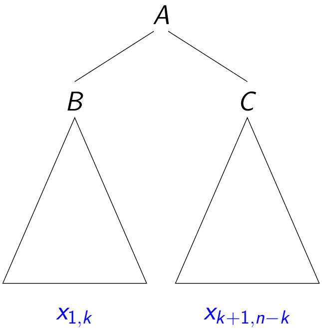
- First, calculate all the variables from which substrings of length 1 can be derived.
- Then, calculate all the variables from which substrings of length 2 can be derived.
- $\vdots$
- Finally, calculate all the variables from which x can be derived. If the start variable S is among these variables, then x is in the language generated by the grammar.

The CYK Algorithm

Notation: We denote by $x_{i,j}$ the substring of x that starts at position i and has length j.

$$x = a_1 \cdots a_n \quad \Longrightarrow \quad x_{i,j} = a_i \cdots a_{i+j-1}$$

Thus, the above diagram looks as follows:



The CYK Algorithm

We denote by $T_{i,j}$ the set of all variables from which $x_{i,j}$ can be derived:

$$T_{i,j} = \{A \in V \mid A \Rightarrow_G^* x_{i,j}\}$$

Then the following holds:

- $T_{i,1} = \{A \in V \mid (A \to a_i) \in P\}$.
- For $j \geq 2$, $T_{i,j}$ can be determined from the sets $T_{\ell,k}$ with $k < j$ as follows:

$$T_{i,j} = \{A \mid \exists (A \to BC) \in P \ \exists 1 \leq k < j : B \in T_{i,k} \text{ and } C \in T_{i+k,j-k}\}$$

Practical execution of the CYK algorithm:

We enter the variable sets $T_{i,j}$ in the following table:

	a_1	a_2	$\dots$		a_{n-1}	a_n
$j = 1$	$T_{1,1}$	$T_{2,1}$	$\dots$	$\dots$	$T_{n-1,1}$	$T_{n,1}$
$j = 2$	$T_{1,2}$	$T_{2,2}$	$\dots$	$\dots$	$T_{n-1,2}$	
	$\dots$	$\dots$	$\dots$	$\dots$		
$\dots$	$\dots$	$\dots$	$\dots$			
$j = n-1$	$T_{1,n-1}$	$T_{2,n-1}$				
$j = n$	$T_{1,n}$					

The following illustrates which variable set derives which substring:

	a_1	a_2	a_3	a_4	a_5	a_6
$j = 1$	$T_{1,1}$	$T_{2,1}$	$T_{3,1}$	$T_{4,1}$	$T_{5,1}$	$T_{6,1}$
$j = 2$	$T_{1,2}$	$T_{2,2}$	$T_{3,2}$	$T_{4,2}$	$T_{5,2}$	
$j = 3$	$T_{1,3}$	$T_{2,3}$	$T_{3,3}$	$T_{4,3}$		
$j = 4$	$T_{1,4}$	$T_{2,4}$	$T_{3,4}$			
$j = 5$	$T_{1,5}$	$T_{2,5}$				
$j = 6$	$T_{1,6}$					

The CYK Algorithm

	a_1	a_2	a_3	a_4	a_5	a_6
$j = 1$						$T_{6,1}$
$j = 2$						
$j = 3$						
$j = 4$						
$j = 5$	$T_{1,5}$					
$j = 6$	$T_{1,6}$					

$x = a_1 a_2 a_3 a_4 a_5 \mid a_6$

$(A \to BC) \in P$,

$B \in T_{1,5},\ C \in T_{6,1} \Rightarrow A \in T_{1,6}$

The CYK Algorithm

	a_1	a_2	a_3	a_4	a_5	a_6
$j = 1$						
$j = 2$					$T_{5,2}$	
$j = 3$						
$j = 4$	$T_{1,4}$					
$j = 5$						
$j = 6$	$T_{1,6}$					

$x = a_1 a_2 a_3 a_4 \mid a_5 a_6$

$(A \to BC) \in P$,
$B \in T_{1,4}$, $C \in T_{5,2} \Rightarrow A \in T_{1,6}$

The CYK Algorithm

	a_1	a_2	a_3	a_4	a_5	a_6
$j = 1$						
$j = 2$						
$j = 3$	$T_{1,3}$			$T_{4,3}$		
$j = 4$						
$j = 5$						
$j = 6$	$T_{1,6}$					

$x = a_1 a_2 a_3 \mid a_4 a_5 a_6$

$(A \to BC) \in P$,

$B \in T_{1,3},\ C \in T_{4,3} \Rightarrow A \in T_{1,6}$

The CYK Algorithm

	a_1	a_2	a_3	a_4	a_5	a_6
$j=1$						
$j=2$	$T_{1,2}$					
$j=3$						
$j=4$			$T_{3,4}$			
$j=5$						
$j=6$	$T_{1,6}$					

$x = a_1 a_2 \,|\, a_3 a_4 a_5 a_6$

$(A \to BC) \in P$,

$B \in T_{1,2}$, $C \in T_{3,4} \Rightarrow A \in T_{1,6}$

The CYK Algorithm

	a_1	a_2	a_3	a_4	a_5	a_6
$j = 1$	$T_{1,1}$					
$j = 2$						
$j = 3$						
$j = 4$						
$j = 5$		$T_{2,5}$				
$j = 6$	$T_{1,6}$					

$x = a_1 \mid a_2 a_3 a_4 a_5 a_6$

$(A \to BC) \in P$,
$B \in T_{1,1}$, $C \in T_{2,5} \Rightarrow A \in T_{1,6}$

Example 1: Consider a grammar for the language $L = \{a^k b^k c^j \mid k, j > 0\}$ with the following productions:

$$\begin{array}{rcl} S & \to & AB \\ A & \to & ab \mid aAb \\ B & \to & c \mid cB \end{array}$$

We show using the CYK algorithm that $aaabbbcc \in L$ holds.

First, we need to transform the grammar into Chomsky Normal Form.

This results in the grammar on the next slide.

The CYK Algorithm

$$S \to AB \qquad\qquad A_a \to a$$
$$A \to A_a A_b \mid A_a C \qquad\qquad A_b \to b$$
$$C \to A A_b \qquad\qquad A_c \to c$$
$$B \to c \mid A_c B$$

	a	a	a	b	b	b	c	c
$j = 1$								
$j = 2$								
$j = 3$								
$j = 4$								
$j = 5$								
$j = 6$								
$j = 7$								
$j = 8$								

The CYK Algorithm

$$S \to AB \qquad A_a \to a$$
$$A \to A_a A_b \mid A_a C \qquad A_b \to b$$
$$C \to A A_b \qquad A_c \to c$$
$$B \to c \mid A_c B$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$								
$j=3$								
$j=4$								
$j=5$								
$j=6$								
$j=7$								
$j=8$								

The CYK Algorithm

$$S \to AB \qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad A_b \to b$$
$$C \to AA_b \qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$							
$j=3$								
$j=4$								
$j=5$								
$j=6$								
$j=7$								
$j=8$								

$$S \to AB \qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad A_b \to b$$
$$C \to AA_b \qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$						
$j=3$								
$j=4$								
$j=5$								
$j=6$								
$j=7$								
$j=8$								

$$S \to AB \qquad\qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad\qquad A_b \to b$$
$$C \to AA_b \qquad\qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A					
$j=3$								
$j=4$								
$j=5$								
$j=6$								
$j=7$								
$j=8$								

The CYK Algorithm

$$S \to AB \qquad\qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad\qquad A_b \to b$$
$$C \to AA_b \qquad\qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$				
$j=3$								
$j=4$								
$j=5$								
$j=6$								
$j=7$								
$j=8$								

$$S \to AB \qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad A_b \to b$$
$$C \to AA_b \qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$			
$j=3$								
$j=4$								
$j=5$								
$j=6$								
$j=7$								
$j=8$								

$$S \to AB \qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad A_b \to b$$
$$C \to AA_b \qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$		
$j=3$								
$j=4$								
$j=5$								
$j=6$								
$j=7$								
$j=8$								

$$S \to AB \qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad A_b \to b$$
$$C \to AA_b \qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$								
$j=4$								
$j=5$								
$j=6$								
$j=7$								
$j=8$								

The CYK Algorithm

$$S \to AB \qquad A_a \to a$$
$$A \to A_a A_b \mid A_a C \qquad A_b \to b$$
$$C \to A A_b \qquad A_c \to c$$
$$B \to c \mid A_c B$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$							
$j=4$								
$j=5$								
$j=6$								
$j=7$								
$j=8$								

The CYK Algorithm

$$S \to AB \qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad A_b \to b$$
$$C \to AA_b \qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$							
$j=4$								
$j=5$								
$j=6$								
$j=7$								
$j=8$								

$$S \to AB \qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad A_b \to b$$
$$C \to AA_b \qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$						
$j=4$								
$j=5$								
$j=6$								
$j=7$								
$j=8$								

$$S \to AB \qquad\qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad\qquad A_b \to b$$
$$C \to AA_b \qquad\qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$						
$j=4$								
$j=5$								
$j=6$								
$j=7$								
$j=8$								

$$S \to AB \qquad\qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad\qquad A_b \to b$$
$$C \to AA_b \qquad\qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C					
$j=4$								
$j=5$								
$j=6$								
$j=7$								
$j=8$								

$$
\begin{aligned}
S &\to AB & \qquad A_a &\to a \\
A &\to A_a A_b \mid A_a C & A_b &\to b \\
C &\to A A_b & A_c &\to c \\
B &\to c \mid A_c B & &
\end{aligned}
$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C					
$j=4$								
$j=5$								
$j=6$								
$j=7$								
$j=8$								

$$S \to AB \qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad A_b \to b$$
$$C \to AA_b \qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$				
$j=4$								
$j=5$								
$j=6$								
$j=7$								
$j=8$								

$$S \to AB \qquad\qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad\qquad A_b \to b$$
$$C \to AA_b \qquad\qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$				
$j=4$								
$j=5$								
$j=6$								
$j=7$								
$j=8$								

$$S \to AB \qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad A_b \to b$$
$$C \to AA_b \qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$			
$j=4$								
$j=5$								
$j=6$								
$j=7$								
$j=8$								

$$
\begin{aligned}
S &\to AB & A_a &\to a \\
A &\to A_aA_b \mid A_aC & A_b &\to b \\
C &\to AA_b & A_c &\to c \\
B &\to c \mid A_cB & &
\end{aligned}
$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$			
$j=4$								
$j=5$								
$j=6$								
$j=7$								
$j=8$								

The CYK Algorithm

$$S \to AB \qquad\qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad\qquad A_b \to b$$
$$C \to AA_b \qquad\qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$								
$j=5$								
$j=6$								
$j=7$								
$j=8$								

$$S \to AB \qquad\qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad\qquad A_b \to b$$
$$C \to AA_b \qquad\qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$								
$j=5$								
$j=6$								
$j=7$								
$j=8$								

The CYK Algorithm

$$S \to AB \qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad A_b \to b$$
$$C \to AA_b \qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$							
$j=5$								
$j=6$								
$j=7$								
$j=8$								

The CYK Algorithm

$$S \to AB \qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad A_b \to b$$
$$C \to AA_b \qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$							
$j=5$								
$j=6$								
$j=7$								
$j=8$								

The CYK Algorithm

$$S \to AB \qquad\qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad\qquad A_b \to b$$
$$C \to AA_b \qquad\qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$							
$j=5$								
$j=6$								
$j=7$								
$j=8$								

The CYK Algorithm

$$S \to AB \qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad A_b \to b$$
$$C \to AA_b \qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	$\emptyset$						
$j=5$								
$j=6$								
$j=7$								
$j=8$								

The CYK Algorithm

$$S \to AB \qquad A_a \to a$$
$$A \to A_a A_b \mid A_a C \qquad A_b \to b$$
$$C \to A A_b \qquad A_c \to c$$
$$B \to c \mid A_c B$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	$\emptyset$						
$j=5$								
$j=6$								
$j=7$								
$j=8$								

$$S \to AB \qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad A_b \to b$$
$$C \to AA_b \qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A						
$j=5$								
$j=6$								
$j=7$								
$j=8$								

The CYK Algorithm

$$S \to AB \qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad A_b \to b$$
$$C \to AA_b \qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$					
$j=5$								
$j=6$								
$j=7$								
$j=8$								

$$S \to AB \qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad A_b \to b$$
$$C \to AA_b \qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$					
$j=5$								
$j=6$								
$j=7$								
$j=8$								

$$S \to AB \qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad A_b \to b$$
$$C \to AA_b \qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$					
$j=5$								
$j=6$								
$j=7$								
$j=8$								

$$S \to AB \qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad A_b \to b$$
$$C \to AA_b \qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$	$\emptyset$				
$j=5$								
$j=6$								
$j=7$								
$j=8$								

$$S \to AB \qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad A_b \to b$$
$$C \to AA_b \qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$	$\emptyset$				
$j=5$								
$j=6$								
$j=7$								
$j=8$								

$$S \to AB \qquad A_a \to a$$
$$A \to A_a A_b \mid A_a C \qquad A_b \to b$$
$$C \to A A_b \qquad A_c \to c$$
$$B \to c \mid A_c B$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$	$\emptyset$				
$j=5$								
$j=6$								
$j=7$								
$j=8$								

The CYK Algorithm

$$S \to AB \qquad\qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad\qquad A_b \to b$$
$$C \to AA_b \qquad\qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$			
$j=5$								
$j=6$								
$j=7$								
$j=8$								

The CYK Algorithm

$$S \to AB \qquad A_a \to a$$
$$A \to A_a A_b \mid A_a C \qquad A_b \to b$$
$$C \to A A_b \qquad A_c \to c$$
$$B \to c \mid A_c B$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$			
$j=5$								
$j=6$								
$j=7$								
$j=8$								

The CYK Algorithm

$$
\begin{aligned}
S &\to AB & A_a &\to a\\
A &\to A_aA_b \mid A_aC & A_b &\to b\\
C &\to AA_b & A_c &\to c\\
B &\to c \mid A_cB
\end{aligned}
$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$			
$j=5$								
$j=6$								
$j=7$								
$j=8$								

$$S \to AB \qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad A_b \to b$$
$$C \to AA_b \qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$			
$j=5$	$\emptyset$							
$j=6$								
$j=7$								
$j=8$								

The CYK Algorithm

$$S \to AB \qquad\qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad\qquad A_b \to b$$
$$C \to AA_b \qquad\qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$			
$j=5$	$\emptyset$							
$j=6$								
$j=7$								
$j=8$								

$$S \to AB \qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad A_b \to b$$
$$C \to AA_b \qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$			
$j=5$	$\emptyset$							
$j=6$								
$j=7$								
$j=8$								

$$S \to AB \qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad A_b \to b$$
$$C \to AA_b \qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$			
$j=5$	$\emptyset$							
$j=6$								
$j=7$								
$j=8$								

The CYK Algorithm

$$S \to AB \qquad\qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad\qquad A_b \to b$$
$$C \to AA_b \qquad\qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$			
$j=5$	$\emptyset$	C						
$j=6$								
$j=7$								
$j=8$								

$$S \to AB \qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad A_b \to b$$
$$C \to AA_b \qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$			
$j=5$	$\emptyset$	C						
$j=6$								
$j=7$								
$j=8$								

The CYK Algorithm

$$S \to AB \qquad\qquad A_a \to a$$
$$A \to A_a A_b \mid A_a C \qquad\qquad A_b \to b$$
$$C \to AA_b \qquad\qquad A_c \to c$$
$$B \to c \mid A_c B$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$			
$j=5$	$\emptyset$	C						
$j=6$								
$j=7$								
$j=8$								

The CYK Algorithm

$$S \to AB \qquad\qquad A_a \to a$$

$$A \to A_a A_b \mid A_a C \qquad\qquad A_b \to b$$

$$C \to A A_b \qquad\qquad A_c \to c$$

$$B \to c \mid A_c B$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$			
$j=5$	$\emptyset$	C						
$j=6$								
$j=7$								
$j=8$								

$$S \to AB \qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad A_b \to b$$
$$C \to AA_b \qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$			
$j=5$	$\emptyset$	C	$\emptyset$					
$j=6$								
$j=7$								
$j=8$								

$$
\begin{array}{ll}
S \to AB & A_a \to a \\
A \to A_aA_b \mid A_aC & A_b \to b \\
C \to AA_b & A_c \to c \\
B \to c \mid A_cB &
\end{array}
$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$			
$j=5$	$\emptyset$	C	$\emptyset$					
$j=6$								
$j=7$								
$j=8$								

The CYK Algorithm

$$S \to AB \qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad A_b \to b$$
$$C \to AA_b \qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$			
$j=5$	$\emptyset$	C	$\emptyset$					
$j=6$								
$j=7$								
$j=8$								

$$S \to AB \qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad A_b \to b$$
$$C \to AA_b \qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$			
$j=5$	$\emptyset$	C	$\emptyset$					
$j=6$								
$j=7$								
$j=8$								

The CYK Algorithm

$$S \to AB \qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad A_b \to b$$
$$C \to AA_b \qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$			
$j=5$	$\emptyset$	C	$\emptyset$	$\emptyset$				
$j=6$								
$j=7$								
$j=8$								

The CYK Algorithm

$$S \to AB \qquad\qquad A_a \to a$$
$$A \to A_a A_b \mid A_a C \qquad\qquad A_b \to b$$
$$C \to A A_b \qquad\qquad A_c \to c$$
$$B \to c \mid A_c B$$

	a	a	a	b	b	b	c	c
$j = 1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j = 2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j = 3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j = 4$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$			
$j = 5$	$\emptyset$	C	$\emptyset$	$\emptyset$				
$j = 6$								
$j = 7$								
$j = 8$								

$$S \to AB \qquad\qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad\qquad A_b \to b$$
$$C \to AA_b \qquad\qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$			
$j=5$	$\emptyset$	C	$\emptyset$	$\emptyset$				
$j=6$								
$j=7$								
$j=8$								

The CYK Algorithm

$$S \to AB \qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad A_b \to b$$
$$C \to AA_b \qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$			
$j=5$	$\emptyset$	C	$\emptyset$	$\emptyset$				
$j=6$								
$j=7$								
$j=8$								

$$S \to AB \qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad A_b \to b$$
$$C \to AA_b \qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$			
$j=5$	$\emptyset$	C	$\emptyset$	$\emptyset$				
$j=6$	$\emptyset$							
$j=7$								
$j=8$								

$$S \to AB \qquad\qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad\qquad A_b \to b$$
$$C \to AA_b \qquad\qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$			
$j=5$	$\emptyset$	C	$\emptyset$	$\emptyset$				
$j=6$	$\emptyset$							
$j=7$								
$j=8$								

$$S \to AB \qquad A_a \to a$$
$$A \to A_a A_b \mid A_a C \qquad A_b \to b$$
$$C \to A A_b \qquad A_c \to c$$
$$B \to c \mid A_c B$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$			
$j=5$	$\emptyset$	C	$\emptyset$	$\emptyset$				
$j=6$	$\emptyset$							
$j=7$								
$j=8$								

The CYK Algorithm

$$S \to AB \qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad A_b \to b$$
$$C \to AA_b \qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$			
$j=5$	$\emptyset$	C	$\emptyset$	$\emptyset$				
$j=6$	$\emptyset$							
$j=7$								
$j=8$								

$$S \to AB \qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad A_b \to b$$
$$C \to AA_b \qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$			
$j=5$	$\emptyset$	C	$\emptyset$	$\emptyset$				
$j=6$	A							
$j=7$								
$j=8$								

The CYK Algorithm

$$S \to AB \qquad\qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad\qquad A_b \to b$$
$$C \to AA_b \qquad\qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j = 1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j = 2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j = 3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j = 4$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$			
$j = 5$	$\emptyset$	C	$\emptyset$	$\emptyset$				
$j = 6$	A	$\emptyset$						
$j = 7$								
$j = 8$								

The CYK Algorithm

$$
\begin{aligned}
S &\to AB & A_a &\to a \\
A &\to A_aA_b \mid A_aC & A_b &\to b \\
C &\to AA_b & A_c &\to c \\
B &\to c \mid A_cB & &
\end{aligned}
$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$			
$j=5$	$\emptyset$	C	$\emptyset$	$\emptyset$				
$j=6$	A	$\emptyset$						
$j=7$								
$j=8$								

$$
\begin{aligned}
S &\to AB & A_a &\to a \\
A &\to A_aA_b \mid A_aC & A_b &\to b \\
C &\to AA_b & A_c &\to c \\
B &\to c \mid A_cB & &
\end{aligned}
$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$			
$j=5$	$\emptyset$	C	$\emptyset$	$\emptyset$				
$j=6$	A	$\emptyset$						
$j=7$								
$j=8$								

$$
\begin{aligned}
S &\to AB & A_a &\to a \\
A &\to A_aA_b \mid A_aC & A_b &\to b \\
C &\to AA_b & A_c &\to c \\
B &\to c \mid A_cB & &
\end{aligned}
$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$			
$j=5$	$\emptyset$	C	$\emptyset$	$\emptyset$				
$j=6$	A	$\emptyset$						
$j=7$								
$j=8$								

$$
\begin{aligned}
S &\to AB & A_a &\to a \\
A &\to A_aA_b \mid A_aC & A_b &\to b \\
C &\to AA_b & A_c &\to c \\
B &\to c \mid A_cB & &
\end{aligned}
$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$			
$j=5$	$\emptyset$	C	$\emptyset$	$\emptyset$				
$j=6$	A	$\emptyset$						
$j=7$								
$j=8$								

$$S \to AB \qquad\qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad\qquad A_b \to b$$
$$C \to AA_b \qquad\qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$			
$j=5$	$\emptyset$	C	$\emptyset$	$\emptyset$				
$j=6$	A	$\emptyset$	$\emptyset$					
$j=7$								
$j=8$								

The CYK Algorithm

$$S \to AB \qquad\qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad\qquad A_b \to b$$
$$C \to AA_b \qquad\qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$			
$j=5$	$\emptyset$	C	$\emptyset$	$\emptyset$				
$j=6$	A	$\emptyset$	$\emptyset$					
$j=7$								
$j=8$								

$$S \to AB \qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad A_b \to b$$
$$C \to AA_b \qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$			
$j=5$	$\emptyset$	C	$\emptyset$	$\emptyset$				
$j=6$	A	$\emptyset$	$\emptyset$					
$j=7$								
$j=8$								

The CYK Algorithm

$$S \to AB \qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad A_b \to b$$
$$C \to AA_b \qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$			
$j=5$	$\emptyset$	C	$\emptyset$	$\emptyset$				
$j=6$	A	$\emptyset$	$\emptyset$					
$j=7$								
$j=8$								

$$S \to AB \qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad A_b \to b$$
$$C \to AA_b \qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$			
$j=5$	$\emptyset$	C	$\emptyset$	$\emptyset$				
$j=6$	A	$\emptyset$	$\emptyset$					
$j=7$								
$j=8$								

The CYK Algorithm

$$S \to AB \qquad\qquad A_a \to a$$
$$A \to A_a A_b \mid A_a C \qquad\qquad A_b \to b$$
$$C \to A A_b \qquad\qquad A_c \to c$$
$$B \to c \mid A_c B$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$			
$j=5$	$\emptyset$	C	$\emptyset$	$\emptyset$				
$j=6$	A	$\emptyset$	$\emptyset$					
$j=7$	S							
$j=8$								

The CYK Algorithm

$$S \to AB \qquad\qquad A_a \to a$$
$$A \to A_a A_b \mid A_a C \qquad\qquad A_b \to b$$
$$C \to A A_b \qquad\qquad A_c \to c$$
$$B \to c \mid A_c B$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$			
$j=5$	$\emptyset$	C	$\emptyset$	$\emptyset$				
$j=6$	A	$\emptyset$	$\emptyset$					
$j=7$	S							
$j=8$								

The CYK Algorithm

$$S \to AB \qquad\qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad\qquad A_b \to b$$
$$C \to AA_b \qquad\qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$			
$j=5$	$\emptyset$	C	$\emptyset$	$\emptyset$				
$j=6$	A	$\emptyset$	$\emptyset$					
$j=7$	S							
$j=8$								

The CYK Algorithm

$$S \to AB \qquad A_a \to a$$
$$A \to A_a A_b \mid A_a C \qquad A_b \to b$$
$$C \to A A_b \qquad A_c \to c$$
$$B \to c \mid A_c B$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$			
$j=5$	$\emptyset$	C	$\emptyset$	$\emptyset$				
$j=6$	A	$\emptyset$	$\emptyset$					
$j=7$	S							
$j=8$								

The CYK Algorithm

$$S \to AB \qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad A_b \to b$$
$$C \to AA_b \qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$			
$j=5$	$\emptyset$	C	$\emptyset$	$\emptyset$				
$j=6$	A	$\emptyset$	$\emptyset$					
$j=7$	S							
$j=8$								

$$S \to AB \qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad A_b \to b$$
$$C \to AA_b \qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$			
$j=5$	$\emptyset$	C	$\emptyset$	$\emptyset$				
$j=6$	A	$\emptyset$	$\emptyset$					
$j=7$	S							
$j=8$								

The CYK Algorithm

$$S \to AB \qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad A_b \to b$$
$$C \to AA_b \qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$			
$j=5$	$\emptyset$	C	$\emptyset$	$\emptyset$				
$j=6$	A	$\emptyset$	$\emptyset$					
$j=7$	S	$\emptyset$						
$j=8$								

The CYK Algorithm

$$S \to AB \qquad\qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad\qquad A_b \to b$$
$$C \to AA_b \qquad\qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$			
$j=5$	$\emptyset$	C	$\emptyset$	$\emptyset$				
$j=6$	A	$\emptyset$	$\emptyset$					
$j=7$	S	$\emptyset$						
$j=8$								

The CYK Algorithm

$$S \to AB \qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad A_b \to b$$
$$C \to AA_b \qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$			
$j=5$	$\emptyset$	C	$\emptyset$	$\emptyset$				
$j=6$	A	$\emptyset$	$\emptyset$					
$j=7$	S	$\emptyset$						
$j=8$								

$$S \to AB \qquad A_a \to a$$
$$A \to A_a A_b \mid A_a C \qquad A_b \to b$$
$$C \to A A_b \qquad A_c \to c$$
$$B \to c \mid A_c B$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$			
$j=5$	$\emptyset$	C	$\emptyset$	$\emptyset$				
$j=6$	A	$\emptyset$	$\emptyset$					
$j=7$	S	$\emptyset$						
$j=8$								

The CYK Algorithm

$$S \to AB \qquad\qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad\qquad A_b \to b$$
$$C \to AA_b \qquad\qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$			
$j=5$	$\emptyset$	C	$\emptyset$	$\emptyset$				
$j=6$	A	$\emptyset$	$\emptyset$					
$j=7$	S	$\emptyset$						
$j=8$								

The CYK Algorithm

$$S \to AB \qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad A_b \to b$$
$$C \to AA_b \qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$			
$j=5$	$\emptyset$	C	$\emptyset$	$\emptyset$				
$j=6$	A	$\emptyset$	$\emptyset$					
$j=7$	S	$\emptyset$						
$j=8$								

The CYK Algorithm

$$S \to AB \qquad A_a \to a$$
$$A \to A_a A_b \mid A_a C \qquad A_b \to b$$
$$C \to A A_b \qquad A_c \to c$$
$$B \to c \mid A_c B$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$			
$j=5$	$\emptyset$	C	$\emptyset$	$\emptyset$				
$j=6$	A	$\emptyset$	$\emptyset$					
$j=7$	S	$\emptyset$						
$j=8$	$\emptyset$							

The CYK Algorithm

$$S \to AB \qquad A_a \to a$$
$$A \to A_a A_b \mid A_a C \qquad A_b \to b$$
$$C \to A A_b \qquad A_c \to c$$
$$B \to c \mid A_c B$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$			
$j=5$	$\emptyset$	C	$\emptyset$	$\emptyset$				
$j=6$	A	$\emptyset$	$\emptyset$					
$j=7$	S	$\emptyset$						
$j=8$	S							

The CYK Algorithm

$$S \to AB \qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad A_b \to b$$
$$C \to AA_b \qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$			
$j=5$	$\emptyset$	C	$\emptyset$	$\emptyset$				
$j=6$	A	$\emptyset$	$\emptyset$					
$j=7$	S	$\emptyset$						
$j=8$	S							

The CYK Algorithm

$$S \to AB \qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad A_b \to b$$
$$C \to AA_b \qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$			
$j=5$	$\emptyset$	C	$\emptyset$	$\emptyset$				
$j=6$	A	$\emptyset$	$\emptyset$					
$j=7$	S	$\emptyset$						
$j=8$	S							

The CYK Algorithm

$$S \to AB \qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad A_b \to b$$
$$C \to AA_b \qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$			
$j=5$	$\emptyset$	C	$\emptyset$	$\emptyset$				
$j=6$	A	$\emptyset$	$\emptyset$					
$j=7$	S	$\emptyset$						
$j=8$	S							

The CYK Algorithm

$$S \to AB \qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad A_b \to b$$
$$C \to AA_b \qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$			
$j=5$	$\emptyset$	C	$\emptyset$	$\emptyset$				
$j=6$	A	$\emptyset$	$\emptyset$					
$j=7$	S	$\emptyset$						
$j=8$	S							

The CYK Algorithm

$$S \to AB \qquad\qquad A_a \to a$$
$$A \to A_aA_b \mid A_aC \qquad\qquad A_b \to b$$
$$C \to AA_b \qquad\qquad A_c \to c$$
$$B \to c \mid A_cB$$

	a	a	a	b	b	b	c	c
$j=1$	A_a	A_a	A_a	A_b	A_b	A_b	B, A_c	B, A_c
$j=2$	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
$j=3$	$\emptyset$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$		
$j=4$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$			
$j=5$	$\emptyset$	C	$\emptyset$	$\emptyset$				
$j=6$	A	$\emptyset$	$\emptyset$					
$j=7$	S	$\emptyset$						
$j=8$	S							

Example 2: Consider a grammar with the following productions:

$$\begin{aligned} S &\to AD \mid FG \\ D &\to SE \mid BC \\ E &\to BC \\ F &\to AF \mid a \\ G &\to BG \mid CG \mid b \\ A &\to a \\ B &\to b \\ C &\to c \end{aligned}$$

Question: Let $x = aabcbc$. Does $x \in L$?

Here is the table resulting from the CYK algorithm:
(You should verify this):

	a	a	b	c	b	c
$j=1$	A,F	A,F	B,G	C	B,G	C
$j=2$	F	S	D,E	G	D,E	
$j=3$	S	S	G			
$j=4$		S				
$j=5$	S	D				
$j=6$	S					

The CYK Algorithm

Complexity of the CYK Algorithm

Let $n = |x|$ be the length of the word being analyzed. The size of the grammar is considered constant. Then:

- $O(n^2)$ table entries need to be filled.
- For filling each table entry, up to $O(n)$ other entries must be considered.

  (For $T_{1,n}$, for example, the entries $T_{1,n-1}$, $T_{n,1}$ and $T_{1,n-2}$, $T_{n-1,2}$ and ... and $T_{1,1}$, $T_{2,n-1}$ must be considered. In total, $n-1$ pairs of entries.)

Hence, the overall time complexity is: $O(n^3)$.

The time complexity is still polynomial, but it is not well-suited for parsing large programs.

Pushdown Automata

What is a suitable automaton model for context-free languages?

Analogous to regular languages, we seek an automaton model for context-free languages.

Answer: Pushdown automata, i.e., automata equipped with an additional stack.

Utility of such an automaton model

Some constructions and procedures can be performed more effectively using the automaton model (instead of grammars).

- Word problem: We will discover that the word problem can, under certain circumstances, be solved more efficiently than in $O(n^3)$ time.
- Closure properties: The closure of context-free languages under intersection with regular languages can be demonstrated effectively using pushdown automata.

We consider the language

$$L = \{a_1 a_2 \cdots a_n \$ a_n \cdots a_2 a_1 \mid a_i \in \Delta\}$$

with $\Sigma = \Delta \cup \{\$\}$, $\$ \notin \Delta$.

A finite automaton cannot recognize this language because it cannot "remember" arbitrarily long words of the form $a_1 a_2 \cdots a_n$.

However, it would need to remember such words to verify the match with the part of the word after \$.

Pushdown Automata

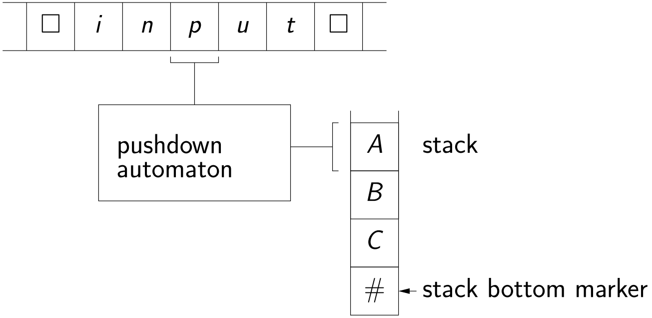
To define an automaton model for context-free languages:

- We introduce a stack or pushdown storage where an arbitrarily long sequence of symbols can be stored.
- When reading a new symbol, the top symbol of the stack can be accessed and modified as follows:
  - Either the stack remains unchanged, or
  - the top symbol of the stack is removed and replaced by a (possibly empty) sequence of symbols.

  At other times, the stack cannot be read or modified.

Pushdown Automata

Schematic representation of a pushdown automaton:



Pushdown Automata

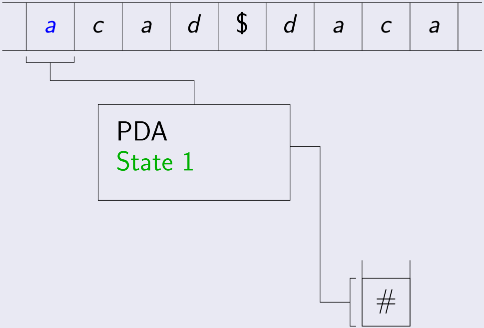
Let $\Delta = \{a, b, c, d\}$ and $L = \{a_1 a_2 \cdots a_n \$ a_n \cdots a_2 a_1 \mid a_i \in \Delta\}$.

A pushdown automaton recognizes this language as follows:

- The word w is read from left to right.
- As long as \$ has not been reached, each symbol read is pushed onto the stack as an uppercase letter ($a \rightsquigarrow A$, $b \rightsquigarrow B$, $\dots$).
- When \$ is read, the stack remains unchanged.
- Subsequently, for each new symbol read, it is checked whether the corresponding uppercase letter is on top of the stack. This letter is then removed.
- If at any point no match is found, the pushdown automaton halts.
- If matches are always found, the stack bottom marker $\#$ is eventually removed, and the automaton accepts with an empty stack.

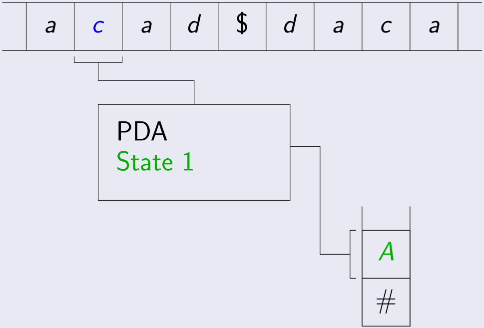
Pushdown Automata

Simulation



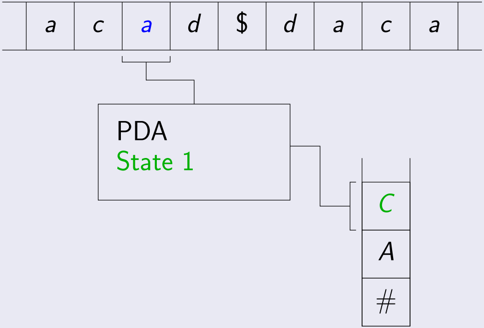
Pushdown Automata

Simulation

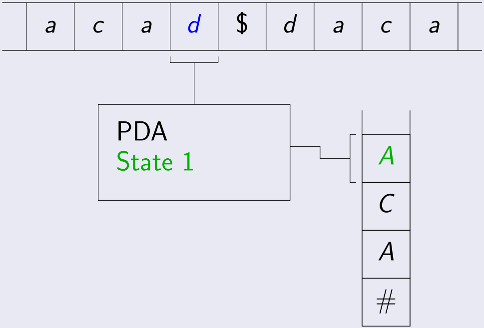


Pushdown Automata

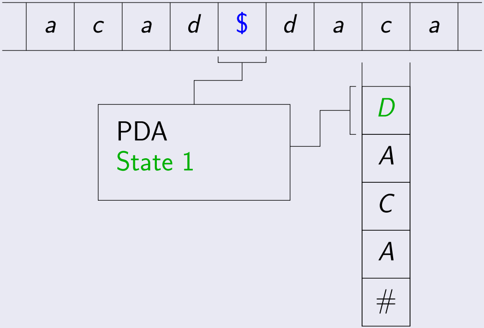
Simulation



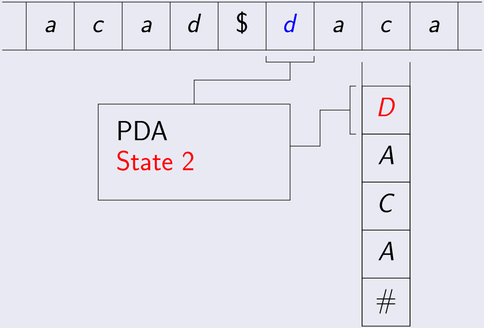
Simulation



Simulation

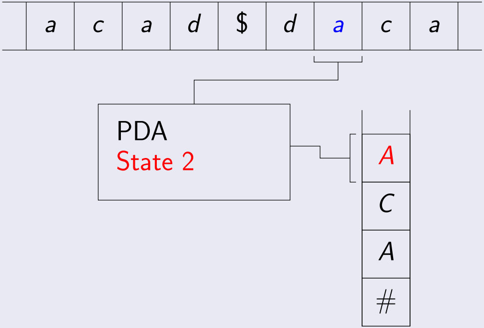


Simulation



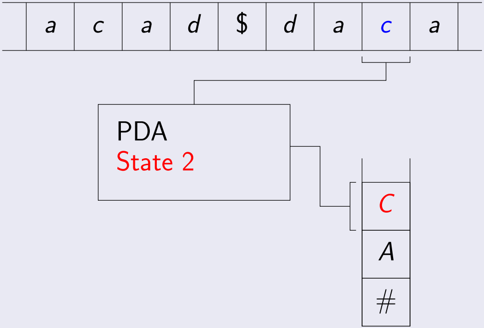
Pushdown Automata

Simulation



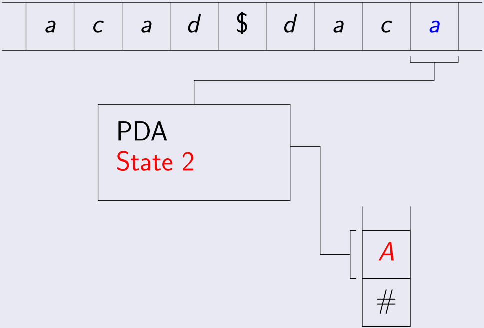
Pushdown Automata

Simulation

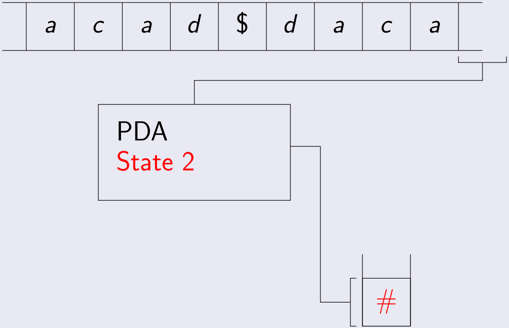


Pushdown Automata

Simulation

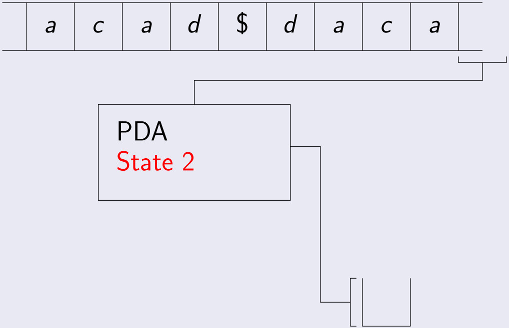


Simulation



Pushdown Automata

Simulation



Pushdown Automata

Definition (Pushdown Automaton)

A nondeterministic pushdown automaton M is a 6-tuple $M = (Z, \Sigma, \Gamma, \delta, z_0, \#)$, where

- Z is the finite set of states,
- Σ is the finite input alphabet (with $Z \cap \Sigma = \emptyset$),
- Γ is the finite stack alphabet,
- $z_0 \in Z$ is the initial state,
- $\# \in \Gamma$ is the bottom-of-stack symbol or stack base symbol, and
- $\delta\colon Z \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to 2^{Z \times \Gamma^*}$ is the transition function, where $\delta(z, a, A)$ for all $(z, a, A) \in Z \times (\Sigma \cup \{\varepsilon\}) \times \Gamma$ must be finite.

Abbreviation: KA or PDA (pushdown automaton).

- We consider the transition function

$$\delta\colon Z \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to 2^{Z \times \Gamma^*}.$$

If $(z', B_1 \cdots B_k) \in \delta(z, a, A)$, this means:

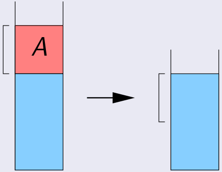
- When in state z, if the input symbol a is read and the symbol A is on top of the stack, then
- A is removed from the stack and replaced with $B_1 \cdots B_k$ (B_1 is on top), and the automaton transitions to state z'.

It is also possible for $a = \varepsilon$. In this case, no input symbol is read. We refer to this as an ε-transition.

We consider different cases for the values of the transition function δ:

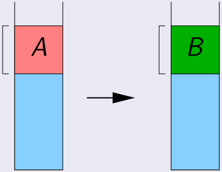
$(z', \varepsilon) \in \delta(z, a, A)$

- Symbol a is read.
- State changes from z to z'.
- Symbol A is removed from the stack:



$(z', B) \in \delta(z, a, A)$

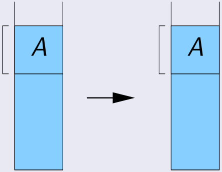
- Symbol a is read.
- State changes from z to z'.
- Symbol A on the stack is replaced with B:



Pushdown Automata

$(z', A) \in \delta(z, a, A)$

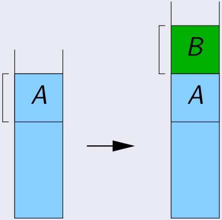
- Symbol a is read.
- State changes from z to z'.
- Symbol A remains on the stack:



Pushdown Automata

$(z', BA) \in \delta(z, a, A)$

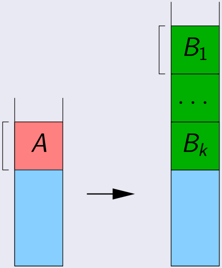
- Symbol a is read.
- State changes from z to z'.
- Symbol B is newly pushed onto the stack:



Pushdown Automata

$(z', B_1 \cdots B_k) \in \delta(z, a, A)$

- Symbol a is read.
- State changes from z to z'.
- Symbol A is replaced with multiple new symbols:



Pushdown Automata

- At the start of every computation, the stack contains only the stack bottom symbol $\#$.
- The stack is unbounded and can grow arbitrarily. There are infinitely many possible stack contents, which distinguishes pushdown automata from finite automata.
- The pushdown automata we consider always accept with an empty stack (in this case, no further transitions are possible). However, there are other variants of pushdown automata that accept with an end state.

Pushdown Automata

Example:

PDA for $L = \{a_1 a_2 \cdots a_n \$ a_n \cdots a_2 a_1 \mid n \geq 0, a_1, \ldots, a_n \in \{a, b\}\}$:

$$M = (\{z_1, z_2\}, \{a, b, \$\}, \{\#, A, B\}, \delta, z_1, \#),$$

where δ is defined as follows (we write $(z, a, A) \to (z', x)$ if $(z', x) \in \delta(z, a, A)$).

$$\begin{array}{ccc}
(z_1, a, \#) \to (z_1, A\#) & (z_1, a, A) \to (z_1, AA) & (z_1, a, B) \to (z_1, AB) \\
(z_1, b, \#) \to (z_1, B\#) & (z_1, b, A) \to (z_1, BA) & (z_1, b, B) \to (z_1, BB) \\
(z_1, \$, \#) \to (z_2, \#) & (z_1, \$, A) \to (z_2, A) & (z_1, \$, B) \to (z_2, B) \\
(z_2, a, A) \to (z_2, \varepsilon) & (z_2, b, B) \to (z_2, \varepsilon) & (z_2, \varepsilon, \#) \to (z_2, \varepsilon)
\end{array}$$

Definition (Configuration of a PDA)

A configuration of a PDA is a triple $k \in Z \times \Sigma^* \times \Gamma^*$.

Meaning of the components of $k = (z, w, \gamma) \in Z \times \Sigma^* \times \Gamma^*$:

- $z \in Z$ is the current state of the PDA.
- $w \in \Sigma^*$ is the remaining input to be read.
- $\gamma \in \Gamma^*$ is the current stack content, with the top stack symbol at the far left.

Pushdown Automata

Transitions between configurations are derived from the transition function δ:

Definition (Configuration transitions of a PDA)

It holds that

$$(z, aw, A\gamma) \vdash (z', w, B_1 \cdots B_k\gamma),$$

if $(z', B_1 \cdots B_k) \in \delta(z, a, A)$, and

$$(z, w, A\gamma) \vdash (z', w, B_1 \cdots B_k\gamma),$$

if $(z', B_1 \cdots B_k) \in \delta(z, \varepsilon, A)$.

Here, $\gamma \in \Gamma^*$ is an arbitrary sequence of stack symbols, $A, B_1, \ldots, B_k \in \Gamma$, $w \in \Sigma^*$, $a \in \Sigma$, and $z, z' \in Z$.

In the first case, an input symbol is read, while in the second case, no input is read.

We define $\vdash^*$ as the reflexive and transitive closure of $\vdash$.

Using this, the language accepted by a PDA can now be defined:

Definition (Accepted language of a PDA)

Let $M = (Z, \Sigma, \Gamma, \delta, z_0, \#)$ be a PDA. Then the accepted language of M is:

$$N(M) = \{x \in \Sigma^* \mid (z_0, x, \#) \vdash^* (z, \varepsilon, \varepsilon) \text{ for some } z \in Z\}.$$

This means the accepted language contains all words that allow the stack to be completely emptied.

However, since pushdown automata are non-deterministic, there may also be computations for this word that do not empty the stack.

The following sequence of configuration transitions demonstrates that the pushdown automaton from Slide 264 accepts the word $ab\$ba$:

$$\begin{array}{rll}
(z_1, ab\$ba, \#) & \vdash (z_1, b\$ba, A\#) & \text{due to } (z_1, a, \#) \to (z_1, A\#) \\
& \vdash (z_1, \$ba, BA\#) & \text{due to } (z_1, b, A) \to (z_1, BA) \\
& \vdash (z_2, ba, BA\#) & \text{due to } (z_1, \$, B) \to (z_2, B) \\
& \vdash (z_2, a, A\#) & \text{due to } (z_2, b, B) \to (z_2, \varepsilon) \\
& \vdash (z_2, \varepsilon, \#) & \text{due to } (z_2, a, A) \to (z_2, \varepsilon) \\
& \vdash (z_2, \varepsilon, \varepsilon) & \text{due to } (z_2, \varepsilon, \#) \to (z_2, \varepsilon)
\end{array}$$

Another **example**: a PDA for the language

$$L = \{a_1 a_2 \cdots a_n a_n \cdots a_2 a_1 \mid n \geq 0, a_1, \ldots, a_n \in \{a, b\}\}.$$

Idea: Instead of waiting for the symbol \$, the automaton can non-deterministically decide to transition to state z_2 ($=$ clearing the stack) as soon as the current symbol on the tape matches the symbol on the stack (or if the stack is empty).

Pushdown Automata

Modified transition function δ (3rd row is changed):

$$
\begin{array}{lll}
(z_1, a, \#) \to (z_1, A\#) & (z_1, a, A) \to (z_1, AA) & (z_1, a, B) \to (z_1, AB) \\
(z_1, b, \#) \to (z_1, B\#) & (z_1, b, A) \to (z_1, BA) & (z_1, b, B) \to (z_1, BB) \\
(z_1, \varepsilon, \#) \to (z_2, \#) & (z_1, a, A) \to (z_2, \varepsilon) & (z_1, b, B) \to (z_2, \varepsilon) \\
(z_2, a, A) \to (z_2, \varepsilon) & (z_2, b, B) \to (z_2, \varepsilon) & (z_2, \varepsilon, \#) \to (z_2, \varepsilon)
\end{array}
$$

Note: This pushdown automaton is (unlike the previous one) non-deterministic, meaning a configuration can have multiple possible successors. (Some configuration sequences may lead to dead ends and fail to empty the stack.)

Example: The pushdown automaton receives the input *aabbaa*.

The following sequence of configuration transitions shows that this input is accepted:

$$
\begin{array}{rll}
(z_1, aabbaa, \#) & \vdash (z_1, abbaa, A\#) & \text{due to } (z_1, a, \#) \to (z_1, A\#) \\
& \vdash (z_1, bbaa, AA\#) & \text{due to } (z_1, a, A) \to (z_1, AA) \\
& \vdash (z_1, baa, BAA\#) & \text{due to } (z_1, b, A) \to (z_1, BA) \\
& \vdash (z_2, aa, AA\#) & \text{due to } (z_1, b, B) \to (z_2, \varepsilon) \\
& \vdash (z_2, a, A\#) & \text{due to } (z_2, a, A) \to (z_2, \varepsilon) \\
& \vdash (z_2, \varepsilon, \#) & \text{due to } (z_2, a, A) \to (z_2, \varepsilon) \\
& \vdash (z_2, \varepsilon, \varepsilon) & \text{due to } (z_2, \varepsilon, \#) \to (z_2, \varepsilon)
\end{array}
$$

Note: There are also many other possible computations where the stack is not empty at the end, such as:

$$
\begin{array}{llll}
(z_1, aabbaa, \#) & \vdash (z_1, abbaa, A\#) & & \text{due to } (z_1, a, \#) \to (z_1, A\#) \\
& \vdash (z_1, bbaa, AA\#) & & \text{due to } (z_1, a, A) \to (z_1, AA) \\
& \vdash (z_1, baa, BAA\#) & & \text{due to } (z_1, b, A) \to (z_1, BA) \\
& \vdash (z_1, aa, BBAA\#) & & \text{due to } (z_1, b, B) \to (z_1, BB) \\
& \vdash (z_1, a, ABBAA\#) & & \text{due to } (z_1, a, B) \to (z_1, AB) \\
& \vdash (z_1, \varepsilon, AABBAA\#) & & \text{due to } (z_1, a, A) \to (z_1, AA)
\end{array}
$$

However, such computations do not change the fact that the word *aabbaa* is accepted.

For this, the existence of the one computation on the previous slide, where the stack is empty after reading the input, suffices.

We now need to show that pushdown automata indeed precisely accept the context-free languages.

Theorem (Context-Free Grammars $\to$ Pushdown Automata)

For every context-free grammar G, there exists a PDA M such that $L(G) = N(M)$.

Proof Idea:

1. We can assume without loss of generality that G is in Greibach Normal Form.
2. We simulate a derivation of G by using the stack to store variables that still need to be derived.
3. A production $A \to aA_1 \cdots A_n$ is simulated as follows:

   If a is the next input symbol and A is on top of the stack, A can be replaced with $A_1 \cdots A_n$.
4. When the entire input has been read and the stack is simultaneously empty, a complete derivation for the input word has been successfully simulated.

Pushdown Automata

Formal: First, we assume that $\varepsilon \notin L(G)$.

Then we can assume without loss of generality that $G = (V, \Sigma, P, S)$ is in Greibach Normal Form.

We define the PDA

$$M = (\{z\}, \Sigma, V, \delta, z, S)$$

with the following transition function: For $A \in V$ and $a \in \Sigma$, let

$$\delta(z, a, A) = \{(z, A_1 \cdots A_m) \mid (A \to aA_1 \cdots A_m) \in P\}.$$

Note:

- M has only one state (z).
- M has no ε-transitions.
- The start symbol S of G serves as the stack bottom marker.
- Since G is in Greibach Normal Form, all productions in P are of the form $A \to aA_1 \cdots A_m$ with $m \geq 0$, $A, A_1, \ldots, A_m \in V$, and $a \in \Sigma$.

Claim: For all $u \in \Sigma^*$ and $\gamma \in V^*$, the following holds:

$$(z, u, \gamma) \vdash^* (z, \varepsilon, \varepsilon) \iff \gamma \Rightarrow_G^* u.$$

Proof: By induction on $|u|$.

Base Case: $u = \varepsilon$.

In this case:

$$(z, \varepsilon, \gamma) \vdash^* (z, \varepsilon, \varepsilon) \iff \gamma = \varepsilon \iff \gamma \Rightarrow_G^* \varepsilon.$$

Inductive Step: Let $u = av$ with $a \in \Sigma$, $v \in \Sigma^*$.

If $\gamma = \varepsilon$, neither $\gamma \Rightarrow_G^* av$ nor $(z, av, \gamma) \vdash^* (z, \varepsilon, \varepsilon)$ holds.

Now assume $\gamma = A\gamma'$ with $A \in V$ and $\gamma' \in V^*$.

Then:

$$\begin{aligned}
& A\gamma' \Rightarrow_G^* av \\
\iff & \exists (A \to aA_1 \cdots A_m) \in P : A_1 \cdots A_m\gamma' \Rightarrow_G^* v \\
\iff & \exists (A \to aA_1 \cdots A_m) \in P : (z, v, A_1 \cdots A_m\gamma') \vdash^* (z, \varepsilon, \varepsilon) \\
\iff & \exists (z, A_1 \cdots A_m) \in \delta(z, a, A) : (z, v, A_1 \cdots A_m\gamma') \vdash^* (z, \varepsilon, \varepsilon) \\
\iff & (z, av, A\gamma') \vdash^* (z, \varepsilon, \varepsilon).
\end{aligned}$$

From the above claim, it follows:

$$w \in L(G) \iff S \Rightarrow_G^* w \iff (z, w, S) \vdash^* (z, \varepsilon, \varepsilon) \iff w \in N(M).$$

If $\varepsilon \in L(G)$, we can assume without loss of generality that, except for the production $S \to \varepsilon$, all productions of G are in Greibach Normal Form, and S does not appear on any right-hand side in G.

We then add to the PDA defined on Slide 275 $\delta(z, \varepsilon, S) = \{(z, \varepsilon)\}$ (the only ε-transition).

This transition can only be applied at the very beginning of a computation $(z, w, S) \vdash^* (z, \varepsilon, \varepsilon)$ (which implies that $w = \varepsilon$).

Then it again holds as desired that $L(G) = N(M)$. $\square$

Alternative Construction:

We can also directly construct a PDA M from any context-free grammar $G = (V, \Sigma, P, S)$ such that $L(G) = N(M)$.

Define the PDA $M = (\{z\}, \Sigma, V \cup \Sigma, \delta, z, S)$ with a single state z and stack alphabet $V \cup \Sigma$.

Transition function δ:

$$\delta(z, \varepsilon, A) = \{(z, \alpha) \mid (A \to \alpha) \in P\} \text{ for } A \in V$$
$$\delta(z, a, a) = \{(z, \varepsilon)\} \text{ for } a \in \Sigma$$

Productions of the first type simulate derivation steps on the stack without reading the input.

Productions of the second type compare a symbol from the input with the stack.

Note: M contains ε-productions.

Pushdown Automata

We consider the following context-free grammar with the two-element alphabet $\Sigma = \{[,]\}$, which generates correct bracket structures:

$$S \to [S]S \mid \varepsilon$$

Task: Convert this grammar into a pushdown automaton and use it to accept the word $[\,[\,]\,]\,[\,]$.

We use the construction from Slide 279:

- State set $= \{z\}$
- Stack alphabet $= \{[,], S\}$
- Stack bottom symbol $= S$
- Transition function:

$$\begin{aligned} \delta(z, \varepsilon, S) &= \{(z, \varepsilon), (z, [S]S)\} \\ \delta(z, a, a) &= \{(z, \varepsilon)\} \text{ for } a \in \{[,]\} \end{aligned}$$

On the left is a derivation of the word $[\,[\,]\,]\,[\,]$ using the grammar, and on the right is the corresponding computation of the above pushdown automaton:

$$
\begin{array}{lll}
S \Rightarrow [S]S & (z,[[\,]][\,],S) & \vdash (z,[[\,]][\,],[S]S) \\
 & & \vdash (z,[\,]][\,],S]S) \\
\Rightarrow [[S]S]S & & \vdash (z,[\,]][\,],[S]S]S) \\
 & & \vdash (z,]][\,],S]S]S) \\
\Rightarrow [[\,]S]S & & \vdash (z,]][\,],]S]S) \\
 & & \vdash (z,][\,],S]S) \\
\Rightarrow [[\,]]S & & \vdash (z,][\,],]S) \\
 & & \vdash (z,[\,],S) \\
\Rightarrow [[\,]][S]S & & \vdash (z,[\,],[S]S) \\
 & & \vdash (z,],S]S) \\
\Rightarrow [[\,]][\,]S & & \vdash (z,],]S) \\
 & & \vdash (z,\varepsilon,S) \\
\Rightarrow [[\,]][\,] & & \vdash (z,\varepsilon,\varepsilon)
\end{array}
$$

Now, we aim to show that for every pushdown automaton, there is a corresponding context-free grammar.

This is the more difficult direction.

Theorem (Pushdown Automata $\to$ Context-Free Grammars)

For every pushdown automaton M, there exists a context-free grammar G such that $N(M) = L(G)$.

Proof idea:

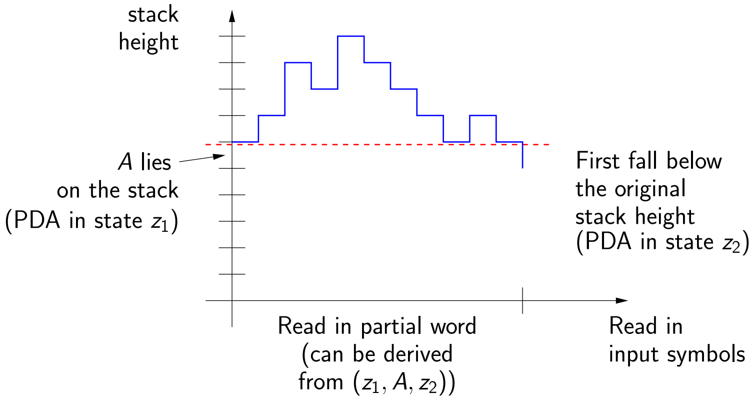
1. We want to describe which words can be accepted by reducing a specific stack symbol. The language accepted by the automaton consists of all words that can be generated by reducing $\#$.

   "Reducing" means: additional symbols can be pushed onto the stack during the process, but ultimately, the stack must be shorter by exactly one symbol.

2. The context-free grammar to be constructed will have variables of the form (z_1, A, z_2), which means:

   From (z_1, A, z_2), one can derive exactly the words that the pushdown automaton reads when it starts in state z_1, pops A from the stack, and halts in state z_2.

Pushdown Automata



In the process, A can be replaced by another symbol. However, the original stack height will not be reduced.

Formal meaning of the symbols (z_1, A, z_2):

$$(z_1, A, z_2) \Rightarrow^* x \quad \Longleftrightarrow \quad (z_1, x, A) \vdash^* (z_2, \varepsilon, \varepsilon)$$

Let $M = (Z, \Sigma, \Gamma, \delta, z_0, \#)$ be a pushdown automaton.

We define a grammar $G = (V, \Sigma, P, S)$ as follows
(see the next slide):

- Variables: $V = \{S\} \cup Z \times \Gamma \times Z$
  (Own start variable and variables of the form (z_1, A, z_2))

- Productions have the following form:

$$S \rightarrow (z_0, \#, z) \quad \text{for all } z \in Z$$

(Removing the stack bottom symbol)

$$(z, A, z') \rightarrow a \quad \text{if } (z', \varepsilon) \in \delta(z, a, A)$$

(Symbol A can – when reading symbol a – be removed immediately)

$$(z, A, z') \rightarrow a\,(z_1, B_1, z_2)(z_2, B_2, z_3) \cdots (z_k, B_k, z') \quad \text{for all}$$

$$(z_1, B_1 \cdots B_k) \in \delta(z, a, A),\ z_2, \ldots, z_k \in Z,\ k \geq 1$$

(Symbol A is replaced by $B_1 \ldots B_k$, these must be removed via intermediate states $z_1, \ldots, z_k$.)

Example: Consider the pushdown automaton

$$M = (\{z_1, z_2\}, \{a, b\}, \{A, \#\}, \delta, z_1, \#)$$

with the following transition function δ:

$$\begin{array}{rcl} (z_1, \varepsilon, \#) & \to & (z_2, \varepsilon) \\ (z_1, a, \#) & \to & (z_1, AA) \\ (z_1, a, A) & \to & (z_1, AAA) \\ (z_1, b, A) & \to & (z_2, \varepsilon) \\ (z_2, b, A) & \to & (z_2, \varepsilon) \end{array}$$

It holds: $N(M) = \{a^n b^{2n} \mid n \geq 0\}$.

Task: Convert M into a context-free grammar.

$$
\begin{array}{rcl}
S & \to & (z_1, \#, z_1) \\
S & \to & (z_1, \#, z_2) \\
(z_1, \#, z_2) & \to & \varepsilon \\
(z_1, A, z_2) & \to & b \\
(z_2, A, z_2) & \to & b \\
(z_1, \#, z_i) & \to & a(z_1, A, z_j)(z_j, A, z_i) \\
(z_1, A, z_i) & \to & a(z_1, A, z_j)(z_j, A, z_k)(z_k, A, z_i)
\end{array}
$$

The last two productions are present for all $i, j, k \in \{1, 2\}$.

Overall, the grammar has 17 productions.

Remark on conversions
"Context-free Grammar $\leftrightarrow$ Pushdown Automaton":

For every pushdown automaton M, there is always an equivalent pushdown automaton M' with only one state and without ε-transitions (if $\varepsilon \notin N(M)$).

1. First, convert M into a context-free grammar G.
2. Then, convert G into a context-free grammar G' in Greibach normal form.
3. Finally, convert G' into a pushdown automaton M'.

It is used that when converting a grammar (in Greibach normal form) into a pushdown automaton, only automata with one state and without ε-transitions are constructed.

Deterministic Context-Free Languages

We now consider a subclass of pushdown automata that can be used to recognize languages deterministically and therefore efficiently.

Definition (Deterministic Pushdown Automaton)

A deterministic pushdown automaton M is a 7-tuple
$M = (Z, \Sigma, \Gamma, \delta, z_0, \#, E)$, where

- $(Z, \Sigma, \Gamma, \delta, z_0, \#)$ is a pushdown automaton,
- $E \subseteq Z$ is a set of final states, and
- the transition function $\delta\colon Z \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to 2^{Z \times \Gamma^*}$ is deterministic in the following sense:

  For all $z \in Z$, $a \in \Sigma$, and $A \in \Gamma$:

$$|\delta(z, a, A)| + |\delta(z, \varepsilon, A)| \leq 1.$$

Differences between pushdown automata and deterministic pushdown automata:

- Deterministic pushdown automata have a set of final states and accept with a final state – not with an empty stack.

  For deterministic pushdown automata, this distinction matters, whereas for non-deterministic pushdown automata, both acceptance modes are equivalent.

- For each state z and each stack symbol A:
  - either there is at most one ε-transition,
  - or there is at most one transition for each alphabet symbol.

Deterministic Context-Free Languages

Configurations and transitions between configurations remain defined the same way.

Configuration sequences, however, become linear chains, i.e., there is always at most one subsequent configuration.

This property is utilized for the efficient solution of the word problem.

Deterministic Context-Free Languages

Definition (Accepted Language for Det. PDA)

Let $M = (Z, \Sigma, \Gamma, \delta, z_0, \#, E)$ be a deterministic PDA. Then the accepted language of M is:

$$D(M) = \{x \in \Sigma^* \mid (z_0, x, \#) \vdash^* (z, \varepsilon, \gamma) \text{ for some } z \in E, \gamma \in \Gamma^*\}.$$

Compare this definition with that for non-deterministic pushdown automata!

For deterministic pushdown automata, the following is different:

- The reached state z must be a final state.
- A stack content γ may remain.

Deterministic Context-Free Languages

Definition (Deterministic Context-Free Languages)

A language is called deterministic context-free if and only if it is accepted by a deterministic PDA.

Examples:

- The language $L = \{a_1 a_2 \dots a_n \$ a_n \dots a_2 a_1 \mid a_i \in \Delta\}$ is deterministic context-free (see the corresponding PDA).
- The language $L = \{a_1 a_2 \dots a_n a_n \dots a_2 a_1 \mid a_i \in \Delta\}$ is not deterministic context-free (without proof).

Note: A priori, the definition of deterministic context-free languages does not immediately imply that deterministic context-free languages are also context-free (acceptance by final states versus empty stack).

However, this is the case: From a deterministic PDA $M = (Z, \Sigma, \Gamma, \delta, z_0, \#, E)$, we construct a (non-deterministic) PDA $M' = (Z \cup \{z_0', z_f\}, \Sigma, \Gamma \cup \{\#'\}, \delta', z_0', \#')$, where:

$$\delta'(z_0', \varepsilon, \#') = \{(z_0, \#\#')\}$$

$$\delta'(z, a, A) = \begin{cases} \delta(z, a, A) & \text{if } (z \in Z \setminus E \text{ or } a \in \Sigma),\ A \in \Gamma \\ \delta(z, a, A) \cup \{(z_f, \varepsilon)\} & \text{if } z \in E, a = \varepsilon, A \in \Gamma \end{cases}$$

$$\delta'(z, \varepsilon, \#') = \{(z_f, \varepsilon)\} \quad \text{if } z \in E$$

$$\delta'(z_f, \varepsilon, A) = \{(z_f, \varepsilon)\} \quad \text{if } A \in \Gamma \cup \{\#'\}$$

Then: $N(M') = D(M)$.

The construction on the previous slide also shows how to transform a (non-deterministic) PDA that accepts by final states into a (non-deterministic) PDA that accepts by empty stack.

Conversely, a (non-deterministic) PDA $M = (Z, \Sigma, \Gamma, \delta, z_0, \#)$ that accepts by empty stack can be transformed into a (non-deterministic) PDA that accepts by final states as follows:

Let $M' = (Z \cup \{z_0', z_f\}, \Sigma, \Gamma \cup \{\#'\}, \delta', z_0', \#', \{z_f\})$, where:

$$\begin{aligned} \delta'(z_0', \varepsilon, \#') &= \{(z_0, \#\#')\} \\ \delta'(z, a, A) &= \delta(z, a, A) \quad \text{if } z \in Z, a \in \Sigma \cup \{\varepsilon\}, A \in \Gamma \\ \delta'(z, \varepsilon, \#') &= \{(z_f, \varepsilon)\} \quad \text{for all } z \in Z \end{aligned}$$

Then: $N(M') = N(M)$.

Deterministic Context-Free Languages

Additional Remarks:

- Efficiency: Using deterministic pushdown automata provides a method to solve the word problem with complexity $O(n)$, where n is the length of the input word.

  The procedure involves simply running the automaton on the word and checking whether it reaches a final state.

- Deterministic Context-Free Grammars: Since the syntax of languages can be more easily defined using grammars rather than automata, it is necessary to define the corresponding class of deterministic context-free grammars for deterministic pushdown automata.

  As this is not straightforward, there are multiple approaches to it. The most well-known are the $LR(k)$ grammars (see compiler construction and syntax analysis).

The closure properties of deterministic context-free languages differ somewhat from those of general context-free languages.

Theorem (Closure under Complement)

If L is a deterministic context-free language, then $\overline{L} = \Sigma^* \setminus L$ is also deterministic context-free.

We omit the rather technical proof here.

No Closure under Intersection

There exist deterministic context-free languages L_1 and L_2 such that $L_1 \cap L_2$ is not deterministic context-free.

Justification:

The example languages used in the argument that context-free languages are not closed under intersection are actually deterministic context-free, but their intersection is not even context-free:

$$
\begin{aligned}
L_1 &= \{a^j b^k c^k \mid j \geq 0, k \geq 0\} \\
L_2 &= \{a^k b^k c^j \mid j \geq 0, k \geq 0\}
\end{aligned}
$$

Deterministic Context-Free Languages

No Closure under Union

There exist deterministic context-free languages L_1 and L_2 such that $L_1 \cup L_2$ is not deterministic context-free.

Justification:

Closure under union and complement would imply closure under intersection (since $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$).

Deterministic Context-Free Languages

It is, however, true that there is closure under intersection with regular languages:

Theorem (Closure under Intersection with Regular Languages)

Let L be a deterministic context-free language and R a regular language. Then $L \cap R$ is a deterministic context-free language.

Proof Idea: (analogous to the cross-product construction for NFAs)

Let $M = (Z_1, \Sigma, \Gamma, \delta_1, z_0^1, \#, E_1)$ be a deterministic PDA for L.

Let $A = (Z_2, \Sigma, \delta_2, z_0^2, E_2)$ be a DFA for R.

Construct a deterministic PDA M' for $L \cap R$:

$$M' = (Z_1 \times Z_2, \Sigma, \Gamma, \delta', (z_0^1, z_0^2), \#, E_1 \times E_2).$$

Here, the transition function δ' is defined as follows:

$$\begin{aligned}
\delta'((z_1,z_2),a,A) &= \{((z_1',z_2'),B_1\cdots B_k) \mid (z_1',B_1\cdots B_k)\in\delta_1(z_1,a,A),\\
&\qquad\qquad\qquad\qquad \delta_2(z_2,a)=z_2', a\in\Sigma\}\\
\delta'((z_1,z_2),\varepsilon,A) &= \{((z_1',z_2),B_1\cdots B_k) \mid (z_1',B_1\cdots B_k)\in\delta_1(z_1,\varepsilon,A)\}
\end{aligned}$$

Note: The transition function thus defined satisfies the requirements of the definition of deterministic PDAs. □

Revisiting Closure Properties

Using the same technique and leveraging the fact that for general (non-deterministic) pushdown automata, acceptance by empty stack is equivalent to acceptance by final state, the following can also be shown:

Theorem (Closure under Intersection with Regular Languages II)

Let L be a context-free language and R a regular language. Then $L \cap R$ is a context-free language.

Decidability

We now examine problems for context-free languages and determine whether they are decidable, i.e., whether there are algorithms to solve them.

Word Problem for a Context-Free Language L

Given $w \in \Sigma^*$.
Is $w \in L$?

If the context-free language L is defined by a context-free grammar in Chomsky Normal Form, the word problem can be solved using the CYK algorithm in $O(|w|^3)$ time.

If L is deterministic context-free and given by a deterministic PDA, the word problem for L can be solved in $O(n)$ time.

Decidability

Emptiness Problem for Context-Free Languages

Given a context-free grammar $G = (V, \Sigma, P, S)$.
Is $L(G) = \emptyset$?

Determine the set

$$W = \{A \in V \mid \exists w \in \Sigma^* : A \Rightarrow_G^* w\}$$

of all productive variables (variables that can derive a terminal word):

$W := \{A \in V \mid \exists w \in \Sigma^* : (A \to w) \in P\}$
$W' := \emptyset$
while $W' \neq W$ do
  $W' := W$
  $W := W \cup \{A \in V \mid \exists w \in (\Sigma \cup W)^* : (A \to w) \in P\}$
endwhile

Then it holds: $L(G) \neq \emptyset \iff S \in W$.

Decidability

Finiteness Problem for Context-Free Languages

Given a context-free grammar $G = (V, \Sigma, P, S)$.
Is $L(G)$ finite?

Without loss of generality, we can assume that G is in Chomsky Normal Form.

We define a graph (W, E) on the set W of productive variables (see previous slide) with the following edge relation:

$$E = \{(A, B) \in W \times W \mid \exists C \in W : (A \to BC) \in P \text{ or } (A \to CB) \in P\}$$

Claim: $|L(G)| = \infty \iff \exists A \in W : (S, A) \in E^* \text{ and } (A, A) \in E^+$.

Note: $(B, C) \in E^*$ (or $(B, C) \in E^+$) means there is a path (or a non-empty path, i.e., a path with at least one edge) from B to C in the binary relation E. $(B, B) \in E^*$ always holds!

Decidability

"$\Leftarrow$": Let $A \in W$ be such that $(S, A) \in E^*$ and $(A, A) \in E^+$.

Then there exist derivations in G of the form:

$$S \Rightarrow_G^* uAy, \qquad A \Rightarrow_G^+ vAx, \qquad A \Rightarrow_G^* w$$

with $u, v, w, x, y \in \Sigma^*$.

Hence, $S \Rightarrow_G^* uv^iwx^iy \in \Sigma^*$ for all $i \geq 0$.

Since in the derivation $A \Rightarrow_G^+ vAx$ at least one derivation step is made, and G is in Chomsky Normal Form, it must be the case that $vx \neq \varepsilon$.

Therefore, $\{uv^iwx^iy \mid i \geq 0\}$ is infinite, so $L(G)$ is infinite.

"$\Rightarrow$": Let $L(G)$ be infinite.

Let n be the constant from the Pumping Lemma ($= 2^{|V|}$) and let $z \in L(G)$ with $|z| \geq n$ (such a word z exists if $L(G)$ is infinite!).

In the proof of the Pumping Lemma, we saw that there exists a variable A with derivations $S \Rightarrow_G^* uAy$, $A \Rightarrow_G^+ vAx$, and $A \Rightarrow_G^* w$, where $z = uvwxy$.

Hence, A is productive: $A \in W$.

The derivations $S \Rightarrow^* uAy$ and $A \Rightarrow^+ vAx$ (more precisely, the path in the syntax tree from the root S to the second occurrence of A) show that $(S, A) \in E^*$ and $(A, A) \in E^+$ holds. $\square$

Example:

Let G be the grammar in Chomsky Normal Form with the productions

$$\begin{aligned} S &\to AC \\ A &\to BC \\ B &\to CA \mid b \\ C &\to a \end{aligned}$$

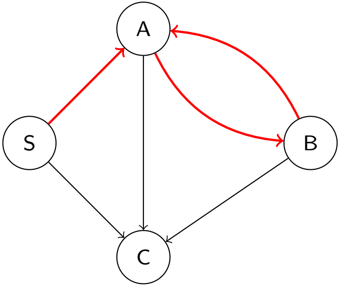
In this case, $W = \{S, A, B, C\}$, meaning all variables are productive: After running i iterations through the **while loop** (Slide 305), we obtain

1. for $i = 0$: $W = \{B, C\}$
2. for $i = 1$: $W = \{A, B, C\}$
3. for $i = 2$: $W = \{S, A, B, C\}$

Since $S \in W$, it follows that $L(G) \neq \emptyset$.

Example (Continued):

The graph (W, E) is then



The red path shows that $L(G)$ is infinite.

Undecidability for Context-Free Languages

The following problems are undecidable for context-free languages, i.e., it can be shown that there is no corresponding algorithm to solve them:

- Equivalence Problem: Given two context-free languages L_1, L_2. Is $L_1 = L_2$?
- Intersection Problem: Given two context-free languages L_1, L_2. Is $L_1 \cap L_2 = \emptyset$?

Note: In the lecture **Computability and Logic**, we will see how such undecidability results can be proven.

Decidability

The Intersection Problem is, however, decidable when it is known that one of the two languages L_1, L_2 is regular and given as a finite automaton.

Algorithm:

1. In this case, a pushdown automaton M can be constructed (construction shown earlier), which accepts $L_1 \cap L_2$.
2. The pushdown automaton M can then be transformed into a context-free grammar G.
3. By determining the productive variables of G, it can be determined whether S is non-productive and thus whether $L_1 \cap L_2$ is empty.

Decidability

Decidability for Deterministic Context-Free Languages

The following problems are decidable for deterministic context-free languages (represented by a deterministic pushdown automaton):

- Word Problem for a Deterministic Context-Free Language L: Given $w \in \Sigma^*$. Is $w \in L$?

  With a deterministic pushdown automaton in $O(|w|)$ time.

- Emptiness Problem: Given a deterministic context-free language L. Is $L = \emptyset$?

  See the corresponding decision procedure for context-free languages.

Decidability

Decidability for Deterministic Context-Free Languages

- Finiteness Problem: Given a deterministic context-free language L. Is L finite?

  See the corresponding decision procedure for context-free languages.

- Equivalence Problem: Given two deterministic context-free languages L_1, L_2. Is $L_1 = L_2$?

  This was an open problem for a long time, and decidability was shown by Gérard Sénizergues in 1997.

Decidability

Undecidability for Deterministic Context-Free Languages

The following problems are undecidable for deterministic context-free languages, i.e., it can be shown that there is no corresponding procedure:

- Intersection Problem: Given two deterministic context-free languages L_1, L_2. Is $L_1 \cap L_2 = \emptyset$?

  As with context-free languages, this problem is decidable when one of the two languages is regular.
- Inclusion Problem: Given two deterministic context-free languages L_1, L_2. Is $L_1 \subseteq L_2$?

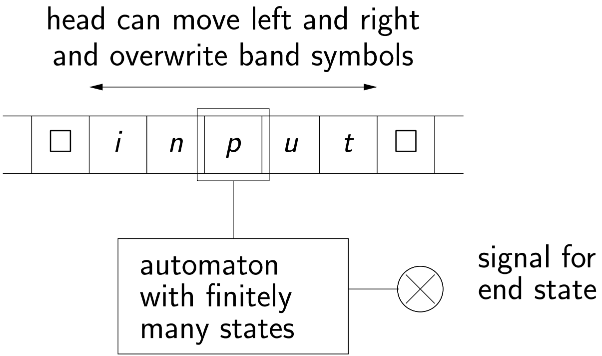
Turing Machines

In the remainder of the lecture, we will introduce machine models for Chomsky-0 and Chomsky-1 languages.

- Chomsky-0 languages: Turing machines (named after Alan Turing, 1912-1954)
- Chomsky-1 languages: Linear bounded automata (a restriction of Turing machines)

Turing Machines

Schematic representation of a Turing machine:



Turing Machines

Properties of Turing machines:

- Like finite automata, Turing machines have a finite number of states and read an input from a tape, which is divided into cells (fields).
- In each field of the tape, there is a symbol from a finite tape alphabet. A read/write head moves over the tape.
- Difference from finite automata: the read/write head can move left and right and can also overwrite symbols.
- If only symbols from the input word are overwritten, the Turing machine is called linear bounded (machine model for Chomsky-1 languages).
- If the read/write head can move beyond the left and right boundaries of the input word and write there, the Turing machine is called general with an unbounded tape (machine model for Chomsky-0 languages).

Turing Machines and Computers:

- The concept of the Turing machine was invented by Alan Turing in 1936, even before the first real computers were built.
- It is interesting not only for historical reasons but also because it represents a very simple computational model.

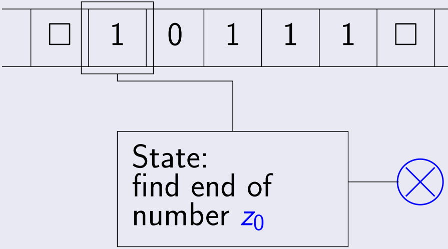
  When one wants to show that something is *not* computable, it is much better to do this with a as simple as possible computational model. (Of course, one should first ensure that this computational model is equivalent to more complex models.)
- Analogy to a modern computer:
  - Control with a finite number of states $\rightsquigarrow$ Program
  - (Input) Tape $\rightsquigarrow$ Memory

Example 1: Turing machine that increments a binary number on the tape by one.

Idea:

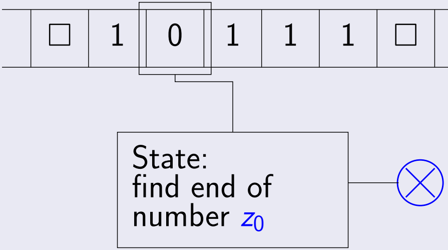
- The head of the Turing machine starts on the leftmost (most significant) bit of the binary number.
- Move the head to the right until a blank space is found.
- Then move the head back to the left, replacing each 1 with 0 until a 0 or a blank space $\Box$ (a special tape symbol) is encountered.
- Replace this symbol with 1, then move to the beginning of the number and transition to a final state.

Simulation (increment binary number 10111)



$\square$ = empty space

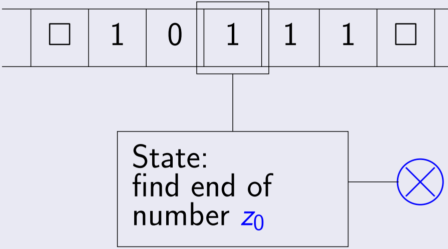
Simulation (increment binary number 10111)



$\square$ = empty space

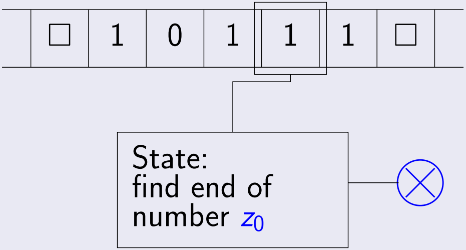
Turing Machines

Simulation (increment binary number 10111)



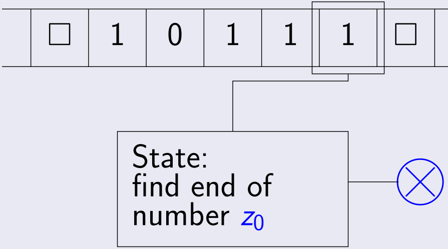
□ = empty space

Simulation (increment binary number 10111)



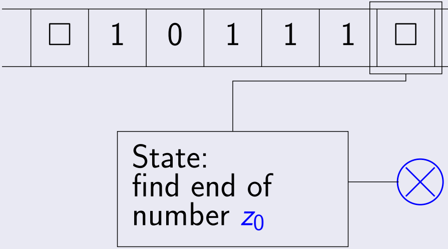
□ = empty space

Simulation (increment binary number 10111)



□ = empty space

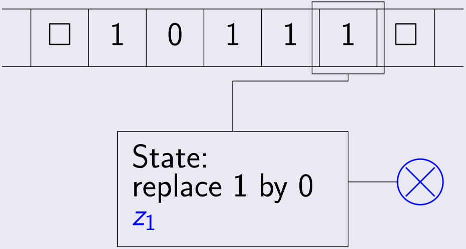
Simulation (increment binary number 10111)



$\square$ = empty space

Turing Machines

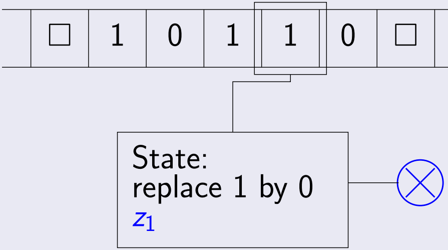
Simulation (increment binary number 10111)



$\square$ = empty space

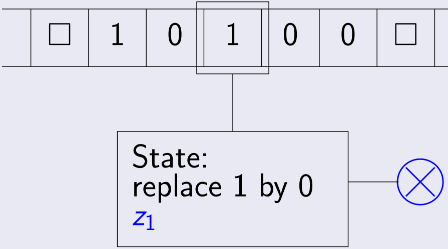
Turing Machines

Simulation (increment binary number 10111)



□ = empty space

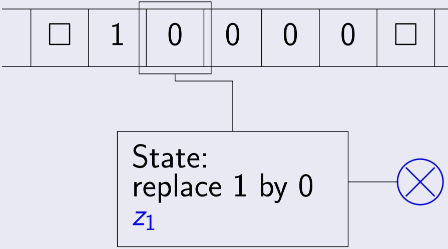
Simulation (increment binary number 10111)



□ = empty space

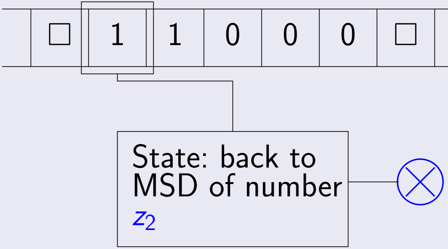
Turing Machines

Simulation (increment binary number 10111)



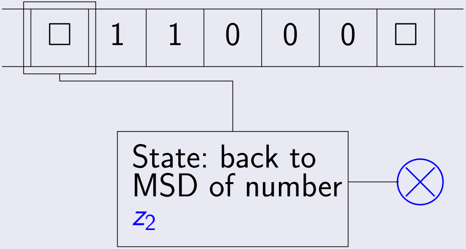
$\square$ = empty space

Simulation (increment binary number 10111)



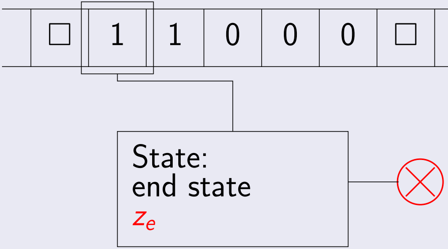
□ = empty space

Simulation (increment binary number 10111)



$\square$ = empty space

Simulation (increment binary number 10111)



□ = empty space

Turing Machine (Definition)

A deterministic Turing machine M is a 7-tuple $M = (Z, \Sigma, \Gamma, \delta, z_0, \Box, E)$, where

- Z is the finite set of states,
- Σ is the finite input alphabet,
- Γ with $\Gamma \supseteq \Sigma$ is the finite working alphabet or tape alphabet (it should hold that $\Gamma \cap Z = \emptyset$),
- $z_0 \in Z$ is the start state,
- $E \subseteq Z$ is the set of final states,
- $\delta\colon (Z \setminus E) \times \Gamma \to Z \times \Gamma \times \{L, R, N\}$ is the transition function, and
- $\Box \in \Gamma \setminus \Sigma$ is the blank or blank space.

Abbreviation: TM

Meaning of the Transition Function:
Let $\delta(z, a) = (z', b, x)$ with $z, z' \in Z$, $a, b \in \Gamma$, and $x \in \{L, R, N\}$.

If the Turing machine is in state z and the tape symbol a is currently in the cell where the (read-write) head is located, then

- it transitions to state z',
- overwrites the a in the current cell with b, and
- performs the following head movement:
  - Move the head one cell to the left if $x = L$.
  - Keep the head in place if $x = N$.
  - Move the head one cell to the right if $x = R$.

Note: Since $\delta\colon (Z \setminus E) \times \Gamma \to Z \times \Gamma \times \{L, R, N\}$ (i.e., δ is not defined for pairs (z, a) with $z \in E$), the Turing machine halts exactly when the current state is a final state from E.

In addition to deterministic Turing machines, there are also non-deterministic Turing machines.

Transition function for non-deterministic Turing machines:

$$\delta \colon (Z \setminus E) \times \Gamma \to 2^{Z \times \Gamma \times \{L,R,N\}}.$$

A (possibly empty) set of possible actions is assigned to each state and tape symbol.

However, for now, we will focus on deterministic Turing machines.

Turing Machines

Example: Turing machine for incrementing a binary number

$$M = (\{z_0, z_1, z_2, z_e\}, \{0, 1\}, \{0, 1, \Box\}, \delta, z_0, \Box, \{z_e\}) \quad \text{with}$$

Transition function: finding end of number

$$\begin{aligned} \delta(z_0, 0) &= (z_0, 0, R) \\ \delta(z_0, 1) &= (z_0, 1, R) \\ \delta(z_0, \Box) &= (z_1, \Box, L) \end{aligned}$$

Transition function: replace 1 by 0

$$\begin{aligned} \delta(z_1, 0) &= (z_2, 1, L) \\ \delta(z_1, 1) &= (z_1, 0, L) \\ \delta(z_1, \Box) &= (z_e, 1, N) \end{aligned}$$

Transition function: back to MSD of number (not so important)

$$\begin{aligned}\delta(z_2, 0) &= (z_2, 0, L)\\ \delta(z_2, 1) &= (z_2, 1, L)\\ \delta(z_2, \Box) &= (z_e, \Box, R)\end{aligned}$$

Example 2: Turing machines for language recognition

We are looking for a Turing machine that recognizes the language $L = \{a^{2^n} \mid n \geq 0\}$ (not context-free!).

Idea:

- The head initially stands at the leftmost end of the sequence of a's.
- Write the binary number 0 next to the sequence of a's on the tape.
- Replace the a's one by one with another symbol ($\#$). After each replacement, move left to the counter and increment it by one.
- Once all the a's are gone (after the last $\#$ comes a $\square$), check if the counter has the form $10\cdots0$.

  Note: A number n is a power of two if and only if its binary representation has the form $10\cdots0$.

Turing Machines

As with other machine models (e.g., pushdown automata), Turing machines also have the concept of a configuration, i.e., a snapshot of a Turing machine's computation.

Configuration (Definition)

A configuration of a Turing machine is a word

$$k \in \Gamma^* Z \Gamma^+.$$

Meaning: $k = \alpha z \beta$ with $z \in Z$, $\alpha \in \Gamma^*$, $\beta \in \Gamma^+$ (so β is a non-empty word)

- To the left of the head, the tape contains the word $\cdots \Box \alpha$
- From the cell where the head is currently positioned, and to the right of it, the tape contains the word $\beta \Box \cdots$ The head is positioned on the first symbol of β (here, $\beta \neq \varepsilon$ is important).
- $z \in Z$ is the current state.

Turing Machines

$\cdots\Box$ represents an infinite sequence of $\Box$'s extending to the left.

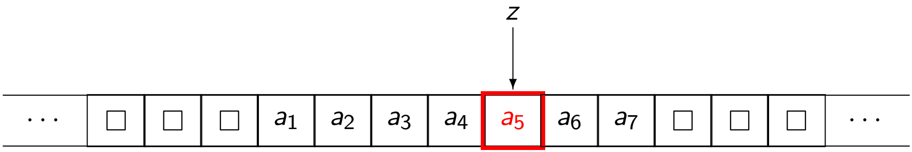
$\Box\cdots$ represents an infinite sequence of $\Box$'s extending to the right.

The tape is therefore unbounded to the left and right, but only a finite section of the tape contains tape symbols from $\Gamma \setminus \{\Box\}$.

Note: The words $\alpha z \beta$ and $\Box\alpha z\beta\Box$ describe the same configuration (the blanks at the beginning and end of $\Box\alpha z\beta\Box$ are effectively redundant).

Example: A graphical representation of the configuration

$a_1 a_2 a_3 a_4 \; z \; a_5 a_6 a_7$



Definition of a transition relation $\vdash_M$, which describes which configuration transitions are possible.

No Movement

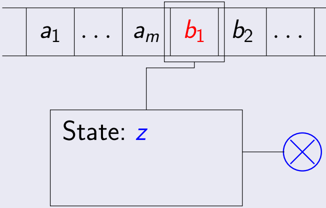
We have: $a_1 \cdots a_m z b_1 b_2 \cdots b_n \vdash_M a_1 \cdots a_m \mathbf{z}' c b_2 \cdots b_n$,
if $\delta(z, b_1) = (\mathbf{z}', c, N) \quad (m \geq 0,\ n \geq 1)$.

Definition of a transition relation $\vdash_M$, which describes which configuration transitions are possible.

No Movement

We have: $a_1 \cdots a_m z b_1 b_2 \cdots b_n \vdash_M a_1 \cdots a_m \mathbf{z}' c b_2 \cdots b_n$,

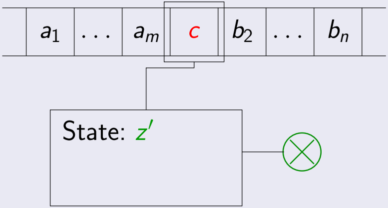
if $\delta(z, b_1) = (\mathbf{z}', c, N) \quad (m \geq 0,\ n \geq 1)$.



Definition of a transition relation $\vdash_M$, which describes which configuration transitions are possible.

No Movement

We have: $a_1 \cdots a_m z b_1 b_2 \cdots b_n \vdash_M a_1 \cdots a_m \mathbf{z}' c b_2 \cdots b_n$,
if $\delta(z, b_1) = (\mathbf{z}', c, N) \quad (m \geq 0,\ n \geq 1)$.



Definition of a transition relation $\vdash_M$, which describes which configuration transitions are possible.

Step to the Left

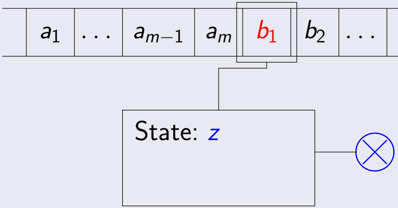
We have: $a_1 \cdots a_{m-1} a_m z b_1 b_2 \cdots b_n \vdash_M a_1 \cdots a_{m-1} \mathbf{z}' a_m c b_2 \cdots b_n$,
if $\delta(z, b_1) = (\mathbf{z}', c, L) \quad (m \geq 1,\ n \geq 1)$.

Definition of a transition relation $\vdash_M$, which describes which configuration transitions are possible.

Step to the Left

We have: $a_1 \cdots a_{m-1} a_m z b_1 b_2 \cdots b_n \vdash_M a_1 \cdots a_{m-1} \mathbf{z}' a_m c b_2 \cdots b_n$,

if $\delta(z, b_1) = (\mathbf{z}', c, L) \quad (m \geq 1,\ n \geq 1)$.



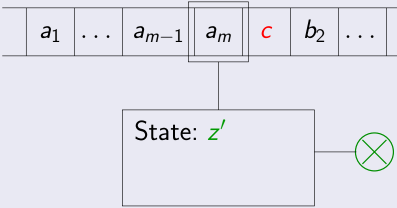
Turing Machines

Definition of a transition relation $\vdash_M$, which describes which configuration transitions are possible.

Step to the Left

We have: $a_1 \cdots a_{m-1} a_m z b_1 b_2 \cdots b_n \vdash_M a_1 \cdots a_{m-1} \mathbf{z}' a_m c b_2 \cdots b_n$,

if $\delta(z, b_1) = (\mathbf{z}', c, L) \qquad (m \geq 1,\ n \geq 1)$.



Definition of a transition relation $\vdash_M$, which describes which configuration transitions are possible.

Step to the Right

We have: $a_1 \cdots a_m z b_1 b_2 \cdots b_n \vdash_M a_1 \cdots a_m c \mathbf{z}' b_2 \cdots b_n$,

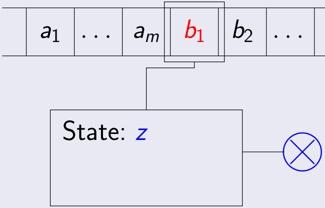
if $\delta(z, b_1) = (\mathbf{z}', c, R) \quad (m \geq 0,\ n \geq 2)$.

Definition of a transition relation $\vdash_M$, which describes which configuration transitions are possible.

Step to the Right

We have: $a_1 \cdots a_m z b_1 b_2 \cdots b_n \vdash_M a_1 \cdots a_m c \mathbf{z}' b_2 \cdots b_n$,

if $\delta(z, b_1) = (\mathbf{z}', c, R) \quad (m \geq 0,\ n \geq 2)$.

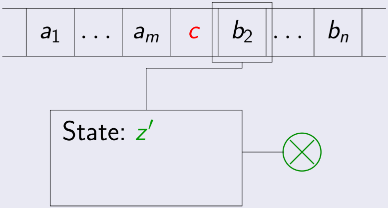


Definition of a transition relation $\vdash_M$, which describes which configuration transitions are possible.

Step to the Right

We have: $a_1 \cdots a_m z b_1 b_2 \cdots b_n \vdash_M a_1 \cdots a_m c \mathbf{z}' b_2 \cdots b_n$,

if $\delta(z, b_1) = (\mathbf{z}', c, R) \quad (m \geq 0,\ n \geq 2)$.



Special Cases: Reaching the end of the tape $\rightsquigarrow$ additional blank must be added

Left Tape End

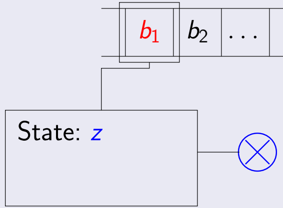
It holds: $z b_1 b_2 \cdots b_n \vdash_M \mathbf{z}' \Box c b_2 \cdots b_n$,
if $\delta(z, b_1) = (\mathbf{z}', c, L)$.

Special Cases: Reaching the end of the tape $\rightsquigarrow$ additional blank must be added

Left Tape End

It holds: $z b_1 b_2 \cdots b_n \vdash_M \mathbf{z}' \Box c b_2 \cdots b_n$,

if $\delta(z, b_1) = (\mathbf{z}', c, L)$.



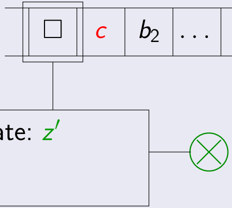
Turing Machines

Special Cases: Reaching the end of the tape $\rightsquigarrow$ additional blank must be added

Left Tape End

It holds: $z b_1 b_2 \cdots b_n \vdash_M \mathbf{z}' \Box c b_2 \cdots b_n$,

if $\delta(z, b_1) = (\mathbf{z}', c, L)$.



Special Cases: Reaching the end of the tape $\rightsquigarrow$ additional blank must be added

Right Tape End

It holds: $a_1 \cdots a_m z b_1 \vdash_M a_1 \cdots a_m c \mathbf{z}' \Box$,

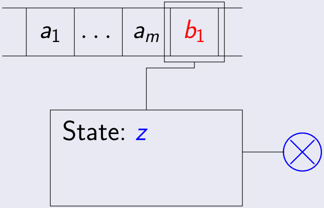
if $\delta(z, b_1) = (\mathbf{z}', c, R)$.

Special Cases: Reaching the end of the tape $\rightsquigarrow$ additional blank must be added

Right Tape End

It holds: $a_1 \cdots a_m z b_1 \vdash_M a_1 \cdots a_m c \mathbf{z}' \square$,

if $\delta(z, b_1) = (\mathbf{z}', c, R)$.

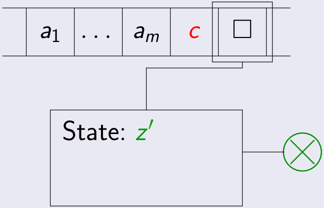


Special Cases: Reaching the end of the tape $\rightsquigarrow$ additional blank must be added

Right Tape End

It holds: $a_1 \cdots a_m z b_1 \vdash_M a_1 \cdots a_m c \mathbf{z}' \square$,

if $\delta(z, b_1) = (\mathbf{z}', c, R)$.



Turing Machines

Accepted Language (Definition)

Let $M = (Z, \Sigma, \Gamma, \delta, z_0, \Box, E)$ be a Turing machine. Then, the accepted language of M is:

$$T(M) = \{x \in \Sigma^* \mid \exists k \in \Gamma^* E \Gamma^+ : z_0 x \Box \vdash_M^* k\}.$$

Accepted Language: All input words for which the Turing machine can reach an accepting state. The Turing machine starts in the initial state z_0, with the head positioned at the first symbol of the input word. If no input exists (the input x is the empty word), the head reads a blank symbol $\Box$.

Example: The computation on the next slide corresponds to the simulation on Slide 321 for the Turing machine of Slides 325–326.

$$
\begin{aligned}
z_0 10111 &\vdash_M 1 z_0 0111 && \text{because } \delta(z_0, 1) = (z_0, 1, R) \\
&\vdash_M 10 z_0 111 && \text{because } \delta(z_0, 0) = (z_0, 0, R) \\
&\vdash_M 101 z_0 11 && \text{because } \delta(z_0, 1) = (z_0, 1, R) \\
&\vdash_M 1011 z_0 1 && \text{because } \delta(z_0, 1) = (z_0, 1, R) \\
&\vdash_M 10111 z_0 \Box && \text{because } \delta(z_0, 1) = (z_0, 1, R) \\
&\vdash_M 1011 z_1 1 \Box && \text{because } \delta(z_0, \Box) = (z_1, \Box, L) \\
&\vdash_M 101 z_1 10 \Box && \text{because } \delta(z_1, 1) = (z_1, 0, L) \\
&\vdash_M 10 z_1 100 \Box && \text{because } \delta(z_1, 1) = (z_1, 0, L) \\
&\vdash_M 1 z_1 0000 \Box && \text{because } \delta(z_1, 1) = (z_1, 0, L) \\
&\vdash_M z_2 11000 \Box && \text{because } \delta(z_1, 0) = (z_2, 1, L) \\
&\vdash_M z_2 \Box 11000 \Box && \text{because } \delta(z_2, 1) = (z_2, 1, L) \\
&\vdash_M \Box z_e 11000 \Box && \text{because } \delta(z_2, \Box) = (z_e, \Box, R)
\end{aligned}
$$

Turing Machines

For **non-deterministic Turing machines**, the definitions must be adjusted as follows:

- If the Turing machine is in state z and the symbol b is on the tape, all configuration transitions described by the set $\delta(z, b)$ are possible.
- A word is accepted if there exists a possible sequence of configurations leading to an accepting state, even if other sequences result in dead ends or run infinitely without reaching an accepting state.

Linearly Bounded Automata

We now define a machine model for Chomsky-1 languages (generated by monotone grammars): linearly bounded automata, which must never work outside the input.

Linearly Bounded Automata

A (non)deterministic linearly bounded automaton (LBA) is a tuple $A = (Z, \Sigma, \Gamma, \delta, z_0, \Box, E)$, which satisfies the same properties as a (non)deterministic Turing machine, except that (i) A cannot overwrite the blank symbol $\Box$ with a non-blank symbol, and (ii) A cannot overwrite a non-blank symbol with $\Box$.

The relation $\vdash_A$ is defined as for a Turing machine, except that the special cases for the left and right tape ends (Slides 333 and 334) are omitted.

The accepted language of the LBA A is

$$T(A) = \{w \in \Sigma^* \mid \exists k \in \Gamma^* E \Gamma^+ : z_0 w \Box \vdash_A^* k\}$$

Chomsky-1-Languages

Remark: The trailing blank symbol $\Box$ allows A to detect the right tape end. It serves as a right boundary symbol.

Theorem 3 (Kuroda)

A language L is recognized by a non-deterministic LBA if and only if there exists a Type-1 grammar G such that $L = L(G)$.

Proof:

Let $G = (V, \Sigma, P, S)$ be a Type-1 grammar, i.e., for all $(\ell, r) \in P$, we have $|\ell| \leq |r|$ (the only exception is $S \to \varepsilon$, see ε-special rule, Slide 35).

Let $w \in \Sigma^*$ be an input.

We now simulate a derivation $S \Rightarrow_G^* w$ backwards using a non-deterministic LBA A.

$B[i]$ is the i-th symbol on the tape of the LBA A.

$\tilde{\Box}$ is a new tape symbol, which functions as a copy of the blank symbol $\Box$.

The LBA A operates as follows:

1. A moves the head to the leftmost symbol on the tape, non-deterministically selects a rule $(\ell, r) \in P$ and remembers it in the state.
2. Then, the head of A moves to the right to a non-deterministically chosen position i.
3. If $B[i] \cdots B[i + |r| - 1] = r$ holds, A writes the word ℓ over the tape segment $B[i] \cdots B[i + |\ell| - 1]$. Otherwise, return to step (1).

4. If $|\ell| < |r|$, the LBA must shift every symbol on the tape from position $i + |r|$ by exactly $|r| - |\ell|$ positions to the left.

   If this creates a sentential form of length $< |w|$ on the tape, the LBA fills the sentential form with symbols $\tilde{\Box}$ at the right end (note: A is not allowed to overwrite non-blanks with the actual blank symbol $\Box$).

5. A accepts if the current tape starts with $S\Box$ or $S\tilde{\Box}$; otherwise, return to step (1).

If $S \to \varepsilon$ is a production in P (i.e., $\varepsilon \in L(G)$), A can transition directly from the start state to an accept state upon reading $\Box$.

For this LBA A, it holds that $L(G) = T(A)$.

We now prove the other direction.

The following lemma will be helpful.

Lemma 4

Let $G = (V, \Sigma \cup \{r\}, P, S)$ be a Type-1 grammar with $r \notin \Sigma$ and $L(G) \subseteq \Sigma^* r$. Then there exists a Type-1 grammar G' with

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

Proof:

Without loss of generality, we can assume that:

- For each production $(u, v) \in P$, it holds that $0 \leq |v| - |u| \leq 1$
- S does not appear on the right-hand side of any production.

Chomsky-1-Languages

We define a new set of variables V' by

$$V' = V \cup \{r\} \cup \{A_{ab} \mid a, b \in V \cup \Sigma \cup \{r\}\}.$$

Intuition: A_{ab} is a nonterminal that combines the last two symbols ab in a sentential form into one symbol.

The new production set P' of the grammar G' consists of the productions on the next slide.

In all cases, $a, b, c, d \in V \cup \Sigma \cup \{r\}$ and $x, y \in (V \cup \Sigma \cup \{r\})^*$.

- $S \to \varepsilon \quad$ if $\quad r \in L(G)$,
- $S \to A_{ab} \quad$ if $\quad S \Rightarrow_G^* ab$
- $xA_{ab} \to yA_{cd} \quad$ if $\quad (xab \to ycd) \in P$
- $xA_{ab} \to yA_{cb} \quad$ if $\quad (xa \to yc) \in P$
- $A_{ab} \to A_{ac} \quad$ if $\quad (b \to c) \in P$
- $A_{ab} \to aA_{cd} \quad$ if $\quad (b \to cd) \in P$
- all productions from P
- $A_{ar} \to a$ if $a \in \Sigma$

Then $G' = (V', \Sigma, P', S)$ is the required grammar. $\square$

Chomsky-1-Languages

Analogously, we prove the following:

Lemma 5

Let $G = (V, \Sigma \cup \{\ell\}, P, S)$ be a Type-1 grammar with $\ell \notin \Sigma$ and $L(G) \subseteq \ell\,\Sigma^*$. Then there exists a Type-1 grammar G' such that

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

By applying both lemmas:

Lemma 6

Let $G = (V, \Sigma \cup \{\ell, r\}, P, S)$ be a Type-1 grammar with $\ell, r \notin \Sigma$ and $L(G) \subseteq \ell\,\Sigma^*\,r$. Then there exists a Type-1 grammar G' such that

$$L(G') = \{w \in \Sigma^* \mid \ell w r \in L(G)\}.$$

Now, back to the proof of Theorem 3. Let $A = (Z, \Sigma, \Gamma, \delta, z_0, \Box, E)$ be an LBA.

Based on Lemma 6, it is sufficient to provide a Type-1 grammar for the language $\{\$w\Box \mid w \in T(A)\}$ (where \$ is a new terminal symbol).

To do so, we simulate A backwards using the Type-1 grammar $G = (V, \Sigma \cup \{\$, \Box\}, P, S)$ with the variable set

$$V = \{S, B, C\} \cup (\Gamma \setminus (\Sigma \cup \{\Box\})) \cup (Z \times \Gamma)$$

and the following production set P (Slides 346–348):

$$\begin{array}{lcl} S & \to & \$B \\ B & \to & aB \mid (z, a)C \mid (z, \Box) \quad \text{for all } a \in \Gamma \setminus \{\Box\}, z \in E \\ C & \to & aC \mid \Box \quad \text{for all } a \in \Gamma \setminus \{\Box\} \end{array}$$

Chomsky-1-Languages

With the rules for S, B, and C, one can generate any word of the form

$$\$a_1 a_2 \cdots a_n (z,a) b_1 b_2 \cdots b_m \Box \quad \text{or} \quad \$a_1 a_2 \cdots a_n (z, \Box)$$

with $a_1, \ldots, a_n, a, b_1, \ldots, b_m \in \Gamma \setminus \{\Box\}$ and $z \in E$.

These are exactly the configurations in which A accepts, except for the detail that we combine the state z and the currently read tape symbol a into a nonterminal $(z,a) \in Z \times \Gamma$ (which simplifies the rest of the grammar).

The following productions simulate the LBA A backwards:

$$\begin{array}{rcll}
(z',a') & \to & (z,a) & \text{for all } (z',a',N) \in \delta(z,a) \\
a'(z',b) & \to & (z,a)b & \text{for all } (z',a',R) \in \delta(z,a), b \in \Gamma \\
(z',b)a' & \to & b(z,a) & \text{for all } (z',a',L) \in \delta(z,a), b \in \Gamma
\end{array}$$

Using the productions from the previous slide, the initially generated accepting configuration eventually derives into an initial configuration of the form

$$\$(z_0, c_1)c_2 \cdots c_n\Box \quad \text{or} \quad \$(z_0, \Box)$$

(note: z_0 is the initial state of the LBA A). Then, using the following productions, the word $\$c_1c_2 \cdots c_n\Box$ or $\$\Box$ is derived:

$$\begin{aligned} \$(z_0, a) &\to \$a \quad \text{for all } a \in \Sigma \\ \$(z_0, \Box) &\to \$\Box \end{aligned}$$

Thus, we have $L(G) = \{\$w\Box \mid w \in T(A)\}$. $\Box$

Chomsky-0-Languages

Satz 7 (Turing Machines and Chomsky-0-Languages)

A language L is recognized by a nondeterministic Turing machine if and only if there exists a Type-0 grammar G such that $L = L(G)$.

Proof Idea: By modifying the proof of Theorem 3:

Grammars $\rightarrow$ Turing Machines: In this case, when simulating the grammar on the Turing machine tape, for reducing rules (the left side is longer than the right side), the tape contents must be shifted apart.

Turing Machines $\rightarrow$ Grammars: Here, it must be ensured that the grammar can generate spaces on both sides when simulating the Turing machine, and can also delete them after successful computation.

Formal: We simulate a Turing machine $M = (Z, \Sigma, \Gamma, \delta, z_0, \square, E)$ using the Type-0 grammar $G = (\{S, B, C, \$_1, \$_2\} \cup (\Gamma \setminus \Sigma) \cup (Z \times \Gamma),\ \Sigma, P, S)$ with the following production set P:

$$
\begin{array}{rcl}
S & \to & \$_1 B \\
B & \to & aB \mid (z,a)C \quad \text{for all } a \in \Gamma, z \in E \\
C & \to & aC \mid \$_2 \quad \text{for all } a \in \Gamma \\
(z',a') & \to & (z,a) \quad \text{for all } (z',a',N) \in \delta(z,a) \\
a'(z',b) & \to & (z,a)b \quad \text{for all } (z',a',R) \in \delta(z,a), b \in \Gamma \\
(z',b)a' & \to & b(z,a) \quad \text{for all } (z',a',L) \in \delta(z,a), b \in \Gamma \\
\$_1 \square & \to & \$_1 \\
\$_1 (z_0, a) & \to & a \quad \text{for all } a \in \Sigma \\
\square \$_2 & \to & \$_2 \\
a \$_2 & \to & a \quad \text{for all } a \in \Sigma \\
\$_1 (z_0, \square) \$_2 & \to & \varepsilon
\end{array}
$$

Chomsky-0-Languages

Again, the Turing machine M is simulated backwards.

The shortening rules $\$_1\Box \to \$_1$ and $\Box\$_2 \to \$_2$ allow blank symbols at the beginning and end of the configuration to be deleted.

This is important to derive, from an initial configuration obtained by backward simulation of the TM, initially:

$$\$_1\Box\cdots\Box(z_0, c_1)c_2\cdots c_n\Box\cdots\Box\$_2 \quad \text{or} \quad \$_1\Box\cdots\Box(z_0, \Box)\Box\cdots\Box\$_2$$

to:

$$\$_1(z_0, c_1)c_2\cdots c_n\$_2 \quad \text{or} \quad \$_1(z_0, \Box)\$_2$$

Then, using the productions $\$_1(z_0, c_1) \to c_1$ and $c_n\$_2 \to c_n$ or $\$_1(z_0, \Box)\$_2 \to \varepsilon$, the input word $c_1c_2\cdots c_n$ or ε is derived.

Thus, $L(G) = T(M)$. $\Box$

Results for Chomsky-1 and Chomsky-0 Languages

Satz 8 (Closure under Complement of Type-1 Languages, Immerman, Szelepcsényi)

If L is a Type-1 language, then $\overline{L} = \Sigma^* \backslash L$ is also a Type-1 language.

A proof will be presented in the lecture **Structural Complexity Theory**.

Satz 9 (Non-closure under Complement of Type-0 Languages)

There exists a Type-0 language $L \subseteq \Sigma^*$ such that $\overline{L} = \Sigma^* \backslash L$ is not a Type-0 language.

Justification and examples in the lecture **Computability and Logic**.

Satz 10 (Determinism and Nondeterminism in Turing Machines)

For every nondeterministic Turing machine, there exists a deterministic Turing machine that accepts the same language.

Proof:

Let $M = (Z, \Sigma, \Gamma, \delta, z_0, \Box, E)$ be a nondeterministic Turing machine, i.e.,

$$\delta\colon (Z \setminus E) \times \Gamma \to 2^{Z \times \Gamma \times \{L,R,N\}}.$$

Idea: We construct a deterministic Turing machine that, given input $x \in \Sigma^*$, systematically searches for a successful computation of M.

Let $\# \notin Z \cup \Gamma$ be a new symbol.

Results for Chomsky-1 and Chomsky-0 Languages

A successful computation of M on input x is a word of the form

$$k_0 \# k_1 \# \cdots k_{m-1} \# k_m$$

with the following properties:

1. $k_0, k_1, \ldots, k_m \in \Gamma^* Z \Gamma^+$
2. $k_0 = z_0 x \Box$.
3. $\forall i \in \{0, 1, \ldots m-1\} : k_i \vdash_M k_{i+1}$
4. $k_m \in \Gamma^* E \Gamma^+$

Clearly, $x \in T(M)$ if and only if a successful computation of M on input x exists.

A deterministic Turing machine M' can, given input x and $w \in (Z \cup \Gamma \cup \{\#\})^*$, check whether w is a successful computation of M on input x (this can even be done with a deterministic LBA).

To do this, M' only needs to check the four properties (1)–(4).

Now, we just need to construct a deterministic Turing machine M'' that systematically goes through all words $w \in (Z \cup \Gamma \cup \{\#\})^*$ and, each time, (using M') checks whether w is a successful computation of M on input x.

"Systematically in order" can be formally defined here using a length-lexicographical order.

Let $\sqsubset$ be any linear order on the alphabet $\Omega = Z \cup \Gamma \cup \{\#\}$.

The length-lexicographical order $\sqsubset_{\text{lex}}$ on Ω^* corresponding to $\sqsubset$ is defined as follows:

For $u, v \in \Omega^*$, we have $u \sqsubset_{\text{lex}} v$ if and only if

- $|u| < |v|$ (i.e., u is shorter than v), or
- $|u| = |v|$ and there exist $x, y, z \in \Omega^*$, $a, b \in \Omega$ such that $u = xay$, $v = xbz$, and $a \sqsubset b$ (i.e., at the first position where u and v differ, u has the smaller symbol).

Results for Chomsky-1 and Chomsky-0 Languages

General structure of the deterministic Turing machine M'':

1. Initialize a word $w \in (Z \cup \Gamma \cup \{\#\})^*$ with ε behind the input x on the tape.
2. Check using M' whether w is a successful computation of M on input x.
   If yes, transition to an accepting state; otherwise, proceed to (3).
3. Increment w, i.e., overwrite w with the next word w' in the length-lexicographical order (formally: w' is the smallest word with respect to $\sqsubset_{\text{lex}}$ such that $w \sqsubset_{\text{lex}} w'$).
4. Go to (2). $\square$

Determinism and Nondeterminism for LBAs (First LBA Problem)

It is not known whether for every LBA A, there exists a deterministic LBA A' with $T(A) = T(A')$.