Automata and Grammars
TIN071

Marta Vomlelová

marta@ktiml.mff.cuni.cz
http://ktiml.mff.cuni.cz/~marta

August 9, 2019
Organization

- **Lecture:**
  - you meet more than on slides
  - you may give me your feedback, influence the lecture

- **Exercises:**
  - you construct automata, languages needed also for the exam
  - to practice is very different than read
  - you need the credit.

- **Examination:**
  - Written exam with oral discussion
  - Understanding of lectures + ability to formalize
    - Exercises as you met and similar,
    - Write a definition, a theorem, the idea of a proof, an algorithm.
Exam Requirements

- **You need credits from the exercises before the exam.**
  - The form of study verification is a credit and an exam. Obtaining credit first is a necessary condition for taking an exam, with the exception of early exam terms. The credit is granted by teachers leading the tutorials based on point evaluation of tests during the semester, possible homeworks, activity etc. The nature of study verification for the credit excludes the possibility of its repetition.

- **The exam consists of written and oral parts.** The written part precedes the oral part. If the written part is evaluated as unsatisfactory, the exam ends as unsatisfactory without taking the oral part. If the oral part is evaluated as unsatisfactory, the student has to repeat both written and oral parts on the next exam term. Final exam grade is based on evaluation of both written and oral parts.

- **Written part** An example of the written part from previous years are available on the course webpage.

- **Oral part** Typical questions:
  - Characterize a given language w.r.t. Chomsky hierarchy.
  - Write down a definition, a theorem, the idea of the proof of a key theorem, apply a CYK algorithm.
J.E. Hopcroft, R. Motwani, J.D. Ullman: *Introduction to Automata Theory, Languages, and Computations*, Addison–Wesley

- Exercises

in Czech M. Chytil: *Automaty a gramatiky*, SNTL Praha, 1984
The lecture goal

- Get used to abstract 'computer' models,
- describe the model formally,
- understand small changes in the definition lead to significantly different classes
- get used to infinite objects

Automata and Grammars – two kinds of description

- Turing machines
  - multi-tape, nondeterministic
- lineary bounded automata
- pushdown automata
- finite automata
  - DFA, NFA, \( \lambda \)NFA
- grammars of Type 0
- context grammars
  - monotone grammars
- context–free grammars
- regular grammars
Practical usage

- Justification of correctness of a program, an algorithm, a translator
- natural language processing
- lexical analyzer
- syntax analyzer
- hardware design and verification
  - digital circuits
  - machines
  - automata
- software implementation
  - search of a word in text
  - system verification
- applications in biology

Automata and Complexity

- Automata are essential for the study of the limits of computation.
- What can computer do at all? **decidability**.
- What can computer do efficiently? **intractability**. Time of calculation as a function of the size of the input. Growing faster than any polynomial is deemed to grow too fast.
Finite Automata

- Software for designing and checking the behavior of digital circuits.

A Finite automaton modeling an on/off switch.

A finite automaton modeling recognition of `then`.

- Lexical analyzer, web page analyzer.
• NAIL062 Propositional and Predicate Logic

• **Deductive Proofs** (axioms, statements, Modus Ponens)

• **Contrapositive** \( \neg C \Rightarrow \neg H \) instead of \( H \Rightarrow C \)

• **Proof by Contradiction** \( H \) and \( \neg C \) implies falsehood \( \bot \).

• **Counterexamples** We prove that a statement is **not** true by presenting an example where the statement does not hold.

• **Inductive Proofs**
  - Prove the basis, usually \( i = 1 \) or single node tree.
  - Prove the inductive step:
    - For any \( n \), if \( S(n) \) then also \( S(n + 1) \).

• **Structural Induction** on trees.
Definition 1.1 (Deterministic Finite Automata)

A **deterministic finite automation (DFA)** $A = (Q, \Sigma, \delta, q_0, F)$ consists of:

- A finite set of **states**, often denoted $Q$.
- A finite set of **input symbols**, denoted $\Sigma$.
- A **transition function** $Q \times \Sigma \rightarrow Q$, denoted $\delta$, represented by arcs.
- A **start state** $q_0 \in Q$.
- A **set accepting states** (final states) $F \subseteq Q$.

**Convention:** If some transitions are missing, we add a new state $fail$ and make the transition $\delta$ total by adding edges to $fail$ for any 'undefined' pair $q, s$.

If the set $F$ is empty, we add to $F$ and $Q$ a new state $final$, with no transitions from other states, just 'staying in final' for any $s \in \Sigma$: $\delta(final, s) = final$. 

![DFA Diagram](image)
Example 1.1

An automaton $A$ that accepts $L = \{x01y : x, y \in \{0, 1\}^*\}$.

- State diagram (graph) Automaton $A = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_1\})$.

- Table
  
  \[
  \begin{array}{c|cc}
    & 0 & 1 \\
  \hline
  \delta & \rightarrow q_0 & q_2 \\
  \star q_1 & q_1 & q_1 \\
  q_2 & q_2 & q_1 \\
  \end{array}
  \]

- State tree
  
  - nodes = states
  - edges = transitions
  - only reachable states
  - we need it only for nondeterministic FA.
Definition 1.2 (Word, λ, ε, Σ*, Σ+, Language)

**Alphabet** is a finite, nonempty set of symbols, denoted Σ (or X).

- **String** (=word) is a finite sequence of symbols s chosen from some alphabet Σ. For example 0011010101; **empty string** λ (or ε).
- The set of all strings in the alphabet is denoted Σ*,
- the set of all non–empty strings $\Sigma^+ = \Sigma^* \setminus \{\lambda\}$.
- A **language** $L \subseteq \Sigma^*$ is a set of strings in the alphabet Σ.

Definition 1.3 (concatenation, power, length)

We define operations on strings from Σ*: 

- **concatenation** $u \cdot v$ or $uv$
- **powers** $u^n$ ($u^0 = \lambda$, $u^1 = u$, $u^{n+1} = u^n \cdot u$)
- **length** $|u|$ ($|\lambda| = 0$, $|auto| = 4$).
- **number of occurrences of** $s \in \Sigma$ in the string $u$ is denoted $|u|_s$ ($|zmrzlina|_z = 2$).
Definition 1.4 (Extended Transition Function to Strings)

If $\delta$ is our transition function, then the **extended transition function** $\delta^*$, $\delta^* : Q \times \Sigma^* \rightarrow Q$ (transitive closure of $\delta$) takes a state $q$ and a string $w$ and returns a state $p$ and is defined by induction:

- $\delta^*(q, \lambda) = q$.
- Let $x \in \Sigma$, $w \in \Sigma^*$. Then $\delta^*(q, wx) = \delta(\delta^*(q, w), x)$.

**Remark:** Any time $\delta$ is applied to strings, $\delta^*$ was intended.

$\delta^*(1100) = q_2$, $\delta^*(1100111111111001) = q_1$
Definition 1.5 (Language of the DFA)

- The **language of the DFA** $A$ is $L(A) = \{w : \delta^*(q_0, w) \in F\}$.
- String $w$ is **accepted, recognized** by the automaton $A$, iff $w \in L(A)$.
- A language $L$ can be **recognized** by a DFA, if there exists a DFA $A$ such that $L = L(A)$.
- Languages accepted by DFA’s (and FA’s) are called **regular languages**. The set of regular languages is denoted by $\mathcal{F}$.

Example 1.2 (A regular language)

- $L = \{w | w = xux, w \in \{0, 1\}^*, x \in \{0, 1\}, u \in \{0, 1\}^*\}$. 
Example 1.3 (Regular Language)

- \( L = \{ w \mid w = ubaba, \ w \in \{a, b\}^*, \ u \in \{a, b\}^* \} \).

Example 1.4 (Regular Language)

- \( L = \{ w \mid w \in \{0, 1\}^* \& w \text{ binary encoding of a number dividible by 5} \} \).

Example 1.5 (A language that is not regular)

- \( L = \{ 0^n1^n \mid w \in \{0, 1\}^*, \ n \geq 1 \} \) is not regular.
Theorem 1.1 (Pumping Lemma For Regular Languages)

Let $L$ be a regular language. Then there exists a constant $n \in \mathbb{N}$ (which depends on $L$) such that for every string $w \in L$ such that $|w| \geq n$, we can break $w$ into three strings, $w = xyz$, such that:

- $y \neq \lambda$.
- $|xy| \leq n$.
- For all $k \in \mathbb{N}_0$, the string $xy^kz$ is also in $L$.

Example 1.6

- $abbbba = a(b)bbba$; $\forall i \geq 1; a(b)^i bbba \in L(A)$.
- $aaaaba = (aaa)aba$; $\forall i \geq 1; (aaa)^i aba \in L(A)$.
- $aa$, a short word, $|aa| < n$. 

Diagram:

```
0 -> a -> 1 -> a -> 2
  |            |            |
  b            b            b
```

Automata and Grammars  Introduction, Pumping lemma for FA  August 9, 2019 15 / 1 - 24
Proof.

Suppose $L$ is regular, then $L = L(A)$ for some DFA $A$ with $n$ states.

Take any string $w \in L$, $w = a_1a_2 \ldots a_m$ of length $m \geq n$, $a_i \in \Sigma$.

Define $\forall i \ p_i = \delta^\ast(q_0, a_1a_2 \ldots a_i)$. Note $p_0 = q_0$.

We have $n + 1$ $p_i$’s and $n$ states, therefore there are $i, j$ such that $0 \leq i < j \leq n : p_i = p_j$.

Define: $x = a_1a_2 \ldots a_i$, $y = a_{i+1}a_{i+2} \ldots a_j$, $z = a_{j+1}a_{j+2} \ldots a_m$. Note $w = xyz$.

$$y = a_{i+1}a_{i+2} \ldots a_j$$

The loop above $p_i$ can be repeated any number of times and the input is also accepted.
Example 1.7 (The Pumping Lemma as an Adversarial Game)

The language \( L_{eq} = \{ w; |w|_0 = |w|_1 \} \) of all strings with an equal number of 0’s and 1’s is not regular language.

Proof.

- Suppose it is regular. Take \( n \) from the pumping lemma.
- Pick \( w = 0^n1^n \in L_{eq} \).
- Break \( w = xyz \) as in the pumping lemma, \( y \neq \lambda, |xy| \leq n \).
- Since \( |xy| \leq n \) and it comes at front of \( w \), it consists only of 0’s. The pumping lemma says: \( xz \in L_{eq} \) (for \( k = 0 \)). However, it has less 0’s and the same amount of 1’s as \( w \), so one of them must not be in \( L_{eq} \).

Example 1.8

The language \( L = \{ 0^i1^i; i \geq 0 \} \) is not regular.
Example 1.9

The language $L_{pr}$ of all strings of 1's whose length is a prime is not a regular language.

Proof.

- Suppose it were. Take a constant $n$ from the pumping lemma. Consider some prime $p \geq n + 2$, let $w = 1^p$.
- Break $w = xyz$ by the pumping lemma, let $|y| = m$. Then $|xz| = p - m$.
- $xy^{p-m}z \in L_{pr}$ by pumping lemma, but
  
  $$|xy^{p-m}z| = |xz| + (p - m)|y| = p - m + (p - m)m = (m + 1)(p - m)$$

  that is not a prime (none of two factors are 1).

Example 1.10 (Non–regular language that can be 'pumped')

The language $L = \{u | u = a^+b^ic^i \lor u = b^ic^j\}$ is not regular (Myhill–Nerode theorem), but the first symbol can be always pumped.
Summary

- **Definition**
  - Deterministic Finite Automaton $A = (Q, \Sigma, \delta, q_0, F)$
  - Language $L \subseteq \Sigma^*$
  - The Language accepted by the DFA $A$: $L(A) = \{w \mid w \in \Sigma^* \land \delta^*(q_0, w) \in F\}$
  - the class of Regular Languages

- **Pumping lemma for regular languages**
- Example proof on non-regularity of the language $L = \{w|0^i1^i, i \in \mathbb{N}\}$
- Examples of regular languages.
Example 1.11

$L = \{w \mid w \in \{0, 1\}^*, |w|_0 = 2k \& |w|_1 = 2\ell, k, \ell \in \mathbb{N}_0\}$. DFA accepting all and only strings with an even number of 0’s and an even number of 1’s.
Three parties: the customer, the store, the bank.

Only one 'money' file (for simplicity).

Example 1.12
Customer may decide to transfer money to the store, which will the redeem the file from the bank and ship goods to the customer. The customer has the option to cancel the file.

Five events:
- Customer may pay.
- Customer may cancel.
- Store may ship the goods to the customer.
- Store may redeem the money.
- The bank may transfer the money by creating a new, suitably encrypted money file and sending it to the store.
(Incomplete) Finite Automata for the Bank Example

Customer

Bank

Store

pay
re redeem
transfer
ship
redeem
transfer
ship
ship
cancel
Formally, the automaton should perform an action for any input. The automaton for the store needs an additional arc from each state to itself, labeled *cancel*.

Customer must not kill the automaton by execution *pay* again, so loops with *pay* label are necessary. Similarly with other commands.

\[
\text{pay, ship, redeem, cancel}
\]

---

**Extended Automaton for the Bank**
Product Automaton

- Product automaton of Bank and Store has states pairs $B \times S$.
- To construct the arc of the product automaton, we need to run the bank and store automata 'in parallel'. If any automaton dies, the product dies too.

![Diagram of product automaton]
**Finite Automata, Regular Languages**

- **Deterministic Finite Automaton (DFA)**
  \[ A = (Q, \Sigma, \delta, q_0, F) \].

- **Language accepted (recognized) by a DFA**
  \[ A = (Q, \Sigma, \delta, q_0, F) \] is the language
  \[ L(A) = \{ w \mid w \in \Sigma^* \land \delta^*(q_0, w) \in F \} \].

- Language \( L \) is **recognizable** by a DFA, if there exists DFA \( A \) such that \( L = L(A) \).

- The class of languages recognizable by a DFA \( \mathcal{F} \) is called **regular languages**.

- Finite automaton encodes only finite information.

- It can recognize infinite languages.
Is a given language regular?

- **YES** Construct an automaton.
- **NO** Find the contradiction with Myhill–Nerode theorem or with the Pumping Lemma.
Definition 2.1 (congruence)

Let us have a finite alphabet $\Sigma$ and an equivalence relation $\sim$ on $\Sigma^*$ (reflexive, symmetric, transitive). Then:

- $\sim$ is a **right congruence** iff
  $$(\forall u, v, w \in \Sigma^*) u \sim v \Rightarrow uw \sim vw.$$  
- $\sim$ has a **finite index** iff the partition $\Sigma^*/\sim$ has a finite number of classes.
- The class of congruence $\sim$ that contains the word $u$ is denoted $[u]_{\sim}$ or simply $[u]$.

Theorem 2.1 (Mihyl–Nerode theorem)

Let us have a language $L \subset \Sigma^*$. The following statements are equivalent:

a) $L$ can be recognized by DFA,

b) there exists a right congruence $\sim$ with a finite index over $\Sigma^*$ such that $L$ is a union of some classes of the partition $\Sigma^*/\sim$. 
Proof: Proof of the Myhill–Nerode Theorem

a)⇒b); from an automaton ⇒ right congruence of a finite index
- we define $u \sim v \equiv \delta^*(q_0, u) = \delta^*(q_0, v)$.
- it is an equivalence
- it is a congruence (from the definition of $\delta^*$)
- it has a finite index ($Q$ is finite)
- $L = \{w|\delta^*(q_0, w) \in F\} = \bigcup_{q \in F}\{w|\delta^*(q_0, w) = q\} = \bigcup_{q \in F}[w|\delta^*(q_0, w) = q]_{\sim}$.

b)⇒a); from the right congruence with a finite index ⇒ automaton
- we set the alphabet $\Sigma$
- states $Q$ are the congruence classes $\Sigma^*/_{\sim}$
- the initial state $q_0 \equiv [\lambda]$
- final states $F = \{c_1, \ldots, c_n\}$ where $L = \bigcup_{i=1,\ldots,n} c_i$
- transition function $\delta([u], x) = [ux]$ (it is properly defined, right congruence).
- $L(A) = L$

$w \in L \iff w \in \bigcup_{i=1,\ldots,n} c_i \iff w \in c_1 \lor \ldots w \in c_n \iff [w] = c_1 \lor \ldots [w] = c_n \iff [w] \in F \iff w \in L(A)$

$\delta^*([\lambda], w) = [w]$
Example 2.1

Construct an automaton that accepts the language
$L = \{w | w \in \{a, b\}^* \& \ |w|_a = 3k + 2 \}$, words with $3k + 2$ symbols $a$.

we define $u \sim v \equiv (|u|_a \mod 3 = |v|_a \mod 3)$
equivalence classes are 0, 1, 2
$L$ corresponds to the class 2
$a$ – transitions to the next class
$b$ – stay in the same class.
Example 2.2 (Non–regular language)

The language \( L = \{ u | u = a^+ b^i c^i \lor u = b^i c^j \} \) is not regular.

- Assume \( L \) was regular
  - there must exist a right congruence \( \sim_L \) with a finite index \( m \), where \( L \) is a union of some classes \( \Sigma^*/\sim_L \)
  - we take the set of words \( S = \{ ab, abb, abbb, \ldots, ab^n, \ldots \} \), \( n \in \mathbb{N} \)
  - for any two \( i \neq j \) there is a string \( (c^i) \) distinguishing the words to/out of the language \( ab^i c^i \in L \& ab^j c^j \notin L \)
  - none two elements of \( S \) can be in the same class of \( \sim_L \) (\( L \) would split the class)
  - \( S \) is infinite, \( \sim_L \) should have a finite index
  - it is a contradiction with the 'finite index' in Mihyll-Nerode theorem.
Theorem (Pumping Lemma For Regular Languages)

Let $L$ be a regular language. Then there exists a constant $n \in \mathbb{N}$ (which depends on $L$) such that for every string $w \in L$ such that $|w| \geq n$, we can break $w$ into three strings, $w = xyz$, such that:

- $y \neq \lambda$.
- $|xy| \leq n$.
- For all $k \geq 0$, the string $xy^kz$ is also in $L$. 
Theorem 2.2

Regular language $L$ is infinite iff there exists $u \in L; n \leq |u| < 2n$, such that $n$ is the number from the pumping lemma.

Proof:

$\leftarrow$ If $(\exists u \in L) \ n \leq |u| < 2n$, then we can pump $u$, $(\forall i \in \mathbb{N}) u^i \in L$, infinite number of elements $L$.

$\implies$ Language $L$ is infinite, it contains a string $w$ such that $n \leq |w|$.

- If $|w| < 2n$, we are done.
- Otherwise, from the Pumping lemma $w = xyz$ and $xz \in L$, we have a shorter word.
- If $2n \leq |xz|$, we shorten $xz$ further (Pumping lemma again).
- Each shortening cuts no more than $n$ symbols therefore we meet the interval $[n, 2n)$ (we do not jump over it).

To check language (in)finity it is sufficient to check a finite number of strings $\{u: \text{such that } n \leq |u| < 2 \cdot n\}$. 
**Definition 2.2 (Reachable states)**

Let’s have a DFA \( A = (Q, \Sigma, \delta, q_0, F) \) and \( q \in Q \). The state \( q \) is **reachable** iff there exists \( w \in \Sigma^* \) such that \( \delta^*(q_0, w) = q \).

**Algorithm: Reachable States**

We search reachable states iteratively.
- **Start:** \( M_0 = \{ q_0 \} \).
- **Repeat:** \( M_{i+1} = M_i \cup \{ q \mid q \in Q, (\exists p \in M_i, \exists x \in \Sigma) \delta(p, x) = q \} \)
- **while:** \( M_{i+1} \neq M_i \).

**Proof: Correctness and completeness**

- **Correctness:** \( M_0 \subseteq M_1 \subseteq \ldots \subseteq Q \) and any \( M_i \) contains only reachable states.
- **Completeness:**
  - assume \( q \) is reachable, that is \( (\exists w \in \Sigma^*) \delta^*(q_0, w) = q \)
  - we take the shortest \( w = x_1 \ldots x_n \) such that \( \delta^*(q_0, x_1 \ldots x_n) = q \)
  - obviously \( \delta^*(q_0, x_1 \ldots x_i) \in M_i \) (more precisely \( M_i \setminus M_{i-1} \))
  - therefore \( \delta^*(q_0, x_1 \ldots x_n) \in M_n \), therefore \( q \in M_n \).
**Definition 2.3 (Automata Equivalence)**

Finite automata $A, B$ are **equivalent** iff they recognize the same language, that is $L(A) = L(B)$.

**Definition 2.4 (Automata Homomorphism)**

Let $A_1, A_2$ are two DFA. A mapping $h : Q_1 \to Q_2$ is an (**automata**) homomorphism iff:

- $h(q_{01}) = q_{02}$ initial state maps to the initial state
- $h(\delta_1(q, x)) = \delta_2(h(q), x)$ transition function correspondence
- $q \in F_1 \iff h(q) \in F_2$ final states correspondence.

Homomorfismus, that is a bijection, is called **isomorphism**.

**Theorem (Automata Equivalence Theorem)**

*If there exists a homomorphism of automata $A_1$ and $A_2$, then $A_1$ and $A_2$ are equivalent.*
Theorem 2.3 (Automata Equivalence Theorem)

If there exists a homomorphism of automata $A_1$ and $A_2$, then $A_1$ and $A_2$ are equivalent.

Proof:

- For any string $w \in \Sigma^*$, $q \in Q_1$ we prove by finite iteration
  - $h(\delta_1^*(q, w)) = \delta_2^*(h(q), w)$
- further:
  - $w \in L(A_1) \iff \delta_1^*(q_{01}, w) \in F_1$  
  - $\iff h(\delta_1^*(q_{01}, w)) \in F_2$
  - $\iff \delta_2^*(h(q_{01}), w) \in F_2$
  - $\iff \delta_2^*(q_{02}, w) \in F_2$
  - $\iff w \in L(A_2)$
Ambiguity

There exist different automata accepting the same language $L$.

- The language $L = \{ w | w \in \{1\}^* \text{ and } |w| = 3k \}$. 

![Diagram of automata accepting the language $L$]
Definition 2.5 (State Equivalence)

States $p, q \in Q$ of a FA $A$ are **equivalent** iff:

- For all strings $w$ holds: $\delta^*(p, w) \in F$ iff $\delta^*(q, w) \in F$.

If states are not equivalent we call them **distinguishable**.

Example 2.3

Automaton in the figure right:

- $C, G$ are distinguishable, $\delta^*(C, \lambda) \in F$ a $\delta^*(G, \lambda) \notin F$.
- $A, G$: $\delta^*(A, 01) = C$ is accepting, $\delta^*(G, 01) = E$ is not accepting.
- $A, E$ are equivalent – $\lambda$, $1^*$ obvious, $0$ transits to non–accepting states, $01$ and $00$ meet in the same state.

Theorem 2.4

*State equivalence is transitive.*
Algorithm: Equivalent States Recognition for a DFA

Following algorithm finds distinguishable states:

- **Basis:** If \( p \in F \) is accepting and \( q \notin F \) is not, the pair \( \{p, q\} \) is distinguishable.
- **Induction:** Let \( p, q \in Q \), \( a \in \Sigma \) and the pair \( r, s; r = \delta(p, a) \) and \( s = \delta(q, a) \) is distinguishable. Then also \( \{p, q\} \) are distinguishable.

![Diagram of DFA]

The cross denotes distinguishable pairs. C is distinguishable immediately, other except \( \{A, G\}, \{E, G\} \) too. We can see three pairs of equivalent states.
Distinguishable states

Accepting and non–accepting states

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>G</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>H</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

1st step1: \( \delta(q, 1) \in \mathcal{F} \) for \( q \in \{B, C, H\} \)

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>H</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

1st step0 \( \delta(q, 0) \in \mathcal{F} \) for \( q \in \{D, F\} \)

B and G are distinguishable, \( \delta(A, 0) = B, \delta(G, 0) = H \), therefore A,G are distinguishable. Similarly, \( \delta(\ast, 0) \) for E,G goes to distinguishable states H,G.

Three pairs of equivalent states remain.
Theorem 2.5

If two states are not distinguished by previous algorithm, they are equivalent.

Proof: Algorithm Correctness

- Consider bad pairs of states that are distinguishable but not distinguish by the algorithm.
- Take the pair \( p, q \) that can be distinguished by the shortest string \( w = a_1 \ldots a_k \).
- States \( r = \delta(p, a_1) \) and \( s = \delta(q, a_1) \) are distinguishable by a shorter string \( a_2 \ldots a_k \), therefore \( r, s \) is not a bad pair. They must be distinguished by the algorithm.
- Then, in the next step, the algorithm distinguishes also \( p, q \).

The time is polynomial function of the number of states \( n \).
- In one iteration, we consider all pairs, that is \( O(n^2) \).
- In each iteration we add a cross, that means no more than \( O(n^2) \) iterations.
- Together, \( O(n^4) \).

The algorithm may be speeded up to \( O(n^2) \) by memorizing states that depend on the pair \( \{r, s\} \) and following the list backwards.
**Testing the Equivalence of Regular Languages**

**Algorithm: Testing the Equivalence of Regular Languages**

The equivalence of regular languages $L, M$ may be tested as follows:

- Find DFA $A_L, A_M$ recognizing them, $L(A_L) = L, L(A_M) = M$, $Q_L \cap Q_M = \emptyset$.
- Construct a DFA $(Q_L \cup Q_M, \Sigma, \delta_L \cup \delta_M, q_L, F_L \cup F_M)$ as a union of states and transitions; select one (any) initial state.
- Languages are equivalent iff initial states of original DFA’s are equivalent.

---

**Example 2.4**

Language $L = \{\lambda\} \cup \{0, 1\}^*0$ accepting empty string and strings ending with 0. Two DFA’s accepting $L$ and a table of distinguishable states.

**Diagram:**

- DFA $A$: States $A, B, C, D, E$ with transitions:
  - $A \xrightarrow{0} A$
  - $A \xrightarrow{1} B$
  - $B \xrightarrow{0} B$
  - $B \xrightarrow{1} A$
  - $C \xrightarrow{0} D$
  - $C \xrightarrow{1} C$
  - $D \xrightarrow{0} C$
  - $D \xrightarrow{1} D$
  - $E \xrightarrow{1} E$

- DFA $B$: States $B, C, D, E$ with transitions:
  - $B \xrightarrow{x} E$
  - $C \xrightarrow{x} E$
  - $D \xrightarrow{x} E$
  - $E \xrightarrow{x} E$

**Table:**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>
Definition 2.6 (minimum–state DFA)

A DFA is **minimum–state (reduced)** DFA, iff
- has no non–reachable states and
- no two states are equivalent.

Theorem 2.6 (DFA Minimization)

*For any DFA A there exist an equivalent DFA B that is minimum–state (reduced) DFA.*
Algorithm: Algorithm for minimum automaton for DFA $A$

An algorithm to find a minimum–state DFA equivalent with a DFA $A$

- Eliminate from DFA $A$ all non–reachable states.
- Find a partition of remaining states to equivalence classes.
- Construct DFA $B$ with equivalence classes as states. We denote the transition function of $B$ $\delta_B$. Let us have $S \in Q_B$. For any $q \in S$ we define $\delta_B(S, a) = [\delta_A(q, a)]$ the equivalence class of $\delta_A(q, a)$. This class is the same for all states $a \in S$ since they are equivalent.
- Initial state $q_{0B}$ is the class containing the initial state $q_{0A}$.
- The set of accepting states $F_B$ are equivalence classes containing accepting states $F_A$. 

Automata and Grammars

Pumping Lemma, DFA minimization and equivalence 2

August 9, 2019  43 / 25 - 74
Example of minimal–state DFA construction

Equivalence classes:

\{A, E\}, \{B, H\}, \{C\}, \{F\}, \{G\}
A NFA can be in several states at once. It has an ability to ‘guess’ something about input.

A NFA accepting all strings that end in 01.

NFA processes input 00101.
Definition 2.7 (Nondeterministic Finite Automata)

A **nondeterministic finite automation (NFA)** $A = (Q, \Sigma, \delta, q_0, F)$ consists of:

- A finite set of **states**, often denoted $Q$.
- A finite set of **input symbols**, denoted $\Sigma$.
- A **transition function** $\delta : Q \times \Sigma \rightarrow \mathcal{P}(Q)$ returns a subset of $Q$.
- A **start state** $q_0 \in Q$.
- A **set accepting states** (final states) $F \subseteq Q$.

Example 2.5

The NFA from previous slide is $A = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_2\})$.

\[
\begin{array}{c|c|c}
\delta & 0 & 1 \\
\hline 
\rightarrow q_0 & \{q_0, q_1\} & \{q_0\} \\
q_1 & \emptyset & \{q_2\} \\
* q_2 & \emptyset & \emptyset \\
\end{array}
\]
Definition 2.8 (Extended Transition Function to Strings)

If $\delta$ is our transition function, then the extended transition function $\delta^*$, $\delta^*: Q \times \Sigma^* \rightarrow \mathcal{P}(Q)$ takes a state $q$ and a string $w$ and returns a set of states $\subseteq Q$ and is defined by induction:

$\delta^*(q, \lambda) = \{q\}$.

Let $w = xa$, $a \in \Sigma$, $x \in \Sigma^*$, suppose $\delta^*(q, x) = \{p_1, \ldots, p_k\}$. Let $\bigcup_{i=1}^k \delta(p_i, a) = \{r_1, r_2, \ldots, r_m\}$. Then $\delta^*(q, xa) = \{r_1, r_2, \ldots, r_m\}$.

First compute $\delta^*(q, x)$ and then follow any transition from any of these states that is labeled $a$.

\[
\begin{align*}
\delta^*(q_0, \lambda) &= \{q_0\} \\
\delta^*(q_0, 0) &= \delta(q_0, 0) = \{q_0, q_1\} \\
\delta^*(q_0, 00) &= \delta(q_0, 0) \cup \delta(q_1, 0) = \{q_0, q_1\} \\
\delta^*(q_0, 001) &= \delta(q_0, 1) \cup \delta(q_1, 1) = \{q_0, q_2\} \\
\delta^*(q_0, 0010) &= \delta(q_0, 0) \cup \delta(q_2, 0) = \{q_0, q_1\} \\
\delta^*(q_0, 00101) &= \delta(q_0, 1) \cup \delta(q_1, 1) = \{q_0, q_2\}
\end{align*}
\]
**Definition 2.9 (Language of an NFA)**

If \( A = (Q, \Sigma, \delta, q_0, F) \) is an NFA, then

\[
L(A) = \{ w | \delta^*(q_0, w) \cap F \neq \emptyset \}
\]

is the language accepted by NFA \( A \).

That is, \( L(A) \) is the set of strings \( w \in \Sigma^* \) such that \( \delta^*(q_0, w) \) contains at least one accepting state.

**Example 2.6**

The NFA from previous slide accepts the language \( L = \{ w | w \text{ ends in 01} \} \). The proof is a mutual induction:

- \( \delta^*(q_0, w) \) contains \( q_0 \) for every \( w \).
- \( \delta^*(q_0, w) \) contains \( q_1 \) iff \( w \) ends in 0.
- \( \delta^*(q_0, w) \) contains \( q_2 \) iff \( w \) ends in 01.
A non–deterministic FA in Figure may be simplified by removing the state C. States \{A, C\} are distinguishable by the input 0, therefore the previous algorithm will not find this reduction. (We may use exhaustive enumeration or reduce the NFA to DFA and find a minimal DFA for it.)
Mihyli–Nerode theorem
the use to prove a language is non–regular
there exist a language, that can be 'pumped' and is not regular
to check (in)finity of a regular language is possible by checking a finite number of strings

reachable states, an algorithm to identify them
equivalent automata, states
distinguishable states, an algorithm to identify them

automata homomorphism
Theorem: If there is a homomorphism from $A_1$ to $A_2$, they are equivalent.

minimum–state DFA, an algorithm to find equivalent minimum state DFA to a given DFA $A$

Regular languages are languages recognizable by finite automata.

Finite automata are either deterministic or nondeterministic – these can be in more states (in parallel).

Both characterizes the same class of languages. The nondeterministic make the description simpler.

There exists an algorithm transforming a NFA to a DFA.

Another extension: \( \lambda \)-transitions without reading any input symbol. Again, they accept only regular languages.
Definition 3.1 (Subset Construction)

The subset construction starts from an NFA $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$. Its goal is the description of an DFA $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ such that $L(N) = L(D)$.

- $Q_D$ is the set of subsets of $Q_N$, $Q_D = \mathcal{P}(Q_N)$ (the power set).
- Inaccessible states can be thrown away so the number of states may be smaller.
- $F_D = \{S : S \in \mathcal{P}(Q_N) \land S \cap F_N \neq \emptyset\}$, i.e. $S$ include at least one accepting state of $N$.
- For each $S \subseteq Q_N$ and for each input symbol $a \in \Sigma$,

$$\delta_D(S, a) = \bigcup_{p \in S} \delta_N(p, a).$$
Example of Subset Construction for language \((0 + 1)^*01\)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>\emptyset</td>
<td>\emptyset</td>
<td>\emptyset</td>
</tr>
<tr>
<td>{q_2}</td>
<td>\emptyset</td>
<td>\emptyset</td>
</tr>
<tr>
<td>{q_0, q_1}</td>
<td>{q_0, q_1}</td>
<td>{q_0, q_2}</td>
</tr>
<tr>
<td>{q_0, q_2}</td>
<td>{q_0, q_1}</td>
<td>{q_0}</td>
</tr>
<tr>
<td>{q_1, q_2}</td>
<td>\emptyset</td>
<td>{q_0, q_2}</td>
</tr>
<tr>
<td>{q_0, q_1, q_2}</td>
<td>{q_0, q_1}</td>
<td>{q_0, q_2}</td>
</tr>
</tbody>
</table>
Theorem 3.1 (DFA for any NFA)

If $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ is the DFA constructed from the NFA $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$ by subset construction, then $L(N) = L(D)$.

Proof.
By induction we prove: $\delta^*_D(\{q_0\}, w) = \delta^*_N(q_0, w)$.

Example 3.1 (A Bad Case for the Subset Construction)

A bad case for the subset construction is a language $L(N)$ of all strings of 0’s and 1’s such that the $n$th symbol from the end is 1. Intuitively, a DFA must remember the last $n$ symbols it has read.

Text search applications.
Finite Automata With $\lambda$–Transitions

- The new feature is that we allow a transition on $\lambda$, the empty string, that is without reading any input symbol.

Example 3.2 ($\lambda$ transition NFA)

(1) Any optional + or - sign,
(2) a string of digits,
(3) A decimal point, and
(4) another string of digits. At least one of strings (2) and (4) must be nonempty.
Definition 3.2 (λ-NFA)

λ-NFA is $E = (Q, \Sigma, \delta, q_0, F)$, where all components have their same interpretation as for NFA, except that $\delta$ is now a function that takes arguments $Q \times (\Sigma \cup \{\lambda\})$.

We require $\lambda \not\in \Sigma$, so no confusion results.

Example 3.3

Previous λ-NFA is: $E = (\{q_0, q_1, \ldots, q_5\}, \{., +, -, 0, 1, \ldots, 9\}, \delta, q_0, \{q_5\})$, where

<table>
<thead>
<tr>
<th></th>
<th>λ</th>
<th>+,-</th>
<th>.</th>
<th>0,1,…,9</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>${q_1}$</td>
<td>${q_1}$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$q_1$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>${q_2}$</td>
<td>${q_1, q_4}$</td>
</tr>
<tr>
<td>$q_2$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>${q_3}$</td>
</tr>
<tr>
<td>$q_3$</td>
<td>${q_5}$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>${q_3}$</td>
</tr>
<tr>
<td>$q_4$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>${q_3}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$q_5$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

$\delta$ is:
Definition 3.3 (λ-closure)

For $q \in Q$ we define the λ-closure $\lambda \text{CLOSE}(q)$ recursively, as follows:

- State $q$ is in $\lambda \text{CLOSE}(q)$.
- If $p \in \lambda \text{CLOSE}(q)$ and $r \in \delta(p, \lambda)$ then $r \in \lambda \text{CLOSE}(q)$.

For $S \subseteq Q$ we define $\lambda \text{CLOSE}(S) = \bigcup_{q \in S} \lambda \text{CLOSE}(q)$.
Definition 3.4

Suppose that $E = (Q, \Sigma, \delta, q_0, F)$ is an $\lambda$-NFA. We define $\delta^*$ as follows:

- $\delta^*(q, \lambda) = \lambda \text{CLOSE}(q)$.
- Suppose $w = \nu a$ where $a \in \Sigma, \nu \in \Sigma^*$. Let $\delta^*(q, \nu) = \{p_1, \ldots, p_k\}$.
- Let $\bigcup_{i=1}^{k} \delta(p_i, a) = \{r_1, \ldots, r_m\}$.
- Then $\delta^*(q, w) = \lambda \text{CLOSE}(\{r_1, \ldots, r_m\})$.

Example 3.4

\begin{align*}
\delta^*(q_0, \lambda) &= \lambda \text{CLOSE}(q_0) = \{q_0, q_1\} \\
\delta^*(q_0, 5) &= \lambda \text{CLOSE}(\bigcup_{q \in \delta^*(q, \lambda)} \delta(q, 5)) = \lambda \text{CLOSE}(\delta(q_0, 5) \cup \delta(q_1, 5)) = \{q_1, q_4\} \\
\delta^*(q_0, .5) &= \lambda \text{CLOSE}(\delta(q_1, .) \cup \delta(q_4, .)) = \{q_2, q_3, q_5\} \\
\delta^*(q_0, .6) &= \lambda \text{CLOSE}(\delta(q_2, 6) \cup \delta(q_3, 6) \cup \delta(q_5, 6)) = \{q_3, q_5\}
\end{align*}
Definition 3.5 (Eliminating λ-Transition)

Given any λ-NFA $E = (Q_E, \Sigma, \delta_E, q_0, F_E)$, we define a DFA $D = (Q_D, \Sigma, \delta_D, q_D, F_D)$ that accepts the same language as $E$.

$Q_D \subseteq \mathcal{P}(Q_E)$, $\forall S \subseteq Q_E : \lambda \text{CLOSE}(S) \in Q_D$. Note that $\emptyset$ may be in $Q_D$.

$q_D = \lambda \text{CLOSE}(q_0)$.

$F_D = \{S | S \text{ is in } Q_D \text{ and } S \cap F_E \neq \emptyset\}$.

For $S \subseteq Q_D$, $a \in \Sigma$ define $\delta_D(S, a) = \lambda \text{CLOSE}(\bigcup_{p \in S} \delta(p, a))$.

Theorem 3.2 (Eliminating λ-Transition)

A language $L$ is accepted by some λ-NFA if and only if $L$ is regular.
Proof.

(IF) Just add $\delta(q, \lambda) = \emptyset$.

(Only-if) Taking the $D$ constructed above we prove $L(D) = L(E)$ by induction on the length of $w$.

- $|w| = 0$, then $w = \lambda$, we know $\delta^*_E(q_0, \lambda) = \lambda \text{CLOSE}(q_0)$, $q_D = \lambda \text{CLOSE}(q_0)$. For DFA, $\forall p \in Q_D : \delta^*_D(p, \lambda) = p$, therefore $\delta^*_D(q_D, \lambda) = \lambda \text{CLOSE}(q_0)$ and $\delta^*_E(q_0, \lambda) = \delta^*_D(q_D, \lambda)$.

- Suppose $w = va$, $a \in \Sigma$, $v \in \Sigma^*$, $\delta^*_E(q_0, v) = \delta^*_D(q_D, v)$. The recursive step is the same in definition of $\delta^*$ and in the definition of DFA eliminating $\lambda$ transitions.
Set operations on languages

**Definition 3.6 (Set operations on languages)**

Let us have two languages \( L, M \). We define following operations:

1. **union** \( L \cup M = \{w | w \in L \lor w \in M\} \)
   - Example: the language of words starting with \( a^i \) or in the form \( b^i c^j \).

2. **intersection** \( L \cap M = \{w | w \in L \land w \in M\} \)
   - Example: The language of words of even length that end by \( 'baa' \).

3. **difference** \( L - M = \{w | w \in L \land w \notin M\} \)
   - Example: the language of words of even length that do not end by \( 'baa' \).

4. **complement** \( \bar{L} = -L = \{w | w \notin L\} = \Sigma^* - L \)
   - Example: the language of words that do not end by \( 'a' \)

**Theorem 3.3 (de Morgan rules)**

It holds:

\[
L \cap M = \overline{L \cup M} \\
L \cup M = \overline{L \cap M} \\
L - M = L \cap \overline{M}.
\]

Proof from logic properties of \&, \lor, \neg.
Theorem 3.4 (Closure on set operations)

Let us have regular languages $L, M$. Following languages are also regular:

1. the union $L \cup M$
2. the intersection $L \cap M$
3. the difference $L - M$
4. the complement $\overline{L} = \Sigma^* - L$.

Proof: RL closure on the complement

Switch accepting and non-accepting states of the DFA $F = Q_A - F_A$. 

Example 3.5

The language $\{w; w \in \{0, 1\}^*01\}$
Proof: Intersection, Union, Difference

- We construct the product automaton,
  \[ Q = (Q_1 \times Q_2, \Sigma, \delta((p_1, p_2), x) = (\delta_1(p_1, x), \delta_2(p_2, x)), (q_{01}, q_{02}), F) \]
- intersection: \( F = F_1 \times F_2 \)
- union: \( F = (F_1 \times Q_2) \cup (Q_1 \times F_2) \)
- difference: \( F = F_1 \times (Q_2 - F_2) \).

Product automaton example (intersection). Words containing 0,1, both.
Example 3.6

Construct an automaton accepting words, that contain $3k + 2$ of 1’s and do not contain the substring 11.

- The direct construction is complicated.
- $L_1 = \{w | w \in \{0, 1\}^* \& |w|_1 = 3k + 2\}$
- $L_2 = \{w | u, v \in \{0, 1\}^* \& w = u11v\}$
- $L = L_1 - L_2$.

Example 3.7

The language $M$ containing non–equal number of 0’s and 1’s is not regular.

- If $M$ is regular, then $\overline{M}$ is also regular.
- We know $\overline{M}$ is not regular (pumping lemma).
Definition 3.7 (String Operations on Languages)

Let us have languages $L, M$. We define operations:

- **concatenation** of languages $L \cdot M = \{uv \mid u \in L \& v \in M\}$
- **powers** of languages $L^0 = \{\lambda\}$
  
  $L^{i+1} = L^i \cdot L$

- **positive iteration** $L^+ = L^1 \cup L^2 \ldots = \bigcup_{i \geq 1} L^i$
- **(general) iteration** $L^* = L^0 \cup L^1 \cup L^2 \ldots = \bigcup_{i \geq 0} L^i$
  
  that is $L^* = L^+ \cup \{\lambda\}$

- **reverse** of the language $L^R = \{u^R \mid u \in L\}$
  
  (=reverse) $(x_1x_2 \ldots x_n)^R = x_nx_{n-1} \ldots x_2x_1$

- **left quotient** $L$ with $M$ $M \setminus L = \{v \mid uv \in L \& u \in M\}$

- **left derivation** $L$ with $w$ $\partial_w L = \{w\} \setminus L$

- **(right) quotient** $L$ with $M$ $L/M = \{u \mid uv \in L \& v \in M\}$

- **right derivation** $L$ with $w$ $\partial^R_w L = L/\{w\}$. 
Theorem 3.5

Assume \( L, M \) to be regular languages. The language \( L \cdot M \) is also regular.

Proof:

We start with the DFA \( A_1 = (Q_1, \Sigma, \delta_1, q_1, F_1) \) and \( A_2 = (Q_2, \Sigma, \delta_2, q_2, F_2) \) such that \( L = L(A_1) \) and \( M = L(A_2) \).

We define a nondeterministic NFA \( B = (Q \cup \{q_0\}, \Sigma, \delta, q_0, F_2) \) where:

- \( Q = Q_1 \cup Q_2 \) (we assume different names of states, \( Q_1 \cap Q_2 = \emptyset \))
- \( \delta(q_0, \lambda) = \{q_1, q_2\} \) for \( q_1 \in F_1 \) (that is \( \lambda \in L(A_1) \))
- \( \delta(q_0, \lambda) = \{q_1\} \) for \( q_1 \notin F_1 \) (that is \( \lambda \notin L(A_1) \))
- \( \delta(q_0, x) = \emptyset \) for \( x \in \Sigma \)
- \( \delta(q, x) = \{\delta_1(q, x)\} \) for \( q \in Q_1 \) & \( \delta_1(q, x) \notin F_1 \) (we stay in \( A_1 \))
- \( \delta(q, x) = \{\delta_1(q, x), q_2\} \) for \( q \in Q_1 \) & \( \delta_1(q, x) \in F_1 \) (nondet. transit. from \( A_1 \) to \( A_2 \))
- \( \delta(q, x) = \{\delta_2(q, x)\} \) for \( q \in Q_2 \) (we stay in \( A_2 \))

Then, \( L(B) = L(A_1) \cdot L(A_2) \).


**Theorem 3.6**

Let us have a regular language $L$. The languages $L^*$, $L^+$ are also regular.

- Idea: repeated computation of $A = (Q, \Sigma, \delta, q_0, F)$
- that is: a nondeterministic decision whether restart or continue the evaluation
- a new state to accept $\lambda \in L^0$ (for $L^+$ without it or $\notin F$).

**Proof: Proof for $L^*$**

Let us take a DFA $A = (Q, \Sigma, \delta, q_0, F)$ such that $L = L(A)$.

We define nondeterministic NFA $B = (Q \cup \{q_B\}, \Sigma, \delta_B, q_B, F \cup \{q_B\})$ where:

- $\delta_B(q_B, \lambda) = \{q_0\}$ a new state $q_B$ to accept $\lambda$, we move to $q_0$
- $\delta_B(q_B, x) = \emptyset$ for $x \in \Sigma$
- $\delta_B(q, x) = \{\delta(q, x)\}$ if $q \in Q$ & $\delta(q, x) \notin F$ inside $A$
- $\delta_B(q, x) = \{\delta(q, x), q_0\}$ if $q \in Q$ & $\delta(q, x) \in F$ restart is possible

Then $L(B) = L(A)^* (q_B \in F_B), L(B) = L(A)^+ (q_B \notin F_B)$.  

**Theorem 3.7 (R Closed under Reverse)**

Assume $L$ is regular language. The language $L^R$ is also regular.

- Obviously $(L^R)^R = L$ therefore only one direction proof is needed.
- idea: reverse the edges in the state diagram; we get a NFA

**Proof:**

We start with the DFA $A = (Q, \Sigma, \delta, q_0, F)$ such that $L = L(A)$. We define a NFA $B = (Q \cup \{q_B\}, \Sigma, \delta_B, q_B, \{q_0\})$ where:

- $\delta_B(q, x) = \{ p | \delta(p, x) = q \}$ for $q \in Q$
- $\delta_B(q_B, \lambda) = F$, $\delta_B(q_B, x) = \emptyset$.
- For any word $w = x_1x_2 \ldots x_n$
  - $q_0, q_1, q_2, \ldots, q_n$ is an accepting computation $w$ in $A$
  \[ \Leftrightarrow \]
  - $q_B, q_n, q_{n-1}, \ldots, q_2, q_1, q_0$ is an accepting computation $w^R$ in $B$.

Remark. Sometimes $L$ or $L^R$ has much simpler accepting automaton.
Closed under Quotient

Theorem 3.8

If $L, M$ are regular languages so is $M \setminus L$ and $L/M$.

- Idea: $A_L$, start in states reachable by a word in $M$

Proof:

We start with a DFA $A_L = (Q_1, \Sigma, \delta_1, q_1, F_1)$ such that $L = L(A_1)$. We define NFA $B = (Q_1, \Sigma, \delta_1, S, F_1)$ where:

- $S = \{ q \mid q \in Q_1 \land \exists u \in M \delta_1(q_1, u) \}$
  - can be found algorithmically ($A_q = (Q_1, \Sigma, \delta_1, q_1, \{ q \})$, then $q \in S \iff L(A_q) \cap M \neq \emptyset$)

- $\nu \in M \setminus L$
  - $\iff \exists u \in Muv \in L$
  - $\iff \exists u \in M \exists q \in Q_1 \delta_1(q_1, u) \& \delta_1(q, \nu) \in F_1$
  - $\iff \exists q \in S \& \delta_1(q, \nu) \in F_1$
  - $\iff \nu \in L(B)$

$L/M = (M^R \setminus L^R)^R$
Definition 3.8 (RL – Algebraic Description of Regular Languages)

Assume $\Sigma$ is a finite alphabet. We denote $RL(\Sigma)$ the smallest set of languages that:

- contains the empty language $\emptyset$
- for any letter $x \in \Sigma$ RL contains the language $\{x\}$
- it is closed under union $A, B \in RL(\Sigma) \Rightarrow A \cup B \in RL(\Sigma)$
- it is closed under concatenation $A, B \in RL(\Sigma) \Rightarrow A.B \in RL(\Sigma)$
- it is closed under iteration $A \in RL(\Sigma) \Rightarrow A^* \in RL(\Sigma)$

Specifically:
\[
\begin{align*}
\{\lambda\} &\in RL(\Sigma) \quad \text{since } \{\lambda\} = \emptyset^* \\
\Sigma &\in RL(\Sigma) \quad \text{since } \Sigma = \bigcup_{x \in \Sigma} \{x\} \text{ (finite union)} \\
\Sigma^* &\in RL(\Sigma)
\end{align*}
\]

Theorem (Kleene)

Any language can be recognized by a deterministic finite automaton if and only if it is in $RL$ (over the same alphabet).
Proof: $RL \Rightarrow$ recognizable by a DFA

- trivial languages are recognizable by a DFA
- regular languages are closed under operations union, concatenation, iteration (previous theorems).
Lemma (Further Closure Properties without a proof)

\[
L \cdot \emptyset = \emptyset \cdot L = \emptyset \\
\{\lambda\} \cdot L = L \cdot \{\lambda\} = L \\
(L^*)^* = L^* \\
(L_1 \cup L_2)^* = L_1^*(L_2 \cdot L_1^*)^* = L_2^*(L_1 \cdot L_2^*)^* \\
(L_1 \cdot L_2)^R = L_2^R \cdot L_1^R \\
\partial_w(L_1 \cup L_2) = \partial_w(L_1) \cup \partial_w(L_2) \\
\partial_w(\Sigma^* - L) = \Sigma^* - \partial_w L \\
h(L_1 \cup L_2) = h(L_1) \cup h(L_2)
\]

Example 3.8 (Yet another Non–regularity Proof)

- \( L = \{w | w \in \{0, 1\}^*, |w|_1 = |w|_2\} \) is not regular since
- \( L \cap \{0^i1^j | i, j \geq 0\} = \{0^i1^i | i \geq 0\} \) is not regular (pumping lemma).
Definition (Algebraic definition of the Regular Languages)

For a non-empty finite alphabet $\Sigma$ we define $RJ(\Sigma)$ as the smallest class of languages, that:

- includes the empty language $\emptyset$
- for any symbol $x \in \Sigma$ it includes $\{x\}$
- is closed under the union $A, B \in RJ(\Sigma) \Rightarrow A \cup B \in RJ(\Sigma)$
- is closed under the concatenation $A, B \in RJ(\Sigma) \Rightarrow A.B \in RJ(\Sigma)$
- is closed under the iteration $A \in RJ(\Sigma) \Rightarrow A^* \in RJ(\Sigma)$.

Theorem (Kleene)

Any language is in $RJ(\Sigma)$ if and only if it is accepted by some finite automaton.

Regular languages are closed under

- union, intersection, complement
- concatenation, iteration, substitution, homomorphism, inverse homomorphism,
- revers, left and right quotient.
Alternative Proof of Kleene Theorem

Proof: Recognizable by a DFA ⇒ RL

We have an nondeterministic automaton $A = (Q, \Sigma, \delta, q_0, F)$ accepting the language $L(A)$. By the induction on the number of edges of $A$ we prove $L(A) \in RL(\Sigma)$.

- no edges – only languages $\emptyset$ and $\{\lambda\}$, from the definition and $\emptyset^*$.
- $(n + 1)$ edges
  - select one edge $p \xrightarrow{a} q$, that is $q \in \delta(p, a)$
  - we construct four automata without this edge ($\delta^\dagger$)
    - $A_1 = (Q, \Sigma, \delta^\dagger, S, F)$
    - $A_2 = (Q, \Sigma, \delta^\dagger, S, \{p\})$
    - $A_3 = (Q, \Sigma, \delta^\dagger, \{q\}, \{p\})$
    - $A_4 = (Q, \Sigma, \delta^\dagger, \{q\}, F)$

Then $L(A) = L(A_1) \cup (L(A_2).a).(L(A_3).a)^* L(A_4)$,

languages $L(A_1), L(A_2), L(A_3), L(A_4) \in RL(\Sigma)$ by induction hypothesis $(n$ edges).
Simplification of the automata design

\[ L \cdot \emptyset = \emptyset \cdot L = \emptyset \]
\[ \{ \lambda \} \cdot L = L \cdot \{ \lambda \} = L \]
\[ (L^*)^* = L^* \]
\[ (L_1 \cup L_2)^* = L_1^*(L_2 \cdot L_1)^* = L_2^*(L_1 \cdot L_2)^* \]
\[ (L_1 \cdot L_2)^R = L_2^R \cdot L_1^R \]
\[ \partial_w(L_1 \cup L_2) = \partial_w(L_1) \cup \partial_w(L_2) \]
\[ \partial_w(\Sigma^* - L) = \Sigma^* - \partial_w L \]

Proof of non-regularity

- \( L = \{ w | w \in \{0, 1\}^*, |w|_1 = |w|_2 \} \) is not regular since
- \( L \cap \{0^i1^j | i, j \geq 0 \} = \{0^i1^i | i \geq 0 \} \) is not regular (pumping lemma).
Definition 4.1 (Regular Expression (RegE), value of a RegE $L(\alpha)$)

Regular expressions $\alpha, \beta \in \text{RegE}(\Sigma)$ over a finite non-empty alphabet $\Sigma = \{x_1, x_2, \ldots, x_n\}$ and their value $L(\alpha)$ is defined by induction:

<table>
<thead>
<tr>
<th>expression $\alpha$</th>
<th>for</th>
<th>value $L(\alpha) \equiv [\alpha]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\emptyset$</td>
<td>empty expression</td>
<td>$L(\emptyset) = {}$ $\equiv \emptyset$</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>empty string</td>
<td>$L(\lambda) = {\lambda}$</td>
</tr>
<tr>
<td>$a$</td>
<td>$a \in \Sigma$</td>
<td>$L(a) = {a}$.</td>
</tr>
</tbody>
</table>

- Basis:
- Induction:

<table>
<thead>
<tr>
<th>expression $\alpha + \beta$</th>
<th>$L(\alpha + \beta) = L(\alpha) \cup L(\beta)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha \beta$</td>
<td>$L(\alpha \beta) = L(\alpha)L(\beta)$</td>
</tr>
<tr>
<td>$\alpha^*$</td>
<td>$L(\alpha^<em>) = L(\alpha)^</em>$</td>
</tr>
<tr>
<td>$(\alpha)$</td>
<td>$L((\alpha)) = L(\alpha)$</td>
</tr>
</tbody>
</table>

Remark: $\alpha^*$ may be used.

The class of regular expressions over $\Sigma$: $\text{RegE}(\Sigma)$ is the smallest class closed under operations above.
Example 4.1 (Regular Expressions)

The language of alternating 0’s and 1’s may be written:

either
\[(01)^* + (10)^* + 1(01)^* + 0(10)^*\]

or
\[(\lambda + 1)(01)^*(\lambda + 0)\].

The language \(L((0^*10^*10^*1)^*0^*) = \{w | w \in \{0, 1\}^*, |w|_1 = 3k, k \geq 0\}\).

Definition 4.2 (Precedence)

The star * is the operator with highest precedence, then concatenation ., the lowest precedence has the union +.

Theorem 4.1 (RegE and DFA \ Kleene theorem (a variant))

Any language recognizable by a DFA can be expressed by a regular expression.
Any language of a regular expression can be recognized by a \(\lambda\)-NFA (therefore also a DFA).
Example

\[ R_{12}^{(2)} = 1^*0(0 + 1)^* \]

\[
\begin{array}{c|c}
R_{11}^{(0)} & \lambda + 1 \\
R_{12}^{(0)} & 0 \\
R_{21}^{(0)} & \emptyset \\
R_{22}^{(0)} & \lambda + 0 + 1 \\
R_{11}^{(1)} & \lambda + 1 + (\lambda + 1)(\lambda + 1)^*(\lambda + 1) \\
R_{12}^{(1)} & 0 + (\lambda + 1)(\lambda + 1)^*0 \\
R_{21}^{(1)} & \emptyset + \emptyset(\lambda + 1)^*(\lambda + 1) \\
R_{22}^{(1)} & \lambda + 0 + 1 + \emptyset(\lambda + 1)^*0 \\
R_{11}^{(2)} & 1^* + 1^*0(\lambda + 0 + 1)^*\emptyset \\
R_{12}^{(2)} & 1^*0 + 1^*0(\lambda + 0 + 1)^*(\lambda + 0 + 1) \\
R_{21}^{(2)} & \emptyset + (\lambda + 0 + 1)(\lambda + 0 + 1)^*\emptyset \\
R_{22}^{(2)} & \lambda + 0 + 1 + (\lambda + 0 + 1)(\lambda + 0 + 1)^*(\lambda + 0 + 1) = (0 + 1)^* \\
\end{array}
\]
From a DFA to RegE

Let us have a DFA $A$, $Q_A = \{1, \ldots, n\}$ with $n$ states.
Let $R^{(k)}_{ij}$ be a regular expression, $L(R^{(k)}_{ij}) = \{w | \delta^* (i, w) = j\}$ the set of words transferring the state $i$ into $j$ in $A$ where no state with an index higher than $k$ is on the path.
We iteratively construct $R^{(k)}_{ij}$ pro $k = 0, \ldots, n$.

$k = 0, i \neq j$: $R^{(0)}_{ij} = a_1 + a_2 + \ldots + a_m$ where $a_1, a_2, \ldots, a_m$ are symbols on edges $i$ into $j$ (or $R^{(0)}_{ij} = \emptyset$ or $R^{(0)}_{ij} = a$ for $m = 0, 1$).

$k = 0, i = j$: loops, $R^{(0)}_{ii} = \lambda + a_1 + a_2 + \ldots + a_m$ where $a_1, a_2, \ldots, a_m$ are symbols on loops in $i$. 
Induction. We have \( \forall i, j \in Q \ R_{ij}^{(k)} \). We construct \( R_{ij}^{(k+1)} \).

\[
R_{ij}^{(k+1)} = R_{ij}^{(k)} + R_{(k+1)}^{(k)} \left( R_{(k+1)(k+1)}^{(k)} \right)^* R_{(k+1)j}^{(k)}
\]

- Paths from \( i \) into \( j \) not meeting \( (k + 1) \) are already in \( R_{ij}^{(k)} \).
- Paths from \( i \) into \( j \) through \( (k + 1) \) with possible loops can be expressed

\[
R_{i(k+1)}^{(k)} \left( R_{(k+1)(k+1)}^{(k)} \right)^* R_{(k+1)j}^{(k)}.
\]

Finally \( RegE = \bigoplus_{j \in F_A} R_{1j}^{(n)} \) the union over all accepting states \( j \).
Previous method may generate up to $4^n$ symbols.
Following algorithm sometimes avoids duplicity.
We allow regular expressions to annotate the graph (a transformation of the automaton).

State $s$ selected for elimination

After $s$ is eliminated.
A RegE from a DFA

For every accepting state \( q \in F \) we eliminate all states \( p \in Q \setminus \{q, q_0\} \).

For \( q \neq q_0 \) we take

\[
\text{RegE}(q) = (R + SU^* T)^* SU^*.
\]

For \( q = q_0 \) we take

\[
\text{RegE}(q) = R^*.
\]

And take the union (addition) over all accepting states:

\[
\text{RegE}(DFA) = \bigoplus_{q \in F} \text{RegE}(q).
\]
Example 4.2

DFA that accepts 1 at the second last or the third-last position.

We get RegE: $$(0 + 1)^*1(0 + 1) + (0 + 1)^*1(0 + 1)(0 + 1).$$

[Elimination Order]
We start by non-accepting nor initial nodes $q \notin F, q \neq q_0$. 
From a RegE to $\lambda$–NFA

By induction by the structure of $R$. In each step we construct $\lambda$-NFA $E$ that recognizes the same language $L(R) = L(E)$ with three additional properties:

1. Exactly one accepting state.
2. No edges into the initial state.
3. No edges from the accepting state.

**Basis:**
- Empty string $\lambda$
- Empty set $\emptyset$
- A single string $a$

**Addition $R + S$:**

**Concatenation $RS$:**

**Iteration $R^*$:**
Pattern search in the text

- Static text: we create indexes rather than using RegE.
- RegE are useful in the dynamic text (like news).

### Example 4.3 (Search for streets in addresses on the web)

<table>
<thead>
<tr>
<th>Street identification</th>
<th>Street\</th>
<th>St\</th>
<th>Avenue\</th>
<th>Ave\</th>
<th>Road\</th>
<th>Rd\</th>
</tr>
</thead>
<tbody>
<tr>
<td>the name before</td>
<td>’ [A-Z] [a-z] <em>( [A-Z] [a-z] )</em> ’</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>house number</td>
<td>[0-9]+[A-Z] ?</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>all together</td>
<td>’ [0-9]+[A-Z] ? [A-Z] [a-z] <em>( [A-Z] [a-z] )</em></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We are missing:
- Boulevard, Place, Way
- Streets without any identifier (almost all Czech streets)
- Street names with numbers.
- ...
Converting Among Representations

Converting NFA to DFA
- \(\lambda\) closure in \(O(n^3)\). Search \(n\) states multiplied by \(n^2\) arcs for \(\lambda\) transitions.
- Subset construction, DFA with possibly \(2^n\) states. For each state, \(O(n^3)\) time to compute transitions.

Converting DFA to NFA
- Just modify transition table by putting set-brackets around states and adding column for \(\lambda\) in the case of \(\lambda\)-NFA.

Automaton to Regular Expression Conversion
- \(O(n^34^n)\)

RegE to Automaton Conversion
- \(\lambda\)-NFA in the time \(O(n)\).
Definition 4.3 (String Substitution, (String) Homomorphism)

We have a finite alphabet $\Sigma$. For each $x \in \Sigma$ we have $\sigma(x)$ a language over the alphabet $Y_x$. Further, we define:

$$\sigma(\lambda) = \{\lambda\}$$
$$\sigma(u.v) = \sigma(u).\sigma(v)$$

- The mapping $\sigma : \Sigma^* \rightarrow P(Y^*)$ where $Y = \bigcup_{x \in \Sigma} Y_x$ is substitution.
- $\sigma(L) = \bigcup_{w \in L} \sigma(w)$
- e–free, $\lambda$–free substitution is a substitution where none $\sigma(x)$ contains $\lambda$.

For $w = a_1 \ldots a_n \in \Sigma^n$ $\sigma(w) = \sigma(a_1) \ldots \sigma(a_n)$.

Example 4.4 (substitution)

- $\sigma(0) = \{a^i b^j, i, j \geq 0\}$, $\sigma(1) = \{cd\}$
- $\sigma(010) = \{a^i b^j c d^k b^l, i, j, k, l \geq 0\}$
(String) Homomorphism

Definition 4.4 ((String) Homomorphism)

**Homomorphism** $h$ is a special case of a substitution where $h(x) = wx \forall x \in \Sigma$. If $\forall x : wx \neq \lambda$ is is **e–free** (\(\lambda\)-free) **homomorphism**.

**Inverse homomorphism** $h^{-1}(L) = \{w|h(w) \in L}\}$.

Example 4.5 (homomorphism)

The function $h$ defined by: $h(0) = ab$, and $h(1) = \lambda$ is a homomorphism. For example, $h(0011) = abab$.

For $L = 10^*1$ is $h(L) = (ab)^*$.

Theorem (Closure under homomorphism)

If language $L$ and all $\forall x \in \Sigma \sigma(x)$ are regular, so is also $\sigma(L), h(L), h^{-1}(L)$. 

- Automata and Grammars
- Regular Expressions, Kleene Theorem, Subst., Homom. 4
- August 9, 2019 88 / 75 - 95
Homomorphism preserves regularity

Theorem 4.2

If \( L \) is a regular language over alphabet \( \Sigma \), and \( h \) is a homomorphism on \( \Sigma \), then \( h(L) \) is also regular.

Proof.

Let \( L = L(R) \) for some regular expression \( R \). The proof is done by structural induction on sub-expressions \( E \) of \( R \): we claim \( L(h(E)) = h(L(E)) \).

- **Basis:** \( h(\{\lambda\}) = \lambda \), \( h(\emptyset) = \emptyset \). If \( E = a \) then \( L(E) = \{a\} \), so \( h(L(E)) = \{h(a)\} \). Thus, \( L(h(E)) = \{h(a)\} \).

- **Induction:**
  - **Union:** \( L(h(F + G)) = L(h(F) + h(G)) = L(h(F)) \cup L(h(G)) \) and \( h(L(F + G)) = h(L(F) \cup L(G)) = h(L(F)) \cup h(L(G)) \). Right sides are equal from inductive hypothesis therefore left sides also equal.
  - concatenation, closure proofs are similar.
**Inverse Homomorphism**

**Definition 4.5 (Inverse homomorphism)**
Suppose \( h \) is a homomorphism from some alphabet \( \Sigma \) to strings in another alphabet \( T \). Then \( h^{-1}(L) \) ‘\( h \) inverse of \( L \)’ is the set of strings \( w \) in \( \Sigma^* \) such that \( h(w) \) is in \( L \).

**Example 4.6**
Let \( L = (00 + 1)^* \), \( h(a) = 01 \) and \( h(b) = 10 \). We claim \( h^{-1}(L) = (ba)^* \).

Proof: \( h((ba)^*) \in L \) is easy to see. Other \( w \) generates isolated 0 (4 cases to consider).

A homomorphism applied in the forward and inverse direction.
Theorem 4.3

If $h$ is a homomorphism from alphabet $\Sigma$ to alphabet $T$, and $L$ is a regular language over $T$, then $h^{-1}(L)$ is also a regular language.

Proof.

The proof starts with a DFA $A$ for $L$. We construct a DFA for $h^{-1}(L)$.

1. For $A = (Q, T, \delta, q_0, F)$ we define
   $B(Q, \Sigma, \delta_B, q_0, F)$ where
   $\delta_B(q, a) = \delta^*(q, h(a))$ ($\delta^*$ operates on strings).
2. By induction on $|w|$, 
   $\delta^*_B(q_0, w) = \delta^*(q_0, h(w))$.
3. Therefore, $B$ accepts exactly those strings $w$ that are in $h^{-1}(L)$.
Visit every state example

**Example 4.7**

Suppose $A = (Q, \Sigma, \delta, q_0, F)$ is an DFA. The language $L$ of all strings $w$ in $\Sigma^*$ such that $\delta^*(q_0, w)$ is in $F$ and for every state $q \in Q$ there is some prefix $x_q$ of $w$ such that $\delta^*(q_0, x_q) = q$. This language is regular.

- $M = L(A)$ the language accepted by DFA $A$ in the usual way.
- $T$ We define a new alphabet $T$ of triples $\{[paq]; p, q \in Q, a \in \Sigma, \delta(p, a) = q\}$.
- $h$ We define the homomorphism $h([paq]) = a$ for all $p, q, a$.
- $L_1$ Language $L_1 = h^{-1}(M)$ is regular since $M$ is regular (DFA and inverse homomorphism).

- $h^{-1}(101)$ includes $2^3 = 8$ strings, like $[p1p][q0q][p1p] \in \{[p1p], [q1q]\} \{[p0q], [q0q]\} \{[p1p], [q1q]\}$.
- We construct $L$ from $L_1$ (next slide).
Enforce start at $q_0$. Define $$E_1 = \bigcup_{a \in \Sigma, q \in Q} \{[q_0aq]\} =$$ $$E_1 = \{[q_0a_1q_0], [q_0a_2q_1], \ldots, [q_0a_mq_n]\}.$$ Then, $L_2 = L_1 \cap L(E_1 \cdot T^*)$.

**L3** Adjacent states must equal. Define non-matching pairs $$E_2 = \bigcup_{q \neq r, p, q, r, s \in Q, a, b \in \Sigma} \{[paq][rbs]\}.$$ Define $L_3 = L_2 - L(T^* \cdot E_2 \cdot T^*)$.

**L3** It ends in accepting state since we started from $M$ language of accepting computations on the DFA $A$.

**L4** All states. For each state $q \in Q$, define $E_q$ be the regular expression that is the sum of all the symbols in $T$ such that $q$ appears in neither its first or last position. We subtract $L(E_q^*)$ from $L_3$. $$L_4 = L_3 - \bigcup_{q \in Q} \{E_q^*\}.$$ 

**L** Remove states, leave symbols. $L = h(L_4)$. We conclude $L$ is regular.

In brief:

$M = L(A)$

Inverse homomorphism

$L_1 h^{-1}(M) \subseteq \{[qap]\}^*$

Intersection with a RL

$L_2 + q_0$

Difference with a RL

$L_3 +$ adjacent states equal

Difference with a RL

$L_4 +$ all states on the path

Homomorphism

$L h([qap]) = a$
Lemma (Testing Emptiness of Regular Languages)

For finite automata, it is a question of graph reachability of any final state from the initial one. Reachability is $O(n^2)$.

Lemma

For regular expressions, we can convert it to $\lambda$-NFA in $O(n)$ time and then check reachability.

It can be done also by direct inspection:

- Basis: $\emptyset$ denotes empty language; $\lambda$ and $a$ are not empty.
- Induction:
  - $R = R_1 + R_2$ is empty iff both $L(R_1)$ and $L(R_2)$ are empty.
  - $R = R_1 R_2$ is empty iff either $L(R_1)$ or $L(R_2)$ is empty.
  - $R = R_1^*$ is never empty, including $\lambda$.
  - $R = (R_1)$ is empty iff $R_1$ is empty, since they are the same language.
Given a string \( w \); \( |w| = n \) and a regular language \( L \), is \( w \in L \)?

- **DFA**: Run automaton; if \( |w| = n \), suitable representation, constant time transitions, it is \( O(n) \).
- **NFA with \( s \) states**: running time \( O(ns^2) \). Each input symbol can be processed by taking the previous set of states, which numbers at most \( s \) states.
- **\( \lambda \)-NFA**: first compute \( \lambda \)-closure. Then, for each symbol proceed it and compute \( \lambda \)-closure of the result.
- For a regular expression of size \( s \) we convert it to an \( \lambda \)-NFA with at most \( 2s \) states and then simulate, taking \( O(ns^2) \).
Further Generalisation

- Finite automaton makes following actions:
  - read a symbol
  - changes its state
  - moves its reading head to the right
- The head is not allowed to move to the left.

- What happens, if we allow the head to move left and right?
- **The automaton does not write anything on the tape!**
**Two way finite automata**

**Definition 5.1 (Two way finite automata)**

**Two way deterministic finite automaton** is a five–tuple $A = (Q, \Sigma, \delta, q_0, F)$, where:

- $Q$ is a finite set of states,
- $\Sigma$ is a finite set of input symbols,
- transition function $\delta$ is a mapping $Q \times \Sigma \rightarrow Q \times \{-1, 1\}$ extended by head transitions,
- $q_0 \in Q$ initial state,
- a set of accepting states $F \subseteq Q$.

*We may represent it by a graph or a table.*
Definition 5.2 (Two–way DFA computation)

A string $w$ is **accepted by the two–way DFA**, iff:

- computation started in the initial state at the left–most symbol of $w$
- the first transition from $w$ to the right was in an accepting state
- the computation is not defined outside the word $w$ (computation ends without accepting $w$).

We may add special end–symbols $\#$ \not\in \Sigma$ to any word

If $L(A) = \{\# w \# | w \in L \subseteq \Sigma^*\}$ is regular, then also $L$ is regular

$L = \partial_\# \partial_\#^R (L(A) \cap \# \Sigma^* \#)$
Example 5.1 (Two–way automaton example)

Let \( A = (Q, \Sigma, \delta, q_1, F) \). We define a two–way DFA

\[ B = (Q \cup Q^\| \cup Q^{||} \cup \{q_0, q_N, q_F\}, \Sigma, \delta^|, q_0, \{q_F\}) \]

accepting the language

\[ L(B) = \{#u#| uu \in L(A)\} \]

(it is neither left nor right quotient!):

| \( \delta^| \) | \( x \in \Sigma \) | \# | remark |
|---|---|---|---|
| \( q_0 \) | \( q_N, -1 \) | \( q_1, +1 \) | \( q_1 \) is starting in \( A \) |
| \( q \) | \( p, +1 \) | \( q^|, -1 \) | \( p = \delta(q, x) \) |
| \( q^| \) | \( q^|, -1 \) | \( q^{||}, +1 \) |
| \( q^{||} \) | \( p^{||}, +1 \) | \( q_F, +1 \) | \( q \in F, p = \delta(q, x) \) |
| \( q^\| \) | \( p^\|, +1 \) | \( q_N, +1 \) | \( q \notin F, p = \delta(q, x) \) |
| \( q_N \) | \( q_N, +1 \) | \( q_N, +1 \) |
| \( q_F \) | \( q_N, +1 \) | \( q_N, +1 \) |

Theorem 5.1

Languages accepted by two–way DFA are exactly regular languages.
Two–way DFA and Regular Languages

Proof: DFA $\rightarrow$ two–way DFA

- To a DFA we add the move of the head to the right
- $A = (Q, \Sigma, \delta, q_0, F) \rightarrow 2A = (Q, \Sigma, \delta^\dagger, q_0, F)$, where $\delta^\dagger(q, x) = (\delta(q, x), +1)$.

For the other direction, we need introduction.

The influence of $u \in \Sigma^*$ on the computation over $v \in \Sigma^*$

- the first time we leave $u$ to the right
- we leave $v$ to the left and return back $v$
Function $f_u$ describing computation two–way DFA over $u$

We define $f_u : Q \cup \{q_0\} \to Q \cup \{0\}$

- $f_u(q_0)$ the state of the first transition to the right in case the computation begins left in the state $q_0$,
- $f_u(p); p \in Q$ the state of the right transition in case the computation begins right in $p$
- the symbol 0 denotes failure (a cycle or the head moves left from the initial symbol)
- We define similarity $\sim$ on strings: $u \sim w \iff_{\text{def}} f_u = f_w$,
  - strings are similar iff they define identical function $f$

Languages recognized by two–way DFA are regular

Similarity $\sim$ is a right congruence with a finite index. According to Myhill–Nerode theorem is the language $L(A)$ regular.
Constructive proof

- We need the left–right movement transcript to a linear computation.
- We are interested in accepting computations only.
- We focus on transitions in cuts between input symbols.

Observations:
- The direction of movement repeats (right, left)
- The first and the last transitions are to the right
- Automaton is deterministic, any accepting computation is without cycles
- The first and the last cut contain only one state.

Algorithm: 2DFA → NFA

- Find all possible cuts – state sequences (it’s a finite number).
- Define non-deterministic transition between cuts according to the input symbol.
- We re-construct the computation by composing cuts like a puzzle.
Algorithm: Formal reduction two–way DFA to NFA

Let $A = (Q, \Sigma, \delta, q_0, F)$ be a two–way DFA. We define an equivalent NFA $B = (Q^\dagger, \Sigma, \delta^\dagger, (q_0), F^\dagger)$ as follows:

- $Q^\dagger$ all possible correct transition sequences
  - sequences of states $(q^1, \ldots, q^k); q^i \in Q$
  - with an odd length ($k = 2m + 1$)
  - no state repeats at odd nor at even position
    $(\forall i \neq j) \ (q^{2i} \neq q^{2j}) \land (\forall i \neq j) \ (q^{2i+1} \neq q^{2j+1})$

- $F^\dagger = \{(q) | q \in F\}$ sequences of the length 1

- $\delta^\dagger(c, a) = \{d | d \in Q^\dagger \land c \xrightarrow{a} d \text{ is a locally consistent transition for } a\}$
  - there is a bijection: $h : c_{\text{odd}} \cup d_{\text{even}} \rightarrow c_{\text{even}} \cup d_{\text{odd}}$ so that:
    - for $h(q) \in c_{\text{even}}$ is $(h(q), -1) = \delta(q, a)$
    - for $h(q) \in d_{\text{odd}}$ is $(h(q), +1) = \delta(q, a)$

$L(A) = L(B)$

Trajektory two–way DFA $A$ corresponds to cuts in NFA $B$, therefore $L(A) = L(B)$. 
Example Reduction Two–way DFA to NFA

Possible cuts and their transitions

- leftwards only $r$ – all even positions $r$, that means only one even position
- possible cuts: $(p)$, $(q)$, $(p, r, q)$, $(q, r, p)$.

Non–accepting computation example:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>a</th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>a</th>
<th>b</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>p</td>
<td>p</td>
<td>p</td>
<td>q</td>
<td>q</td>
<td>q</td>
<td>q</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>r</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Resulting NFA:
Definition 5.3 (Moore machine)

**Moore machine** is a sixtuple \( A = (Q, \Sigma, Y, \delta, \mu, q_0) \) consisting of

- \( Q \) non-empty set of states
- \( \Sigma \) finite nonempty set of symbols (input alphabet)
- \( Y \) finite nonempty set of symbols (output alphabet)
- \( \delta \) a mapping \( Q \times \Sigma \to Q \) (transition function)
- \( \mu \) a mapping \( Q \to Y \) (output function)
- \( q_0 \in Q \) (initial state)

- the output function may imitate final states
  - \( F \subseteq Q \) may be replaced by output function \( \mu : Q \to \{0, 1\} \) as follows:
    - \( \mu(q) = 0 \) if \( q \notin F \),
    - \( \mu(q) = 1 \) if \( q \in F \).
A machine calculates the tennis score.

- **Input alphabet**: ID of the player who scored a point
- **Output alphabet & states**: the score (\( Q = Y \) and \( \mu(q) = q \))

<table>
<thead>
<tr>
<th>State/output</th>
<th>( A )</th>
<th>( B )</th>
</tr>
</thead>
<tbody>
<tr>
<td>00:00</td>
<td>15:00</td>
<td>00:15</td>
</tr>
<tr>
<td>15:00</td>
<td>30:00</td>
<td>15:15</td>
</tr>
<tr>
<td>15:15</td>
<td>30:15</td>
<td>15:30</td>
</tr>
<tr>
<td>00:15</td>
<td>15:15</td>
<td>00:30</td>
</tr>
<tr>
<td>30:00</td>
<td>40:00</td>
<td>30:15</td>
</tr>
<tr>
<td>30:15</td>
<td>40:15</td>
<td>30:30</td>
</tr>
<tr>
<td>30:30</td>
<td>40:30</td>
<td>30:40</td>
</tr>
<tr>
<td>15:30</td>
<td>30:30</td>
<td>15:40</td>
</tr>
<tr>
<td>00:30</td>
<td>15:30</td>
<td>00:40</td>
</tr>
<tr>
<td>40:00</td>
<td>( A )</td>
<td>40:15</td>
</tr>
<tr>
<td>40:15</td>
<td>( A )</td>
<td>40:30</td>
</tr>
<tr>
<td>40:30</td>
<td>( A )</td>
<td>deuce</td>
</tr>
<tr>
<td>30:40</td>
<td>deuce</td>
<td>( B )</td>
</tr>
<tr>
<td>15:40</td>
<td>30:40</td>
<td>( B )</td>
</tr>
<tr>
<td>00:40</td>
<td>15:00</td>
<td>( B )</td>
</tr>
<tr>
<td>deuce</td>
<td>( A:40 )</td>
<td>40:40</td>
</tr>
<tr>
<td>( A:40 )</td>
<td>( A )</td>
<td>deuce</td>
</tr>
<tr>
<td>40:40</td>
<td>deuce</td>
<td>( B )</td>
</tr>
<tr>
<td>( A )</td>
<td>15:00</td>
<td>00:15</td>
</tr>
<tr>
<td>( B )</td>
<td>15:00</td>
<td>00:15</td>
</tr>
</tbody>
</table>
Definition 5.4 (Mealy machine)

**Mealy machine** is a six–tuple $A = (Q, \Sigma, Y, \delta, \lambda_M, q_0)$ consisting of:

- $Q$ non–empty set of states
- $\Sigma$ finite nonempty set of symbols (input alphabet)
- $Y$ finite nonempty set of symbols (output alphabet)
- $\delta$ a mapping $Q \times \Sigma \rightarrow Q$ (transition function)
- $\lambda_M$ a mapping $Q \times \Sigma \rightarrow Y$ (output function)
- $q_0 \in Q$ (initial state)

The output is determined by a state and the input symbol

- Mealy machine is more general then Moore
- The output function may be replaced as follows

$$\forall x \in \Sigma \lambda_M(q, x) = \mu(q)$$

or

$$\forall x \in \Sigma \lambda_M(q, x) = \mu(\delta(q, x))$$
Example 5.3 (Mealy Machine)

The automaton for integer division of the input by 8 (the remainder is discarded).

- Three bit move to the left
- We need to remember last three bits
- Three–bit dynamic memory.

<table>
<thead>
<tr>
<th>State \ symbol</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>→000</td>
<td>000/0</td>
<td>001/0</td>
</tr>
<tr>
<td>001</td>
<td>010/0</td>
<td>011/0</td>
</tr>
<tr>
<td>010</td>
<td>100/0</td>
<td>101/0</td>
</tr>
<tr>
<td>011</td>
<td>110/0</td>
<td>111/0</td>
</tr>
<tr>
<td>100</td>
<td>000/1</td>
<td>001/1</td>
</tr>
<tr>
<td>101</td>
<td>010/1</td>
<td>011/1</td>
</tr>
<tr>
<td>110</td>
<td>100/1</td>
<td>101/1</td>
</tr>
<tr>
<td>111</td>
<td>110/1</td>
<td>111/1</td>
</tr>
</tbody>
</table>

- After three steps calculates properly non–regarding the initial state.
for any word in the input alphabet $\Sigma^* \rightarrow$ we get a word in the output alphabet $Y^*$

**Moore machine**

output function $\mu : Q \rightarrow Y$

$\mu^* : Q \times \Sigma^* \rightarrow Y^*$

$\mu^*(q, \lambda) = \lambda$ (sometimes $\mu^*(q, \lambda) = q$)

$\mu^*(q, wx) = \mu^*(q, w).\mu(\delta^*(q, wx))$

**Example:** $\mu^*(00:00, AABA) = (00:00 \cdot) 15:00 \cdot 30:00 \cdot 30:15 \cdot 40:15$

**Mealy machine**

output function $\lambda_M : Q \times \Sigma \rightarrow Y$

$\lambda_M^* : Q \times \Sigma^* \rightarrow Y^*$

$\lambda_M^*(q, \lambda) = \lambda$

$\lambda_M^*(q, wx) = \lambda_M^*(q, w).\lambda_M(\delta^*(q, w), x)$

**Example:** $\lambda_M^*(000, 1101010) = 0001101$
Lemma (Moore and Mealy Machines Reductions)

- For any Moore machine there exists a Mealy machine mapping each input word to the same output word.
- For any Mealy machine there exists a Moore machine mapping each input word to the same output word.

Proof.

⇒ Mealy machine $B = (Q, \Sigma, Y, \delta, \lambda_M, q_0)$ where $\lambda_M(q, x) = \mu(\delta(q, x))$

⇐ We define states of the Moore machine $Q \times Y$, 
$\delta^{|}[q, y], x] = [\delta(q, x), \lambda(q, x)], \mu([q, y]) = y.$
Theorem

Any two equivalent reduce automata are isomorphic.

Proof.

- Any state $q \in Q_1$ is reachable. Find a word $q = \delta_1^*(q_0^1, w)$.
- Define $h(q) = \delta_2^*(q_0^2, w)$.
- Show it is properly defined, bijection, homomorphism.
Finite Automata Summary

Finite Automata

- DFA, minimal DFA
- NFA $2^n$, $\lambda$–NFA, two–way FA $n^n$

Regular Expressions

Automata and Languages

- regular languages
- closed under set operations
- closed under string operations
- closed under substitution, homomorphism, inverse homomorphism
- all finite automata and regular expressions describe the same class of languages.

Key theorems

- MihyU–Nerode theorem (kongruence)
- Kleene theorem (elementar languages and operations)
- Pumping lemma.

Automata with the output

- Moore machine
- Mealy machine.
More powerful than finite automata.

Used to describe document formats, via DTD - document-type definition used in XML (extensible markup language)

We introduce context-free grammar notation

parse tree.

There exist an 'pushdown automaton' that describes all and only the context-free languages. Will be described later.
A string the same forward and backward, like otto or Madam, I’m Adam.

$w$ is a palindrome iff $w = w^R$.

The language $L_{pal}$ of palindromes is not a regular language.

- We use the pumping lemma.
- If $L_{pal}$ is a regular language, let $n$ be the associated constant, and consider: $w = 0^n1^n$.
- For regular $L$, we can break $w = xyz$ such that $y$ consists of one or more 0’s from the first group. Thus, $xz$ would be also in $L_{pal}$ if $L_{pal}$ were regular.

A context-free grammar for palindromes

1. $S \rightarrow \lambda$
2. $S \rightarrow 0$
3. $S \rightarrow 1$
4. $S \rightarrow 0S0$
5. $S \rightarrow 1S1$

A context-free grammar (right) consists of one or more variables, that represent classes of strings, i.e., languages.
Definition 6.1 (Grammar)

A Grammar $G = (V, T, P, S)$ consists of

- Finite set of **terminal symbols** (**terminals**) $T$, like $\{0, 1\}$ in the previous example.
- Finite set of **variables** $V$ (**nonterminals, syntactic categories**), like $S$ in the previous example.
- **Start symbol** $S \in V$ is a variable that represents the language being defined.
- Finite set of **rules** (**productions**) $P$ that represent the recursive definition of the language. Each has the form:
  - $\alpha A \beta \rightarrow \omega$, $A \in V$, $\alpha, \beta, \omega \in (V \cup T)^*$
    - notice the left side (head) contains at least one variable.

  The **head** - the left side, the production symbol $\rightarrow$, the **body** - the right side.

Definition 6.2 (Context free grammar CFG)

**Context free grammar (CFG)** $G = (V, T, P, S)$ has only productions of the form

$$A \rightarrow \alpha, \ A \in V, \alpha \in (V \cup T)^*.$$
Chomsky hierarchy

- Grammar types according to productions allowed.

- **Type 0** (recursively enumerable languages $L_0$)
  - general rules $\alpha \rightarrow \beta$, $\alpha, \beta \in (V \cup T)^*$, $\alpha$ contains at least one variable

- **Type 1** (context languages $L_1$)
  - productions of the form $\alpha A \beta \rightarrow \alpha \omega \beta$
    - $A \in V$, $\alpha, \beta \in (V \cup T)^*$, $\omega \in (V \cup T)^+$
  - with only exception $S \rightarrow \lambda$, then $S$ does not appear at the right side of any production

- **Type 2** (context free languages $L_2$)
  - productions of the form $A \rightarrow \omega$, $A \in V$, $\omega \in (V \cup T)^*$

- **Type 3** (regular (right linear) languages $L_3$)
  - productions of the form $A \rightarrow \omega B$, $A \rightarrow \omega$, $A, B \in V$, $\omega \in T^*$
The classes of languages are ordered

\[ \mathcal{L}_0 \supseteq \mathcal{L}_1 \supseteq \mathcal{L}_2 \supseteq \mathcal{L}_3 \]

later we show proper inclusions

\[ \mathcal{L}_0 \supset \mathcal{L}_1 \supset \mathcal{L}_2 \supset \mathcal{L}_3 \]

\( \mathcal{L}_0 \supseteq \mathcal{L}_1 \) recursively enumerable contain context

productions \( \alpha A \beta \rightarrow \alpha \omega \beta \) have variable \( A \) in the head

\( \mathcal{L}_2 \supseteq \mathcal{L}_3 \) context free contain regular languages

productions \( A \rightarrow \omega B, A \rightarrow \omega \) have in the body a string \((V \cup T)^*\)

\( \mathcal{L}_1 \supseteq \mathcal{L}_2 \) context contain context free languages

we have to eliminate rules \( A \rightarrow \lambda \), we can do it (later).
Definition 6.3 (One step derivation)

Suppose \( G = (V, T, P, S) \) is grammar. Let \( \alpha, \omega, \eta, \nu \in (V \cup T)^* \).

- Let \( \alpha \rightarrow \omega \) be a production rule of \( G \).
- Then one derivation step is: \( \eta \alpha \nu \Rightarrow_G \eta \omega \nu \) or just \( \eta \alpha \nu \Rightarrow \eta \omega \nu \).

We extend \( \Rightarrow \) to any number of derivation steps as follows.

Definition 6.4 (Derivation \( \Rightarrow^* \))

Let \( G = (V, T, P, S) \) is CFG.

- Basis: For any string \( \alpha \in (V \cup T)^* \) it derives itself, \( \alpha \Rightarrow^*_G \alpha \).
- Induction: If \( \alpha \Rightarrow^*_G \beta \) and \( \beta \Rightarrow_G \gamma \) then \( \alpha \Rightarrow^*_G \gamma \).

If grammar \( G \) is understood, then we use \( \Rightarrow^* \) in place of \( \Rightarrow^*_G \).

Example 6.1 (derivation \( E \Rightarrow^* a \ast (a + b00) \))

\[
E \Rightarrow E \ast E \Rightarrow I \ast E \Rightarrow a \ast E \Rightarrow a \ast (E) \Rightarrow a \ast (E + E) \Rightarrow a \ast (I + E) \Rightarrow a \ast (a + E) \Rightarrow \\
\Rightarrow a \ast (a + l) \Rightarrow a \ast (a + l0) \Rightarrow a \ast (a + l00) \Rightarrow a \ast (a + b00)
\]
The Language of a Grammar, Notation Convention for CFG Derivations

Let $G = (V, T, P, S)$ is CFG. The **language** $L(G)$ of $G$ is the set of terminal strings that have derivations from the start symbol.

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

**Language of a variable** $A \in V$ is defined $L(A) = \{ w \in T^* | A \Rightarrow^*_G w \}$.

**Example 6.2 (Not CFL example)**

$L = \{ 0^n1^n2^n | n \geq 1 \}$ is not context-free, there does not exist CFG grammar recognizing it.
Type 3 grammars and regular languages

- productions has the form $A \rightarrow wB$, $A \rightarrow w$, $A, B \in V$, $w \in T^*$
- an example of derivation:

  $P : S \rightarrow 0S|1A|\lambda$, $A \rightarrow 0A|1B$, $B \rightarrow 0B|1S$
  
  $S \Rightarrow 0S \Rightarrow 01A \Rightarrow 011B \Rightarrow 0110B \Rightarrow 01101S \Rightarrow 01101$

- Observations:
  - each word contains exactly one variable (except the last one)
  - the variable is always on the rightmost position
  - the production $A \rightarrow w$ is the last one of the derivation
  - any step generates terminal string and (possibly) changes the variable

- The relation of the grammar and a finite automata
  - variable = state of the finite automata
  - productions = transition function
Example of the reduction FA to a grammar

Example 6.3 (G, FA binary numbers divisible by 5)

$L = \{ w \mid w \in \{ a, b \}^* \text{ & } w \text{ binary numbers divisible by 5} \}$

\[
A \rightarrow 1B | 0A | \lambda \\
B \rightarrow 0C | 1D \\
C \rightarrow 0E | 1A \\
D \rightarrow 0B | 1C \\
E \rightarrow 0D | 1E
\]

Derivation examples

\[
\begin{align*}
A & \Rightarrow 0A \Rightarrow 0 & (0) \\
A & \Rightarrow 1B \Rightarrow 10C \Rightarrow 101A \Rightarrow 101 & (5) \\
A & \Rightarrow 1B \Rightarrow 10C \Rightarrow 101A \Rightarrow 1010A \Rightarrow 1010 & (10) \\
A & \Rightarrow 1B \Rightarrow 11D \Rightarrow 111C \Rightarrow 1111A \Rightarrow 1111 & (15)
\end{align*}
\]
FA to Grammar reduction

Theorem 6.1 ($L \in RE \Rightarrow L \in \mathcal{L}_3$)

For any language recognized by a finite automata there exists a grammar Type 3 recognizing the language.

Proof: FA to Grammar reduction

- $L = L(A)$ for some automaton $A = (Q, \Sigma, \delta, q_0, F)$.
- We define a grammar $G = (Q, \Sigma, P, q_0)$, with productions $P$
  
  $p \rightarrow aq$,  \hspace{0.5cm} \text{iff} \hspace{0.5cm} \delta(p, a) = q$
  
  $p \rightarrow \lambda$, \hspace{0.5cm} \text{iff} \hspace{0.5cm} p \in F$

- Is $L(A) = L(G)$?
  
  - $\lambda \in L(A) \iff q_0 \in F \iff (q_0 \rightarrow \lambda) \in P \iff \lambda \in L(G)$
  
  - $a_1 \ldots a_n \in L(A) \iff \exists q_0, \ldots, q_n \in Q \text{ so that } \delta(q_i, a_{i+1}) = q_{i+1}, q_n \in F$
    
    $\iff (q_0 \Rightarrow a_1q_1 \Rightarrow \ldots a_1 \ldots a_nq_n \Rightarrow a_1 \ldots a_n) \text{ is derivation of } a_1 \ldots a_n$
    
    $\iff a_1 \ldots a_n \in L(G)$
We aim to construct Grammar to FA reduction

- **Opposite direction**
  - production \( A \rightarrow aB \) are encoded to transition function
  - productions \( A \rightarrow \lambda \) define the accepting states
  - we rewrite productions \( A \rightarrow a_1 \ldots a_n B, A \rightarrow a_1 \ldots a_n \) with more terminals
    - we introduce new variables \( H_1, H_2, \ldots, H_n \)
    - define productions \( A \rightarrow a_1 H_2, H_2 \rightarrow a_2 H_3, \ldots, H_n \rightarrow a_n B \)
    - or \( A \rightarrow a_1 H_1, H_1 \rightarrow a_2 H_2, \ldots, H_{n-1} \rightarrow a_n H_n, H_n \rightarrow \lambda \)
  - productions \( A \rightarrow B \) correspond to \( \lambda \) transitions

**Lemma**

*For any Type 3 grammar there exist a Type 3 grammar with the same languages with all productions of the form: \( A \rightarrow aB, A \rightarrow B, A \rightarrow \lambda, A, B \in V, a \in T. *
Standard form of a grammar Type 3

Lemma

For any Type 3 grammar there exist a Type 3 grammar with the same languages with all productions of the form: $A \rightarrow aB$, $A \rightarrow B$, $A \rightarrow \lambda$, $A, B \in V, a \in T$.

Proof.

We define $G\parallel = (V\parallel, T, S, P\parallel)$, for each rule we introduce set of new variables $Y_2, \ldots, Y_n, Z_1, \ldots, Z_n$ and define

<table>
<thead>
<tr>
<th>P</th>
<th>$P\parallel$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A \rightarrow aB$</td>
<td>$A \rightarrow aB$</td>
</tr>
<tr>
<td>$A \rightarrow \lambda$</td>
<td>$A \rightarrow \lambda$</td>
</tr>
<tr>
<td>$A \rightarrow a_1 \ldots a_n B$</td>
<td>$A \rightarrow a_1 Y_2, Y_2 \rightarrow a_2 Y_3, \ldots, Y_n \rightarrow a_n B$</td>
</tr>
<tr>
<td>$Z \rightarrow a_1 \ldots a_n$</td>
<td>$Z \rightarrow a_1 Z_1, Z_1 \rightarrow a_2 Z_2, \ldots, Z_n-1 \rightarrow a_n Z_n, Z_n \rightarrow \lambda$</td>
</tr>
</tbody>
</table>

we eliminate also rules:

$A \rightarrow B$

transitive closure $U(A) = \{B | B \in V & A \Rightarrow^* B\}$

$A \rightarrow w$ for all $Z \in U(A)$ and $(Z \rightarrow w) \in P\parallel$
Theorem 6.2 (Reduction Type 3 grammar to a $\lambda$–NFA)

For any language $L$ of a Type 3 grammar there exists a $\lambda$–NFA recognizing the same language.

Proof: Reduction Type 3 grammar to a $\lambda$–NFA

We take a grammar $G = (V, T, P, S)$ with all productions of the form $A \rightarrow aB$, $A \rightarrow B$, $A \rightarrow \lambda$, $A, B \in V$, $a \in T$ generating $L$ (previous lemma).

we define a non–deterministic $\lambda$–NFA $A = (V, T, \delta, S, F)$ where:

$F = \{ A | (A \rightarrow \lambda) \in P \}$

$\delta(A, a) = \{ B | (A \rightarrow aB) \in P \}$

$L(G) = L(A)$

$\lambda \in L(G) \iff (S \rightarrow \lambda) \in P \iff S \in F \iff \lambda \in L(A)$

$a_1 \ldots a_n \in L(G) \iff$ there exists a derivation $$(S \Rightarrow^* a_1H_1 \Rightarrow \ldots \Rightarrow a_1 \ldots a_nH_n \Rightarrow^* a_1 \ldots a_n)$$

$\iff \exists H_0, \ldots, H_n \in V$ tak že $H_0 = S$, $H_n \in F$

$H_{i+1} \in \delta(H_i, a_k)$ for the step $a_1 \ldots a_{k-1}H_i \Rightarrow a_1 \ldots a_{k-1}a_kH_{i+1}$

$H_{i+1} \in \delta(H_i, \lambda)$ for the step $a_1 \ldots a_kH_i \Rightarrow a_1 \ldots a_kH_{i+1}$

$a_1 \ldots a_n \in L(A)$
Left (and right) linear grammars

Definition 6.6 (Left (and right) linear grammars)

Type 3 grammars are also called **right linear** (the variable is always at right). A grammar $G$ is **left linear** iff all production has the form $A \rightarrow Bw, A \rightarrow w, A, B \in V, w \in T^*.$

Lemma

*Languages generated by left linear grammars are exactly regular languages.*

Proof:

- by ‘reversing’ productions we get right linear grammar $A \rightarrow Bw, A \rightarrow w$ we reduce to $A \rightarrow w^R B, A \rightarrow w^R$
- the new grammar generates $L^R$
- we know regular languages are closed under reverse, $L^R$ is regular, so is $L = (L^R)^R$
- any regular language can be expressed in this form
  - $(FA \Rightarrow \text{reverse} \Rightarrow \text{right linear grammar} \Rightarrow \text{left linear grammar})$
Linear grammars and languages

- Left and right linear grammars together are stronger.

**Definition 6.7 (linear grammar, language)**

A grammar is linear iff all productions have the form
\[ A \rightarrow uBw, A \rightarrow w, A, B \in V, u, w \in T^* \] (at most one variable in the body).

**Linear languages** are languages generated by linear grammars.

- Obviously: regular languages \( \subseteq \) linear languages.
- It is a proper subset \( \subset \).

**Example 6.4 (linear, non–regular language)**

The language \( L = \{0^n1^n | n \geq 1\} \) is not regular but it is linear. It is generated by
the grammar \( S \rightarrow 0S1|01 \).

Observation:

- Linear rules can be splitted to left linear and right linear rules:
  \( S \rightarrow 0A, A \rightarrow S1 \).
A context-free grammar for simple expressions

Example 6.5 (CFG for simple expressions)

A grammar for (simple) expression is
\[ G = \left( \{ E, I \}, \{ +, \ast, (, ), a, b, 0, 1 \}, P, E \right) \]
where \( P \) is the set of rules right.

- Rules 1-4 describe expression.
- Rules 5-10 describe identifiers \( I \), correspond to regular expression \((a + b)(a + b + 0 + 1)^\ast\).

CFG for simple expressions

1. \( E \rightarrow I \)
2. \( E \rightarrow E + E \)
3. \( E \rightarrow E \ast E \)
4. \( E \rightarrow (E) \)
5. \( I \rightarrow a \)
6. \( I \rightarrow b \)
7. \( I \rightarrow Ia \)
8. \( I \rightarrow Ib \)
9. \( I \rightarrow I0 \)
10. \( I \rightarrow I1 \)
Definition 6.8 (Leftmost and Rightmost Derivation)

**Leftmost derivation** $\Rightarrow_{lm}, \Rightarrow_{lm}^*$ replaces at each step the leftmost variable by one of its production bodies.

**Rightmost derivation** $\Rightarrow_{rm}, \Rightarrow_{rm}^*$ replaces at each step the rightmost variable by one of its production bodies.

Example 6.6 (leftmost derivation)

$E \Rightarrow_{lm} E \cdot E \Rightarrow_{lm} I \cdot E \Rightarrow_{lm} a \cdot E \Rightarrow_{lm} a \cdot (E) \Rightarrow_{lm} a \cdot (E + E) \Rightarrow_{lm} a \cdot (I + E) \Rightarrow_{lm} a \cdot (a + E) \Rightarrow_{lm} a \cdot (a + I) \Rightarrow_{lm} a \cdot (a + I0) \Rightarrow_{lm} a \cdot (a + I00) \Rightarrow_{lm} a \cdot (a + b00)$

Example 6.7 (rightmost derivation)

$E \Rightarrow_{rm} E \cdot E \Rightarrow_{rm} E \cdot (E) \Rightarrow_{rm} E \cdot (E + E) \Rightarrow_{rm} E \cdot (E + I) \Rightarrow_{rm} E \cdot (E + I0) \Rightarrow_{rm} E \cdot (E + I00) \Rightarrow_{rm} E \cdot (E + b00) \Rightarrow_{rm} E \cdot (I + b00) \Rightarrow_{rm} E \cdot (a + b00) \Rightarrow_{rm} I \cdot (a + b00) \Rightarrow_{rm} a \cdot (a + b00)$
The Language of a Grammar

Theorem 6.3

\[ L(G_{pal}) \], where \( G_{pal} \) is the set of palindromes over \{0, 1\}.

Proof.

IF: Suppose \( w \) is a palindrome. Induction on \( |w| \) that \( w \) is in \( L(G_{pal}) \).

- BASIS: If \( |w| = 0 \) or \( |w| = 1 \), then \( w \) is \( \lambda \), 0 or 1. We have production rules \( P \to \lambda, P \to 0, P \to 1 \), therefore \( P \Rightarrow^* w \) in any basis case.
- INDUCTION: Suppose \( |w| \geq 2 \). Since \( w = w^R \), so \( w = 0x0 \) or \( w = 1x1 \). Moreover, \( x \) must be a palindrome, \( x = x^R \).
  - If \( w = 0x0 \), then we invoke the inductive hypothesis to claim \( P \Rightarrow^* x \). Then:
    \[ P \Rightarrow 0P0 \Rightarrow^* 0x0 = w. \]
  - If \( w = 1x1 \), do it yourself.

ONLY-IF We assume \( w \in L(G_{pal}) \), that is \( P \Rightarrow^* w \). We claim that \( w \) is a palindrome. Induction on the number of steps in a derivation of \( w \) from \( P \).

- BASIS: \( \lambda, 0, 1 \) are palindromes.
- INDUCTION: Derivation \( P \Rightarrow 0P0 \Rightarrow^* 0x0 = w \). By inductive hypothesis \( x \) is a palindrome. Therefore, \( 0x0 \) and \( 1x1 \) are also palindromes.
Definition 6.9 (Sentential Forms)

Let $G = (V, T, P, S)$ is CFG. Any string $\alpha \in (V \cup T)^*$ such that $S \Rightarrow^* \alpha$ is an **sentential form**.

If $S \Rightarrow_{lm}^* \alpha$, then $\alpha$ is a left sentential form, if $S \Rightarrow_{rm}^* \alpha$, then $\alpha$ is a right sentential form.

Example 6.8

The string $E \ast (I + E)$ is an sentential form but is neither left nor right sentential form.
The tree is the data structure of choice to represent the source program in a compiler. The structure facilitates the translation into executable code.

**Definition 6.10 (Parse Tree)**

Let us fix on a grammar \( G = (V, T, P, S) \). The parse trees for \( G \) are trees such that:

- Each interior node is labeled by a variable in \( V \).
- Each leaf is labeled \( \in V \cup T \cup \{\lambda\} \). If is labeled \( \lambda \), it must be the only child of its parent.
- If an interior node is labeled \( A \) and its children \( X_1, \ldots, X_k \) then \( (A \rightarrow X_1, \ldots, X_k) \in P \) is a production rule.

The children of a node are ordered from the left.

**Notation 1 (Tree terminology)**

Nodes, parent, child, root, interior nodes, leaves, descendants, ancestors.
Tree Examples, Yield Definition

A parse tree of $E \Rightarrow^* l + E$. A parse tree of $P \Rightarrow^* 0110$.

Definition 6.11 (The Yield)

The **yield** is a string of leaves of a parse tree concatenated from the left.

Special importance has yield that:
- is a terminal string,
- the root is labeled by the start symbol.

These yields are strings in the language of the underlying grammar.
Theorem 6.4

Given a context free grammar $G = (V, T, P, S)$, $w \in T^*$. The following are equivalent:

- $A \Rightarrow^* w$.
- $A \Rightarrow_{lm}^* w$.
- $A \Rightarrow_{rm}^* w$.
- There is a parse tree with root $A$ and yield $w$.

Proof directions:
Example 6.9 (Context free derivation)

Assume following is a derivation:

\[ E \Rightarrow I \Rightarrow Ib \Rightarrow ab. \]

Then for any strings \( \alpha, \beta \) following is also derivation:

\[ \alpha E \beta \Rightarrow \alpha I \beta \Rightarrow \alpha Ib \beta \Rightarrow \alpha ab \beta. \]

Theorem

Let \( G = (V, T, P, S) \) be a CFG, suppose there is a parse tree with root labeled by variable \( A \) and with yield \( w \in T^* \).

Then there is a leftmost derivation \( A \Rightarrow_{lm}^* w \) in grammar \( G \).
Proof.

Induction on the height of the tree.

- **Basis**: height 1: Root $A$ with children that read $w$. Since it is a parse tree, $A \Rightarrow w$ is a production, thus $A \Rightarrow_{lm} w$ is a one step.

- **Induction**: Height $n > 1$. Root $A$ with children $X_1, X_2, \ldots, X_k$.
  - If $X_i$ is a terminal, define $w_i$ to be a string consisting of $X_i$ alone.
  - If $X_i$ is a variable, then by inductive hypothesis $X_i \Rightarrow_{lm}^* w_i$.

We construct leftmost derivation, inductively on $i = 1, \ldots, k$ we show $A \Rightarrow_{lm}^* w_1 w_2 \ldots w_i X_{i+1} X_{i+2} \ldots X_k$.

  - If $X_i$ is a terminal, do nothing, just $i \leftarrow i + 1$.
  - If $X_i$ is a variable, rewrite derivation: $X_i \Rightarrow_{lm} \alpha_1 \Rightarrow_{lm} \alpha_2 \ldots \Rightarrow_{lm} w_i$ to

$$w_1 w_2 \ldots w_{i-1} X_i X_{i+1} X_{i+2} \ldots X_k \Rightarrow_{lm}$$

$$w_1 w_2 \ldots w_{i-1} \alpha_1 X_{i+1} X_{i+2} \ldots X_k \Rightarrow_{lm}$$

$$\ldots$$

$$\Rightarrow_{lm} w_1 w_2 \ldots w_{i-1} w_i X_{i+1} X_{i+2} \ldots X_k.$$

When $i = k$, the result is a leftmost derivation of $w$ from $A$. 

$\square$
Leftmost Derivation from Parse Tree Example

- Leftmost child of the root: \( E \Rightarrow_{lm} I \Rightarrow_{lm} a \)
- Rightmost child of the root:
  \[
  E \Rightarrow_{lm} (E) \Rightarrow_{lm} (E + E) \Rightarrow_{lm} (I + E) \Rightarrow_{lm} (a + E)
  \Rightarrow_{lm} (a + I) \Rightarrow_{lm} (a + I0) \Rightarrow_{lm} (a + I00) \Rightarrow_{lm} (a + b00)
  \]
- Root: \( E \Rightarrow_{lm} E \ast E \)
- Leftmost integrated to root:
  \[
  E \Rightarrow_{lm} E \ast E \Rightarrow_{lm} I \ast E \Rightarrow_{lm} a \ast E
  \]
- Full derivation:
  \[
  E \Rightarrow_{lm} E \ast E \Rightarrow_{lm} I \ast E \Rightarrow_{lm} a \ast E \Rightarrow_{lm}
  \Rightarrow_{lm} a \ast (E) \Rightarrow_{lm} a \ast (E + E) \Rightarrow_{lm} a \ast (I + E) \Rightarrow_{lm}
  \Rightarrow_{lm} a \ast (a + E) \Rightarrow_{lm} a \ast (a + I) \Rightarrow_{lm} a \ast (a + I0) \Rightarrow_{lm}
  \Rightarrow_{lm} a \ast (a + I00) \Rightarrow_{lm} a \ast (a + b00).
  \]
Applications of Context-Free Grammars

- Parsers
- Markup Languages (HTML)
- XML and Document Type Definitions (DTD)

**Definition 6.12 (Grammar Equivalence)**

Grammars $G_1, G_2$ are **equivalent**, iff $L(G_1) = L(G_2)$ they generate the same language.
Ambiguity in Grammars

Two derivations of the same expression:

\[ E \Rightarrow E + E \Rightarrow E + E \ast E \quad E \Rightarrow E \ast E \Rightarrow E + E \ast E \]

- The difference is important; left \( 1 + (2 \ast 3) = 7 \), right \( (1 + 2) \ast 3 = 9 \).
- This grammar can be modified to be unambiguous.

Example 6.10

Different derivations may represent the same parse tree. Then, it is not a problem.
1. \( E \Rightarrow E + E \Rightarrow I + E \Rightarrow a + E \Rightarrow a + I \Rightarrow a + b \)
2. \( E \Rightarrow E + E \Rightarrow E + I \Rightarrow I + I \Rightarrow I + b \Rightarrow a + b \).
Definition 6.13 (ambiguous CFG)

We say a CFG $G = (V, T, P, S)$ is **ambiguous** if there is at least one string $w \in T^*$ for which we can find two different parse trees, each with root labeled $S$ and yield $w$.

If each string has at most one parse tree in the grammar, then the grammar is **unambiguous**.

Two derivation trees with yield $a + a \ast a$ showing the ambiguity of the grammar.
There is no algorithm that can even tell us whether a CFG is ambiguous. There are context-free languages that have nothing but ambiguous CFG's. There are some hints for removing ambiguity.

There are two causes of ambiguity:

- The precedence of operators is not respected.
- A sequence of identical operators can group either from the left or from the right.
The solution enforcing precedence is to introduce several different variables, each for one level of 'binding strength'. Specifically:

- A **factor** is an expression that cannot be broken by any operator.
  - identifiers.
  - Any parenthesized expression.
- A **term** is an expression that cannot be broken by + operator.
- An **expression** can be broken by either * or +.

An unambiguous expression grammar

1. \( I \rightarrow a | b | Ia | Ib | I0 | I1 \)
2. \( F \rightarrow I|(E) \)
3. \( T \rightarrow F | T \ast F \)
4. \( E \rightarrow T | E + T \).

The sole parse tree for \( a + a \ast a \) is shown in the diagram.
Unambiguity of our grammar.

Points why no string can have two different parse trees.

- A factor is either a single identifier or any parenthesized expression.
- Any string derived from $T$ is a sequence of factors connected by $\ast$.
- Because of two productions of $T$, the only parse tree breaks $f_1 \ast f_2 \ldots \ast f_n$ into a term $f_1 \ast f_2 \ldots \ast f_{n-1}$ and a factor $f_n$.
- Likewise, an expression is a sequence of terms connected by $\plus$. The production $E \rightarrow E + T$ takes as the term always the last one.
Different leftmost derivations for different parse trees.

\[ E \Rightarrow_{lm} E + E \Rightarrow_{lm} I + E \Rightarrow_{lm} a + E \Rightarrow_{lm} a + E \ast E \Rightarrow_{lm} a + I \ast E \Rightarrow_{lm} a + a \ast E \Rightarrow_{lm} a + a \ast I \Rightarrow_{lm} a + a \ast a \]

\[ E \Rightarrow_{lm} E \ast E \Rightarrow_{lm} E + E \ast E \Rightarrow_{lm} I + E \ast E \Rightarrow_{lm} a + E \ast E \Rightarrow_{lm} a + a \ast E \Rightarrow_{lm} a + a \ast I \Rightarrow_{lm} a + a \ast a \]
Theorem 6.5

For each grammar \( G = (V, T, P, S) \) and string \( w \in T^* \), \( w \) has two distinct parse trees if and only if \( w \) has two distinct leftmost derivations from \( S \).

Proof.

(Only-if) Wherever the two parse trees first have a node at which different productions are used, the leftmost derivations constructed will also use different productions and thus be different derivations.

(If) Start constructing a tree only the root, labeled \( S \). From each production used determine the node is being changed and what the children of this node should be. If there are two distinct derivations, then at the first step where the derivations differ, the nodes being constructed will get different lists of children, and this difference guarantees that the parse trees are distinct.
Inherent Ambiguity

Definition 6.14 (Inherent Ambiguity)
A context-free language $L$ is said to be **inherently ambiguous** if all its grammars are ambiguous.
If one grammar for $L$ is unambiguous, then $L$ is an unambiguous language.

Example 6.11 (Inherently ambiguous language)
An example of an inherently ambiguous language:

$$L = \{a^n b^n c^m d^m | n \geq 1, m \geq 1\} \cup \{a^n b^m c^m d^n | n \geq 1, m \geq 1\}.$$ 

1. $S \rightarrow AB \mid C$
2. $A \rightarrow aAb \mid ab$
3. $B \rightarrow cBd \mid cd$
4. $C \rightarrow aCd \mid aDd$
5. $D \rightarrow bDc \mid bc$.

This grammar is ambiguous. For example, the string $aabbccdd$ has the two leftmost derivations:

1. $S \Rightarrow_{lm} AB \Rightarrow_{lm} aAbB \Rightarrow_{lm} aabbB \Rightarrow_{lm} aabbcBd \Rightarrow_{lm} aabbccdd$
2. $S \Rightarrow_{lm} C \Rightarrow_{lm} aCd \Rightarrow_{lm} aaDdd \Rightarrow_{lm} aabDcdd \Rightarrow_{lm} aabbccdd$
Two parse trees for $aabbccdd$.

No matter what modifications we make to the basic grammar, it will generate at least some of the strings of the form $a^n b^n c^n d^n$ in the two ways that the grammar presented.
Summary of Chapter 5

- **Context-Free Grammars** $G = (V, T, P, S)$, $P$ recursive rules called productions.

- **Derivations and Languages** Beginning with $S$ we repeatedly replace variable by the body. The language is the set of terminal strings we can so derive. **Leftmost, Rightmost Derivations**.

- **Sentential Forms** any step of derivation.

- **Parse Trees** Interior nodes are labeled by variables, and leaves are labeled by terminals or $\lambda$. For internal node, there must be a production justifying the node-children relation.

- **Equivalence of Parse Trees and Derivations** A terminal string is in the language of a grammar iff it is the yield of at least one parse tree. The existence of leftmost derivations and parse trees define exactly the strings in the language of a CFG.

- **Ambiguous Grammars** For some CFG’s, it is possible to find a terminal string with more than one parse tree.

- **Eliminating Ambiguity** For many useful grammars it is possible to find an unambiguous grammar that generates the same language.
Pushdown Automata

- Pushdown automata is an extension of the $\lambda$–NFA.
- The additional feature is the stack. **Stack** can be read, pushed, and popped only at the top.
- It can remember an infinite amount of information.
- Pushdown automata define context-free languages.
- Deterministic pushdown automata accept only a proper subset of the CFL’s.

$$\begin{align*}
\text{Input } w \in \Sigma^* & \quad \rightarrow \quad \lambda - \text{NFA} \\
\delta(q, a, X) & \rightarrow (q, \gamma) \\
\gamma & \in \Gamma \\
\text{only the top can be accessed}
\end{align*}$$

A pushdown automaton.
Definition 7.1 (Pushdown Automata)

A pushdown automaton (PDA) is $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, where

- $Q$ is a finite set of states.
- $\Sigma$ is a finite set of input symbols.
- $\Gamma$ is a finite stack alphabet.
- $\delta$ is the transition function.
  - $\delta : Q \times (\Sigma \cup \{\lambda\}) \times \Gamma \rightarrow P_{FIN}(Q \times \Gamma^*)$,
  - $\delta(q, a, X) \ni (p, \gamma)$ where $p$ is the new state and $\gamma$ a string of stack symbols that replace $X$ on top of the stack.
- $q_0$ is the initial state.
- $Z_0$ is the initial stack symbol. The only symbol on the stack at the beginning.
- $F$ is the set of accepting (final) states. May be undefined.
Example 7.1

PDA for the language $ww^R$: $L_{wwr} = \{ww^R | w \in (0 + 1)^* \}$.

A PDA accepting $L_{wwr}$:

- Start $q_0$ represents a guess that we have not yet seen the middle.
- At any time, non-deterministically guess
  - Stay $q_0$ (not yet in the middle).
  - Spontaneously go to state $q_1$ (we have seen the middle).
- In $q_0$, read the input symbol and push it onto the stack.
- In $q_1$, compare the input symbol with the one on top of the stack. If they match, consume the input symbol and pop the stack.
- If we empty the stack, accept by going to $q_2$. No input should remain.
PDA for $L_{\text{wwr}}$

**Example 7.2 (PDA for $L_{\text{wwr}}$)**

PDA for $L_{\text{wwr}}$ can be described $P = (\{q_0, q_1, q_2\}, \{0, 1\}, \{0, 1, Z_0\}, \delta, q_0, Z_0, \{q_2\})$

where $\delta$ is defined:

<table>
<thead>
<tr>
<th>Transition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\delta(q_0, 0, Z_0)$</td>
<td>${(q_0, 0Z_0)}$ Push the input on stack, leave the start symbol there.</td>
</tr>
<tr>
<td>$\delta(q_0, 1, Z_0)$</td>
<td>${(q_0, 1Z_0)}$</td>
</tr>
<tr>
<td>$\delta(q_0, 0, 0)$</td>
<td>${q_0, 00}$ Stay in $q_0$, read the input and push it onto stack.</td>
</tr>
<tr>
<td>$\delta(q_0, 0, 1)$</td>
<td>${q_0, 01}$</td>
</tr>
<tr>
<td>$\delta(q_0, 1, 0)$</td>
<td>${q_0, 10}$</td>
</tr>
<tr>
<td>$\delta(q_0, 1, 1)$</td>
<td>${q_0, 11}$</td>
</tr>
<tr>
<td>$\delta(q_0, \lambda, Z_0)$</td>
<td>${q_1, Z_0}$ Spontaneous transition to $q_1$, no change on stack.</td>
</tr>
<tr>
<td>$\delta(q_0, \lambda, 0)$</td>
<td>${q_1, 0}$ State $q_1$ matches the input and the stack symbols.</td>
</tr>
<tr>
<td>$\delta(q_0, \lambda, 1)$</td>
<td>${q_1, 1}$</td>
</tr>
<tr>
<td>$\delta(q_1, 0, 0)$</td>
<td>${q_1, \lambda}$</td>
</tr>
<tr>
<td>$\delta(q_1, 1, 1)$</td>
<td>${q_1, \lambda}$</td>
</tr>
<tr>
<td>$\delta(q_1, \lambda, Z_0)$</td>
<td>${q_2, Z_0}$ We have found $ww^R$ and go to the accepting state.</td>
</tr>
</tbody>
</table>
Definition 7.2 (Transition diagram for PDA)

A transition diagram for PDA contains:

- The nodes correspond to the states of the PDA.
- The first arrow indicates the start state, and doubly circled states are accepting.
- The arc correspond to transitions of the PDA. An arc labeled $a, X \rightarrow \alpha$ from state $q$ to $p$ means that $\delta(q, a, X) \ni (p, \alpha)$.
- Conventionally, the start stack symbol is $Z_0$.

Labels:

- `input_symbol, stack_symbol` → `string_to_push`

0, $Z_0 \rightarrow 0Z_0$
1, $Z_0 \rightarrow 1Z_0$
0, 0 → 00
0, 1 → 01
1, 0 → 10
0, 0 → λ
1, 1 → 11
1, 1 → λ
Definition 7.3 (PDA situation, ID)
We represent the situation of a PDA by a triple \((q, w, \gamma)\), where

- \(q\) is the state
- \(w\) is the remaining input and
- \(\gamma\) is the stack contents (top on the left).

Such a triple is called an **instantaneous description (ID), situation** of the pushdown automaton.

Definition 7.4 (\(\vdash\), \(\vdash^*\) PDA Computation, Sequences of situations)
Let \(P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)\) be a PDA. Define \(\vdash_P\) or just \(\vdash\) as follows. Suppose \(\delta(q, a, X) \ni (p, \alpha)\). Then for all strings \(w \in \Sigma^*\) and \(\beta \in \Gamma^*\):

\[
(q, aw, X\beta) \vdash (p, w, \alpha\beta).
\]

We also use the symbol \(\vdash_P^*\) or \(\vdash^*\) to represent zero or more moves of the PDA, i.e.

- \(I \vdash^* I\) for any ID \(I\)
- \(I \vdash^* J\) if there exists some ID \(K\) such that \(I \vdash K\) and \(K \vdash^* J\).
Definition 7.5 (PDA language accepted by final state)

Let $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ be a PDA. Then $L(P)$, the language accepted by $F$ by a final state, is

$$L(P) = \{w | (q_0, w, Z_0) \vdash^*_P (q, \lambda, \alpha) \text{ for some } q \in F \text{ and any stack string } \alpha \}.$$ 

Example 7.3

The PDA example for $L_{wwr}$ accepts the language.

- (IF) For any $x = ww^R$, we have a accepting computation

  $$(q_0, ww^R, Z_0) \vdash^* (q_0, w^R, w^R Z_0) \vdash (q_1, w^R, w^R Z_0) \vdash^* (q_1, \lambda, Z_0) \vdash (q_2, \lambda, Z_0).$$

- (Only If)
  - The only way to enter $q_2$ is from $q_1$ and $Z_0$ at the top of the stack.
  - Any accepting computation starts in $q_0$, changes to $q_1$ and never returns to $q_0$.
  - We can prove by induction on $|x|$ that $(q_0, x, Z_0) \vdash^* (q_1, \lambda, Z_0)$ exactly for the strings of the form $x = ww^R$. 

ID’s of the PDA on input 1111

(q₀, 1111, Z₀) → (q₀, 111, 1Z₀) → (q₀, 11, 11Z₀) → (q₀, 1, 111Z₀) → (q₀, λ, 1111Z₀) → (q₁, λ, 1111Z₀)

(q₀, 111, 1Z₀) → (q₁, 111, 1Z₀) → (q₁, 11, 11Z₀) → (q₁, 1, 111Z₀) → (q₁, λ, 1111Z₀) → (q₂, λ, 1111Z₀)

(q₁, 1111, Z₀) → (q₁, 111, Z₀) → (q₁, 11, Z₀) → (q₁, 1, Z₀) → (q₂, 11, Z₀) → (q₂, λ, Z₀) → (q₂, λ, Z₀)
Notation Convention for PDA’s

- $a, b, c$: symbols of the input alphabet
- $q, p, r$: states
- $u, w, x, y, z$: strings of input symbols
- $X, Y, E, Z_0$: stack symbols
- $\alpha, \beta, \gamma$: strings stack symbols.
Data $P$ never looks at cannot affect its computation.

Lemma

If $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ is a PDA, and $(q, x, \alpha) \vdash_P^* (p, y, \beta)$, then for any strings $w \in \Sigma^*$ and $\gamma \in \Gamma^*$, it is also true that $(q, xw, \alpha \gamma) \vdash_P^* (p, yw, \beta \gamma)$. Specially, it is true for $\gamma = \lambda$ and/or $w = \lambda$.

Proof.

Induction on the number of steps in the sequence of ID's that take $(q, xw, \alpha \gamma)$ to $(p, yw, \beta \gamma)$. Each of the moves $(q, x, \alpha) \vdash_P^* (p, y, \beta)$ is justified without sing $w$ and/or $\gamma$ in any way. Therefore, each move is still justified when these strings are sitting on the input and stack.

Remark

The same with the stack does not hold. The PDA may use $\gamma$ on the stack and push it back again. $(q_0, (a + a), E) \vdash_P^* (q_0, a + a), E + E))$.
Definition 7.6 (PDA language accepted by empty stack)

For each PDA \( P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F) \) we define \( N(P) \), the language accepted by \( P \) by empty stack

\[
N(P) = \{ w | (q_0, w, Z_0) \vdash^* P (q, \lambda, \lambda) \text{ for some } q \in Q \}.
\]

The set of accepting states is in this case irrelevant, we shall sometimes leave it off and state \( P \) as a six-tuple \( P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0) \).

Example 7.4

In the previous example, we change the transition \( \delta(q_1, \lambda, Z_0) = \{(q_2, Z_0)\} \) to pop the last symbol: \( \delta(q_1, \lambda, Z_0) = \{(q_2, \lambda)\} \). Now, \( L(P') = N(P') = L_{wwr} \).
Lemma (From Empty Stack to Final State)

If \( L = N(P_N) \) for some PDA \( P_N = (Q, \Sigma, \Gamma, \delta_N, q_0, Z_0) \), then there is a PDA \( P_F \) such that \( L = L(P_F) \).

Proof.

\[ P_F = (Q \cup \{p_0, p_f\}, \Sigma, \Gamma \cup \{X_0\}, \delta_F, p_0, X_0, \{p_F\}) \], where \( \delta_F \) is

- \( \delta_F(p_0, \lambda, X_0) = \{(q_0, Z_0X_0)\} \) (start).
- \( \forall (q \in Q, a \in \Sigma \cup \{\lambda\}, Y \in \Gamma), \delta_F(q, a, Y) = \delta_N(q, a, Y) \).
- In addition, \( \delta_F(q, \lambda, X_0) \ni (p_f, \lambda) \) for every \( q \in Q \).

We must show that \( w \in L(P_N) \) iff \( w \in L(P_F) \).

- (If) \( P_F \) accepts as follows:
  \( (p_0, w, X_0) \vdash_{P_F} \)
  \( (q_0, w, Z_0X_0) \vdash_{P_F=N_F}^* (q, \lambda, X_0) \vdash_{P_F} \)
  \( (p_f, \lambda, \lambda) \).

- (Only if) No other way to go to \( p_f \) than that above.
If-Else Example

Example 7.5 (Accepts by empty stack)
We design a PDA that stops at the first error on if (i) and else (e) sequence, i.e. we have more else’s then if’s.

\[ P_N = (\{q\}, \{i, e\}, \{Z\}, \delta_N, q, Z) \] where

- \( \delta_N(q, i, Z) = \{(q, ZZ)\} \) push.
- \( \delta_N(q, e, Z) = \{(q, \lambda)\} \) pop.

Example 7.6 (Accepts by final state)

\[ P_F = (\{p, q, r\}, \{i, e\}, \{Z, X_0\}, \delta_F, p, X_0, \{r\}) \] where

- \( \delta_F(p, \lambda, X_0) = \{(q, ZX_0)\} \) start.
- \( \delta_F(q, i, Z) = \{(q, ZZ)\} \) push.
- \( \delta_F(q, e, Z) = \{(q, \lambda)\} \) pop.
- \( \delta_F(q, \lambda, X_0) = \{(r, \lambda)\} \) accept.
 Lemma

Let \( L = L(P_F) \) for some PDA \( P_F = (Q, \Sigma, \Gamma, \delta_F, q_0, Z_0, F) \).
There is a PDA \( P_N \) such that \( L = N(P_N) \).

Proof.

Let \( P_N = (Q \cup \{p_0, p\}, \Sigma, \Gamma \cup \{X_0\}, \delta_N, p_0, X_0) \),
where

- \( \delta_N(p_0, \lambda, X_0) = \{(q, Z_0X_0)\} \) start.
- \( \forall(q \in Q, a \in \Sigma \cup \{\lambda\}, Y \in \Gamma), \delta_N(q, a, Y) = \delta_F(q, a, Y) \) simulate.
- \( \forall(q \in F, Y \in \Gamma \cup \{X_0\}), \delta_N(q, \lambda, Y) \ni (p, \lambda) \) i.e. accept if \( P_F \) accepts.
- \( \forall(Y \in \Gamma \cup \{X_0\}), \delta_N(p, \lambda, Y) = \{(p, \lambda)\} \) clean the stack.

The proof \( w \in N(P_n) \) iff \( w \in L(P_F) \) is similar as before.
Equivalence of PDA’s and CFG’s

Theorem 7.1 ($L_2$ Context-free languages characterization)

Following three statements about a language $L \subseteq \Sigma^*$ are equivalent:

- There exists a context context-free grammar such that $L(G) = L$.
- There exists a PDA $P$ such that $L(P) = L$.
- There exists a PDA $P$ such that $N(P) = L$.

Organization of constructions showing equivalence of three ways of defining the CFL’s.
Construction of PDA from a CFG $G$.

Let $G = (V, T, P, S)$ be a CFG. Construct the PDA $P = (\{q\}, T, V \cup T, \delta, q, S)$.

1. For each variable $A \in V$, $\delta(q, \lambda, A) = \{(q, \beta) \mid A \to \beta$ is a production of $G\}$.
2. For each terminal $a \in T$, $\delta(q, a, a) = \{(q, \lambda)\}$.

Given a CFG $G$, we construct a PDA that simulates the leftmost derivations on $G$.

Any not terminal left-sentential form can be written $uA\alpha$, $A$ the leftmost nonterminal, $u$ is a string of terminals. We call $A\alpha$ to be a tail, it is $\lambda$ for terminals only sentential form.

The tail of each sentential form appears on the stack.

Suppose ID $(q, y, A\alpha)$ representing left-sentential form $uA\alpha$. Production guesses a rule $A \to \beta$. The PDA replaces $A$ on the stack by $\beta$, entering ID $(q, y, \beta\alpha)$.

Remove all terminals $u_\ell$ from the left of $\beta\alpha$ comparing them against the input.

Then, either the stack is empty or we reach the left-sentential form $uuu_\ell\gamma$ and iterate.
Example 7.7

Let us convert the grammar:

\[ I \rightarrow a | b | Ia | Ib | I0 | I1 \]

\[ E \rightarrow I | E \ast E | E + E | (E) \].

The set of input symbols for the PDA is \( \Sigma = \{a, b, 0, 1, (,), +, \ast\} \), \( \Gamma = \Sigma \cup \{I, E\} \), the transition \( \delta \):

\[ \delta(q, \lambda, I) = \{(q, a), (q, b), (q, Ia), (q, Ib), (q, I0), (q, I1)\}. \]

\[ \delta(q, \lambda, E) = \{(q, I), (q, E \ast E), (q, E + E), (q, (E))\}. \]

For all \( s \in \Sigma \) is \( \delta(q, s, s) = \{(q, \lambda)\} \), for example \( \delta(q, +, +) = \{(q, \lambda)\} \).

Also, \( \delta \) is empty otherwise.

Theorem 7.2 (PDA empty stack acceptance from a CFG.)

If PDA \( P \) if constructed from CFG \( G \) by the construction above, then \( N(P) = L(G) \).

- Leftmost derivation: \( E \Rightarrow E \ast E \Rightarrow I \ast E \Rightarrow a \ast E \Rightarrow a \ast I \Rightarrow a \ast b \)
- PDA: \( (q, a \ast b, E) \vdash (q, a \ast b, E \ast E) \vdash (q, a \ast b, I \ast E) \vdash (q, a \ast b, a \ast E) \vdash (q, \ast b, \ast E) \vdash (q, b, E) \vdash (q, b, I) \vdash (q, b, b) \vdash (q, \lambda, \lambda) \)
We shall prove \( w \in N(P) \) iff \( w \in L(G) \).

(If) We need to prove: If \( (q, u, A) \vdash_P^* (q, \lambda, \lambda) \), then \( A \xrightarrow{\ast}_G u \).

Proof is an induction on the number of moves taken by \( P \).

- \( n = 1 \) move: The only possibility the production \((A \rightarrow \lambda) \in G\), in construction only for \( u = \lambda \).
- \( n > 1 \) moves; First move type (1), \( A \) replaced by \( Y_1 Y_2 \ldots Y_k \) from a production \( A \rightarrow Y_1 Y_2 \ldots Y_k \).

We split \( u = u_1 u_2 \ldots u_k \) as on the Figure and use inductive hypothesis on each \( i = 1, \ldots, k \):

\((q, u_i u_{i+1} \ldots u_k, Y_i) \vdash^* (q, u_{i+1} \ldots u_k, \lambda)\) to get \( Y_i \Rightarrow^* u_i \).

Together, \( A \Rightarrow Y_1 Y_2 \ldots Y_k \Rightarrow u_1 Y_2 \ldots Y_k \ldots \Rightarrow u_1 u_2 \ldots u_k \).

(Only If) Suppose \( w \in L(G) \), \( w \) has leftmost derivation \( S = \gamma_1 \xrightarrow{lm} \gamma_2 \xrightarrow{lm} \ldots \xrightarrow{lm} \gamma_n = w \).

By induction on \( i \): \( (q, w, S) \vdash_P^* (q, y_i, \alpha_i) \) where \( \gamma_i = u_i \alpha_i \) is the left-sentential form and \( u_i y_i = w \).
The key event: PDA pops one symbol off the stack.
   The state before and after the popping may be different.

Grammar variables are composite symbols \([qXr_k]\),
   \(q, r_k \in Q, X \in \Gamma\)

we add new starting variable \(S\).

**Theorem 7.3 (Grammar for a PDA)**

*Let \(P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0)\) be a PDA. Then there is a context-free grammar \(G\) such that \(L(G) = N(P)\).*

Productions will be:

- For all \(p_\ell \in Q\):
  \(S \rightarrow [q_0Z_0p_\ell]\), i.e. guess the end state \(p_\ell\) and run the PDA to
  \((q_0, w, Z_0) \vdash^* (p_\ell, \lambda, \lambda)\).

- For all pairs \((r, Y_1Y_2 \ldots Y_k)\) in all \(\delta(q, a, X)\), all \(r_i \in Q\), create a production
  \([qXr_k] \rightarrow a[rY_1r_1][r_1Y_2r_2] \ldots [r_{k-1}Y_kr_k]\).
Proof.

The proof that: 
\[ qXp \Rightarrow^{\ast} \] wif and only if \((q, w, X) \vdash^{\ast} (p, \lambda, \lambda)\) is done in both direction by induction (number of moves, number of steps in the derivation.)

Example 7.8 (\(\{0^n1^n; n > 0\}\))

<table>
<thead>
<tr>
<th>(\delta)</th>
<th>Productions</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\delta(p, 0, Z) \ni (p, A))</td>
<td>(S \rightarrow [pZp][pZq]) (1)</td>
</tr>
<tr>
<td>(\delta(p, 0, A) \ni (p, AA))</td>
<td>([pZp] \rightarrow 0[pAp]) (2)</td>
</tr>
<tr>
<td></td>
<td>([pZq] \rightarrow 0[pAq]) (3)</td>
</tr>
<tr>
<td></td>
<td>([pAp] \rightarrow 0[pAq][qAp]) (5)</td>
</tr>
<tr>
<td></td>
<td>([pAq] \rightarrow 0[pAp][pAq]) (6)</td>
</tr>
<tr>
<td></td>
<td>([pAq] \rightarrow 0[pAq][qAq]) (7)</td>
</tr>
<tr>
<td>(\delta(p, 1, A) \ni (q, \lambda))</td>
<td>([pAq] \rightarrow 1) (8)</td>
</tr>
<tr>
<td>(\delta(q, 1, A) \ni (q, \lambda))</td>
<td>([qAq] \rightarrow 1) (9)</td>
</tr>
</tbody>
</table>

Derivation of 0011
\(S \Rightarrow^{(1)} [pZq] \Rightarrow^{(3)} 0[pAq] \Rightarrow^{(7)} 00[pAq][qAq] \Rightarrow^{(8)} 001[qAq] \Rightarrow^{(9)} 0011\)
Normal Forms for Context-Free Grammars

- We simplify CFL’s.
  - Chomsky Normal Form
  - Greibach Normal Form
- We prove *pumping lemma* for CFL’s.
- We study closure properties and decision properties. Some of them remain, some not.

**Definition (Chomsky Normal Form)**

A grammar is in the Chomsky Normal Form iff it has no useless symbols and all production are of the form $A \rightarrow BC$ or $A \rightarrow a$, $A, B, C$ where are variables, $a$ is a terminal.

- Every CFL (without $\lambda$) is generated by a CFG in Chomsky Normal Form.

To get there, we perform simplifications
- Eliminate *useless symbols*
- Eliminate $\lambda$- productions $A \rightarrow \lambda$ for some variable $A$
- Eliminate *unit productions* $A \rightarrow B$ for variables $A, B$. 

Eliminating Useless symbols

**Definition 8.1 (useful symbol)**

- A symbol $X \in V \cup T$ is **useful** for a grammar $G = (V, T, P, S)$ if there is some derivation of the form $S \Rightarrow^* \alpha X \beta \Rightarrow^* w$ where $w \in T^*$, $X \in (V \cup T)$.
- If $X$ is not useful, we say it is **useless**.
- $X$ is **generating** iff $X \Rightarrow^* w$ for some terminal string $w$. Note always $w \Rightarrow^* w$ by zero steps.
- $X$ is **reachable** iff there is $S \Rightarrow^* \alpha X \beta$ for some $\alpha, \beta$.

We aim eliminate non-generating and not reachable symbols.

**Example 8.1**

Consider the grammar:

- $S \rightarrow AB | a$
- $A \rightarrow b$

Eliminate $B$ (nongenerating):
- $S \rightarrow a$
- $A \rightarrow b$.

Eliminate $A$ (not reachable):
- $S \rightarrow a$. 
Theorem 8.1 (Eliminating useless symbols)

Let $G = (V, T, P, S)$ be a CFG, and assume that $L(G) \neq \emptyset$. Let $G_1 = (V_1, T_1, P_1, S)$ is obtained:

- Eliminate nongenerating symbols and all productions involving them.
- Eliminate all symbols that are not reachable after previous step.

Then $G_1$ has no useless symbols, and $L(G_1) = L(G)$.

Generating symbols:
- BASIS: Every $a \in T$ is generating.
- INDUCTION: For any production $A \rightarrow \alpha$ and every symbol of $\alpha$ is generating. Then $A$ is generating. (This includes $A \rightarrow \lambda$).

Reachable symbols:
- BASIS: $S$ is surely reachable.
- INDUCTION: If $A$ is reachable, for all production with $A$ in the head, all symbols of the bodies are also reachable.

Theorem 8.2

The algorithms above find all and only the generating / reachable symbols.
Eliminating $\lambda$-Productions

- Without $\lambda$-production, $\lambda \notin L$.
- We aim to prove: $L$ has a CFG, then $L - \{\lambda\}$ has a CFG without $\lambda$-productions.

**Definition 8.2 (nullable variable)**

A variable $A$ is **nullable** if $A \Rightarrow^* \lambda$.

For nullable variables in the body $B \rightarrow CAD$, we create two versions of the production - with and without this variable.

An algorithm to find all nullable symbols of $G$:
- If $A \rightarrow \lambda$ is a production of $G$, then $A$ is nullable.
- If $B \rightarrow C_1 \ldots C_k$ where each $C_i$ is nullable, then $B$ is nullable (note terminal $C_i \in T$ is not nullable).
Construction of a grammar without $\lambda$-productions from $G = (V, T, P, S)$.

- Determine nullable symbols.
- For each production $A \to X_1 \ldots X_k \in P$, $k \geq 1$ suppose $m$ of $X_i$’s are nullable. The new grammar $G_1$ will have $2^m$ versions of this production with/without each nullable symbol except $\lambda$ in case $m = k$.

Example 8.2

Consider the grammar:

<table>
<thead>
<tr>
<th>Production</th>
<th>Final grammar:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \to AB$</td>
<td>$S \to AB</td>
</tr>
<tr>
<td>$A \to aAA</td>
<td>\lambda$</td>
</tr>
<tr>
<td>$B \to bBB</td>
<td>\lambda$</td>
</tr>
</tbody>
</table>

Final grammar:

<table>
<thead>
<tr>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \to AB</td>
</tr>
<tr>
<td>$A \to aAA</td>
</tr>
<tr>
<td>$B \to bBB</td>
</tr>
</tbody>
</table>
Eliminating Unit Productions

Definition 8.3 (unit production)

A unit production is $A \rightarrow B \in P$ where both $A, B$ are variables.

Example 8.3

- $I \rightarrow a|b|la|lb|I0|I1$
- $F \rightarrow I|(E)$
- $T \rightarrow F|T \ast F$
- $E \rightarrow T|E + T$

Expanding $T$ in $E \rightarrow T$

Expanding $E \rightarrow I$

Expanding $E \rightarrow F$

Expanding $E \rightarrow I$(E)

Together: $E \rightarrow a|b|la|lb|I0|I1|(E)|T \ast F|E + T$.

We have to avoid possible cycles.

Definition 8.4 (unit pair)

A pair $A, B \in V$ such that $A \Rightarrow^* B$ using only unit productions is called a unit pair.
Unit pairs identification

- **BASIS:** $(A, A)$ for $A \in V$ is a unit pair.
- **INDUCTION:** If $(A, B)$ is a unit pair and $B \rightarrow C \in P$, then $(A, C)$ is a unit pair.

Example 8.4 (Unit pairs from previous grammar)

$(E, E), (T, T), (F, F), (I, I), (E, T), (E, F), (E, I), (T, F), (T, I), (F, I)$.

To eliminate unit productions from $G = (V, T, P, S)$:

- Find all unit pairs of $G$.
- For each unit pair $(A, B)$ new grammar all productions $A \rightarrow \alpha$ where $B \rightarrow \alpha \in P$ and $B \rightarrow \alpha$ is not a unit production.

Example 8.5

\[
\begin{align*}
I & \rightarrow a | b | la | lb | I0 | I1 \\
F & \rightarrow I | (E) \\
T & \rightarrow F | T \ast F \\
E & \rightarrow T | E + T
\end{align*}
\]

\[
\begin{align*}
I & \rightarrow a | b | la | lb | I0 | I1 \\
F & \rightarrow (E) | a | b | la | lb | I0 | I1 \\
T & \rightarrow T \ast F | (E) | a | b | la | lb | I0 | I1 \\
E & \rightarrow E + T | T \ast F | (E) | a | b | la | lb | I0 | I1
\end{align*}
\]
Theorem 8.3 (Normal Form of CFGs)

If $G$ is a CFG, $L(G) - \{\lambda\} \neq \emptyset$, then there is a CFG $G_1$ such that $L(G_1) = L(G) - \{\lambda\}$ and $G_1$ has no $\lambda$-productions, unit productions, or useless symbols.

Proof.

Proof outline:

- Start by eliminating $\lambda$-productions.
- Eliminate unit productions. This does not introduce $\lambda$-productions.
- Eliminate useless symbols. This does not introduce any new production.
Definition 8.5 (Chomsky Normal Form)

A CFG grammar $G = (V, T, P, S)$ that has no useless symbols and all productions are in one of two forms:

- $A \rightarrow BC$, $A, B, C \in V$,
- $A \rightarrow a$, $A \in V$, $a \in T$,

is said to be in **Chomsky Normal Form (ChNF)**.

To put a grammar to ChNF, we need two additional steps:

- Bodies of length 2 or more consist only of variables.
- Break bodies of length 3 or more to bodies of two variables.

For every reachable terminal $a$

- create a new variable, say $A$,
- add one production rule $A \rightarrow a$,
- Use $A$ in place of $a$ everywhere $a$ appears in a body of length 2 or more.

For a production $A \rightarrow B_1 \ldots B_k$ introduce $k - 2$ variables $C_i$

And productions $A \rightarrow B_1 C_1, C_1 \rightarrow B_2 C_2, \ldots, C_{k-2} \rightarrow B_{k-1} B_k$. 
Example 8.6

$I \rightarrow a|b|IA|IB|IZ|IU$

$F \rightarrow LER|a|b|IA|IB|IZ|IU$

$T \rightarrow TMF|LER|a|b|IA|IB|IZ|IU$

$E \rightarrow EPT|TMF|LER|a|b|IA|IB|IZ|IU$

$A \rightarrow a$

$B \rightarrow b$

$Z \rightarrow 0$

$U \rightarrow 1$

$P \rightarrow +$

$M \rightarrow *$

$L \rightarrow ( $

$R \rightarrow )$

$F \rightarrow LC_{3}|a|b|IA|IB|IZ|IU$

$T \rightarrow TC_{2}|LC_{3}|a|b|IA|IB|IZ|IU$

$E \rightarrow EC_{1}|TC_{2}|LC_{3}|a|b|IA|IB|IZ|IU$

$C_{1} \rightarrow PT$

$C_{2} \rightarrow MF$

$C_{3} \rightarrow ER$

$I, A, B, Z, U, P, M, L, R$ as before

Theorem 8.4 (ChNF)

If $G$ is a CFG, $L(G) - \{\lambda\} \neq \emptyset$, then there is a grammar $G_{1}$ in Chomsky Normal Form such that $L(G_{1}) = L(G) - \{\lambda\}$. 
Lemma (The Size of Parse Trees)

Suppose we have a parse tree according the a ChNF grammar $G = (V, T, P, S)$ that yield a terminal string $w$. If the length of the longest path is $n$, then $|w| \leq 2^{n-1}$.

Proof.

By induction on $n$, BASIS: $|a| = 1 = 2^0$, INDUCTION: $2^{n-2} + 2^{n-2} = 2^{n-1}$. 

Lemma ((Consequence))

Let us have a parse tree for a ChNF grammar $G = (V, T, P, S)$ that yield a terminal string $w$; $|w| > 2^{n-1}$. Then the longest path of the tree has more than $n$ edges.
Theorem 8.5 (Pumping Lemma for Context Free Languages)

Let $L$ be a CFL. Then there exists constants $p, q \in \mathbb{N}$ such that any $z \in L, |z| > p$ can be written $z = uvwxy$ subject to:

- $|vwx| \leq q$.
- $vx \neq \lambda$.
- $\forall i \geq 0, uv^iwx^iy \in L$.

Proof Idea:
- take the parse tree for $z$
- find the longest path
- there must be two equal variables
- these variables define two subtrees
- the subtrees define partition of $z = uvwxy$
- we can move the tree $T^1$ ($i > 1$)
- or replace $T^1$ by $T^2$ ($i = 0$)
Proof: \(|z| > p : z = uvwxy, |vwx| \leq q, vx \neq \lambda, \forall i \geq 0uv^iwx^iy \in L\)

- we take the grammar in Chomsky NF (for \(L = \{\lambda\}\) and \(\emptyset\) aside).
- Let \(|V| = k\). We set \(p = 2^{k-1}, q = 2^k\).
- For \(z \in L, |z| \geq p\), the parse tree has a path \(z\) of length \(> k\)
  we denote the terminal of the longest path \(t\)
- At least two of the last \(k\) variables on the path to \(t\) are equal
- we take the couple \(A^1, A^2\) closest to \(t\) (it defines subtrees \(T^1, T^2\))
- the path from \(A^1\) to \(t\) is the longest in \(T^1\) and the length is maximally \(k + 1\)
  the yield of \(T^1\) is no longer than \(2^k\) (so \(|vwx| \leq q\))
- there are two paths from \(A^1\) (ChNF), one to \(T^2\) other to the rest of \(vx\)
  ChNF not nullable, so \(vx \neq \lambda\)
- derivation of the word \((A^1 \Rightarrow^* vA^2x, A^2 \Rightarrow^* w)\)
  \(S \Rightarrow^* uA^1y \Rightarrow^* uvA^2xy \Rightarrow^* uvwxy\)
- if we move \(A^2\) to \(A^1\) \((i = 0)\)
  \(S \Rightarrow^* uA^2y \Rightarrow^* uwy\)
- if we move \(A^1\) to \(A^2\) \((i = 2, 3, \ldots)\)
  \(S \Rightarrow^* uA^1y \Rightarrow^* uvA^1xy \Rightarrow^* uvvA^2xxy \Rightarrow^* uvvwxxy\)
"Adversary game" as for regular languages:

- Pick a language \( L \) that is not CFL.
- Our 'adversary' gets to pick \( p, q \) (or \( n \)), which we do not know.
- We get to pick \( z \), and we may use \( p \) as a parameter.
- Our adversary gets to break \( z \) into \( uvwx \), subject to \( |vwx| \leq q \) and \( vx \neq \lambda \).
- We 'win' the game, if by picking \( i \) and showing \( uv^iwx^iy \) is not in \( L \).

**Lemma (Not CFL)**

Following languages are not CFL:

- \( \{0^n1^n2^n | n \geq 1 \} \)
- \( \{0^i1^i2^i3^j | i \geq 1 \& j \geq 1 \} \)
- \( \{ww | w \text{ is in } \{0,1\}^* \} \)
Example 8.7 (non CFL)
Following language is not CFL
\{0^n 1^n 2^n \mid n \geq 1\}
- assume it were CFL
- we get \( p, q \) from the Pumping Lemma
- select \( k = \max(p, q) \), then \( |0^k 1^k 2^k| > p \)
- \( |vwx| \leq q \)
- we pump at most two different symbols
- the equality of symbols is violated – CONTRADICTION.

Example 8.8 (not a CFL)
Following language is not CFL
\{0^i 1^j 2^k \mid 0 \leq i \leq j \leq k\}
- assume it were CFL
- we get \( p, q \) from the Pumping Lemma
- select \( n = \max(p, q) \), then \( |0^n 1^n 2^n| > p \)
- \( |vwx| \leq q \)
- we pump at most two different symbols
- in the case of 0 (or 1), pump up – CONTRADICTION \( i \leq j \) (or \( j \leq k \))
- if 2 (or 1), pump down – CONTRADICTION \( j \leq k \) (or \( i \leq j \))
### Example 8.9 (non CFL)

Following language is not CFL

\[ \{0^i1^i2^i3^i | i, j \geq 1 \} \]

- Assume it were CFL
- We get \( p, q \) from the Pumping Lemma
- Select \( k = \max(p, q) \), then \( |0^k1^k2^k3^k| > p \)
- \( |vwx| \leq q \)
- We do pump both 0 and 2 nor 1 and 3
- The equality of symbols is violated – CONTRADICTION.

### Example 8.10 (not a CFL)

Following language is not CFL

\[ \{ww | w \text{ is in } \{0, 1\}^* \} \]

- Assume it were CFL
- We get \( p, q \) from the Pumping Lemma
- Select \( n = \max(p, q) \), then \( |0^n1^n0^n1^n| > p \)
- \( |vwx| \leq q \)
- We do not reach both positions of 0 nor both positions of 1
- The equality of symbols is violated – CONTRADICTION.
Infinity of Context Free Languages

Lemma

For any CFL \( L \) there exists two numbers \( m, n \in \mathbb{N} \) such that \( L \) is infinite iff
\[ \exists z \in L : m < |z| < n. \]

Proof:

We have from Pumping lemma \( p, q \), set \( m = p, n = p + q \)
\[ \Leftarrow p < |z|, \text{ therefore we may pump } z \Rightarrow L \text{ is infinite} \]
\[ \Rightarrow L \text{ is infinite} \Rightarrow \exists z \in L : p = m < |z|. \]
we take the shortest \( z; |z| > p \) and prove \( |z| \leq n = p + q \) by contradiction

Assume \( p + q < |z|, z \) may be pumped down, therefore \( |z^1| < |z| \)
we remove part no longer than \( q \), therefore \( p < |z^1| \)
CONTRADICTION \( z \) was shortest.

Faster algorithm:

take reduced grammar \( G \) in ChNF so that \( L = L(G) \)
create an oriented graph
- nodes = variables, edges = \{ \( (A, B), (A, C) \) for \( A \to BC \in P_G \) \}
- we look for an oriented cycle (if it exists \( \Rightarrow \) the language is infinite).
Generalizations of the Pumping Lemma

Pumping lemma is only an implication.

Example 8.11 (non CFL, that can be pumped)

\[ L = \{a^i b^j c^k d^l | i = 0 \lor j = k = l\} \] is not a CFL but it can be pumped.

- \( i = 0 : b^j c^k d^l \) can be pumped in any letter
- \( i > 0 : a^i b^n c^n d^n \) can be pumped in \( a^* \)

A solution?

- generalisations (Ogden lemma and others)
  - pumping marked symbols
- closure properties.
Example 9.1

1. $I \rightarrow a | b | Ia | Ib | I0 | I1$
2. $F \rightarrow I(\langle E \rangle)$
3. $T \rightarrow F | T \ast F$
4. $E \rightarrow T | E + T$.

Compile: (stack + two registers):

1. $E \rightarrow E + T$ ... pop r1; pop r2; add r1, r2; push r2
2. $E \rightarrow T$
3. $T \rightarrow T \ast F$ ... pop r1; pop r2; mul r1, r2; push r2
4. $T \rightarrow F$
5. $F \rightarrow (\langle E \rangle)$
6. $F \rightarrow a$ ... push a

- 'a + a * a' yield by the rules 1, 2, 4, 6, 3, 4, 6, 6
- reverse the sequence
  6, 6, 3, 6, 1
- and take the corresponding code
  push a; push a; pop r1; pop r2; mul r1, r2; push r2; push a; pop r1; pop r2; add r1, r2; push r2.
Cocke-Younger-Kasami algorithm for membership in CFL

Exponentially to $|w|$: try all parse trees of the appropriate length for $L$.

Algorithm: CYK algorithm, in time $O(n^3)$

- Let us have a ChNF grammar $G = (V, T, P, S)$ generating the language $L$ and a word $w = a_1a_2\ldots a_n \in T^*$.
- We create triangular table (right),
  - horizontal axis is $w$
  - $X_{ij}$ are sets of variables $A$ so that $A \Rightarrow^* a_ia_{i+1}\ldots a_j$.

**Basis**: $X_{ii} = \{A; A \rightarrow a_i \in P\}$

**Induction**: $X_{ij} = \{A \rightarrow BC; B \in X_{ik}, C \in X_{k+1,j}\}$

- We fill the table bottom up.
- If $S \in X_{1,n}$ then $w \in L(G)$. 
Example 9.2 (CYK algorithm)

Grammar

\[
\begin{align*}
S & \rightarrow AB | BC \\
A & \rightarrow BA | a \\
B & \rightarrow CC | b \\
C & \rightarrow AB | a
\end{align*}
\]

BASIS \( X_{ii} = \{ A; A \rightarrow a_i \in P \} \)

DUCT. \( X_{ij} = \{ A \rightarrow BC; B \in X_{ik}, C \in X_{k+1,j} \} \)

Filled from the bottom up:

\[
\begin{array}{cccc}
S, A, C \\
B & A, C & S, C & S, A \\
B & A, C & S, A \\
\end{array}
\]

b a a b a
Greibach Normal Form

- It would be nice to know which rule to select
- Especially difficult is the left recursion $A \rightarrow A\alpha$

**Definition 9.1**

Greibach Normal Form or a CFG We say a grammar $G$ is in **Greibach Normal Form** iff all rules are in the form $A \rightarrow a\beta$, where $a \in T$, $\beta \in V^*$ (a string of variables).

- The terminal in the body of the rule helps to select appropriate rule
- Especially if we have a unique rule with this terminal.

**Theorem 9.1 (Greibach Normal Form)**

*For each CFL language $L$ there exists a CFG grammar $G$ in Greibach Normal Form such that $L(G) = L - \{\lambda\}$.***
Lemma (Joining the rules)

Let us have a rule $A \rightarrow \alpha B \beta$ in grammar $G$ and $B \rightarrow \omega_1, \ldots, B \rightarrow \omega_k$ are all rules for $B$.

- If we replace the rule $A \rightarrow \alpha B \beta$ by rules $A \rightarrow \alpha \omega_1 \beta, \ldots, A \rightarrow \alpha \omega_k \beta$
- we get an equivalent grammar.

Proof:

$A \Rightarrow \alpha B \beta \Rightarrow^* \alpha \mid B \beta \Rightarrow \alpha \mid \omega_i \beta$ in the original grammar

$A \Rightarrow \alpha \omega_i \beta \Rightarrow^* \alpha \mid \omega_i \beta$ in the new grammar
Lemma (Left Recursion)

Let \( A \rightarrow A\omega_1, \ldots, A \rightarrow A\omega_k \) be all productions with left recursion in the grammar \( G \) for \( A \) and \( A \rightarrow \alpha_1, \ldots, A \rightarrow \alpha_m \) be all other rules for \( A \), \( Z \) is a new variable.

Then, by replacing these rules by the rules:
1. \( A \rightarrow \alpha_i, A \rightarrow \alpha_iZ, Z \rightarrow \omega_j, Z \rightarrow \omega_jZ \) or
2. \( A \rightarrow \alpha_iZ, Z \rightarrow \omega_jZ, Z \rightarrow \lambda \)

we get an equivalent grammar.

Proof:

\[
\begin{align*}
A \Rightarrow A\omega_{i_n} \Rightarrow \ldots \Rightarrow A\omega_{i_1} \ldots \omega_{i_n} & \Rightarrow \alpha_j\omega_{i_1} \ldots \omega_{i_n} \quad \text{(G)} \\
A \Rightarrow \alpha_jZ \Rightarrow \alpha_j\omega_{i_1}Z \ldots \Rightarrow \alpha_j\omega_{i_1} \ldots \omega_{i_{n-1}}Z & \Rightarrow \alpha_j\omega_{i_1} \ldots \omega_{i_n} \quad \text{(1)} \\
A \Rightarrow \alpha_jZ \Rightarrow \alpha_j\omega_{i_1}Z \ldots \Rightarrow \alpha_j\omega_{i_1} \ldots \omega_{i_n}Z & \Rightarrow \alpha_j\omega_{i_1} \ldots \omega_{i_n} \quad \text{(2)}
\end{align*}
\]

Theorem ((9.1) Greibach Normal Form)

For each CFL language \( L \) there exists a CFG grammar \( G \) in Greibach Normal Form such that \( L(G) = L - \{\lambda\} \).
Proof: Greibach Normal Form

join rules and remove left recursion

- we enumerate all variables \( \{A_1, \ldots, A_n\} \)
- we allow recursion only in the form \( A_i \rightarrow A_j \omega \), where \( i < j \)

we iterate \( i \) from 1 to \( n \)

\[ A_i \rightarrow A_j \omega \quad \text{for } j < i \]
removed by joining the rules

\[ A_i \rightarrow a \omega \quad (a \in T), \ Z_i \rightarrow \omega \]

- rules with \( A_i \) (original variables) only in the form \( A_i \rightarrow a \omega \)

iteratively join the rules for \( i \) from \( n \) to 1 (for \( n \) already holds)

- rules with \( Z_i \) (new variables) only in the form \( Z_i \rightarrow a \omega \)
  - In none rule for \( Z_i \) the body begins with \( Z_j \)
  - either it is in required form or we join it with the rule \( A_j \rightarrow a \omega \)

- we remove terminals inside rules.
Reduction to GreibachNF Example

Original Grammar

\[
E \rightarrow E + T | T \\
T \rightarrow T * F | F \\
F \rightarrow (E) | a
\]

Left Recursion Removed

\[
E \rightarrow T | TE \\
E | \rightarrow + T | + TE \\
T \rightarrow F | FT \\
T | \rightarrow * F | * FT \\
F \rightarrow (E) | a
\]

(almost) Greibach Normal Form

\[
E \rightarrow (E)|a|(E)T|aT|(E)E|aE|(E)TE|aTE
\]

\[
E | \rightarrow + T | + TE \\
T \rightarrow (E)|a|(E)T|aT \\
T | \rightarrow * F | * FT \\
F \rightarrow (E) | a
\]

Greibach Normal Form

\[
E \rightarrow (EP|a|(EPT|aT|(EPE|aE|(EPT|E|aT|E
\]

\[
E | \rightarrow + T | + TE \\
T \rightarrow (EP|a|(EPT|aT \\
T | \rightarrow * F | * FT \\
F \rightarrow (EP|a \\
P \rightarrow )
\]
A PDA $P = (Q, \Sigma, \gamma, \delta, q_0, z_0, F)$ is deterministic PDA iff both:

- $\delta(q, a, X)$ has at most one member $\forall q \in Q, a \in \Sigma \cup \{\lambda\}$ and $X \in \Gamma$.
- If $\delta(q, a, X)$ is nonempty for some $a \in \Sigma$, then $\delta(q, \lambda, X)$ must be empty.

The language $L_{wwr}$ before is CFL that has no DPDA. The second condition assure that there is no choice between $\lambda$ transition and consumption of the input symbol. By putting a 'center-mark' $c$ in the middle $L_{wcwr} = \{wcw^R | w \in (0+1)^*\}$ it is recognizable by DPDA.
\[
RL \subset L(P_{DPDA}) \subset CFL \supset N(P_{DPDA}).
\]

**Theorem 9.2**

If \( L \) is a regular language, then \( L = L(P) \) for some DPDA \( P \).

**Proof.**

Essentially, a DPDA can simulate a deterministic FA and ignores the stack (keeping \( Z_0 \) there). 

**Definition 9.3 (prefix property)**

Say that a language \( L \) has the **prefix property** if there are no two different strings \( x, y \in L \) such that \( x \) is a prefix of \( y \).

**Example 9.4**

The language \( L_{wcwr} \) has the prefix property.

On the other hand, \( L = \{0\}^* \) does not have the prefix property.
Theorem 9.3 \((L \in N(P_{DPDA}) \text{ iff prefix property } L \text{ and } L \in L(P'_{DPDA}))\)

A language \(L\) is \(N(P)\) for some DPDA \(P\) iff \(L\) has the prefix property and \(L\) is \(L(P')\) for some DPDA \(P'\).

Lemma

The language \(L_{wcwr}\) is accepted by a DPDA but it is not regular.

To prove non-regularity use pumping lemma on the string \(0^n c 0^n\).

Lemma

The language \(L_{wwr}\) is CFL but not accepted by any DPDA.

Formal proof is complex.
The idea is that words \(0^n 110^n 0^n 110^n\), \(0^n 110^n 0^m 110^m\) are both accepted or both rejected since in the middle the stack is empty and there is no other way to remember any number \(n\) than on the stack.
The first word is in \(L_{wwr}\), the second is not.
Theorem 9.4 \((L = N(P_{DPDA}) \rightarrow L \text{ has an unambiguous CFG.})\)

- Let \(L = N(P)\) for some DPDA \(P\). Then \(L\) has an unambiguous CFG.
- Let \(L = L(P)\) for some DPDA \(P\). Then \(L\) has an unambiguous CFG.

The opposite is not true since \(L_{wwr}\) has an unambiguous grammar \(S \rightarrow 0S0|1S1|\lambda\) but is not a DPDA language.

**Proof.**

- \(N(P)\): The construction of a CFG from a PDA accepting by empty stack applied to NPDA yields an unambiguous CFG \(G\).
- \(L(P)\):
  - Construct a language that has the prefix property by adding a new symbol $ at the end of any word \(w \in L\).
  - Construct a DPDA \(P'\) for that \(L' = N(P')\).
  - Construct a grammar \(G'\) generating the language \(N(P')\).
  - Construct \(G\) such that \(L(G) = L\). Just get rid of $ keeping it as a variable and adding a production $ \rightarrow \emptyset\). All other productions remain the same as in \(G'\).
  - \(G\) is unambiguous since \(G'\) is unambiguous and we did not add ambiguity.
\( \mathcal{L}_3 \) regular lang. 
\{0, 00\} 
\{010\}

prefix prop & DPDA 
\{0^n1^n; n > 0\}

deterministic PDA 
\{0^n1^m; 0 < n \leq m\}

context free (=CFL) 
\{ww^R|w \in \{0, 1\}\}

\( \mathcal{L}_1 \)
\{a^i b^i c^i|i = 0, 1, \ldots\}

context (=CL) 
\( \mathcal{L}_0 \)

\( \mathcal{L}_0 \)

recursively enumerable
Theorem 9.5 (CFLs are closed on substitution)

If \( L \) is a CFL over \( \Sigma \), and \( s \) is a substitution on \( \Sigma \) such that \( s(a) \) is a CFL for each \( a \in \Sigma \), then \( s(L) \) is CFL.

Proof.

- Idea: Replace each \( a \) by the start symbol of a CFG for language \( s(a) \).
- First, rename variables to be unique in all \( G = (V, T, P, S) \), \( G_a = (V_a, T_a, P_a, S_a) \), \( a \in \Sigma \).
- We construct a new grammar \( G' = (V', T', P', S) \) for \( s(L) \):
  - \( V' = V \cup \bigcup_{a \in \Sigma} V_a \)
  - \( T' = \bigcup_{a \in \Sigma} T_a \)
  - \( P' = \bigcup_{a \in \Sigma} P_a \cup \{ p \in P \text{ with all } a \text{ replaced by } S_a \} \).

\( G' \) generates the language \( s(L) \).
Substitution and homomorphism (defined earlier)

Let us have a language $L$ over the alphabet $\Sigma$.

**Substitution** $\sigma$; $\forall a \in \Sigma : \sigma(a) = L_a$ language over the alphabet $\Sigma_a$ where $\sigma(a) \subseteq \Sigma_a^*$

maps words to languages:

- $\sigma(\lambda) = \{\lambda\}$,
- $\sigma(a_1 \ldots a_n) = \sigma(a_1) \ldots \sigma(a_n)$ (concatenation), that is $\sigma : \Sigma^* \rightarrow P((\bigcup_{a \in \Sigma} \Sigma_a)^*)$
- $\sigma(L) = \bigcup_{w \in L} \sigma(w)$.

**Homomorphism** $h$, $\forall a \in \Sigma : h(a) \in \Sigma_a$ maps words to words

- $h(\lambda) = \{\lambda\}$,
- $h(a_1 \ldots a_n) = h(a_1) \ldots h(a_n)$ (concatenation), that is $h : \Sigma^* \rightarrow (\bigcup_{a \in \Sigma} \Sigma_a)^*$
- $h(L) = \{h(w) \mid w \in L\}$.

**Inverse homomorphism** maps words backwards

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

Automata and Grammars

CYK algorithm, Closure properties of CFLs

August 9, 2019 201 / 187 - 223
Theorem ((9.5) CFLs are closed under substitution)

Let’s have a CFL $L$ over $\Sigma$ and a substitution $\sigma$ on $\Sigma$ such that $\sigma(a)$ is CFL $\forall a \in \Sigma$. The also $\sigma(L)$ is context–free language (CFL).

Proof:

- Idea: leaves in the parse tree generate further trees.
- We rename all variables to unique names in all grammars $G = (V, T, P, S)$, $G_a = (V_a, T_a, P_a, S_a)$, $a \in \Sigma$.
- We construct a new grammar $G' = (V', T', P', S)$ for $\sigma(L)$:
  - $V' = V \cup \bigcup_{a \in \Sigma} V_a$
  - $T' = \bigcup_{a \in \Sigma} T_a$
  - $P' = \bigcup_{a \in \Sigma} P_a \cup \{p \in P \text{ with all } a \text{ replaced by } S_a\}$.
- $G'$ generates the language $\sigma(L)$. 

---

Automata and Grammars  CYK algorithm, Closure properties of CFLs  9

August 9, 2019  202 / 187 - 223
Substitution on CFL’s

Example 9.5 (substitution)

\[ L = \{a^ib^j \mid 0 \leq i \leq j\} \]

\[ \sigma(a) = L_1 = \{c^id^i \mid i \geq 0\} \]

\[ \sigma(b) = L_2 = \{c^i \mid i \geq 0\} \]

\[ \sigma(L): S \rightarrow S_1SS_2|SS_2|\lambda, \quad S_1 \rightarrow cS_1d|\lambda, \quad S_2 \rightarrow cS_2|\lambda \]

Theorem 9.6 (homomorphism)

Context–free languages are closed under homomorphism.

Proof:

- Direct consequence: homomorphism is a special case of the substitution.
- Each terminal \(a\) in the parse tree replace by \(h(a)\).
Theorem 9.7 (CFL are closed under inverse homomorphism)

Let us have a CFL language \( L \) and a homomorphism \( h \). Then \( h^{-1}(L) \) is also context-free.

If \( L \) is deterministic CFL, \( h^{-1}(L) \) is also deterministic CFL.

Idea:

- use PDA recognizing the language \( L \)
- read a letter \( a \), place \( h(a) \) into an inner buffer
- simulate \( M \), the input is taken from the buffer
- if the buffer is empty, read next letter from the real input
- the input \( w \) is accepted iff the buffer is empty and \( M \) is in an accepting state

! buffer is finite therefore it can be modeled in the finite state (states are tuples, the state and the buffer).
Proof:

for $L$ we have a PDA $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ (by final state)

$h : \Xi \rightarrow \Sigma^*$

we define PDA $M' = (Q', \Xi, \Gamma, \delta', [q_0, \lambda], Z_0, F \times \{\lambda\})$ where

$Q' = \{[q, u] | q \in Q, u \in \Sigma^*, \exists (a \in \Xi) \exists (v \in \Sigma^*) h(a) = vu\}$  \hspace{1cm} u \text{ is the buffer}

$\delta'([q, u], \lambda, Z) = \{([p, u], \gamma) | (p, \gamma) \in \delta(q, \lambda, Z)\}$

$\cup \{([p, v], \gamma) | (p, \gamma) \in \delta(q, b, Z), u = bv\}$  \hspace{1cm} \text{reads the buffer}

$\delta'([q, \lambda], a, Z) = \{([q, h(a)], Z)\}$

For a deterministic PDA $M$ is $M'$ also deterministic.
Theorem 9.8 (CFL are closed under union, concatenation, closure, reverse)

CFL are closed under union, concatenation, closure (\(\ast\)), positive closure (\(+\)), reverse \(w^R\).

Proof:

- Union:
  - if \(V_1 \cap V_2 \neq \emptyset\) rename variable to unique names,
  - add new variable \(S_{\text{new}}\) and the rule \(S_{\text{new}} \rightarrow S_1|S_2\)

- concatenation \(L_1.L_2\)
  \(S_{\text{new}} \rightarrow S_1S_2\) (pro \(V_1 \cup V_2 = \emptyset\), otherwise rename variables)

- iteration \(L^* = \bigcup_{i \geq 0} L^i\)
  \(S_{\text{new}} \rightarrow SS_{\text{new}}|\lambda\)

- positive iteration \(L^+ = \bigcup_{i \geq 1} L^i\)
  \(S_{\text{new}} \rightarrow SS_{\text{new}}|S\)

- reverse \(L^R = \{w^R | w \in L\}\)
  \(X \rightarrow \omega^R\) reverse the right side of all rules.
Quotient with a regular language

**Lemma**

Context–free languages are closed under left and right quotient with regular languages.

**Idea:**
- PDA and FA run in parallel, do not read the input
- If FA is in an accepting state, PDA begins to read the real input

**Proof:**

1. FA $A_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$
2. PDA $M_2 = (Q_2, \Sigma, \Gamma, \delta_2, q_2, Z_0, F_2)$
3. We define PDA $M = (Q', \Sigma, \Gamma, \delta, (q_1, q_2), Z_0, F_2)$ where $Q' = (Q_1 \times Q_2) \cup Q_2$ couples of states for parallel run
   
   $$\delta((p, q), \lambda, Z) = \{((p', q'), \gamma) | \exists (a \in \Sigma) p' \in \delta_1(p, a) \& (q', \gamma) \in \delta_2(q, a, Z)\}$$
   $$\cup \{((p, q'), \gamma) | (q', \gamma) \in \delta_2(q, \lambda, Z)\}$$
   $$\cup \{(q, Z) | p \in F_1\}$$

4. $\delta(q, a, Z) = \delta_2(q, a, Z), a \in \Sigma \cup \{\lambda\}, q \in Q_2, Z \in \Gamma$
5. Obviously $L(M) = L(A_1) \setminus L(M_2)$.
6. Right quotient from closure properties of reverse and the left quotient

   $$L / M = (M^R \setminus L^R)^R.$$
Intersection of context–free languages

Example 9.6 (CFL are not closed under intersection)

- The language \( L = \{0^n1^n2^n | n \geq 1\} = \{0^n1^n2^i | n, i \geq 1\} \cap \{0^i1^n2^n | n, i \geq 1\} \)

is not CFL, even though both members of the intersection are CFL’s, they are also deterministic CFLs.

\[
\{0^n1^n2^i | n, i \geq 1\} \quad \{S \rightarrow AC, A \rightarrow 0A1|\lambda, C \rightarrow 2C|\lambda\} \\
\{0^i1^n2^n | n, i \geq 1\} \quad \{S \rightarrow AB, A \rightarrow 0A|\lambda, B \rightarrow 1B2|\lambda\}
\]

- the intersection is not CFL from the pumping lemma.

two PDAs in parallel
- FA unit can be joined (as in the finite automata)
- reading the input can be joined (one automaton can wait)
- two stacks is not possible to join to one stack

two general stacks  = Turing machine
= recursively enumerable languages \( \mathcal{L}_0 \)
Intersection of an CFL an a regular language

**Theorem 9.9** (Both CFL and DCFL are closed under intersection with a regular language)

- Let’s have a CFL $L$ and a regular language $R$. Then $L \cap R$ is context free.
- Let’s have a deterministic CFL $L$ and a regular language $R$. Then $L \cap R$ is deterministic CFL.

**Proof:**

- PDA and FA can be joined
  - FA $A_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$
  - PDA accepts by the final state $M_1 = (Q_2, \Sigma, \Gamma, \delta_2, q_2, Z_0, F_2)$
- new automaton $M = (Q_1 \times Q_2, \Sigma, \Gamma, \delta, (q_1, q_2), Z_0, F_1 \times F_2)$
  - $((r, s), \alpha) \in \delta((p, q), a, Z)$ iff
    - $a \neq \lambda$: $r = \delta_1(p, a) \& (s, \alpha) \in \delta_2(q, a, Z)$
    - $a = \lambda$: $(s, \alpha) \in \delta_2(q, \lambda, Z)$

- obviously $L(M) = L(A_1) \cap L(M_2)$
- automata run in parallel.
Example 9.7

The language $L = \{0^i1^j2^k3^l | i = 0 \lor j = k = l \}$ is not CFL.

Proof: Proof by contradiction:

- Assume $L$ to be CFL
- $L_1 = \{0^i1^j2^k3^l | i, j, k \geq 0 \}$ is a regular language
  - \{ $S \to 0B, B \to 1B|C, C \to 2C|D, D \to 3D|\lambda$ \}
- $L \cap L_1 = \{0^i1^j2^k3^l | i \geq 0 \}$ is not CFL $\Rightarrow$ CONTRADICTION

$L$ is context language

\[
\begin{align*}
S & \to B_1|0A \\
B_1 & \to 1B_1|C_1, C_1 \to 2C_1|D_1, D_1 \to 3D_1|\lambda \\
A & \to 0A|P \\
P & \to 1PCD|\lambda \\
DC & \to CD \text{ we rewrite to } \{DC \to XC, XC \to XY, XY \to CY, CY \to CD\}
\end{align*}
\]

\[
\begin{align*}
1C & \to 12, 2C \to 22, 2D \to 23, 3D \to 33
\end{align*}
\]
Difference and the complement

**Theorem 9.10 (Difference with a regular language)**

Let's have a CFL language $L$ and a regular language $R$. Then:

- $L - R$ is CFL.

**Proof.**

$L - R = L \cap \overline{R}$, $\overline{R}$ is regular.

**Theorem 9.11 (CFL are not closed under difference nor complement)**

The class of the context–free languages is not closed under the difference nor the complement.

**CFL are not closed under difference nor complement.**

Let us have CFL languages $L, L_1, L_2$, and a regular language $R$. Then:

- $\overline{L}$ is not necessary context–free.
- $L_1 - L_2$ is not necessary context–free.
- $L_1 \cap L_2 = \overline{L_1} \cup \overline{L_2}$.
- $\Sigma^* - L$ not necessary a CFL.
Closure properties of deterministic CFL’s

- Good programming languages are deterministic CFL’s.
- Deterministic context–free languages
  - are not closed under intersection
  - are closed under intersection with a regular language
  - are closed under inverse homomorphism.

**Theorem 9.12**

The complement of a deterministic CFL is a deterministic CFL.

**Proof:**

- idea: accept iff $q \in Q - F$
- undefined steps ‘caught’ by new bottom of the stack $Z_B$
- cycle can be recognized by a counter
- when reading the input is finished check transition through an accepting state (may be more $\lambda$ transitions)
  - if an accepting state was visited, original PDA accepted, do not accept.
Deterministic CFL’s are not closed under union

Example 9.8 (DCFL are not closed under union)

The language \( L = \{ a^i b^j c^k | i \neq j \lor j \neq k \lor i \neq k \} \) is a CFL, but not a deterministic CFL.

Proof.

Due to closeness of DCFL’s under complement would be deterministic CFL also \( \overline{L} \cap a^* b^* c^* = \{ a^i b^j c^k | i = j = k \} \), and we know it is not a CFL (pumping lemma).

Example 9.9 (DCFL are not closed under homomorphism)

Languages \( L_1 = \{ a^i b^j c^k | i = j \} \), \( L_2 = \{ a^i b^j c^k | j = k \} \) are deterministic context–free.

- The language \( 0L_1 \cup 1L_2 \) is deterministic context free
- The language \( 1L_1 \cup 1L_2 \) is not a deterministic CFL
  
  \[ \text{let be} \quad h(0) = 1 \]
  \[ h(x) = x \text{ for other symbols} \]
  \[ h(0L_1 \cup 1L_2) = 1L_1 \cup 1L_2. \]
## Closure properties in brief

<table>
<thead>
<tr>
<th>language</th>
<th>regular (RL)</th>
<th>context–free</th>
<th>deterministic CFL</th>
</tr>
</thead>
<tbody>
<tr>
<td>union</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>intersection</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>$\cap$ with RL</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>complement</td>
<td>YES</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>homomorphism</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>inverse hom.</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
</tbody>
</table>
Chomsky hierarchy

- Grammar types according to productions allowed.

- **Type 0** (recursively enumerable languages $L_0$)
  - general rules $\alpha \rightarrow \beta$, $\alpha, \beta \in (V \cup T)^*$, $\alpha$ contains at least one variable

  
  **Type 1** (context languages $L_1$)
  - productions of the form $\alpha A \beta \rightarrow \alpha \omega \beta$
    
    $A \in V, \alpha, \beta \in (V \cup T)^*, \omega \in (V \cup T)^+$
  - with only exception $S \rightarrow \lambda$, then $S$ does not appear at the right side of any production

- **Type 2** (context free languages $L_2$)
  - productions of the form $A \rightarrow \omega$, $A \in V, \omega \in (V \cup T)^*$

- **Type 3** (regular (right linear) languages $L_3$)
  - productions of the form $A \rightarrow \omega B, A \rightarrow \omega, A, B \in V, \omega \in T^*$
Context Grammars

- productions in the form $\alpha A \beta \rightarrow \alpha \omega \beta$
  
  $A \in V$, $\alpha, \beta \in (V \cup T)^{*}$, $\omega \in (V \cup T)^+$

- with one exception $S \rightarrow \lambda$,
  
  in this case $S$ does not appear in the body (right side) of any rule

- variable $A$ rewrites only in the context $\alpha, \beta$

- $S \rightarrow \lambda$ is used only to add $\lambda$ to the language

Example 9.10 (a context language)

$L = \{a^n b^n c^n | n \geq 1\}$ is a context language that is not context–free.

**Grammar:**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \rightarrow aSBC</td>
<td>aBC$</td>
</tr>
<tr>
<td>$CB \rightarrow BC$</td>
<td>(*)</td>
</tr>
<tr>
<td>$bB \rightarrow bb$</td>
<td>$CB \rightarrow XB, XB \rightarrow XY, XY \rightarrow BY, BY \rightarrow BC$</td>
</tr>
<tr>
<td>$bC \rightarrow bc$</td>
<td>$CB \rightarrow XB, XB \rightarrow XY, XY \rightarrow BY, BY \rightarrow BC$</td>
</tr>
<tr>
<td>$cC \rightarrow cc$</td>
<td>$CB \rightarrow XB, XB \rightarrow XY, XY \rightarrow BY, BY \rightarrow BC$</td>
</tr>
</tbody>
</table>
Definition 9.4 (Separated grammar)

A grammar is **separated**, iff all productions are in the form \( \alpha \to \beta \), where:
- either \( \alpha, \beta \in V^+ \) (non-empty sequences of variables)
- or \( \alpha \in V \) and \( \beta \in T \cup \{\lambda\} \).

Lemma

For any grammar \( G \) there exists equivalent separated grammar \( G' \).

Proof:
- Let \( G = (V, T, P, S) \)
- for each terminal \( a \in V \) we introduce new variable \( A' \).
- in productions in \( P \)
  - we replace terminals by corresponding variables
  - we add rules \( A' \to a \)
- Resulting grammar is separated and obviously \( L(G) = L(G') \).
Definition 9.5 (monotone grammar)

A grammar is **monotone**, iff for each rule \((\alpha \rightarrow \beta) \in P\) holds \(|\alpha| \leq |\beta|\). Monotone grammar never shortens sentential form during the derivation.

Lemma

*To any monotone grammar there exists an equivalent context grammar.*

Proof:

- first, find an equivalent separated grammar
  - this does not violate the monotony (and rules \(A' \rightarrow a\) are context rules)
- remaining productions \(A_1 \ldots A_m \rightarrow B_1 \ldots B_n, m \leq n\) replace by productions with new variables \(C\)

\[
\begin{align*}
A_1 A_2 \ldots A_m & \rightarrow C_1 A_2 \ldots A_m & C_1 C_2 \ldots C_m & \rightarrow B_1 C_2 \ldots C_m \\
C_1 A_2 \ldots A_m & \rightarrow C_1 C_2 \ldots A_m & B_1 C_2 \ldots C_m & \rightarrow B_1 B_2 \ldots C_m \\
\vdots & \vdots & \vdots & \vdots \\
C_1 \ldots C_{m-1} A_m & \rightarrow C_1 \ldots C_{m-1} C_m & B_1 \ldots B_{m-1} C_m & \rightarrow B_1 \ldots B_{m-1} B_m \ldots B_n
\end{align*}
\]
Example 9.11

The language \( L = \{a^i b^j c^k \mid 1 \leq i \leq j \leq k\} \) is a context language, that is not a context-free language.

Proof:

\[
\begin{align*}
S & \to aSBC \mid aBC \quad \text{generate symbols } a \\
BC & \to BBCC \quad \text{adds symbols } BC \\
C & \to CC \quad \text{adds symbols } C \\
CB & \to BC \quad \text{ordering of } B \text{'s and } C \text{'s (*)} \\
aB & \to ab \quad \text{start rewriting of } B \text{ to } b \\
bB & \to bb \quad \text{continue rewriting } B \text{ to } b \\
bC & \to bc \quad \text{start rewriting } C \text{ to } c \\
cC & \to cc \quad \text{continue rewriting } C \text{ to } c \\
\end{align*}
\]

(*) \( CB \to BC \) is not a context rule, we replace it with

\[
\begin{align*}
CB & \to XB, \quad XB \to XY, \quad XY \to BY, \quad BY \to BC
\end{align*}
\]
Other Characterization of Context Free Languages

- Until we only PUSH to the stack, we may remember the last stack symbol.
- finite automaton can simulate this.

- we need to POP the stack
  we need an infinite memory for this

- PUSH and POP is not arbitrary
  it is a stack, that is LIFO (last in, first out) structure

- we put the stack actions into a linear structure
  \[
  X \quad \text{PUSH}(X) \\
  X^{-1} \quad \text{POP}(X)
  \]

- PUSH(X) and POP(X) make a pair \( ZZ^{-1} B A A^{-1} C C^{-1} B^{-1} \)
  that behaves as a bracket in the sequence
Definition 9.6 (Dyck Language)

**Dyck Language** \( D_n \) is defined on the alphabet \( Z_n = \{ a_1, a_1^\dagger, \ldots, a_n, a_n^\dagger \} \) by the grammar:

\[
S \rightarrow \lambda | SS | a_1 Sa_1^\dagger | \ldots | a_n Sa_n^\dagger.
\]

**Observations:**

- Dyck language is context free
- Dyck language \( D_n \) describes correctly formed formulas with \( n \) pairs of brackets
- We may describe any context free language in the following form:

\[
L = h(D \cap R)
\]
Theorem 9.13 (Dyck Languages)

For any context free language $L$ there exists a regular language $R$ so that $L = h(D \cap R)$ for some Dyck language $D$ and a homomorphism $h$.

Proof:

- we have a PDA $P$ such that $L = N(P)$ (by empty stack)
- we transform instructions to $\delta(q, a, Z) \in (p, w), |w| \leq 2$
  - we split longer PUSH sequences by additional states
- regular language $R^\dagger$ contains expressions
  - $q^{-1}a a^{-1}Z^{-1}BAp$ for transition $\delta(q, a, Z) \ni (p, AB)$
  - similarly $\delta(q, a, Z) \in (p, A), \delta(q, a, Z) \in (p, \lambda)$
  - for $a = \lambda$ we leave $aa^{-1}$ out
- we define $R$ as follows: $Z_0q_0(R^\dagger)^*Q^{-1}$
- Dyck language is defined over the alphabet $\Sigma \cup \Sigma^{-1} \cup Q \cup Q^{-1} \cup \Gamma \cup \Gamma^{-1}$
- $D \cap Z_0q_0(R^\dagger)^*Q^{-1}$ describes correct sequences of PDA moves
  
  $Z_0q_0q_0^{-1}a a^{-1}Z_0^{-1}BAp^{-1}bb^{-1}A^{-1}qq^{-1}cc^{-1}B^{-1}rr^{-1}$

- homomorphism $h$ select the input word, that is
  - $h(a) = a$ for input symbols
  - $h(y) = \lambda$ for any other symbols
Summary of Chapter 9

- **Pushdown automata** A PDA is a non-deterministic FA couplet with a stack.
- **Moves of a Pushdown Automata**: \( \delta(p, a, X) = \{(q, \beta)\} \), \( p, q \in Q, a \in \Sigma \cup \{\lambda\}, X \in \Gamma, \beta \in \Gamma^* \).
- **Acceptance by Pushdown Automata**: by an empty stack \( N(P) \) or by final states \( L(P) \).
- **instantaneous Descriptions**: ID = state, remaining input and the stack. A transition \( \vdash \) between ID’s represents single moves of a PDA.
- **Pushdown Automata and Grammars** For (non-deterministic) PDA’s, the languages accepted either by final state or by empty stack, are exactly the context-free languages.
- **Deterministic Pushdown Automata**: It never has a choice of mover for a given state, input symbol (including \( \lambda \)) and the stack symbol, nor between move using a true input symbol and a move using \( \lambda \) symbol.
- **Acceptance by DPDA’s**: \( L \in N(P_{DPDA}) \) iff prefix property \( L \) and \( L \in L(P_{DPDA}') \).
- **The Languages Accepted by DPDA’s**: \( RL \subset L(P_{DPDA}) \subset CFG \) languages with unambiguous grammars, i.e. \( L(P_{DPDA}) \subset CFG \).
Introduction to Turing Machines

- General model of any computer
- We aim to show problems undecidable by any computer.
Turing Machine

Definition 10.1 (Turing Machine)

**Turing Machine (TM)** is the 7–tuple \( M = (Q, \Sigma, \Gamma, \delta, q_0, B, F) \) with the components:

- \( Q \) The finite set of **states** of the finite control.
- \( \Sigma \) The finite set of **input symbols**.
- \( \Gamma \) The complete set of **tape symbols**. Always \( \Gamma \supseteq \Sigma \), \( Q \cap \Gamma = \emptyset \).
- \( \delta \) The partial **transition function** \( (Q - F) \times \Gamma \to Q \times \Gamma \times \{L, R\} \).
  \( \delta(q, x) = (p, Y, D) \), where:
  - \( q \in Q \) is the current state.
  - \( X \in \Gamma \) is the current tape symbol.
  - \( p \) is the next state, \( p \in Q \).
  - \( Y \in \Gamma \) is written in the cell being scanned, replacing anything there.
  - \( D \in \{L, R\} \) is a **direction** in which the head moves (left, right).
- \( q_0 \in Q \) is the **start state**.
- \( B \in \Gamma \setminus \Sigma \). It appears initially in all but the finite number of initial cells that hold input symbols.
- \( F \subseteq Q \) The set of **final** or **accepting** states.
  - Note there are no transitions for accepting states.
Instantaneous Description for TM

**Definition 10.2 (Instantaneous Description (ID) for TM)**

Instantaneous Description (ID) for a Turing machine is a string

\[ X_1X_2 \ldots X_{i-1}qX_iX_{i+1} \ldots X_n \]

where

- \( q \) is the state of the Turing machine.
- The tape head is scanning the \( i \)th symbol from the left.
- \( X_1 \ldots X_n \) is the portion of the tape between the leftmost and the rightmost nonblank. As an exception, if the head is at any end one blank is added on that side.

**Definition 10.3 (Moves of a TM)**

We describe moves of a TM \( M \) by the \( \vdash_M, \vdash_M^*, \vdash_M^\ast \) notation as used for PDA’s.

For \( \delta(q, X_i) = (p, Y, R) \)

\[ X_1X_2 \ldots X_{i-1}qX_iX_{i+1} \ldots X_n \vdash_M X_1X_2 \ldots X_{i-2}pX_{i-1}YX_{i+1} \ldots X_n \]

For \( \delta(q, X_i) = (p, Y, R) \)

\[ X_1X_2 \ldots X_{i-1}qX_iX_{i+1} \ldots X_n \vdash_M X_1X_2 \ldots X_{i-1}YPX_{i+1} \ldots X_n. \]
A TM for \( \{0^n1^n; n \geq 1\} \)

**Definition 10.4 (TM accepts the language)**

A TM \( M = (Q, \Sigma, \Gamma, \delta, q_0, B, F) \) accepts the language \( L(M) = \{ w \in \Sigma^* : q_0 w \xrightarrow{M}^* \alpha p \beta, p \in F, \alpha, \beta \in \Gamma^* \} \).

**Example 10.1**

A TM \( M = (\{q_0, q_1, q_2, q_3, q_4\}, \{0, 1\}, \{0, 1, X, Y, B\}, \delta, q_0, B, \{q_4\}) \) with \( \delta \) in the table accepts the language \( \{0^n1^n; n \geq 1\} \).

<table>
<thead>
<tr>
<th>State</th>
<th>0</th>
<th>1</th>
<th>X</th>
<th>Y</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q_0 )</td>
<td>((q_1, X, R))</td>
<td>–</td>
<td>–</td>
<td>((q_3, Y, R))</td>
<td>–</td>
</tr>
<tr>
<td>( q_1 )</td>
<td>((q_1, 0, R))</td>
<td>((q_2, Y, L))</td>
<td>–</td>
<td>((q_1, Y, R))</td>
<td>–</td>
</tr>
<tr>
<td>( q_2 )</td>
<td>((q_2, 0, L))</td>
<td>–</td>
<td>((q_0, X, R))</td>
<td>((q_2, Y, L))</td>
<td>–</td>
</tr>
<tr>
<td>( q_3 )</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>((q_3, Y, R))</td>
<td>((q_4, B, R))</td>
</tr>
<tr>
<td>( q_4 )</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>
Definition 10.5 (Recursively enumerable languages (RE))
The set of languages that we can accept by a Turing machine.

Definition 10.6
Turing Machine halts A TM halts if it enters a state $q$, scanning a tape symbol $X$, and there is no move in this situation, i.e., $\delta(q, X)$ is undefined.

- We assume that a TM always halt when it is in an accepting state.
- We can require that a TM halts even if it does not accept only for recursive languages, a proper subset of recursively enumerable languages.
Definition 10.7 (Transition diagram)

A transition diagram consists of a set of nodes corresponding to the states of the TM. Any arc $q \rightarrow p$ is labeled the list of items $X/YD$ for all $\delta(q, X) = (p, Y, D)$, $D \in \{\leftarrow, \rightarrow\}$. We assume that the blank symbol is B unless we state otherwise.

<table>
<thead>
<tr>
<th>State</th>
<th>0</th>
<th>1</th>
<th>X</th>
<th>Y</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>$(q_1, X, R)$ – –</td>
<td></td>
<td>$(q_3, Y, R)$ –</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$q_1$</td>
<td>$(q_1, 0, R)$ $(q_2, Y, L)$ –</td>
<td>–</td>
<td>$(q_1, Y, R)$ –</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$q_2$</td>
<td>$(q_2, 0, L)$ – $(q_0, X, R)$</td>
<td>$(q_2, Y, L)$ –</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$q_3$</td>
<td>– – –</td>
<td>$(q_3, Y, R)$ $(q_4, B, R)$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$q_4$</td>
<td>– – –</td>
<td>– –</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A TM for \( \{0^n1^n; n \geq 1\} \)

Word 0011

\[
q_00011 \vdash Xq_1011 \vdash X0q_111 \vdash Xq_20Y1 \vdash q_2X0Y1 \vdash Xq_00Y1 \vdash XXq_1Y1 \vdash XXYq_11 \vdash XXq_2YY \vdash Xq_2XYY \vdash XXq_0YY \vdash XXYq_3Y \vdash XXYYq_3B \vdash XXYYBq_4B
\]

Word 0010

\[
q_00010 \vdash Xq_1010 \vdash X0q_110 \vdash Xq_20Y0 \vdash q_2X0Y0 \vdash Xq_00Y0 \vdash XXq_1Y0 \vdash XXYq_10 \vdash XXY0q_1B \text{ ends up with a failure since there is no instruction for } q_1,0.
\]
A TM that computes **monus, proper subtraction** \( m \div n = \max(m - n, 0) \).

- \( M = (\{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}, \{0, 1\}, \{0, 1, B\}, \delta, q_0, B) \), accepting set omitted (TM used for output, not acceptance).

- Start tape \( 0^m10^n \).

- \( M \) halts with the tape \( 0^{m-n} \) surrounded by blanks.

- Find leftmost 0, replace it by a blank.

- Search right, looking for a 1; continue, find 0 and replace it by 1.

- Return left.

- End if no 0 found, either left or right;
  - right: replace all 1 by B.
  - left: \( m < n \): replace all 1 and 0 by B, leave the tape blank.
Storage in the FA unit

- Storage in the State
- Consider state as a tuple
- \( M = (\{q_0, q_1\} \times \{0, 1, B\}, \{0, 1\}, \{0, 1, B\}, \delta, [q_0, B], B, \{[q_1, B]\}) \)
- \( L(M) = (01^* + 10^*) \),

<table>
<thead>
<tr>
<th>( \delta )</th>
<th>0</th>
<th>1</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>( [q_0, B] \rightarrow [q_0, B] )</td>
<td>([q_1, 0], 0, R)</td>
<td>([q_1, 1], 1, R)</td>
<td>([q_1, B], B, R)</td>
</tr>
<tr>
<td>( [q_1, 0] )</td>
<td>([q_1, 0], 0, R)</td>
<td>([q_1, 0], 1, R)</td>
<td>([q_1, B], B, R)</td>
</tr>
<tr>
<td>( [q_1, 1] )</td>
<td>([q_1, 1], 0, R)</td>
<td>([q_1, 0], 1, R)</td>
<td>([q_1, B], B, R)</td>
</tr>
<tr>
<td>( *[q_1, B] )</td>
<td>([q_1, 1], 0, R)</td>
<td>([q_1, 0], 1, R)</td>
<td>([q_1, B], B, R)</td>
</tr>
</tbody>
</table>
Multiple Tracks

\[ L_{wcw} = \{ wcw \mid w \in (0 + 1)^+ \} , \]

\[ M = (\{ q_0, \ldots, q_9 \} \times \{ 0, 1, B \}, \{ [B, 0], [B, 1], [B, c] \}, \{ B, * \} \times \{ 0, 1, B, c \}, \delta, [q_1, B], [B, B], \{ [q_9, B] \}) \]

\( \delta \) is defined as \((a, b \in \{ 0, 1 \}):\)

\[ \delta([q_1, B], [B, a]) = ([q_2, a], [*, a], R) \text{ picks up the symbol } a \]
\[ \delta([q_2, a], [B, b]) = ([q_2, a], [B, b], R) \text{ move right, look for } c, \]
\[ \delta([q_2, a], [B, c]) = ([q_3, a], [B, c], R) \text{ continue right, the state changed,} \]
\[ \delta([q_3, a], [*, b]) = ([q_3, a], [*, b], R) \text{ continue right,} \]
\[ \delta([q_3, a], [B, a]) = ([q_4, B], [*, a], L) \text{ check correct, drop memory and go left,} \]
\[ \delta([q_4, B], [*, a]) = ([q_4, B], [*, a], L) \text{ go left,} \]
\[ \delta([q_4, B], [B, c]) = ([q_5, B], [B, c], L) \text{ } c \text{ found, continue left,} \]
\[ \text{decide whether all inputs left and right are checked, branch adequately} \]
\[ \delta([q_5, B], [B, a]) = ([q_6, B], [B, a], L) \text{ left symbol unchecked,} \]
\[ \delta([q_6, B], [B, a]) = ([q_6, B], [B, a], L) \text{ proceed left,} \]
\[ \delta([q_6, B], [*, a]) = ([q_1, B], [*, a], R) \text{ start again,} \]
\[ \delta([q_5, B], [*, a]) = ([q_7, B], [*, a], R) \text{ symbol left from } c \text{ checked, go right,} \]
\[ \delta([q_7, B], [B, c]) = ([q_8, B], [B, c], R) \text{ proceed right,} \]
\[ \delta([q_8, B], [*, a]) = ([q_8, B], [*, a], R) \text{ proceed right,} \]
\[ \delta([q_8, B], [B, B]) = ([q_8, B], [B, B], R) \text{ accept.} \]
Theorem 10.1 (Recursively enumerable languages are of the type $L_0$)

Any recursively enumerable language is of the type $0$.

Proof: From a Turing Machine to the Grammar

for any given Turing Machine $T$ we construct a grammar $G$, $L(T) = L(G)$

- first, we generate the tape and a copy of the input word
- then, we simulate TM $T$ (states are inside the word)
- at the end, we erase the tape leaving only the input word $w$

$wB^n w^R q_0 B^m$ where $B^i$ is an empty space for TM work

1) $S \rightarrow DQ_0 E$
   $D \rightarrow xDX | E$ generates the word and a reverse copy
   $E \rightarrow BE | B$ generates empty space

2) $XP \rightarrow QX'$
   $XPY \rightarrow X'YQ$ for $\delta(p, x) = (q, x', R)$

3) $P \rightarrow C$
   $CA \rightarrow C, AC \rightarrow C$ for $\delta(p, x) = (q, x', L)$
   $C \rightarrow \lambda$ cleaning $A \in \Sigma \cup \{B\}$
   end of derivation
Ještě $L(T) = L(G)$?

- $w \in L(T)$
  - there exists a finite accepting sequence of steps $T$ (finite space)
  - the grammar generates sufficient space
  - it simulates the moves and cleans non-input symbols

- $w \in L(G)$
  - the rules need not be in our order
  - we may sort the derivation steps to ordering 1), 2), 3).
  - underlined symbols were erased, therefore accepting state must be generated.

### Grammar after simplification

<table>
<thead>
<tr>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \rightarrow Dq_0$</td>
</tr>
<tr>
<td>$D \rightarrow aDa</td>
</tr>
<tr>
<td>$Bq_0 \rightarrow C$</td>
</tr>
<tr>
<td>$aq_0 \rightarrow q_1a$</td>
</tr>
<tr>
<td>$aq_1 \rightarrow q_0a$</td>
</tr>
<tr>
<td>$Ca \rightarrow C$</td>
</tr>
<tr>
<td>$C \rightarrow \lambda$</td>
</tr>
</tbody>
</table>

Example 10.2

$q_0, B \rightarrow q_F, B, R$
$q_0, a \rightarrow q_1, a, R$
$q_1, a \rightarrow q_0, a, R$
Theorem 10.2

Any Type 0 language is recursively enumerable.

Proof:

the idea: TM generates sequentially all possible derivations

- we code $S \Rightarrow \omega_1 \Rightarrow \ldots \Rightarrow \omega_n = w$ as a word $\#S\#\omega_1\#\ldots\#w\#$
- we can construct a TM accepting words $\#\alpha\#\beta\#$ where $\alpha \Rightarrow \beta$
- we can construct a TM accepting words $\#\omega_1\#\ldots\#\omega_k\#$, kde $\omega_1 \Rightarrow^* \omega_k$
- we can construct a TM generating sequentially all possible words.
Definition 10.8 (Multitape Turing Machine)

**Initial position**
- the input word on the first tape, other tapes blank
- the first head left from the input, others anywhere
- FA control in the initial state

**One step of a multitape TM**
- FA control moves to the new state
- on each tape we write the appropriate new symbol
- each head moves independently left or right.

Theorem 10.3 (Multitape TM)

Any language accepted by a multitape TM is accepted also by some (one tape) Turing machine TM.
Proof: Multitape TM TM

- we construct a TM $M$
- the tape will have $2k$ tracks
  - odd tracks: $i^{th}$ head position
  - even tracks: symbols on the $i^{th}$ tape
- to simulate one step we visit all heads
- we remember in the FA control state
  - the number of heads left
  - $\forall i$ symbol under $i^{th}$ head
- then we know enough to simulate one step (visit all heads again)

The simulation of a $k$–tape machine with $n$ steps can be performed in $O(n^2)$ (one step simulation takes $4n + 2k$, heads are no further than $2n$ – read, write, move the head position marks).
Definition 10.9 (Nondeterministic TM)

**Nondeterministic Turing machine** is a 7–tuple \( M = (Q, \Sigma, \Gamma, \delta, q_0, B, F) \) with \( Q, \Sigma, \Gamma, q_0, B, F \) as TM and \( \delta : (Q - F) \times \Gamma \rightarrow P(Q \times \Gamma \times \{L, R\}) \).

A word \( w \in \Sigma^* \) is accepted by the nondeterministic TM \( M \) iff there exists an accepting sequence of moves \( q_0 w \vdash^* \alpha p \beta, p \in F \).

Theorem 10.4 (Nondeterministic TM)

To any nondeterministic Turing machine \( M_N \) there exist a deterministic TM \( M_D \) such that \( L(M_N) = L(M_D) \).

The main idea

- breath search all calculations of \( M_N \)
  - derived in \( k \) steps
  - no more that \( m^k \) IDs (instantaneous descriptions)
  - where \( m = \max |\delta(q, x)| \) the maximum number of choices in \( M_N \)
Proof: proof idea

- the tape is infinite – subset construction is not possible
- breath search of all possible calculations of $M_N$
- we simulate by a two–tape TM
  - first tape: a sequence of IDs
    - current position marked (by a cross)
    - left already evaluated we may erase
    - right the current ID and a waiting list
  - second tape: scratch tape
- to evaluate one ID means
  - read the state and the head symbol of current ID
  - if in the accepting state $\in F$, accept and halt
  - write ID on the scratch tape
  - for each step $\delta$ (stored in the head $M_D$)
    - perform the step and write the new ID to the end of the first tape
  - visit the marked ID, erase the mark and move it 1 ID to the right
  - repeat
Summary

- **The Turing Machine**
- **Acceptance by a Turing Machine**: Start: finite length string of tape symbols, the rest is blank. The TM accepts its input if it ever enters an accepting state.
- **Recursively Enumerable Languages (RE)**: the languages accepted by a TM.
- **Instantaneous Descriptions of a TM**: All tape symbols between the leftmost and the rightmost nonblank. The state and the position of the head are shown by placing the state within the sequence of tape symbols, just to the left of the cell scanned.
- **Storage in the Finite Control**: The state has two or more components, one for the control, others the 'storage'.
- **Multiple Tracks**: The tape symbols are vectors with a fixed number of components.
- **Multitape TM**: An extended TM model has some fixed number of tapes greater than one. Although able to recognize certain languages faster than the conventional TM, it cannot recognize any language that is not RE.
- **Nondeterministic Turing Machines**: The NTM has a finite number of choices of next move. It accepts an input if any sequence of choices leads to an ID with an accepting state. It does not recognize any language that is not RE.
Multiplication: Input: $0^m10^n1$, Output: $0^{mn}$.

- **Strategy:** On the tape generally $0^i10^n10^{kn}$
- In one basic step, change a 0 in the first group to B and add n 0’s to the last group, giving us the string of the form $0^{i−1}10^n10^{(k+1)n}$.
- When finished, change the leading $10^n1$ to blanks.

Multiply

Start \[ q_0 \]
\[ 0/B \rightarrow \]
\[ 0/0 \rightarrow \]
\[ 1/1 \rightarrow \]
\[ q_0 \]
\[ q_6 \]
\[ Copy \]
\[ q_7 \]
\[ q_8 \]
\[ B/B \rightarrow \]
\[ 0/0 \leftarrow \]
\[ 1/1 \leftarrow \]
\[ q_9 \]
\[ q_{10} \]

Copy

Start \[ q_1 \]
\[ 0/X \rightarrow \]
\[ X/X \rightarrow \]
\[ 0/0 \rightarrow \]
\[ 1/1 \leftarrow \]
\[ 1/1 \rightarrow \]
\[ q_2 \]
\[ q_3 \]
\[ 1/1 \rightarrow \]
\[ 0/0 \rightarrow \]
\[ B/0 \leftarrow \]
\[ X/X \rightarrow \]
\[ q_4 \]
\[ q_5 \]
\[ 1/1 \rightarrow \]
\[ X/0 \leftarrow \]
Theorem 10.5 (Semi–infinite Tape, Never Writes a Blank)

Every language accepted by a TM $M_2$ is also accepted by a TM $M_1$ with the following restrictions:

- $M_1$’s head never moves left of its initial position.
- $M_1$ never writes a blank.
Grammars Type 1 (context languages $L_1$)

- all productions in the form $\alpha A \beta \rightarrow \alpha \omega \beta$

  $A \in V, \alpha, \beta \in (V \cup T)^*, \omega \in (V \cup T)^+$

- with one exception $S \rightarrow \lambda$, in this case $S$ does not appear in the body of any rule
Context Grammars

- productions in the form \( \alpha A \beta \rightarrow \alpha \omega \beta \)
  
  \( A \in V, \alpha, \beta \in (V \cup T)^*, \omega \in (V \cup T)^+ \)

- with one exception \( S \rightarrow \lambda \),
  
  in this case \( S \) does not appear in the body (right side) of any rule

- variable \( A \) rewrites only in the context \( \alpha, \beta \)

- \( S \rightarrow \lambda \) is used only to add \( \lambda \) to the language

Example 11.1 (a context language)

\( L = \{a^n b^n c^n | n \geq 1\} \) is a context language that is not context–free.

Grammar:

\[
\begin{align*}
S & \rightarrow aSBC | abC \\
CB & \rightarrow BC \\
bB & \rightarrow bb \\
bC & \rightarrow bc \\
cC & \rightarrow cc \\
\end{align*}
\]

(\(*\) is not a context rule, will be rewritten)

\[
\begin{align*}
\text{Grammar:} & \quad \text{\( (*) \ CB \rightarrow BC \) we replace this rule by} \\
& \quad \text{\( CB \rightarrow XB, \ XB \rightarrow XY, \ XY \rightarrow BY, \ BY \rightarrow BC \) }
\end{align*}
\]
Definition 11.1 (Separated grammar)

A grammar is separated, iff all productions are in the form $\alpha \rightarrow \beta$, where:

- either $\alpha, \beta \in V^+$ (non–empty sequences of variables)
- or $\alpha \in V$ and $\beta \in T \cup \{\lambda\}$.

Lemma

For any grammar $G$ there exists equivalent separated grammar $G'$.

Proof:

- Let $G = (V, T, P, S)$
- for each terminal $a \in V$ we introduce new variable $A'$.
- in productions in $P$
  - we replace terminals by corresponding variables
  - we add rules $A' \rightarrow a$
- Resulting grammar is separated and obviously $L(G) = L(G')$.  

Definition 11.2 (monotone grammar)

A grammar is **monotone**, iff for each rule \((\alpha \rightarrow \beta) \in P\) holds \(|\alpha| \leq |\beta|\). Monotone grammar never shortens sentential form during the derivation.

Lemma

*To any monotone grammar there exists an equivalent context grammar.*

Proof:

- first, find an equivalent separated grammar
  - this does not violate the monotony (and rules \(A' \rightarrow a\) are context rules)
- remaining productions \(A_1 \ldots A_m \rightarrow B_1 \ldots B_n, m \leq n\) replace by productions with new variables \(C\)
  
  \[
  \begin{align*}
  A_1 A_2 \ldots A_m & \rightarrow C_1 A_2 \ldots A_m \\
  C_1 A_2 \ldots A_m & \rightarrow C_1 C_2 \ldots A_m \\
  \vdots & \quad \vdots \\
  C_1 \ldots C_{m-1} A_m & \rightarrow C_1 \ldots C_{m-1} C_m \\
  B_1 C_2 \ldots C_m & \rightarrow B_1 C_2 \ldots C_m \\
  B_1 \ldots B_{m-1} C_m & \rightarrow B_1 \ldots B_{m-1} B_m \ldots B_n
  \end{align*}
  \]
Example 11.2

The language $L = \{a^i b^j c^k | 1 \leq i \leq j \leq k\}$ is a context language, that is not a context–free language.

Proof:

- $S \rightarrow aSBC|aBC$ generate symbols $a$
- $B \rightarrow BBC$ adds symbols $B$
- $C \rightarrow CC$ adds symbols $C$
- $CB \rightarrow BC$ ordering of $B$'s and $C$'s (*)
- $aB \rightarrow ab$ start rewriting of $B$ to $b$
- $bB \rightarrow bb$ continue rewriting $B$ to $b$
- $bC \rightarrow bc$ start rewriting $C$ to $c$
- $cC \rightarrow cc$ continue rewriting $C$ to $c$

(*) $CB \rightarrow BC$ is not a context rule, we replace it with

- $CB \rightarrow XB, XB \rightarrow XY, XY \rightarrow BY, BY \rightarrow BC$
Linear Bounded Automata

**Definition 11.3 (Linear Bounded automaton (LBA))**

**Linear Bounded Automaton** LBA is a nondeterministic TM, with left and right endmark on the tape \( l, r \). These symbols are not possible to rewrite and cannot be stepped 'out' during evaluation (left from \( l \) and right from \( r \)).

A string \( w \) is accepted by a LBA, \( \equiv \) iff \( q_0 l w r \vdash^* \alpha p \beta \), \( p \in F \).

- The space is defined by the input string and the LBA cannot use more space
- monotone (context) derivations do not need more space – no sentential form in the derivation is longer then the resulting word
Theorem 11.1

Each context language can be accepted by a LBA.

Proof: from a context grammar to LBA

- derivation in the grammar will be simulated by a LBA
- we use two track tape
- upper track: the string $w$, lower track: $S$ (start symbol of the grammar followed by blanks)
- we rewrite the sentential form on the second tape by productions in $G$
  - we nondeterministically choose the part to rewrite
  - we rewrite appropriate part (the right side may be moved to the right)
- if only terminals on the second track, we compare it with the first track
  - accept or reject according the result of the comparison
Theorem 11.2

LBA’s accept only context languages.

Proof: From LBA to context grammars

- we reduce LBA to monotone grammar
  - the grammar cannot use more space than the resulting string of terminals.
- we hide the steps into two–tracks variables

1) generate string in the form \((a_0, [q_0, l, a_0]), (a_1, a_1), \ldots, (a_n, [a_n, r])\)

\[
\begin{array}{c|c|c}
\text{w} & q_0, l, a_0 & a_n, r \\
\end{array}
\]

2) simulate LBA steps on the second track (similar to TM simulation)

- \(\text{pro } \delta(p, x) = (q, x', R): PX \rightarrow X'Q\)
- \(\text{pro } \delta(p, x) = (q, x', L): YPX \rightarrow QYX'\)

- if in the accepting state, erase the second track
- acceptance of the empty string:
  - If LBA accepts \(\lambda\), we add the special rule.
We aim to prove undecidable the language consisting of pairs \((M, w)\) such that:
- \(M\) is a TM (binary coded) with input alphabet \(\{0, 1\}\),
- \(w\) is a string of 0’s and 1’s, and
- \(M\) accepts input \(w\).

Our plan:
- Encode TM’s by binary code regardless of how many states the TM has.
- Treat TM as a binary string.
- If a string is not well formed, think of it as a TM with no moves. Therefore, every binary string represents some TM.
- **Diagonalization language** \(L_d\);
  \[L_d = \{w; \text{TM represented as } w \text{ that does not accept } w\}\].
- There does not exist a TM recognizing the language \(L_d\). Running it on its own code leads to the paradox.
We call $w_i$ the $i$–th string, where $\epsilon$ is the first string, 0 the second, 1 the third, 00 the fourth and so on.

Strings are ordered by length, equal length are ordered lexicographically.

- To represent a TM $M = (Q, \{0, 1\}, \Gamma, \delta, q_1, B, F)$ as a binary string, we must first assign integers to the states, tape symbols, and directions $L, R$.

- Assume:
  - Start state is always $q_1$.
  - Always is $q_2$ the only accepting state (we do not need more, TM halts).
  - First symbol is always 0, the second 1, the third B, the blank. Other tape symbols can be assigned arbitrarily.
  - Direction L is 1, direction R is 2.

- One transaction $\delta(q_i, X_j) = (q_k, X_l, D_m)$ is coded: $0^i10^j10^k10^l10^m$. Notice all $i, j, k, l, m \geq 1$ so no substring 11 occurs here.

- The entire TM consists of all the codes for transaction in some order, separated by pair of 1's: $C_111C_211\ldots C_{n-1}11C_n$. 
## TM encoding example

### Turing Machine

\[ M = (\{q_1, q_2, q_3\}, \{0, 1\}, \{0, 1, B\}, \delta, q_1, B, \{q_2\}) \]

<table>
<thead>
<tr>
<th>( \delta )</th>
<th>0</th>
<th>1</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \rightarrow q_1 )</td>
<td>( q_1 )</td>
<td>( (q_3, 0, R) )</td>
<td>( (q_3, 1, L) )</td>
</tr>
<tr>
<td>( *q_2 )</td>
<td>( (q_1, 1, R) )</td>
<td>( (q_2, 0, R) )</td>
<td>( (q_3, 1, L) )</td>
</tr>
</tbody>
</table>

- The code for transitions
  - \( C_1 \): 0100100010100
  - \( C_2 \): 0001010100100
  - \( C_3 \): 0001001001010
  - \( C_4 \): 00010001000100

- The overall TM code:
  01001000101001100010100100110001001010100110001001001001100010010010010010010010010.
Definition 12.1 (The Diagonal Language)

The Diagonal Language \( L_d \) is defined as
\[
L_d = \{ w ; \text{TM represented as } w \text{ that does not accept } w \}.
\]

Proof.
- Assume \( L_d \) is RE, \( L_d = L(M) \) for some TM \( M \).
- It has language \( \{0, 1\} \), so it is in the list in the figure: 'Does TM \( M_i \) accept input string \( w_j \)?'
- There is at least one code for it, say \( i, M = M_i \).
- Is \( w_i \in L_d \)
  - 'Yes' imply \( \text{diagonal}(w_i) = 0 \), therefore \( w_i \notin L(M_i) \). Contradiction \( L(M_i) = L_d \).
  - 'No' imply \( \text{diagonal}(w_i) = 1 \), therefore \( w_i \in L(M_i) \). Contradiction \( L(M_i) = L_d \).

Therefore, such \( M \) does not exist. \( L_d \) is not RE.
Definition 12.2 (TM halts)

TM **halts** iff it enters a state \( q \), reading \( X \), and there is no instruction for this situation, that is \( \delta(q, X) \) is undefined.

- We assume TM halts in any accepting state \( q \in F \),
- We are not sure whether it accepts until TM halts.

Definition 12.3 (Recursive languages, Decidable problems)

- We say that a TM \( M \) **decides a language** \( L \) iff \( L = L(M) \) and for any \( w \in \Sigma^* \) the TM with the input \( w \) halts.
- For a computational problem with yes/no answer, we say it is a **decidable problem** iff there exists a computer program that always halts and gives the correct answer.
- **Recursive languages** are such languages, for those there exists a TM \( M \) that decides the language.
Complements of Recursive and RE languages

Theorem 12.2

If \( L \) is a recursive language, so is \( \overline{L} \).

Proof.

- \( L = L(M) \) for some TM \( M \) that always halts.
- We construct TM \( \overline{M} \) such that \( \overline{L} = L(\overline{M}) \).
- Accepting states of \( M \) are non-accepting in \( \overline{M} \) without any transaction out of them.
- \( \overline{M} \) has a new accepting state \( r \); no transition from \( r \).
- For each non-accepting state of \( M \) and each tape symbol such that \( M \) has no transition, add a transition to the accepting state \( r \).
- Since \( M \) is guaranteed to halt, \( \overline{M} \) is also guaranteed to halt.
- \( \overline{M} \) accepts \( \overline{L} \).
Theorem 12.3 (Post Theorem)

A language $L$ is recursive iff both $L$ and $\overline{L}$ (the complement) are recursively enumerable.

Proof:

- We have TM $L = L(M_1)$ and $\overline{L} = L(M_2)$.
- For the word $w$ we simulate both $M_1$ and $M_2$ (two tapes, states with two components).
- If any $M_i$ accepts, $M$ halts and answers.
- Languages are complementary, one of $M_i$'s always halts, therefore $L$ is recursive.

Theorem 12.4

If $L$ is recursive, so is also $\overline{L}$. 
The basic types of languages:

- **Recursive, decidable**: Accepted by a TM that always halts, regardless of whether or not it accepts.

- **RE Recursively enumerable**: accepted by some TM. The computation may take long, we never know whether it rejects or we shall wait longer.

- **Some languages are not even recursively enumerable**, *non*-RE languages, like $L_d$ is *non*-RE language.
**The Universal Language**

**Definition 12.4 (The Universal Language)**

We define $L_u$, the **universal language**, the set of binary strings that encode a pair $(M, w)$, such that $M$ is a TM and $w \in L(M)$, that is $L_u = \{(M, w) : \text{TM } M \text{ accepts } w\}$.

**Theorem 12.5 (The Universal Turing Machine)**

There is a TM $U$, called the **universal Turing machine**, such that $L_u = L(U)$.

We describe $U$ as a multitape Turing machine.

- Transactions of $M$ are stored initially on the first tape, along with the string $w$, separated by 111.
- Second tape holds the simulated tape of $M$, using format as code of $M$, i.e. symbols $0^i$ separated by 1’s.
- Third tape holds the state of $M$ represented by $i$ 0’s.
Operations of the Universal Turing Machine

The operation of $U$ can be summarized as follows:

- Examine the input whether the code for $M$ is legitimate; if not, $U$ halts without accepting.
- Initialize the second tape with $w$ in its encoded form: 10 for 0 in $w$, 100 for 1; blanks are left blank and replaced with 1000 only 'on demand'.
- Place 0, the start state of $M$, on the third tape. Move the head on the second tape to the first simulated cell.
- To simulate a move of $M$
  - Search on the first tape for a proper transition $0^i10^j10^k10^l10^m$, $0^i$ on tape 3, $0^j$ on tape 2.
  - Change the content of tape 3 to $0^k$.
  - Replace $0^j$ on tape 2 by $0^l$. Use scratch tape to manage the spacing.
  - Move the head on tape 2 to the position of the next 1 to the left or right, depending on $m$.
- If $M$ has no transition that matches the simulated state and tape symbol, halt.
- If $M$ enters its accepting state, then $U$ accepts.
**Theorem 12.6 (Undecidability of the Universal Language)**

$L_u$ is RE but not recursive.

**Proof.**

- We have proved that $L_u$ is RE.
- Suppose $L_u$ were recursive.
- Then, $\overline{L_u}$ would also be recursive.
- If we have TM to accept $\overline{L_u}$, then we can construct a TM to accept $L_d$ (see right).
- Since we already know $L_d$ is not RE, $\overline{L_u}$ is not RE and $L_u$ is not recursive.

**Modification of TM for $\overline{L_u}$ to TM for $L_d$:**

- Given string $w$, change it to $w111w$ (2-tapes, convert to 1-tape).
- Simulate $M$ on the new input. Accept iff $M$ accepts.
- Choose $i$ s.t. $w_i = w$. Previous line accepts $\overline{L_u}$, that is cases where $M_i$ does not accept $w_i$, that is the language $L_d$. 

![Diagram of TM modification](image)
Definition 12.5 (Reduction)

**Reduction** $R$ is a mapping of all instances $P_1$ into $P_2$ that halts for every instance $w \in P_1$ with the output $R(w) \in P_2$ with the property so that

- $P_1(w) = \text{yes}$ iff $P_2(R(w)) = \text{yes}$
- therefore $P_1(w) = \text{no}$ iff $P_2(R(w)) = \text{no}$.

**Example 12.1**

We reduce $L_d$ to $\overline{L_u}$:

- $P_1 = \text{Does TM } w \text{ not accept } w$?
- $P_2 = \text{Does TM } M \text{ not accept } w$?
Theorem 12.7 (Reductions)

If there is a reduction from \( P_1 \) to \( P_2 \), then:

- If \( P_1 \) is undecidable then so is \( P_2 \).
- If \( P_1 \) is non–RE then so is \( P_2 \).

Proof.

Assume \( P_1 \) is undecidable. If it is possible to decide \( P_2 \), then we can combine the reduction from \( P_1 \) to \( P_2 \) with the algorithm that decides \( P_2 \) to construct an algorithm that decides \( P_1 \). Therefore, \( P_2 \) is undecidable.

Assume \( P_1 \) non–RE, but \( P_2 \) is RE. Similarly as above, we combine the reduction and \( P_2 \) result to show \( P_1 \) is RE; we assumed \( P_1 \) is non–RE. Contradiction.
Definition 12.6 (Post’s Correspondence Problem)

An instance of **Post’s Correspondence Problem (PCP)** consists of two lists of strings over some alphabet \( \Sigma \) denoted as \( A = w_1, w_2, \ldots, w_k \) and \( B = x_1, x_2, \ldots, x_k \); the two lists must be of equal length. For each \( i \), the pair \((w_i, x_i)\) is said to be a **corresponding** pair. We say this instance of PCP has a **solution**, if there is a sequence of one or more integers \( i_1, i_2, \ldots, i_m \) such that \( w_{i_1}, w_{i_2}, \ldots, w_{i_m} = x_{i_1}, x_{i_2}, \ldots, x_{i_m} \), that is, yield the same string. We say the sequence \( i_1, i_2, \ldots, i_m \) is a **solution**, if so.

The Post's correspondence problem is: Given an instance of PCP, tell whether this instance has a solution.

### Example 12.2

<table>
<thead>
<tr>
<th>( i )</th>
<th>List ( A ) ( w_i )</th>
<th>List ( B ) ( x_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>111</td>
</tr>
<tr>
<td>2</td>
<td>10111</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>0</td>
</tr>
</tbody>
</table>

- \( \Sigma = \{0, 1\} \), lists A,B see the table.
- The solution is 2, 1, 1, 3 creating the string 101111110.
- Another solution: 2,1,1,3,2,1,1,3.
Partial Solutions

Example 12.3

\[ \Sigma = \{0, 1\}. \] There does not exist solution for the lists:

<table>
<thead>
<tr>
<th></th>
<th>List A</th>
<th>List B</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>(w_i)</td>
<td>(x_i)</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>101</td>
</tr>
<tr>
<td>2</td>
<td>011</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>101</td>
<td>011.</td>
</tr>
</tbody>
</table>

Reasoning:

- \(i_1 = 1\), the first symbol does not match otherwise.
- We have a partial solution:
  
  A: 10\cdots
  
  B: 101\cdots

Definition 12.7 (Partial Solution)

Sequence of indexes \(i_1, i_2, \ldots, i_r\) such that one of the strings \(w_{i_1}, w_{i_2}, \ldots, w_{i_r}\) and \(x_{i_1}, x_{i_2}, \ldots, x_{i_r}\) is a prefix of the other, although the two strings are not equal.

Lemma

If a sequence of integers is a solution then every prefix of that sequence must be a partial solution.

- \(i_2 = 1\), the strings
  
  1010
  
  101101
  
  disagree on the fourth position.
- \(i_2 = 2\),
  
  10011
  
  10111
  
  disagree on the third position.
- Only \(i_2 = 3\) is possible.

A: 10101\cdots

B: 101011\cdots

- We are in the same position as after choosing \(i_1 = 1\).
- There is no way to get both strings of equal length.
Definition 12.8 (Modified Post’s Correspondence Problem)

An instance of Modified Post’s Correspondence Problem (MPCP) is two lists consists of two lists $A = w_1, w_2, \ldots, w_k$ and $B = x_1, x_2, \ldots, x_k$, and a solution is a list of 0 or more integers $i_1, i_2, \ldots, i_m$ such that $w_{i_1}, w_{i_2}, \ldots, w_{i_m} = x_1, x_i, x_2, \ldots, x_i$.

Example 12.4

This problem seen as MPCP does not have a solution.

<table>
<thead>
<tr>
<th>$i$</th>
<th>$w_i$</th>
<th>$x_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>111</td>
</tr>
<tr>
<td>2</td>
<td>10111</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>0</td>
</tr>
</tbody>
</table>

Proof:

- Partial instances $111, 111111$ never get the same length.
- Other choices lead to mismatch.
MPCP reduces to PCP.

**Theorem 12.8 (MPCP reduces to PCP.)**

**MPCP reduces to PCP.**

The same lists as before as a MPCP.

<table>
<thead>
<tr>
<th>List A</th>
<th>List B</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i)</td>
<td>(w_i)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>10111</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
</tr>
</tbody>
</table>

**Example 12.5 (MPCP reduces to PCP.)**

<table>
<thead>
<tr>
<th></th>
<th>List C</th>
<th>List D</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i)</td>
<td>(y_i)</td>
<td>(z_i)</td>
</tr>
<tr>
<td>0</td>
<td><em>1</em></td>
<td><em>1</em>1*1</td>
</tr>
<tr>
<td>1</td>
<td>1*</td>
<td><em>1</em>1*1</td>
</tr>
<tr>
<td>2</td>
<td>1<em>0</em>1<em>1</em>1*</td>
<td>1*0</td>
</tr>
<tr>
<td>3</td>
<td>1<em>0</em></td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>$</td>
<td>*$</td>
</tr>
</tbody>
</table>

- Assume \(*, \$_{\notin \Sigma}, the alphabet of this MPCP instance.
- For \(i = 1, \ldots, k\) let \(y_i\) be \(w_i\) with a \(*\) after each symbol of \(w_i\).
- For \(i = 1, \ldots, k\) let \(z_i\) be \(x_i\) with a \(*\) before each symbol of \(x_i\).
- \(y_0 = *y_1, z_0 = z_1\).
- \(y_{k+1} = $, z_{k+1} = *$.\)
- If \(i_1, i_2, \ldots, i_m\) is a solution of MPCP, then \(0, i_1, i_2, \ldots, i_m, (k + 1)\) is a solution of PCP.
We aim to prove PCP is undecidable.
We know MPCP reduces to PCP.
We are going to reduce $L_u$ to MPCP.

$L_u$ to MPCP

MPCP construction for a TM $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$, that never moves left of its initial position and never writes blank. Let $w \in \Sigma^*$ be an input string.

<table>
<thead>
<tr>
<th>List A</th>
<th>List B</th>
</tr>
</thead>
<tbody>
<tr>
<td>$##q_0w#$</td>
<td>$##$</td>
</tr>
<tr>
<td>$X$</td>
<td>$X$</td>
</tr>
<tr>
<td>$#$</td>
<td>$#$</td>
</tr>
<tr>
<td>$qX$</td>
<td>$Yp$</td>
</tr>
<tr>
<td>$ZqX$</td>
<td>$pZY$</td>
</tr>
<tr>
<td>$q#$</td>
<td>$Yp#$</td>
</tr>
<tr>
<td>$Zq#$</td>
<td>$pZY#$</td>
</tr>
<tr>
<td>$XqY$</td>
<td>$q$</td>
</tr>
<tr>
<td>$Xq$</td>
<td>$q$</td>
</tr>
<tr>
<td>$qY$</td>
<td>$q$</td>
</tr>
<tr>
<td>$q##$</td>
<td>$#$</td>
</tr>
</tbody>
</table>
Example 12.6

Let us convert the TM

\[
M = (\{q_1, q_2, q_3\}, \{0, 1\}, \{0, 1, X, Y, B\}, \delta, q_0, B, \{q_3\})
\]

\[
\begin{array}{c|c|c|c}
q_i & \delta(q_i, 0) & \delta(q_i, 1) & \delta(q_i, B) \\
\hline
q_1 & (q_2, 1, R) & (q_2, 0, L) & (q_2, 1, L) \\
q_2 & (q_3, 0, L) & (q_1, 0, R) & (q_2, 0, R) \\
q_3 & - & - & - \\
\end{array}
\]

and input string \( w = 01 \) to an instance of MPCP.

The list of pairs without \( B \) symbol (broken into two tables)

<table>
<thead>
<tr>
<th>List A</th>
<th>List B</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>#q_101#</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>0q_30</td>
<td>q_3</td>
</tr>
<tr>
<td>0q_31</td>
<td>q_3</td>
</tr>
<tr>
<td>1q_30</td>
<td>q_3</td>
</tr>
<tr>
<td>1q_31</td>
<td>q_3</td>
</tr>
<tr>
<td>0q_3</td>
<td>q_3</td>
</tr>
<tr>
<td>1q_3</td>
<td>q_3</td>
</tr>
<tr>
<td>q_30</td>
<td>q_3</td>
</tr>
<tr>
<td>q_31</td>
<td>q_3</td>
</tr>
<tr>
<td>q_3##</td>
<td>#</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>from ( \delta(q_1, 0) = (q_2, 1, R) )</td>
</tr>
<tr>
<td>from ( \delta(q_1, 1) = (q_2, 0, L) )</td>
</tr>
<tr>
<td>from ( \delta(q_1, B) = (q_2, 1, L) )</td>
</tr>
<tr>
<td>from ( \delta(q_1, B) = (q_2, 1, L) )</td>
</tr>
<tr>
<td>from ( \delta(q_2, 0) = (q_3, 0, L) )</td>
</tr>
<tr>
<td>from ( \delta(q_2, 0) = (q_3, 0, L) )</td>
</tr>
<tr>
<td>from ( \delta(q_2, 1) = (q_1, 0, R) )</td>
</tr>
<tr>
<td>from ( \delta(q_2, B) = (q_2, 0, R) )</td>
</tr>
</tbody>
</table>
MPCP simulation of a TM

<table>
<thead>
<tr>
<th>List $A$</th>
<th>List $B$</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_10$</td>
<td>1$q_2$</td>
<td>from $\delta(q_1, 0) = (q_2, 1, R)$</td>
</tr>
<tr>
<td>0$q_11$</td>
<td>$q_20$</td>
<td>from $\delta(q_1, 1) = (q_2, 0, L)$</td>
</tr>
<tr>
<td>1$q_11$</td>
<td>$q_21$</td>
<td>from $\delta(q_1, 1) = (q_2, 0, L)$</td>
</tr>
<tr>
<td>0$q_1#$</td>
<td>$q_201$#</td>
<td>from $\delta(q_1, B) = (q_2, 1, L)$</td>
</tr>
<tr>
<td>1$q_1#$</td>
<td>$q_211#$</td>
<td>from $\delta(q_1, B) = (q_2, 1, L)$</td>
</tr>
<tr>
<td>0$q_20$</td>
<td>$q_300$</td>
<td>from $\delta(q_2, 0) = (q_3, 0, L)$</td>
</tr>
<tr>
<td>1$q_20$</td>
<td>$q_310$</td>
<td>from $\delta(q_2, 0) = (q_3, 0, L)$</td>
</tr>
<tr>
<td>$q_21$</td>
<td>0$q_1$</td>
<td>from $\delta(q_2, 1) = (q_1, 0, R)$</td>
</tr>
<tr>
<td>$q_2#$</td>
<td>0$q_2#$</td>
<td>from $\delta(q_2, B) = (q_2, 0, R)$</td>
</tr>
</tbody>
</table>

- $M$ never writes $B$, so we shall never have $B$ in an ID. Thus, we shall omit all the pairs that involve $B$.

- $M$ accepts by the sequence

$$q_101 \vdash 1q_21 \vdash 10q_1 \vdash 1q_201 \vdash q_31011.$$

$A: \ #q_101\#1q_21\#10q_1\#1q_201\#q_3101\#q_301\#q_31\#q_3\#\#

B: \ #q_101\#1q_21\#10q_1\#1q_201\#q_3101\#q_301\#q_31\#q_3\#\#.$
Theorem 12.9 (PCP is undecidable.)

*Post’s Correspondence Problem is undecidable.*

**Proof.**

We have an algorithm to reduce $L_u$ to MPCP. We sell prove that:

- $M$ accepts $w$ if and only if the constructed MPCP instance has a solution.

- **Only–if** If $w \in L(M)$, we start with the initial pair, and simulate the computation of $M$ on $w$.

- **If** If the MPCP instance has a solution, it is because $M$ accepts $w$.
  
  - MPCP must start with the first pair.
  - As long as $q \neq F$, erasing and end rules are useless.
  - Unless $q \in F$, partial solution has the form: $A:x$ $B:xy$, i.e. $B$ is longer than $A$. 

□
Theorem 12.10

It is undecidable whether a CFG is ambiguous.

Consider a PCP instance \((A = w_{i_1}, w_{i_2}, \ldots, w_{i_m}, B = x_{i_1}, x_{i_2}, \ldots, x_{i_m})\), a set of distinct indexes \(a_1, a_2, \ldots, a_m \in N\) and following three grammars,

\[ G_A \quad A \rightarrow w_1 A a_1 | w_2 A a_2 | \ldots | w_k A a_k | w_1 a_1 | w_2 a_2 | \ldots | w_k a_k \]
\[ G_B \quad B \rightarrow x_1 B a_1 | x_2 B a_2 | \ldots | x_k B a_k \]
\[ G_{AB} \quad \{S \rightarrow A | B\} \cup G_A \cup G_B. \]

We claim that \(G_{AB}\) is ambiguous if and only if the instance \((A, B)\) of PCP has a solution.

- Observe that any terminal string derived from \(A\) in \(G_A\) has a unique derivation (set by \(a_i\)'s right). Similarly for \(B\).
Theorem 12.11

Let $G_1, G_2$ be context–free grammars, and let $R$ be a regular expression. Then the following are undecidable:

- Is $L(G_1) \cap L(G_2) = \emptyset$?
- Is $L(G_1) = T^*$ for some alphabet $T$?
- Is $L(G_1) = L(G_2)$?
- Is $L(G_1) = L(R)$?
- Is $L(G_1) \subseteq L(G_2)$?
- Is $L(R) \subseteq L(G_1)$?
Proof: \( L(G_1) \cap L(G_2) = \emptyset \)

We reduce to PKP

- choose new variables \( \{a_1, a_2, \ldots, a_m\} \) as index codes
  
  \[
  G_1 \quad A \rightarrow w_1 A a_1 | w_2 A a_2 | \ldots | w_k A a_k \\
  w_1 a_1 | w_2 a_2 | \ldots | w_k a_k
  
  G_2 \quad B \rightarrow x_1 B a_1 | x_2 B a_2 | \ldots | x_k B a_k \\
  x_1 a_1 | x_2 a_2 | \ldots | x_k a_k
  
- PKP has a solution iff \( L(G_1) \cap L(G_2) \neq \emptyset \)
- the first part must be equal, the second \( (a_i) \) guarantees equal ordering.
Proof: \( 2 \, \mathcal{L}(G) = T^* \)

We reduce it to PKP

- we choose new variables \( \{a_1, a_2, \ldots, a_m\} \) as index codes
  
  \[
  \begin{align*}
  G_1 & \quad A \rightarrow \ w_1 Aa_1 | w_2 Aa_2 | \ldots | w_k Aa_k \\
  & \quad \, \, \, \quad w_1 a_1 | w_2 a_2 | \ldots | w_k a_k \\
  G_2 & \quad B \rightarrow \ x_1 Ba_1 | x_2 Ba_2 | \ldots | x_k Ba_k \\
  & \quad \, \, \, \quad x_1 a_1 | x_2 a_2 | \ldots | x_k a_k 
  \end{align*}
  \]

  - languages \( \mathcal{L}(G_1), \mathcal{L}(G_2) \) are deterministic,
  - so \( \overline{\mathcal{L}(G_1)}, \overline{\mathcal{L}(G_2)} \) are deterministic CFLs and \( \overline{\mathcal{L}(G_1)} \cup \overline{\mathcal{L}(G_2)} \) is a CFL
  - we have a context free grammar for \( \overline{\mathcal{L}(G_1)} \cup \overline{\mathcal{L}(G_2)} \)
  - PKP has a solution iff \( \Leftrightarrow \mathcal{L}(G_1) \cap \mathcal{L}(G_2) \neq \emptyset \Leftrightarrow \mathcal{L}(G) = \overline{\mathcal{L}(G_1)} \cup \overline{\mathcal{L}(G_2)} \neq \Sigma^* \). 

- Remark: \( \mathcal{L}(G) = \emptyset \) is decidable.
- CFL are not closed under complement, only deterministic CFLs are.
Proof: 3-6


\[ \text{Je } L(G_1) = L(G_2) ? \quad \text{Proof: } L \text{t } G_1 \text{ generates } \Sigma^* \]
\[ \text{Je } L(G_1) = L(R) ? \quad \text{Proof: } \text{We choose } R \text{ to be } \Sigma^* \]
\[ \text{Je } L(G_1) \subseteq L(G_2) ? \quad \text{Proof: } \text{Let } G_1 \text{ generates } \Sigma^* \]
\[ \text{Je } L(R) \subseteq L(G_1) ? \quad \text{Proof: } \text{We choose } R \text{ to be } \Sigma^* \]

Remark: \( L(G) \subseteq R \) is decidable

\[ L(G) \subseteq R \iff L(G) \cap \overline{R} = \emptyset \text{ and } (L(G) \cap \overline{R}) \text{ is a CFL (closure properties.)} \]
Automata and Grammars – Chomsky Hierarchy

- Finite automata: DFA, NFA, $\lambda$NFA
- Pushdown automata
- Linear bounded automata
- Turing Machines: multitape, nondeterministic
- Type 0 grammars
- Context grammars
- Monotone grammars
- Context free grammars
- Regular (right linear) grammars
\( L_d = \{ w; \; \text{TM coded by } w \text{ does not accept } w \} \)

\( L_0 \)
\[ \{ (M, w); \; \text{TM } M \text{ accepts } w \} \]

\( L_u = \{ w; \; \text{TM coded by } w \text{ does not accept } w \} \)

\( L_1 \)
\[ \{ a^i b^i c^i | i = 0, 1, \ldots \} \]

\( L_2 \)
\[ \{ w w^R | w \in \{0, 1\} \} \]

\( L_3 \)
\[ \text{regular lang.} \{0, 1^2\} \{010\} \]

\( \text{prefix prop & DPDA} \{0^n 1^n; \; n > 0 \} \)

\( \text{deterministic PDA} \{0^n 1^m; \; 0 < n \leq m \} \)

\( \text{context free (=CFL)} \{ w w^R | w \in \{0, 1\} \} \)

\( \text{context (=} CL) \{ a^i b^i c^i | i = 0, 1, \ldots \} \)
Definitions

- Terms from Figure 1, the language accepted by a FA, generated by a grammar and related definitions.
- Regular expressions
- CFG, CFL: a parse tree, ambiguity of a CFL grammar/language, Chomsky normal form
- Deterministic and non-deterministic PDA, $L(P)$, $N(P)$, prefix languages
- Recursive and recursively enumerable languages, Diagonal language $L_d = \{ w; \text{TM coded by } w \text{ does not } w \}$, Universal language (Universal Turing Machine), Post correspondence problem and undecidable problems for CFG’s.
Theorems

- Mihyil–Nerode theorem, Pumping lemma for regular languages, Pumping lemma for context–free languages, Kleene theorem (algebraic definition of regular languages)
- Inclusions and equivalences of classes Figure 1, both inside and outside boxes
- Closure properties – proof YES, counter example NO in the table below, closure properties of regular and CFL languages under string operations.
## Closure properties

<table>
<thead>
<tr>
<th>Language</th>
<th>Regular (RL)</th>
<th>Context Free</th>
<th>Deterministic CFL</th>
</tr>
</thead>
<tbody>
<tr>
<td>union</td>
<td>$F = F_1 \times Q_2 \cup Q_1 \times F_2$</td>
<td>$S \rightarrow S_1</td>
<td>S_2$</td>
</tr>
<tr>
<td>intersection</td>
<td>$F = F_1 \times F_2$</td>
<td>$L = {0^n1^n2^n</td>
<td>n \geq 1} = $</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$= {0^n1^n2^i</td>
<td>n, i \geq 1} \cap {0^i1^n2^n</td>
</tr>
<tr>
<td>$\cap$ with a RL</td>
<td>$F = F_1 \times F_2$</td>
<td>$F = F_1 \times F_2$</td>
<td></td>
</tr>
<tr>
<td>complement</td>
<td>$F = Q_1 - F_1$</td>
<td>$A \cap B = \overline{A} \cup \overline{B}$</td>
<td>$F = Q_1 - F_1, Z_0, \text{cycles}$</td>
</tr>
<tr>
<td>homom.</td>
<td>Kleene + elem. lang. + cl.</td>
<td>$a\ \text{replace } S_a$</td>
<td>$h(0) = h(1) = 0 \text{ app. }$</td>
</tr>
</tbody>
</table>

**Inv. hom.**

![Diagram](image-url)
- Reachable states of a FA, distinguishable and equivalent states FA, FA equivalence, Reduced FA, Subset construction of a FA from a NFA
- Elimination of non-generating and non-reachable symbols from a CFG, Elimination \( \lambda \) rules CFG, Chomsky normal form grammar construction, CYK (membership \( w \) in a CFL).
## Table of Content

1. Introduction, Pumping lemma for FA
2. Pumping Lemma, DFA minimization and equivalence
3. NFA, $\lambda$-transition, Closure Properties
4. Regular Expressions, Kleene Theorem, Subst., Homom.
5. Two-way DFA, Moore, Miele Machine
6. Grammars
7. Pushdown Automata
8. Normal Forms, Pumping Lemma
9. CYK algorithm, Closure properties of CFLs
10. Turing Machines
11. Linear Bounded Automata