

# Úvod do teoretické informatiky

Martin Kot    Zdeněk Sawa

Katedra informatiky, FEI  
Vysoká škola báňská – Technická universita Ostrava  
17. listopadu 15, Ostrava-Poruba 708 33  
Česká republika

27. listopadu 2006

Jméno: Zdeněk Sawa

E-mail: [zdenek.sawa@vsb.cz](mailto:zdenek.sawa@vsb.cz)

Místnost: KP314

WWW: <http://www.cs.vsb.cz/sawa/>

Tel.: 1424

Jméno: Martin Kot

E-mail: [martin.kot@vsb.cz](mailto:martin.kot@vsb.cz)

Místnost: KP314

WWW: <http://www.cs.vsb.cz/kot/>

Tel.: 1424

## Povinné:

- **Zápočet** (35 bodů):
  - Zápočtová písemka (20 bodů)
  - Vypracování zápočtového příkladu (15 bodů):
    - Písemné vypracování (ve formátu PDF) (10 bodů)
    - Předvedení na cvičení (5 bodů)
  
- **Zkouška** (65 bodů)

## Nepovinné: (bonusové body)

- Vypracování prémiového příkladu (15 bodů)

Webové stránky k předmětu naleznete na adrese:

<http://www.cs.vsb.cz/kot/uti>

Na těchto stránkách najdete:

- Informace o předmětu
- Sylaby (autorem je doc. Petr Hliněný)
- Zadání příkladů na cvičení
- Zadání zápočtových příkladů
- Zadání prémiových příkladů
- Slidy z přednášek
- Aktuální informace

- Přednášky jsou číslovány od 1 do 11.

- Cvičení jsou číslována od 0 do 11.

Cvičení číslo  $i$  se vztahuje k přednášce číslo  $i$  a mělo by být následující týden po této přednášce.

- Čísla cvičení uvedená u zápočtových příkladů udávají, na kterém cvičení budete příklad předvádět.

K vypracování zápočtových příkladů ze cvičení číslo  $i$  by měly postačovat znalosti z přednášky a cvičení číslo  $i - 1$ .

Na vypracování zápočtového příkladu budete mít víc než týden času.

Zápočtové příklady začínají cvičením číslo 2.

- Body dostane vždy první student v ročníku, který daný příklad vyřeší.
- Řešení posílejte e-mailem přednášejícím, ne cvičícím:
  - Zdeněk Sawa ([zdenek.sawa@vsb.cz](mailto:zdenek.sawa@vsb.cz))
  - Martin Kot ([martin.kot@vsb.cz](mailto:martin.kot@vsb.cz))
- Příklady budou zveřejňovány na webových stránkách postupně během semestru.
- Každý student může tímto způsobem získat 15 bodů navíc nejvýše jednou.

V tomto předmětu se budeme zabývat dvěma oblastmi teoretické informatiky:

- **Teorii formálních jazyků**
- **Výpočetní složitostí**

## Teorie formálních jazyků

- **Přednáška 1:** Úvod, základní pojmy
- **Přednáška 2:** Konečné automaty
- **Přednáška 3:** Regulární výrazy
- **Přednáška 4:** Minimalizace automatů, omezení konečných automatů
- **Přednáška 5:** Bezkontextové gramatiky
- **Přednáška 6:** Zásobníkové automaty, Chomského hierarchie



## Výpočetní složitost

- **Přednáška 7:** Algoritmy, výpočetní modely, Turingovy stroje
- **Přednáška 8:** Výpočetní složitost, analýza algoritmů
- **Přednáška 9:** Asymptotická notace, řešení rekurentních rovnic
- **Přednáška 10:** Třídy P a NP, NP-úplné problémy
- **Přednáška 11:** Důkazy NP-úplnosti některých problémů

# Motivace 1: Vyhledávání v textu

Potřebujeme řešit následující problém:

- Máme řadu různých textů (např. soubory na disku nebo webové stránky apod.).
- Potřebujeme zjistit, které z těchto textů obsahují nějaké dané slovo či frázi, případně nějakou kombinaci slov apod.

Požadujeme, aby řešení bylo:

- **Rychlé** – můžeme prohledávat mnoho MB dat
- **Dostatečně obecné** – chceme mít možnost formulovat dostatečně obecné dotazy (např. mít možnost používat booleovské spojky)

Při popisu libovolného programovacího jazyka (Java, C, C++, Pascal, ...) musí být řečeno:

1 Co jsou jeho **lexikální elementy (tokeny)** –

- identifikátory
- klíčová slova
- literály (číselné a řetězcové konstanty)
- operátory
- oddělovače
- komentáře
- ...

a jak přesně vypadají.

2 Které sekvence těchto lexikálních elementů tvoří (syntakticky) dobře utvořené programy.

Chceme řešit následující problémy:

- Jak přesně (jednoznačně) popsat jednotlivé typy lexikálních elementů?
- Jak implementovat v překladači rozpoznávání těchto jednotlivých typů?
- Jak přesně popsat všechny možné způsoby jakými je možné z lexikálních elementů „poskládat“ syntakticky správně napsaný program?
- Jak implementovat v překladači rozpoznání dobře utvořených výrazů, příkazů, procedur, metod apod.?

**Lexikální analýza** – činnost překladače, kdy rozpoznává v textu jednotlivé lexikální elementy.

**Syntaktická analýza** – činnost překladače, kdy rozpoznává v dané sekvenci lexikálních elementů např. aritmetické výrazy, příkazy, podprogramy nebo i celé programy.

Co mají všechny dosud zmíněné problémy společného?

- Pracujeme se sekvencemi **znaků** (říkáme též **symbolů**).
- Znak patří do nějaké konečné **abecedy** (typicky např. ASCII nebo Unicode).
- Musíme rozpoznávat ty sekvence znaků, které nějakou vlastnost mají a ty co ji nemají.

**Poznámka:** V teorii formálních jazyků se sekvencím znaků říká **slova**. Při programování máme na mysli například řetězce (stringy) nebo třeba soubory na disku apod.

Množině slov z nějaké abecedy se říká **jazyk**.

## Definice

**Abeceda** je libovolná (neprázdná) konečná množina **symbolů (znaků)**.

**Poznámka:** Abeceda se často označuje řeckým písmenem  $\Sigma$  (velké sigma).

## Definice

**Slovo** v dané abecedě je libovolná posloupnost symbolů z této abecedy.

## Příklad 1:

$\Sigma = \{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z\}$

Slova v abecedě  $\Sigma$ :      AHOJ      ABRACADABRA      ERROR

## Příklad 2:

$\Sigma = \{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, \_ \}$

Slovo v abecedě  $\Sigma$ : HELLO\_WORLD

## Příklad 3:

$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

Slova v abecedě  $\Sigma$ : 0, 314159, 666, 65536

## Příklad 4:

Slova v abecedě  $\Sigma = \{0, 1\}$ : 011010001, 111, 1010101010101010

## Příklad 5:

Slova v abecedě  $\Sigma = \{a, b\}$ : *aababb*, *abbabbba*, *aaab*



## Příklad 6:

Abeceda  $\Sigma$  je množina všech ASCII znaků.

Příklad slova:

```
#include <stdio.h>

int main()
{
    printf("Hello, world!\n");
    return 0;
}
```

```
#include_<stdio.h> ↔↔ int_&main() ↔ { ↔ &&&&printf("He...
```

**Délka slova** je počet znaků ve slově.

Například délka slova *abaab* je 5.

Délku slova  $w$  označujeme  $|w|$ .

Pokud tedy např.  $w = abaab$ , pak  $|w| = 5$ .

**Prázdné slovo** je slovo délky 0, tj. neobsahující žádné znaky.

Prázdné slovo se označuje řeckým písmenem  $\varepsilon$  (epsilon).

(Pozn.: Někteří autoři používají pro označení prázdného slova místo  $\varepsilon$  řecké písmeno  $\lambda$  (lambda).)

$$|\varepsilon| = 0$$

Se slovy je možné provádět operaci **zřetězení**:

Například zřetězením slov **OST** a **RAVA** vznikne slovo **OSTRAVA**.

Operace zřetězení se označuje symbolem  $\cdot$  (podobně jako násobení). Tento symbol je možné vypouštět.

$$\text{OST} \cdot \text{RAVA} = \text{OSTRAVA}$$

Zřetězení je **asociativní**, tj. pro libovolná tři slova  $u$ ,  $v$  a  $w$  platí

$$(u \cdot v) \cdot w = u \cdot (v \cdot w)$$

což znamená, že při zápisu více zřetězení můžeme vypouštět závorky a psát například  $w_1 \cdot w_2 \cdot w_3 \cdot w_4 \cdot w_5$  místo  $(w_1 \cdot (w_2 \cdot w_3)) \cdot (w_4 \cdot w_5)$

# Zřetězení slov

Zřetězení není **komutativní**, tj. obecně pro dvojici slov  $u$  a  $v$  neplatí rovnost

$$u \cdot v = v \cdot u$$

**Příklad:**

$$\text{OST} \cdot \text{RAVA} \neq \text{RAVA} \cdot \text{OST}$$

Zjevně pro libovolná slova  $v$  a  $w$  platí:

$$|v \cdot w| = |v| + |w|$$

Pro libovolné slovo  $w$  také platí:

$$\varepsilon \cdot w = w \cdot \varepsilon = w$$

Množinu všech slov tvořených symboly z abecedy  $\Sigma$  označujeme  $\Sigma^*$ .

## Definice

**(Formální) jazyk** v abecedě  $\Sigma$  je nějaká libovolná podmnožina množiny  $\Sigma^*$ .

**Příklad 1:** Množina  $\{00, 01001, 1101\}$  je jazyk v abecedě  $\{0, 1\}$

**Příklad 2:** Množina všech syntakticky správných programů v jazyce C je jazyk v abecedě tvořené množinou všech ASCII znaků.

**Příklad 3:** Množina všech textů obsahujících sekvenci znaků **ABRACADABRA** je jazyk v abecedě tvořené množinou všech ASCII znaků.

**Příklad 4:** Uvažujme abecedu  $\Sigma$  tvořenou množinou všech Unicode znaků.

Množina všech komentářů v jazyce Java tvoří jazyk:

- jednořádkové komentáře začínající dvojicí znaků `//` a končící znakem konce řádku (nebo koncem souboru).
- víceřádkové komentáře začínající dvojicí znaků `/*` a končící dvojicí znaků `*/`, přičemž uvnitř se nesmí nacházet žádná další dvojice znaků `*/`.

Pokud bychom množinu všech jednořádkových komentářů označili jako jazyk  $L_1$  a množinu všech víceřádkových komentářů jako jazyk  $L_2$ , můžeme množinu všech komentářů označit jako jazyk  $L$ , definovaný jako

$$L = L_1 \cup L_2$$

# Množinové operace na jazycích

Vzhledem k tomu, že jazyky jsou množiny, můžeme s nimi provádět množinové operace:

**Sjednocení** –  $L_1 \cup L_2$  je jazyk tvořený slovy, která patří buď do jazyka  $L_1$  nebo do jazyka  $L_2$  (nebo do obou).

**Průnik** –  $L_1 \cap L_2$  je jazyk tvořený slovy, která patří současně do jazyka  $L_1$  i do jazyka  $L_2$ .

**Doplňěk** –  $\bar{L}$  je jazyk tvořený těmi slovy ze  $\Sigma^*$ , která nepatří do  $L$ .

**Rozdíl** –  $L_1 - L_2$  je jazyk tvořený slovy, která patří do  $L_1$ , ale nepatří do  $L_2$ .

**Poznámka:** Při operacích nad jazyky předpokládáme, že jazyky, se kterými operaci provádíme, používají tutéž abecedu  $\Sigma$ .

## Příklad:

Uvažujme množinu všech textů tvořených ASCII znaky. Jestliže:

- $L_1$  je množina všech textů, ve kterých se vyskytuje sekvence znaků **FOO**, a
- $L_2$  je množina všech textů, ve kterých se vyskytuje sekvence znaků **BAR**,

pak

- $L_1 \cup L_2$  jsou všechny texty, ve kterých se vyskytuje **FOO** nebo **BAR**,
- $L_1 \cap L_2$  jsou všechny texty, ve kterých se vyskytuje **FOO** i **BAR**,
- $\overline{L_1}$  jsou všechny texty, ve kterých se nevyskytuje **FOO**,
- $L_1 - L_2$  jsou všechny texty, ve kterých se vyskytuje **FOO**, ale nevyskytuje **BAR**.



## Definice

**Zřetězení jazyků**  $L_1$  a  $L_2$  je jazyk

$$L = \{uv \mid u \in L_1, v \in L_2\}$$

tj. jazyk všech slov, která začínají slovem z  $L_1$  a pokračují slovem z  $L_2$ .  
Zřetězení jazyků  $L_1$  a  $L_2$  označujeme  $L_1 \cdot L_2$ .

**Příklad:**

$$L_1 = \{abb, ba\}$$

$$L_2 = \{a, ab, bbb\}$$

Jazyk  $L_1 \cdot L_2$  obsahuje slova:

*abba*

*abbab*

*abbbbb*

*baa*

*baab*

*babbb*

## Příklad:

Pro nějaký programovací jazyk chceme definovat, jak mohou vypadat konstanty reprezentující čísla v plovoucí řádové čárce (floating-point), např.:

1e1    2.0    .3    0.0    3.14    1E-9    4.5e137

Abeceda  $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ., e, E, +, -\}$

## Příklad:

Pro nějaký programovací jazyk chceme definovat, jak mohou vypadat konstanty reprezentující čísla v plovoucí řádové čárce (floating-point), např.:

1e1    2.0    .3    0.0    3.14    1E-9    4.5e137

Abeceda  $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ., e, E, +, -\}$

Pokud zvolíme  $L_{num}$  jako množinu všech (neprázdných) slov tvořených pouze číslicemi, a  $L_{dot} = \{.\}$ , můžeme konstanty jako například 3467.982, 3.141592 nebo 0.0 popsat takto:

$$L_{num} \cdot L_{dot} \cdot L_{num}$$

Definice jazyka  $L$  všech možných konstant v plovoucí řádové čárce by pak mohla vypadat například takto:

$$\begin{aligned} L = & L_{num} \cdot L_{dot} \cdot (L_{num} \cup \{\varepsilon\}) \cdot (L_{exp} \cup \{\varepsilon\}) \cup \\ & L_{dot} \cdot L_{num} \cdot (L_{exp} \cup \{\varepsilon\}) \cup \\ & L_{num} \cdot L_{exp} \end{aligned}$$

kde

$$\begin{aligned} L_{num} & - \text{neprázdné sekvence číslic} \\ L_{dot} & = \{.\} \\ L_{exp} & = L_E \cdot L_{sign} \cdot L_{num} \\ L_E & = \{E, e\} \\ L_{sign} & = \{\varepsilon, +, -\} \end{aligned}$$

Používáme následující zápis:

$$\begin{aligned}L^2 &= L \cdot L \\L^3 &= L \cdot L \cdot L \\L^4 &= L \cdot L \cdot L \cdot L \\L^5 &= L \cdot L \cdot L \cdot L \cdot L \\&\dots\end{aligned}$$

**Příklad:** Pokud  $L = \{aa, b\}$ , pak  $L^3$  obsahuje slova:

*aaaaaa aaaab aabaa aabb baaaa baab bbaa bbb*

Definujeme

$$\begin{aligned}L^1 &= L \\L^0 &= \{\varepsilon\}\end{aligned}$$

# Iterace jazyka

Induktivní definice pro libovolné  $k \geq 0$ :

$$L^0 = \{\varepsilon\}, \quad L^{k+1} = L^k \cdot L \quad \text{pro } k \geq 0.$$

## Definice

**Iterace jazyka**  $L$  je jazyk

$$L^* = L^0 \cup L^1 \cup L^2 \cup L^3 \cup \dots$$

tj. jazyk tvořený slovy vzniklými zřetěžením libovolného počtu slov z jazyka  $L$ .

**Příklad:**  $L = \{aa, b\}$

$$L^* = \{\varepsilon, aa, b, aaaa, aab, baa, bb, aaaaaa, aaaab, aabaa, aabb, \dots\}$$

**Příklad:**  $L_{dig} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ .

$$L_{num} = L_{dig} \cdot L_{dig}^*$$

**Poznámka:** Používá se také následující značení:

$$L^+ = L^1 \cup L^2 \cup L^3 \cup L^4 \cup \dots$$

Řešení předchozího příkladu bychom tedy také mohli zapsat stručněji:

$$L_{num} = L_{dig}^+$$

Když chceme nějaký jazyk popsat, máme několik možností:

- Můžeme vyjmenovat všechna jeho slova (což je ale použitelné jen pro konečné jazyky).

**Příklad:**  $L = \{aab, babba, aaaaaa\}$

- Můžeme specifikovat nějakou vlastnost, kterou mají právě ta slova, která do tohoto jazyka patří:

**Příklad:** Jazyk nad abecedou  $\{0, 1\}$ , obsahující všechna slova, ve kterých je počet výskytů symbolu  $1$  sudý.



V teorii formálních jazyků se používají především následující dva přístupy:

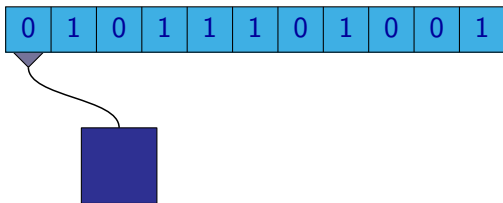
- Popsat (idealizovaný) stroj, zařízení, algoritmus, který rozpozná slova patřící do daného jazyka – vede k použití tzv. **automatů**.
- Popsat postup, jak mechanicky generovat všechna možná slova patřící do daného jazyka – vede k tzv. **gramatikám** a **regulárním výrazům**. (budeme se jim věnovat později)

# Rozpoznávání jazyka

**Příklad:** Uvažujme slova nad abecedou  $\{0, 1\}$ .

Chtěli bychom rozpoznávat jazyk  $L$ , který je tvořen slovy, ve kterých se vyskytuje sudý počet symbolů  $1$ .

Chceme navrhnout zařízení, které přečte slovo, a sdělí nám, zda toto slovo patří do jazyka  $L$  či ne.

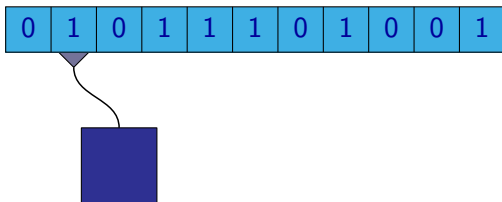


# Rozpoznávání jazyka

**Příklad:** Uvažujme slova nad abecedou  $\{0, 1\}$ .

Chtěli bychom rozpoznávat jazyk  $L$ , který je tvořen slovy, ve kterých se vyskytuje sudý počet symbolů  $1$ .

Chceme navrhnout zařízení, které přečte slovo, a sdělí nám, zda toto slovo patří do jazyka  $L$  či ne.

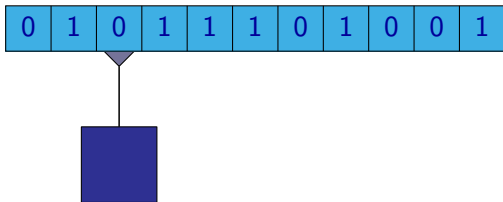


# Rozpoznávání jazyka

**Příklad:** Uvažujme slova nad abecedou  $\{0, 1\}$ .

Chtěli bychom rozpoznávat jazyk  $L$ , který je tvořen slovy, ve kterých se vyskytuje sudý počet symbolů  $1$ .

Chceme navrhnout zařízení, které přečte slovo, a sdělí nám, zda toto slovo patří do jazyka  $L$  či ne.

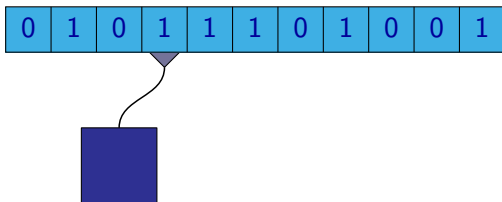


# Rozpoznávání jazyka

**Příklad:** Uvažujme slova nad abecedou  $\{0, 1\}$ .

Chtěli bychom rozpoznávat jazyk  $L$ , který je tvořen slovy, ve kterých se vyskytuje sudý počet symbolů  $1$ .

Chceme navrhnout zařízení, které přečte slovo, a sdělí nám, zda toto slovo patří do jazyka  $L$  či ne.

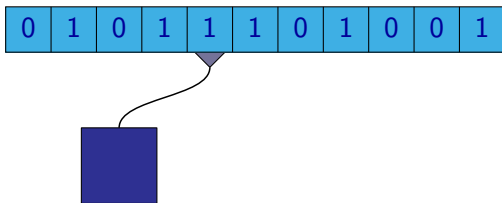


# Rozpoznávání jazyka

**Příklad:** Uvažujme slova nad abecedou  $\{0, 1\}$ .

Chtěli bychom rozpoznávat jazyk  $L$ , který je tvořen slovy, ve kterých se vyskytuje sudý počet symbolů  $1$ .

Chceme navrhnout zařízení, které přečte slovo, a sdělí nám, zda toto slovo patří do jazyka  $L$  či ne.

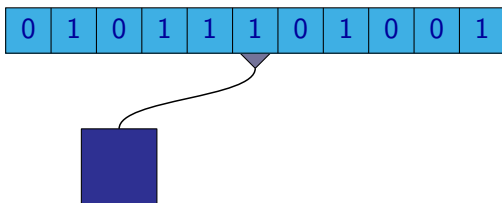


# Rozpoznávání jazyka

**Příklad:** Uvažujme slova nad abecedou  $\{0, 1\}$ .

Chtěli bychom rozpoznávat jazyk  $L$ , který je tvořen slovy, ve kterých se vyskytuje sudý počet symbolů  $1$ .

Chceme navrhnout zařízení, které přečte slovo, a sdělí nám, zda toto slovo patří do jazyka  $L$  či ne.

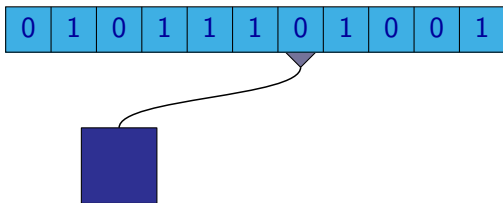


# Rozpoznávání jazyka

**Příklad:** Uvažujme slova nad abecedou  $\{0, 1\}$ .

Chtěli bychom rozpoznávat jazyk  $L$ , který je tvořen slovy, ve kterých se vyskytuje sudý počet symbolů  $1$ .

Chceme navrhnout zařízení, které přečte slovo, a sdělí nám, zda toto slovo patří do jazyka  $L$  či ne.



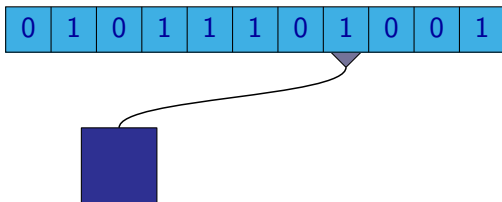


# Rozpoznávání jazyka

**Příklad:** Uvažujme slova nad abecedou  $\{0, 1\}$ .

Chtěli bychom rozpoznávat jazyk  $L$ , který je tvořen slovy, ve kterých se vyskytuje sudý počet symbolů  $1$ .

Chceme navrhnout zařízení, které přečte slovo, a sdělí nám, zda toto slovo patří do jazyka  $L$  či ne.

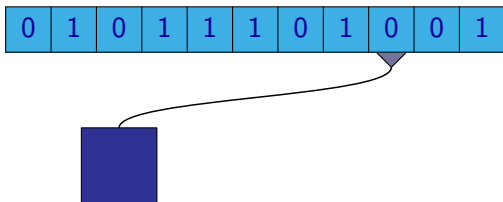


# Rozpoznávání jazyka

**Příklad:** Uvažujme slova nad abecedou  $\{0, 1\}$ .

Chtěli bychom rozpoznávat jazyk  $L$ , který je tvořen slovy, ve kterých se vyskytuje sudý počet symbolů  $1$ .

Chceme navrhnout zařízení, které přečte slovo, a sdělí nám, zda toto slovo patří do jazyka  $L$  či ne.

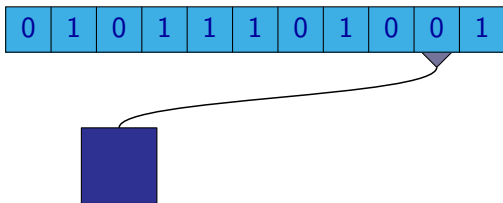


# Rozpoznávání jazyka

**Příklad:** Uvažujme slova nad abecedou  $\{0, 1\}$ .

Chtěli bychom rozpoznávat jazyk  $L$ , který je tvořen slovy, ve kterých se vyskytuje sudý počet symbolů  $1$ .

Chceme navrhnout zařízení, které přečte slovo, a sdělí nám, zda toto slovo patří do jazyka  $L$  či ne.

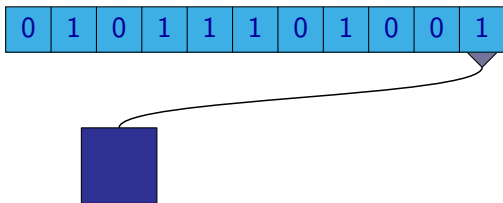


# Rozpoznávání jazyka

**Příklad:** Uvažujme slova nad abecedou  $\{0, 1\}$ .

Chtěli bychom rozpoznávat jazyk  $L$ , který je tvořen slovy, ve kterých se vyskytuje sudý počet symbolů  $1$ .

Chceme navrhnout zařízení, které přečte slovo, a sdělí nám, zda toto slovo patří do jazyka  $L$  či ne.

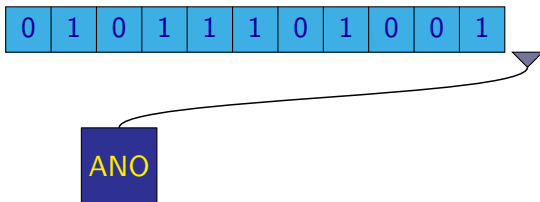


# Rozpoznávání jazyka

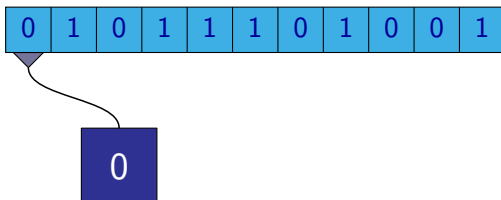
**Příklad:** Uvažujme slova nad abecedou  $\{0, 1\}$ .

Chtěli bychom rozpoznávat jazyk  $L$ , který je tvořen slovy, ve kterých se vyskytuje sudý počet symbolů  $1$ .

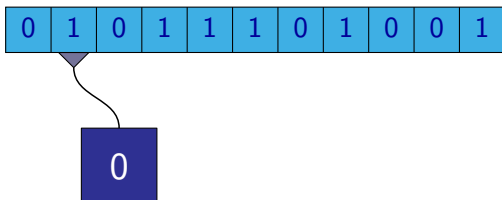
Chceme navrhnout zařízení, které přečte slovo, a sdělí nám, zda toto slovo patří do jazyka  $L$  či ne.



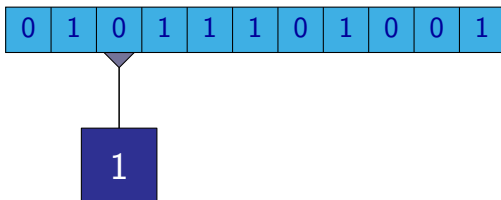
**První nápad:** Počítat počet výskytů symbolů 1.



**První nápad:** Počítat počet výskytů symbolů 1.

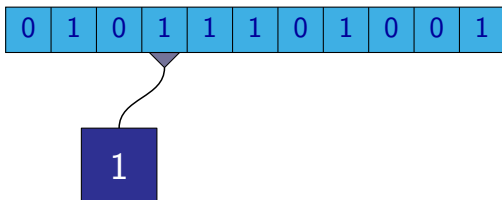


**První nápad:** Počítat počet výskytů symbolů 1.

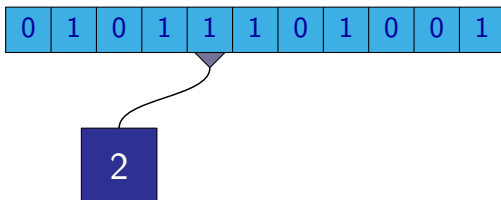




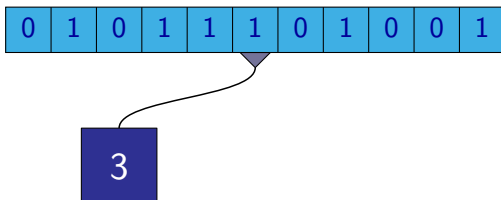
**První nápad:** Počítat počet výskytů symbolů 1.



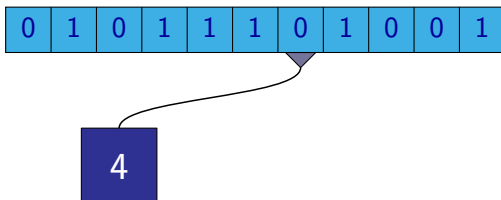
**První nápad:** Počítat počet výskytů symbolů 1.



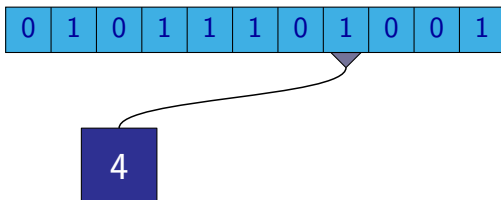
**První nápad:** Počítat počet výskytů symbolů 1.



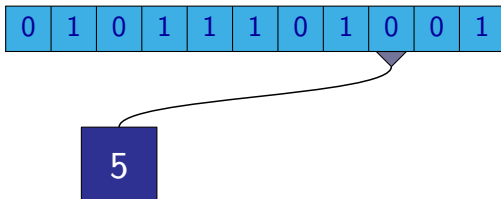
**První nápad:** Počítat počet výskytů symbolů 1.



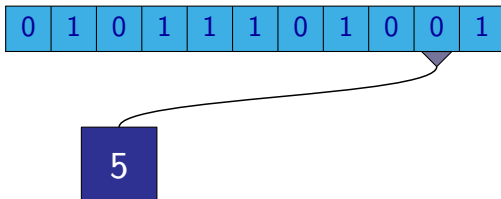
**První nápad:** Počítat počet výskytů symbolů 1.



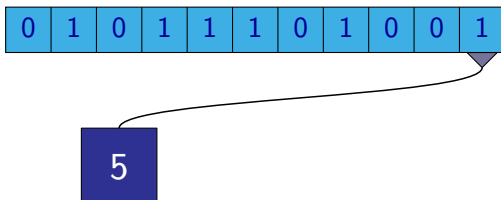
**První nápad:** Počítat počet výskytů symbolů 1.



**První nápad:** Počítat počet výskytů symbolů 1.



**První nápad:** Počítat počet výskytů symbolů 1.





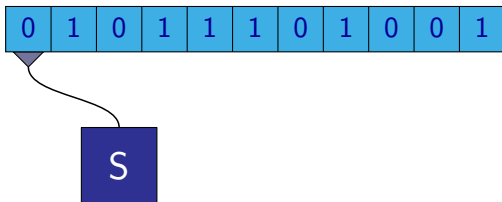
**První nápad:** Počítat počet výskytů symbolů 1.

0	1	0	1	1	1	0	1	0	0	1
---	---	---	---	---	---	---	---	---	---	---

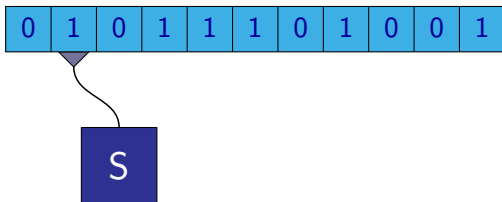
6

ANO – 6 je sudé číslo

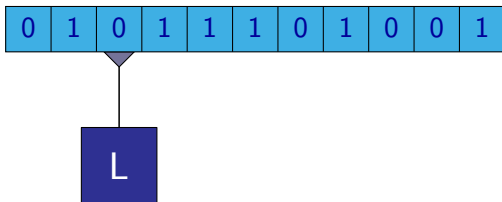
**Druhý nápad:** Ve skutečnosti nás zajímá pouze, zda počet dosud přečtených symbolů **1** je sudý nebo lichý (místo čísla si stačí pamatovat jen jeho poslední bit).



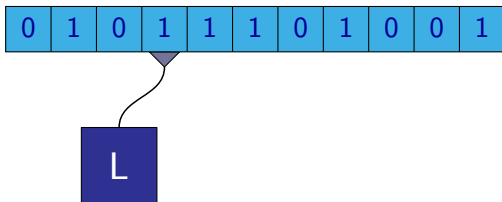
**Druhý nápad:** Ve skutečnosti nás zajímá pouze, zda počet dosud přečtených symbolů **1** je sudý nebo lichý (místo čísla si stačí pamatovat jen jeho poslední bit).



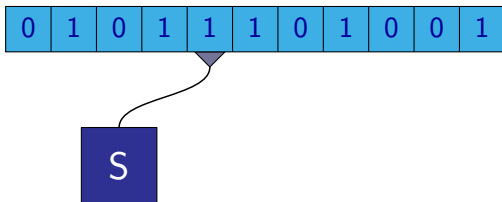
**Druhý nápad:** Ve skutečnosti nás zajímá pouze, zda počet dosud přečtených symbolů **1** je sudý nebo lichý (místo čísla si stačí pamatovat jen jeho poslední bit).



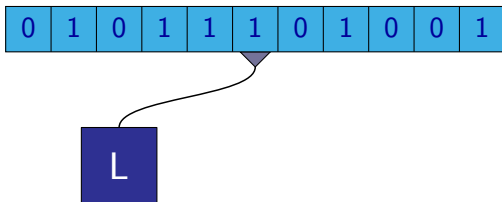
**Druhý nápad:** Ve skutečnosti nás zajímá pouze, zda počet dosud přečtených symbolů **1** je sudý nebo lichý (místo čísla si stačí pamatovat jen jeho poslední bit).



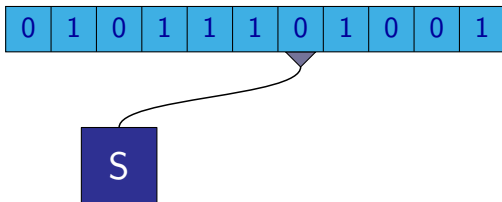
**Druhý nápad:** Ve skutečnosti nás zajímá pouze, zda počet dosud přečtených symbolů **1** je sudý nebo lichý (místo čísla si stačí pamatovat jen jeho poslední bit).



**Druhý nápad:** Ve skutečnosti nás zajímá pouze, zda počet dosud přečtených symbolů **1** je sudý nebo lichý (místo čísla si stačí pamatovat jen jeho poslední bit).

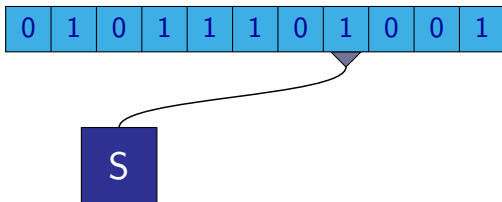


**Druhý nápad:** Ve skutečnosti nás zajímá pouze, zda počet dosud přečtených symbolů **1** je sudý nebo lichý (místo čísla si stačí pamatovat jen jeho poslední bit).

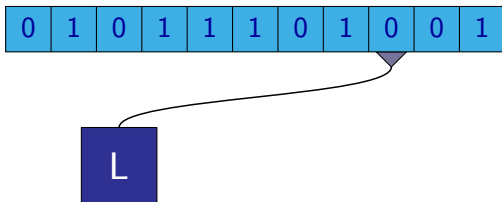




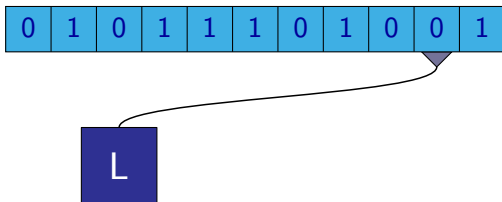
**Druhý nápad:** Ve skutečnosti nás zajímá pouze, zda počet dosud přečtených symbolů **1** je sudý nebo lichý (místo čísla si stačí pamatovat jen jeho poslední bit).



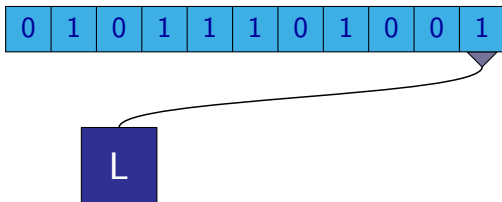
**Druhý nápad:** Ve skutečnosti nás zajímá pouze, zda počet dosud přečtených symbolů **1** je sudý nebo lichý (místo čísla si stačí pamatovat jen jeho poslední bit).



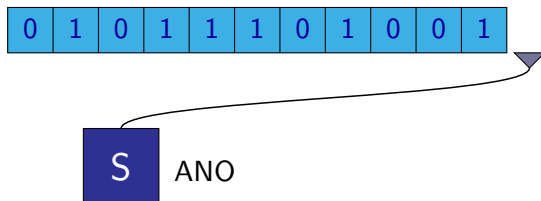
**Druhý nápad:** Ve skutečnosti nás zajímá pouze, zda počet dosud přečtených symbolů **1** je sudý nebo lichý (místo čísla si stačí pamatovat jen jeho poslední bit).



**Druhý nápad:** Ve skutečnosti nás zajímá pouze, zda počet dosud přečtených symbolů **1** je sudý nebo lichý (místo čísla si stačí pamatovat jen jeho poslední bit).



**Druhý nápad:** Ve skutečnosti nás zajímá pouze, zda počet dosud přečtených symbolů **1** je sudý nebo lichý (místo čísla si stačí pamatovat jen jeho poslední bit).



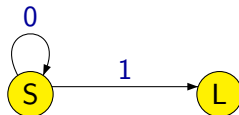
Chování tohoto zařízení můžeme popsat grafem:



Chování tohoto zařízení můžeme popsat grafem:

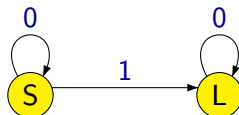


Chování tohoto zařízení můžeme popsat grafem:

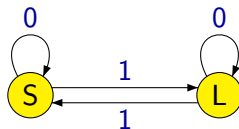




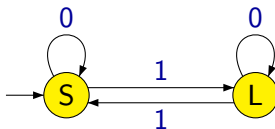
Chování tohoto zařízení můžeme popsat grafem:



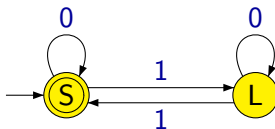
Chování tohoto zařízení můžeme popsat grafem:



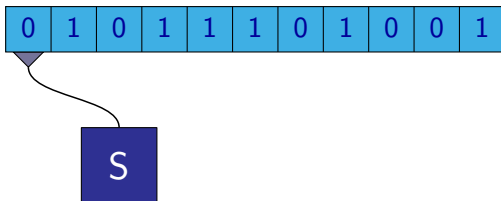
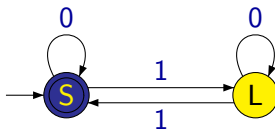
Chování tohoto zařízení můžeme popsat grafem:



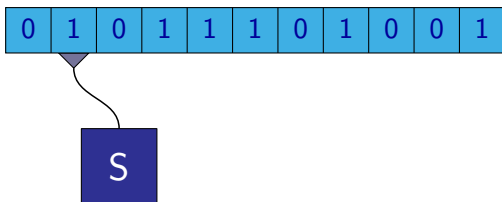
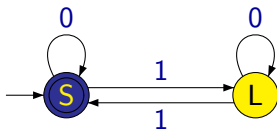
Chování tohoto zařízení můžeme popsat grafem:



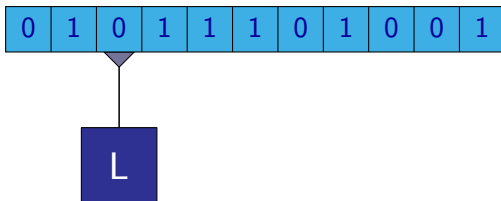
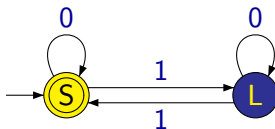
Chování tohoto zařízení můžeme popsat grafem:



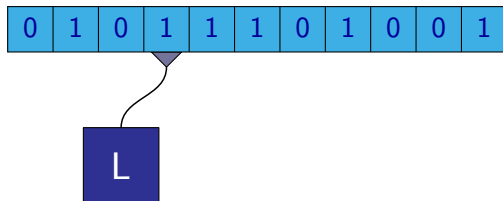
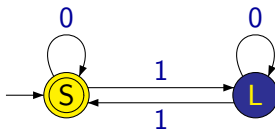
Chování tohoto zařízení můžeme popsat grafem:



Chování tohoto zařízení můžeme popsat grafem:

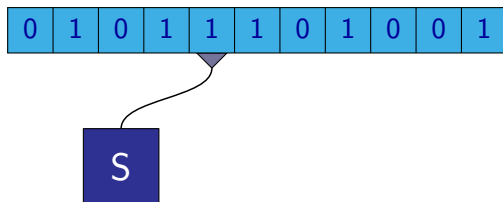
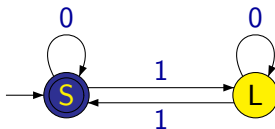


Chování tohoto zařízení můžeme popsat grafem:

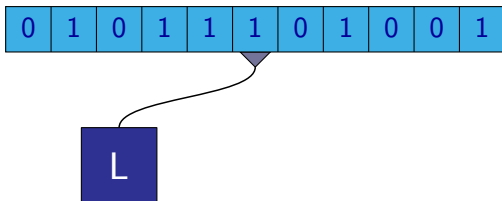
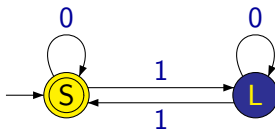




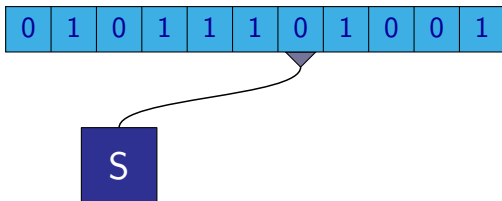
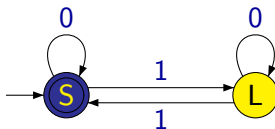
Chování tohoto zařízení můžeme popsat grafem:



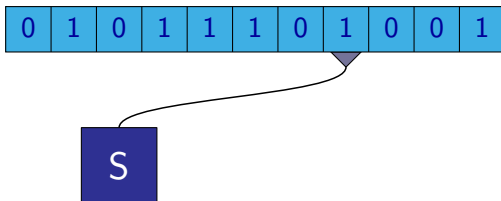
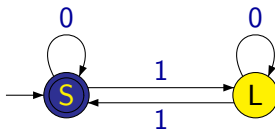
Chování tohoto zařízení můžeme popsat grafem:



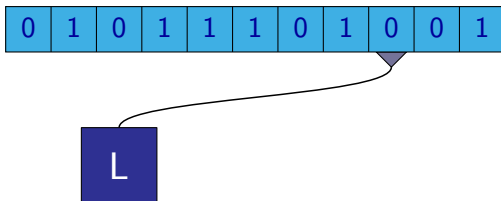
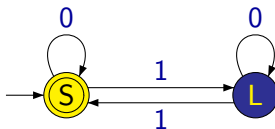
Chování tohoto zařízení můžeme popsat grafem:



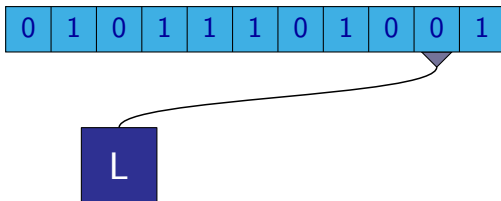
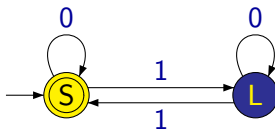
Chování tohoto zařízení můžeme popsat grafem:



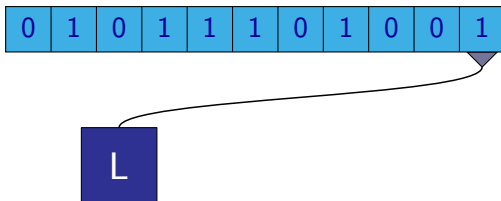
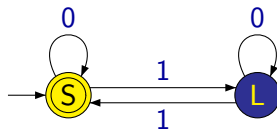
Chování tohoto zařízení můžeme popsat grafem:



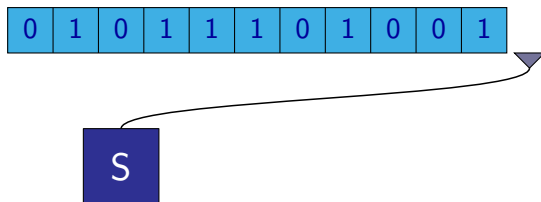
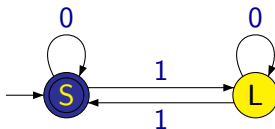
Chování tohoto zařízení můžeme popsat grafem:



Chování tohoto zařízení můžeme popsat grafem:



Chování tohoto zařízení můžeme popsat grafem:





## Problém

Vstup:  $t$  – dlouhý text,  $s$  – hledané řetězec

Výstup: ANO – pokud se řetězec  $s$  nachází v textu  $t$ ,  
NE – pokud se tam nenachází

Například chceme zjistit, zda se v textu **aaababaabab** nachází řetězec **abaa**.

Jednoduchý algoritmus, který nás asi napadne v první chvíli:

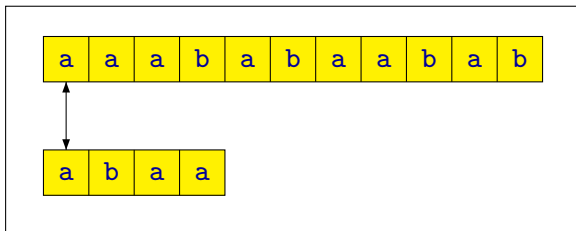
NAIVNÍ-VYHLEDÁVÁNÍ( $t, s$ )

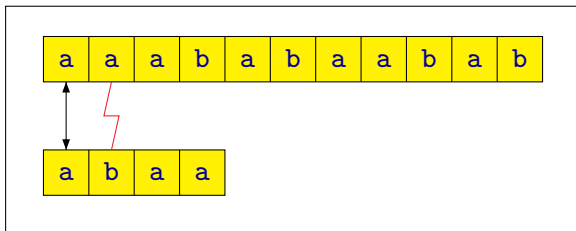
```
1   $n \leftarrow \text{length}(t)$ 
2   $m \leftarrow \text{length}(s)$ 
3  for  $i \leftarrow 0$  to  $n - m$ 
4      do  $\text{nalezen} \leftarrow \text{TRUE}$ 
5          for  $j \leftarrow 0$  to  $m - 1$ 
6              do if  $s[j] \neq t[i + j]$ 
7                  then  $\text{nalezen} \leftarrow \text{FALSE};$  break
8          if  $\text{nalezen}$ 
9              then return  $\text{TRUE}$ 
10 return  $\text{FALSE}$ 
```

a a a b a b a a b a b

a b a a

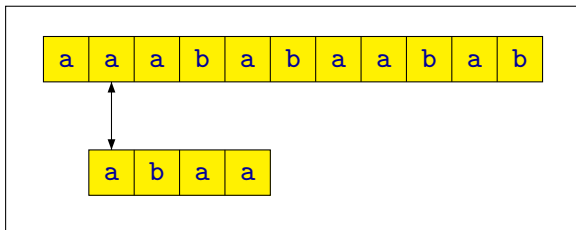
# Vyhledávání v textu

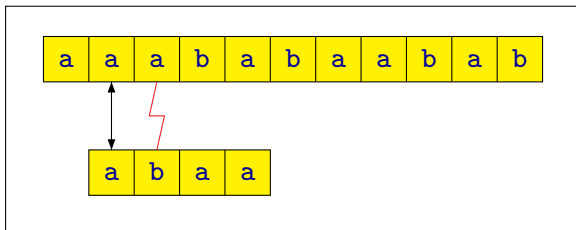




a a a b a b a a b a b

a b a a



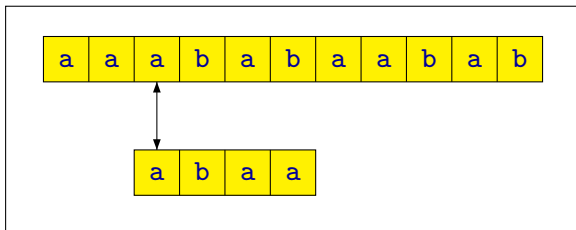


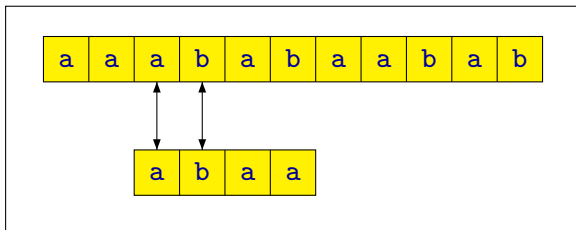


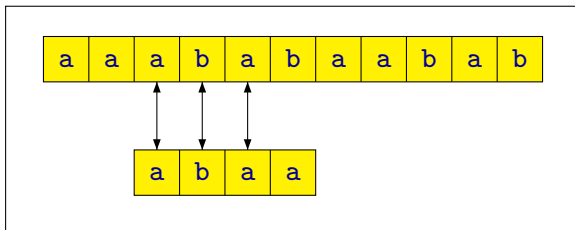
a a a b a b a a b a b

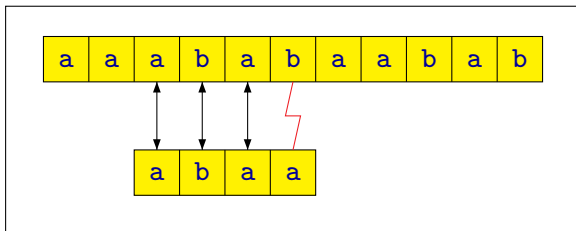
a b a a

# Vyhledávání v textu



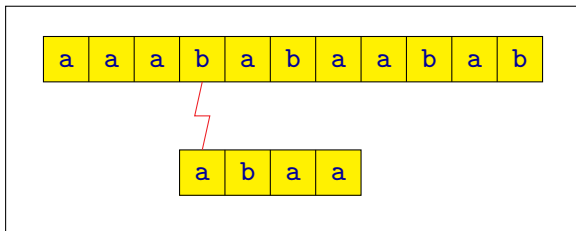






a a a b a b a a b a b

a b a a

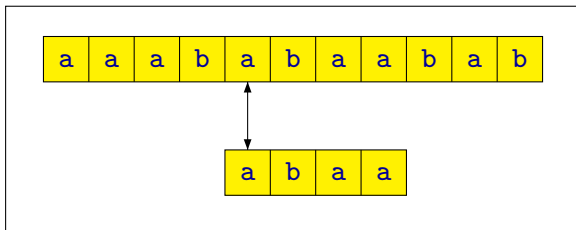


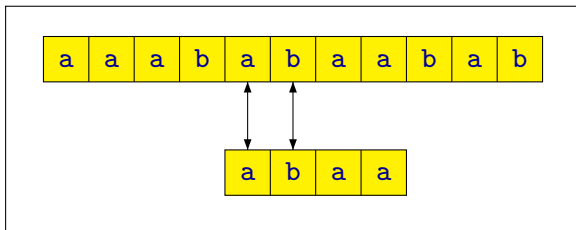
# Vyhledávání v textu

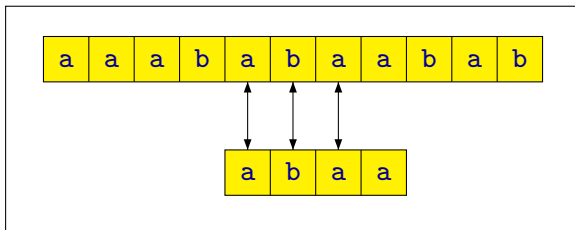
a a a b a b a a b a b

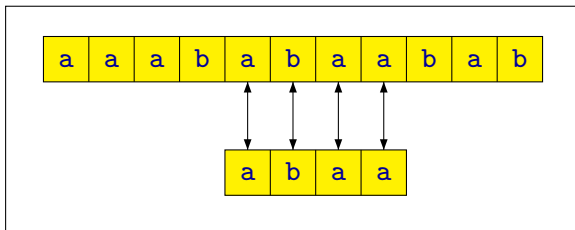
a b a a











**Pozorování:** Nijak nevyužíváme informaci o části slova  $s$ , která souhlasí, a v dalším kroku začínáme zase od začátku slova  $s$ .

**Nápad:** Slovo  $t$  číst znak po znaku a pamatovat si jaká část slova  $s$  s koncem dosud načteného textu. Upravovat tento údaj vždy jen na základě dalšího jednoho načteného znaku:

## LEPŠÍ-VYHLEDÁVÁNÍ( $t, s$ )

```
1   $n \leftarrow \text{length}(t)$ 
2   $m \leftarrow \text{length}(s)$ 
3   $q \leftarrow 0$ 
4  for  $i \leftarrow 0$  to  $n - 1$ 
5      do  $q \leftarrow \delta(q, t[i])$ 
6          if  $q = m$ 
7              then return TRUE
8  return FALSE
```

# Prefixy, sufixy a podslova

V této souvislosti se nám hodí tři následující pojmy:

## Definice

Slovo  $x$  je **prefixem** slova  $y$ , jestliže existuje slovo  $v$  takové, že  $y = xv$ .

Slovo  $x$  je **sufixem** slova  $y$ , jestliže existuje slovo  $u$  takové, že  $y = ux$ .

Slovo  $x$  je **podslovem** slova  $y$ , jestliže existují slova  $u$  a  $v$  taková, že  $y = uxv$ .

## Příklad:

- Prefixy slova **abaab** jsou  $\varepsilon$ , **a**, **ab**, **aba**, **abaa**, **abaab**.
- Sufixy slova **abaab** jsou  $\varepsilon$ , **b**, **ab**, **aab**, **baab**, **abaab**.
- Podslova slova **abaab** jsou  $\varepsilon$ , **a**, **b**, **ab**, **ba**, **ab**, **aba**, **baa**, **abb**, **abaa**, **baab**, **abaab**.

Hodnota  $q$ , kterou si během výpočtu pamatujeme je tedy délka prefixu slova  $s$ , který současně sufixem dosud přečtené části slova  $t$ .

**Poznámka:** Pokud je takových prefixů víc, pamatujeme si délku nejdelšího z nich.

Proměnná zjevně může nabývat jen hodnot  $\{0, 1, \dots, m\}$ . Hodnoty  $m$  nabývá jen v případě, že bylo nalezeno slovo  $s$ .

Hodnota  $q$ , kterou si během výpočtu pamatujeme je tedy délka prefixu slova  $s$ , který současně sufixem dosud přečtené části slova  $t$ .

**Poznámka:** Pokud je takových prefixů víc, pamatujeme si délku nejdelšího z nich.

Proměnná zjevně může nabývat jen hodnot  $\{0, 1, \dots, m\}$ . Hodnoty  $m$  nabývá jen v případě, že bylo nalezeno slovo  $s$ .

Chování tohoto systému si opět můžeme znázornit jako graf:

0  
 $\epsilon$

1  
a

2  
ab

3  
aba

4  
abaa



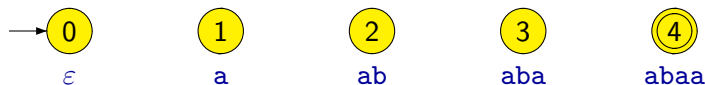
# Vyhledávání v textu

Hodnota  $q$ , kterou si během výpočtu pamatujeme je tedy délka prefixu slova  $s$ , který současně sufixem dosud přečtené části slova  $t$ .

**Poznámka:** Pokud je takových prefixů víc, pamatujeme si délku nejdelšího z nich.

Proměnná zjevně může nabývat jen hodnot  $\{0, 1, \dots, m\}$ . Hodnoty  $m$  nabývá jen v případě, že bylo nalezeno slovo  $s$ .

Chování tohoto systému si opět můžeme znázornit jako graf:



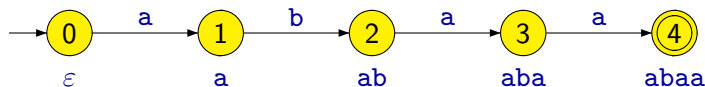
# Vyhledávání v textu

Hodnota  $q$ , kterou si během výpočtu pamatujeme je tedy délka prefixu slova  $s$ , který současně sufixem dosud přečtené části slova  $t$ .

**Poznámka:** Pokud je takových prefixů víc, pamatujeme si délku nejdelšího z nich.

Proměnná zjevně může nabývat jen hodnot  $\{0, 1, \dots, m\}$ . Hodnoty  $m$  nabývá jen v případě, že bylo nalezeno slovo  $s$ .

Chování tohoto systému si opět můžeme znázornit jako graf:



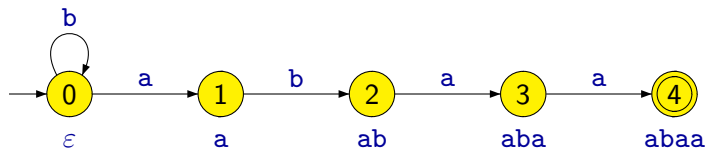
# Vyhledávání v textu

Hodnota  $q$ , kterou si během výpočtu pamatujeme je tedy délka prefixu slova  $s$ , který současně sufixem dosud přečtené části slova  $t$ .

**Poznámka:** Pokud je takových prefixů víc, pamatujeme si délku nejdelšího z nich.

Proměnná zjevně může nabývat jen hodnot  $\{0, 1, \dots, m\}$ . Hodnoty  $m$  nabývá jen v případě, že bylo nalezeno slovo  $s$ .

Chování tohoto systému si opět můžeme znázornit jako graf:



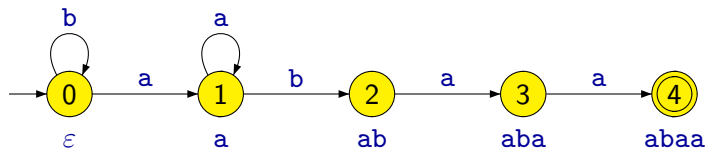
# Vyhledávání v textu

Hodnota  $q$ , kterou si během výpočtu pamatujeme je tedy délka prefixu slova  $s$ , který současně sufikem dosud přečtené části slova  $t$ .

**Poznámka:** Pokud je takových prefixů víc, pamatujeme si délku nejdelšího z nich.

Proměnná zjevně může nabývat jen hodnot  $\{0, 1, \dots, m\}$ . Hodnoty  $m$  nabývá jen v případě, že bylo nalezeno slovo  $s$ .

Chování tohoto systému si opět můžeme znázornit jako graf:



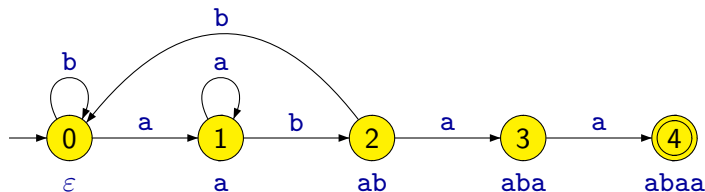
# Vyhledávání v textu

Hodnota  $q$ , kterou si během výpočtu pamatujeme je tedy délka prefixu slova  $s$ , který současně sufixem dosud přečtené části slova  $t$ .

**Poznámka:** Pokud je takových prefixů víc, pamatujeme si délku nejdelšího z nich.

Proměnná zjevně může nabývat jen hodnot  $\{0, 1, \dots, m\}$ . Hodnoty  $m$  nabývá jen v případě, že bylo nalezeno slovo  $s$ .

Chování tohoto systému si opět můžeme znázornit jako graf:



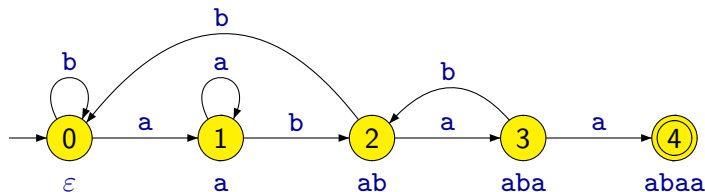
# Vyhledávání v textu

Hodnota  $q$ , kterou si během výpočtu pamatujeme je tedy délka prefixu slova  $s$ , který současně sufixem dosud přečtené části slova  $t$ .

**Poznámka:** Pokud je takových prefixů víc, pamatujeme si délku nejdelšího z nich.

Proměnná zjevně může nabývat jen hodnot  $\{0, 1, \dots, m\}$ . Hodnoty  $m$  nabývá jen v případě, že bylo nalezeno slovo  $s$ .

Chování tohoto systému si opět můžeme znázornit jako graf:



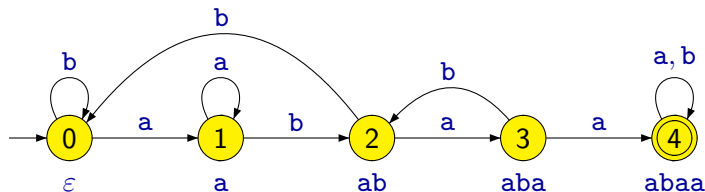
# Vyhledávání v textu

Hodnota  $q$ , kterou si během výpočtu pamatujeme je tedy délka prefixu slova  $s$ , který současně sufixem dosud přečtené části slova  $t$ .

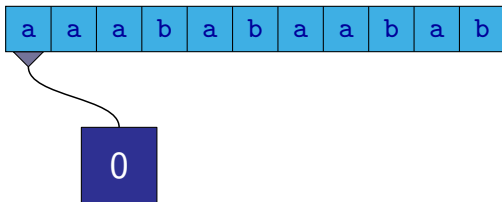
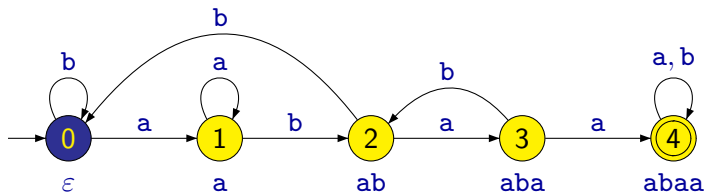
**Poznámka:** Pokud je takových prefixů víc, pamatujeme si délku nejdelšího z nich.

Proměnná zjevně může nabývat jen hodnot  $\{0, 1, \dots, m\}$ . Hodnoty  $m$  nabývá jen v případě, že bylo nalezeno slovo  $s$ .

Chování tohoto systému si opět můžeme znázornit jako graf:

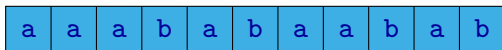
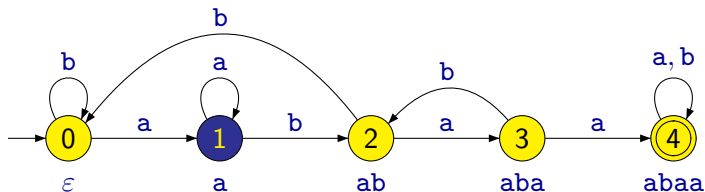


# Vyhledávání v textu

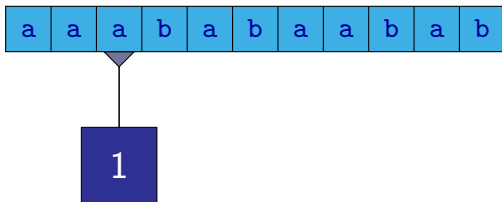
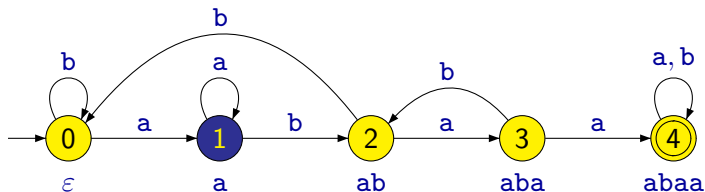




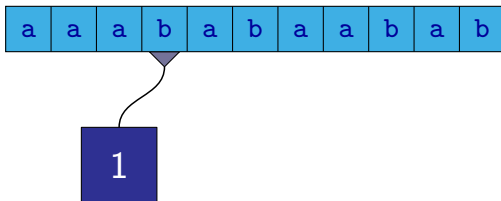
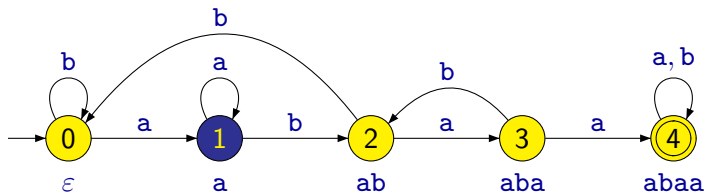
# Vyhledávání v textu



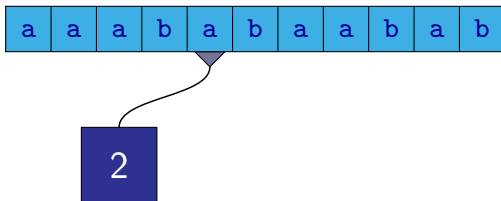
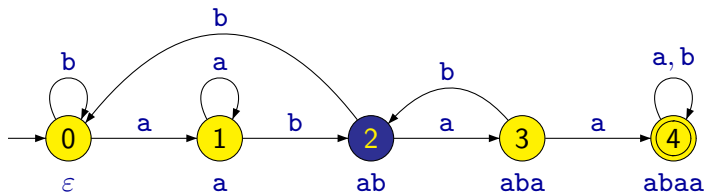
# Vyhledávání v textu



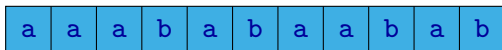
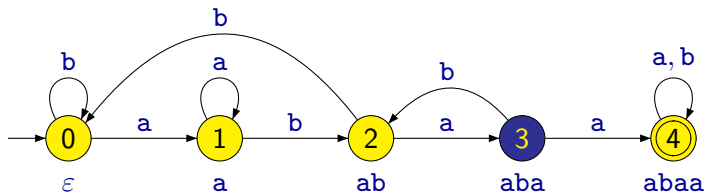
# Vyhledávání v textu



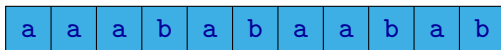
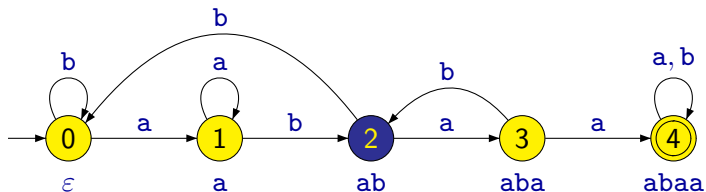
# Vyhledávání v textu



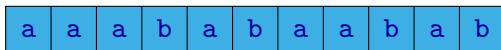
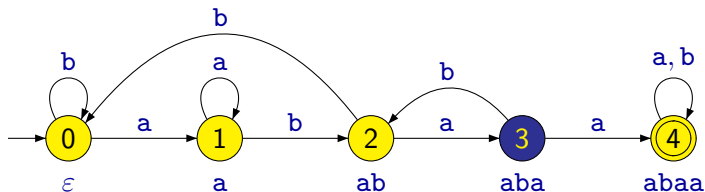
# Vyhledávání v textu



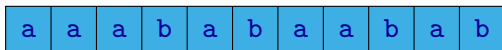
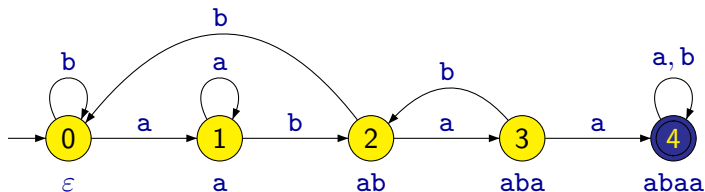
# Vyhledávání v textu



# Vyhledávání v textu

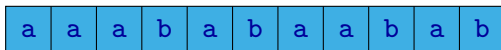
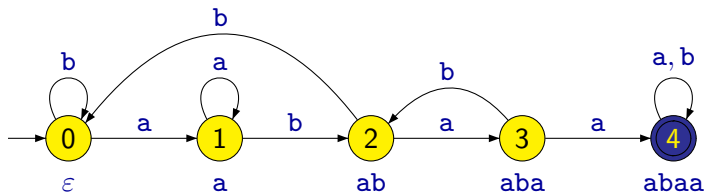


# Vyhledávání v textu

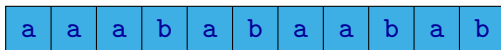
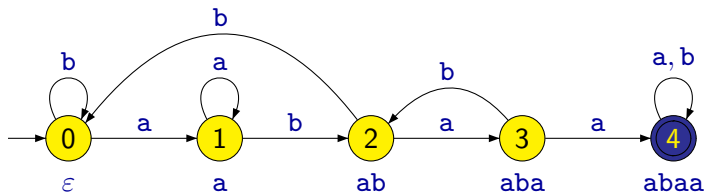




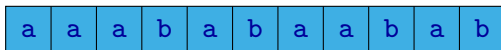
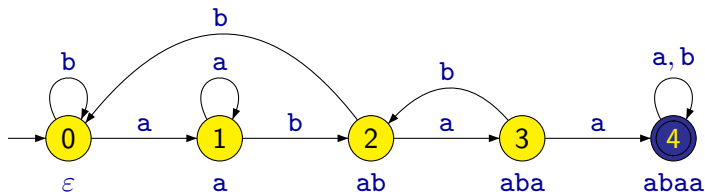
# Vyhledávání v textu

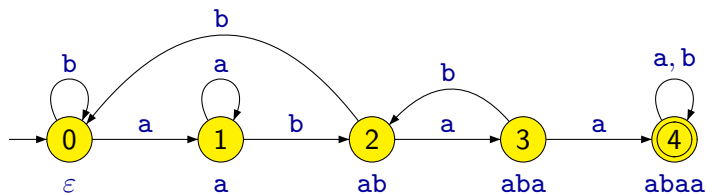


# Vyhledávání v textu



# Vyhledávání v textu





Místo grafu můžeme stejnou informaci reprezentovat tabulkou:

	a	b
→ 0	1	0
1	1	2
2	3	0
3	4	2
← 4	4	4

Jak zvolit pro danou dvojici  $q$  a  $t_i$  novou hodnotu  $q$ , tj. hodnotu  $\delta(q, t_i)$ ?

Jak zvolit pro danou dvojici  $q$  a  $t_i$  novou hodnotu  $q$ , tj. hodnotu  $\delta(q, t_i)$ ?

Zvolíme  $\delta(q, t_i) = q'$  takové, že slovo  $s_0s_1 \cdots s_{q'-1}$  je nejdelším prefixem slova  $s$  takovým, že je současně suffixem slova  $s_0s_1 \cdots s_{q-1}t_i$ .

**Poznámka:** Předpokládáme, že  $s = s_0s_1 \cdots s_{m-1}$ .

Jak zvolit pro danou dvojici  $q$  a  $t_i$  novou hodnotu  $q$ , tj. hodnotu  $\delta(q, t_i)$  ?

Zvolíme  $\delta(q, t_i) = q'$  takové, že slovo  $s_0s_1 \cdots s_{q'-1}$  je nejdelším prefixem slova  $s$  takovým, že je současně suffixem slova  $s_0s_1 \cdots s_{q-1}t_i$ .

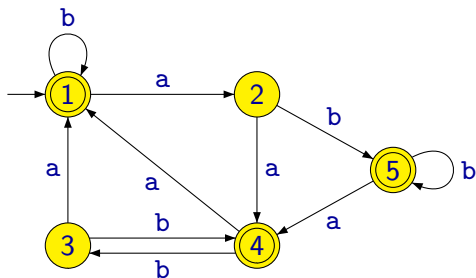
**Poznámka:** Předpokládáme, že  $s = s_0s_1 \cdots s_{m-1}$ .

**Poznámka:** Výše uvedené úvahy vedou k algoritmu nazývanému podle jeho autorů Knuth-Morris-Pratt.

V tomto algoritmu se vyhneme tomu, že bychom výše uvedenou tabulku skutečně sestrojili. Místo ní se sestrojí určitá její stručnější reprezentace, která ale umožňuje hodnoty z tabulky rychle vypočítat.

Zde se ale nebudeme tímto algoritmem blíže zabývat.

# Deterministický konečný automat



**Deterministický konečný automat** se skládá ze **stavů** a **přechodů**. Jeden ze stavů je označen jako **počáteční stav** a některé ze stavů jsou označeny jako přijímající.



# Deterministický konečný automat

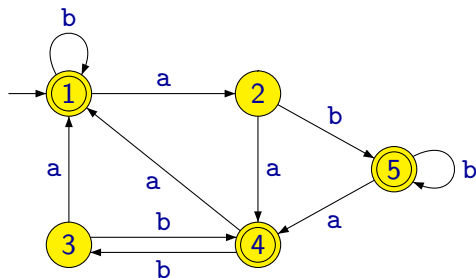
Formálně je **deterministický konečný automat** definován jako pětice

$$(Q, \Sigma, \delta, q_0, F)$$

kde:

- $Q$  je konečná množina **stavů**
- $\Sigma$  je konečná **abeceda**
- $\delta : Q \times \Sigma \rightarrow Q$  je **přechodová funkce**
- $q_0 \in Q$  je **počáteční stav**
- $F \subseteq Q$  je množina **přijímajících stavů**

# Deterministický konečný automat



- $Q = \{1, 2, 3, 4, 5\}$

- $\Sigma = \{a, b\}$

- $q_0 = 1$

- $F = \{1, 4, 5\}$

$$\delta(1, a) = 2 \quad \delta(1, b) = 1$$

$$\delta(2, a) = 4 \quad \delta(2, b) = 5$$

$$\delta(3, a) = 1 \quad \delta(3, b) = 4$$

$$\delta(4, a) = 1 \quad \delta(4, b) = 3$$

$$\delta(5, a) = 4 \quad \delta(5, b) = 5$$

# Deterministický konečný automat

Místo zápisu

$$\begin{array}{ll} \delta(1, a) = 2 & \delta(1, b) = 1 \\ \delta(2, a) = 4 & \delta(2, b) = 5 \\ \delta(3, a) = 1 & \delta(3, b) = 4 \\ \delta(4, a) = 1 & \delta(4, b) = 3 \\ \delta(5, a) = 4 & \delta(5, b) = 5 \end{array}$$

budeme raději používat stručnější tabulku nebo grafické znázornění:

$\delta$	a	b
1	2	1
2	4	5
3	1	4
4	1	3
5	4	5

# Deterministický konečný automat

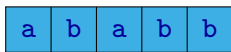
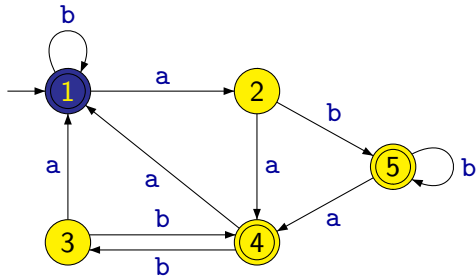
Místo zápisu

$$\begin{array}{ll} \delta(1, a) = 2 & \delta(1, b) = 1 \\ \delta(2, a) = 4 & \delta(2, b) = 5 \\ \delta(3, a) = 1 & \delta(3, b) = 4 \\ \delta(4, a) = 1 & \delta(4, b) = 3 \\ \delta(5, a) = 4 & \delta(5, b) = 5 \end{array}$$

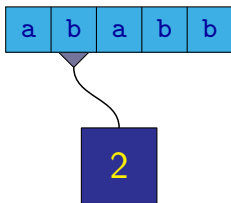
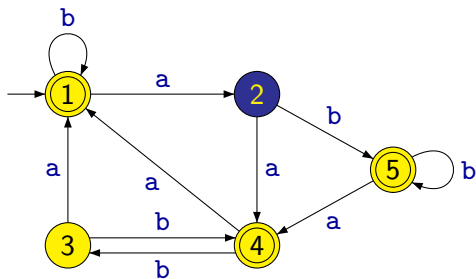
budeme raději používat stručnější tabulku nebo grafické znázornění:

$\delta$	a	b
$\leftrightarrow 1$	2	1
2	4	5
3	1	4
$\leftarrow 4$	1	3
$\leftarrow 5$	4	5

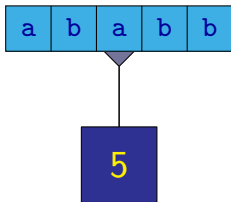
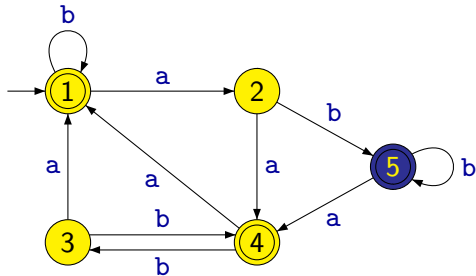
# Deterministický konečný automat



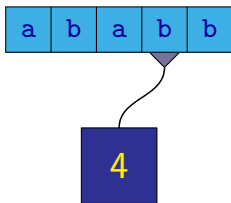
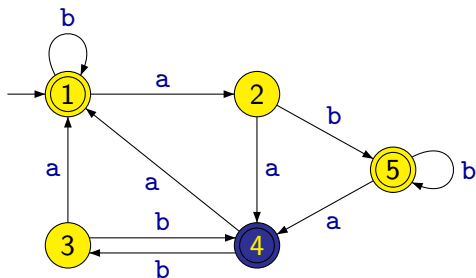
# Deterministický konečný automat



# Deterministický konečný automat

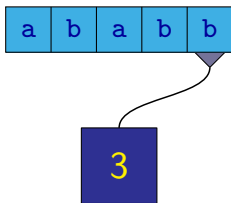
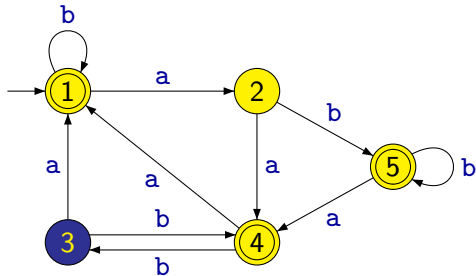


# Deterministický konečný automat

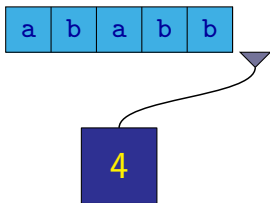
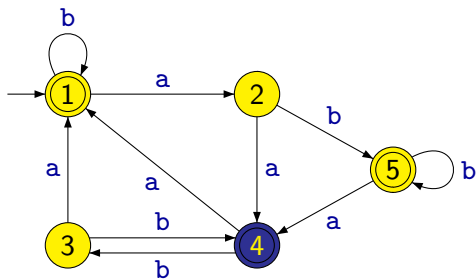




# Deterministický konečný automat



# Deterministický konečný automat



# Deterministický konečný automat

**Konfigurace** konečného automatu je dána stavem jeho řídicí jednotky a dosud nepřechteným obsahem pásky.

Formálně můžeme konfiguraci definovat jako dvojici z množiny  $Q \times \Sigma^*$ .

**Příklad:**  $(2, babb)$  je konfigurace

Na množině všech konfigurací můžeme definovat binární relaci  $\vdash$  s následujícím významem:  $C_1 \vdash C_2$  znamená, že automat může přejít jedním krokem z konfigurace  $C_1$  do konfigurace  $C_2$ .

**Příklad:**

$$(2, babb) \vdash (5, abb)$$

Formálně platí, že  $(q, w) \vdash (q', w')$  právě když  $w = aw'$  a  $q' = \delta(q, a)$  pro nějaké  $a \in \Sigma$ .

# Deterministický konečný automat

Konfigurace  $(q, w)$  se nazývá **počáteční konfigurace**, jestliže  $q = q_0$ .

**Příklad:**  $(1, ababb)$  je počáteční konfigurace.

Konfigurace  $(q, w)$  se nazývá **koncová konfigurace**, jestliže  $w = \varepsilon$ .

**Příklad:**  $(4, \varepsilon)$  je koncová konfigurace.

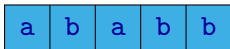
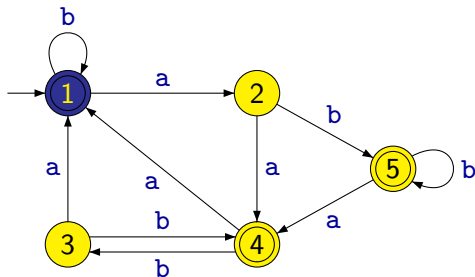
## Definice

**Výpočet** automatu je posloupnost konfigurací

$$C_0, C_1, C_2, \dots, C_k$$

kde  $C_i$  jsou konfigurace,  $C_0$  je počáteční konfigurace,  $C_k$  je koncová konfigurace a pro všechna  $i \in \{1, 2, \dots, k\}$  platí, že  $C_{i-1} \vdash C_i$ .

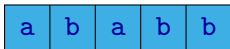
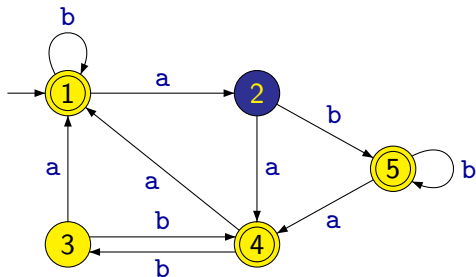
# Deterministický konečný automat



(1, ababb)

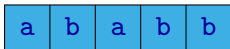
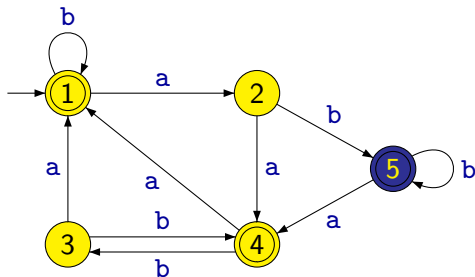


# Deterministický konečný automat



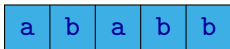
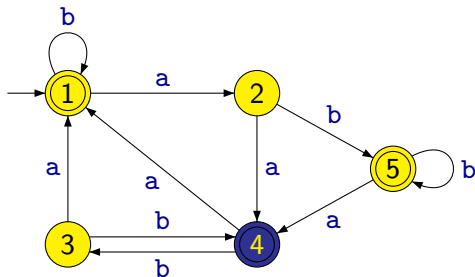
$(1, ababb) \vdash (2, babb)$

# Deterministický konečný automat



$(1, ababb) \vdash (2, babb) \vdash$   
 $(5, abb)$

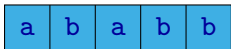
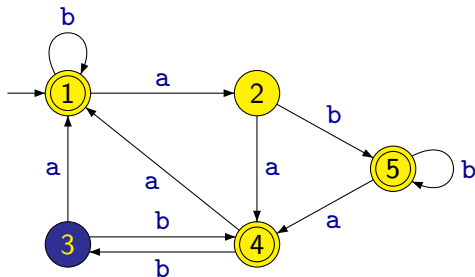
# Deterministický konečný automat



$(1, ababb) \vdash (2, babb) \vdash$   
 $(5, abb) \vdash (4, bb)$

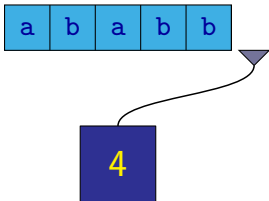
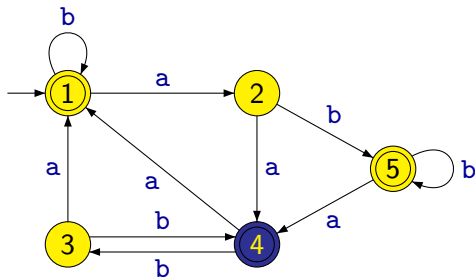


# Deterministický konečný automat



$(1, ababb) \vdash (2, babb) \vdash$   
 $(5, abb) \vdash (4, bb) \vdash$   
 $(3, b)$

# Deterministický konečný automat



$(1, ababb) \vdash (2, babb) \vdash$   
 $(5, abb) \vdash (4, bb) \vdash$   
 $(3, b) \vdash (4, \varepsilon)$

Dále můžeme definovat relaci  $\vdash^*$ , jejíž význam je takový, že  $C \vdash^* C'$  platí právě tehdy, když automat může přejít nějakým libovolným (i nulovým) počtem kroků z konfigurace  $C$  do konfigurace  $C'$ .

Přesněji řečeno,  $C \vdash^* C'$  platí právě tehdy, když existuje posloupnost konfigurací

$$C_0 \vdash C_1 \vdash C_2 \vdash \dots \vdash C_k$$

kde  $C_0 = C$ ,  $C_k = C'$  a pro všechna  $i \in \{1, 2, \dots, k\}$  platí, že  $C_{i-1} \vdash C_i$ .

## Definice

Koncová konfigurace  $(q, \varepsilon)$  je **přijímající**, jestliže  $q \in F$ .

## Definice

Automat **přijímá** slovo  $w \in \Sigma^*$  právě tehdy, jestliže výpočet začínající v počáteční konfiguraci  $(q_0, w)$  skončí v přijímající koncové konfiguraci.

**Poznámka:** Formálně to můžeme definovat tak, že automat přijímá slovo  $w \in \Sigma^*$  právě když  $(q_0, w) \vdash^* (q, \varepsilon)$  pro nějaké  $q \in F$ .

## Definice

**Jazyk** rozpoznávaný (přijímaný) daným deterministickým konečným automatem  $A = (Q, \Sigma, \delta, q_0, F)$ , označovaný  $L(A)$ , je množina všech slov přijímaných tímto automatem, tj.

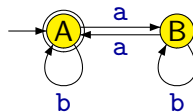
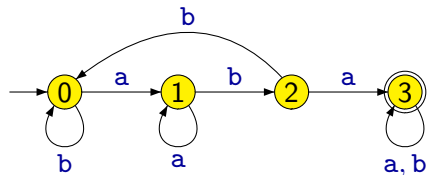
$$L(A) = \{w \in \Sigma^* \mid (q_0, w) \vdash^* (q, \varepsilon), q \in F\}$$

## Definice

Jazyk  $L$  nazýváme **regulární** právě tehdy, když existuje konečný automat, který jej přijímá.

# Automat pro průnik jazyků

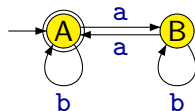
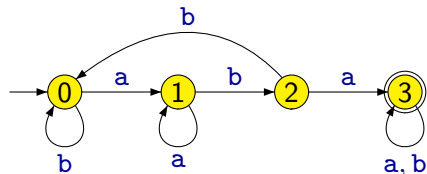
Máme následující dva automaty:



Přijmou oba slovo `ababb`?

# Automat pro průnik jazyků

Máme následující dva automaty:

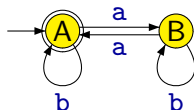
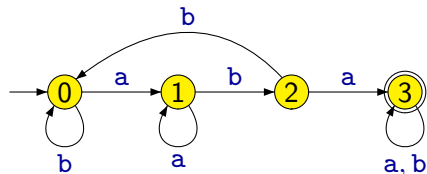


Přijmou oba slovo **ababb**?

Můžeme postupně nechat přečíst slovo oběma automatům. Odpověď bude ano, pokud oba odpoví ano.

# Automat pro průnik jazyků

Máme následující dva automaty:



Přijmou oba slovo `ababb`?

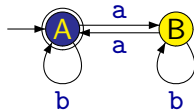
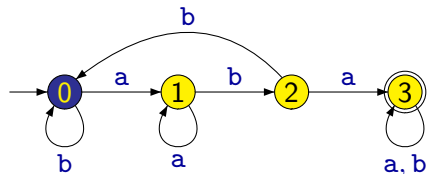
Můžeme postupně nechat přečíst slovo oběma automatům. Odpověď bude ano, pokud oba odpoví ano.

Lepší je číst slovo současně oběma automaty.



# Automat pro průnik jazyků

Máme následující dva automaty:



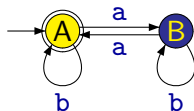
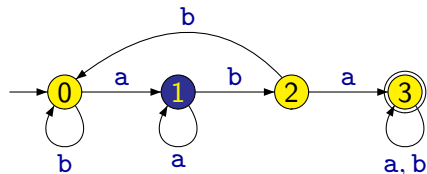
Přijmou oba slovo **a**babbb?

Můžeme postupně nechat přečíst slovo oběma automatům. Odpověď bude ano, pokud oba odpoví ano.

Lepší je číst slovo současně oběma automaty.

# Automat pro průnik jazyků

Máme následující dva automaty:



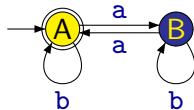
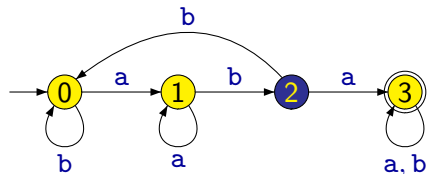
Přijmou oba slovo **a**bab**b**?

Můžeme postupně nechat přechít slovo oběma automatům. Odpověď bude ano, pokud oba odpoví ano.

Lepší je číst slovo současně oběma automaty.

# Automat pro průnik jazyků

Máme následující dva automaty:



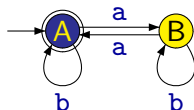
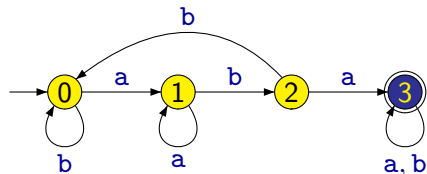
Přijmou oba slovo **ababb**?

Můžeme postupně nechat přečíst slovo oběma automatům. Odpověď bude ano, pokud oba odpoví ano.

Lepší je číst slovo současně oběma automaty.

# Automat pro průnik jazyků

Máme následující dva automaty:



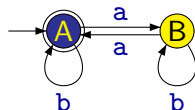
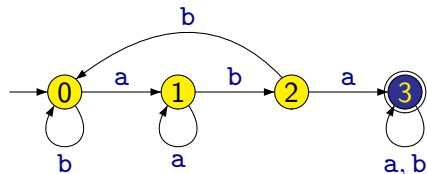
Přijmou oba slovo **ababb**?

Můžeme postupně nechat přečíst slovo oběma automatům. Odpověď bude ano, pokud oba odpoví ano.

Lepší je číst slovo současně oběma automaty.

# Automat pro průnik jazyků

Máme následující dva automaty:



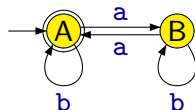
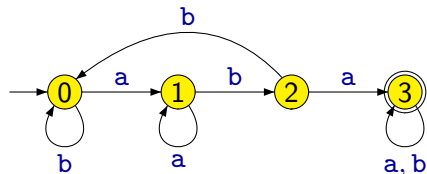
Přijmou oba slovo **ababb**?

Můžeme postupně nechat přečíst slovo oběma automatům. Odpověď bude ano, pokud oba odpoví ano.

Lepší je číst slovo současně oběma automaty.

# Automat pro průnik jazyků

Máme následující dva automaty:



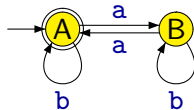
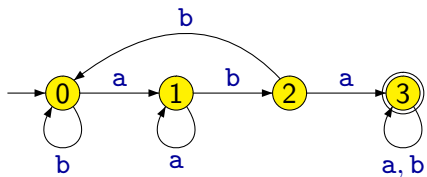
Přijmou oba slovo **ababb**?

Můžeme postupně nechat přečíst slovo oběma automatům. Odpověď bude ano, pokud oba odpoví ano.

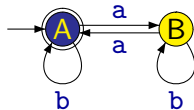
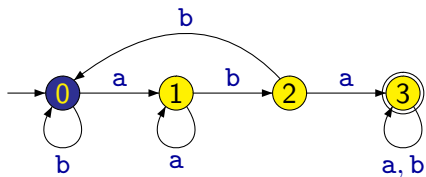
Lepší je číst slovo současně oběma automaty.

Situace při tomto postupu je dána nepřečtenou částí slova a aktuálními stavy obou automatů. Zkusíme vytvořit automat, který toto má jako své konfigurace.

# Automat pro průnik jazyků

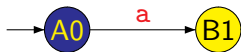
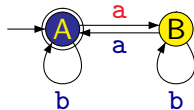
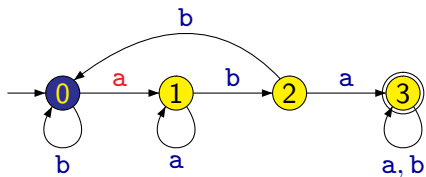


# Automat pro průnik jazyků

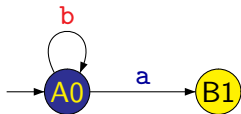
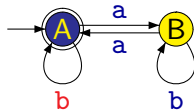
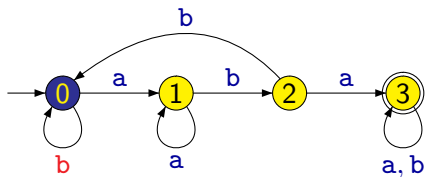




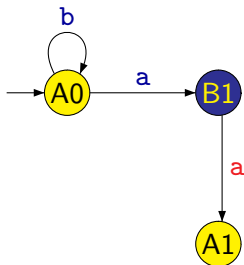
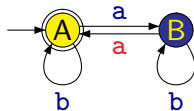
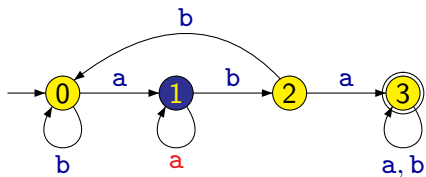
# Automat pro průnik jazyků



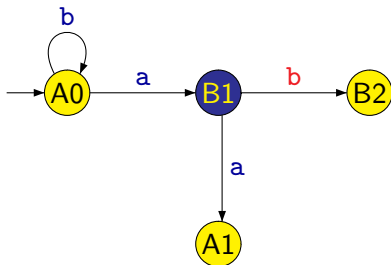
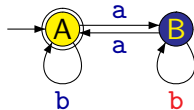
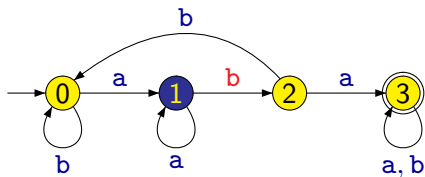
# Automat pro průnik jazyků



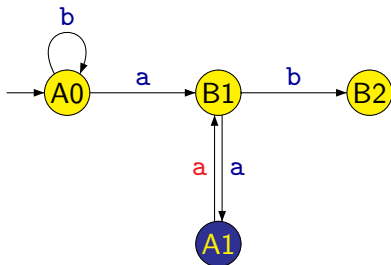
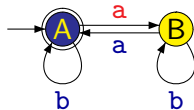
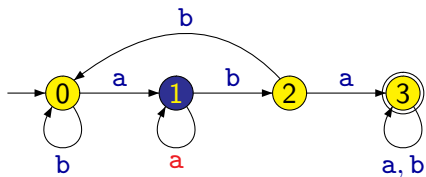
# Automat pro průnik jazyků



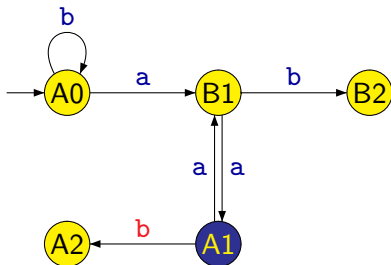
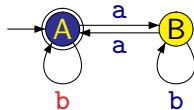
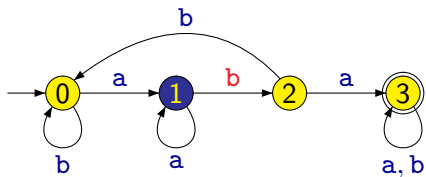
# Automat pro průnik jazyků



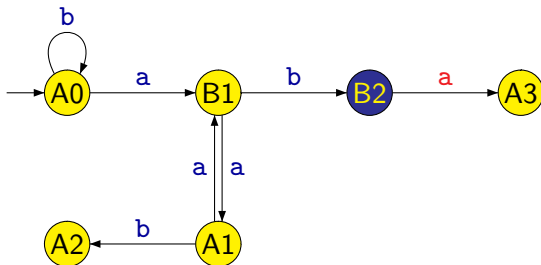
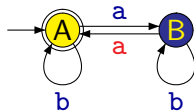
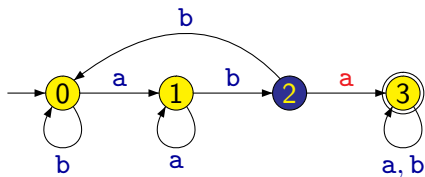
# Automat pro průnik jazyků



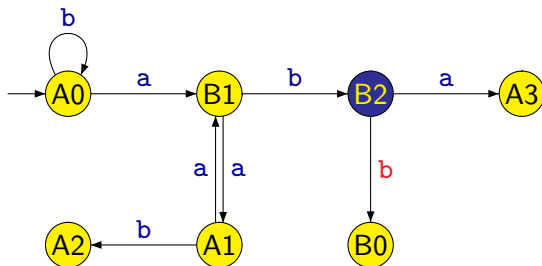
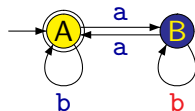
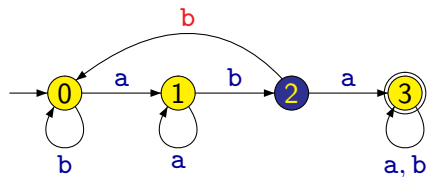
# Automat pro průnik jazyků



# Automat pro průnik jazyků

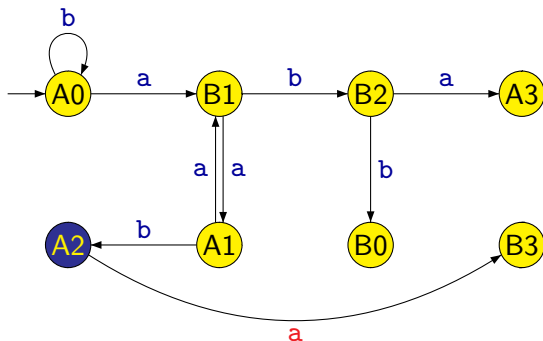
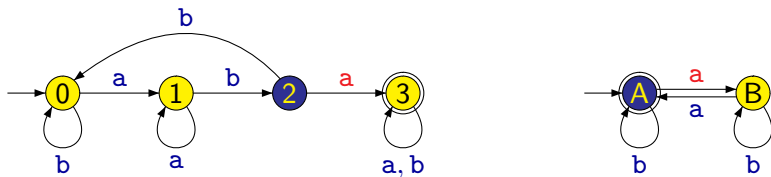


# Automat pro průnik jazyků

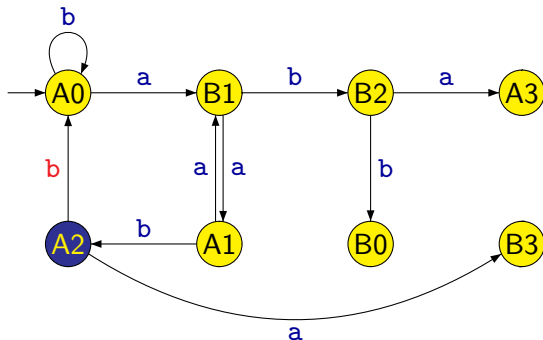
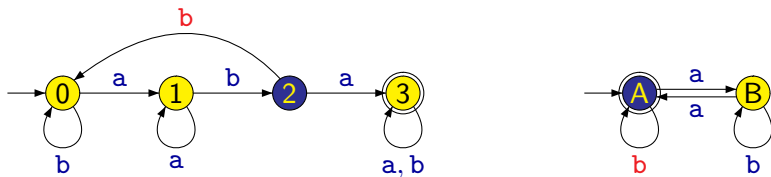




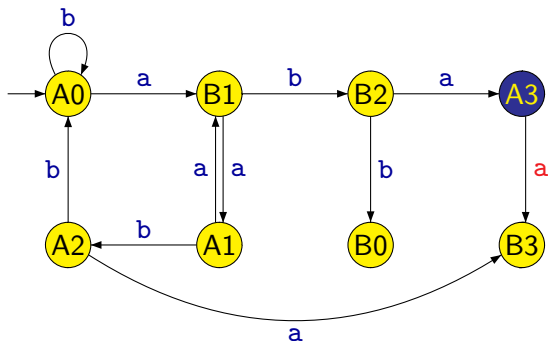
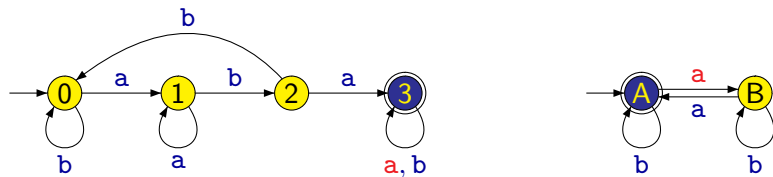
# Automat pro průnik jazyků



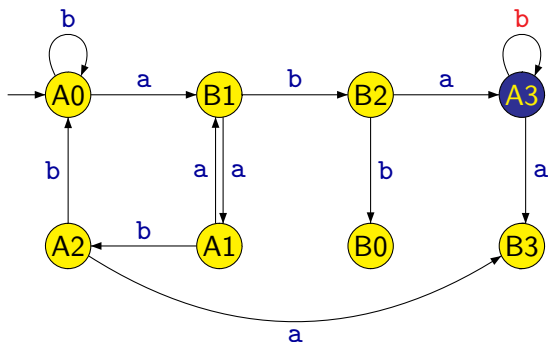
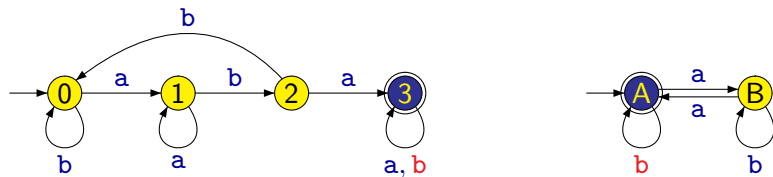
# Automat pro průnik jazyků



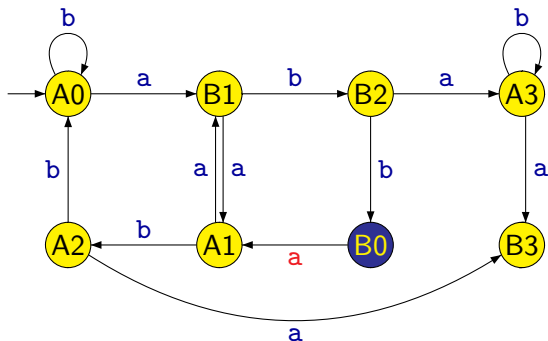
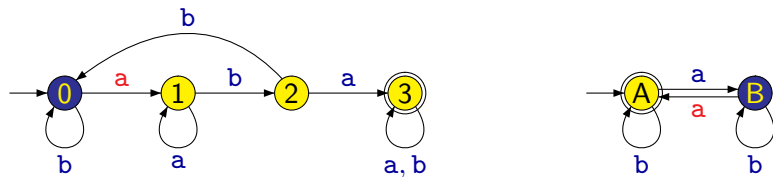
# Automat pro průnik jazyků



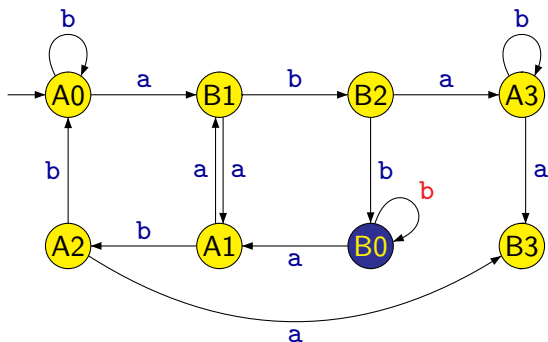
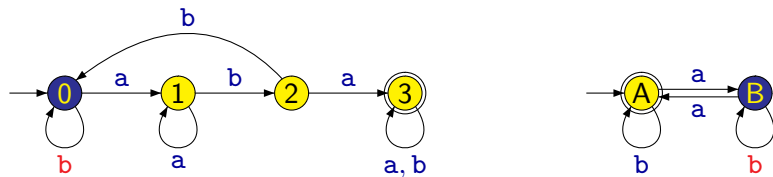
# Automat pro průnik jazyků



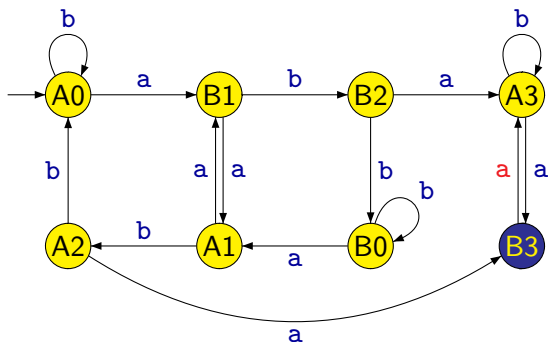
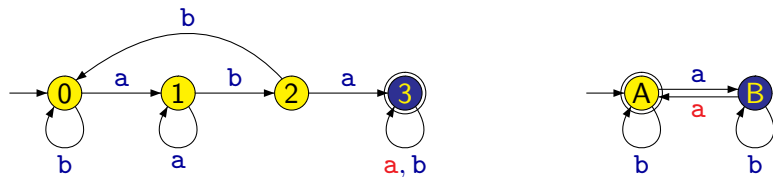
# Automat pro průnik jazyků



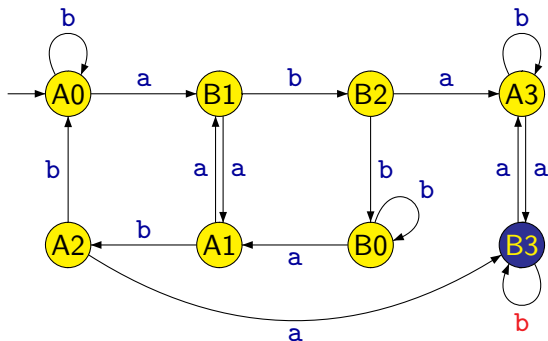
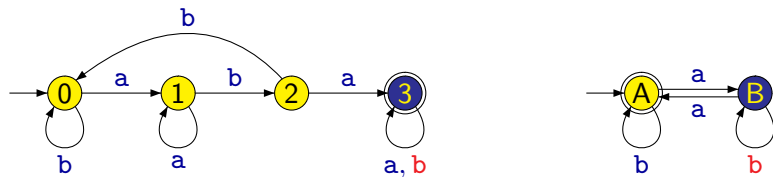
# Automat pro průnik jazyků



# Automat pro průnik jazyků

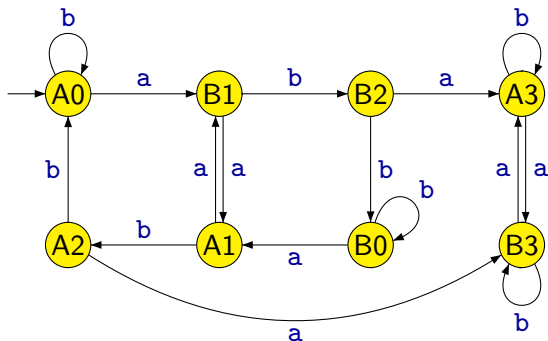
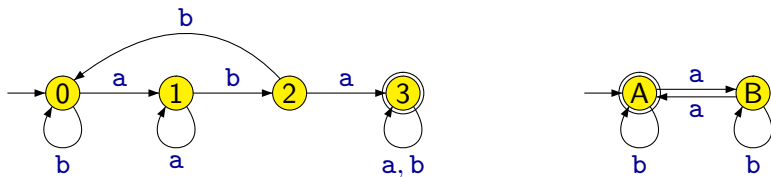


# Automat pro průnik jazyků

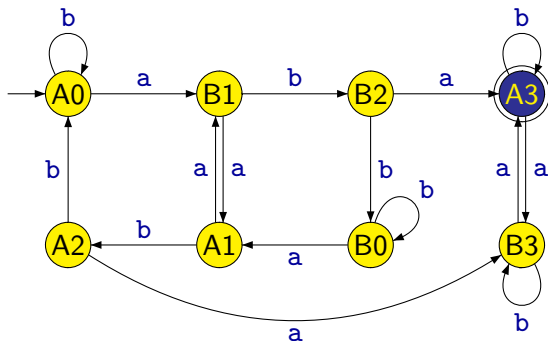
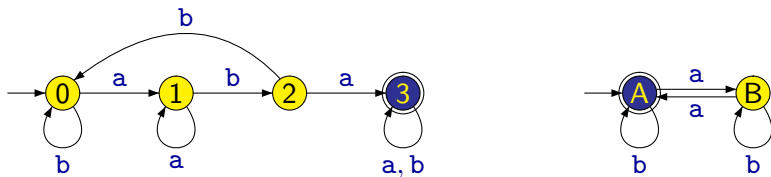




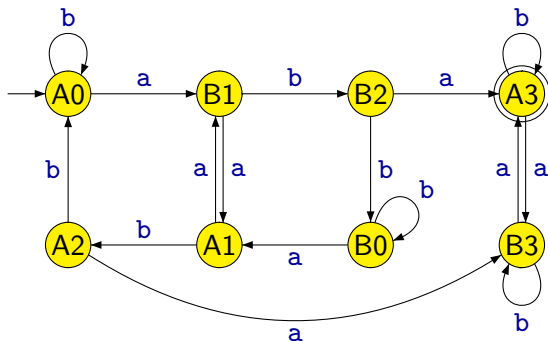
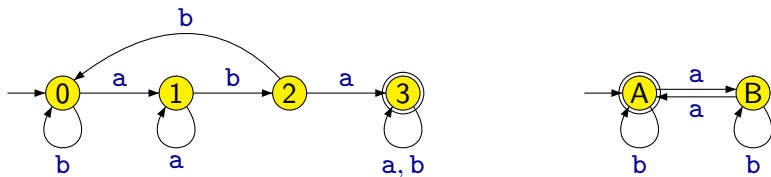
# Automat pro průnik jazyků



# Automat pro průnik jazyků



# Automat pro průnik jazyků



## Věta

Jestliže jazyky  $L_1, L_2 \subseteq \Sigma^*$  jsou regulární, pak také jazyk  $L_1 \cap L_2$  je regulární.

## Věta

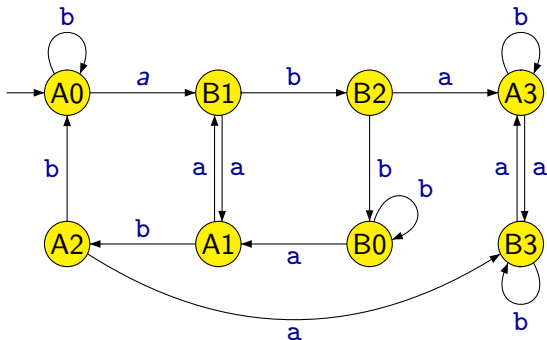
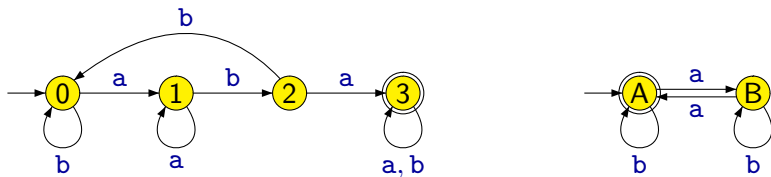
Jestliže jazyky  $L_1, L_2 \subseteq \Sigma^*$  jsou regulární, pak také jazyk  $L_1 \cap L_2$  je regulární.

**Důkaz:** Necht'  $L_1 = L(\mathcal{A}_1)$ ,  $L_2 = L(\mathcal{A}_2)$  pro konečné automaty  $\mathcal{A}_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1)$ ,  $\mathcal{A}_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$ . Definujeme automat  $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$  tž.

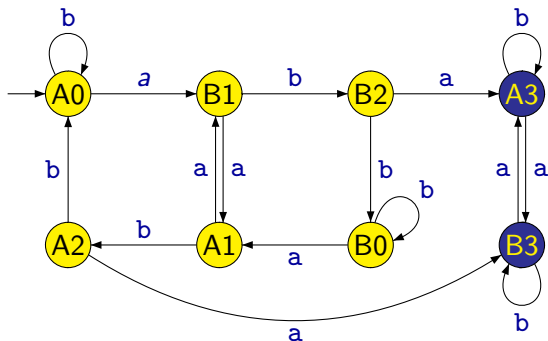
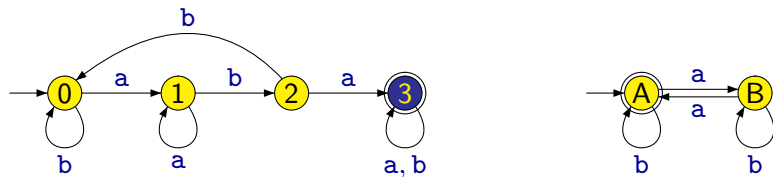
- $Q = Q_1 \times Q_2$ ,
- $\delta((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$  pro všechna  $q_1 \in Q_1$ ,  $q_2 \in Q_2$ ,  $a \in \Sigma$ ,
- $q_0 = (q_{01}, q_{02})$ ,
- $F = (F_1 \times F_2)$ .

Od konstrukce pro sjednocení se tato liší jen množinou koncových stavů v sestrojeném automatu.

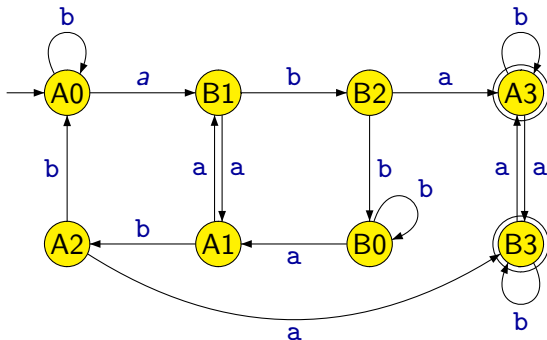
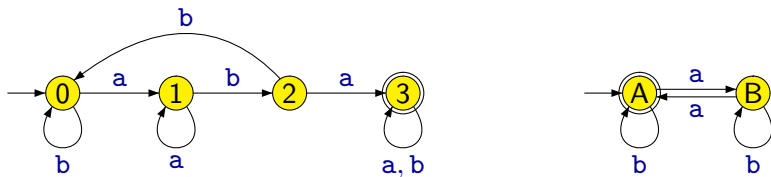
# Automat pro sjednocení jazyků



# Automat pro sjednocení jazyků

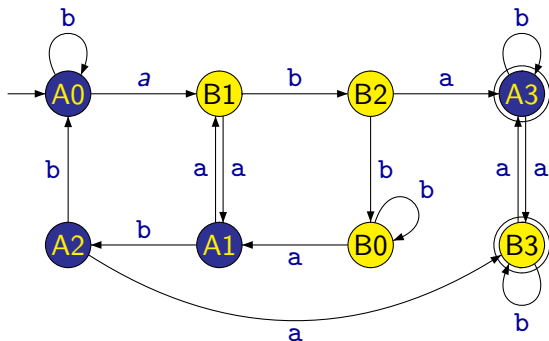
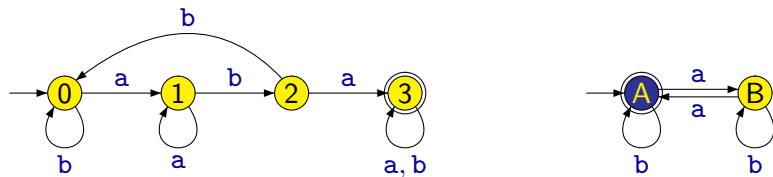


# Automat pro sjednocení jazyků

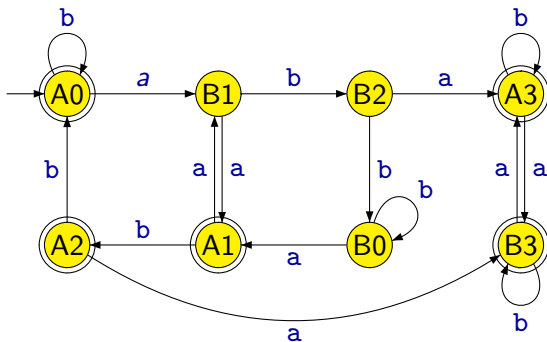
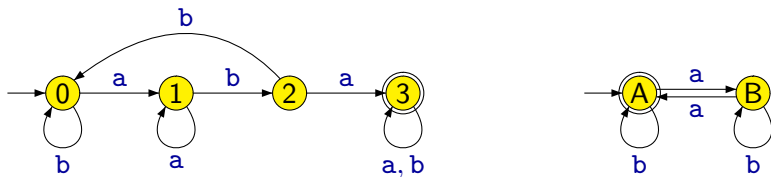




# Automat pro sjednocení jazyků



# Automat pro sjednocení jazyků



## Věta

Jestliže jazyky  $L_1, L_2 \subseteq \Sigma^*$  jsou regulární, pak také jazyk  $L_1 \cup L_2$  je regulární.

## Věta

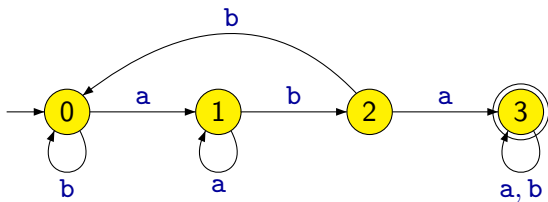
Jestliže jazyky  $L_1, L_2 \subseteq \Sigma^*$  jsou regulární, pak také jazyk  $L_1 \cup L_2$  je regulární.

**Důkaz:** Nechť  $L_1 = L(\mathcal{A}_1)$ ,  $L_2 = L(\mathcal{A}_2)$  pro konečné automaty  $\mathcal{A}_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1)$ ,  $\mathcal{A}_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$ . Definujeme automat  $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$  tž.

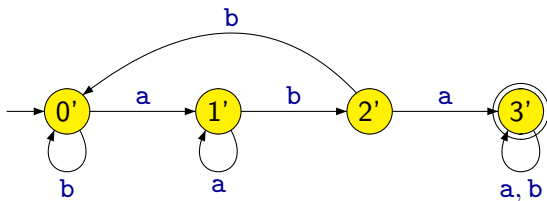
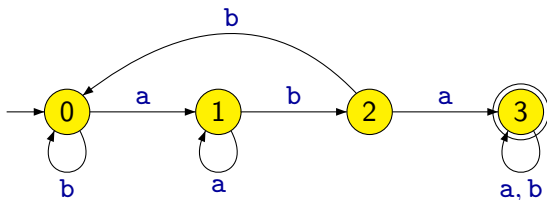
- $Q = Q_1 \times Q_2$ ,
- $\delta((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$  pro všechna  $q_1 \in Q_1$ ,  $q_2 \in Q_2$ ,  $a \in \Sigma$ ,
- $q_0 = (q_{01}, q_{02})$ ,
- $F = (F_1 \times Q_2) \cup (Q_1 \times F_2)$ .

Je zřejmé a např. indukcí podle délky  $|w|$  je možno ukázat, že pro libovolné  $q_1 \in Q_1$ ,  $q_2 \in Q_2$  a  $w \in \Sigma^*$  je  $\delta^*((q_1, q_2), w) = (\delta_1^*(q_1, w), \delta_2^*(q_2, w))$ .

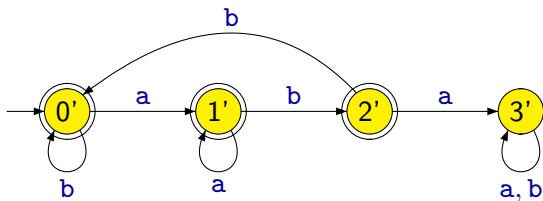
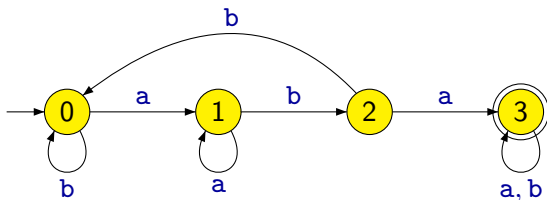
# Automat pro doplněk jazyka



# Automat pro doplněk jazyka



# Automat pro doplněk jazyka



## Věta

Jestliže jazyk  $L$  je regulární, pak také jeho doplňěk  $\bar{L}$  je regulární.



## Věta

Jestliže jazyk  $L$  je regulární, pak také jeho doplňěk  $\bar{L}$  je regulární.

**Důkaz:** Necht'  $L = L(\mathcal{A})$  pro konečný automat  $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ . Definujeme automat  $\mathcal{A}' = (Q, \Sigma, \delta, q_0, Q - F)$ . Potom

- pro každé slovo  $w$  přijímané  $\mathcal{A}$  platí  $\delta^*(q_0, w) \in F$  a tedy  $\delta^*(q_0, w) \notin Q - F$
- pro každé slovo  $w$  nepřijímané  $\mathcal{A}$  platí  $\delta^*(q_0, w) \notin F$  a tedy  $\delta^*(q_0, w) \in Q - F$
- a tedy automat  $\mathcal{A}'$  přijímá právě ta slova, která nepřijímá  $\mathcal{A}$

# Zřetězení regulárních jazyků

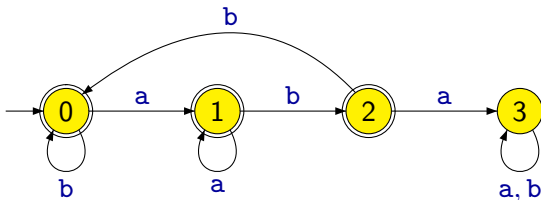
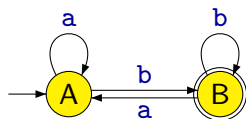
Je možné ke dvěma automatům  $\mathcal{A}_\infty, \mathcal{A}_\epsilon$  přijímajícím jazyky  $L_1, L_2$  sestrojít automat  $\mathcal{A}$  přijímající jazyk  $L_1.L_2$ ?

**Příklad:** Mějme jazyky  $L_1 = \{wb \mid w \in \{a, b\}^*\}$  a  $L_2 = \{w \in \{a, b\}^* \mid w \text{ neobsahuje } aba\}$ .

# Zřetězení regulárních jazyků

Je možné ke dvěma automatům  $\mathcal{A}_\infty, \mathcal{A}_\infty$  přijímajícím jazyky  $L_1, L_2$  sestrojit automat  $\mathcal{A}$  přijímající jazyk  $L_1.L_2$ ?

**Příklad:** Mějme jazyky  $L_1 = \{wb \mid w \in \{a, b\}^*\}$  a  $L_2 = \{w \in \{a, b\}^* \mid w \text{ neobsahuje } aba\}$ .

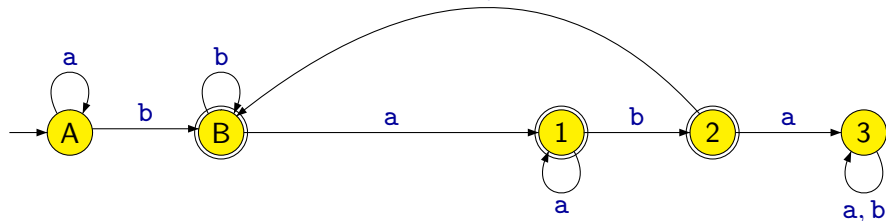


Jak tyto automaty spojit?

# Zřetězení regulárních jazyků

Je možné ke dvěma automatům  $\mathcal{A}_\infty, \mathcal{A}_\epsilon$  přijímajícím jazyky  $L_1, L_2$  sestavit automat  $\mathcal{A}$  přijímající jazyk  $L_1.L_2$ ?

**Příklad:** Mějme jazyky  $L_1 = \{wb \mid w \in \{a, b\}^*\}$  a  $L_2 = \{w \in \{a, b\}^* \mid w \text{ neobsahuje } aba\}_b$



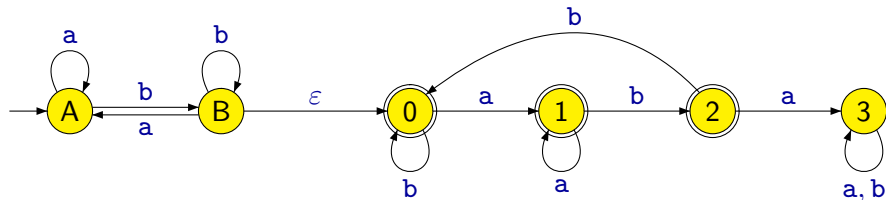
Ve chvíli, kdy přijímá první automat, zkusí začít rozpoznávat druhým automatem.

Správně přijme  $abaaa$ , kde  $ab \in L_1$  a  $aaa \in L_2$ . Ale nepřijme  $ababa$ .

# Zřetězení regulárních jazyků

Je možné ke dvěma automatům  $\mathcal{A}_\infty, \mathcal{A}_\epsilon$  přijímajícím jazyky  $L_1, L_2$  sestrojít automat  $\mathcal{A}$  přijímající jazyk  $L_1.L_2$ ?

**Příklad:** Mějme jazyky  $L_1 = \{wb \mid w \in \{a, b\}^*\}$  a  $L_2 = \{w \in \{a, b\}^* \mid w \text{ neobsahuje } aba\}$ .



Ve chvíli, kdy přijímá první automat, zkusí začít rozpoznávat druhým automatem a současně i zkusí pokračovat v prvním.

Jde o tzv. zobecněný nedeterministický automat rozpoznávající požadovaný jazyk

## Definice

**Nedeterministický konečný automat** je uspořádaná pětice

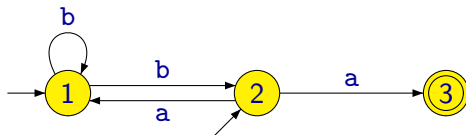
$\mathcal{A} = (Q, \Sigma, \delta, I, F)$ , kde

- $Q$  je konečná neprázdná množina stavů
- $\Sigma$  je konečná neprázdná množina zvaná vstupní abeceda
- $\delta : Q \times \Sigma \rightarrow 2^Q$  je (nedeterministická) přechodová funkce
- $I \subseteq Q$  je neprázdná množina počátečních stavů
- $F \subseteq Q$  je množina přijímajících (koncových) stavů

Rozdíly od deterministického konečného automatu:

- Z daného stavu je možno čtením vstupního symbolu přejít do více různých stavů, ale také přechod pro vstupní symbol nemusí existovat
- Je povolen více než jeden počáteční stav

# Nedeterministický konečný automat



- $Q = \{1, 2, 3\}$
- $\Sigma = \{a, b\}$
- $I = \{1, 2\}$
- $F = \{3\}$

$$\delta(1, a) = \emptyset$$

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

$$\delta(3, a) = \emptyset$$

$$\delta(1, b) = \{1, 2\}$$

$$\delta(2, b) = \emptyset$$

$$\delta(3, b) = \emptyset$$

# Nedeterministický konečný automat

Místo zápisu

$$\delta(1, a) = \emptyset$$

$$\delta(1, b) = \{1, 2\}$$

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

$$\delta(2, b) = \emptyset$$

$$\delta(3, a) = \emptyset$$

$$\delta(3, b) = \emptyset$$

budeme raději používat stručnější tabulku nebo grafické znázornění:

$\delta$	a	b
1		1, 2
2	1, 3	
3		



# Nedeterministický konečný automat

**Konfigurace** nedeterministického konečného automatu je, stejně jako v případě deterministického konečného automatu, dána stavem jeho řídicí jednotky a dosud nepřečteným obsahem pásky.

Formálně můžeme konfiguraci definovat jako dvojici z množiny  $Q \times \Sigma^*$ .

Opět můžeme na množině všech konfigurací definovat binární relaci  $\vdash$  se stejným významem:  $C_1 \vdash C_2$  znamená, že automat může přejít jedním krokem z konfigurace  $C_1$  do konfigurace  $C_2$ . Liší se ovšem formální definice této relace.

Formálně platí, že  $(q, w) \vdash (q', w')$  právě když  $w = aw'$  a  $q' \in \delta(q, a)$  pro nějaké  $a \in \Sigma$ .

# Nedeterministický konečný automat

Konfigurace  $(q, w)$  se nazývá **počáteční konfigurace**, jestliže  $q \in I$ .

**Poznámka:** Pro stejné slovo tedy může existovat více různých počátečních konfigurací

Konfigurace  $(q, w)$  se nazývá **koncová konfigurace**, jestliže  $w = \varepsilon$ .

## Definice

**Výpočet** automatu je posloupnost konfigurací

$$C_0, C_1, C_2, \dots, C_k$$

kde  $C_i$  jsou konfigurace,  $C_0$  je počáteční konfigurace a pro všechna  $i \in \{1, 2, \dots, k\}$  platí, že  $C_{i-1} \vdash C_i$ .

Výpočet je **úplný**, je-li  $C_k$  koncová konfigurace.

Opět můžeme definovat relaci  $\vdash^*$ , jejíž význam je takový, že  $C \vdash^* C'$  platí právě tehdy, když automat může přejít nějakým libovolným (i nulovým) počtem kroků z konfigurace  $C$  do konfigurace  $C'$ .

Přesněji řečeno,  $C \vdash^* C'$  platí právě tehdy, když existuje posloupnost konfigurací

$$C_0 \vdash C_1 \vdash C_2 \vdash \dots \vdash C_k$$

kde  $C_0 = C$ ,  $C_k = C'$  a pro všechna  $i \in \{1, 2, \dots, k\}$  platí, že  $C_{i-1} \vdash C_i$ .

## Definice

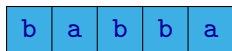
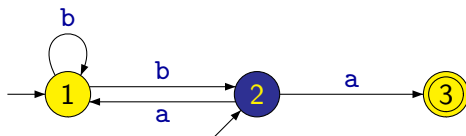
Koncová konfigurace  $(q, \varepsilon)$  je **přijímající**, jestliže  $q \in F$ .

## Definice

Automat **přijímá** slovo  $w \in \Sigma^*$  právě tehdy, jestliže existuje nějaký (úplný) výpočet začínající v počáteční konfiguraci  $(q, w)$  pro nějaké  $q \in I$  a končící v přijímající koncové konfiguraci.

**Poznámka:** Formálně to můžeme definovat tak, že automat přijímá slovo  $w \in \Sigma^*$  právě když  $(q_0, w) \vdash^* (q, \varepsilon)$  pro nějaké  $q \in F$  a  $q_0 \in I$ .

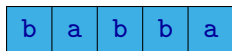
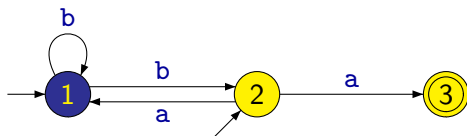
# Nedeterministický konečný automat



(2, babba)

Neúplný výpočet

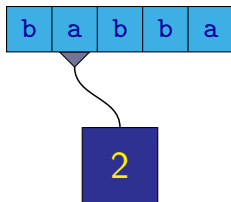
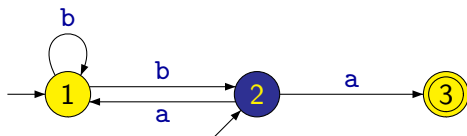
# Nedeterministický konečný automat



(1, babba)

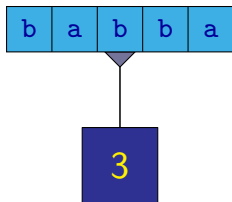
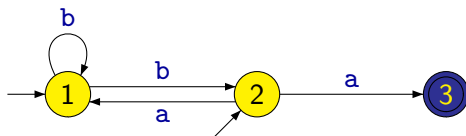


# Nedeterministický konečný automat



$(1, babba) \vdash (2, abba)$

# Nedeterministický konečný automat

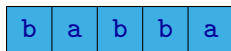
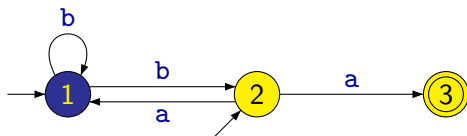


$(1, babba) \vdash (2, abba) \vdash$   
 $(3, bba)$

Neúplný výpočet



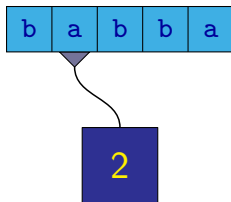
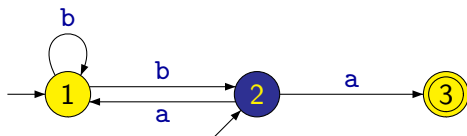
# Nedeterministický konečný automat



(1, babba)

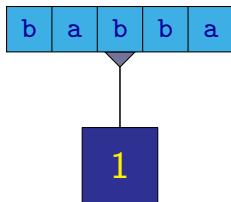
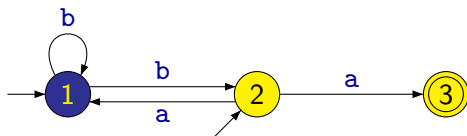


# Nedeterministický konečný automat



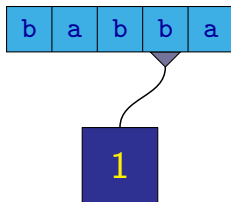
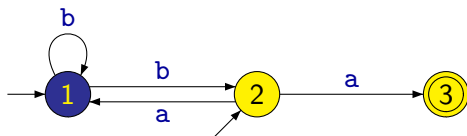
$(1, babba) \vdash (2, abba)$

# Nedeterministický konečný automat



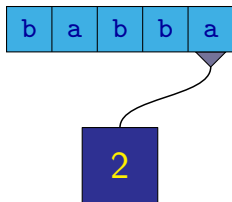
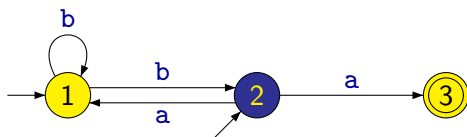
$(1, babba) \vdash (2, abba) \vdash$   
 $(1, bba)$

# Nedeterministický konečný automat



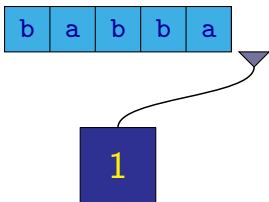
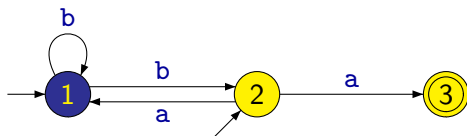
$(1, babba) \vdash (2, abba) \vdash$   
 $(1, bba) \vdash (1, ba)$

# Nedeterministický konečný automat



$(1, babba) \vdash (2, abba) \vdash$   
 $(1, bba) \vdash (1, ba) \vdash$   
 $(2, a)$

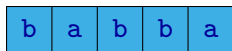
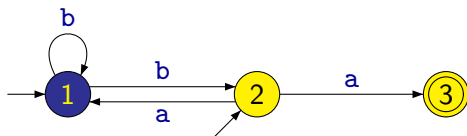
# Nedeterministický konečný automat



$(1, babba) \vdash (2, abba) \vdash$   
 $(1, bba) \vdash (1, ba) \vdash$   
 $(2, a) \vdash (1, \varepsilon)$

Úplný, ale nepřijímající výpočet

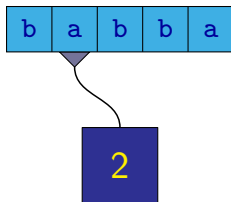
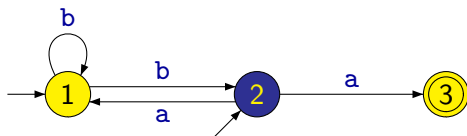
# Nedeterministický konečný automat



(1, babba)



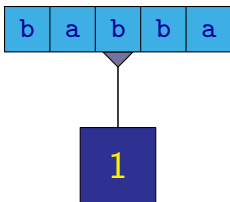
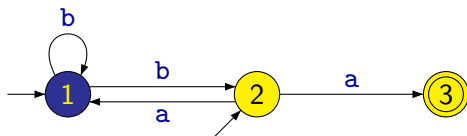
# Nedeterministický konečný automat



$(1, babba) \vdash (2, abba)$

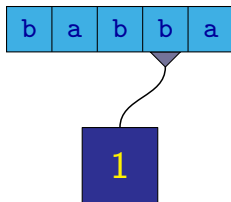
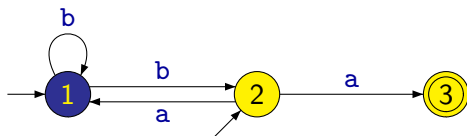


# Nedeterministický konečný automat



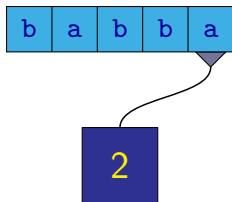
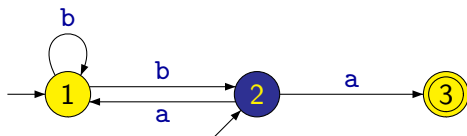
$(1, babba) \vdash (2, abba) \vdash$   
 $(1, bba)$

# Nedeterministický konečný automat



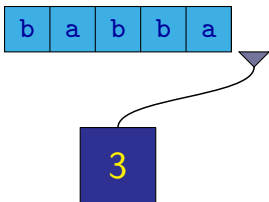
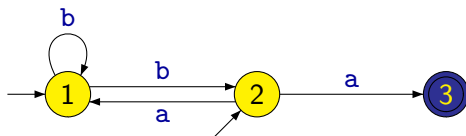
$(1, babba) \vdash (2, abba) \vdash$   
 $(1, bba) \vdash (1, ba)$

# Nedeterministický konečný automat



$(1, babba) \vdash (2, abba) \vdash$   
 $(1, bba) \vdash (1, ba) \vdash$   
 $(2, a)$

# Nedeterministický konečný automat



$(1, babba) \vdash (2, abba) \vdash$   
 $(1, bba) \vdash (1, ba) \vdash$   
 $(2, a) \vdash (3, \varepsilon)$

Úplný přijímající výpočet

# Zobecněný nedeterministický konečný automat

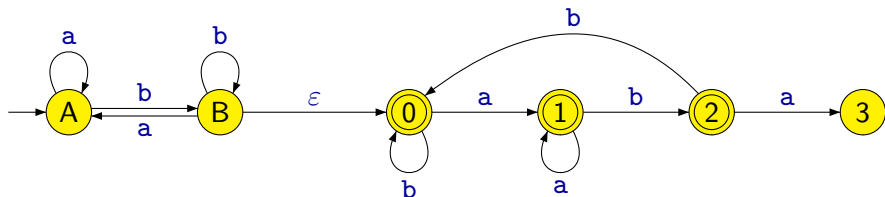
- Někdy je užitečná možnost nechat automat změnit stav bez čtení vstupního symbolu (např. motivační příklad na zřetězení)
- Přidáme tzv.  **$\varepsilon$ -přechody** - mohou být provedeny kdykoliv se automat nachází ve stavu, ze kterého vycházejí a neposouvá se čtecí hlava na další symbol

## Definice

**(Zobecněný) nedeterministický konečný automat** je uspořádaná pětice  $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ , kde

- $Q$  je konečná neprázdná množina stavů
- $\Sigma$  je konečná neprázdná množina zvaná vstupní abeceda
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^Q$  je (nedeterministická) přechodová funkce
- $I \subseteq Q$  je neprázdná množina počátečních stavů
- $F \subseteq Q$  je množina přijímajících (koncových) stavů

# Zobecněný nedeterministický konečný automat



- $Q = \{A, B, 1, 2, 3\}$
- $\Sigma = \{a, b\}$
- $I = \{A\}$
- $F = \{0, 1, 2\}$

$$\delta(A, a) = \{A\}$$

$$\delta(A, b) = \{B\}$$

$$\delta(B, a) = \{A\}$$

$$\delta(B, b) = \{B\}$$

$$\delta(B, \varepsilon) = \{0\}$$

$$\delta(0, a) = \{1\}$$

$$\delta(0, b) = \{0\}$$

$$\delta(1, a) = \{1\}$$

$$\delta(1, b) = \{2\}$$

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

$$\delta(2, b) = \{0\}$$

$$\delta(3, a) = \emptyset$$

$$\delta(3, b) = \emptyset$$

# Zobecněný nedeterministický konečný automat

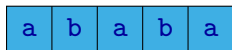
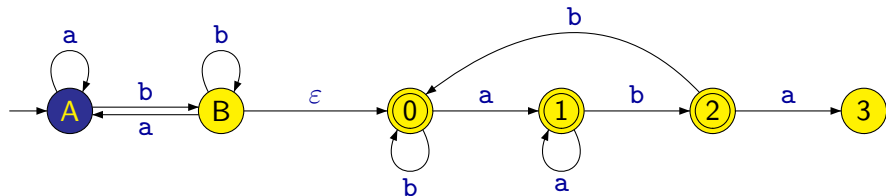
Opět je vhodné zapsat přechodovou funkci tabulkou

$\delta$	a	b	$\varepsilon$
A	A	B	
B	A	B	0
0	1	0	
1	1	2	
2	3	0	
3			

- Konfigurace ZNKA je definována stejně jako pro NKA.
- Relaci  $\vdash$  mezi konfiguracemi definujeme tak, že  $(q, w) \vdash (q', w')$  právě když  $w = aw'$  a  $q' \in \delta(q, a)$  pro nějaké  $a \in \Sigma$  nebo  $w = w'$  a  $q' \in \delta(q, \varepsilon)$ .
- Počáteční a koncová konfigurace, výpočet, přijímání atd. jsou definovány stejně jako pro NKA

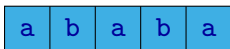
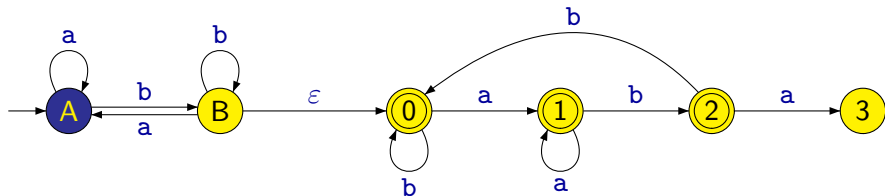


# Zobecněný nedeterministický konečný automat



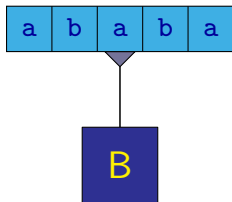
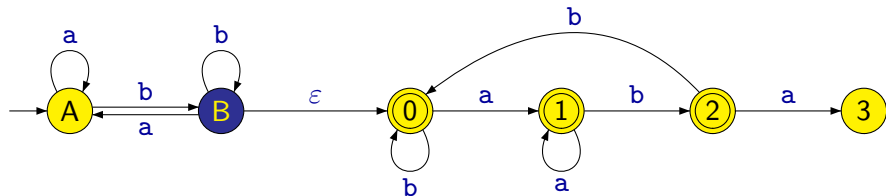
(A, ababa)

# Zobecněný nedeterministický konečný automat



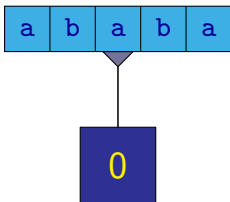
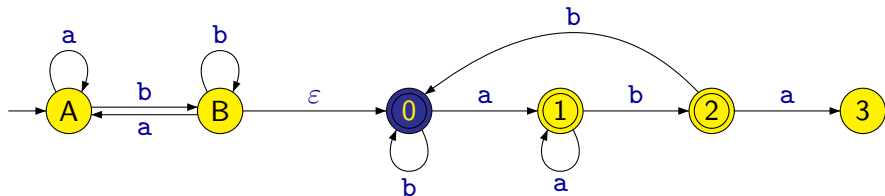
$(A, ababa) \vdash (A, baba)$

# Zobecněný nedeterministický konečný automat



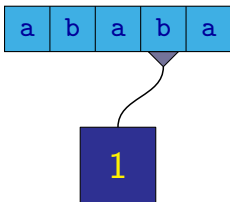
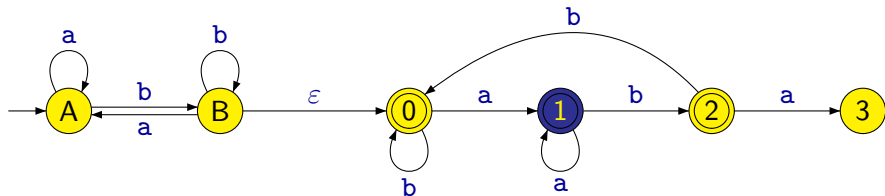
$(A, ababa) \vdash (A, baba) \vdash$   
 $(B, aba)$

# Zobecněný nedeterministický konečný automat



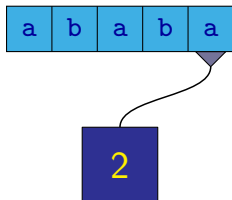
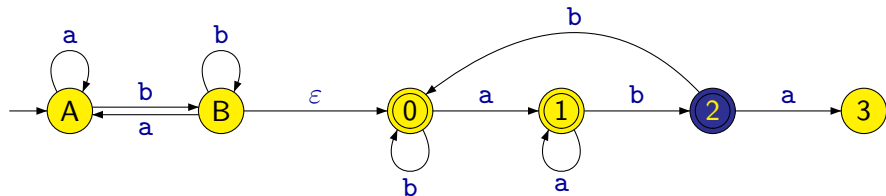
$(A, ababa) \vdash (A, baba) \vdash$   
 $(B, aba) \vdash (0, aba)$

# Zobecněný nedeterministický konečný automat



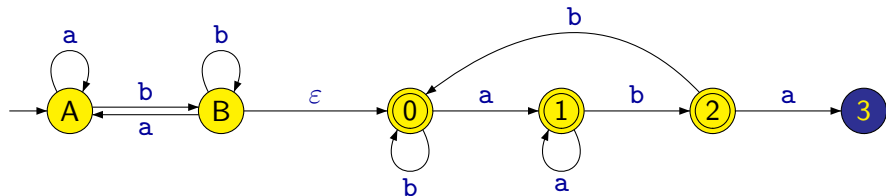
$(A, ababa) \vdash (A, baba) \vdash$   
 $(B, aba) \vdash (0, aba) \vdash$   
 $(1, ba)$

# Zobecněný nedeterministický konečný automat



$(A, ababa) \vdash (A, baba) \vdash$   
 $(B, aba) \vdash (0, aba) \vdash$   
 $(1, ba) \vdash (2, a)$

# Zobecněný nedeterministický konečný automat



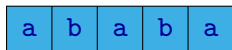
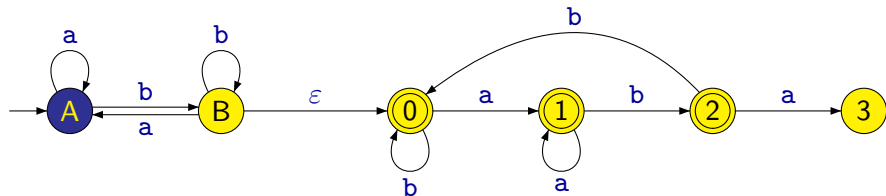
a b a b a

3

$(A, ababa) \vdash (A, baba) \vdash$   
 $(B, aba) \vdash (0, aba) \vdash$   
 $(1, ba) \vdash (2, a) \vdash (3, \epsilon)$

Úplný, ale nepřijímající výpočet

# Zobecněný nedeterministický konečný automat

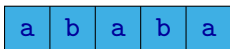
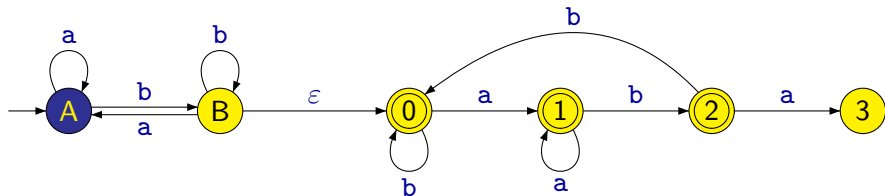


(A, ababa)



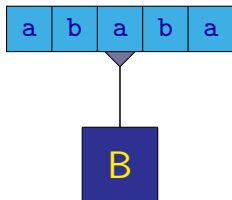
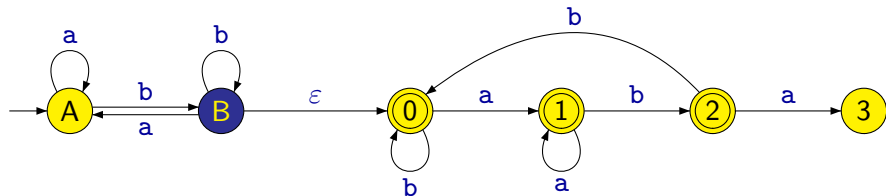


# Zobecněný nedeterministický konečný automat



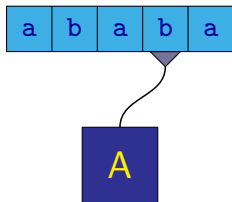
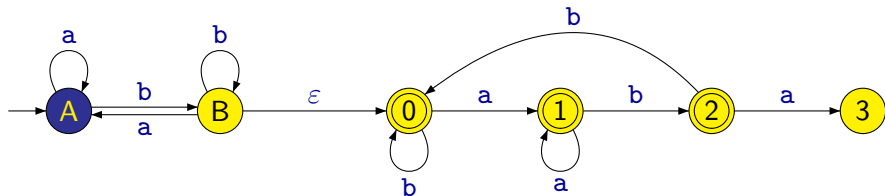
$(A, ababa) \vdash (A, baba)$

# Zobecněný nedeterministický konečný automat



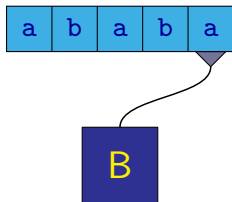
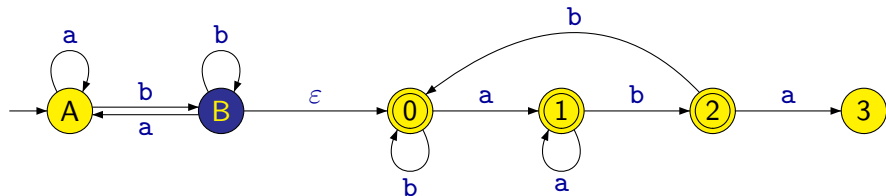
$(A, ababa) \vdash (A, baba) \vdash$   
 $(B, aba)$

# Zobecněný nedeterministický konečný automat



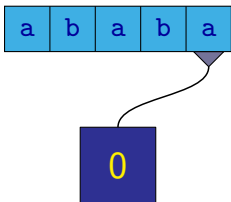
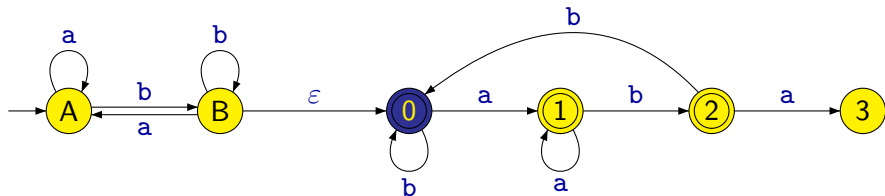
$(A, ababa) \vdash (A, baba) \vdash$   
 $(B, aba) \vdash (A, ba)$

# Zobecněný nedeterministický konečný automat



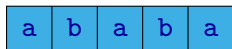
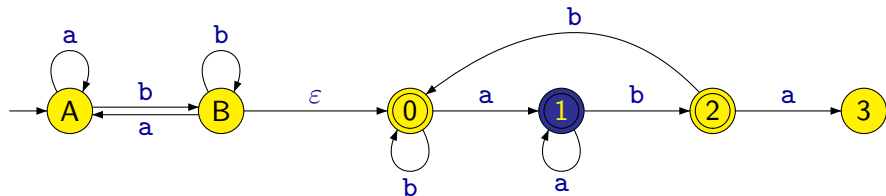
$(A, ababa) \vdash (A, baba) \vdash$   
 $(B, aba) \vdash (A, ba) \vdash$   
 $(B, a)$

# Zobecněný nedeterministický konečný automat



$(A, ababa) \vdash (A, baba) \vdash$   
 $(B, aba) \vdash (A, ba) \vdash$   
 $(B, a) \vdash (0, a)$

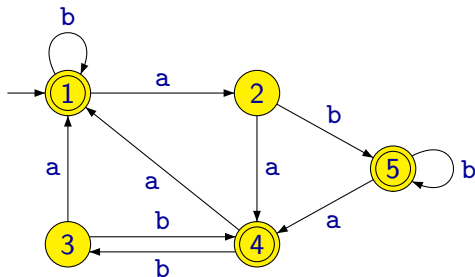
# Zobecněný nedeterministický konečný automat



$(A, ababa) \vdash (A, baba) \vdash$   
 $(B, aba) \vdash (A, ba) \vdash$   
 $(B, a) \vdash (0, a) \vdash (3, \varepsilon)$

Úplný přijímající výpočet

# Jazyk rozpoznávaný automatem

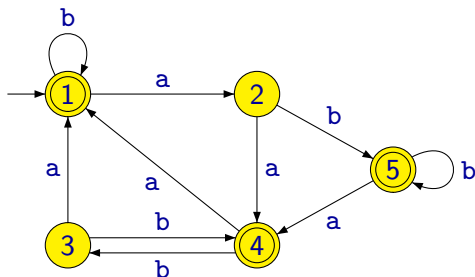


Uvažujme libovolný **tah** v grafu takový, že:

- Začíná v počátečním stavu.
- Končí v některém z přijímajících stavů.

Symbole, jimiž jsou ohodnoceny hrany (tj. přechody) v tomto tahu, tvoří **slovo**, které je přijímáno daným automatem.

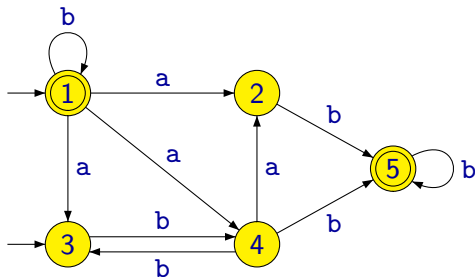
# Jazyk rozpoznávaný automatem



**Jazyk** rozpoznávaný automatem je množina všech slov, pro které v grafu existuje takovýto tah.



# Nedeterministický konečný automat



Je očividné, že pokud jazyk definujeme tímto způsobem, nemusíme se omezovat na grafy, kde:

- Z každého stavu vede právě jedna hrana označená daným symbolem abecedy.
- Máme právě jeden počáteční stav.

# Nedeterministický konečný automat

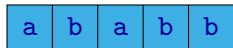
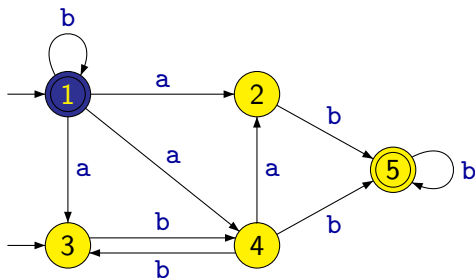
Takto obecněji definovaný automat se nazývá **nedeterministický konečný automat**.

Rozdíly oproti deterministickým konečným automatům:

- Z jednoho stavu může vézt libovolný (i nulový) počet přechodů označených stejným symbolem.
- V automatu může být víc než jeden počáteční stav.

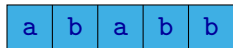
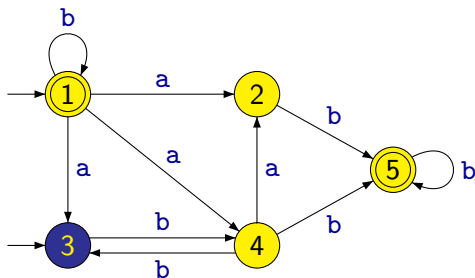
Pokud se na automat díváme jako na zařízení čtoucí slovo, vidíme, že jednomu slovu může odpovídat více než jeden výpočet (nebo naopak žádný).

# Nedeterministický konečný automat



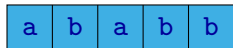
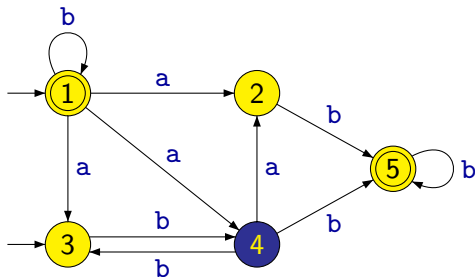
(1, ababb)

# Nedeterministický konečný automat



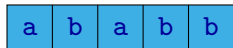
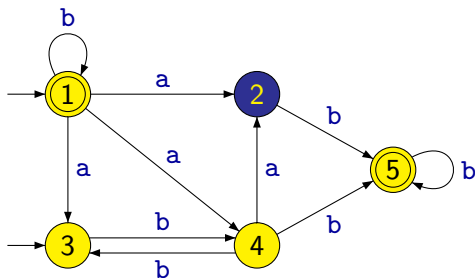
(1, ababb)  
⊢ (3, babb)

# Nedeterministický konečný automat



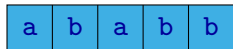
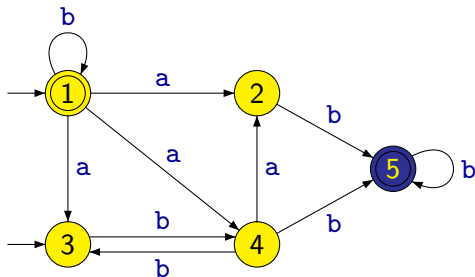
- (1, ababb)
- └ (3, babb)
- └ (4, abb)

# Nedeterministický konečný automat



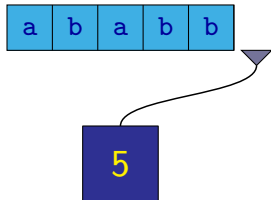
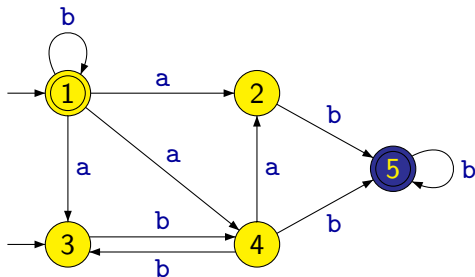
- (1, ababb)
- ┆ (3, babb)
- ┆ (4, abb)
- ┆ (2, bb)

# Nedeterministický konečný automat



- (1, ababb)
- ┆ (3, babb)
- ┆ (4, abb)
- ┆ (2, bb)
- ┆ (5, b)

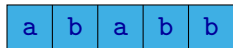
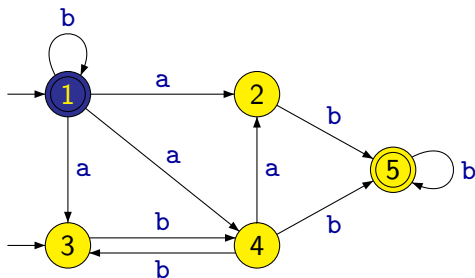
# Nedeterministický konečný automat



- (1, ababb)
- ⊢ (3, babb)
- ⊢ (4, abb)
- ⊢ (2, bb)
- ⊢ (5, b)
- ⊢ (5, ε)

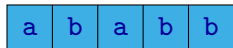
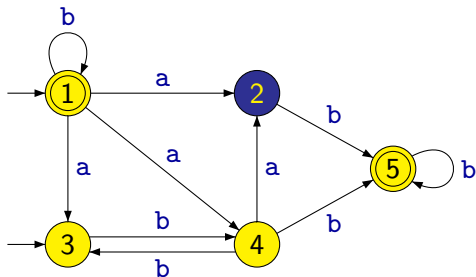


# Nedeterministický konečný automat



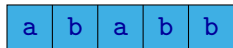
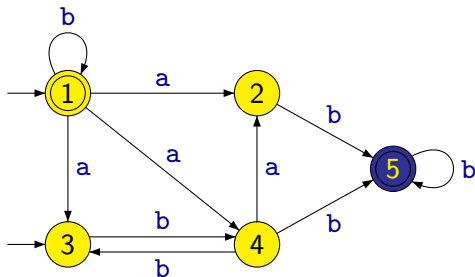
(1, ababb)

# Nedeterministický konečný automat



(1, ababb)  
└ (2, babb)

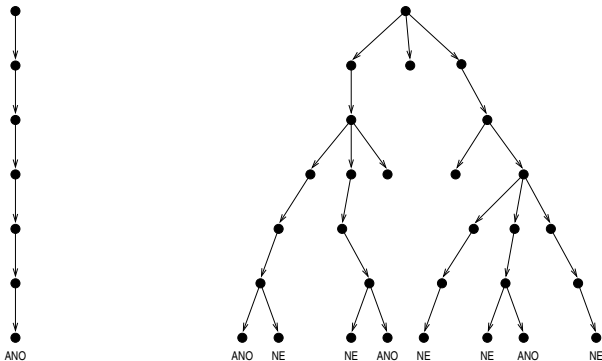
# Nedeterministický konečný automat



(1, ababb)  
⊢ (2, babb)  
⊢ (5, abb)

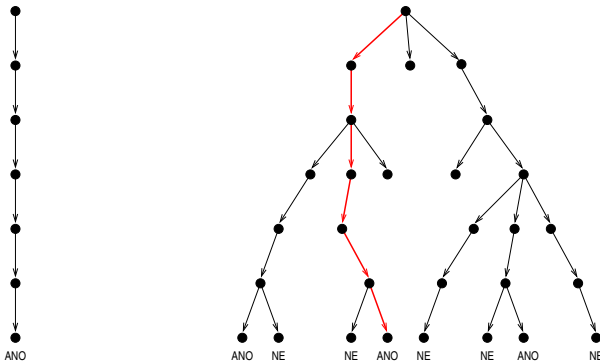
# Nedeterministický konečný automat

Nedeterministický konečný automat přijímá dané slovo, jestliže **existuje** alespoň jeden jeho výpočet, který vede k přijetí tohoto slova.



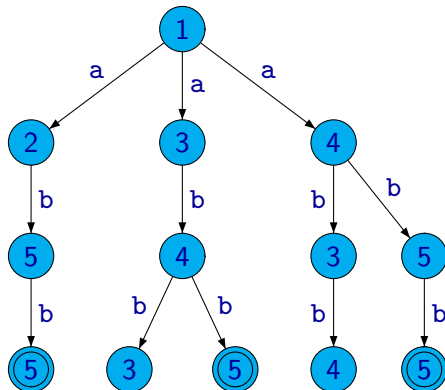
# Nedeterministický konečný automat

Nedeterministický konečný automat přijímá dané slovo, jestliže **existuje** alespoň jeden jeho výpočet, který vede k přijetí tohoto slova.



# Nedeterministický konečný automat

	a	b
↔ 1	2, 3, 4	1
2	—	5
→ 3	—	4
4	2	3, 5
← 5	—	5



**Příklad:** Les reprezentující všechny možné výpočty nad slovem **abb**.

# Nedeterministický konečný automat

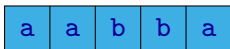
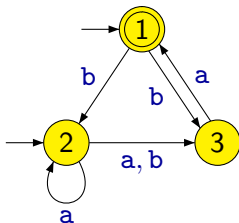
Formálně je **nedeterministický konečný automat** definován jako pětice

$$(Q, \Sigma, \delta, I, F)$$

kde:

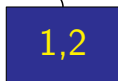
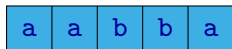
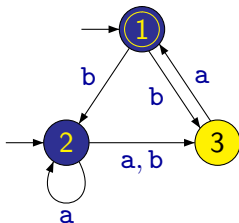
- $Q$  je konečná množina **stavů**
- $\Sigma$  je konečná **abeceda**
- $\delta : Q \times \Sigma \rightarrow \mathcal{P}(Q)$  je **přechodová funkce**
- $I \subseteq Q$  je množina **počátečních stavů**
- $F \subseteq Q$  je množina **přijímajících stavů**

# Převod nedeterministického automatu na deterministický

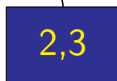
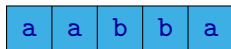
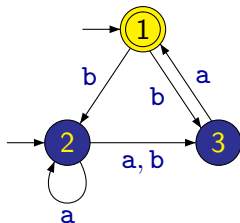




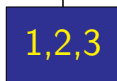
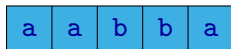
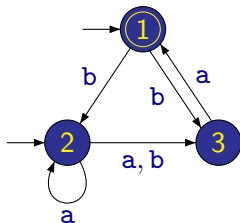
# Převod nedeterministického automatu na deterministický



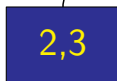
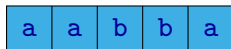
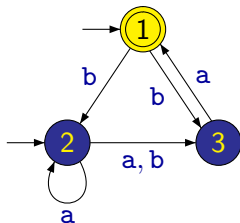
# Převod nedeterministického automatu na deterministický



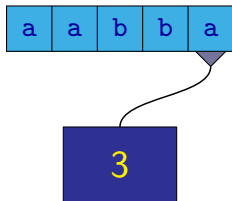
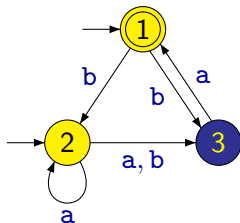
# Převod nedeterministického automatu na deterministický



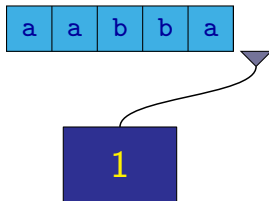
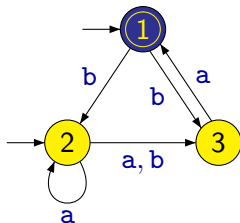
# Převod nedeterministického automatu na deterministický



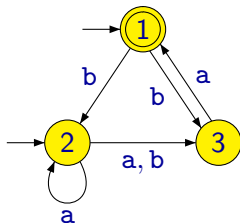
# Převod nedeterministického automatu na deterministický



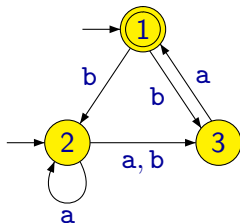
# Převod nedeterministického automatu na deterministický



# Převod nedeterministického automatu na deterministický

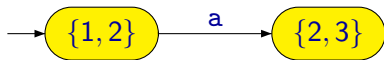
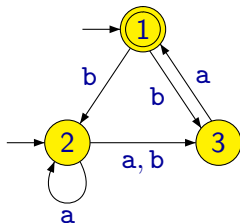


# Převod nedeterministického automatu na deterministický

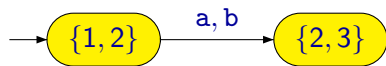
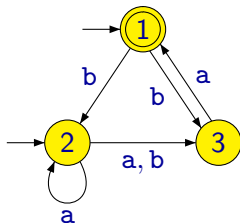




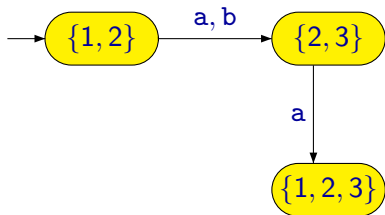
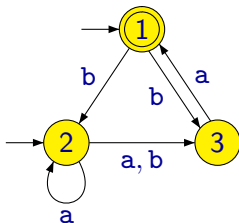
# Převod nedeterministického automatu na deterministický



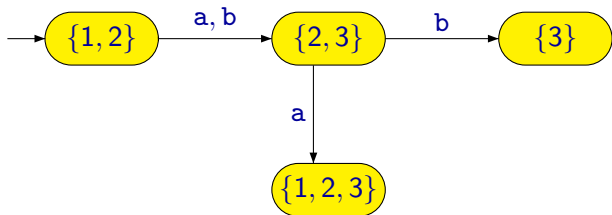
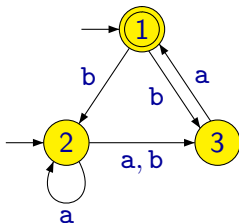
# Převod nedeterministického automatu na deterministický



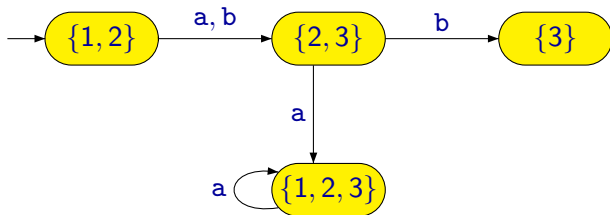
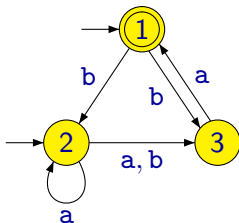
# Převod nedeterministického automatu na deterministický



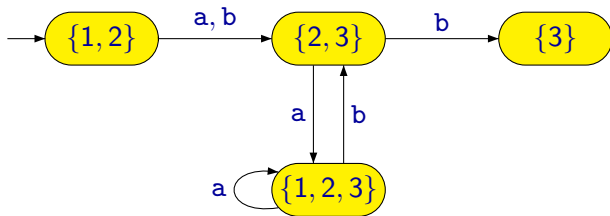
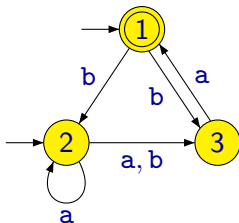
# Převod nedeterministického automatu na deterministický



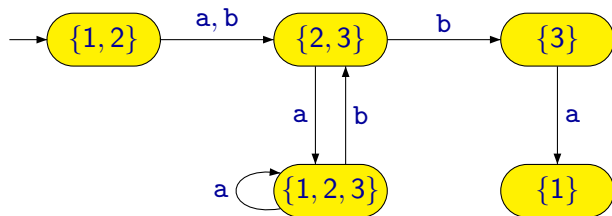
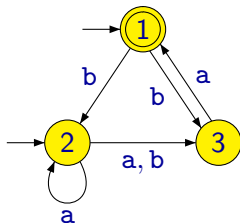
# Převod nedeterministického automatu na deterministický



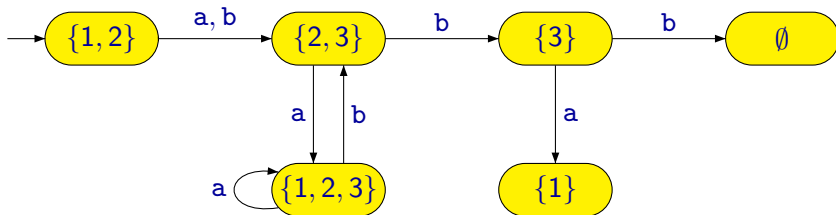
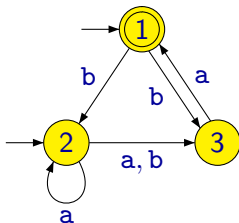
# Převod nedeterministického automatu na deterministický



# Převod nedeterministického automatu na deterministický

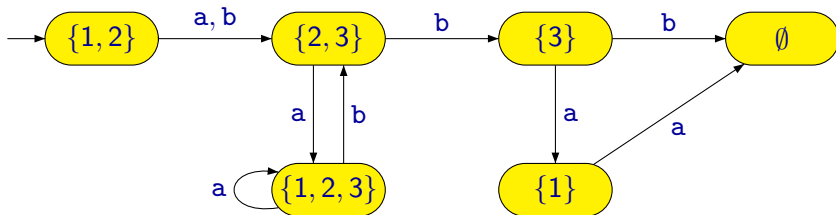
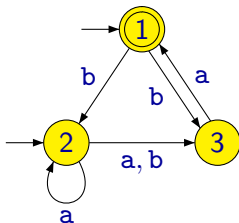


# Převod nedeterministického automatu na deterministický

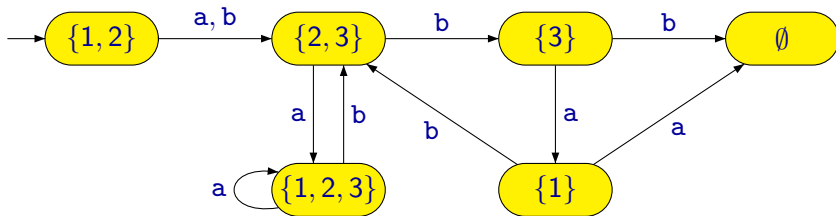
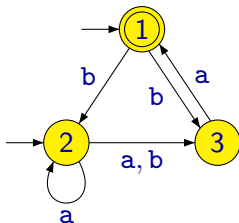




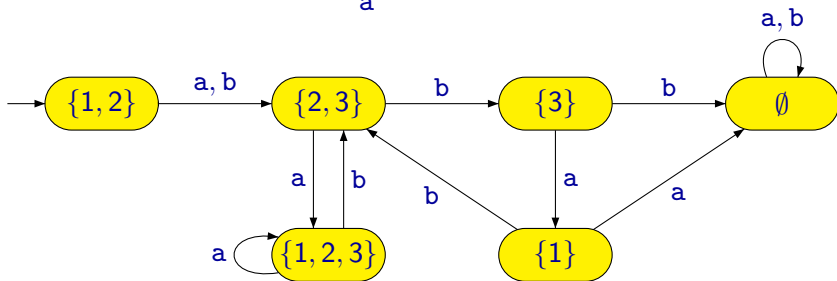
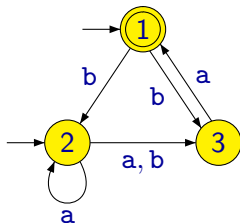
# Převod nedeterministického automatu na deterministický



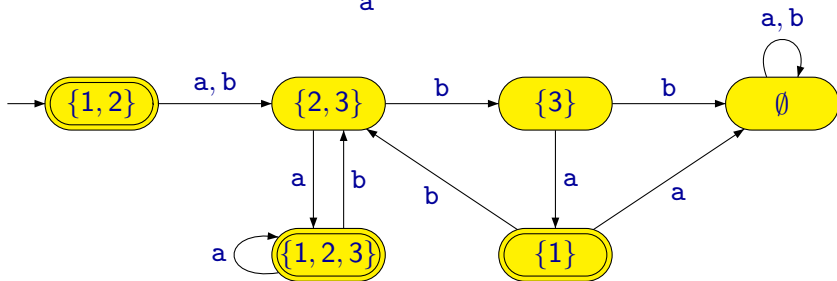
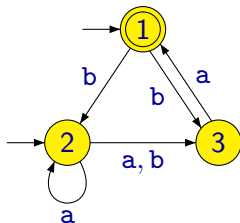
# Převod nedeterministického automatu na deterministický



# Převod nedeterministického automatu na deterministický



# Převod nedeterministického automatu na deterministický



# Převod nedeterministického automatu na deterministický

	a	b
$\leftrightarrow 1$	—	2, 3
$\rightarrow 2$	2, 3	3
3	1	—

# Převod nedeterministického automatu na deterministický

	a	b
$\leftrightarrow 1$	—	2,3
$\rightarrow 2$	2,3	3
3	1	—

	a	b

# Převod nedeterministického automatu na deterministický

	a	b
$\leftrightarrow 1$	—	2, 3
$\rightarrow 2$	2, 3	3
3	1	—

	a	b
$\leftrightarrow \{1, 2\}$		

# Převod nedeterministického automatu na deterministický

	a	b
$\leftrightarrow 1$	—	2, 3
$\rightarrow 2$	2, 3	3
3	1	—

	a	b
$\leftrightarrow \{1, 2\}$	$\{2, 3\}$	



# Převod nedeterministického automatu na deterministický

	a	b
$\leftrightarrow 1$	—	2, 3
$\rightarrow 2$	2, 3	3
3	1	—

	a	b
$\leftrightarrow \{1, 2\}$ $\{2, 3\}$	$\{2, 3\}$	

# Převod nedeterministického automatu na deterministický

	a	b
$\leftrightarrow 1$	—	2, 3
$\rightarrow 2$	2, 3	3
3	1	—

	a	b
$\leftrightarrow \{1, 2\}$ $\{2, 3\}$	$\{2, 3\}$	$\{2, 3\}$

# Převod nedeterministického automatu na deterministický

	a	b
$\leftrightarrow 1$	—	2, 3
$\rightarrow 2$	2, 3	3
3	1	—

	a	b
$\leftrightarrow \{1, 2\}$	$\{2, 3\}$	$\{2, 3\}$
$\{2, 3\}$	$\{1, 2, 3\}$	

# Převod nedeterministického automatu na deterministický

	a	b
$\leftrightarrow 1$	—	2, 3
$\rightarrow 2$	2, 3	3
3	1	—

	a	b
$\leftrightarrow \{1, 2\}$	$\{2, 3\}$	$\{2, 3\}$
$\{2, 3\}$	$\{1, 2, 3\}$	
$\leftarrow \{1, 2, 3\}$		

# Převod nedeterministického automatu na deterministický

	a	b
$\leftrightarrow 1$	—	2, 3
$\rightarrow 2$	2, 3	3
3	1	—

	a	b
$\leftrightarrow \{1, 2\}$	$\{2, 3\}$	$\{2, 3\}$
$\{2, 3\}$	$\{1, 2, 3\}$	$\{3\}$
$\leftarrow \{1, 2, 3\}$		

# Převod nedeterministického automatu na deterministický

	a	b
$\leftrightarrow 1$	—	2, 3
$\rightarrow 2$	2, 3	3
3	1	—

	a	b
$\leftrightarrow \{1, 2\}$	$\{2, 3\}$	$\{2, 3\}$
$\{2, 3\}$	$\{1, 2, 3\}$	$\{3\}$
$\leftarrow \{1, 2, 3\}$		
$\{3\}$		

# Převod nedeterministického automatu na deterministický

	a	b
$\leftrightarrow 1$	—	2, 3
$\rightarrow 2$	2, 3	3
3	1	—

	a	b
$\leftrightarrow \{1, 2\}$	$\{2, 3\}$	$\{2, 3\}$
$\{2, 3\}$	$\{1, 2, 3\}$	$\{3\}$
$\leftarrow \{1, 2, 3\}$	$\{1, 2, 3\}$	
$\{3\}$		

# Převod nedeterministického automatu na deterministický

	a	b
$\leftrightarrow 1$	—	2, 3
$\rightarrow 2$	2, 3	3
3	1	—

	a	b
$\leftrightarrow \{1, 2\}$	$\{2, 3\}$	$\{2, 3\}$
$\{2, 3\}$	$\{1, 2, 3\}$	$\{3\}$
$\leftarrow \{1, 2, 3\}$	$\{1, 2, 3\}$	$\{2, 3\}$
$\{3\}$		



# Převod nedeterministického automatu na deterministický

	a	b
$\leftrightarrow 1$	—	2, 3
$\rightarrow 2$	2, 3	3
3	1	—

	a	b
$\leftrightarrow \{1, 2\}$	$\{2, 3\}$	$\{2, 3\}$
$\{2, 3\}$	$\{1, 2, 3\}$	$\{3\}$
$\leftarrow \{1, 2, 3\}$	$\{1, 2, 3\}$	$\{2, 3\}$
$\{3\}$	$\{1\}$	

# Převod nedeterministického automatu na deterministický

	a	b
$\leftrightarrow 1$	—	2, 3
$\rightarrow 2$	2, 3	3
3	1	—

	a	b
$\leftrightarrow \{1, 2\}$	$\{2, 3\}$	$\{2, 3\}$
$\{2, 3\}$	$\{1, 2, 3\}$	$\{3\}$
$\leftarrow \{1, 2, 3\}$	$\{1, 2, 3\}$	$\{2, 3\}$
$\{3\}$	$\{1\}$	
$\leftarrow \{1\}$		

# Převod nedeterministického automatu na deterministický

	a	b
$\leftrightarrow 1$	—	2, 3
$\rightarrow 2$	2, 3	3
3	1	—

	a	b
$\leftrightarrow \{1, 2\}$	$\{2, 3\}$	$\{2, 3\}$
$\{2, 3\}$	$\{1, 2, 3\}$	$\{3\}$
$\leftarrow \{1, 2, 3\}$	$\{1, 2, 3\}$	$\{2, 3\}$
$\{3\}$	$\{1\}$	$\emptyset$
$\leftarrow \{1\}$		

# Převod nedeterministického automatu na deterministický

	a	b
$\leftrightarrow 1$	—	2, 3
$\rightarrow 2$	2, 3	3
3	1	—

	a	b
$\leftrightarrow \{1, 2\}$	$\{2, 3\}$	$\{2, 3\}$
$\{2, 3\}$	$\{1, 2, 3\}$	$\{3\}$
$\leftarrow \{1, 2, 3\}$	$\{1, 2, 3\}$	$\{2, 3\}$
$\{3\}$	$\{1\}$	$\emptyset$
$\leftarrow \{1\}$		
$\emptyset$		

# Převod nedeterministického automatu na deterministický

	a	b
$\leftrightarrow 1$	—	2, 3
$\rightarrow 2$	2, 3	3
3	1	—

	a	b
$\leftrightarrow \{1, 2\}$	$\{2, 3\}$	$\{2, 3\}$
$\{2, 3\}$	$\{1, 2, 3\}$	$\{3\}$
$\leftarrow \{1, 2, 3\}$	$\{1, 2, 3\}$	$\{2, 3\}$
$\{3\}$	$\{1\}$	$\emptyset$
$\leftarrow \{1\}$	$\emptyset$	
$\emptyset$		

# Převod nedeterministického automatu na deterministický

	a	b
$\leftrightarrow 1$	—	2, 3
$\rightarrow 2$	2, 3	3
3	1	—

	a	b
$\leftrightarrow \{1, 2\}$	$\{2, 3\}$	$\{2, 3\}$
$\{2, 3\}$	$\{1, 2, 3\}$	$\{3\}$
$\leftarrow \{1, 2, 3\}$	$\{1, 2, 3\}$	$\{2, 3\}$
$\{3\}$	$\{1\}$	$\emptyset$
$\leftarrow \{1\}$	$\emptyset$	$\{2, 3\}$
$\emptyset$		

# Převod nedeterministického automatu na deterministický

	a	b
$\leftrightarrow 1$	—	2, 3
$\rightarrow 2$	2, 3	3
3	1	—

	a	b
$\leftrightarrow \{1, 2\}$	$\{2, 3\}$	$\{2, 3\}$
$\{2, 3\}$	$\{1, 2, 3\}$	$\{3\}$
$\leftarrow \{1, 2, 3\}$	$\{1, 2, 3\}$	$\{2, 3\}$
$\{3\}$	$\{1\}$	$\emptyset$
$\leftarrow \{1\}$	$\emptyset$	$\{2, 3\}$
$\emptyset$	$\emptyset$	$\emptyset$

# Převod nedeterministického automatu na deterministický

	a	b
$\leftrightarrow 1$	—	2, 3
$\rightarrow 2$	2, 3	3
3	1	—

	a	b
$\leftrightarrow \{1, 2\}$	$\{2, 3\}$	$\{2, 3\}$
$\{2, 3\}$	$\{1, 2, 3\}$	$\{3\}$
$\leftarrow \{1, 2, 3\}$	$\{1, 2, 3\}$	$\{2, 3\}$
$\{3\}$	$\{1\}$	$\emptyset$
$\leftarrow \{1\}$	$\emptyset$	$\{2, 3\}$
$\emptyset$	$\emptyset$	$\emptyset$

	a	b
$\leftrightarrow 1$	2	2
2	3	4
$\leftarrow 3$	3	2
4	5	6
$\leftarrow 5$	6	2
6	6	6

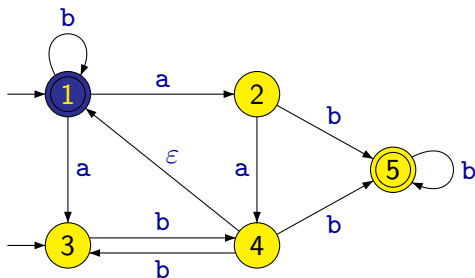


**Poznámka:** Při převodu nedeterministického automatu, který má  $n$  stavů, může mít výsledný deterministický automat až  $2^n$  stavů.

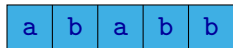
Například při převodu automatu, který má 20 stavů, může vzniknout automat, který má  $2^{20} = 1048576$  stavů.

Často má sice výsledný automat podstatně méně než  $2^n$  stavů, nicméně tyto nejhorší případy občas nastávají.

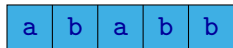
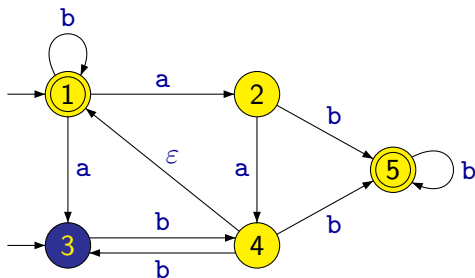
# Zobecněný nedeterministický konečný automat



(1, ababb)

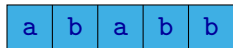
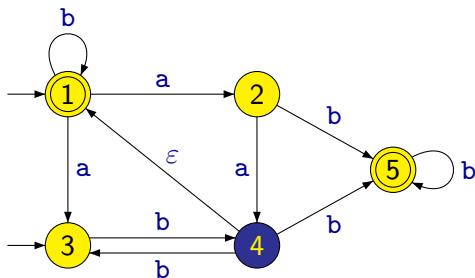


# Zobecněný nedeterministický konečný automat



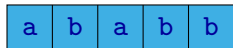
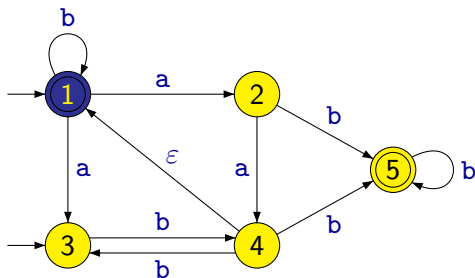
(1, ababb)  
└ (3, babb)

# Zobecněný nedeterministický konečný automat



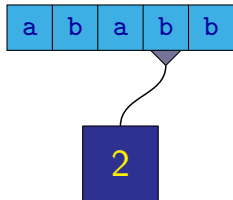
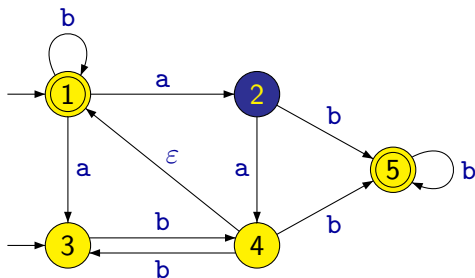
- (1, ababb)
- ┆ (3, babb)
- ┆ (4, abb)

# Zobecněný nedeterministický konečný automat



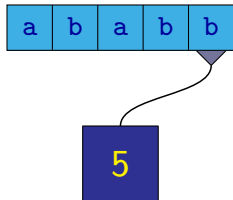
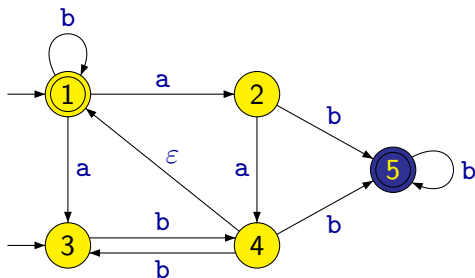
- (1, ababb)
- ⊢ (3, babb)
- ⊢ (4, abb)
- ⊢ (1, abb)

# Zobecněný nedeterministický konečný automat



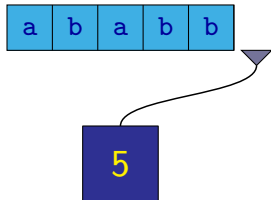
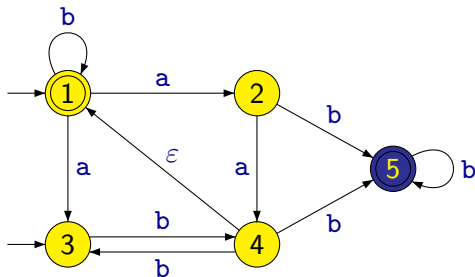
- (1, ababb)
- ┆ (3, babb)
- ┆ (4, abb)
- ┆ (1, abb)
- ┆ (2, bb)

# Zobecněný nedeterministický konečný automat



- (1, ababb)
- ⊢ (3, babb)
- ⊢ (4, abb)
- ⊢ (1, abb)
- ⊢ (2, bb)
- ⊢ (5, b)

# Zobecněný nedeterministický konečný automat



- (1, ababb)
- ⊢ (3, babb)
- ⊢ (4, abb)
- ⊢ (1, abb)
- ⊢ (2, bb)
- ⊢ (5, b)
- ⊢ (5, ε)



Oproti nedeterministickému konečnému automatu má **zobecněný nedeterministický konečný automat** tzv.  **$\varepsilon$ -přechody**, tj. přechody označené symbolem  $\varepsilon$ .

Při provádění  $\varepsilon$ -přechodu se mění pouze stav řídicí jednotky, ale hlava na pásce se neposouvá.

**Poznámka:** Výpočty zobecněného nedeterministického automatu mohou být libovolně dlouhé a dokonce i nekonečné (pokud graf obsahuje cyklus tvořený  $\varepsilon$ -přechody) bez ohledu na délku slova na pásce.

# Zobecněný nedeterministický konečný automat

Formálně je **zobecněný nedeterministický konečný automat** definován jako pětice

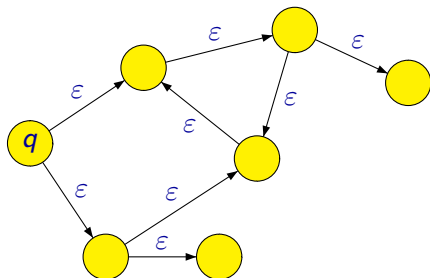
$$(Q, \Sigma, \delta, I, F)$$

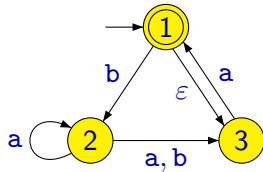
kde:

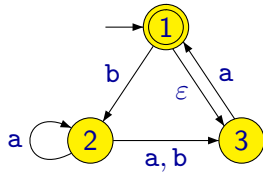
- $Q$  je konečná množina **stavů**
- $\Sigma$  je konečná **abeceda**
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow \mathcal{P}(Q)$  je **přechodová funkce**
- $I \subseteq Q$  je množina **počátečních stavů**
- $F \subseteq Q$  je množina **přijímajících stavů**

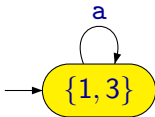
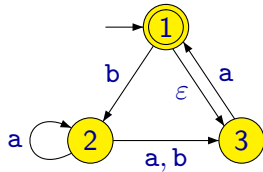
# Převod na deterministický konečný automat

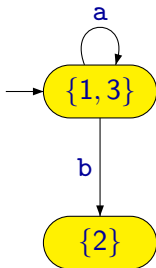
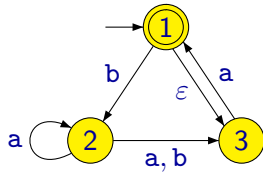
Zobecněný nedeterministický konečný automat je možné převést na deterministický podobnou konstrukcí jako nedeterministický konečný automat, s tím rozdílem, že do množin stavů musíme vždy přidat navíc i všechny stavy dosažitelné daných stavů nějakou sekvencí  $\epsilon$ -přechodů.

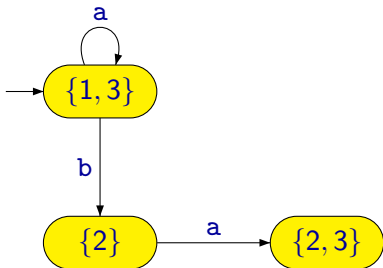
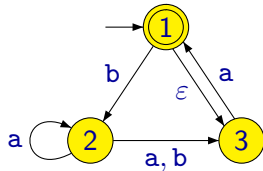




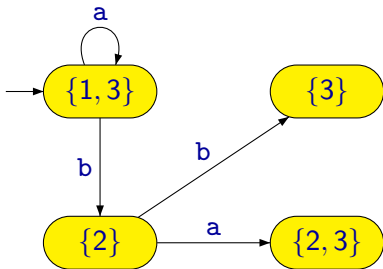
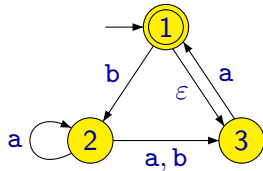


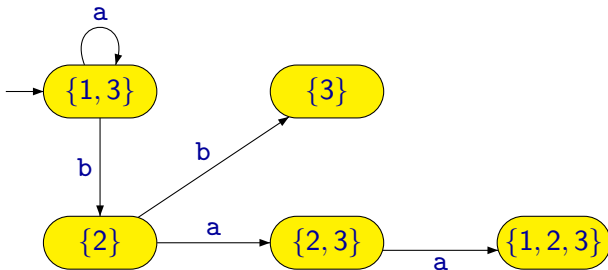
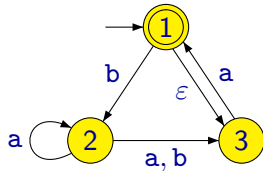


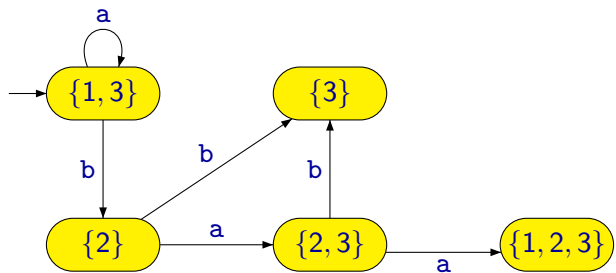
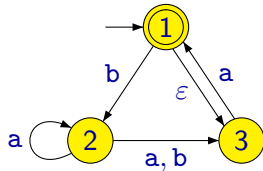


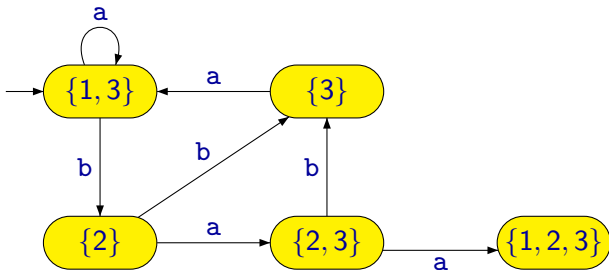
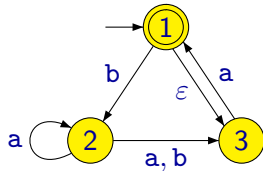


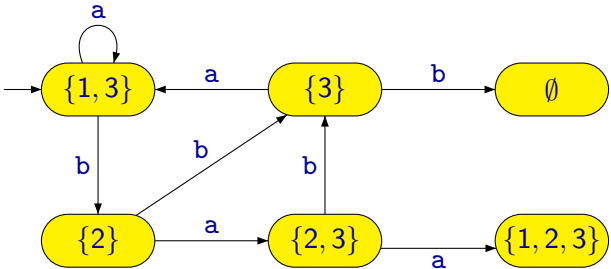
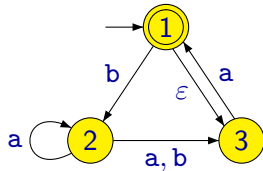


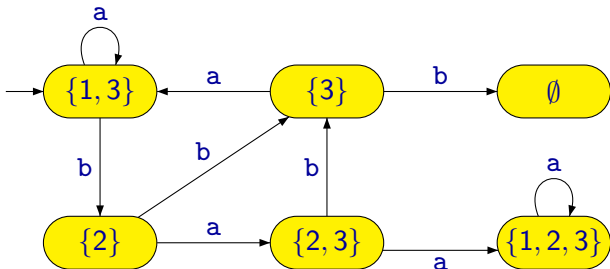
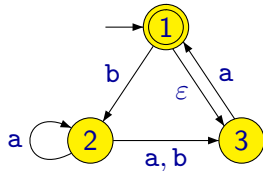


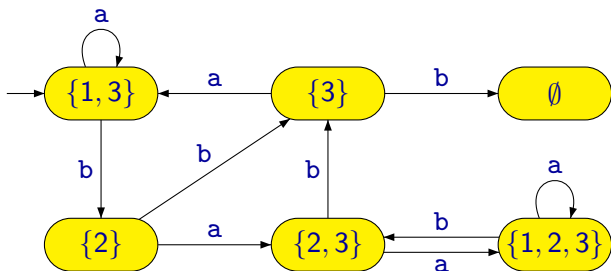
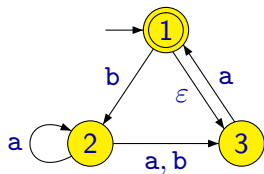


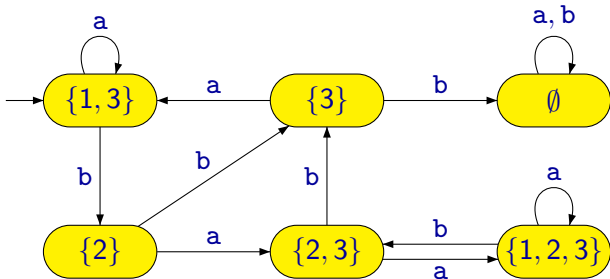
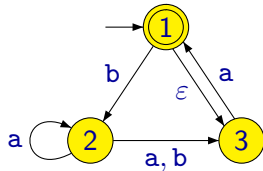




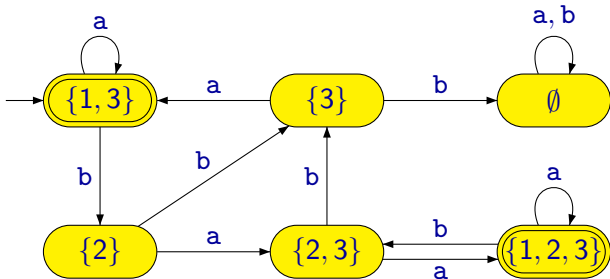
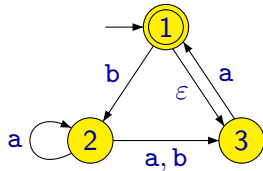






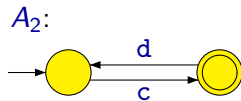
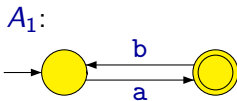






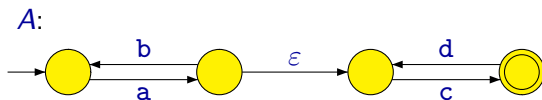
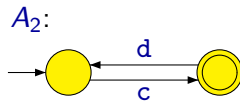
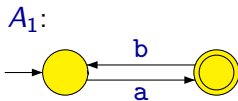
# Zřetězení jazyků

$$\Sigma = \{a, b, c, d\}$$



# Zřetězení jazyků

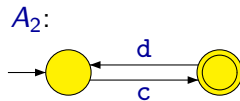
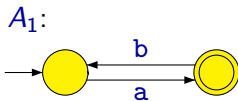
$$\Sigma = \{a, b, c, d\}$$



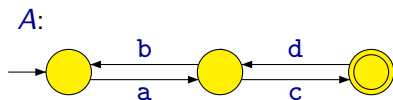
$$L(A) = L(A_1) \cdot L(A_2)$$

# Zřetězení jazyků

$$\Sigma = \{a, b, c, d\}$$

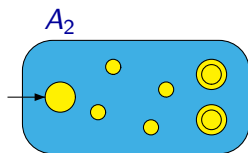
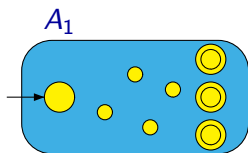


Chybná konstrukce:

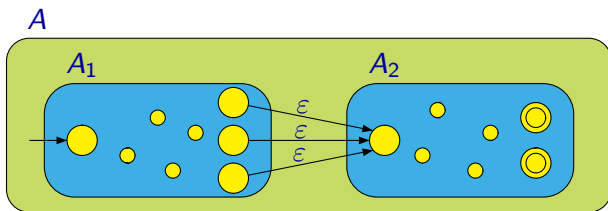
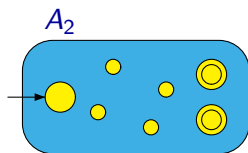
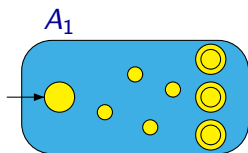


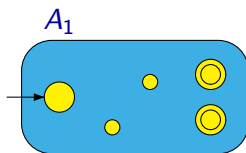
$acdbac \in L(A)$ , ale  $acdbac \notin L(A_1) \cdot L(A_2)$

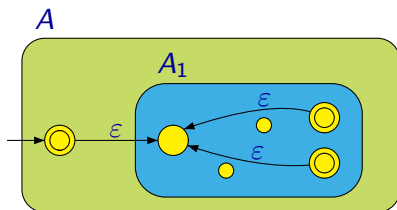
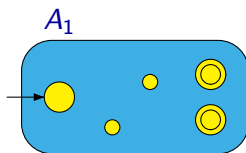
# Zřetězení jazyků



# Zřetězení jazyků

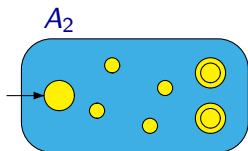
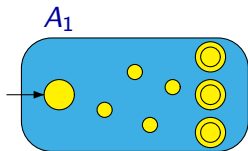






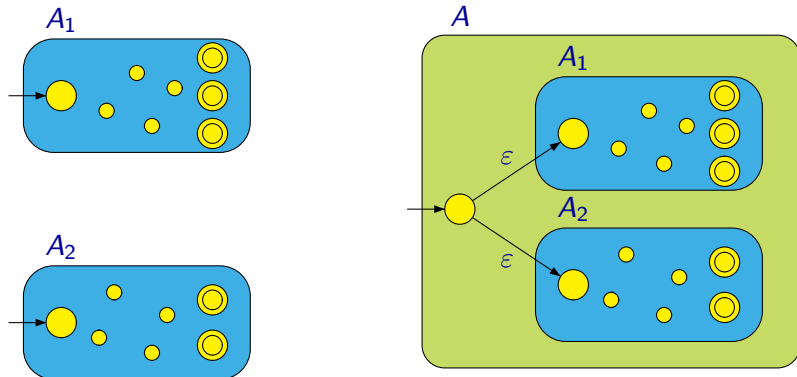


Alternativní konstrukce pro sjednocení jazyků:



# Sjednocení jazyků

Alternativní konstrukce pro sjednocení jazyků:



Předpokládejme, že máme dány množiny  $X$  a  $Y$ , kde  $X \subseteq Y$ , a dále nějakou operaci  $f : Y \times Y \times \cdots \times Y \rightarrow Y$ .

O množině  $X$  řekneme, že je **uzavřená** vůči operaci  $f$ , jestliže platí, že pokud  $x_1, x_2, \dots, x_k \in X$ , pak  $f(x_1, x_2, \dots, x_k) \in X$ .

**Příklad:** Uvažujme množinu přirozených čísel  $\mathbb{N} = \{0, 1, 2, 3, \dots\}$  a množinu reálných čísel  $\mathbb{R}$ .

- Množina  $\mathbb{N}$  je uzavřená vůči operacím  $+$  a  $\times$ , ale není uzavřená vůči operacím  $-$  a  $/$ .  
Například  $3 + 5 \in \mathbb{N}$ , ale  $3 - 5 \notin \mathbb{N}$  a  $3/5 \notin \mathbb{N}$
- Množina  $\mathbb{R}$  je uzavřená vůči operacím  $+$ ,  $\times$ ,  $-$ ,  $/$ .

Množina (všech) regulárních jazyků je uzavřená vůči operacím:

- sjednocení
- průniku
- doplňku
- zřetězení
- iteraci
- ...

**Poznámka:** Jazyk je regulární, pokud existuje konečný automat, který ho rozpoznává.

Existují i jazyky, které nejsou regulární.

# Regulární výrazy

Jako například v aritmetice můžeme pomocí operátorů  $+$  a  $\times$  vytvářet výrazy jako

$$(5 + 3) \times 4$$

můžeme v teorii formálních jazyků pomocí operátorů  $+$ ,  $\cdot$  a  $*$  vytvářet tzv. **regulární výrazy**, jako třeba

$$(0 + 1) \cdot 0^*$$

které reprezentují jazyky.

Jako je hodnotou aritmetického výrazu  $(5 + 3) \times 4$  číslo 32, je hodnotou regulárního výrazu  $(0 + 1) \cdot 0^*$  jazyk

$$(\{0\} \cup \{1\}) \cdot \{0\}^*$$

Induktivní definice regulárních výrazů nad abecedou  $\Sigma$ :

- $\emptyset$ ,  $\varepsilon$ ,  $a$  (kde  $a \in \Sigma$ ) jsou regulární výrazy:
  - $\emptyset$  ... označuje prázdný jazyk
  - $\varepsilon$  ... označuje jazyk  $\{\varepsilon\}$
  - $a$  ... označuje jazyk  $\{a\}$
- Jestliže  $\alpha$ ,  $\beta$  jsou regulární výrazy, pak i  $(\alpha + \beta)$ ,  $(\alpha \cdot \beta)$ ,  $(\alpha^*)$  jsou regulární výrazy:
  - $(\alpha + \beta)$  ... označuje sjednocení jazyků označených  $\alpha$  a  $\beta$
  - $(\alpha \cdot \beta)$  ... označuje zřetězení jazyků označených  $\alpha$  a  $\beta$
  - $(\alpha^*)$  ... označuje iteraci jazyka označeného  $\alpha$
- Neexistují žádné další regulární výrazy než ty definované podle předchozích dvou bodů.

## Příklad:

- Podle definice jsou  $0$  i  $1$  regulární výrazy.

## Příklad:

- Podle definice jsou  $0$  i  $1$  regulární výrazy.
- Protože  $0$  i  $1$  jsou regulární výrazy, je i  $(0 + 1)$  regulární výraz.



## Příklad:

- Podle definice jsou  $0$  i  $1$  regulární výrazy.
- Protože  $0$  i  $1$  jsou regulární výrazy, je i  $(0 + 1)$  regulární výraz.
- Protože  $0$  je regulární výraz, je i  $(0^*)$  regulární výraz.

## Příklad:

- Podle definice jsou  $0$  i  $1$  regulární výrazy.
- Protože  $0$  i  $1$  jsou regulární výrazy, je i  $(0 + 1)$  regulární výraz.
- Protože  $0$  je regulární výraz, je i  $(0^*)$  regulární výraz.
- Protože  $(0 + 1)$  i  $(0^*)$  jsou regulární výrazy, je i  $((0 + 1) \cdot (0^*))$  regulární výraz.

## Příklad:

- Podle definice jsou  $0$  i  $1$  regulární výrazy.
- Protože  $0$  i  $1$  jsou regulární výrazy, je i  $(0 + 1)$  regulární výraz.
- Protože  $0$  je regulární výraz, je i  $(0^*)$  regulární výraz.
- Protože  $(0 + 1)$  i  $(0^*)$  jsou regulární výrazy, je i  $((0 + 1) \cdot (0^*))$  regulární výraz.

**Poznámka:** Jestliže  $\alpha$  je regulární výraz, zápisem  $[\alpha]$  označujeme jazyk definovaný regulárním výrazem  $\alpha$ .

$$[((0 + 1) \cdot (0^*))] = \{0, 1, 00, 10, 000, 100, 0000, 1000, 00000, \dots\}$$

Aby byl zápis regulárních výrazů přehlednější a stručnější, používáme následující pravidla:

- Vynecháváme vnější pár závorek.
- Vynecháváme závorky, které jsou zbytečné vzhledem k asociativitě operací sjednocení (+) a zřetězení (·).
- Vynecháváme závorky, které jsou zbytečné vzhledem k prioritě operací (nejvyšší prioritu má iterace (\*), menší zřetězení (·) a nejmenší sjednocení (+)).
- Nepíšeme tečku pro zřetězení.

**Příklad:** Místo

$$((((((0 \cdot 1)^*) \cdot 1) \cdot (1 \cdot 1)) + (((0 \cdot 0) + 1)^*))$$

obvykle píšeme

$$(01)^*111 + (00 + 1)^*$$

**Příklady:** Ve všech případech  $\Sigma = \{0, 1\}$ .

0 ... jazyk tvořený jediným slovem 0

**Příklady:** Ve všech případech  $\Sigma = \{0, 1\}$ .

0 ... jazyk tvořený jediným slovem 0

01 ... jazyk tvořený jediným slovem 01

**Příklady:** Ve všech případech  $\Sigma = \{0, 1\}$ .

$0$  ... jazyk tvořený jediným slovem  $0$

$01$  ... jazyk tvořený jediným slovem  $01$

$0 + 1$  ... jazyk tvořený dvěma slovy  $0$  a  $1$

**Příklady:** Ve všech případech  $\Sigma = \{0, 1\}$ .

0 ... jazyk tvořený jediným slovem 0

01 ... jazyk tvořený jediným slovem 01

0 + 1 ... jazyk tvořený dvěma slovy 0 a 1

0\* ... jazyk tvořený slovy  $\epsilon$ , 0, 00, 000, ...



**Příklady:** Ve všech případech  $\Sigma = \{0, 1\}$ .

$0$  ... jazyk tvořený jediným slovem  $0$

$01$  ... jazyk tvořený jediným slovem  $01$

$0 + 1$  ... jazyk tvořený dvěma slovy  $0$  a  $1$

$0^*$  ... jazyk tvořený slovy  $\varepsilon, 0, 00, 000, \dots$

$(01)^*$  ... jazyk tvořený slovy  $\varepsilon, 01, 0101, 010101, \dots$

**Příklady:** Ve všech případech  $\Sigma = \{0, 1\}$ .

$0$  ... jazyk tvořený jediným slovem  $0$

$01$  ... jazyk tvořený jediným slovem  $01$

$0 + 1$  ... jazyk tvořený dvěma slovy  $0$  a  $1$

$0^*$  ... jazyk tvořený slovy  $\varepsilon, 0, 00, 000, \dots$

$(01)^*$  ... jazyk tvořený slovy  $\varepsilon, 01, 0101, 010101, \dots$

$(0 + 1)^*$  ... jazyk tvořený všemi slovy nad abecedou  $\{0, 1\}$

**Příklady:** Ve všech případech  $\Sigma = \{0, 1\}$ .

$0$  ... jazyk tvořený jediným slovem  $0$

$01$  ... jazyk tvořený jediným slovem  $01$

$0 + 1$  ... jazyk tvořený dvěma slovy  $0$  a  $1$

$0^*$  ... jazyk tvořený slovy  $\varepsilon, 0, 00, 000, \dots$

$(01)^*$  ... jazyk tvořený slovy  $\varepsilon, 01, 0101, 010101, \dots$

$(0 + 1)^*$  ... jazyk tvořený všemi slovy nad abecedou  $\{0, 1\}$

$(0 + 1)^*00$  ... jazyk tvořený všemi slovy končícími  $00$

**Příklady:** Ve všech případech  $\Sigma = \{0, 1\}$ .

$0$  ... jazyk tvořený jediným slovem  $0$

$01$  ... jazyk tvořený jediným slovem  $01$

$0 + 1$  ... jazyk tvořený dvěma slovy  $0$  a  $1$

$0^*$  ... jazyk tvořený slovy  $\varepsilon, 0, 00, 000, \dots$

$(01)^*$  ... jazyk tvořený slovy  $\varepsilon, 01, 0101, 010101, \dots$

$(0 + 1)^*$  ... jazyk tvořený všemi slovy nad abecedou  $\{0, 1\}$

$(0 + 1)^*00$  ... jazyk tvořený všemi slovy končícími  $00$

$(01)^*111(01)^*$  ... jazyk tvořený všemi slovy obsahujícími podslovo  $111$  předcházené i následované libovolným počtem slov  $01$

$(0 + 1)^*00 + (01)^*111(01)^*$  ... jazyk tvořený všemi slovy, která buď končí **00** nebo obsahují podslovo **111** předcházené i následované libovolným počtem slov **01**

$(0 + 1)^*00 + (01)^*111(01)^*$  ... jazyk tvořený všemi slovy, která buď končí  $00$  nebo obsahují podslovo  $111$  předcházené i následované libovolným počtem slov  $01$

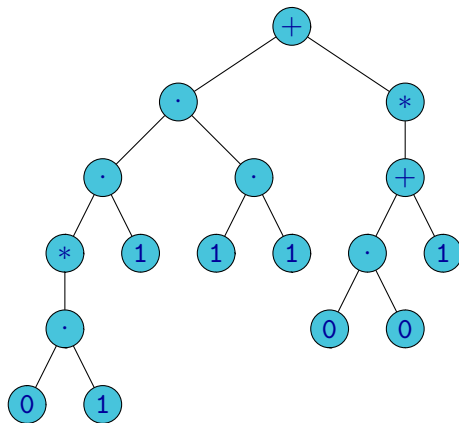
$(0 + 1)^*1(0 + 1)^*$  ... jazyk tvořený všemi slovy obsahujícími alespoň jeden symbol  $1$

$(0 + 1)^*00 + (01)^*111(01)^*$  ... jazyk tvořený všemi slovy, která buď končí  $00$  nebo obsahují podslovo  $111$  předcházené i následované libovolným počtem slov  $01$

$(0 + 1)^*1(0 + 1)^*$  ... jazyk tvořený všemi slovy obsahujícími alespoň jeden symbol  $1$

$(0^*10^*10^*)^*$  ... jazyk tvořený všemi slovy obsahujícími sudý počet symbolů  $1$

Strukturu regulárního výrazu si můžeme znázornit jako strom:



$(((((0 \cdot 1)^*) \cdot 1) \cdot (1 \cdot 1)) + (((0 \cdot 0) + 1)^*))$



## Tvrzení

Každý jazyk, který je možné vyjádřit regulárním výrazem, je regulární (tj. rozpoznávaný nějakým konečným automatem).

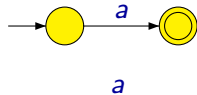
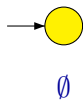
**Důkaz:** Stačí ukázat, jak k danému regulárnímu výrazu  $\alpha$  zkonstruovat konečný automat, který rozpoznává jazyk  $[\alpha]$ .

Konstrukce je rekurzivní a postupuje podle struktury výrazu  $\alpha$ :

- Pokud je  $\alpha$  elementární výraz (tj.  $\emptyset$ ,  $\varepsilon$  nebo  $a$ ):
  - Sestrojíme přímo odpovídající automat.
- Pokud je  $\alpha$  tvaru  $(\beta + \gamma)$ ,  $(\beta \cdot \gamma)$  nebo  $(\beta^*)$ :
  - Rekurzivně sestrojíme automaty rozpoznávající jazyky  $[\beta]$  a  $[\gamma]$ .
  - Z nich sestrojíme automat rozpoznávající jazyk  $[\alpha]$ .

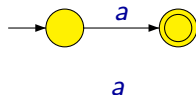
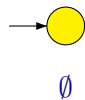
# Převod regulárního výrazu na konečný automat

Automaty pro elementární výrazy:

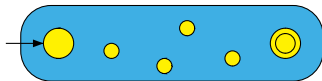
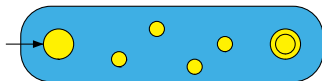


# Převod regulárního výrazu na konečný automat

Automaty pro elementární výrazy:

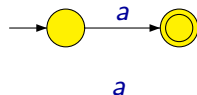
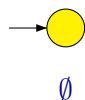


Konstrukce pro sjednocení:

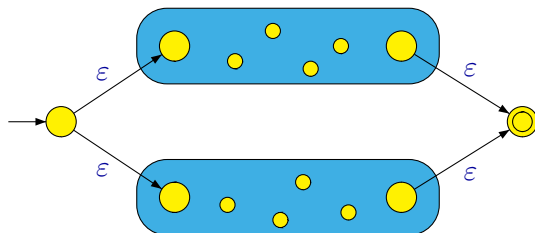


# Převod regulárního výrazu na konečný automat

Automaty pro elementární výrazy:

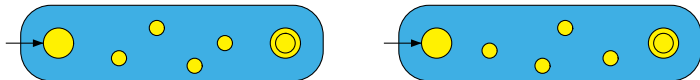


Konstrukce pro sjednocení:



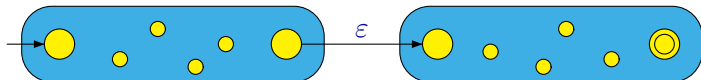
# Převod regulárního výrazu na konečný automat

Konstrukce pro zřetězení:



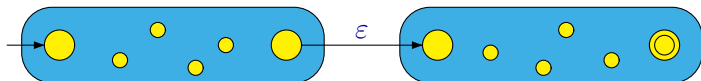
# Převod regulárního výrazu na konečný automat

Konstrukce pro zřetězení:

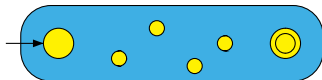


# Převod regulárního výrazu na konečný automat

Konstrukce pro zřetězení:

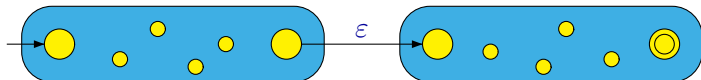


Konstrukce pro iteraci:

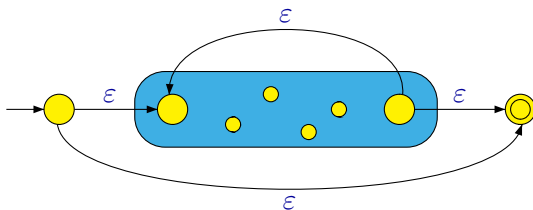


# Převod regulárního výrazu na konečný automat

Konstrukce pro zřetězení:



Konstrukce pro iteraci:



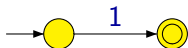


**Příklad:** Konstrukce automatu pro výraz  $((0 + 1) \cdot 1)^*$ :

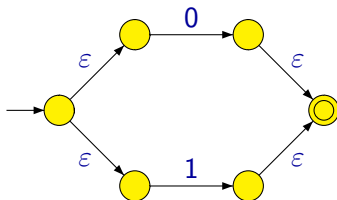
**Příklad:** Konstrukce automatu pro výraz  $((0 + 1) \cdot 1)^*$ :



**Příklad:** Konstrukce automatu pro výraz  $((0 + 1) \cdot 1)^*$ :

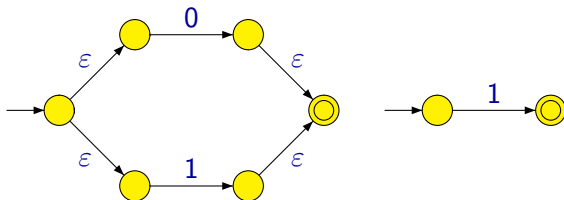


**Příklad:** Konstrukce automatu pro výraz  $((0 + 1) \cdot 1)^*$ :



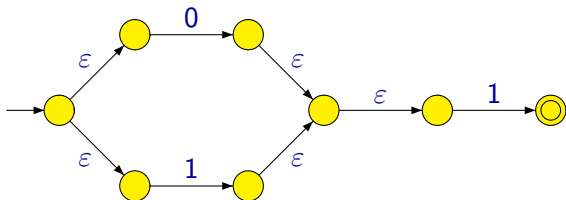
# Převod regulárního výrazu na konečný automat

**Příklad:** Konstrukce automatu pro výraz  $((0 + 1) \cdot 1)^*$ :

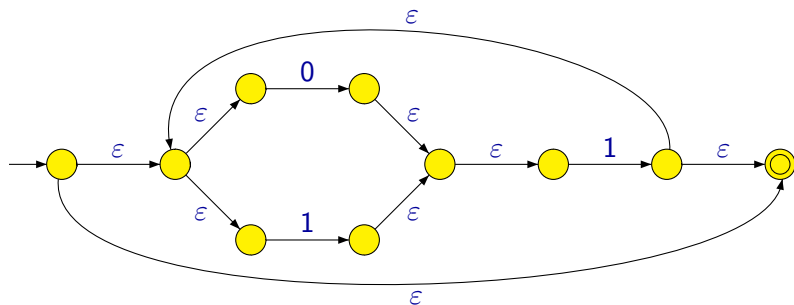


# Převod regulárního výrazu na konečný automat

**Příklad:** Konstrukce automatu pro výraz  $((0 + 1) \cdot 1)^*$ :



**Příklad:** Konstrukce automatu pro výraz  $((0 + 1) \cdot 1)^*$ :



# Převod regulárního výrazu na konečný automat

Pokud se výraz  $\alpha$  skládá z  $n$  znaků (nepočítáme-li závorky), má výsledný automat:

- nejvýše  $2n$  stavů,
- nejvýše  $4n$  přechodů.

**Poznámka:** Převodem ze zobecněného nedeterministického automatu na deterministický však může počet stavů vzrůst exponenciálně, tj. výsledný automat pak může mít až  $2^{2n} = 4^n$  stavů.



## Tvrzení

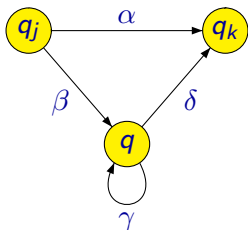
Každý regulární jazyk je možné popsat nějakým regulárním výrazem.

**Důkaz:** Stačí ukázat, jak pro libovolný konečný automat  $A$  zkonstruovat regulární výraz  $\alpha$  takový, že  $[\alpha] = L(A)$ .

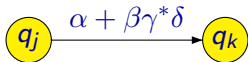
- $A$  upravíme tak, aby měl právě jeden počáteční a právě jeden koncový stav.
- Budeme postupně odebírat jednotlivé stavy.
- Přejchody budou označeny regulárními výrazy.
- Zbude automat se dvěma stavy – počátečním a koncovým, a jedním přechodem ohodnoceným výsledným regulárním výrazem.

# Převod konečného automatu na regulární výraz

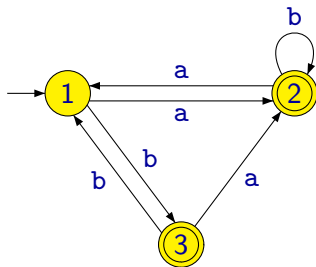
Hlavní myšlenka: Při odstraňování stavu  $q$  nahradit pro každou dvojici zbylých stavů  $q_j$ ,  $q_k$  cestu z  $q_j$  do  $q_k$  vedoucí přes  $q$ .



Po odstranění stavu  $q$ :

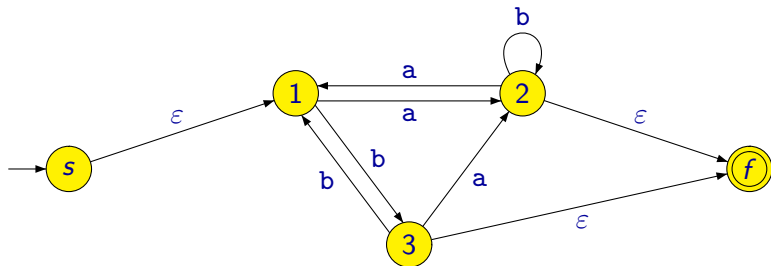


## Příklad:



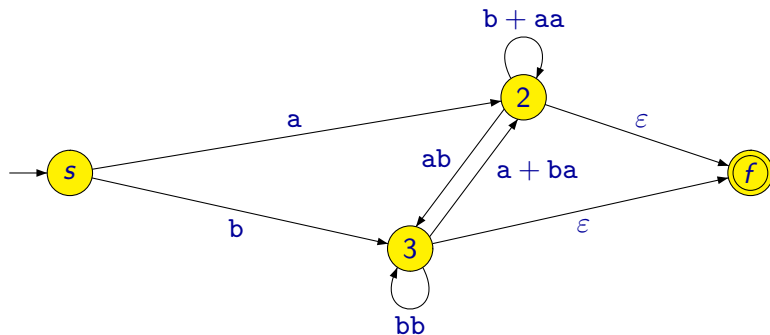
# Převod konečného automatu na regulární výraz

**Příklad:**



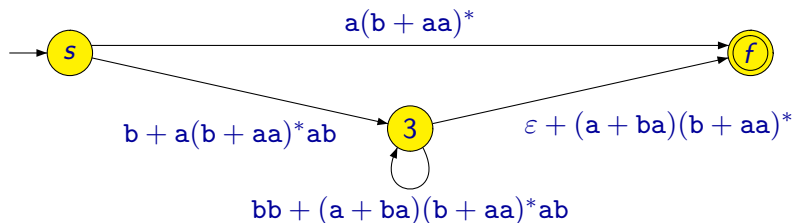
# Převod konečného automatu na regulární výraz

Příklad:



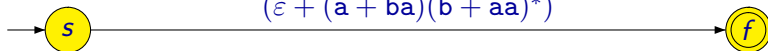
# Převod konečného automatu na regulární výraz

## Příklad:



## Příklad:

$$\begin{aligned} & a(b + aa)^* + \\ & (b + a(b + aa)^* ab) \\ & (bb + (a + ba)(b + aa)^* ab)^* \\ & (\varepsilon + (a + ba)(b + aa)^*) \end{aligned}$$



## Věta

Jazyk je regulární právě tehdy, když je ho možné popsat regulárním výrazem.



Regulární výrazy jsou používány v celé řadě různých nástrojů.

## Příklady:

- Knihovna `regex` jazyka C.
- Package `java.util.regex` v jazyce Java.
- Modul `re` v jazyce Python.
- Programovací jazyk Perl.
- Unixové utility pro zpracování textových souborů `grep`, `sed` a `awk`.
- Generátory lexikálních analyzátorů `lex` a `flex`.
- Textové editory (`vi`, `vim`, `emacs`, ...).

# V praxi používané regulární výrazy

Běžně používaná syntaxe (mezi jednotlivými nástroji jsou však drobné rozdíly):

- $\cdot$  ... zastupuje libovolný znak
- $\alpha\beta$  ... zřetězení  $\alpha$  a  $\beta$
- $\alpha|\beta$  ... sjednocení  $\alpha$  a  $\beta$
- $\alpha^*$  ... iterace  $\alpha$
- $\alpha^+$  ... totéž, co  $\alpha\alpha^*$
- $\alpha?$  ... totéž, co  $\alpha + \epsilon$
- $\alpha\{m\}$  ... totéž co  $m$  krát  $\alpha$
- $\alpha\{m,n\}$  ...  $\alpha$  minimálně  $m$  krát, maximálně  $n$  krát
- $(\alpha)$  ... závorky

# V praxi používané regulární výrazy

- [xyz] ... libovolný ze znaků **x**, **y**, **z**
- [^xyz] ... libovolný znak, kromě **x**, **y**, **z**
- [a-f] ... libovolný ze znaků **a**, **b**, **c**, **d**, **e**, **f**
- ^ ... začátek řádku
- \$ ... konec řádku
- \c ... znak **c**

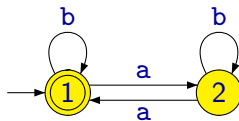
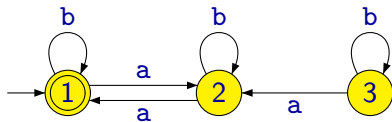
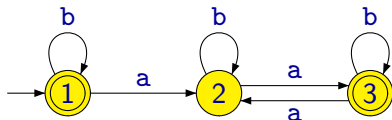
**Příklad:** správně vytvořená e-mailová adresa

```
^[a-zA-Z0-9\.\-]+@[a-zA-Z0-9\.\-]+\.[a-zA-Z]{2,4}$
```

**Poznámka:** Podrobnější informace najdete například v seriálu „Regulární výrazy“ autora Pavla Satrapy na

<http://www.root.cz/serialy/regularni-vyrazy/>

# Ekvivalence automatů



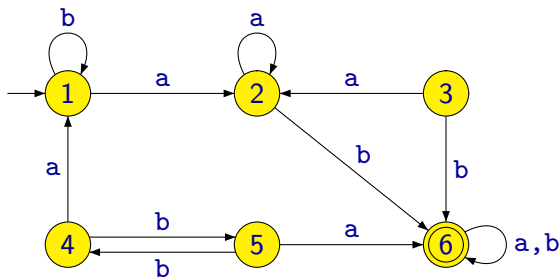
- Všechny 3 automaty přijímají jazyk všech slov se sudým počtem  $a$ -ček
- Nejvýhodnější je pro nás poslední z nich - má nejméně stavů

## Definice

O konečných automatech  $A_1, A_2$  řekneme, že jsou **ekvivalentní**, jestliže  $L(A_1) = L(A_2)$ .

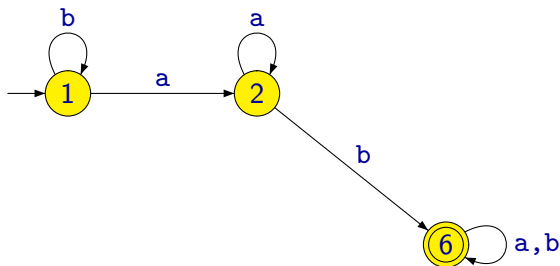
- Existuje nějaký vhodný algoritmus zjišťující, jestli jsou dva automaty ekvivalentní?
- Můžeme zjistit, jestli k danému automatu neexistuje nějaký ekvivalentní menší (s menším počtem stavů)?
- Když o nějakém automatu víme, že menší ekvivalentní už neexistuje, může existovat jiný stejně velký ekvivalentní automat?

# Nedosažitelné stavy automatu



- Automat přijímá jazyk  $L = \{w \in a, b^* \mid w \text{ obsahuje podslovo } ab\}$
- Pro žádnou posloupnost vstupních symbolů se automat nedostane do stavů 3, 4, nebo 5

# Nedosažitelné stavy automatu



- Automat přijímá jazyk  $L = \{w \in a, b^* \mid w \text{ obsahuje podslovo } ab\}$
- Pro žádnou posloupnost vstupních symbolů se automat nedostane do stavů 3, 4, nebo 5
- Pokud tyto stavy odstraníme, pořád automat přijímá stejný jazyk  $L = \{w \in a, b^* \mid w \text{ obsahuje podslovo } ab\}$

# Nedosažitelné stavy automatu

## Definice

Stav  $q$  deterministického konečného automatu  $\mathcal{A}$  je **dosažitelný slovem**  $w$ , pokud se výpočet  $\mathcal{A}$  po přečtení celého slova  $w$  zastaví ve stavu  $q$ .

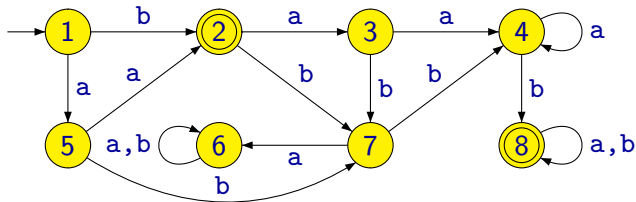
## Definice

Stav  $q$  deterministického konečného automatu  $\mathcal{A}$  je **dosažitelný** pokud existuje nějaké slovo, kterým je stav dosažitelný. V opačném případě stav nazýváme **nedosažitelný**.

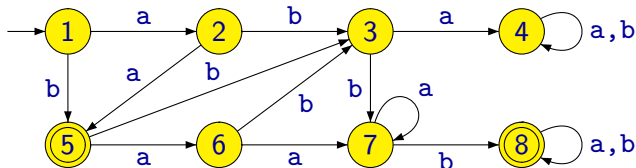
- Do nedosažitelných stavů nevede v grafu automatu žádná orientovaná cesta z počátečního stavu
- Nedosažitelné stavy můžeme z automatu odstranit (spolu se všemi přechody vedoucími z nich). Jazyk přijímaný automatem se nezmění.



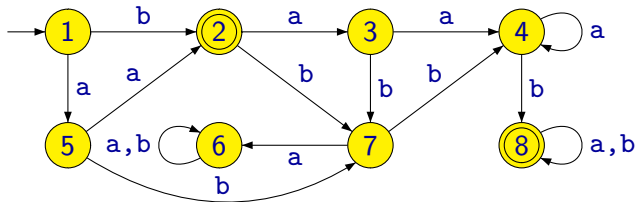
# Normovaný tvar automatu



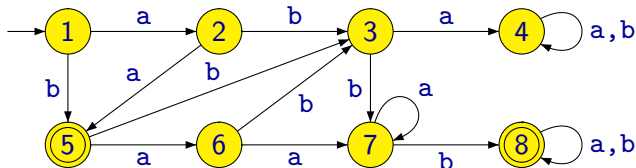
$\mathcal{A}_1$	$a$	$b$
→ 1	5	2
← 2	3	7
3	4	7
4	4	8
5	2	7
6	6	6
7	6	4
← 8	8	8



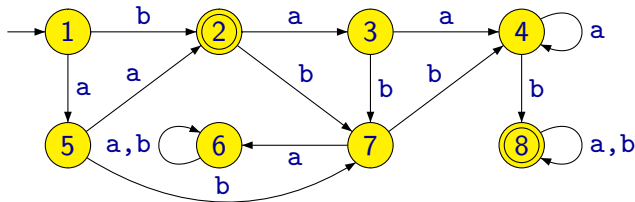
# Normovaný tvar automatu



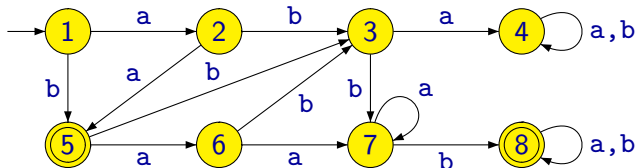
$\mathcal{A}_2$	$a$	$b$
→ 1	2	5
2	5	3
3	4	7
4	4	4
← 5	6	3
6	7	3
7	7	8
← 8	8	8



# Normovaný tvar automatu

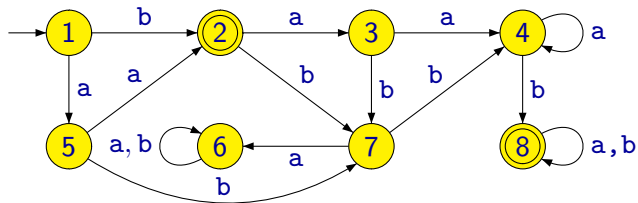


$\mathcal{A}_2$	a	b
→ 1	2	5
2	5	3
3	4	7
4	4	4
← 5	6	3
6	7	3
7	7	8
← 8	8	8

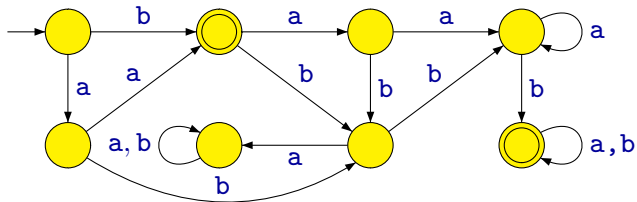


- Jak poznáme, že jsou automaty ekvivalentní, když se liší přechodová funkce?
- Automaty na obrázku můžou čísla 1 – 8 pojmenovat 8! způsoby
- Chtěli bychom jednoznačně vybrat jedno z možných pojmenování

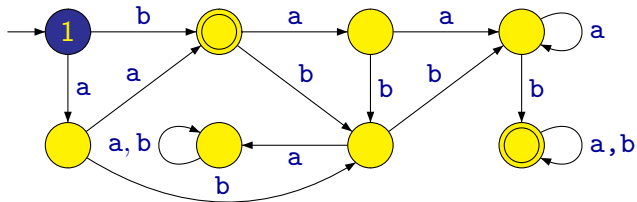
# Normovaný tvar automatu



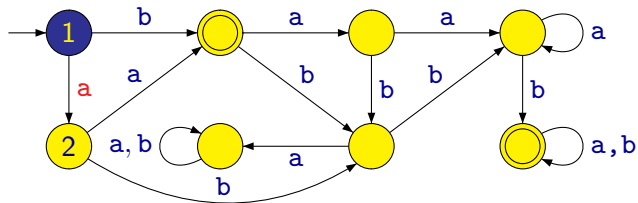
# Normovaný tvar automatu



# Normovaný tvar automatu



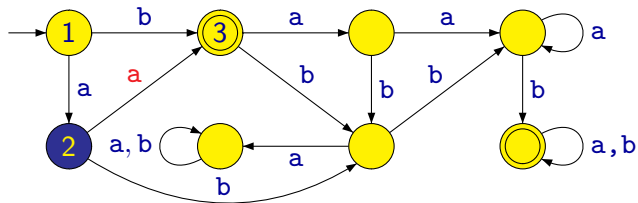
# Normovaný tvar automatu



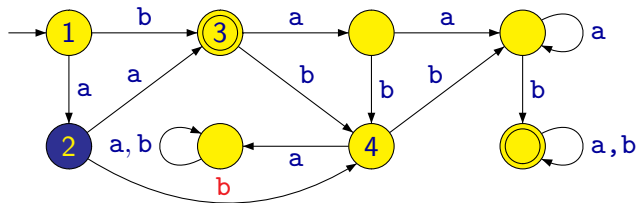




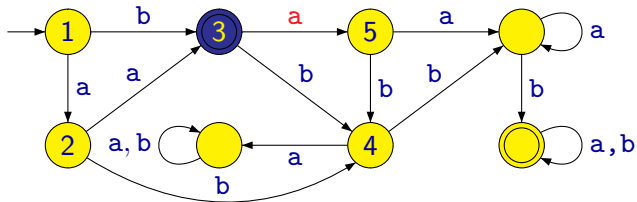
# Normovaný tvar automatu



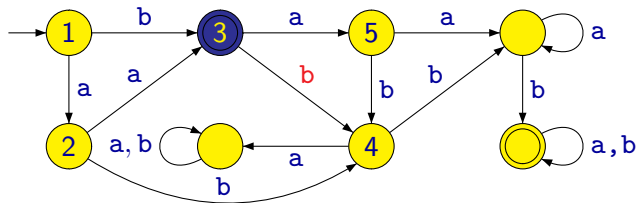
# Normovaný tvar automatu



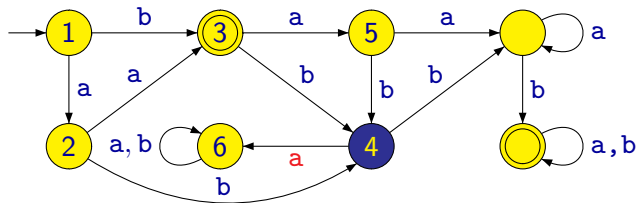
# Normovaný tvar automatu



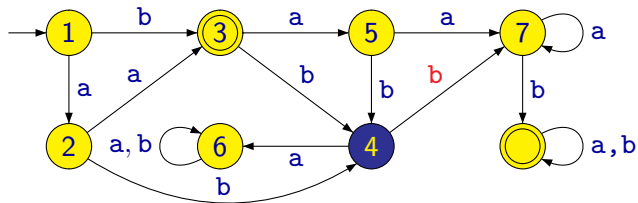
# Normovaný tvar automatu



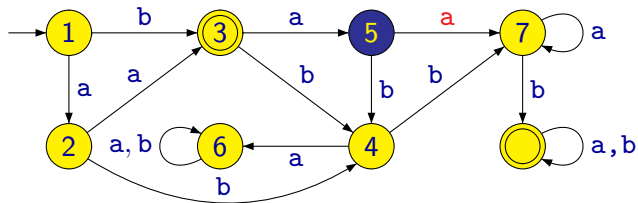
# Normovaný tvar automatu



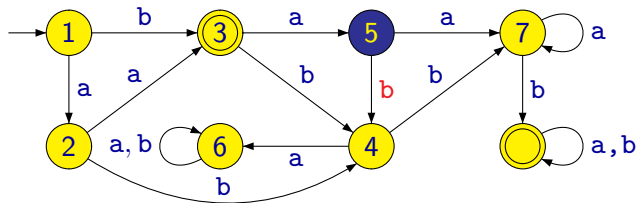
# Normovaný tvar automatu



# Normovaný tvar automatu

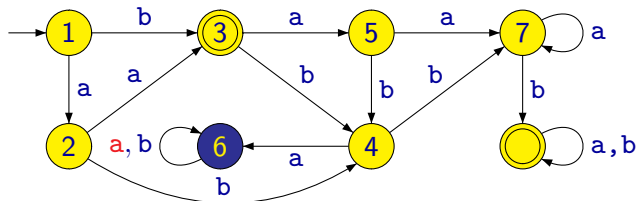


# Normovaný tvar automatu

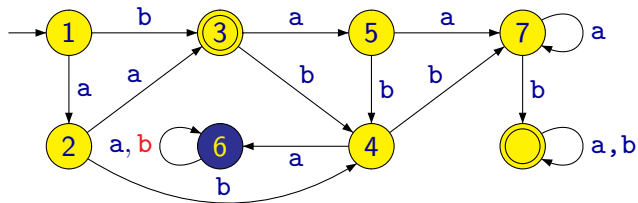




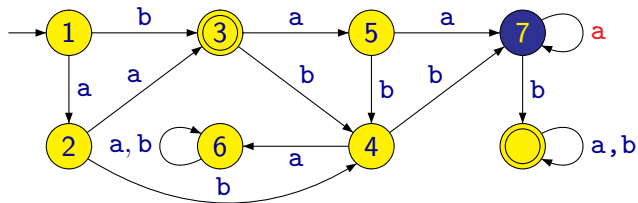
# Normovaný tvar automatu



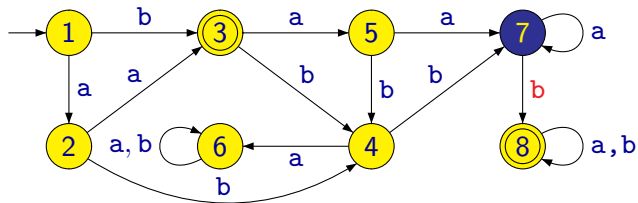
# Normovaný tvar automatu



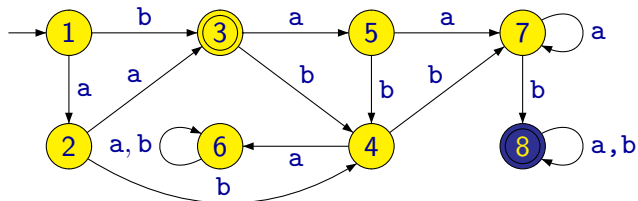
# Normovaný tvar automatu



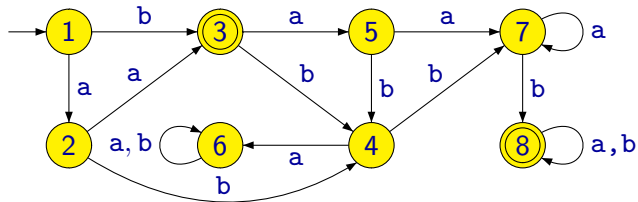
# Normovaný tvar automatu



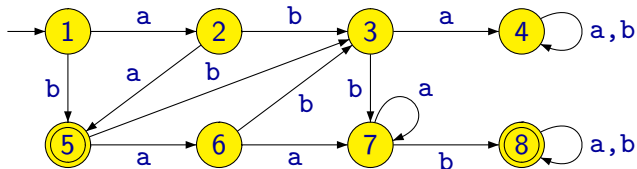
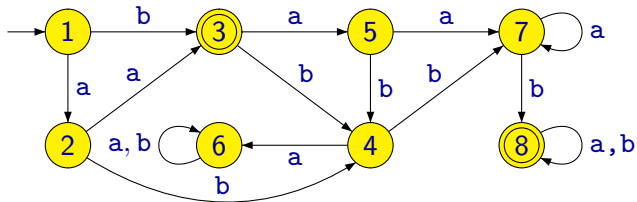
# Normovaný tvar automatu



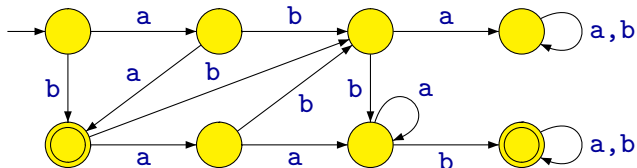
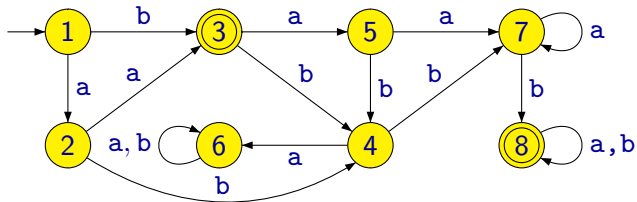
# Normovaný tvar automatu



# Normovaný tvar automatu

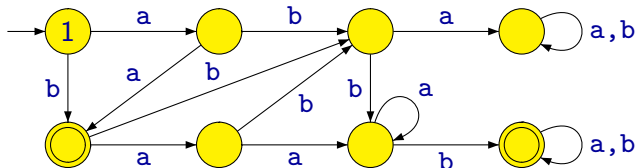
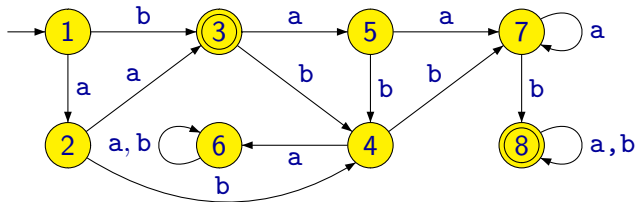


# Normovaný tvar automatu

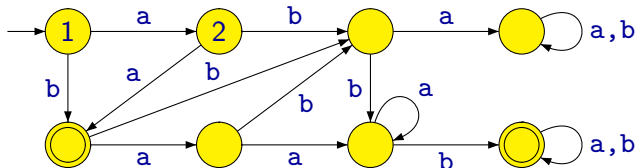
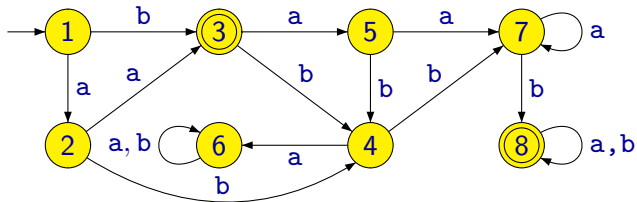




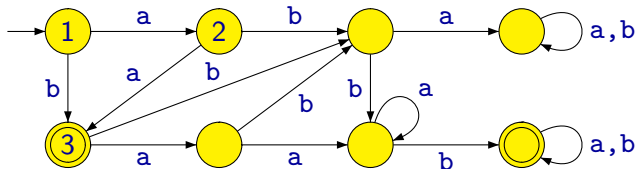
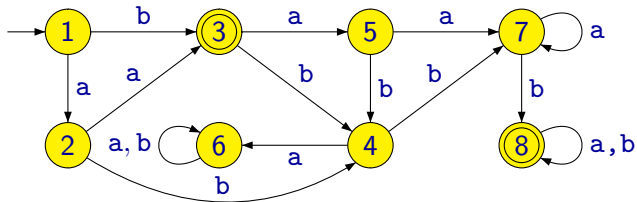
# Normovaný tvar automatu



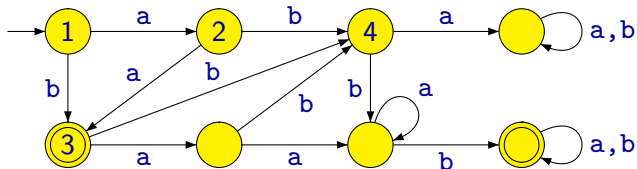
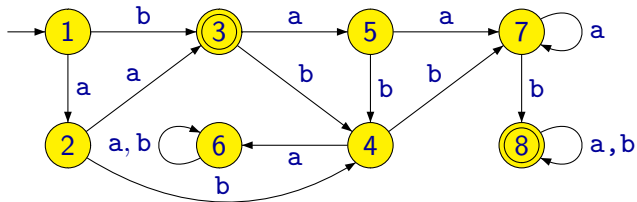
# Normovaný tvar automatu



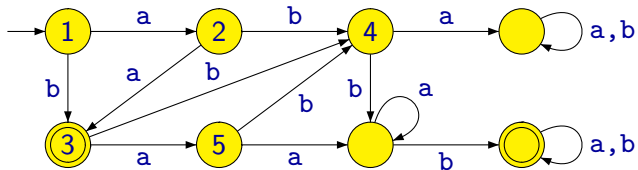
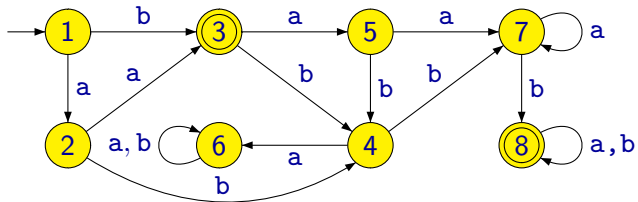
# Normovaný tvar automatu



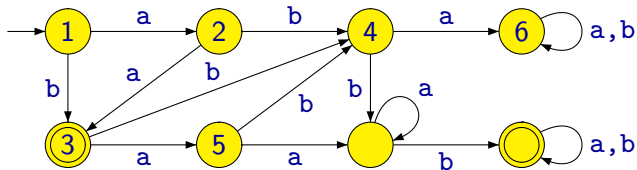
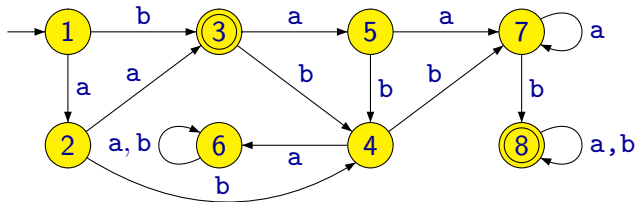
# Normovaný tvar automatu



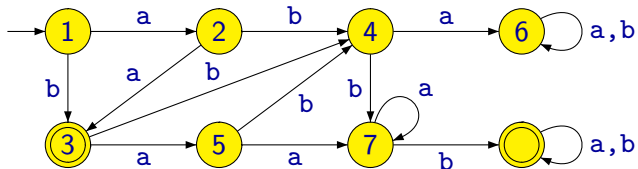
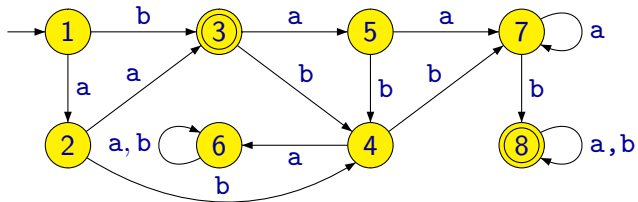
# Normovaný tvar automatu



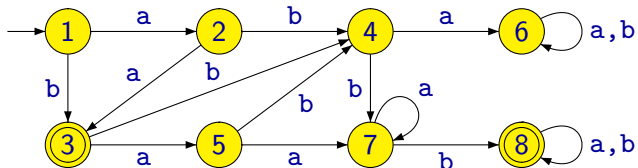
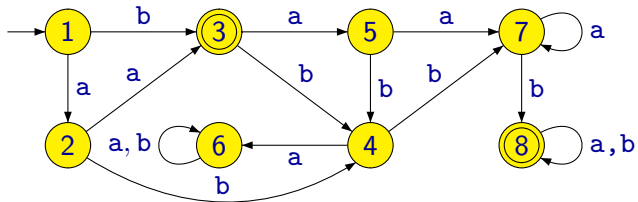
# Normovaný tvar automatu



# Normovaný tvar automatu

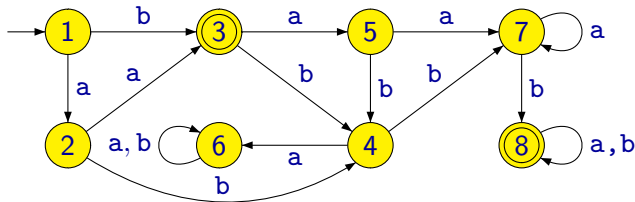


# Normovaný tvar automatu

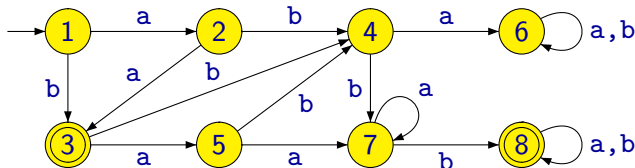




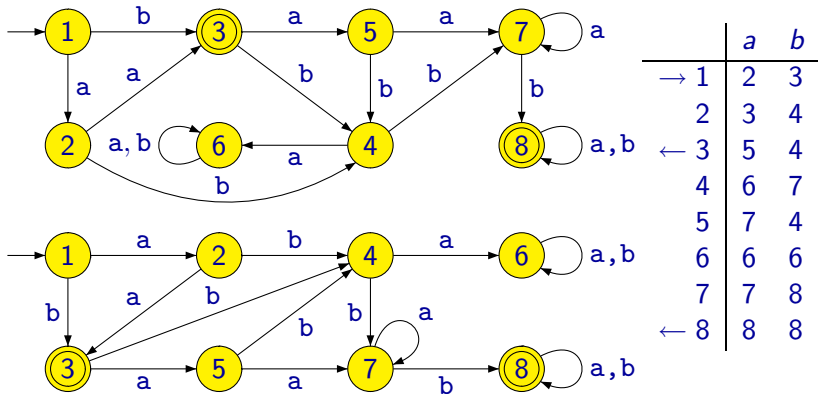
# Normovaný tvar automatu



	a	b
→ 1	2	3
2	3	4
← 3	5	4
4	6	7
5	7	4
6	6	6
7	7	8
← 8	8	8



# Normovaný tvar automatu



- Nyní už je přechodová funkce stejná pro oba automaty, stejné jsou i koncové a počáteční stav
- Automaty jsou tedy nejen ekvivalentní, ale dokonce stejné
- Říkáme, že automaty jsou v normovaném tvaru

## Definice

Mějme abecedu  $\Sigma$  spolu s úplným ostrým uspořádáním  $<$  nad znaky abecedy. Pro slova nad abecedou  $\Sigma$  definujeme úplné ostré uspořádání  $<_L$  následovně (pro všechna možná slova  $u, v \in \Sigma^*$ ):

- Pokud  $|u| < |v|$ , potom  $u <_L v$
- Pokud  $|u| = |v|$  a  $u$  je lexikograficky menší než  $v$ , potom  $u <_L v$

## Definice

Mějme abecedu  $\Sigma$  spolu s úplným ostrým uspořádáním  $<$  nad znaky abecedy. Pro slova nad abecedou  $\Sigma$  definujeme úplné ostré uspořádání  $<_L$  následovně (pro všechna možná slova  $u, v \in \Sigma^*$ ):

- Pokud  $|u| < |v|$ , potom  $u <_L v$
- Pokud  $|u| = |v|$  a  $u$  je lexikograficky menší než  $v$ , potom  $u <_L v$

**Příklad:** Nad abecedou  $\Sigma = \{a, b\}$  uvažujeme běžné uspořádání  $a < b$ . Potom platí  $\varepsilon <_L a <_L b <_L aa <_L bb <_L aba <_L baa <_L aaaa <_L bbbbbb$ .

## Definice

Mějme abecedu  $\Sigma$  spolu s úplným ostrým uspořádáním  $<$  nad znaky abecedy. Pro slova nad abecedou  $\Sigma$  definujeme úplné ostré uspořádání  $<_L$  následovně (pro všechna možná slova  $u, v \in \Sigma^*$ ):

- Pokud  $|u| < |v|$ , potom  $u <_L v$
- Pokud  $|u| = |v|$  a  $u$  je lexikograficky menší než  $v$ , potom  $u <_L v$

**Příklad:** Nad abecedou  $\Sigma = \{a, b\}$  uvažujeme běžné uspořádání  $a < b$ . Potom platí  $\varepsilon <_L a <_L b <_L aa <_L bb <_L aba <_L baa <_L aaaa <_L bbbbbb$ .

**Příklad:** Nad abecedou  $\Sigma = \{1, 2, 3\}$  uvažujeme běžné uspořádání  $1 < 2 < 3$ . Potom platí  $\varepsilon <_L 1 <_L 3 <_L 31 <_L 33 <_L 111 <_L 321$ .

## Definice

Mějme abecedu  $\Sigma$  spolu s úplným ostrým uspořádáním  $<$  nad znaky abecedy. Pro slova nad abecedou  $\Sigma$  definujeme úplné ostré uspořádání  $<_L$  následovně (pro všechna možná slova  $u, v \in \Sigma^*$ ):

- Pokud  $|u| < |v|$ , potom  $u <_L v$
- Pokud  $|u| = |v|$  a  $u$  je lexikograficky menší než  $v$ , potom  $u <_L v$

**Příklad:** Nad abecedou  $\Sigma = \{a, b\}$  uvažujeme běžné uspořádání  $a < b$ . Potom platí  $\varepsilon <_L a <_L b <_L aa <_L bb <_L aba <_L baa <_L aaaa <_L bbbbb$ .

**Příklad:** Nad abecedou  $\Sigma = \{1, 2, 3\}$  uvažujeme běžné uspořádání  $1 < 2 < 3$ . Potom platí  $\varepsilon <_L 1 <_L 3 <_L 31 <_L 33 <_L 111 <_L 321$ .

**Příklad:** Nad abecedou  $\Sigma = \{0, 1, 2\}$  uvažujeme běžné uspořádání  $0 < 1 < 2$ . Potom platí  $\varepsilon <_L 1 <_L 2 <_L 01 <_L 03 <_L 111 <_L 0021$ .

## Definice

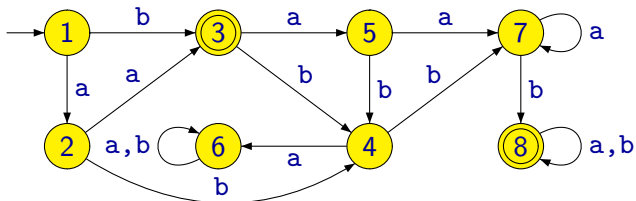
Mějme deterministický konečný automat  $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$  bez nedosažitelných stavů a uspořádání prvků abecedy  $\Sigma$  (které indukuje uspořádání  $<_L$  na  $\Sigma^*$ ).

Označme pro každý stav  $i \in \{1, 2, \dots, n\}$  symbolem  $u_i$  nejmenší slovo podle uspořádání  $<_L$  takové, že pro něj platí  $\delta(1, u_i) = i$ .

Potom řekneme, že  $\mathcal{A}$  je v normovaném tvaru, pokud

- $Q = \{1, 2, 3, \dots, n\}$  pro nějaké  $n \geq 1$
- 1 je počáteční stav
- Pro každou dvojici stavů  $i, j \in \{1, 2, \dots, n\}$  takovou, že  $i < j$ , platí  $u_i <_L u_j$ .

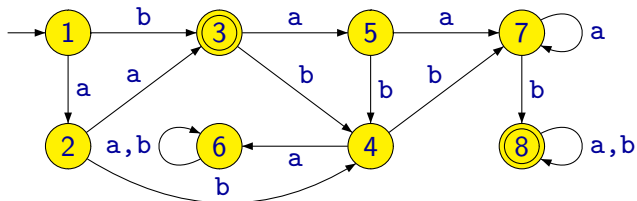
# Normovaný tvar automatu



- Do stavu 4 se dostaneme slovy  $ab, bb, aab, bab$  tedy  $u_4 = ab$
- Do stavu 5 se dostaneme slovy  $ba, aaa$  tedy  $u_5 = ba$
- $ab <_L ba$  proto jsou stavy 4, 5 označeny v tomto pořadí



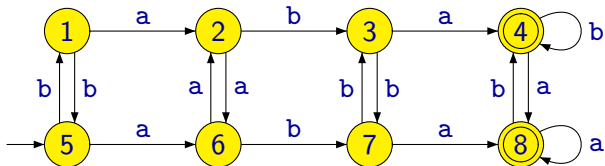
# Normovaný tvar automatu



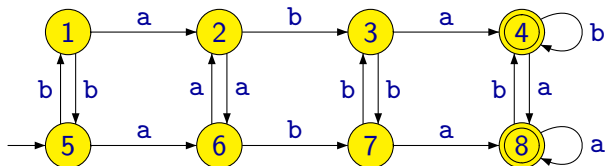
- Do stavu 4 se dostaneme slovy  $ab, bb, aab, bab$  tedy  $u_4 = ab$
- Do stavu 5 se dostaneme slovy  $ba, aaa$  tedy  $u_5 = ba$
- $ab <_L ba$  proto jsou stavy 4, 5 označeny v tomto pořadí
- Pro ostatní stavy jsou nejmenší slova následující:

$$\begin{array}{ll} u_1 = \varepsilon & u_5 = ba \\ u_2 = a & u_6 = aba \\ u_3 = b & u_7 = abb \\ u_4 = ab & u_8 = abbb \end{array}$$

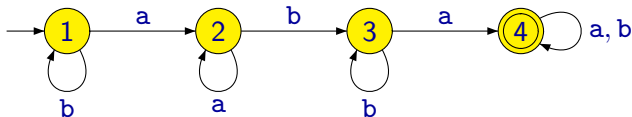
# Minimalizace konečného automatu



# Minimalizace konečného automatu

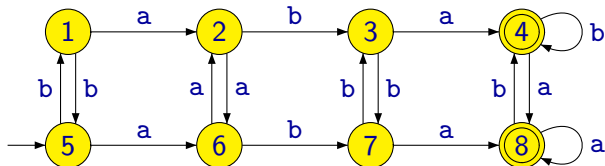


- Automat přijímá taková slova, z nichž by vpuštěním některých znaků zůstalo *aba*
- Existuje automat s méně stavy, který přijímá stejný jazyk?



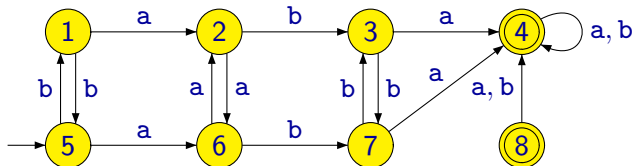
- Automat přijímá taková slova, z nichž by vypuštěním některých znaků zůstalo *aba*
- Existuje automat s méně stavy, který přijímá stejný jazyk?
- Existuje nějaký algoritmus, který by našel ekvivalentní automat s nejmenším možným počtem stavů?

# Minimalizace konečného automatu



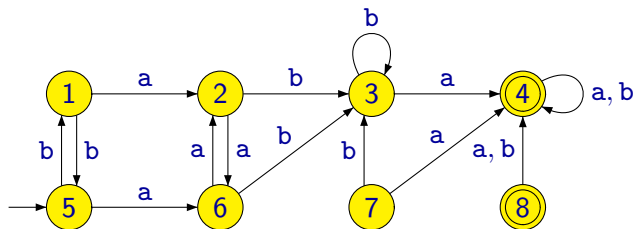
- Když se automat při běhu dostane do stavu 4, přijme už jistě dané slovo, ať už bude pokračovat jakkoliv
- Když se automat při běhu dostane do stavu 8, přijme také už jistě dané slovo
- Je tedy jedno, jestli jde automat některým přechodem do 4 nebo 8

# Minimalizace konečného automatu



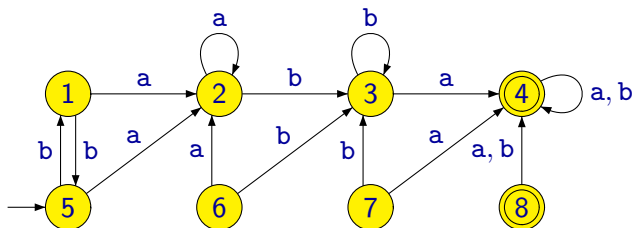
- Když se automat při běhu dostane do stavu 3, přijme každé slovo, které bude v pokračování obsahovat alespoň jedno *a*
- Když se automat při běhu dostane do stavu 7, přijme každé slovo, které bude v pokračování obsahovat alespoň jedno *a*
- Je tedy jedno, jestli jde automat některým přechodem do 3 nebo 7

# Minimalizace konečného automatu



- Když se automat při běhu dostane do stavu 2, přijme každé slovo, které bude v pokračování obsahovat alespoň jedno  $b$  a jedno  $a$  v tomto pořadí
- Když se automat při běhu dostane do stavu 6, přijme stejná slova
- Je tedy jedno, jestli jde automat některým přechodem do 2 nebo 6

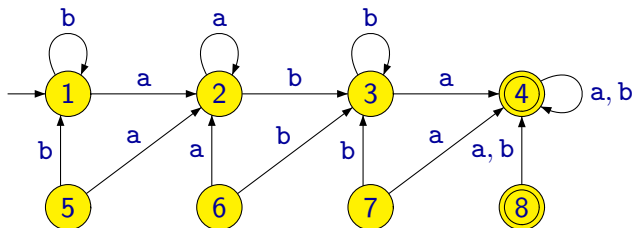
# Minimalizace konečného automatu



- Ve stavu 1 i 5 automat přijme jakékoliv slovo, které patří do jazyka takových slov, z nichž by vypuštěním některých znaků zůstalo *aba*
- Je tedy jedno, jestli jde automat některým přechodem do 1 nebo 5, a je jedno, ve kterém z nich začne

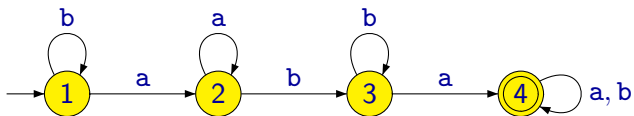


# Minimalizace konečného automatu



- Stavy 5, 6, 7, 8 jsou teď už nedosažitelné a můžeme je odstranit

# Minimalizace konečného automatu



## Definice

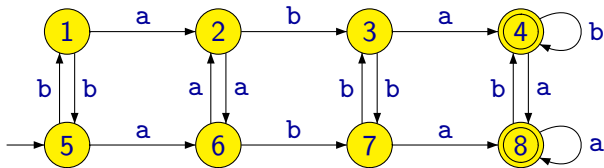
Pro každý stav  $q$  automatu  $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$  definujeme  $L(q) = L(\mathcal{A}_q)$ , kde  $\mathcal{A}_q = (Q, \Sigma, \delta, q, F)$

## Definice

Stavy  $q_1, q_2$  automatu  $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$  nazýváme **jazykově ekvivalentní** nebo zkráceně **ekvivalentní**, jestliže  $L(q_1) = L(q_2)$ .

- Jsou-li dva stavy  $q_1, q_2$  automatu ekvivalentní, můžeme jeden vypustit a všechny šipky do něj směřující přeměřovat do druhého. Pokud byl vypouštěný stav počáteční, bude počáteční ten druhý.
- Dvojice vzájemně ekvivalentních stavů můžeme hledat rychlým (polynomiálním) algoritmem

# Minimalizace konečného automatu



	a	b
1	2	5
2	6	3
3	4	7
← 4	8	4
→ 5	6	1
6	2	7
7	8	3
← 8	8	4

# Minimalizace konečného automatu

	<i>a</i>	<i>b</i>
1	2	5
2	6	3
3	4	7
← 4	8	4
→ 5	6	1
6	2	7
7	8	3
← 8	8	4

- Rozdělíme stavy do dvou skupin na přijímající a nepřijímající. Je jisté, že přijímající stav nikdy není ekvivalentní s nepřijímajícím.

# Minimalizace konečného automatu

	a	b
1	2	5
2	6	3
3	4	7
← 4	8	4
→ 5	6	1
6	2	7
7	8	3
← 8	8	4

	a	b
I 1	2	5
2	6	3
3	4	7
→ 5	6	1
6	2	7
7	8	3
<hr/>		
II ← 4	8	4
← 8	8	4

- Rozdělíme stavy do dvou skupin na přijímající a nepřijímající. Je jisté, že přijímající stav nikdy není ekvivalentní s nepřijímajícím.
- Do tabulky si, místo přechodů do konkrétních stavů, vyznačíme skupinu, do které přecházíme.

# Minimalizace konečného automatu

	a	b
1	2	5
2	6	3
3	4	7
← 4	8	4
→ 5	6	1
6	2	7
7	8	3
← 8	8	4

	a	b
1	2	5
2	6	3
3	4	7
→ 5	6	1
6	2	7
7	8	3
← 4	8	4
← 8	8	4

	a	b
1		
2		
3		
→ 5		
6		
7		
← 4		
← 8		

- Rozdělíme stavy do dvou skupin na přijímající a nepřijímající. Je jisté, že přijímající stav nikdy není ekvivalentní s nepřijímajícím.
- Do tabulky si, místo přechodů do konkrétních stavů, vyznačíme skupinu, do které přecházíme.

# Minimalizace konečného automatu

	<i>a</i>	<i>b</i>
1	2	5
2	6	3
3	4	7
← 4	8	4
→ 5	6	1
6	2	7
7	8	3
← 8	8	4

	<i>a</i>	<i>b</i>
1		
2		
3		
→ 5		
6		
7		
← 4		
← 8		

- Pokud se liší řádky ve stejné skupině, skupinu rozdělíme na menší, kde se lišit nebudou
- Např. 3 se od 1 liší, protože z 1 přejde *a*-čkem do skupiny, která následně nepřijímá prázdné slovo, z 3 přejde *a*-čkem do skupiny, která následně přijímá prázdné slovo.



# Minimalizace konečného automatu

	<i>a</i>	<i>b</i>
1	2	5
2	6	3
3	4	7
← 4	8	4
→ 5	6	1
6	2	7
7	8	3
← 8	8	4

	<i>a</i>	<i>b</i>
1		
2		
3		
→ 5		
6		
7		
<hr/>		
← 4		
← 8		

	<i>a</i>	<i>b</i>
1		
2		
→ 5		
6		
<hr/>		
3		
7		
<hr/>		
← 4		
← 8		

- Pokud se liší řádky ve stejné skupině, skupinu rozdělíme na menší, kde se lišit nebudou
- Např. 3 se od 1 liší, protože z 1 přejde *a*-čkem do skupiny, která následně nepřijímá prázdné slovo, z 3 přejde *a*-čkem do skupiny, která následně přijímá prázdné slovo.
- Musíme znovu vyplnit tabulku, protože se změnily skupiny

# Minimalizace konečného automatu

	<i>a</i>	<i>b</i>
1	2	5
2	6	3
3	4	7
← 4	8	4
→ 5	6	1
6	2	7
7	8	3
← 8	8	4

		<i>a</i>	<i>b</i>
I	1	I	I
	2	I	II
	→ 5	I	I
	6	I	II
II	3	III	II
	7	III	II
III	← 4	III	III
	← 8	III	III

- Teď se 2 se od 1 liší, protože z 1 přejde *b*-čkem do jiné skupiny než z 2. Tedy z každého přijme jiná slova začínající *b*-čkem a nemohou být ekvivalentní.
- Takže zase skupinu I rozdělíme

# Minimalizace konečného automatu

	a	b
1	2	5
2	6	3
3	4	7
← 4	8	4
→ 5	6	1
6	2	7
7	8	3
← 8	8	4

	a	b
I 1	I	I
2	I	II
→ 5	I	I
6	I	II
II 3	III	II
7	III	II
III ← 4	III	III
← 8	III	III

	a	b
I 1	I	I
→ 5	I	I
II 2	I	II
6	I	II
III 3	III	II
7	III	II
IV ← 4	III	III
← 8	III	III

- Teď se 2 se od 1 liší, protože z 1 přejde  $b$ -čkem do jiné skupiny než z 2. Tedy z každého přijme jiná slova začínající  $b$ -čkem a nemohou být ekvivalentní.
- Takže zase skupinu I rozdělíme
- Musíme znovu vyplnit tabulku, protože se změnily skupiny

# Minimalizace konečného automatu

	<i>a</i>	<i>b</i>
1	2	5
2	6	3
3	4	7
← 4	8	4
→ 5	6	1
6	2	7
7	8	3
← 8	8	4

	<i>a</i>	<i>b</i>
I	1	I
→ 5	II	I
II	2	III
6	II	III
III	3	IV
7	IV	III
IV	← 4	IV
← 8	IV	IV

- Teď už se žádná skupina nerozdělí, takže algoritmus končí
- Stavy, které jsou v jedné skupině, jsou ekvivalentní. Můžeme nechat kterýkoliv z nich a ostatní odstranit
- Šipky vedoucí do odstraňovaných stavů povedou do toho, který ze skupiny necháme
- Pokud skupina obsahovala počáteční stav, bude ponechaný zástupce počáteční

# Minimalizace konečného automatu

	a	b
1	2	5
2	6	3
3	4	7
← 4	8	4
→ 5	6	1
6	2	7
7	8	3
← 8	8	4

	a	b
I 1	II	I
→ 5	II	I
II 2	II	III
6	II	III
III 3	IV	III
7	IV	III
IV ← 4	IV	IV
← 8	IV	IV

	a	b
→ I	II	I
II	II	III
III	IV	III
← IV	IV	IV

- Teď už se žádná skupina nerozdělí, takže algoritmus končí
- Stavy, které jsou v jedné skupině, jsou ekvivalentní. Můžeme nechat kterýkoliv z nich a ostatní odstranit
- Šipky vedoucí do odstraňovaných stavů povedou do toho, který ze skupiny necháme
- Pokud skupina obsahovala počáteční stav, bude ponechaný zástupce počáteční

## Definice

Deterministický konečný automat nazveme **minimálním automatem**, jestliže neexistuje automat, který by s ním byl ekvivalentní a měl by menší počet stavů.

# Minimální konečný automat

## Definice

Deterministický konečný automat nazveme **minimálním automatem**, jestliže neexistuje automat, který by s ním byl ekvivalentní a měl by menší počet stavů.

## Lemma

*Mějme v automatu  $A$  dva stav  $q_1, q_2$  dva stavy  $q_1, q_2$  takové, že  $L(q_1) \neq L(q_2)$ . Necht' stav  $q_1$  je dosažitelný z počátečního stavu  $q_0$  slovem  $z_1$  a stav  $q_2$  je dosažitelný slovem  $z_2$ . Potom každý automat  $A'$  ekvivalentní s  $A$  musí být slovem  $z_1$  převeden do jiného stavu než slovem  $z_2$ .*

# Minimální konečný automat

## Definice

Deterministický konečný automat nazveme **minimálním automatem**, jestliže neexistuje automat, který by s ním byl ekvivalentní a měl by menší počet stavů.

## Lemma

Mějme v automatu  $A$  dva stav  $q_1, q_2$  dva stavy  $q_1, q_2$  takové, že  $L(q_1) \neq L(q_2)$ . Necht' stav  $q_1$  je dosažitelný z počátečního stavu  $q_0$  slovem  $z_1$  a stav  $q_2$  je dosažitelný slovem  $z_2$ . Potom každý automat  $A'$  ekvivalentní s  $A$  musí být slovem  $z_1$  převeden do jiného stavu než slovem  $z_2$ .

## Lemma

Automat, ve kterém je každý stav dosažitelný a pro každé dva jeho stavy  $q \neq q'$  platí  $L(q) \neq L(q')$ , je minimální.



## Lemma

*Existuje algoritmus, který pro zadané konečné automaty  $A_1, A_2$  rozhodne, zda jsou ekvivalentní, tj. zda  $L(A_1) = L(A_2)$ .*

## Důkaz:

- Je-li  $A_1$  nebo  $A_2$  nedeterministický, převedeme jej na deterministický
- Automaty  $A_1$  a  $A_2$  převedeme na minimální
- Minimální automaty převedeme do normovaného tvaru
- Pokud jsou vzniklé minimální automaty v normovaném tvaru stejné, byly původní automaty ekvivalentní. Pokud výsledné automaty nejsou stejné, nebyly původní automaty ekvivalentní.

# Neregulární jazyky

Ne všechny jazyky jsou regulární.

Existují jazyky, pro které neexistuje žádný konečný automat, který by je rozpoznával.

Příklady neregulárních jazyků:

- $L_1 = \{a^n b^n \mid n \geq 0\}$
- $L_2 = \{ww \mid w \in \{a, b\}^*\}$
- $L_3 = \{ww^R \mid w \in \{a, b\}^*\}$

**Poznámka:** Existence neregulárních jazyků vyplývá již z faktu, že automatů pracujících nad nějakou abecedou  $\Sigma$  je jen spočetně mnoho, zatímco jazyků nad abecedou  $\Sigma$  je nespočetně mnoho.

# Neregulární jazyky

Jak dokázat o nějakém jazyce  $L$ , že není regulární?

Jazyk není regulární, jestliže neexistuje (tj. není možné sestrojít) konečný automat, který by ho rozpoznával.

Jak ale dokázat, že něco neexistuje?

Jak dokázat o nějakém jazyce  $L$ , že není regulární?

Jazyk není regulární, jestliže neexistuje (tj. není možné sestrojít) konečný automat, který by ho rozpoznával.

Jak ale dokázat, že něco neexistuje?

**Odpověď:** Stačí ukázat, že jazyk  $L$  nemá nějakou vlastnost, kterou má každý jazyk rozpoznávaný nějakým konečným automatem.

Jak dokázat o nějakém jazyce  $L$ , že není regulární?

Jazyk není regulární, jestliže neexistuje (tj. není možné sestrojít) konečný automat, který by ho rozpoznával.

Jak ale dokázat, že něco neexistuje?

**Odpověď:** Stačí ukázat, že jazyk  $L$  nemá nějakou vlastnost, kterou má každý jazyk rozpoznávaný nějakým konečným automatem.

Dva základní postupy dokazování neregularity jazyků:

- využití tzv. pumping lemmatu
- využití Myhillovy-Nerodovy věty

## Tvrzení

Každý konečný jazyk je regulární.

**Důkaz:** Konečný jazyk s  $n$  slovy můžeme popsat regulárním výrazem  $w_1 + w_2 + \dots + w_n$ . Jazyk je tedy regulární.

## Tvrzení

Každý konečný jazyk je regulární.

**Důkaz:** Konečný jazyk s  $n$  slovy můžeme popsat regulárním výrazem  $w_1 + w_2 + \dots + w_n$ . Jazyk je tedy regulární.

## Tvrzení

Je-li jazyk nekonečný, obsahuje pro každou konstantu  $k \in \mathbb{N}$  nějaké slovo  $w$  takové, že  $|w| > k$

**Důkaz:** Uvažujeme jen konečnou abecedu. Pro každou konstantu je tedy jen konečně mnoho slov kratších než tato konstanta. Pokud má být jazyk nekonečný, musí být i slovo delší, než jakákoliv konstanta. Ve skutečnosti je slov delších než jakákoliv konstanta nekonečně mnoho.

# Pumping Lemma

Předpokládejme, že jazyk  $L$  je rozpoznáván nějakým konkrétním deterministickým automatem  $A$ , tj.  $L = L(A)$ .

Vezměme nyní nějaké libovolné slovo  $z \in L$ , kde  $z = a_1 a_2 \cdots a_k$ .

Protože automat  $A$  slovo  $z$  přijímá, musí tomuto slovu odpovídat určitý přijímající výpočet automatu, tj. posloupnost stavů:

$$q_0, q_1, q_2, \dots, q_{k-1}, q_k$$

délky  $k + 1$ , kde

- $q_0$  je počáteční stav
- $\delta(q_{i-1}, a_i) = q_i$  pro  $\forall i \in \{1, 2, \dots, k\}$
- $q_k$  je přijímající stav



Předpokládejme, že  $A$  má  $n$  stavů a že  $|z| \geq n$ .

Pak máme posloupnost délky minimálně  $n + 1$ , ve které se může vyskytovat maximálně  $n$  různých stavů.

Z toho plyne, že musí existovat alespoň jeden stav  $q$ , který se v této posloupnosti vyskytuje alespoň dvakrát.

Jde o aplikaci tzv. **holubníkového principu (pigeonhole principle)**.

## Holubníkový princip

Jestliže mám  $n + 1$  holubů rozmístěných do  $n$  klecí, pak jsou alespoň v jedné kleci minimálně dva holubi.

# Pumping Lemma

Řekněme, že opakující stav se vyskytuje na pozicích  $i, j$ , tj.  $q_i = q_j$ , kde  $i < j$ .

$$q_0, \dots, q_i, \dots, q_j, \dots, q_k$$

**Poznámka:** Zjevně můžeme najít taková  $i, j$ , že  $i < j \leq n$

Slovo  $z$  můžeme rozdělit na tři části:

$$\underbrace{a_1 \cdots a_i}_u \quad \underbrace{a_{i+1} \cdots a_j}_v \quad \underbrace{a_{j+1} \cdots a_k}_w$$

- $\delta^*(q_0, u) = q_i$
- $\delta^*(q_i, v) = q_j = q_i$
- $\delta^*(q_j, w) = q_k$

# Pumping Lemma

Vezměme nyní slova:

$$\begin{array}{c} \underbrace{a_1 \cdots a_j}_u \quad \underbrace{a_{j+1} \cdots a_k}_w \\ \\ \underbrace{a_1 \cdots a_j}_u \quad \underbrace{a_{i+1} \cdots a_j}_v \quad \underbrace{a_{i+1} \cdots a_j}_v \quad \underbrace{a_{j+1} \cdots a_k}_w \\ \\ \underbrace{a_1 \cdots a_j}_u \quad \underbrace{a_{i+1} \cdots a_j}_v \quad \underbrace{a_{i+1} \cdots a_j}_v \quad \underbrace{a_{i+1} \cdots a_j}_v \quad \underbrace{a_{j+1} \cdots a_k}_w \\ \\ \dots \end{array}$$

Je zřejmé, že  $A$  přijme každé z nich, vzhledem k tomu, že

- $\delta^*(q_0, u) = q_i$
- $\delta^*(q_i, v) = q_j = q_i$
- $\delta^*(q_j, w) = q_k, q_k \in F$

## Pumping Lemma

Jestliže jazyk  $L$  je regulární, pak existuje  $n$  takové, že každé slovo  $z \in L$  takové, že  $|z| \geq n$ , je možné rozdělit na podslova  $u, v, w$  taková, že  $z = uvw$ ,  $|uv| \leq n$ ,  $|v| \geq 1$  a pro všechna  $i \geq 0$  platí  $uv^i w \in L$ .

Formálně zapsáno:

Jestliže  $L$  je regulární, pak

$$(\exists n)(\forall z \text{ tž. } z \in L, |z| \geq n)(\exists u, v, w \text{ tž. } z = uvw, |uv| \leq n, |v| \geq 1) \\ (\forall i \geq 0) : uv^i w \in L$$

Tvrzení je možné obrátit. ( $A \Rightarrow B$  je totéž, co  $\neg B \Rightarrow \neg A$ .)

Jestliže

$(\forall n)(\exists z \text{ tž. } z \in L, |z| \geq n)(\forall u, v, w \text{ tž. } z = uvw, |uv| \leq n, |v| \geq 1)$   
 $(\exists i \geq 0) : uv^i w \notin L,$

pak  $L$  není regulární.

Pokud tedy chceme ukázat, že jazyk  $L$  není regulární, stačí ukázat, že splňuje výše uvedenou podmínku.

**Příklad:** Uvažujme jazyk  $L = \{a^i b^i \mid i \geq 0\}$ .

- Předpokládáme, že  $L$  je rozpoznáván nějakým automatem s  $n$  stavy.
- Zvolme slovo  $z = a^n b^n$ .
- Uvažujme všechny možnosti, jak může být  $z$  rozděleno na podslova  $u, v, w$  splňující podmínky  $|uv| \leq n$  a  $|v| \geq 1$ .  
Zjevně slova  $u$  a  $v$  obsahují pouze symboly  $a$ . Pro každé konkrétní rozdělení existují nějaká  $j$  a  $k$  taková, že  $j + k \leq n$ ,  $k \geq 1$  a
  - $u = a^j$
  - $v = a^k$
  - $w = a^{n-(j+k)} b^n$
- Pokud nyní zvolíme  $i = 0$ , dostáváme  $uv^i w = uw = a^{n-k} b^n$ . Protože  $n - k < n$ , zjevně platí  $uv^i w \notin L$ .

**Poznámka:** Dokazování platnosti či neplatnosti formule prvního řádu, kde se střídají universální a existenční kvantifikátory, si můžeme představovat jako hru dvou hráčů, hráče A a hráče B, kde se hráč A snaží dokázat, že formule platí, a hráč B, že formule neplatí.

Hráč A vždy volí hodnotu proměnné vázané existenčním kvantifikátorem a naopak hráč B vždy volí hodnotu proměnné vázané universálním kvantifikátorem.

Pokud například chceme platnost formule vyvrátit, stačí nám najít vítěznou strategii hráče B.

# Pumping Lemma

$$(\exists n)(\forall z \text{ tž. } z \in L, |z| \geq n)(\exists u, v, w \text{ tž. } z = uvw, |uv| \leq n, |v| \geq 1) \\ (\forall i \geq 0) : uv^i w \in L$$

Konkrétně pro Pumping Lemma vypadá hra následovně:

- 1 A zvolí  $n > 0$ .
- 2 B zvolí slovo  $z$  takové, že  $z \in L$  a  $|z| \geq n$ .
- 3 A zvolí slova  $u, v, w$  taková, že  $z = uvw, |uv| \leq n, |v| \geq 1$ .
- 4 B zvolí  $i \geq 0$ .
- 5 Je-li  $uv^i w \in L$ , vyhrává hráč A. Je-li  $uv^i w \notin L$ , vyhrává hráč B.



**Příklad:**  $L = \{a^i b^i \mid i \geq 0\}$

- 1 A zvolí  $n > 0$ .
- 2 B zvolí  $z = a^n b^n$
- 3 A zvolí slova  $u, v, w$  tž.  $z = uvw, |uv| \leq n, |v| \geq 1$
- 4 B zvolí  $i = 0$
- 5 Vyhrál hráč B, protože bez ohledu na to, co volil hráč A, vždy platí  $uv^i w \notin L$ , protože neprázdné slovo  $v$  se nachází v části slova  $z$  tvořené pouze symboly  $a$ , a jeho vypuštěním dostaneme slovo tvaru  $a^k b^n$ , kde  $k < n$ , a tedy nepatřící do  $L$ .

# Myhillova-Nerodova věta

Mějme nějakou abecedu  $\Sigma$  a nějaký jazyk  $L$  nad  $\Sigma$ .

Můžeme definovat následující binární relaci  $\equiv_L$  na množině  $\Sigma^*$ :

$$x \equiv_L y \stackrel{\text{def}}{\iff} \forall z \in \Sigma^* : (xz \in L \iff yz \in L)$$

Není těžké ukázat, že  $\equiv_L$  je ekvivalence (tj. je reflexivní, symetrická a tranzitivní).

Jestliže je tedy  $\equiv_L$  ekvivalence, znamená to, že indukuje rozklad na množině  $\Sigma^*$ .

**Poznámka:** O ekvivalenci říkáme, že je **konečného indexu**, jestliže rozklad indukovaný touto ekvivalencí má konečný počet tříd ekvivalence.

## Myhillova-Nerodova věta

Jazyk  $L$  je regulární právě když ekvivalence  $\equiv_L$  je konečného indexu.

**Poznámka:** Podrobný důkaz nebudeme provádět. Dá se ukázat, že pokud je  $L$  regulární, pak jednotlivé třídy ekvivalence  $\equiv_L$  odpovídají stavům minimálního deterministického automatu, který rozpoznává  $L$ . Do dané třídy patří vždy všechna ta slova, po jejichž přečtení se automat dostane do stavu, který této třídě odpovídá.

Navíc se dá ukázat, že  $\equiv_L$  je **pravou kongruencí**, tj. že pro libovolná  $x, y \in \Sigma^*$ ,  $a \in \Sigma$  platí

$$x \equiv_L y \quad \Rightarrow \quad xa \equiv_L ya$$

Jestliže se nám podaří ukázat, že  $\equiv_L$  není konečného indexu, pak  $L$  není regulární.

Zjevně tedy stačí najít nekonečně mnoho (různých) slov ze  $\Sigma^*$

$$w_1, w_2, w_3, \dots$$

patřících do různých tříd rozkladu indukovaného ekvivalencí  $\equiv_L$ , tj. takových, že  $w_i \not\equiv_L w_j$  pokud  $i \neq j$ .

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

Zvolme  $w_i = a^i$  pro  $i = 0, 1, 2, \dots$ . Pro  $i \neq j$  zjevně platí

$$a^i \not\equiv_L a^j$$

neboť pokud zvolíme  $z = b^i$ , pak  $a^i z \in L$ , ale  $a^j z \notin L$ .

Rozklad podle ekvivalence  $\equiv_L$  má tedy nekonečně mnoho tříd a  $L$  není regulární.

## Příklad:

$\langle \text{STMT} \rangle$	::=	$\langle \text{IF-STMT} \rangle \mid \langle \text{WHILE-STMT} \rangle \mid \langle \text{BEGIN-STMT} \rangle \mid \langle \text{ASSG-STMT} \rangle$
$\langle \text{IF-STMT} \rangle$	::=	<b>if</b> $\langle \text{BOOL-EXPR} \rangle$ <b>then</b> $\langle \text{STMT} \rangle$ <b>else</b> $\langle \text{STMT} \rangle$
$\langle \text{WHILE-STMT} \rangle$	::=	<b>while</b> $\langle \text{BOOL-EXPR} \rangle$ <b>do</b> $\langle \text{STMT} \rangle$
$\langle \text{BEGIN-STMT} \rangle$	::=	<b>begin</b> $\langle \text{STMT-LIST} \rangle$ <b>end</b>
$\langle \text{STMT-LIST} \rangle$	::=	$\langle \text{STMT} \rangle \mid \langle \text{STMT} \rangle ; \langle \text{STMT-LIST} \rangle$
$\langle \text{ASSG-STMT} \rangle$	::=	$\langle \text{VAR} \rangle := \langle \text{ARITH-EXPR} \rangle$
$\langle \text{BOOL-EXPR} \rangle$	::=	$\langle \text{ARITH-EXPR} \rangle \langle \text{COMPARE-OP} \rangle \langle \text{ARITH-EXPR} \rangle$
$\langle \text{COMPARE-OP} \rangle$	::=	$< \mid > \mid \leq \mid \geq \mid = \mid \neq$
$\langle \text{ARITH-EXPR} \rangle$	::=	$\langle \text{VAR} \rangle \mid \langle \text{CONST} \rangle \mid (\langle \text{ARITH-EXPR} \rangle \langle \text{ARITH-OP} \rangle \langle \text{ARITH-EXPR} \rangle)$
$\langle \text{ARITH-OP} \rangle$	::=	$+ \mid - \mid * \mid /$
$\langle \text{CONST} \rangle$	::=	$0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$
$\langle \text{VAR} \rangle$	::=	$a \mid b \mid c \mid \dots \mid x \mid y \mid z$

Symbolům, které mají v předchozím příkladě tvar  $\langle xxx \rangle$ , se říká **neterminální symboly (neterminály)**.

Pravidla popisují, jaké řetězce může daný neterminál reprezentovat.

Z neterminálu  $\langle STMT \rangle$  můžeme například dostat text

```
while  $x \leq y$  do begin  $x := (x + 1)$ ;  $y := (y - 1)$  end
```

**Poznámka:** Výše použité notaci se říká Backus-Naurova forma (BNF).

**while**  $x \leq y$  **do begin**  $x := (x + 1); y := (y - 1)$  **end**



**while**  $x \leq y$  **do begin**  $x := (x + 1); y := (y - 1)$  **end**

$\langle \text{STMT} \rangle$

**while**  $x \leq y$  **do begin**  $x := (x + 1); y := (y - 1)$  **end**

$\langle \text{STMT} \rangle$

$\langle \text{WHILE-STMT} \rangle$

**while**  $x \leq y$  **do begin**  $x := (x + 1); y := (y - 1)$  **end**

$\langle \text{STMT} \rangle$

$\langle \text{WHILE-STMT} \rangle$

**while**  $\langle \text{BOOL-EXPR} \rangle$  **do**  $\langle \text{STMT} \rangle$

**while**  $x \leq y$  **do begin**  $x := (x + 1); y := (y - 1)$  **end**

$\langle \text{STMT} \rangle$

$\langle \text{WHILE-STMT} \rangle$

**while**  $\langle \text{BOOL-EXPR} \rangle$  **do**  $\langle \text{STMT} \rangle$

**while**  $\langle \text{ARITH-EXPR} \rangle \langle \text{COMPARE-OP} \rangle \langle \text{ARITH-EXPR} \rangle$  **do**  $\langle \text{STMT} \rangle$

**while**  $x \leq y$  **do begin**  $x := (x + 1); y := (y - 1)$  **end**

⟨STMT⟩

⟨WHILE-STMT⟩

**while** ⟨BOOL-EXPR⟩ **do** ⟨STMT⟩

**while** ⟨ARITH-EXPR⟩⟨COMPARE-OP⟩⟨ARITH-EXPR⟩ **do** ⟨STMT⟩

**while** ⟨VAR⟩⟨COMPARE-OP⟩⟨ARITH-EXPR⟩ **do** ⟨STMT⟩

**while**  $x \leq y$  **do begin**  $x := (x + 1); y := (y - 1)$  **end**

⟨STMT⟩

⟨WHILE-STMT⟩

**while** ⟨BOOL-EXPR⟩ **do** ⟨STMT⟩

**while** ⟨ARITH-EXPR⟩⟨COMPARE-OP⟩⟨ARITH-EXPR⟩ **do** ⟨STMT⟩

**while** ⟨VAR⟩⟨COMPARE-OP⟩⟨ARITH-EXPR⟩ **do** ⟨STMT⟩

**while** ⟨VAR⟩  $\leq$  ⟨ARITH-EXPR⟩ **do** ⟨STMT⟩

**while**  $x \leq y$  **do begin**  $x := (x + 1); y := (y - 1)$  **end**

⟨STMT⟩

⟨WHILE-STMT⟩

**while** ⟨BOOL-EXPR⟩ **do** ⟨STMT⟩

**while** ⟨ARITH-EXPR⟩⟨COMPARE-OP⟩⟨ARITH-EXPR⟩ **do** ⟨STMT⟩

**while** ⟨VAR⟩⟨COMPARE-OP⟩⟨ARITH-EXPR⟩ **do** ⟨STMT⟩

**while** ⟨VAR⟩  $\leq$  ⟨ARITH-EXPR⟩ **do** ⟨STMT⟩

**while** ⟨VAR⟩  $\leq$  ⟨VAR⟩ **do** ⟨STMT⟩

**while**  $x \leq y$  **do begin**  $x := (x + 1); y := (y - 1)$  **end**

$\langle \text{STMT} \rangle$

$\langle \text{WHILE-STMT} \rangle$

**while**  $\langle \text{BOOL-EXPR} \rangle$  **do**  $\langle \text{STMT} \rangle$

**while**  $\langle \text{ARITH-EXPR} \rangle \langle \text{COMPARE-OP} \rangle \langle \text{ARITH-EXPR} \rangle$  **do**  $\langle \text{STMT} \rangle$

**while**  $\langle \text{VAR} \rangle \langle \text{COMPARE-OP} \rangle \langle \text{ARITH-EXPR} \rangle$  **do**  $\langle \text{STMT} \rangle$

**while**  $\langle \text{VAR} \rangle \leq \langle \text{ARITH-EXPR} \rangle$  **do**  $\langle \text{STMT} \rangle$

**while**  $\langle \text{VAR} \rangle \leq \langle \text{VAR} \rangle$  **do**  $\langle \text{STMT} \rangle$

**while**  $x \leq \langle \text{VAR} \rangle$  **do**  $\langle \text{STMT} \rangle$



**while**  $x \leq y$  **do begin**  $x := (x + 1); y := (y - 1)$  **end**

⟨STMT⟩

⟨WHILE-STMT⟩

**while** ⟨BOOL-EXPR⟩ **do** ⟨STMT⟩

**while** ⟨ARITH-EXPR⟩⟨COMPARE-OP⟩⟨ARITH-EXPR⟩ **do** ⟨STMT⟩

**while** ⟨VAR⟩⟨COMPARE-OP⟩⟨ARITH-EXPR⟩ **do** ⟨STMT⟩

**while** ⟨VAR⟩  $\leq$  ⟨ARITH-EXPR⟩ **do** ⟨STMT⟩

**while** ⟨VAR⟩  $\leq$  ⟨VAR⟩ **do** ⟨STMT⟩

**while**  $x \leq$  ⟨VAR⟩ **do** ⟨STMT⟩

**while**  $x \leq y$  **do** ⟨STMT⟩

**while**  $x \leq y$  **do begin**  $x := (x + 1); y := (y - 1)$  **end**

$\langle$ STMT $\rangle$

$\langle$ WHILE-STMT $\rangle$

**while**  $\langle$ BOOL-EXPR $\rangle$  **do**  $\langle$ STMT $\rangle$

**while**  $\langle$ ARITH-EXPR $\rangle$   $\langle$ COMPARE-OP $\rangle$   $\langle$ ARITH-EXPR $\rangle$  **do**  $\langle$ STMT $\rangle$

**while**  $\langle$ VAR $\rangle$   $\langle$ COMPARE-OP $\rangle$   $\langle$ ARITH-EXPR $\rangle$  **do**  $\langle$ STMT $\rangle$

**while**  $\langle$ VAR $\rangle \leq \langle$ ARITH-EXPR $\rangle$  **do**  $\langle$ STMT $\rangle$

**while**  $\langle$ VAR $\rangle \leq \langle$ VAR $\rangle$  **do**  $\langle$ STMT $\rangle$

**while**  $x \leq \langle$ VAR $\rangle$  **do**  $\langle$ STMT $\rangle$

**while**  $x \leq y$  **do**  $\langle$ STMT $\rangle$

**while**  $x \leq y$  **do**  $\langle$ BEGIN-STMT $\rangle$

**while**  $x \leq y$  **do begin**  $x := (x + 1); y := (y - 1)$  **end**

$\langle$ STMT $\rangle$

$\langle$ WHILE-STMT $\rangle$

**while**  $\langle$ BOOL-EXPR $\rangle$  **do**  $\langle$ STMT $\rangle$

**while**  $\langle$ ARITH-EXPR $\rangle$   $\langle$ COMPARE-OP $\rangle$   $\langle$ ARITH-EXPR $\rangle$  **do**  $\langle$ STMT $\rangle$

**while**  $\langle$ VAR $\rangle$   $\langle$ COMPARE-OP $\rangle$   $\langle$ ARITH-EXPR $\rangle$  **do**  $\langle$ STMT $\rangle$

**while**  $\langle$ VAR $\rangle \leq \langle$ ARITH-EXPR $\rangle$  **do**  $\langle$ STMT $\rangle$

**while**  $\langle$ VAR $\rangle \leq \langle$ VAR $\rangle$  **do**  $\langle$ STMT $\rangle$

**while**  $x \leq \langle$ VAR $\rangle$  **do**  $\langle$ STMT $\rangle$

**while**  $x \leq y$  **do**  $\langle$ STMT $\rangle$

**while**  $x \leq y$  **do**  $\langle$ BEGIN-STMT $\rangle$

**while**  $x \leq y$  **do begin**  $\langle$ STMT-LIST $\rangle$  **end**

**while**  $x \leq y$  **do begin**  $x := (x + 1); y := (y - 1)$  **end**

$\langle$ STMT $\rangle$

$\langle$ WHILE-STMT $\rangle$

**while**  $\langle$ BOOL-EXPR $\rangle$  **do**  $\langle$ STMT $\rangle$

**while**  $\langle$ ARITH-EXPR $\rangle$   $\langle$ COMPARE-OP $\rangle$   $\langle$ ARITH-EXPR $\rangle$  **do**  $\langle$ STMT $\rangle$

**while**  $\langle$ VAR $\rangle$   $\langle$ COMPARE-OP $\rangle$   $\langle$ ARITH-EXPR $\rangle$  **do**  $\langle$ STMT $\rangle$

**while**  $\langle$ VAR $\rangle \leq \langle$ ARITH-EXPR $\rangle$  **do**  $\langle$ STMT $\rangle$

**while**  $\langle$ VAR $\rangle \leq \langle$ VAR $\rangle$  **do**  $\langle$ STMT $\rangle$

**while**  $x \leq \langle$ VAR $\rangle$  **do**  $\langle$ STMT $\rangle$

**while**  $x \leq y$  **do**  $\langle$ STMT $\rangle$

**while**  $x \leq y$  **do**  $\langle$ BEGIN-STMT $\rangle$

**while**  $x \leq y$  **do begin**  $\langle$ STMT-LIST $\rangle$  **end**

...

Formálně je **bezkontextová gramatika** definována jako čtveřice

$$G = (\Pi, \Sigma, S, P)$$

kde:

- $\Pi$  je konečná množina **neterminálních symbolů (neterminálů)**
- $\Sigma$  je konečná množina **terminálních symbolů (terminálů)**,  
přičemž  $\Pi \cap \Sigma = \emptyset$
- $S \in \Pi$  je **počáteční neterminál**
- $P \subseteq \Pi \times (\Pi \cup \Sigma)^*$  je konečná množina **přepisovacích pravidel**

## Poznámky:

- Pro označení neterminálních symbolů budeme používat velká písmena  $A, B, C, \dots$
- Pro označení terminálních symbolů budeme používat malá písmena  $a, b, c, \dots$  nebo číslice  $0, 1, 2, \dots$
- Pro označení řetězců z  $(\Pi \cup \Sigma)^*$  budeme používat malá písmena řecké abecedy  $\alpha, \beta, \gamma, \dots$
- Místo zápisu  $(A, \alpha)$  budeme pro pravidla používat zápis

$$A \rightarrow \alpha$$

$A$  – levá strana pravidla

$\alpha$  – pravá strana pravidla

**Příklad:** Gramatika  $G = (\Pi, \Sigma, S, P)$ , kde

- $\Pi = \{A, B, C\}$
- $\Sigma = \{a, b\}$
- $S = A$
- $P$  obsahuje pravidla

$$A \rightarrow aBBb$$

$$A \rightarrow AaA$$

$$B \rightarrow \varepsilon$$

$$B \rightarrow bCA$$

$$C \rightarrow AB$$

$$C \rightarrow a$$

$$C \rightarrow b$$

**Poznámka:** Pokud máme více pravidel se stejnou levou stranou, jako třeba

$$A \rightarrow \alpha_1 \qquad A \rightarrow \alpha_2 \qquad A \rightarrow \alpha_3$$

můžeme je stručněji zapsat jako

$$A \rightarrow \alpha_1 \mid \alpha_2 \mid \alpha_3$$

Například pravidla dříve uvedené gramatiky můžeme zapsat jako

$$\begin{aligned} A &\rightarrow aBBb \mid AaA \\ B &\rightarrow \varepsilon \mid bCA \\ C &\rightarrow AB \mid a \mid b \end{aligned}$$



Řetězce z  $(\Pi \cup \Sigma)^*$  nazýváme **větné formy**.

Na větných formách definujeme relaci  $\Rightarrow \subseteq (\Pi \cup \Sigma)^* \times (\Pi \cup \Sigma)^*$  takovou, že

$$\alpha \Rightarrow \alpha'$$

právě když  $\alpha = \beta_1 A \beta_2$  a  $\alpha' = \beta_1 \gamma \beta_2$  pro nějaká  $\beta_1, \beta_2, \gamma \in (\Pi \cup \Sigma)^*$  a  $A \in \Pi$ , kde  $(A \rightarrow \gamma) \in P$ .

**Příklad:** Jestliže  $(B \rightarrow bCA) \in P$ , pak

$$aCBbA \Rightarrow aCbCABA$$

**Poznámka:** Neformálně řečeno zápis  $\alpha \Rightarrow \alpha'$  znamená, že z větné formy  $\alpha$  je možné jedním krokem odvodit větnou formu  $\alpha'$ , a to tak, že výskyt nějakého neterminálu  $A$  v  $\alpha$  nahradíme pravou stranou nějakého pravidla  $A \rightarrow \alpha$ , kde se  $A$  vyskytuje na levé straně.

Řetězce z  $(\Pi \cup \Sigma)^*$  nazýváme **větné formy**.

Na větných formách definujeme relaci  $\Rightarrow \subseteq (\Pi \cup \Sigma)^* \times (\Pi \cup \Sigma)^*$  takovou, že

$$\alpha \Rightarrow \alpha'$$

právě když  $\alpha = \beta_1 A \beta_2$  a  $\alpha' = \beta_1 \gamma \beta_2$  pro nějaká  $\beta_1, \beta_2, \gamma \in (\Pi \cup \Sigma)^*$  a  $A \in \Pi$ , kde  $(A \rightarrow \gamma) \in P$ .

**Příklad:** Jestliže  $(B \rightarrow bCA) \in P$ , pak

$$aC\underline{B}bA \Rightarrow aC\underline{bCA}bA$$

**Poznámka:** Neformálně řečeno zápis  $\alpha \Rightarrow \alpha'$  znamená, že z větné formy  $\alpha$  je možné jedním krokem odvodit větnou formu  $\alpha'$ , a to tak, že výskyt nějakého neterminálu  $A$  v  $\alpha$  nahradíme pravou stranou nějakého pravidla  $A \rightarrow \alpha$ , kde se  $A$  vyskytuje na levé straně.

**Derivace** délky  $n$  větné formy  $\alpha'$  z větné formy  $\alpha$  je posloupnost větných forem

$$\beta_0, \beta_1, \beta_2, \dots, \beta_n$$

takových, že

- $\alpha = \beta_0$
- $\beta_{i-1} \Rightarrow \beta_i$  pro všechna  $i \in \{1, 2, \dots, n\}$
- $\alpha' = \beta_n$

což můžeme stručněji zapsat jako

$$\alpha = \beta_0 \Rightarrow \beta_1 \Rightarrow \beta_2 \Rightarrow \dots \Rightarrow \beta_{n-1} \Rightarrow \beta_n = \alpha'$$

Skutečnost, že pro dané  $n$  existuje nějaká derivace délky  $n$  větné formy  $\alpha'$  z větné formy  $\alpha$ , označujeme zápisem

$$\alpha \Rightarrow^n \alpha'$$

Skutečnost, že existuje nějaká derivace (nějaké délky  $n$ , kde  $n \geq 0$ ) větné formy  $\alpha'$  z větné formy  $\alpha$ , označujeme zápisem

$$\alpha \Rightarrow^* \alpha'$$

**Poznámka:** Relace  $\Rightarrow^*$  je reflexivním a tranzitivním uzávěrem relace  $\Rightarrow$  (tj. nejmenší reflexivní a tranzitivní relací obsahující relaci  $\Rightarrow$ ).

**Jazyk**  $L(G)$  generovaný gramatikou  $G = (\Pi, \Sigma, S, P)$  je množina všech slov v abecedě  $\Sigma$ , která lze odvodit nějakou derivací z počátečního neterminálu  $S$  pomocí pravidel z  $P$ , tj.

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

**Jazyk**  $L(G)$  generovaný gramatikou  $G = (\Pi, \Sigma, S, P)$  je množina všech slov v abecedě  $\Sigma$ , která lze odvodit nějakou derivací z počátečního neterminálu  $S$  pomocí pravidel z  $P$ , tj.

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

**Příklad:**  $G = (\Pi, \Sigma, A, P)$ , kde  $\Pi = \{A, B, C\}$ ,  $\Sigma = \{a, b\}$  a  $P$  obsahuje pravidla

$$A \rightarrow aBBb \mid AaA$$

$$B \rightarrow \varepsilon \mid bCA$$

$$C \rightarrow AB \mid a \mid b$$

$abbabb \in L(G)$  protože:

**Jazyk**  $L(G)$  generovaný gramatikou  $G = (\Pi, \Sigma, S, P)$  je množina všech slov v abecedě  $\Sigma$ , která lze odvodit nějakou derivací z počátečního neterminálu  $S$  pomocí pravidel z  $P$ , tj.

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

**Příklad:**  $G = (\Pi, \Sigma, A, P)$ , kde  $\Pi = \{A, B, C\}$ ,  $\Sigma = \{a, b\}$  a  $P$  obsahuje pravidla

$$A \rightarrow aBBb \mid AaA$$

$$B \rightarrow \varepsilon \mid bCA$$

$$C \rightarrow AB \mid a \mid b$$

$abbabb \in L(G)$  protože:

$A$

**Jazyk**  $L(G)$  generovaný gramatikou  $G = (\Pi, \Sigma, S, P)$  je množina všech slov v abecedě  $\Sigma$ , která lze odvodit nějakou derivací z počátečního neterminálu  $S$  pomocí pravidel z  $P$ , tj.

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

**Příklad:**  $G = (\Pi, \Sigma, A, P)$ , kde  $\Pi = \{A, B, C\}$ ,  $\Sigma = \{a, b\}$  a  $P$  obsahuje pravidla

$$\begin{aligned} A &\rightarrow aBBb \mid AaA \\ B &\rightarrow \varepsilon \mid bCA \\ C &\rightarrow AB \mid a \mid b \end{aligned}$$

$abbabb \in L(G)$  protože:

A



**Jazyk**  $L(G)$  generovaný gramatikou  $G = (\Pi, \Sigma, S, P)$  je množina všech slov v abecedě  $\Sigma$ , která lze odvodit nějakou derivací z počátečního neterminálu  $S$  pomocí pravidel z  $P$ , tj.

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

**Příklad:**  $G = (\Pi, \Sigma, A, P)$ , kde  $\Pi = \{A, B, C\}$ ,  $\Sigma = \{a, b\}$  a  $P$  obsahuje pravidla

$$\begin{aligned} A &\rightarrow aBBb \mid AaA \\ B &\rightarrow \varepsilon \mid bCA \\ C &\rightarrow AB \mid a \mid b \end{aligned}$$

$abbabb \in L(G)$  protože:

$$\underline{A} \Rightarrow \underline{aBBb}$$

**Jazyk**  $L(G)$  generovaný gramatikou  $G = (\Pi, \Sigma, S, P)$  je množina všech slov v abecedě  $\Sigma$ , která lze odvodit nějakou derivací z počátečního neterminálu  $S$  pomocí pravidel z  $P$ , tj.

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

**Příklad:**  $G = (\Pi, \Sigma, A, P)$ , kde  $\Pi = \{A, B, C\}$ ,  $\Sigma = \{a, b\}$  a  $P$  obsahuje pravidla

$$A \rightarrow aBBb \mid AaA$$

$$B \rightarrow \varepsilon \mid bCA$$

$$C \rightarrow AB \mid a \mid b$$

$abbabb \in L(G)$  protože:

$$A \Rightarrow aBBb$$

**Jazyk**  $L(G)$  generovaný gramatikou  $G = (\Pi, \Sigma, S, P)$  je množina všech slov v abecedě  $\Sigma$ , která lze odvodit nějakou derivací z počátečního neterminálu  $S$  pomocí pravidel z  $P$ , tj.

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

**Příklad:**  $G = (\Pi, \Sigma, A, P)$ , kde  $\Pi = \{A, B, C\}$ ,  $\Sigma = \{a, b\}$  a  $P$  obsahuje pravidla

$$A \rightarrow aBBb \mid AaA$$

$$\underline{B} \rightarrow \varepsilon \mid bCA$$

$$C \rightarrow AB \mid a \mid b$$

$abbabb \in L(G)$  protože:

$$A \Rightarrow a\underline{B}Bb$$

# Bezkontextové gramatiky

**Jazyk**  $L(G)$  generovaný gramatikou  $G = (\Pi, \Sigma, S, P)$  je množina všech slov v abecedě  $\Sigma$ , která lze odvodit nějakou derivací z počátečního neterminálu  $S$  pomocí pravidel z  $P$ , tj.

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

**Příklad:**  $G = (\Pi, \Sigma, A, P)$ , kde  $\Pi = \{A, B, C\}$ ,  $\Sigma = \{a, b\}$  a  $P$  obsahuje pravidla

$$A \rightarrow aBBb \mid AaA$$

$$\underline{B} \rightarrow \varepsilon \mid \underline{bCA}$$

$$C \rightarrow AB \mid a \mid b$$

$abbabb \in L(G)$  protože:

$$A \Rightarrow a\underline{B}Bb \Rightarrow a\underline{bCA}Bb$$

**Jazyk**  $L(G)$  generovaný gramatikou  $G = (\Pi, \Sigma, S, P)$  je množina všech slov v abecedě  $\Sigma$ , která lze odvodit nějakou derivací z počátečního neterminálu  $S$  pomocí pravidel z  $P$ , tj.

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

**Příklad:**  $G = (\Pi, \Sigma, A, P)$ , kde  $\Pi = \{A, B, C\}$ ,  $\Sigma = \{a, b\}$  a  $P$  obsahuje pravidla

$$A \rightarrow aBBb \mid AaA$$

$$B \rightarrow \varepsilon \mid bCA$$

$$C \rightarrow AB \mid a \mid b$$

$abbabb \in L(G)$  protože:

$$A \Rightarrow aBBb \Rightarrow abCABb$$

**Jazyk**  $L(G)$  generovaný gramatikou  $G = (\Pi, \Sigma, S, P)$  je množina všech slov v abecedě  $\Sigma$ , která lze odvodit nějakou derivací z počátečního neterminálu  $S$  pomocí pravidel z  $P$ , tj.

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

**Příklad:**  $G = (\Pi, \Sigma, A, P)$ , kde  $\Pi = \{A, B, C\}$ ,  $\Sigma = \{a, b\}$  a  $P$  obsahuje pravidla

$$\begin{aligned} A &\rightarrow aBBb \mid AaA \\ B &\rightarrow \varepsilon \mid bCA \\ C &\rightarrow AB \mid a \mid b \end{aligned}$$

$abbabb \in L(G)$  protože:

$$A \Rightarrow aBBb \Rightarrow abC\underline{A}Bb$$

**Jazyk**  $L(G)$  generovaný gramatikou  $G = (\Pi, \Sigma, S, P)$  je množina všech slov v abecedě  $\Sigma$ , která lze odvodit nějakou derivací z počátečního neterminálu  $S$  pomocí pravidel z  $P$ , tj.

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

**Příklad:**  $G = (\Pi, \Sigma, A, P)$ , kde  $\Pi = \{A, B, C\}$ ,  $\Sigma = \{a, b\}$  a  $P$  obsahuje pravidla

$$\begin{aligned} A &\rightarrow \underline{aBBb} \mid AaA \\ B &\rightarrow \varepsilon \mid bCA \\ C &\rightarrow AB \mid a \mid b \end{aligned}$$

$abbabb \in L(G)$  protože:

$$A \Rightarrow aBBb \Rightarrow abC\underline{A}Bb \Rightarrow abC\underline{aBBb}Bb$$

**Jazyk**  $L(G)$  generovaný gramatikou  $G = (\Pi, \Sigma, S, P)$  je množina všech slov v abecedě  $\Sigma$ , která lze odvodit nějakou derivací z počátečního neterminálu  $S$  pomocí pravidel z  $P$ , tj.

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

**Příklad:**  $G = (\Pi, \Sigma, A, P)$ , kde  $\Pi = \{A, B, C\}$ ,  $\Sigma = \{a, b\}$  a  $P$  obsahuje pravidla

$$A \rightarrow aBBb \mid AaA$$

$$B \rightarrow \varepsilon \mid bCA$$

$$C \rightarrow AB \mid a \mid b$$

$abbabb \in L(G)$  protože:

$$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb$$



# Bezkontextové gramatiky

**Jazyk**  $L(G)$  generovaný gramatikou  $G = (\Pi, \Sigma, S, P)$  je množina všech slov v abecedě  $\Sigma$ , která lze odvodit nějakou derivací z počátečního neterminálu  $S$  pomocí pravidel z  $P$ , tj.

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

**Příklad:**  $G = (\Pi, \Sigma, A, P)$ , kde  $\Pi = \{A, B, C\}$ ,  $\Sigma = \{a, b\}$  a  $P$  obsahuje pravidla

$$A \rightarrow aBBb \mid AaA$$

$$\underline{B} \rightarrow \varepsilon \mid bCA$$

$$C \rightarrow AB \mid a \mid b$$

$abbabb \in L(G)$  protože:

$$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaB\underline{B}bBb$$

# Bezkontextové gramatiky

**Jazyk**  $L(G)$  generovaný gramatikou  $G = (\Pi, \Sigma, S, P)$  je množina všech slov v abecedě  $\Sigma$ , která lze odvodit nějakou derivací z počátečního neterminálu  $S$  pomocí pravidel z  $P$ , tj.

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

**Příklad:**  $G = (\Pi, \Sigma, A, P)$ , kde  $\Pi = \{A, B, C\}$ ,  $\Sigma = \{a, b\}$  a  $P$  obsahuje pravidla

$$A \rightarrow aBBb \mid AaA$$

$$\underline{B} \rightarrow \underline{\epsilon} \mid bCA$$

$$C \rightarrow AB \mid a \mid b$$

$abbabb \in L(G)$  protože:

$$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaB\underline{B}bBb \Rightarrow abCaBbBb$$

**Jazyk**  $L(G)$  generovaný gramatikou  $G = (\Pi, \Sigma, S, P)$  je množina všech slov v abecedě  $\Sigma$ , která lze odvodit nějakou derivací z počátečního neterminálu  $S$  pomocí pravidel z  $P$ , tj.

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

**Příklad:**  $G = (\Pi, \Sigma, A, P)$ , kde  $\Pi = \{A, B, C\}$ ,  $\Sigma = \{a, b\}$  a  $P$  obsahuje pravidla

$$A \rightarrow aBBb \mid AaA$$

$$B \rightarrow \varepsilon \mid bCA$$

$$C \rightarrow AB \mid a \mid b$$

$abbabb \in L(G)$  protože:

$$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb$$

# Bezkontextové gramatiky

**Jazyk**  $L(G)$  generovaný gramatikou  $G = (\Pi, \Sigma, S, P)$  je množina všech slov v abecedě  $\Sigma$ , která lze odvodit nějakou derivací z počátečního neterminálu  $S$  pomocí pravidel z  $P$ , tj.

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

**Příklad:**  $G = (\Pi, \Sigma, A, P)$ , kde  $\Pi = \{A, B, C\}$ ,  $\Sigma = \{a, b\}$  a  $P$  obsahuje pravidla

$$A \rightarrow aBBb \mid AaA$$

$$B \rightarrow \varepsilon \mid bCA$$

$$\underline{C} \rightarrow AB \mid a \mid b$$

$abbabb \in L(G)$  protože:

$$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow ab\underline{C}aBbBb$$

# Bezkontextové gramatiky

**Jazyk**  $L(G)$  generovaný gramatikou  $G = (\Pi, \Sigma, S, P)$  je množina všech slov v abecedě  $\Sigma$ , která lze odvodit nějakou derivací z počátečního neterminálu  $S$  pomocí pravidel z  $P$ , tj.

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

**Příklad:**  $G = (\Pi, \Sigma, A, P)$ , kde  $\Pi = \{A, B, C\}$ ,  $\Sigma = \{a, b\}$  a  $P$  obsahuje pravidla

$$A \rightarrow aBBb \mid AaA$$

$$B \rightarrow \varepsilon \mid bCA$$

$$\underline{C} \rightarrow AB \mid a \mid \underline{b}$$

$abbabb \in L(G)$  protože:

$$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow ab\underline{C}aBbBb \Rightarrow ab\underline{b}aBbBb$$

**Jazyk**  $L(G)$  generovaný gramatikou  $G = (\Pi, \Sigma, S, P)$  je množina všech slov v abecedě  $\Sigma$ , která lze odvodit nějakou derivací z počátečního neterminálu  $S$  pomocí pravidel z  $P$ , tj.

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

**Příklad:**  $G = (\Pi, \Sigma, A, P)$ , kde  $\Pi = \{A, B, C\}$ ,  $\Sigma = \{a, b\}$  a  $P$  obsahuje pravidla

$$A \rightarrow aBBb \mid AaA$$

$$B \rightarrow \varepsilon \mid bCA$$

$$C \rightarrow AB \mid a \mid b$$

$abbabb \in L(G)$  protože:

$$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBbBb$$

# Bezkontextové gramatiky

**Jazyk**  $L(G)$  generovaný gramatikou  $G = (\Pi, \Sigma, S, P)$  je množina všech slov v abecedě  $\Sigma$ , která lze odvodit nějakou derivací z počátečního neterminálu  $S$  pomocí pravidel z  $P$ , tj.

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

**Příklad:**  $G = (\Pi, \Sigma, A, P)$ , kde  $\Pi = \{A, B, C\}$ ,  $\Sigma = \{a, b\}$  a  $P$  obsahuje pravidla

$$A \rightarrow aBBb \mid AaA$$

$$\underline{B} \rightarrow \varepsilon \mid bCA$$

$$C \rightarrow AB \mid a \mid b$$

$abbabb \in L(G)$  protože:

$$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBb\underline{B}b$$

**Jazyk**  $L(G)$  generovaný gramatikou  $G = (\Pi, \Sigma, S, P)$  je množina všech slov v abecedě  $\Sigma$ , která lze odvodit nějakou derivací z počátečního neterminálu  $S$  pomocí pravidel z  $P$ , tj.

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

**Příklad:**  $G = (\Pi, \Sigma, A, P)$ , kde  $\Pi = \{A, B, C\}$ ,  $\Sigma = \{a, b\}$  a  $P$  obsahuje pravidla

$$A \rightarrow aBBb \mid AaA$$

$$\underline{B} \rightarrow \underline{\epsilon} \mid bCA$$

$$C \rightarrow AB \mid a \mid b$$

$abbabb \in L(G)$  protože:

$$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBb\underline{B}b \Rightarrow abbaBbb$$



# Bezkontextové gramatiky

**Jazyk**  $L(G)$  generovaný gramatikou  $G = (\Pi, \Sigma, S, P)$  je množina všech slov v abecedě  $\Sigma$ , která lze odvodit nějakou derivací z počátečního neterminálu  $S$  pomocí pravidel z  $P$ , tj.

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

**Příklad:**  $G = (\Pi, \Sigma, A, P)$ , kde  $\Pi = \{A, B, C\}$ ,  $\Sigma = \{a, b\}$  a  $P$  obsahuje pravidla

$$A \rightarrow aBBb \mid AaA$$

$$B \rightarrow \varepsilon \mid bCA$$

$$C \rightarrow AB \mid a \mid b$$

$abbabb \in L(G)$  protože:

$$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBbBb \Rightarrow abbaBbb$$

**Jazyk**  $L(G)$  generovaný gramatikou  $G = (\Pi, \Sigma, S, P)$  je množina všech slov v abecedě  $\Sigma$ , která lze odvodit nějakou derivací z počátečního neterminálu  $S$  pomocí pravidel z  $P$ , tj.

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

**Příklad:**  $G = (\Pi, \Sigma, A, P)$ , kde  $\Pi = \{A, B, C\}$ ,  $\Sigma = \{a, b\}$  a  $P$  obsahuje pravidla

$$A \rightarrow aBBb \mid AaA$$

$$\underline{B} \rightarrow \varepsilon \mid bCA$$

$$C \rightarrow AB \mid a \mid b$$

$abbabb \in L(G)$  protože:

$$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBbBb \Rightarrow abba\underline{B}bb$$

# Bezkontextové gramatiky

**Jazyk**  $L(G)$  generovaný gramatikou  $G = (\Pi, \Sigma, S, P)$  je množina všech slov v abecedě  $\Sigma$ , která lze odvodit nějakou derivací z počátečního neterminálu  $S$  pomocí pravidel z  $P$ , tj.

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

**Příklad:**  $G = (\Pi, \Sigma, A, P)$ , kde  $\Pi = \{A, B, C\}$ ,  $\Sigma = \{a, b\}$  a  $P$  obsahuje pravidla

$$A \rightarrow aBBb \mid AaA$$

$$\underline{B} \rightarrow \underline{\epsilon} \mid bCA$$

$$C \rightarrow AB \mid a \mid b$$

$abbabb \in L(G)$  protože:

$$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBbBb \Rightarrow abba\underline{B}bb \Rightarrow abbabb$$

# Bezkontextové gramatiky

**Jazyk**  $L(G)$  generovaný gramatikou  $G = (\Pi, \Sigma, S, P)$  je množina všech slov v abecedě  $\Sigma$ , která lze odvodit nějakou derivací z počátečního neterminálu  $S$  pomocí pravidel z  $P$ , tj.

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

**Příklad:**  $G = (\Pi, \Sigma, A, P)$ , kde  $\Pi = \{A, B, C\}$ ,  $\Sigma = \{a, b\}$  a  $P$  obsahuje pravidla

$$A \rightarrow aBBb \mid AaA$$

$$B \rightarrow \varepsilon \mid bCA$$

$$C \rightarrow AB \mid a \mid b$$

$abbabb \in L(G)$  protože:

$$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBbBb \Rightarrow abbaBbb \Rightarrow abbabb$$

**Příklad:** Chceme vytvořit gramatiku generující jazyk

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

**Příklad:** Chceme vytvořit gramatiku generující jazyk

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

Gramatika  $G = (\Pi, \Sigma, S, P)$ , kde  $\Pi = \{S\}$ ,  $\Sigma = \{a, b\}$  a  $P$  obsahuje

$$S \rightarrow aSb \mid \varepsilon$$

**Příklad:** Chceme vytvořit gramatiku generující jazyk

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

Gramatika  $G = (\Pi, \Sigma, S, P)$ , kde  $\Pi = \{S\}$ ,  $\Sigma = \{a, b\}$  a  $P$  obsahuje

$$S \rightarrow aSb \mid \varepsilon$$

$$S \Rightarrow \varepsilon$$

$$S \Rightarrow aSb \Rightarrow ab$$

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$$

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabbb$$

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaaaSbbbb \Rightarrow aaaabbbb$$

...

**Příklad:** Chceme vytvořit gramatiku generující jazyk tvořený všemi palindromy nad abecedou  $\{a, b\}$ , tj.

$$L = \{w \in \{a, b\}^* \mid w = w^R\}$$

**Poznámka:**  $w^R$  označuje tzv. **zrcadlový obraz** slova  $w$ , tj. slovo  $w$  zapsané pozpátku.



**Příklad:** Chceme vytvořit gramatiku generující jazyk tvořený všemi palindromy nad abecedou  $\{a, b\}$ , tj.

$$L = \{w \in \{a, b\}^* \mid w = w^R\}$$

**Poznámka:**  $w^R$  označuje tzv. **zrcadlový obraz** slova  $w$ , tj. slovo  $w$  zapsané pozpátku.

*Řešení:*

$$S \rightarrow aSa \mid bSb \mid a \mid b \mid \varepsilon$$

**Příklad:** Chceme vytvořit gramatiku generující jazyk tvořený všemi palindromy nad abecedou  $\{a, b\}$ , tj.

$$L = \{w \in \{a, b\}^* \mid w = w^R\}$$

**Poznámka:**  $w^R$  označuje tzv. **zrcadlový obraz** slova  $w$ , tj. slovo  $w$  zapsané pozpátku.

*Řešení:*

$$S \rightarrow aSa \mid bSb \mid a \mid b \mid \varepsilon$$

$$S \Rightarrow aSa \Rightarrow abSba \Rightarrow abaSaba \Rightarrow abaaaba$$

**Příklad:** Chceme vytvořit gramatiku generující jazyk  $L$  tvořený všemi dobře uzávorkovanými sekvencemi symbolů '(' a ')'.  
Například  $((()())()) \in L$ , ale  $)() \notin L$ .

**Příklad:** Chceme vytvořit gramatiku generující jazyk  $L$  tvořený všemi dobře uzávorkovanými sekvencemi symbolů '(' a ')'.  
Například  $((()())()) \in L$ , ale  $)() \notin L$ .

*Řešení:*

$$S \rightarrow \varepsilon \mid (S) \mid SS$$

**Příklad:** Chceme vytvořit gramatiku generující jazyk  $L$  tvořený všemi dobře uzávorkovanými sekvencemi symbolů '(' a ')'.  
Například  $((()())()) \in L$ , ale  $)() \notin L$ .

*Řešení:*

$$S \rightarrow \varepsilon \mid (S) \mid SS$$

$$\begin{aligned} S &\Rightarrow SS \Rightarrow (S)S \Rightarrow (S)(S) \Rightarrow (SS)(S) \Rightarrow ((S)S)(S) \Rightarrow \\ &((()S)(S) \Rightarrow (() (S))(S) \Rightarrow (() ()) (S) \Rightarrow (() ()) ((S)) \Rightarrow \\ &(() ()) (()) \end{aligned}$$

**Příklad:** Chceme vytvořit gramatiku generující jazyk  $L$  tvořený všemi dobře vytvořenými aritmetickými výrazy, kde operandy jsou vždy tvaru 'a', a kde jako operátory můžeme používat symboly  $+$  a  $*$ .

Například  $(a + a) * a + (a * a) \in L$ .

**Příklad:** Chceme vytvořit gramatiku generující jazyk  $L$  tvořený všemi dobře vytvořenými aritmetickými výrazy, kde operandy jsou vždy tvaru 'a', a kde jako operátory můžeme používat symboly  $+$  a  $*$ .

Například  $(a + a) * a + (a * a) \in L$ .

*Řešení:*

$$E \rightarrow a \mid E + E \mid E * E \mid (E)$$

**Příklad:** Chceme vytvořit gramatiku generující jazyk  $L$  tvořený všemi dobře vytvořenými aritmetickými výrazy, kde operandy jsou vždy tvaru 'a', a kde jako operátory můžeme používat symboly  $+$  a  $*$ .

Například  $(a + a) * a + (a * a) \in L$ .

*Řešení:*

$$E \rightarrow a \mid E + E \mid E * E \mid (E)$$

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



$$A \rightarrow aBBb \mid AaA$$
$$B \rightarrow \varepsilon \mid bCA$$
$$C \rightarrow AB \mid a \mid b$$

A

$A \rightarrow aBBb \mid AaA$

$B \rightarrow \varepsilon \mid bCA$

$C \rightarrow AB \mid a \mid b$

A

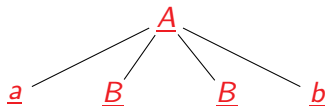
A

A  $\rightarrow$  aBBb | AaA

B  $\rightarrow$   $\varepsilon$  | bCA

C  $\rightarrow$  AB | a | b

A

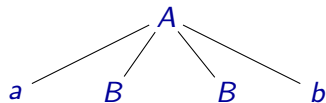


A  $\rightarrow$  aBBb | AaA

B  $\rightarrow$   $\varepsilon$  | bCA

C  $\rightarrow$  AB | a | b

A  $\Rightarrow$  aBBb

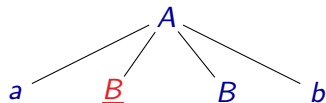


$A \rightarrow aBBb \mid AaA$

$B \rightarrow \varepsilon \mid bCA$

$C \rightarrow AB \mid a \mid b$

$A \Rightarrow aBBb$



$A \rightarrow aBBb \mid AaA$

$\underline{B} \rightarrow \varepsilon \mid bCA$

$C \rightarrow AB \mid a \mid b$

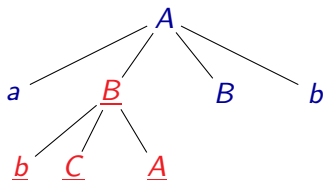
$A \Rightarrow a\underline{B}Bb$

# Derivační strom

$A \rightarrow aBBb \mid AaA$

$B \rightarrow \varepsilon \mid \underline{bCA}$

$C \rightarrow AB \mid a \mid b$



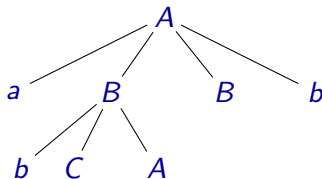
$A \Rightarrow a\underline{B}Bb \Rightarrow a\underline{bCA}Bb$

# Derivační strom

$A \rightarrow aBBb \mid AaA$

$B \rightarrow \varepsilon \mid bCA$

$C \rightarrow AB \mid a \mid b$



$A \Rightarrow aBBb \Rightarrow abCABb$

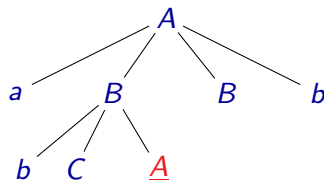


# Derivační strom

$\underline{A} \rightarrow aBBb \mid AaA$

$B \rightarrow \varepsilon \mid bCA$

$C \rightarrow AB \mid a \mid b$



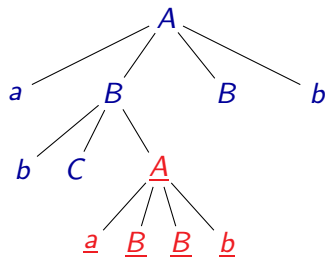
$A \Rightarrow aBBb \Rightarrow abC\underline{A}Bb$

# Derivační strom

$\underline{A} \rightarrow \underline{aBBb} \mid AaA$

$B \rightarrow \varepsilon \mid bCA$

$C \rightarrow AB \mid a \mid b$



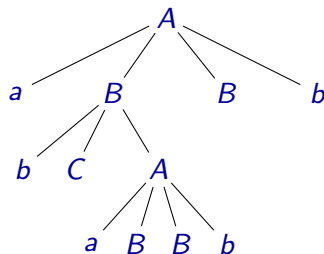
$A \Rightarrow aBBb \Rightarrow abC\underline{A}Bb \Rightarrow abC\underline{aBBb}Bb$

# Derivační strom

$A \rightarrow aBBb \mid AaA$

$B \rightarrow \varepsilon \mid bCA$

$C \rightarrow AB \mid a \mid b$



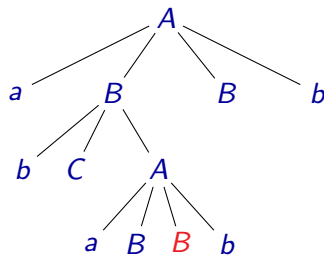
$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb$

# Derivační strom

$A \rightarrow aBBb \mid AaA$

$\underline{B} \rightarrow \varepsilon \mid bCA$

$C \rightarrow AB \mid a \mid b$



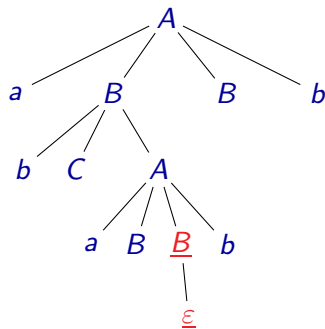
$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaB\underline{B}bBb$

# Derivační strom

$A \rightarrow aBBb \mid AaA$

$\underline{B} \rightarrow \underline{\epsilon} \mid bCA$

$C \rightarrow AB \mid a \mid b$



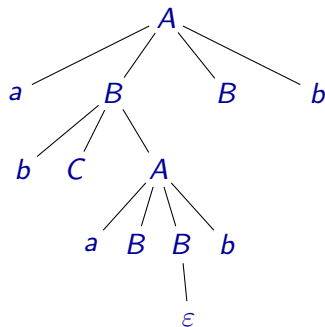
$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaB\underline{B}bBb \Rightarrow abCaBbBb$

# Derivační strom

$A \rightarrow aBBb \mid AaA$

$B \rightarrow \varepsilon \mid bCA$

$C \rightarrow AB \mid a \mid b$



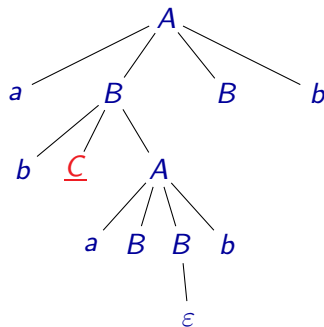
$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb$

# Derivační strom

$A \rightarrow aBBb \mid AaA$

$B \rightarrow \varepsilon \mid bCA$

$\underline{C} \rightarrow AB \mid a \mid b$



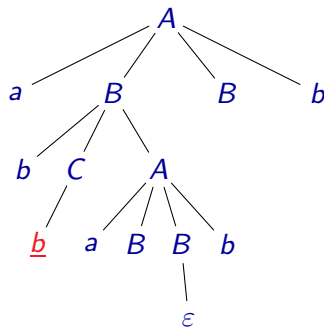
$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow ab\underline{C}aBbBb$

# Derivační strom

$A \rightarrow aBBb \mid AaA$

$B \rightarrow \varepsilon \mid bCA$

$\underline{C} \rightarrow AB \mid a \mid \underline{b}$



$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow ab\underline{C}aBbBb \Rightarrow ab\underline{b}aBbBb$

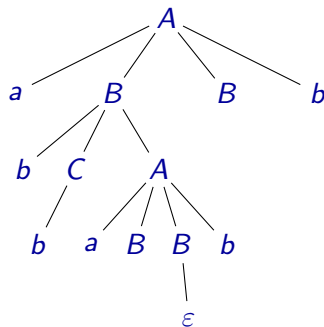


# Derivační strom

$A \rightarrow aBBb \mid AaA$

$B \rightarrow \varepsilon \mid bCA$

$C \rightarrow AB \mid a \mid b$



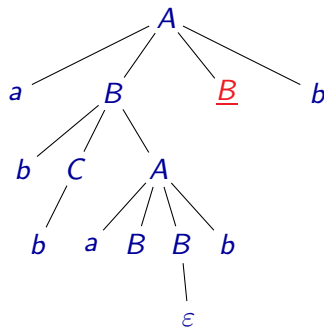
$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBbBb$

# Derivační strom

$A \rightarrow aBBb \mid AaA$

$\underline{B} \rightarrow \varepsilon \mid bCA$

$C \rightarrow AB \mid a \mid b$



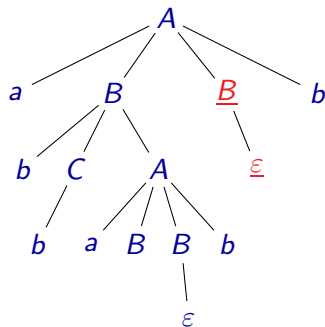
$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBb\underline{B}b$

# Derivační strom

$A \rightarrow aBBb \mid AaA$

$\underline{B} \rightarrow \underline{\epsilon} \mid bCA$

$C \rightarrow AB \mid a \mid b$



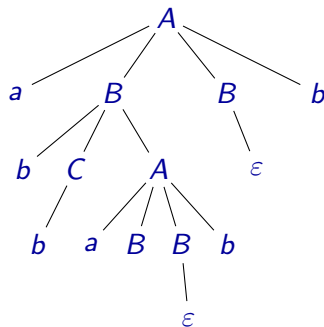
$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBb\underline{B}b \Rightarrow abbaBbb$

# Derivační strom

$A \rightarrow aBBb \mid AaA$

$B \rightarrow \varepsilon \mid bCA$

$C \rightarrow AB \mid a \mid b$



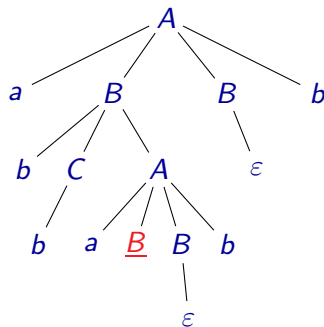
$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBbBb \Rightarrow abbaBbb$

# Derivační strom

$A \rightarrow aBBb \mid AaA$

$\underline{B} \rightarrow \varepsilon \mid bCA$

$C \rightarrow AB \mid a \mid b$



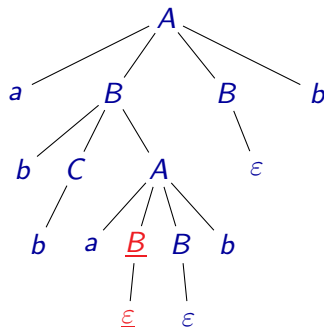
$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBbBb \Rightarrow abba\underline{B}bb$

# Derivační strom

$A \rightarrow aBBb \mid AaA$

$\underline{B} \rightarrow \underline{\epsilon} \mid bCA$

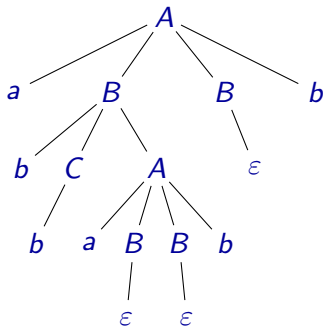
$C \rightarrow AB \mid a \mid b$



$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBbBb \Rightarrow abba\underline{B}bb \Rightarrow abbabb$

# Derivační strom

$A \rightarrow aBBb \mid AaA$   
 $B \rightarrow \varepsilon \mid bCA$   
 $C \rightarrow AB \mid a \mid b$



$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBbBb \Rightarrow abbaBbb \Rightarrow abbabb$

Každé derivaci odpovídá nějaký **derivační strom**:

- Vrcholy stromu jsou ohodnoceny terminály a neterminály.
- Kořen stromu je ohodnocen počátečním neterminálem.
- Listy stromu jsou ohodnoceny terminály nebo symboly  $\varepsilon$ .
- Ostatní vrcholy stromu jsou ohodnoceny neterminály.
- Pokud je vrchol ohodnocen neterminálem  $A$ , pak jeho potomci jsou ohodnoceni symboly pravé strany nějakého přepisovacího pravidla  $A \rightarrow \alpha$ .



$$E \rightarrow E + E \mid E * E \mid (E) \mid a$$

**Levá derivace** je derivace, ve které v každém kroku nahrazujeme vždy nejlevější neterminál.

$$\underline{E} \Rightarrow \underline{E} + E \Rightarrow \underline{E} * E + E \Rightarrow a * \underline{E} + E \Rightarrow a * a + \underline{E} \Rightarrow a * a + a$$

**Pravá derivace** je derivace, ve které v každém kroku nahrazujeme vždy nejpravější neterminál.

$$\underline{E} \Rightarrow E + \underline{E} \Rightarrow \underline{E} + a \Rightarrow E * \underline{E} + a \Rightarrow \underline{E} * a + a \Rightarrow a * a + a$$

Derivace však nemusí být ani levá ani pravá:

$$\underline{E} \Rightarrow \underline{E} + E \Rightarrow E * \underline{E} + E \Rightarrow E * a + \underline{E} \Rightarrow \underline{E} * a + a \Rightarrow a * a + a$$

- Jednomu derivačnímu stromu může odpovídat více různých derivací.
- Každému derivačnímu stromu odpovídá právě jedna levá a právě jedna pravá derivace.

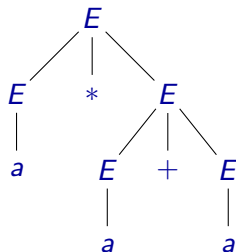
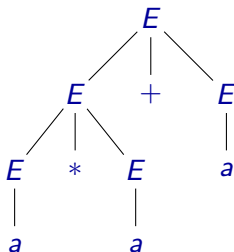
# Nejednoznačné gramatiky

Gramatika  $G$  je **nejednoznačná**, jestliže existuje nějaké slovo  $w \in L(G)$ , kterému přísluší dva různé derivační stromy, resp. dvě různé levé či dvě různé pravé derivace.

## Příklad:

$E \Rightarrow E + E \Rightarrow E * E + E \Rightarrow a * E + E \Rightarrow a * a + E \Rightarrow a * a + a$

$E \Rightarrow E * E \Rightarrow E * E + E \Rightarrow a * E + E \Rightarrow a * a + E \Rightarrow a * a + a$



# Nejednoznačné gramatiky

Někdy je možné nejednoznačnou gramatiku nahradit gramatikou, která generuje tentýž jazyk, ale není nejednoznačná.

**Příklad:** Gramatiku

$$E \rightarrow E + E \mid E * E \mid (E) \mid a$$

můžeme nahradit ekvivalentní gramatikou

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

**Poznámka:** Pokud se nejednoznačná gramatika žádnou ekvivalentní jednoznačnou gramatikou nahradit nedá, říkáme, že je **podstatně nejednoznačná**.

Gramatiky  $G_1$  a  $G_2$  jsou **ekvivalentní**, jestliže generují tentýž jazyk, tj. jestliže  $L(G_1) = L(G_2)$ .

**Poznámka:** Problém ekvivalence bezkontextových gramatik je algoritmicky nerozhodnutelný. Dá se dokázat, že není možné vytvořit algoritmus, který by pro libovolné dvě bezkontextové gramatiky rozhodl, zda jsou ekvivalentní či ne.

Dokonce je algoritmicky nerozhodnutelný i problém, zda gramatika generuje jazyk  $\Sigma^*$ .

## Definice

Jazyk  $L$  je **bezkontextový**, jestliže existuje bezkontextová gramatika  $G$  taková, že  $L = L(G)$ .

Třída bezkontextových jazyků je uzavřená vůči:

- zřetězení
- sjednocení
- iteraci

Třída bezkontextových jazyků však není uzavřená vůči:

- doplňku
- průniku

Máme dány gramatiky  $G_1 = (\Pi_1, \Sigma, S_1, P_1)$  a  $G_2 = (\Pi_2, \Sigma, S_2, P_2)$ , přičemž můžeme předpokládat, že  $\Pi_1 \cap \Pi_2 = \emptyset$  a  $S \notin \Pi_1 \cup \Pi_2$ .

- Gramatika  $G$  taková, že  $L(G) = L(G_1)L(G_2)$ :

$$G = (\Pi_1 \cup \Pi_2 \cup \{S\}, \Sigma, S, P_1 \cup P_2 \cup \{S \rightarrow S_1S_2\})$$

- Gramatika  $G$  taková, že  $L(G) = L(G_1) \cup L(G_2)$ :

$$G = (\Pi_1 \cup \Pi_2 \cup \{S\}, \Sigma, S, P_1 \cup P_2 \cup \{S \rightarrow S_1, S \rightarrow S_2\})$$

- Gramatika  $G$  taková, že  $L(G) = L(G_1)^*$ :

$$G = (\Pi_1 \cup \{S\}, \Sigma, S, P_1 \cup \{S \rightarrow \varepsilon, S \rightarrow S_1S\})$$

## Definice

Bezkontextová gramatika  $G = (\Pi, \Sigma, S, P)$  je **redukovaná**, jestliže:

- Každý neterminál  $X \in \Pi$  je možné přepsat na sekvenci terminálů, tj. existuje  $w \in \Sigma^*$  takové, že  $X \Rightarrow^* w$ .
- Každý neterminál  $X \in \Pi$  je dosažitelný z počátečního neterminálu  $S$ , tj. existují  $\alpha, \beta \in (\Pi \cup \Sigma)^*$  takové, že  $S \Rightarrow^* \alpha X \beta$ .

Ke každé bezkontextové gramatice je možné sestrojít ekvivalentní redukovanou gramatiku.



# Redukované gramatiky

Algoritmus, který zkonstruuje množinu

$$\mathcal{T} = \{X \in \Pi \mid \exists w \in \Sigma^* : X \Rightarrow^* w\}$$

```
1  $\mathcal{T} \leftarrow \emptyset$ 
2  $change \leftarrow \text{TRUE}$ 
3 while  $change$ 
4     do  $change \leftarrow \text{FALSE}$ 
5         for each  $(X \rightarrow \alpha) \in P$ 
6             do if  $X \notin \mathcal{T}$  and  $\alpha \in (\mathcal{T} \cup \Sigma)^*$ 
7                 then  $\mathcal{T} \leftarrow \mathcal{T} \cup \{X\}$ 
8                      $change \leftarrow \text{TRUE}$ 
9 return  $\mathcal{T}$ 
```

Z gramatiky  $G$  pak vytvoříme gramatiku  $G'$  tak, že z  $G$  odstraníme všechny neterminály nepatřící do  $\mathcal{T}$  a pravidla, kde se vyskytují.

**Příklad:** Gramatika  $G = (\{S, A, B, C\}, \{a, b\}, S, P)$

$$S \rightarrow A \mid B$$

$$A \rightarrow aB \mid bS \mid b$$

$$B \rightarrow AB \mid Ba$$

$$C \rightarrow AS \mid b$$

**Příklad:** Gramatika  $G = (\{S, A, B, C\}, \{a, b\}, S, P)$

$$S \rightarrow A \mid B$$

$$A \rightarrow aB \mid bS \mid b$$

$$B \rightarrow AB \mid Ba$$

$$C \rightarrow AS \mid b$$

$$\mathcal{T} = \{$$

**Příklad:** Gramatika  $G = (\{S, A, B, C\}, \{a, b\}, S, P)$

$$S \rightarrow A \mid B$$

$$A \rightarrow aB \mid bS \mid b$$

$$B \rightarrow AB \mid Ba$$

$$C \rightarrow AS \mid b$$

$$\mathcal{T} = \{A,$$

**Příklad:** Gramatika  $G = (\{S, A, B, C\}, \{a, b\}, S, P)$

$$S \rightarrow A \mid B$$

$$A \rightarrow aB \mid bS \mid b$$

$$B \rightarrow AB \mid Ba$$

$$C \rightarrow AS \mid b$$

$$\mathcal{T} = \{A, C,$$

**Příklad:** Gramatika  $G = (\{S, A, B, C\}, \{a, b\}, S, P)$

$$S \rightarrow A \mid B$$

$$A \rightarrow aB \mid bS \mid b$$

$$B \rightarrow AB \mid Ba$$

$$C \rightarrow AS \mid b$$

$$\mathcal{T} = \{A, C, S\}$$

**Příklad:** Gramatika  $G = (\{S, A, B, C\}, \{a, b\}, S, P)$

$$S \rightarrow A \mid B$$

$$A \rightarrow aB \mid bS \mid b$$

$$B \rightarrow AB \mid Ba$$

$$C \rightarrow AS \mid b$$

$$\mathcal{T} = \{A, C, S\}$$

**Příklad:** Gramatika  $G = (\{S, A, B, C\}, \{a, b\}, S, P)$

$$S \rightarrow A \mid B$$

$$A \rightarrow aB \mid bS \mid b$$

$$B \rightarrow AB \mid Ba$$

$$C \rightarrow AS \mid b$$

$$\mathcal{T} = \{A, C, S\}$$

Gramatika  $G' = (\{S, A, C\}, \{a, b\}, S, P')$

$$S \rightarrow A$$

$$A \rightarrow bS \mid b$$

$$C \rightarrow AS \mid b$$



# Redukované gramatiky

Algoritmus, který zkonstruuje množinu

$$\mathcal{D} = \{X \in \Pi' \mid \exists \alpha, \beta \in (\Pi' \cup \Sigma)^* : S \Rightarrow^* \alpha X \beta\}$$

```
1   $\mathcal{D} \leftarrow \{S\}$ 
2  change  $\leftarrow$  TRUE
3  while change
4      do change  $\leftarrow$  FALSE
5          for each  $(X \rightarrow \alpha) \in P'$  where  $X \in \mathcal{D}$ 
6              do for  $i \leftarrow 1$  to  $|\alpha|$ 
7                  do if  $\alpha[i] \in \Pi'$  and  $\alpha[i] \notin \mathcal{D}$ 
8                      then  $\mathcal{D} \leftarrow \mathcal{D} \cup \{\alpha[i]\}$ 
9                          change  $\leftarrow$  TRUE
10 return  $\mathcal{D}$ 
```

Z gramatiky  $G'$  pak vytvoříme gramatiku  $G''$  tak, že z  $G'$  odstraníme všechny neterminály nepatřící do  $\mathcal{D}$  a pravidla, kde se vyskytují.

**Příklad:** Gramatika  $G' = (\{S, A, C\}, \{a, b\}, S, P')$

$$S \rightarrow A$$

$$A \rightarrow bS \mid b$$

$$C \rightarrow AS \mid b$$

**Příklad:** Gramatika  $G' = (\{S, A, C\}, \{a, b\}, S, P')$

$$S \rightarrow A$$

$$A \rightarrow bS \mid b$$

$$C \rightarrow AS \mid b$$

$$\mathcal{D} = \{$$

**Příklad:** Gramatika  $G' = (\{S, A, C\}, \{a, b\}, S, P')$

$$S \rightarrow A$$

$$A \rightarrow bS \mid b$$

$$C \rightarrow AS \mid b$$

$$\mathcal{D} = \{S,$$

**Příklad:** Gramatika  $G' = (\{S, A, C\}, \{a, b\}, S, P')$

$$S \rightarrow A$$

$$A \rightarrow bS \mid b$$

$$C \rightarrow AS \mid b$$

$$\mathcal{D} = \{S, A$$

**Příklad:** Gramatika  $G' = (\{S, A, C\}, \{a, b\}, S, P')$

$$S \rightarrow A$$

$$A \rightarrow bS \mid b$$

$$C \rightarrow AS \mid b$$

$$\mathcal{D} = \{S, A\}$$

**Příklad:** Gramatika  $G' = (\{S, A, C\}, \{a, b\}, S, P')$

$$S \rightarrow A$$

$$A \rightarrow bS \mid b$$

$$C \rightarrow AS \mid b$$

$$\mathcal{D} = \{S, A\}$$

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

$$S \rightarrow A$$

$$A \rightarrow bS \mid b$$

# Redukované gramatiky

Pořadí obou kroků je třeba dodržet. Pokud bychom je provedli v opačném pořadí, můžeme dostat gramatiku, která není redukovaná.

**Příklad:**

$$\begin{aligned} S &\rightarrow a \mid A \\ A &\rightarrow AB \\ B &\rightarrow b \end{aligned}$$

Pokud provedeme oba kroky algoritmu ve správném pořadí, dostaneme

$$S \rightarrow a$$

Pokud provedeme kroky algoritmu v opačném pořadí, dostaneme

$$\begin{aligned} S &\rightarrow a \\ B &\rightarrow b \end{aligned}$$



Předchozí algoritmus lze použít ke zjištění, zda  $L(G) \neq \emptyset$ .

Stačí ověřit, zda  $S \in \mathcal{T}$ :

- Pokud ano, je  $L(G) \neq \emptyset$ .
- Pokud ne, je  $L(G) = \emptyset$ .

## Definice

Gramatika  $G$  je **nevypouštějící**, jestliže neobsahuje žádná  $\varepsilon$ -pravidla, tj. pravidla tvaru  $X \rightarrow \varepsilon$ , kde  $X \in \Pi$ .

Ke každé bezkontextové gramatice je možné sestrojít nevypouštějící gramatiku  $G'$  takovou, že  $L(G') = L(G) - \{\varepsilon\}$ .

Algoritmus, který zkonstruuje množinu

$$\mathcal{E} = \{X \in \Pi \mid X \Rightarrow^* \varepsilon\}$$

```
1   $\mathcal{E} \leftarrow \emptyset$ 
2  change  $\leftarrow$  TRUE
3  while change
4      do change  $\leftarrow$  FALSE
5          for each  $(X \rightarrow \alpha) \in P$ 
6              do if  $X \notin \mathcal{E}$  and  $\alpha \in \mathcal{E}^*$ 
7                  then  $\mathcal{E} \leftarrow \mathcal{E} \cup \{X\}$ 
8                      change  $\leftarrow$  TRUE
9  return  $\mathcal{E}$ 
```

# Nevypouštějící gramatiky

Nevypouštějící gramatiku  $G'$  pak z  $G$  vytvoříme tak, že pro každé pravidlo  $X \rightarrow \alpha$  z  $G$  přidáme do  $G'$  všechna možná pravidla

$$X \rightarrow \alpha'$$

kde  $\alpha'$  vznikne z  $\alpha$  vypuštěním libovolného počtu symbolů z  $\mathcal{E}$ , přičemž ale  $\alpha' \neq \varepsilon$ .

**Příklad:** Pokud například  $\mathcal{E} = \{A, C, D\}$ , pak místo pravidla

$$A \rightarrow aASCbA$$

přidáme pravidla

$$A \rightarrow aASCbA \mid aSCbA \mid aASbA \mid aASCb \mid aSbA \mid aSCb \mid aASb \mid aSb$$

# Nevypouštějící gramatiky

**Příklad:** Gramatika  $G = (\{S, A, B, C\}, \{a, b\}, S, P)$

$$S \rightarrow AB \mid \varepsilon$$

$$A \rightarrow aAAb \mid BS \mid CA$$

$$B \rightarrow BbA \mid CaC \mid \varepsilon$$

$$C \rightarrow aBB \mid bS$$

# Nevypouštějící gramatiky

**Příklad:** Gramatika  $G = (\{S, A, B, C\}, \{a, b\}, S, P)$

$$S \rightarrow AB \mid \varepsilon$$

$$A \rightarrow aAAb \mid BS \mid CA$$

$$B \rightarrow BbA \mid CaC \mid \varepsilon$$

$$C \rightarrow aBB \mid bS$$

$$\mathcal{E} = \{$$

# Nevypouštějící gramatiky

**Příklad:** Gramatika  $G = (\{S, A, B, C\}, \{a, b\}, S, P)$

$$S \rightarrow AB \mid \varepsilon$$

$$A \rightarrow aAAb \mid BS \mid CA$$

$$B \rightarrow BbA \mid CaC \mid \varepsilon$$

$$C \rightarrow aBB \mid bS$$

$$\mathcal{E} = \{S,$$

# Nevypouštějící gramatiky

**Příklad:** Gramatika  $G = (\{S, A, B, C\}, \{a, b\}, S, P)$

$$S \rightarrow AB \mid \varepsilon$$

$$A \rightarrow aAAb \mid BS \mid CA$$

$$B \rightarrow BbA \mid CaC \mid \varepsilon$$

$$C \rightarrow aBB \mid bS$$

$$\mathcal{E} = \{S, B,$$



# Nevypouštějící gramatiky

**Příklad:** Gramatika  $G = (\{S, A, B, C\}, \{a, b\}, S, P)$

$$S \rightarrow AB \mid \varepsilon$$

$$A \rightarrow aAAb \mid BS \mid CA$$

$$B \rightarrow BbA \mid CaC \mid \varepsilon$$

$$C \rightarrow aBB \mid bS$$

$$\mathcal{E} = \{S, B, A$$

# Nevypouštějící gramatiky

**Příklad:** Gramatika  $G = (\{S, A, B, C\}, \{a, b\}, S, P)$

$$S \rightarrow AB \mid \varepsilon$$

$$A \rightarrow aAAb \mid BS \mid CA$$

$$B \rightarrow BbA \mid CaC \mid \varepsilon$$

$$C \rightarrow aBB \mid bS$$

$$\mathcal{E} = \{S, B, A\}$$

# Nevypouštějící gramatiky

**Příklad:** Gramatika  $G = (\{S, A, B, C\}, \{a, b\}, S, P)$

$$\begin{aligned}S &\rightarrow AB \mid \varepsilon \\A &\rightarrow aAAb \mid BS \mid CA \\B &\rightarrow BbA \mid CaC \mid \varepsilon \\C &\rightarrow aBB \mid bS\end{aligned}$$

$$\mathcal{E} = \{S, B, A\}$$

Gramatika  $G' = (\{S, A, B, C\}, \{a, b\}, S, P')$

$$\begin{aligned}S &\rightarrow AB \mid A \mid B \\A &\rightarrow aAAb \mid aAb \mid ab \mid BS \mid B \mid S \mid CA \mid C \\B &\rightarrow BbA \mid bA \mid Bb \mid b \mid CaC \\C &\rightarrow aBB \mid aB \mid a \mid bS \mid b\end{aligned}$$

## Definice

Gramatika  $G$  je v **Chomského normální formě**, jestliže všechna její pravidla jsou pouze následujícího tvaru:

- $X \rightarrow YZ$ , kde  $X, Y, Z \in \Pi$ , nebo
- $X \rightarrow a$ , kde  $X \in \Pi$  a  $a \in \Sigma$ .

Ke každé bezkontextové gramatice je možné sestavit gramatiku  $G'$  v Chomského normální formě takovou, že  $L(G') = L(G) - \{\varepsilon\}$ .

## Definice

Gramatika  $G$  je v **Greibachové normální formě**, jestliže všechna její pravidla jsou pouze následujícího tvaru:

- $X \rightarrow a\alpha$ , kde  $a \in \Sigma$ ,  $X \in \Pi$  a  $\alpha \in \Pi^*$ .

Ke každé bezkontextové gramatice je možné sestrojít gramatiku  $G'$  v Greibachové normální formě takovou, že  $L(G') = L(G) - \{\varepsilon\}$ .

## Definice

Gramatika  $G$  je **regulární**, jestliže všechna její pravidla jsou tvaru:

- $X \rightarrow wY$ , kde  $X, Y \in \Pi$ ,  $w \in \Sigma^*$ , nebo
- $X \rightarrow w$ , kde  $X \in \Pi$ ,  $w \in \Sigma^*$ .

Regulární gramatiky generují právě třídu regulárních jazyků, tj.:

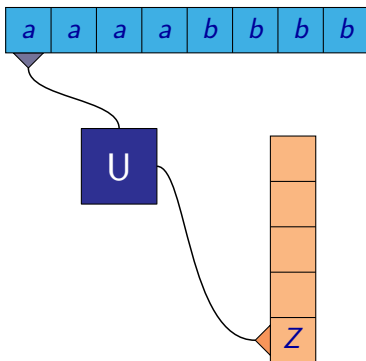
- Jazyk generovaný regulární gramatikou je vždy regulární.
- Každý regulární jazyk je generován nějakou regulární gramatikou.

- Chtěli bychom rozpoznávat jazyk  $L = \{a^i b^i \mid i \geq 1\}$
- Snažíme se navrhnout zařízení (podobné konečným automatům), které přečte slovo, a sdělí nám, zda toto slovo patří do jazyka  $L$  či ne.
- Při čtení  $a$ -ček si musíme pamatovat jejich počet, ať víme, kolik musí následovat  $b$ -ček

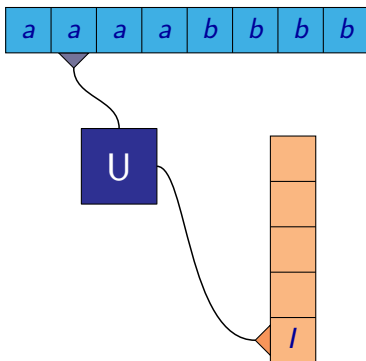
- Chtěli bychom rozpoznávat jazyk  $L = \{a^i b^i \mid i \geq 1\}$
- Snažíme se navrhnout zařízení (podobné konečným automatům), které přečte slovo, a sdělí nám, zda toto slovo patří do jazyka  $L$  či ne.
- Při čtení  $a$ -ček si musíme pamatovat jejich počet, ať víme, kolik musí následovat  $b$ -ček
- Můžeme využít paměť typu zásobník
- Každé přečtené  $a$  si na zásobník zapíšeme, za každé přečtené  $b$  jeden symbol ze zásobníku odstraníme
- Pokud bude zásobník prázdný a podaří se přečíst celé slovo, tak patří do jazyka



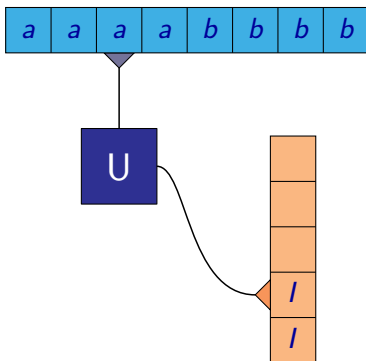
- Slovo *aaaabbbb* patří do jazyka  $L = \{a^i b^i \mid i \geq 1\}$



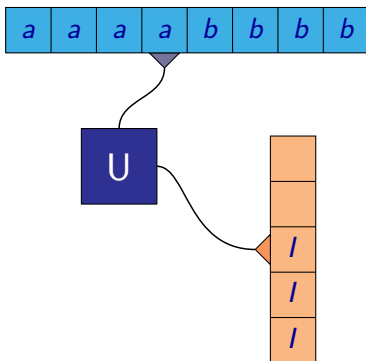
- Slovo *aaaabbbb* patří do jazyka  $L = \{a^i b^i \mid i \geq 1\}$



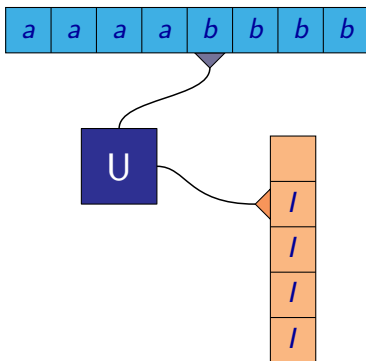
- Slovo *aaaabbbb* patří do jazyka  $L = \{a^i b^i \mid i \geq 1\}$



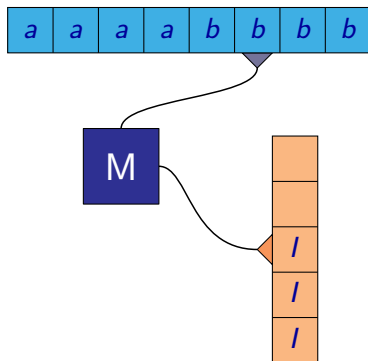
- Slovo *aaaabbbb* patří do jazyka  $L = \{a^i b^i \mid i \geq 1\}$



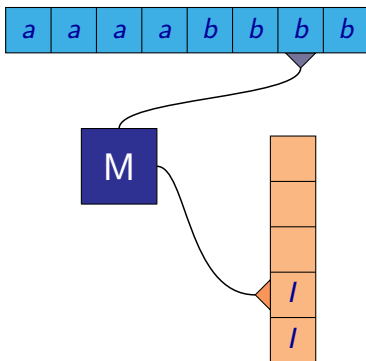
- Slovo *aaaabbbb* patří do jazyka  $L = \{a^i b^i \mid i \geq 1\}$



- Slovo *aaaabbbb* patří do jazyka  $L = \{a^i b^i \mid i \geq 1\}$

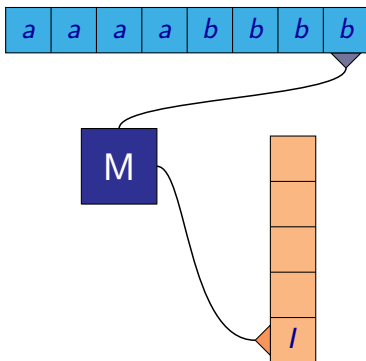


- Slovo *aaaabbbb* patří do jazyka  $L = \{a^i b^i \mid i \geq 1\}$



# Zásobníkový automat

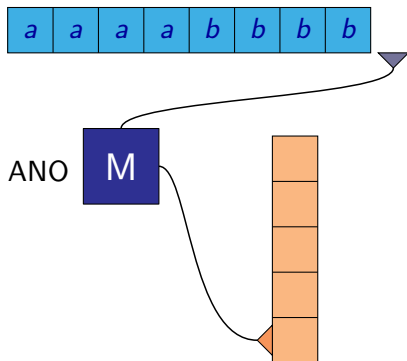
- Slovo *aaaabbbb* patří do jazyka  $L = \{a^i b^i \mid i \geq 1\}$



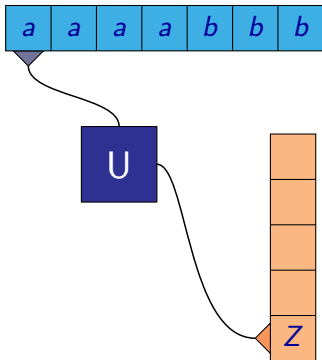


# Zásobníkový automat

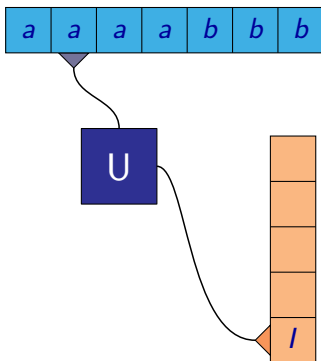
- Slovo *aaaabbbb* patří do jazyka  $L = \{a^i b^i \mid i \geq 1\}$
- Automat přečetl celé slovo a skončil s prázdným zásobníkem, takže slovo přijal



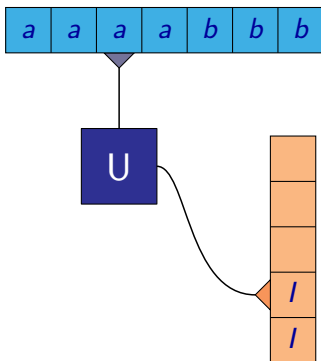
- Slovo *aaaabb* nepatří do jazyka  $L = \{a^i b^i \mid i \geq 1\}$



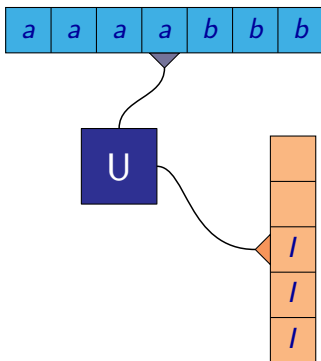
- Slovo *aaaabb* nepatří do jazyka  $L = \{a^i b^i \mid i \geq 1\}$



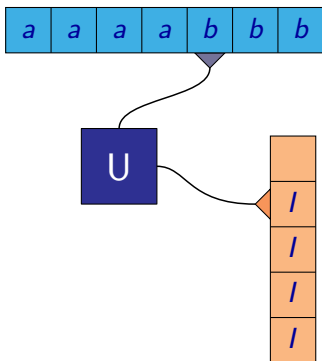
- Slovo *aaaabb* nepatří do jazyka  $L = \{a^i b^i \mid i \geq 1\}$



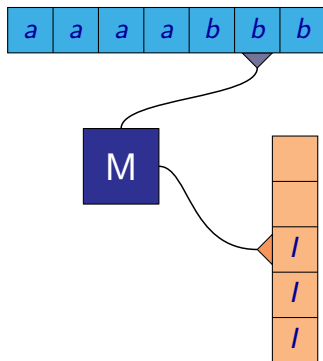
- Slovo *aaaabb* nepatří do jazyka  $L = \{a^i b^i \mid i \geq 1\}$



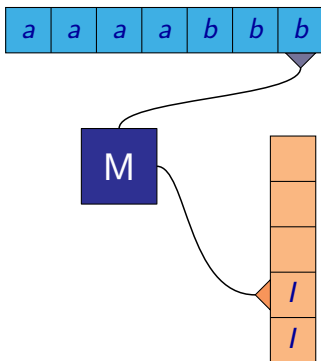
- Slovo *aaaabb* nepatří do jazyka  $L = \{a^i b^i \mid i \geq 1\}$



- Slovo *aaaabbb* nepatří do jazyka  $L = \{a^i b^i \mid i \geq 1\}$



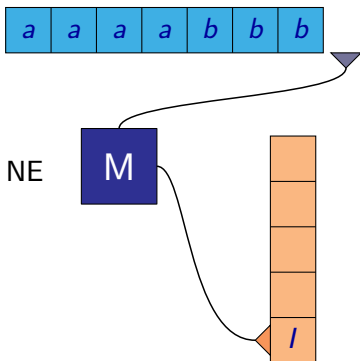
- Slovo *aaaabb* nepatří do jazyka  $L = \{a^i b^i \mid i \geq 1\}$



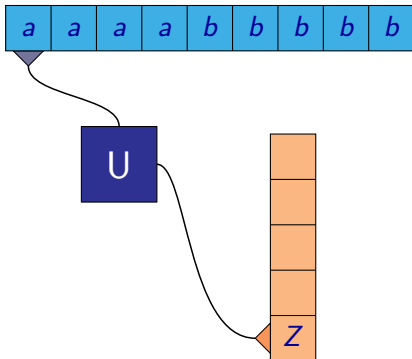


# Zásobníkový automat

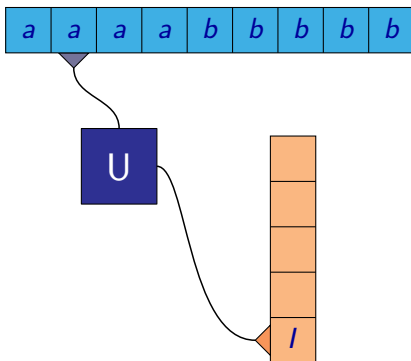
- Slovo *aaaabb* nepatří do jazyka  $L = \{a^i b^i \mid i \geq 1\}$
- Automat přečetl celé slovo, ale nevyprázdnil zásobník, takže slovo nepřijal



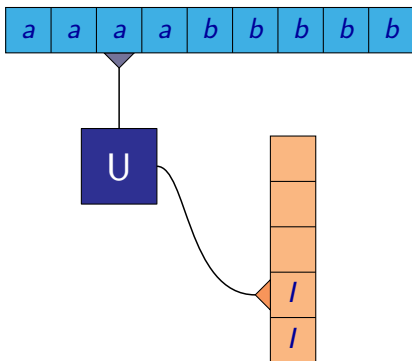
- Slovo *aaaabbbb* nepatří do jazyka  $L = \{a^i b^i \mid i \geq 1\}$



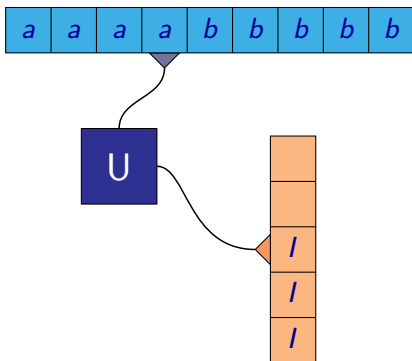
- Slovo *aaaabbbb* nepatří do jazyka  $L = \{a^i b^i \mid i \geq 1\}$



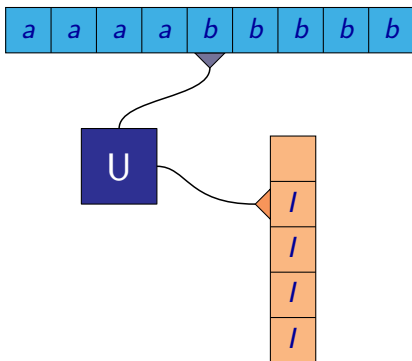
- Slovo *aaaabbbb* nepatří do jazyka  $L = \{a^i b^i \mid i \geq 1\}$



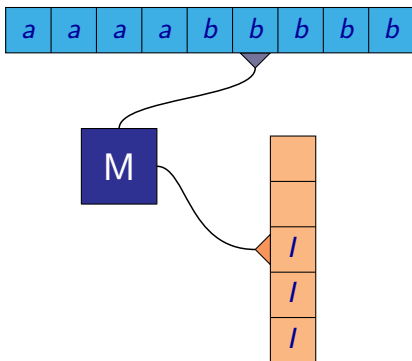
- Slovo *aaaabbbb* nepatří do jazyka  $L = \{a^i b^i \mid i \geq 1\}$



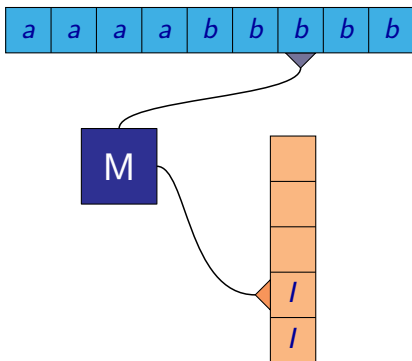
- Slovo *aaaabbbb* nepatří do jazyka  $L = \{a^i b^i \mid i \geq 1\}$



- Slovo *aaaabbbb* nepatří do jazyka  $L = \{a^i b^i \mid i \geq 1\}$

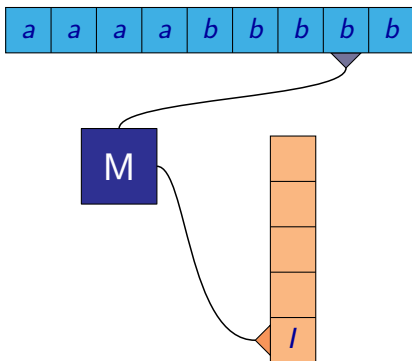


- Slovo *aaaabbbb* nepatří do jazyka  $L = \{a^i b^i \mid i \geq 1\}$



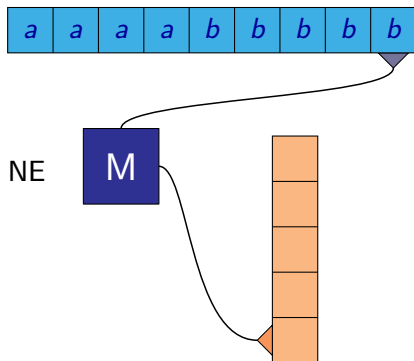


- Slovo *aaaabbbb* nepatří do jazyka  $L = \{a^i b^i \mid i \geq 1\}$

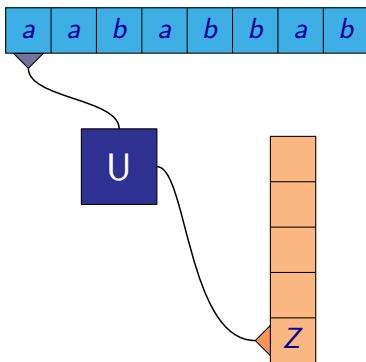


# Zásobníkový automat

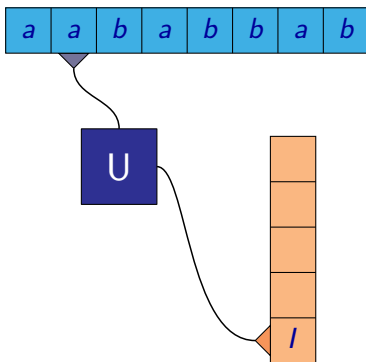
- Slovo  $aaaabbbb$  nepatří do jazyka  $L = \{a^i b^i \mid i \geq 1\}$
- Automat čte  $b$ , má smazat symbol na zásobníku a tam žádný není, takže slovo nepřijal



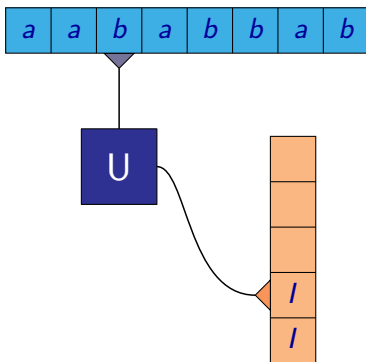
- Slovo *aababbab* nepatří do jazyka  $L = \{a^i b^i \mid i \geq 1\}$



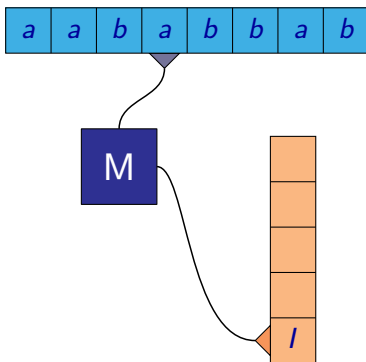
- Slovo *aababbab* nepatří do jazyka  $L = \{a^i b^i \mid i \geq 1\}$



- Slovo *aababbab* nepatří do jazyka  $L = \{a^i b^i \mid i \geq 1\}$

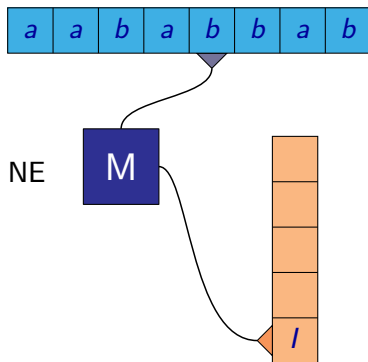


- Slovo *aababbab* nepatří do jazyka  $L = \{a^i b^i \mid i \geq 1\}$



# Zásobníkový automat

- Slovo *aababbab* nepatří do jazyka  $L = \{a^i b^i \mid i \geq 1\}$
- Automat přečetl *a*, ale již byl ve stavu, kdy maže, takže slovo nepřijal



# Zásobníkový automat

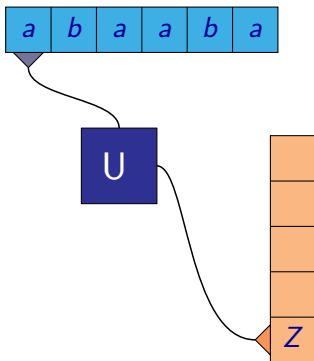
- Uvedený zásobníkový automat měl vždy jasně určeno pokračování - byl deterministický
- Je možné každý bezkontextový jazyk poznat deterministickým zásobníkovým automatem?



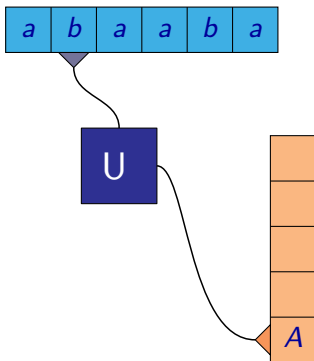
# Zásobníkový automat

- Uvedený zásobníkový automat měl vždy jasně určeno pokračování - byl deterministický
- Je možné každý bezkontextový jazyk poznat deterministickým zásobníkovým automatem?
- Uvažujme jazyk  $L = \{w(w)^R \mid w \in \{a, b\}^*\}$
- První půlku slova můžeme uložit na zásobník
- Při čtení druhé půlky mažeme symboly ze zásobníku, pokud jsou stejné jako na vstupu
- Pokud bude zásobník prázdný po přečtení celého slova, byla druhá půlka stejná jako první

- Slovo *abaaba* patří do jazyka  $L = \{w(w)^R \mid w \in \{a, b\}^*\}$

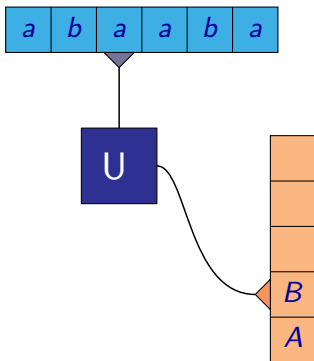


- Slovo *abaaba* patří do jazyka  $L = \{w(w)^R \mid w \in \{a, b\}^*\}$



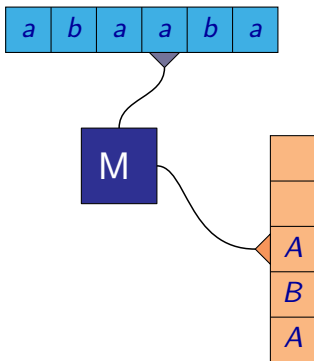
# Zásobníkový automat

- Slovo *abaaba* patří do jazyka  $L = \{w(w)^R \mid w \in \{a, b\}^*\}$
- Při čtení *a*, stavu *U* a vrcholu zásobníku *B*, musí změnit stav na *M* a uložit *A* na zásobník



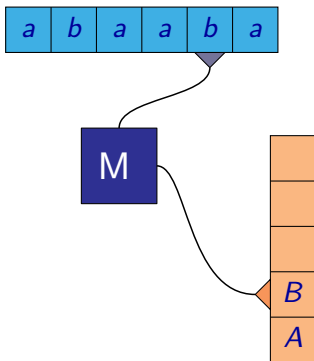
# Zásobníkový automat

- Slovo *abaaba* patří do jazyka  $L = \{w(w)^R \mid w \in \{a, b\}^*\}$
- Při čtení *a*, stavu *U* a vrcholu zásobníku *B*, musí změnit stav na *M* a uložit *A* na zásobník



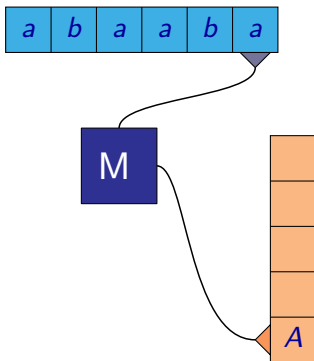
# Zásobníkový automat

- Slovo *abaaba* patří do jazyka  $L = \{w(w)^R \mid w \in \{a, b\}^*\}$
- Při čtení *a*, stavu *U* a vrcholu zásobníku *B*, musí změnit stav na *M* a uložit *A* na zásobník



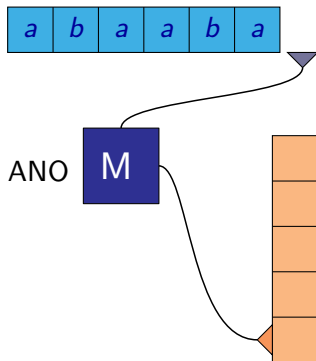
# Zásobníkový automat

- Slovo *abaaba* patří do jazyka  $L = \{w(w)^R \mid w \in \{a, b\}^*\}$
- Při čtení *a*, stavu *U* a vrcholu zásobníku *B*, musí změnit stav na *M* a uložit *A* na zásobník



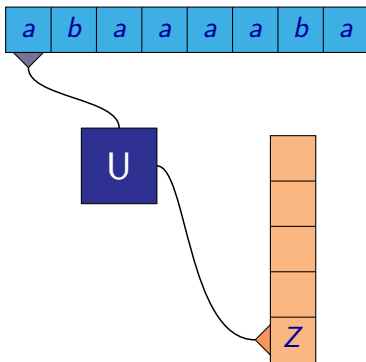
# Zásobníkový automat

- Slovo *abaaba* patří do jazyka  $L = \{w(w)^R \mid w \in \{a, b\}^*\}$
- Při čtení *a*, stavu *U* a vrcholu zásobníku *B*, musí změnit stav na *M* a uložit *A* na zásobník

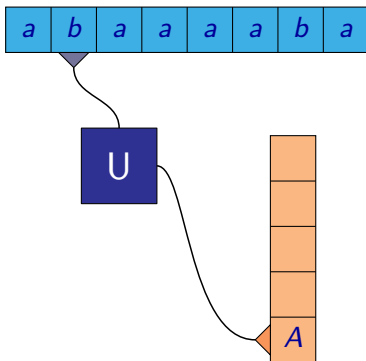




- Slovo *abaaaaba* patří do jazyka  $L = \{w(w)^R \mid w \in \{a, b\}^*\}$

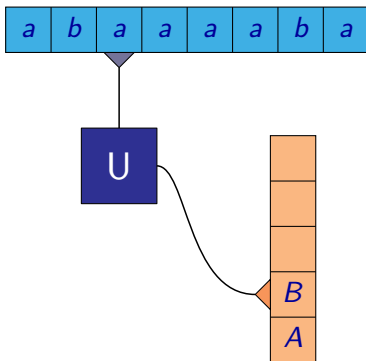


- Slovo *abaaaaba* patří do jazyka  $L = \{w(w)^R \mid w \in \{a, b\}^*\}$



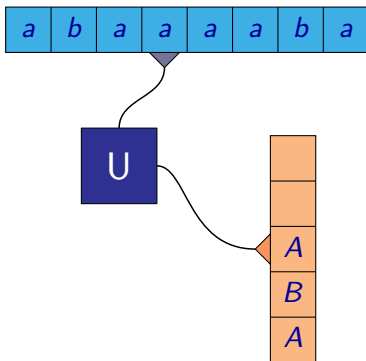
# Zásobníkový automat

- Slovo  $abaaaa$  patří do jazyka  $L = \{w(w)^R \mid w \in \{a, b\}^*\}$
- Při čtení  $a$ , stavu  $U$  a vrcholu zásobníku  $B$ , nemění stav a uloží  $A$  na zásobník



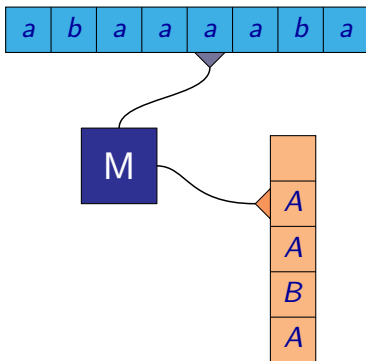
# Zásobníkový automat

- Slovo  $abaaaaba$  patří do jazyka  $L = \{w(w)^R \mid w \in \{a, b\}^*\}$
- Při čtení  $a$ , stavu  $U$  a vrcholu zásobníku  $B$ , nemění stav a uloží  $A$  na zásobník



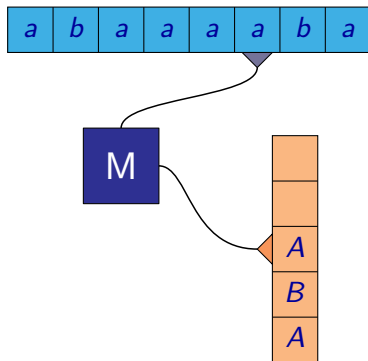
# Zásobníkový automat

- Slovo  $abaaaaba$  patří do jazyka  $L = \{w(w)^R \mid w \in \{a, b\}^*\}$
- Při čtení  $a$ , stavu  $U$  a vrcholu zásobníku  $B$ , nemění stav a uloží  $A$  na zásobník



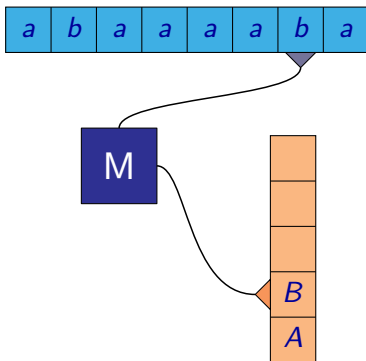
# Zásobníkový automat

- Slovo  $abaaaaba$  patří do jazyka  $L = \{w(w)^R \mid w \in \{a, b\}^*\}$
- Při čtení  $a$ , stavu  $U$  a vrcholu zásobníku  $B$ , nemění stav a uloží  $A$  na zásobník



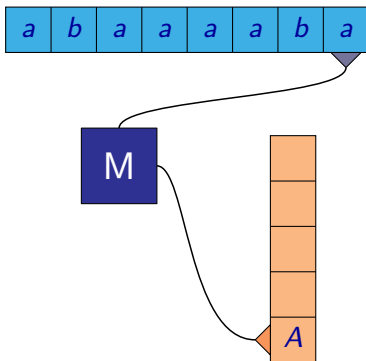
# Zásobníkový automat

- Slovo  $abaaaa$  patří do jazyka  $L = \{w(w)^R \mid w \in \{a, b\}^*\}$
- Při čtení  $a$ , stavu  $U$  a vrcholu zásobníku  $B$ , nemění stav a uloží  $A$  na zásobník



# Zásobníkový automat

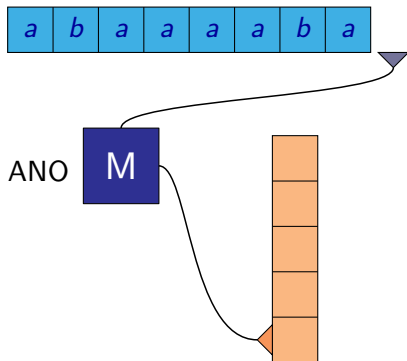
- Slovo  $abaaaa$  patří do jazyka  $L = \{w(w)^R \mid w \in \{a, b\}^*\}$
- Při čtení  $a$ , stavu  $U$  a vrcholu zásobníku  $B$ , nemění stav a uloží  $A$  na zásobník





# Zásobníkový automat

- Slovo  $abaaaa$  patří do jazyka  $L = \{w(w)^R \mid w \in \{a, b\}^*\}$
- Při čtení  $a$ , stavu  $U$  a vrcholu zásobníku  $B$ , nemění stav a uloží  $A$  na zásobník



- Uvedený zásobníkový automat se nemůže jednoznačně rozhodnout, jak má pokračovat. Musí „uhádnout“, kde je půlka slova.
- Na rozdíl od konečných automatů je deterministická verze zásobníkových slabší a proto definujeme přímo nedeterministické

## Definice

**Zásobníkový automat** je uspořádaná šestice  $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, Z_0)$ , kde

- $Q$  je konečná neprázdná množina stavů
- $\Sigma$  je konečná neprázdná množina zvaná vstupní abeceda
- $\Gamma$  je konečná neprázdná množina zvaná zásobníková abeceda
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \rightarrow 2^{(Q \times \Gamma^*)}$  je (nedeterministická) přechodová funkce
- $q_0 \in Q$  je počáteční stav
- $Z_0 \in \Gamma$  je počáteční zásobníkový symbol

# Zásobníkový automat

- **Konfigurace ZA** je trojice  $(q, w, \alpha)$ , kde  $q \in Q$ ,  $w \in \Sigma^*$ ,  $\alpha \in \Gamma^*$ .
- Konfiguraci  $(q_0, w, Z_0)$ , kde  $w \in \Sigma^*$ , nazýváme **počáteční**
- Konfiguraci  $(q, \varepsilon, \varepsilon)$ , kde  $q \in Q$ , nazýváme **koncová**

- **Konfigurace ZA** je trojice  $(q, w, \alpha)$ , kde  $q \in Q$ ,  $w \in \Sigma^*$ ,  $\alpha \in \Gamma^*$ .
- Konfiguraci  $(q_0, w, Z_0)$ , kde  $w \in \Sigma^*$ , nazýváme **počáteční**
- Konfiguraci  $(q, \varepsilon, \varepsilon)$ , kde  $q \in Q$ , nazýváme **koncová**
- Relaci  $\vdash$  mezi konfiguracemi definujeme tak, že  $(q, aw, X\beta) \vdash (q', w, \alpha\beta)$  právě když  $(q', \alpha) \in \delta(q, a, X)$  pro nějaké  $a \in (\Sigma \cup \{\varepsilon\})$ ,  $w \in \Sigma^*$ ,  $\beta \in \Gamma^*$ .
- Relace dosažitelnosti mezi konfiguracemi  $\vdash^*$  je reflexivní tranzitivní uzávěr relace  $\vdash$

- **Konfigurace ZA** je trojice  $(q, w, \alpha)$ , kde  $q \in Q$ ,  $w \in \Sigma^*$ ,  $\alpha \in \Gamma^*$ .
- Konfiguraci  $(q_0, w, Z_0)$ , kde  $w \in \Sigma^*$ , nazýváme **počáteční**
- Konfiguraci  $(q, \varepsilon, \varepsilon)$ , kde  $q \in Q$ , nazýváme **koncová**
- Relaci  $\vdash$  mezi konfiguracemi definujeme tak, že  $(q, aw, X\beta) \vdash (q', w, \alpha\beta)$  právě když  $(q', \alpha) \in \delta(q, a, X)$  pro nějaké  $a \in (\Sigma \cup \{\varepsilon\})$ ,  $w \in \Sigma^*$ ,  $\beta \in \Gamma^*$ .
- Relace dosažitelnosti mezi konfiguracemi  $\vdash^*$  je reflexivní tranzitivní uzávěr relace  $\vdash$
- Slovo  $w \in \Sigma^*$  je **přijímáno** ZA  $\mathcal{M}$  právě tehdy, když  $(q_0, w, Z_0) \vdash^* (q, \varepsilon, \varepsilon)$  pro nějaké  $q \in Q$
- **Jazyk rozpoznávaný ZA**  $\mathcal{M}$  je jazyk  $L(\mathcal{M}) = \{w \in \Sigma^* \mid w \text{ je přijímáno ZA } \mathcal{M}\}$ .

- Definovali jsme přijímání prázdným zásobníkem
- K ZA  $\mathcal{M}$  můžeme přidat množinu **přijímajících** (koncových) stavů  $F \subseteq Q$
- Potom můžeme definovat jazyk přijímaný koncovým stavem  $L_{KS}(\mathcal{M}) = \{w \in \Sigma^* \mid (q_0, w, Z_0) \vdash^* (q, \varepsilon, \alpha) \text{ pro něj. } q \in F, \alpha \in \Gamma^*\}$
- Obě zmíněné definice přijímání ZA jsou ekvivalentní - jazyk je přijíman nějakým ZA  $\mathcal{M}$  koncovým stavem právě tehdy, když je rozpoznáván nějakým ZA  $\mathcal{M}'$  prázdným zásobníkem

**Příklad:**  $L = \{a^i b^i \mid i \geq 1\}$

$\mathcal{M} = (Q, \Sigma, \Gamma, \delta, U, Z)$ , kde

- $Q = \{U, M\}$
- $\Sigma = \{a, b\}$
- $\Gamma = \{Z, I\}$

$$\delta(U, a, Z) = \{(U, I)\} \quad \delta(U, b, Z) = \emptyset$$

- $\delta(U, a, I) = \{(U, II)\} \quad \delta(U, b, I) = \{(M, \varepsilon)\}$   
 $\delta(M, a, I) = \emptyset \quad \delta(M, b, I) = \{(M, \varepsilon)\}$   
 $\delta(M, a, Z) = \emptyset \quad \delta(M, b, Z) = \emptyset$

**Poznámka:** Často se uvádí jen ty přechody přechodové funkce, které nejsou do prázdné množiny, tedy kdy je skutečně nějaký přechod definován



**Příklad:**  $L = \{w(w)^R \mid w \in \{a, b\}^*\}$

$\mathcal{M} = (Q, \Sigma, \Gamma, \delta, U, Z)$ , kde

- $Q = \{U, M\}$
- $\Sigma = \{a, b\}$
- $\Gamma = \{Z, A, B\}$

•

$$\delta(U, a, Z) = \{(U, A)\}$$

$$\delta(U, a, A) = \{(U, AA), (M, AA)\}$$

$$\delta(U, a, B) = \{(U, AB), (M, AB)\}$$

$$\delta(M, a, A) = \{(M, \varepsilon)\}$$

$$\delta(M, a, B) = \emptyset$$

$$\delta(M, a, Z) = \emptyset$$

$$\delta(U, \varepsilon, Z) = \{(U, \varepsilon)\}$$

$$\delta(U, \varepsilon, A) = \emptyset$$

$$\delta(U, \varepsilon, B) = \emptyset$$

$$\delta(U, b, Z) = \{(U, B)\}$$

$$\delta(U, b, A) = \{(U, BA), (M, BA)\}$$

$$\delta(U, b, B) = \{(U, BB), (M, BB)\}$$

$$\delta(M, b, A) = \emptyset$$

$$\delta(M, b, B) = \{(M, \varepsilon)\}$$

$$\delta(M, b, Z) = \emptyset$$

$$\delta(M, \varepsilon, Z) = \emptyset$$

$$\delta(M, \varepsilon, A) = \emptyset$$

$$\delta(M, \varepsilon, B) = \emptyset$$

## Lemma

Ke každé bezkontextové gramatice  $G$  lze sestavit zásobníkový automat  $\mathcal{M}$  (s jedním stavem) tž.  $L(\mathcal{M}) = L(G)$ .

**Důkaz:** Pro BG  $G = (\Pi, \Sigma, S, P)$  vytvoříme  $\mathcal{M} = (\{q_0\}, \Sigma, \Pi \cup \Sigma, \delta, q_0, S)$ , kde

- pro  $X \in \Pi$ :  $\delta(q_0, \varepsilon, X) = \{(q_0, \alpha) \mid (X \rightarrow \alpha) \in P\}$ ,
- pro  $a \in \Sigma$ :  $\delta(q_0, a, a) = \{(q_0, \varepsilon)\}$ ,
- v ostatních případech přiřadíme  $\emptyset$

Indukcí můžeme dokázat

$$S \Rightarrow_G^* u\alpha \Leftrightarrow_{df} (q_0, u, S) \vdash_{\mathcal{M}}^* (q_0, \varepsilon, \alpha)$$

kde  $u \in \Sigma^*$ ,  $\alpha \in \{\varepsilon\} \cup \Pi(\Pi \cup \Sigma)^*$

**Příklad:**  $L = \{a^i b^i \mid i \geq 1\}$

$G = (\{S\}, \{a, b\}, S, P)$ , kde  $P$  je definována jako:  $S \rightarrow aSb \mid ab$

**Příklad:**  $L = \{a^i b^i \mid i \geq 1\}$

$G = (\{S\}, \{a, b\}, S, P)$ , kde  $P$  je definována jako:  $S \rightarrow aSb \mid ab$

Sestrojíme  $\mathcal{M} = (\{q_0\}, \{a, b\}, \{S, a, b\}, \delta, q_0, S)$ , kde

$\delta(q_0, \varepsilon, S) = \{(q_0, aSb), (q_0, ab)\}$ ,  $\delta(q_0, a, a) = \{(q_0, \varepsilon)\}$ ,

$\delta(q_0, b, b) = \{(q_0, \varepsilon)\}$

**Příklad:**  $L = \{a^i b^i \mid i \geq 1\}$

$G = (\{S\}, \{a, b\}, S, P)$ , kde  $P$  je definována jako:  $S \rightarrow aSb \mid ab$

Sestrojíme  $\mathcal{M} = (\{q_0\}, \{a, b\}, \{S, a, b\}, \delta, q_0, S)$ , kde

$\delta(q_0, \varepsilon, S) = \{(q_0, aSb), (q_0, ab)\}$ ,  $\delta(q_0, a, a) = \{(q_0, \varepsilon)\}$ ,

$\delta(q_0, b, b) = \{(q_0, \varepsilon)\}$

Ukážeme si situaci pro  $aaaabbbb$

$S \Rightarrow aSb$

$(q_0, aaaabbbb, S) \vdash (q_0, aaaabbbb, aSb)$

**Příklad:**  $L = \{a^i b^i \mid i \geq 1\}$

$G = (\{S\}, \{a, b\}, S, P)$ , kde  $P$  je definována jako:  $S \rightarrow aSb \mid ab$

Sestrojíme  $\mathcal{M} = (\{q_0\}, \{a, b\}, \{S, a, b\}, \delta, q_0, S)$ , kde

$\delta(q_0, \varepsilon, S) = \{(q_0, aSb), (q_0, ab)\}$ ,  $\delta(q_0, a, a) = \{(q_0, \varepsilon)\}$ ,

$\delta(q_0, b, b) = \{(q_0, \varepsilon)\}$

Ukážeme si situaci pro  $aaaabbbb$

$S \Rightarrow aSb$

$(q_0, aaaabbbb, S) \vdash (q_0, aaaabbbb, aSb) \vdash (q_0, aaabbbb, Sb)$

**Příklad:**  $L = \{a^i b^i \mid i \geq 1\}$

$G = (\{S\}, \{a, b\}, S, P)$ , kde  $P$  je definována jako:  $S \rightarrow aSb \mid ab$

Sestrojíme  $\mathcal{M} = (\{q_0\}, \{a, b\}, \{S, a, b\}, \delta, q_0, S)$ , kde

$\delta(q_0, \varepsilon, S) = \{(q_0, aSb), (q_0, ab)\}$ ,  $\delta(q_0, a, a) = \{(q_0, \varepsilon)\}$ ,

$\delta(q_0, b, b) = \{(q_0, \varepsilon)\}$

Ukážeme si situaci pro *aaaabbbb*

$S \Rightarrow aSb \Rightarrow aaSbb$

$(q_0, aaaabbbb, S) \vdash (q_0, aaaabbbb, aSb) \vdash (q_0, aaabbbb, Sb)$

$\vdash (q_0, aaabbbb, aSbb)$

**Příklad:**  $L = \{a^i b^i \mid i \geq 1\}$

$G = (\{S\}, \{a, b\}, S, P)$ , kde  $P$  je definována jako:  $S \rightarrow aSb \mid ab$

Sestrojíme  $\mathcal{M} = (\{q_0\}, \{a, b\}, \{S, a, b\}, \delta, q_0, S)$ , kde

$\delta(q_0, \varepsilon, S) = \{(q_0, aSb), (q_0, ab)\}$ ,  $\delta(q_0, a, a) = \{(q_0, \varepsilon)\}$ ,

$\delta(q_0, b, b) = \{(q_0, \varepsilon)\}$

Ukážeme si situaci pro  $aaaabbbb$

$S \Rightarrow aSb \Rightarrow aaSbb$

$(q_0, aaaabbbb, S) \vdash (q_0, aaaabbbb, aSb) \vdash (q_0, aaabbbb, Sb)$

$\vdash (q_0, aaabbbb, aSbb) \vdash (q_0, aabbbb, Sbb)$



**Příklad:**  $L = \{a^i b^i \mid i \geq 1\}$

$G = (\{S\}, \{a, b\}, S, P)$ , kde  $P$  je definována jako:  $S \rightarrow aSb \mid ab$

Sestrojíme  $\mathcal{M} = (\{q_0\}, \{a, b\}, \{S, a, b\}, \delta, q_0, S)$ , kde

$\delta(q_0, \varepsilon, S) = \{(q_0, aSb), (q_0, ab)\}$ ,  $\delta(q_0, a, a) = \{(q_0, \varepsilon)\}$ ,

$\delta(q_0, b, b) = \{(q_0, \varepsilon)\}$

Ukážeme si situaci pro  $aaaabbbb$

$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb$

$(q_0, aaaabbbb, S) \vdash (q_0, aaaabbbb, aSb) \vdash (q_0, aaabbbb, Sb)$

$\vdash (q_0, aaabbbb, aSbb) \vdash (q_0, aabbbb, Sbb) \vdash (q_0, aabbbb, aSbbb)$

**Příklad:**  $L = \{a^i b^i \mid i \geq 1\}$

$G = (\{S\}, \{a, b\}, S, P)$ , kde  $P$  je definována jako:  $S \rightarrow aSb \mid ab$

Sestrojíme  $\mathcal{M} = (\{q_0\}, \{a, b\}, \{S, a, b\}, \delta, q_0, S)$ , kde

$\delta(q_0, \varepsilon, S) = \{(q_0, aSb), (q_0, ab)\}$ ,  $\delta(q_0, a, a) = \{(q_0, \varepsilon)\}$ ,

$\delta(q_0, b, b) = \{(q_0, \varepsilon)\}$

Ukážeme si situaci pro  $aaaabbbb$

$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb$

$(q_0, aaaabbbb, S) \vdash (q_0, aaaabbbb, aSb) \vdash (q_0, aaabbbb, Sb)$

$\vdash (q_0, aaabbbb, aSbb) \vdash (q_0, aabbbb, Sbb) \vdash (q_0, aabbbb, aSbbb)$

$\vdash (q_0, abbbb, Sbbb)$

**Příklad:**  $L = \{a^i b^i \mid i \geq 1\}$

$G = (\{S\}, \{a, b\}, S, P)$ , kde  $P$  je definována jako:  $S \rightarrow aSb \mid ab$

Sestrojíme  $\mathcal{M} = (\{q_0\}, \{a, b\}, \{S, a, b\}, \delta, q_0, S)$ , kde

$\delta(q_0, \varepsilon, S) = \{(q_0, aSb), (q_0, ab)\}$ ,  $\delta(q_0, a, a) = \{(q_0, \varepsilon)\}$ ,

$\delta(q_0, b, b) = \{(q_0, \varepsilon)\}$

Ukážeme si situaci pro  $aaaabbbb$

$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaaabbbb$

$(q_0, aaaabbbb, S) \vdash (q_0, aaaabbbb, aSb) \vdash (q_0, aaabbbb, Sb)$

$\vdash (q_0, aaabbbb, aSbb) \vdash (q_0, aabbbb, Sbb) \vdash (q_0, aabbbb, aSbbb)$

$\vdash (q_0, abbbb, Sbbb) \vdash (q_0, abbbb, abbbb)$

**Příklad:**  $L = \{a^i b^i \mid i \geq 1\}$

$G = (\{S\}, \{a, b\}, S, P)$ , kde  $P$  je definována jako:  $S \rightarrow aSb \mid ab$

Sestrojíme  $\mathcal{M} = (\{q_0\}, \{a, b\}, \{S, a, b\}, \delta, q_0, S)$ , kde

$\delta(q_0, \varepsilon, S) = \{(q_0, aSb), (q_0, ab)\}$ ,  $\delta(q_0, a, a) = \{(q_0, \varepsilon)\}$ ,

$\delta(q_0, b, b) = \{(q_0, \varepsilon)\}$

Ukážeme si situaci pro  $aaaabbbb$

$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaaabbbb$

$(q_0, aaaabbbb, S) \vdash (q_0, aaaabbbb, aSb) \vdash (q_0, aaabbbb, Sb)$

$\vdash (q_0, aaabbbb, aSbb) \vdash (q_0, aabbbb, Sbb) \vdash (q_0, aabbbb, aSbbb)$

$\vdash (q_0, abbbb, Sbbb) \vdash (q_0, abbbb, abbbb) \vdash (q_0, bbbb, bbbb)$

**Příklad:**  $L = \{a^i b^i \mid i \geq 1\}$

$G = (\{S\}, \{a, b\}, S, P)$ , kde  $P$  je definována jako:  $S \rightarrow aSb \mid ab$

Sestrojíme  $\mathcal{M} = (\{q_0\}, \{a, b\}, \{S, a, b\}, \delta, q_0, S)$ , kde

$\delta(q_0, \varepsilon, S) = \{(q_0, aSb), (q_0, ab)\}$ ,  $\delta(q_0, a, a) = \{(q_0, \varepsilon)\}$ ,

$\delta(q_0, b, b) = \{(q_0, \varepsilon)\}$

Ukážeme si situaci pro  $aaaabbbb$

$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaaabbbb$

$(q_0, aaaabbbb, S) \vdash (q_0, aaaabbbb, aSb) \vdash (q_0, aaabbbb, Sb)$

$\vdash (q_0, aaabbbb, aSbb) \vdash (q_0, aabbbb, Sbb) \vdash (q_0, aabbbb, aSbbb)$

$\vdash (q_0, abbbb, Sbbb) \vdash (q_0, abbbb, abbbb) \vdash (q_0, bbbb, bbbb)$

$\vdash (q_0, bbb, bbb) \vdash (q_0, bb, bb) \vdash (q_0, b, b) \vdash (q_0, \varepsilon, \varepsilon)$

## Lemma

Ke každému zásobníkovému automatu  $\mathcal{M}$  s jedním stavem lze sestavit bezkontextovou gramatiku  $G$  tž.  $L(G) = L(\mathcal{M})$ .

**Důkaz:** Pro ZA  $\mathcal{M} = (\{q_0\}, \Sigma, \Gamma, \delta, q_0, Z_0)$ , kde  $\Sigma \cap \Gamma = \emptyset$  vytvoříme  $G = (\Gamma, \Sigma, Z_0, P)$ , kde  $(A \rightarrow a\alpha) \in P \Leftrightarrow \delta(q_0, a, A) \ni (q_0, \alpha)$ , kde  $u \in \Sigma^*, \alpha \in \Gamma^*$

Indukcí můžeme dokázat

$$Z_0 \Rightarrow_G^* u\alpha \Leftrightarrow (q_0, u, Z_0) \vdash_{\mathcal{M}}^* (q_0, \varepsilon, \alpha)$$

kde  $u \in \Sigma^*, \alpha \in \Gamma^*$

## Lemma

Ke každému zásobníkovému automatu  $\mathcal{M}$  lze sestavit zásobníkový automat  $\mathcal{M}'$  s jedním stavem tž.  $L(\mathcal{M}') = L(\mathcal{M})$ .

### Myšlenka důkazu:

- Stav automatu  $\mathcal{M}$  si budeme pamatovat na zásobníku.
- Pro  $\delta(q, a, X) = \{(q', \varepsilon)\}$  musíme kontrolovat nejen, že jsme ve stavu  $q$ , ale také, že se dostaneme do stavu  $q'$
- Každý zásobníkový symbol automatu  $\mathcal{M}'$  je tedy trojice, kde si pamatujeme zásobníkový symbol, aktuální stav a aktuální stav ze symbolu o jedna níže na zásobníku
- Převod na ZA s jedním stavem tedy způsobí nárůst počtu zásobníkových symbolů

# Omezení bezkontextových jazyků

- Ani bezkontextové gramatiky a zásobníkové automaty nejsou dostatečně silné, aby rozpoznávaly všechny jazyky
- Existuje verze pumping lemmatu pro bezkontextové jazyky, kterou je možno využít k důkazu nebezkontextovosti jazyka
- Např.  $L = \{a^i b^i c^i \mid i \geq 0\}$  nebo  $L = \{ww \mid w \in \{a, b\}^*\}$  nejsou bezkontextové
- Potřebujeme nějaký silnější mechanismus pro rozpoznávání takových jazyků
  - Turingův stroj
  - Lineárně omezený automat



- Rozšíříme deterministické konečné automaty o možnost zápisu na vstupní pásku a pohyb čtecí hlavy oběma směry

- Rozšíříme deterministické konečné automaty o možnost zápisu na vstupní pásku a pohyb čtecí hlavy oběma směry

## Definice

Formálně je **Turingův stroj** definován jako šestice  $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, F)$  kde:

- $Q$  je konečná množina **stavů**
- $\Gamma$  je konečná množina **páskových symbolů**
- $\Sigma \subseteq \Gamma, \Sigma \neq \emptyset$  je konečná množina **vstupních symbolů**
- $\delta : (Q - F) \times \Gamma \rightarrow Q \times \Gamma \times \{-1, 0, +1\}$  je **přechodová funkce**
- $q_0 \in Q$  je **počáteční stav**
- $F \subseteq Q$  je množina **koncových stavů**

# Turingův stroj

- Předpokládáme, že v  $\Gamma - \Sigma$  je vždy speciální prvek  $\square$  označující prázdný znak
- Konfigurace je dána slovem na pásce, stavem a pozicí čtecí hlavy
- Konfigurace je **počáteční**, pokud je hlava na prvním symbolu, stav  $q_0$  a na pásce jsou symboly jen z množiny  $\Sigma$
- Konfigurace je **koncová**, je-li stav z množiny  $F$

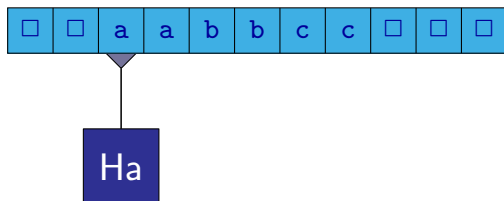
Máme-li stav  $q$  a  $b$  na vstupu, tak pro:

- $\delta(q, b) = (q', b', 0)$  změníme stav na  $q'$ , změníme na aktuální pozici  $b$  na  $b'$
- $\delta(q, b) = (q', b', +1)$  změníme stav na  $q'$ , změníme na aktuální pozici  $b$  na  $b'$  a posuneme aktuální pozici o 1 doprava
- $\delta(q, b) = (q', b', -1)$  změníme stav na  $q'$ , změníme na aktuální pozici  $b$  na  $b'$  a posuneme aktuální pozici o 1 doleva

- Výpočet v koncovém stavu **končí**
- Slovo je přijato, pokud na něm stroj někdy dojde do přijímajícího stavu
- Slovo není přijato, pokud běh stroje nikdy neskončí
- Jazyk  $L(\mathcal{M})$  Turingova stroje  $\mathcal{M}$  je množina všech slov, která stroj  $\mathcal{M}$  přijímá

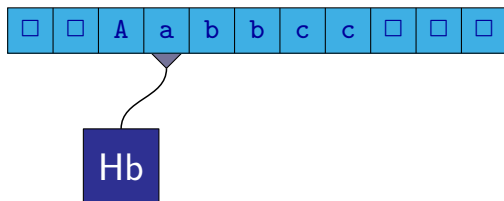
- Výpočet v koncovém stavu **končí**
- Slovo je přijato, pokud na něm stroj někdy dojde do přijímajícího stavu
- Slovo není přijato, pokud běh stroje nikdy neskončí
- Jazyk  $L(\mathcal{M})$  Turingova stroje  $\mathcal{M}$  je množina všech slov, která stroj  $\mathcal{M}$  přijímá
- U Turingova stroje nás někdy zajímá, co zůstane na pásce po skončení běhu.

- Výpočet v koncovém stavu **končí**
- Slovo je přijato, pokud na něm stroj někdy dojde do přijímajícího stavu
- Slovo není přijato, pokud běh stroje nikdy neskončí
- Jazyk  $L(\mathcal{M})$  Turingova stroje  $\mathcal{M}$  je množina všech slov, která stroj  $\mathcal{M}$  přijímá
- U Turingova stroje nás někdy zajímá, co zůstane na pásce po skončení běhu.
- Turingův stroj můžeme reprezentovat grafem obdobně jako KA. Jen popis šipek rozšíříme o informaci, co se zapíše na pásku (pokud se obsah pásky mění) a jak se posune hlava.



Jazyk  $L = \{a^i b^i c^i \mid i \geq 0\}$

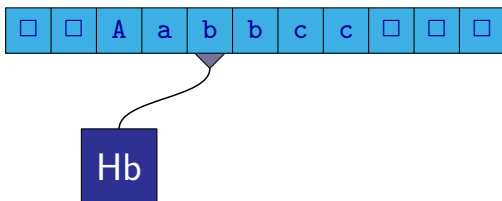
- 1 Čte do prvního **a**, nahradí jej **A**. Pokud najde dříve **b** nebo **c**, zacyklí se.



Jazyk  $L = \{a^i b^i c^i \mid i \geq 0\}$

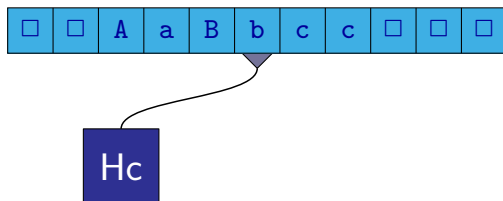
- 1 Čte do prvního **a**, nahradí jej **A**. Pokud najde dříve **b** nebo **c**, zacyklí se.
- 2 Čte do prvního **b**, nahradí jej **B**. Na  $\square$  nebo **c** se zacyklí.





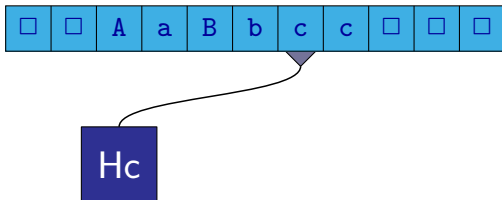
Jazyk  $L = \{a^i b^i c^i \mid i \geq 0\}$

- 1 Čte do prvního **a**, nahradí jej **A**. Pokud najde dříve **b** nebo **c**, zacyklí se.
- 2 Čte do prvního **b**, nahradí jej **B**. Na  $\square$  nebo **c** se zacyklí.



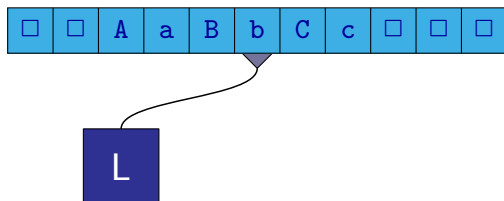
Jazyk  $L = \{a^i b^i c^i \mid i \geq 0\}$

- 1 Čte do prvního **a**, nahradí jej **A**. Pokud najde dříve **b** nebo **c**, zacyklí se.
- 2 Čte do prvního **b**, nahradí jej **B**. Na  $\square$  nebo **c** se zacyklí.
- 3 Čte do prvního **c**, nahradí jej **C**. Na  $\square$  se zacyklí.



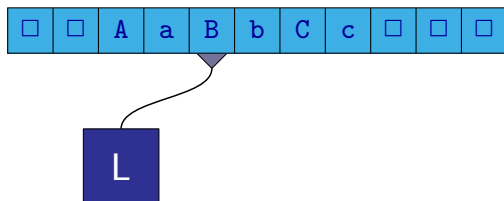
Jazyk  $L = \{a^i b^i c^i \mid i \geq 0\}$

- 1 Čte do prvního **a**, nahradí jej **A**. Pokud najde dříve **b** nebo **c**, zacyklí se.
- 2 Čte do prvního **b**, nahradí jej **B**. Na  $\square$  nebo **c** se zacyklí.
- 3 Čte do prvního **c**, nahradí jej **C**. Na  $\square$  se zacyklí.



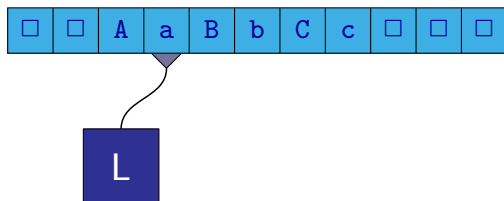
Jazyk  $L = \{a^i b^i c^i \mid i \geq 0\}$

- 1 Čte do prvního **a**, nahradí jej **A**. Pokud najde dříve **b** nebo **c**, zacyklí se.
- 2 Čte do prvního **b**, nahradí jej **B**. Na  $\square$  nebo **c** se zacyklí.
- 3 Čte do prvního **c**, nahradí jej **C**. Na  $\square$  se zacyklí.
- 4 Vrací se doleva na nejbližší **A**.



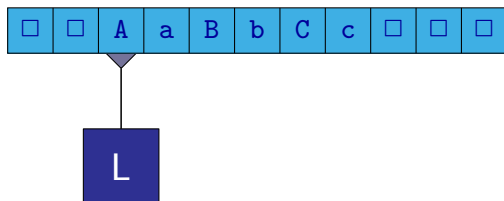
Jazyk  $L = \{a^i b^i c^i \mid i \geq 0\}$

- 1 Čte do prvního **a**, nahradí jej **A**. Pokud najde dříve **b** nebo **c**, zacyklí se.
- 2 Čte do prvního **b**, nahradí jej **B**. Na  $\square$  nebo **c** se zacyklí.
- 3 Čte do prvního **c**, nahradí jej **C**. Na  $\square$  se zacyklí.
- 4 Vrací se doleva na nejbližší **A**.



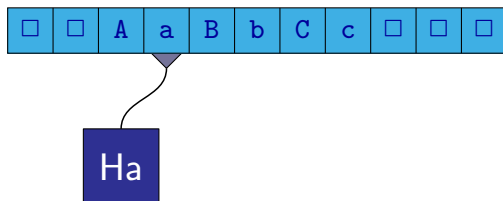
Jazyk  $L = \{a^i b^i c^i \mid i \geq 0\}$

- 1 Čte do prvního **a**, nahradí jej **A**. Pokud najde dříve **b** nebo **c**, zacyklí se.
- 2 Čte do prvního **b**, nahradí jej **B**. Na  $\square$  nebo **c** se zacyklí.
- 3 Čte do prvního **c**, nahradí jej **C**. Na  $\square$  se zacyklí.
- 4 Vrací se doleva na nejbližší **A**.



Jazyk  $L = \{a^i b^i c^i \mid i \geq 0\}$

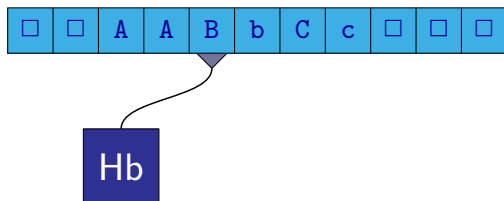
- 1 Čte do prvního **a**, nahradí jej **A**. Pokud najde dříve **b** nebo **c**, zacyklí se.
- 2 Čte do prvního **b**, nahradí jej **B**. Na  $\square$  nebo **c** se zacyklí.
- 3 Čte do prvního **c**, nahradí jej **C**. Na  $\square$  se zacyklí.
- 4 Vrací se doleva na nejbližší **A**.



Jazyk  $L = \{a^i b^i c^i \mid i \geq 0\}$

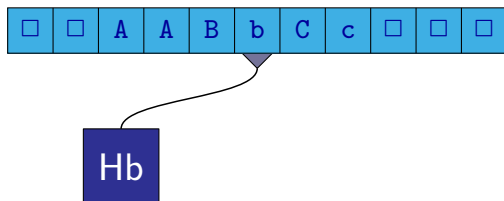
- 1 Čte do prvního **a**, nahradí jej **A**. Pokud najde dříve **b** nebo **c**, zacyklí se.
- 2 Čte do prvního **b**, nahradí jej **B**. Na  $\square$  nebo **c** se zacyklí.
- 3 Čte do prvního **c**, nahradí jej **C**. Na  $\square$  se zacyklí.
- 4 Vrací se doleva na nejbližší **A**.
- 5 Pokračuje bodem 1





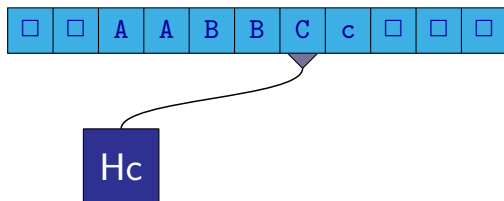
Jazyk  $L = \{a^i b^i c^i \mid i \geq 0\}$

- 1 Čte do prvního **a**, nahradí jej **A**. Pokud najde dříve **b** nebo **c**, zacyklí se.
- 2 Čte do prvního **b**, nahradí jej **B**. Na  $\square$  nebo **c** se zacyklí.
- 3 Čte do prvního **c**, nahradí jej **C**. Na  $\square$  se zacyklí.
- 4 Vrací se doleva na nejbližší **A**.
- 5 Pokračuje bodem 1



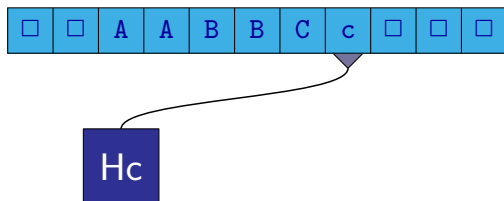
Jazyk  $L = \{a^i b^i c^i \mid i \geq 0\}$

- 1 Čte do prvního **a**, nahradí jej **A**. Pokud najde dříve **b** nebo **c**, zacyklí se.
- 2 Čte do prvního **b**, nahradí jej **B**. Na  $\square$  nebo **c** se zacyklí.
- 3 Čte do prvního **c**, nahradí jej **C**. Na  $\square$  se zacyklí.
- 4 Vrací se doleva na nejbližší **A**.
- 5 Pokračuje bodem 1



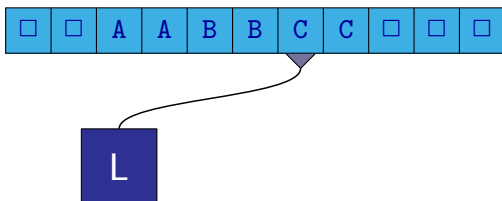
Jazyk  $L = \{a^i b^i c^i \mid i \geq 0\}$

- 1 Čte do prvního **a**, nahradí jej **A**. Pokud najde dříve **b** nebo **c**, zacyklí se.
- 2 Čte do prvního **b**, nahradí jej **B**. Na  $\square$  nebo **c** se zacyklí.
- 3 Čte do prvního **c**, nahradí jej **C**. Na  $\square$  se zacyklí.
- 4 Vrací se doleva na nejbližší **A**.
- 5 Pokračuje bodem 1



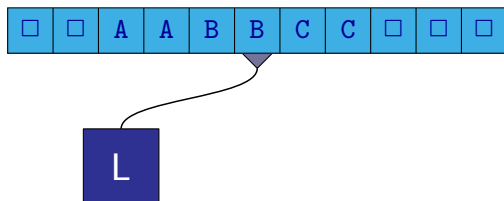
Jazyk  $L = \{a^i b^i c^i \mid i \geq 0\}$

- 1 Čte do prvního **a**, nahradí jej **A**. Pokud najde dříve **b** nebo **c**, zacyklí se.
- 2 Čte do prvního **b**, nahradí jej **B**. Na  $\square$  nebo **c** se zacyklí.
- 3 Čte do prvního **c**, nahradí jej **C**. Na  $\square$  se zacyklí.
- 4 Vrací se doleva na nejbližší **A**.
- 5 Pokračuje bodem 1



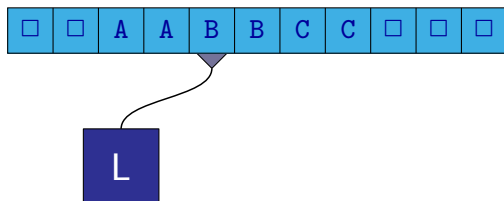
Jazyk  $L = \{a^i b^i c^i \mid i \geq 0\}$

- 1 Čte do prvního **a**, nahradí jej **A**. Pokud najde dříve **b** nebo **c**, zacyklí se.
- 2 Čte do prvního **b**, nahradí jej **B**. Na  $\square$  nebo **c** se zacyklí.
- 3 Čte do prvního **c**, nahradí jej **C**. Na  $\square$  se zacyklí.
- 4 Vrací se doleva na nejbližší **A**.
- 5 Pokračuje bodem 1



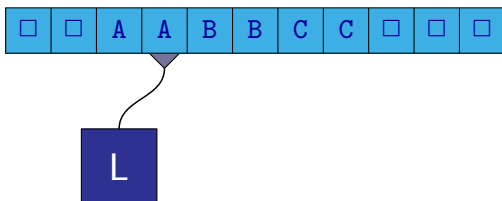
Jazyk  $L = \{a^i b^i c^i \mid i \geq 0\}$

- 1 Čte do prvního **a**, nahradí jej **A**. Pokud najde dříve **b** nebo **c**, zacyklí se.
- 2 Čte do prvního **b**, nahradí jej **B**. Na  $\square$  nebo **c** se zacyklí.
- 3 Čte do prvního **c**, nahradí jej **C**. Na  $\square$  se zacyklí.
- 4 Vrací se doleva na nejbližší **A**.
- 5 Pokračuje bodem 1



Jazyk  $L = \{a^i b^i c^i \mid i \geq 0\}$

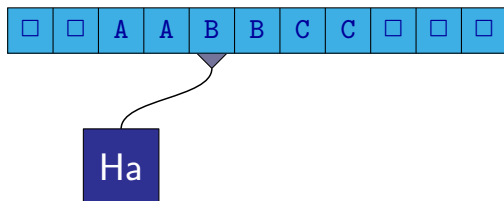
- 1 Čte do prvního **a**, nahradí jej **A**. Pokud najde dříve **b** nebo **c**, zacyklí se.
- 2 Čte do prvního **b**, nahradí jej **B**. Na  $\square$  nebo **c** se zacyklí.
- 3 Čte do prvního **c**, nahradí jej **C**. Na  $\square$  se zacyklí.
- 4 Vrací se doleva na nejbližší **A**.
- 5 Pokračuje bodem 1



Jazyk  $L = \{a^i b^i c^i \mid i \geq 0\}$

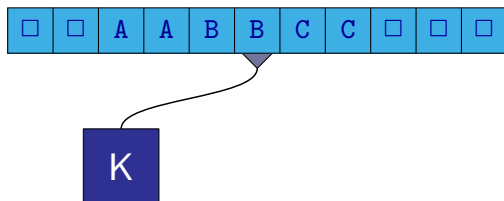
- 1 Čte do prvního **a**, nahradí jej **A**. Pokud najde dříve **b** nebo **c**, zacyklí se.
- 2 Čte do prvního **b**, nahradí jej **B**. Na  $\square$  nebo **c** se zacyklí.
- 3 Čte do prvního **c**, nahradí jej **C**. Na  $\square$  se zacyklí.
- 4 Vrací se doleva na nejbližší **A**.
- 5 Pokračuje bodem 1





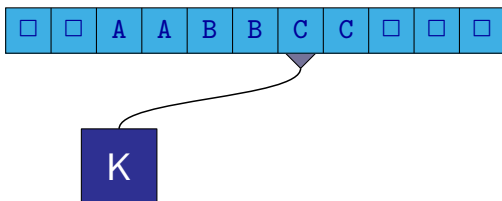
Jazyk  $L = \{a^i b^i c^i \mid i \geq 0\}$

- 1 Čte do prvního **a**, nahradí jej **A**. Pokud najde dříve **b** nebo **c**, zacyklí se. Pokud už **a** není, zkontroluje, zda už jsou jen velká písmena.
- 2 Čte do prvního **b**, nahradí jej **B**. Na  $\square$  nebo **c** se zacyklí.
- 3 Čte do prvního **c**, nahradí jej **C**. Na  $\square$  se zacyklí.
- 4 Vrací se doleva na nejbližší **A**.
- 5 Pokračuje bodem 1



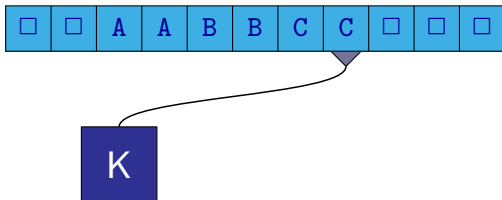
Jazyk  $L = \{a^i b^i c^i \mid i \geq 0\}$

- 1 Čte do prvního **a**, nahradí jej **A**. Pokud najde dříve **b** nebo **c**, zacyklí se. Pokud už **a** není, zkontroluje, zda už jsou jen velká písmena.
- 2 Čte do prvního **b**, nahradí jej **B**. Na  $\square$  nebo **c** se zacyklí.
- 3 Čte do prvního **c**, nahradí jej **C**. Na  $\square$  se zacyklí.
- 4 Vrací se doleva na nejbližší **A**.
- 5 Pokračuje bodem 1



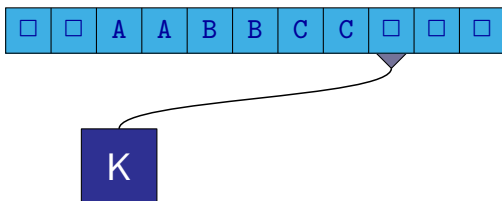
Jazyk  $L = \{a^i b^i c^i \mid i \geq 0\}$

- 1 Čte do prvního **a**, nahradí jej **A**. Pokud najde dříve **b** nebo **c**, zacyklí se. Pokud už **a** není, zkontroluje, zda už jsou jen velká písmena.
- 2 Čte do prvního **b**, nahradí jej **B**. Na  $\square$  nebo **c** se zacyklí.
- 3 Čte do prvního **c**, nahradí jej **C**. Na  $\square$  se zacyklí.
- 4 Vrací se doleva na nejbližší **A**.
- 5 Pokračuje bodem 1



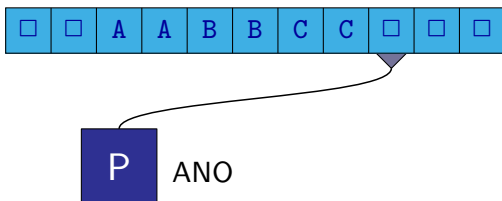
Jazyk  $L = \{a^i b^i c^i \mid i \geq 0\}$

- 1 Čte do prvního **a**, nahradí jej **A**. Pokud najde dříve **b** nebo **c**, zacyklí se. Pokud už **a** není, zkontroluje, zda už jsou jen velká písmena.
- 2 Čte do prvního **b**, nahradí jej **B**. Na  $\square$  nebo **c** se zacyklí.
- 3 Čte do prvního **c**, nahradí jej **C**. Na  $\square$  se zacyklí.
- 4 Vrací se doleva na nejbližší **A**.
- 5 Pokračuje bodem 1



Jazyk  $L = \{a^i b^i c^i \mid i \geq 0\}$

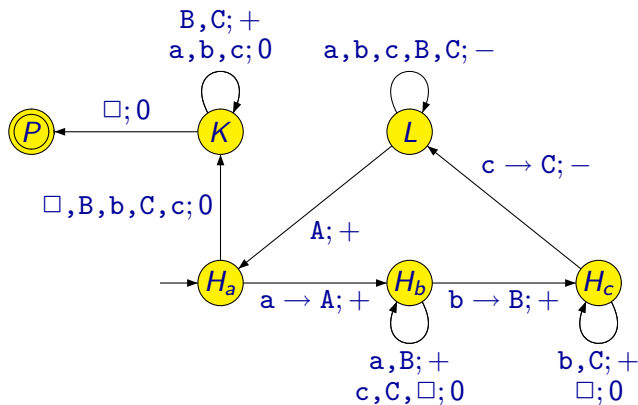
- 1 Čte do prvního **a**, nahradí jej **A**. Pokud najde dříve **b** nebo **c**, zacyklí se. Pokud už **a** není, zkontroluje, zda už jsou jen velká písmena.
- 2 Čte do prvního **b**, nahradí jej **B**. Na  $\square$  nebo **c** se zacyklí.
- 3 Čte do prvního **c**, nahradí jej **C**. Na  $\square$  se zacyklí.
- 4 Vrací se doleva na nejbližší **A**.
- 5 Pokračuje bodem 1



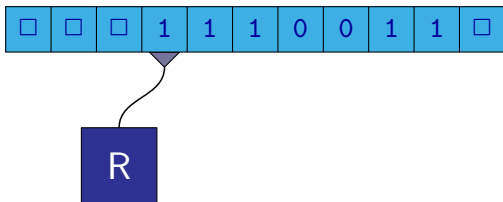
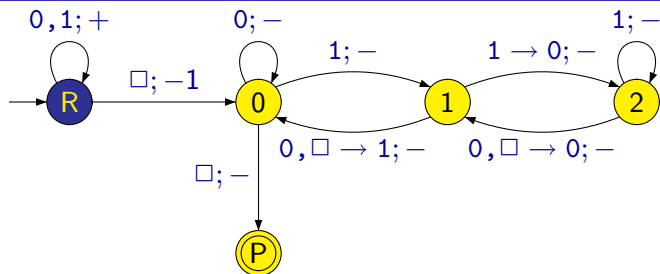
Jazyk  $L = \{a^i b^i c^i \mid i \geq 0\}$

- 1 Čte do prvního **a**, nahradí jej **A**. Pokud najde dříve **b** nebo **c**, zacyklí se. Pokud už **a** není, zkontroluje, zda už jsou jen velká písmena.
- 2 Čte do prvního **b**, nahradí jej **B**. Na  $\square$  nebo **c** se zacyklí.
- 3 Čte do prvního **c**, nahradí jej **C**. Na  $\square$  se zacyklí.
- 4 Vrací se doleva na nejbližší **A**.
- 5 Pokračuje bodem 1

# Turingův stroj

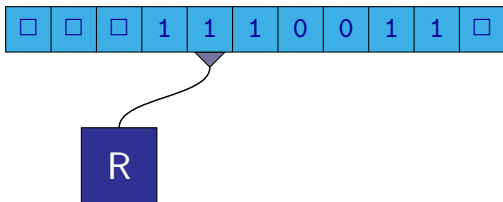
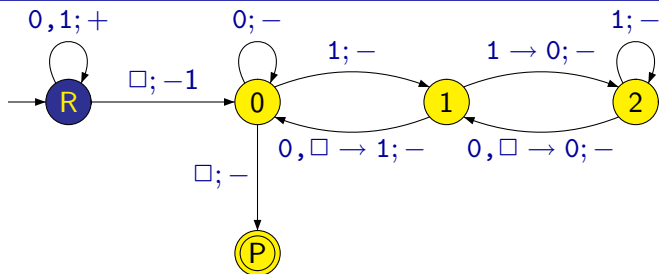


# Turingův stroj - násobení třemi

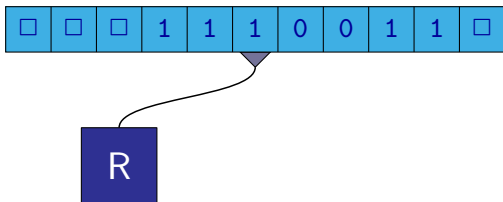
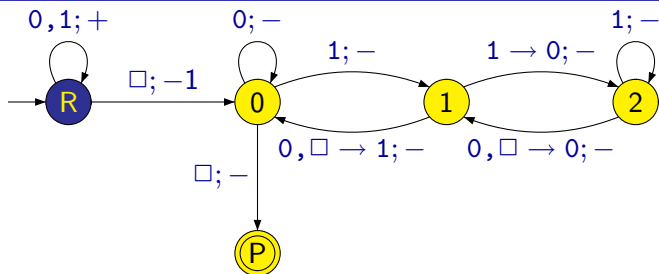




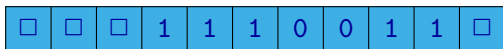
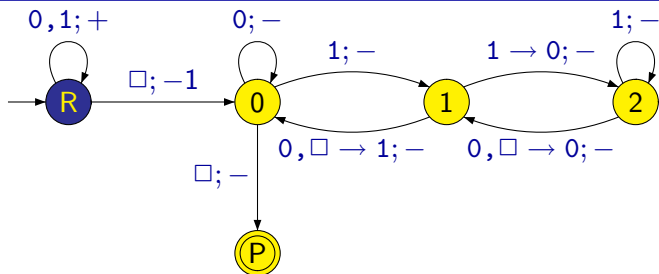
# Turingův stroj - násobení třemi



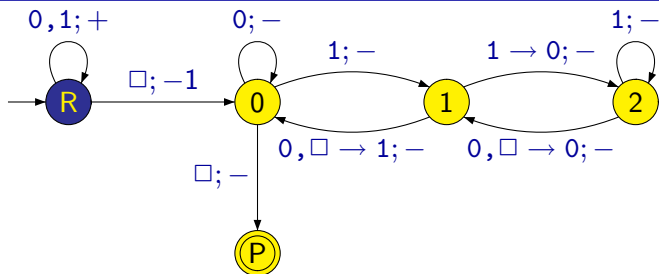
# Turingův stroj - násobení třemi



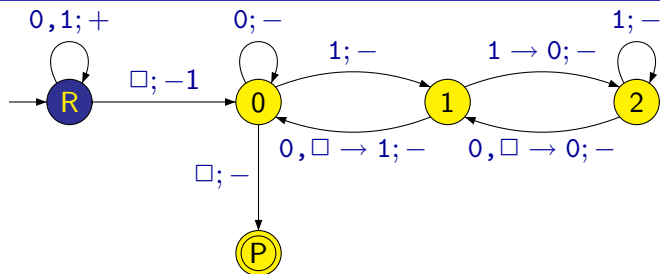
# Turingův stroj - násobení třemi



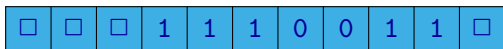
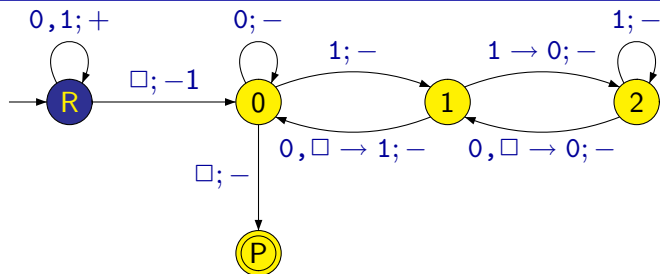
# Turingův stroj - násobení třemi



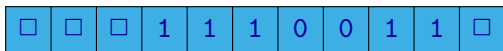
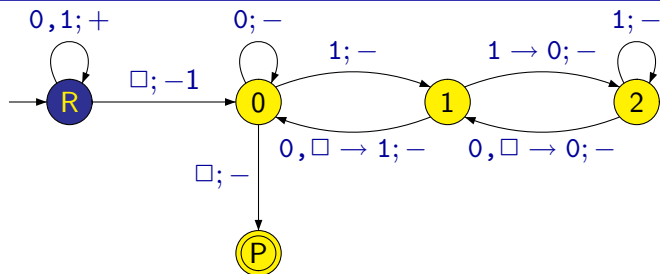
# Turingův stroj - násobení třemi



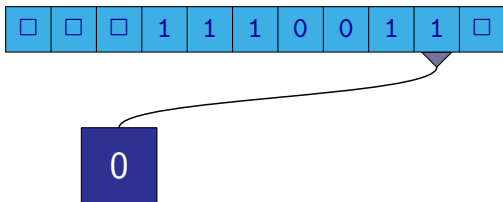
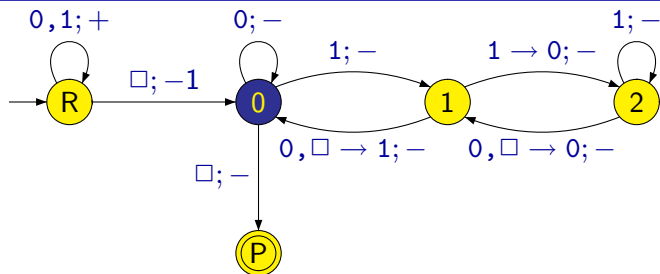
# Turingův stroj - násobení třemi



# Turingův stroj - násobení třemi

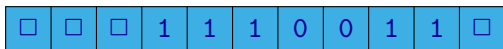
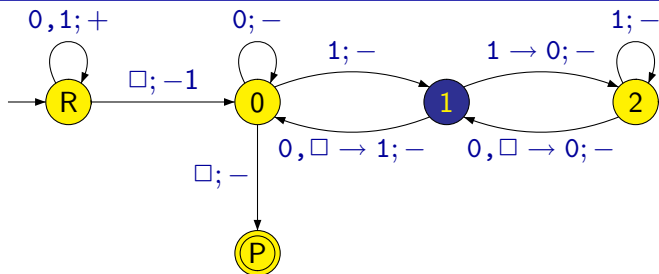


# Turingův stroj - násobení třemi

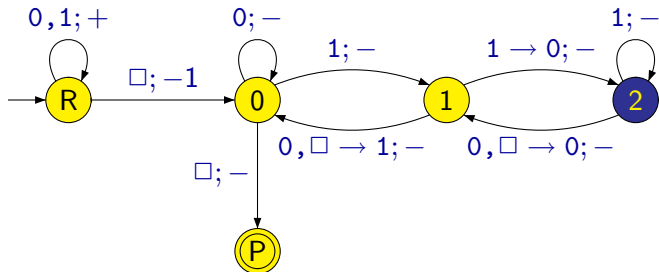




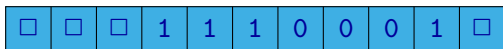
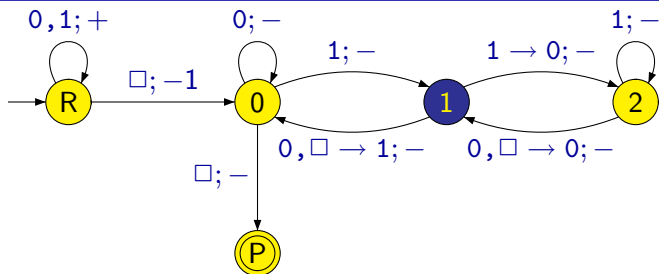
# Turingův stroj - násobení třemi



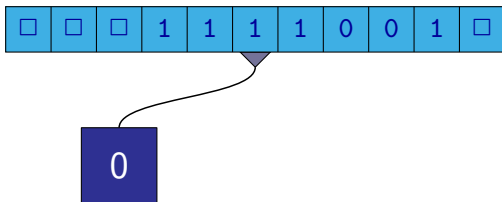
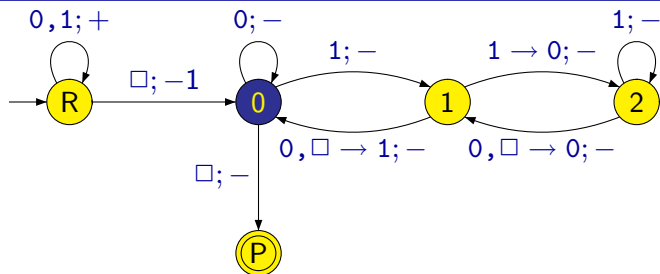
# Turingův stroj - násobení třemi



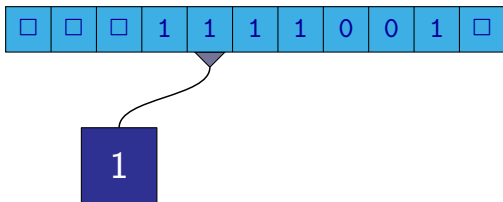
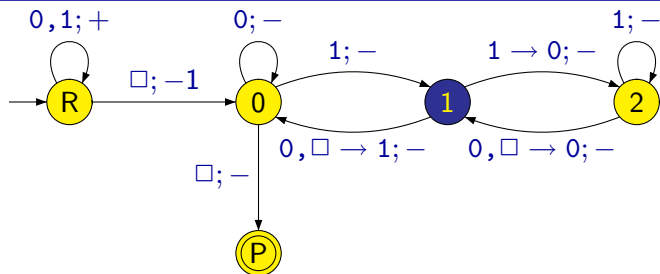
# Turingův stroj - násobení třemi



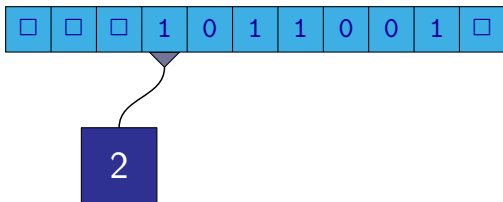
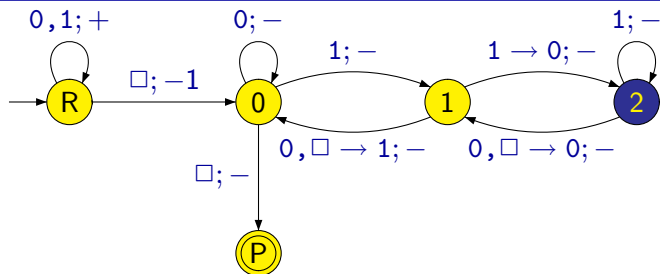
# Turingův stroj - násobení třemi



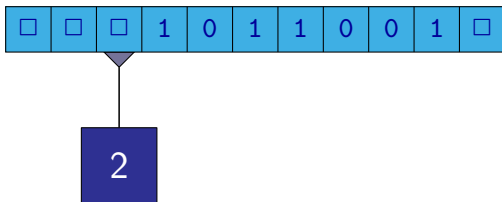
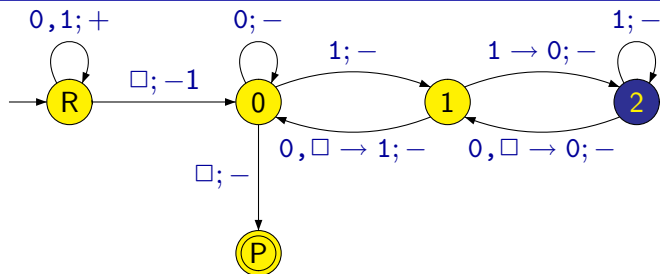
# Turingův stroj - násobení třemi



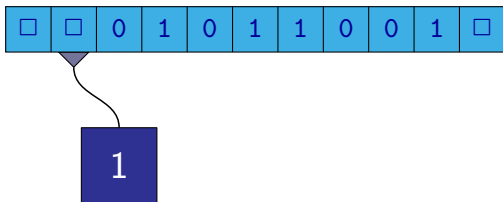
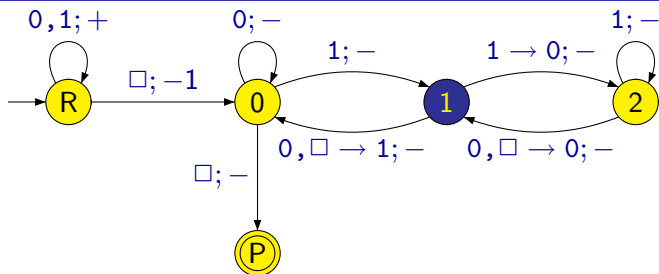
# Turingův stroj - násobení třemi



# Turingův stroj - násobení třemi

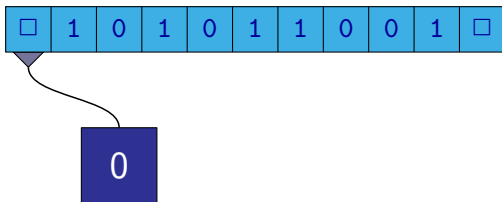
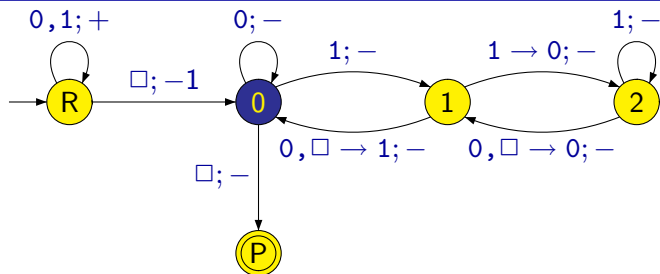


# Turingův stroj - násobení třemi

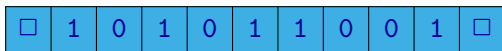
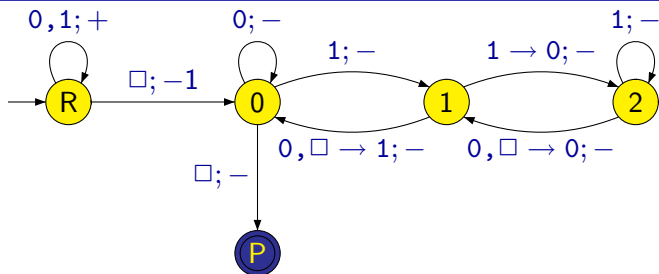




# Turingův stroj - násobení třemi



# Turingův stroj - násobení třemi



P ANO

- Turingův stroj, který využívá jen úsek pásky, kde je zapsáno vstupní slovo
- Představa levé a pravé zarážky kolem slova, které nemohou být přepsány
- Z levé zarážky je možný pohyb jen vpravo, z pravé zarážky jen vlevo
- LBA (linear bounded automata) se uvažují v nedeterministické verzi
- Je otevřená otázka, jestli jsou deterministické LBA stejně „silné“ jako nedeterministické

## Definice

**Generativní gramatika** je dána čtveřicí parametrů  $G = (\Pi, \Sigma, S, P)$ , kde

- $\Pi$  je konečná množina neterminálů
  - $\Sigma$  je konečná množina terminálů,  $\Pi \cap \Sigma = \emptyset$
  - $S \in \Pi$  je počáteční neterminál
  - $P$  je konečná množina pravidel typu  $\alpha \rightarrow \beta$ , kde  $\alpha \in (\Pi \cup \Sigma)^* \Pi (\Pi \cup \Sigma)^*$  a  $\beta \in (\Pi \cup \Sigma)^*$ .
- 
- $\mu_1 \alpha \mu_2 \Rightarrow_G \mu_1 \beta \mu_2$  pokud  $\alpha \rightarrow \beta$  je pravidlo v  $P$
  - $L(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$

Podle tvaru pravidel, která v gramatice povolíme, dělíme gramatiky na gramatiky typu:

- 0 - Obecné **generativní gramatiky** - pravidla bez omezení
- 1 - **Kontextové gramatiky** - pravidla tvaru  $\alpha X \beta \rightarrow \alpha \gamma \beta$ , kde  $|\gamma| \geq 1$   
(Výjimka  $S \rightarrow \varepsilon$ , ale  $S$  pak není na pravé straně žádného pravidla)
- 2 - **Bezkontextové gramatiky** - pravidla tvaru  $X \rightarrow \gamma$
- 3 - **Regulární gramatiky** - pravidla tvaru  $X \rightarrow wY$  nebo  $X \rightarrow w$

kde  $\alpha, \beta, \gamma \in (\Pi \cup \Sigma)^*$ ,  $X \in \Pi$ ,  $w \in \Sigma^*$

Podle tvaru pravidel, která v gramatice povolíme, dělíme gramatiky na gramatiky typu:

- 0 - Obecné **generativní gramatiky** - pravidla bez omezení
- 1 - **Kontextové gramatiky** - pravidla tvaru  $\alpha X \beta \rightarrow \alpha \gamma \beta$ , kde  $|\gamma| \geq 1$   
(Výjimka  $S \rightarrow \varepsilon$ , ale  $S$  pak není na pravé straně žádného pravidla)
- 2 - **Bezkontextové gramatiky** - pravidla tvaru  $X \rightarrow \gamma$
- 3 - **Regulární gramatiky** - pravidla tvaru  $X \rightarrow wY$  nebo  $X \rightarrow w$

kde  $\alpha, \beta, \gamma \in (\Pi \cup \Sigma)^*$ ,  $X \in \Pi$ ,  $w \in \Sigma^*$

- Jazyk je typu  $i$ , jestliže jej generuje nějaká gramatika typu  $i$
- Označíme-li  $\mathcal{L}_i$  třídu jazyků typu  $i$ , pak  $\mathcal{L}_3 \subset \mathcal{L}_2 \subset \mathcal{L}_1 \subset \mathcal{L}_0$

Pojmenování jazyků jednotlivých typů:

- 0 - Rekurzivně spočetné
- 1 - Kontextové
- 2 - Bezkontextové
- 3 - Regulární

Pojmenování jazyků jednotlivých typů:

- 0 - Rekurzivně spočetné
- 1 - Kontextové
- 2 - Bezkontextové
- 3 - Regulární

Každé třídě jazyků odpovídá typ stroje nebo automatu, který dokáže rozpoznat všechny jazyky z dané třídy

- 0 - Turingův stroj (deterministický nebo nedeterministický)
- 1 - Lineárně omezený automat (nedeterministický, pro deterministické to není známo)
- 2 - Zásobníkový automat (jen nedeterministický)
- 3 - Konečný automat (deterministický nebo nedeterministický)



# Co je to algoritmus?

## Algoritmus

**Algoritmus** je mechanický postup skládající se z nějakých jednoduchých elementárních kroků, který pro nějaký zadaný **vstup** vyprodukuje nějaký **výstup**.

Algoritmus může být zadán:

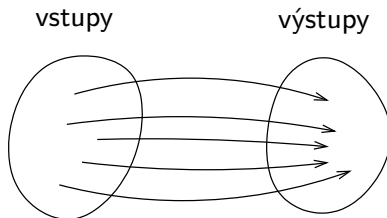
- slovním popisem v přirozeném jazyce
- pseudokódem
- jako počítačový program v nějakém programovacím jazyce
- jako hardwarový obvod
- ...

Algoritmy slouží k řešení různých **problémů**.

## Problém

V zadání **problému** musí být určeno:

- co je množinou možných vstupů
- co je množinou možných výstupů
- jaký je vztah mezi vstupy a výstupy



## Problém „Třídění“

**Vstup:** Sekvence prvků  $a_1, a_2, \dots, a_n$ .

**Výstup:** Prvky sekvence  $a_1, a_2, \dots, a_n$  seřazené od nejmenšího po největší.

### Příklad:

- Vstup: 8, 13, 3, 10, 1, 4
- Výstup: 1, 3, 4, 8, 10, 13

**Poznámka:** Konkrétní vstup nějakého problému se nazývá **instance** problému.

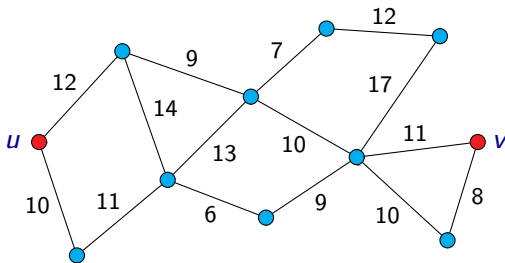
# Příklady problému

## Problém „Hledání nejkratší cesty v (neorientovaném) grafu“

**Vstup:** Neorientovaný graf  $G = (V, E)$  s ohodnocením hran, a dvojice vrcholů  $u, v \in V$ .

**Výstup:** Nejkratší cesta z vrcholu  $u$  do vrcholu  $v$ .

### Příklad:



## Problém „Prvočíselnost“

**Vstup:** Přirozené číslo  $n$ .

**Výstup:** ANO pokud je  $n$  prvočíslo, NE v opačném případě.

**Poznámka:** Přirozené číslo  $n$  je **prvočíslo**, pokud je větší než 1 a je dělitelné beze zbytku pouze čísly 1 a  $n$ .

Prvních několik prvočísel: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, ...

Situace, kdy množina výstupů je  $\{ANO, NE\}$  je poměrně častá. Takovým problémům se říká **rozhodovací problémy**.

Rozhodovací problémy většinou specifikujeme tak, že místo popisu toho, co je výstupem, uvedeme otázku.

## Příklad:

### Problém „Prvočíselnost“

**Vstup:** Přirozené číslo  $n$ .

**Otázka:** Je  $n$  prvočíslo?

# Optimalizační problémy

Dalším speciálním případem jsou tzv. optimalizační problémy.

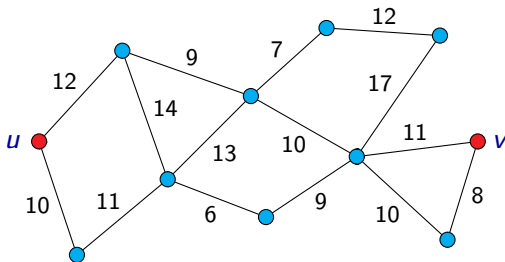
**Optimalizační problém** je problém, kde je úkolem vybrat z nějaké množiny přípustných řešení takové řešení, které je v nějakém ohledu optimální.

# Optimalizační problémy

Dalším speciálním případem jsou tzv. optimalizační problémy.

**Optimalizační problém** je problém, kde je úkolem vybrat z nějaké množiny přípustných řešení takové řešení, které je v nějakém ohledu optimální.

**Příklad:** V problému „Hledání nejkratší cesty v grafu“ je množina všech přípustných řešení tvořena všemi cestami z vrcholu  $u$  do vrcholu  $v$ . Kritériem, podle kterého cesty hodnotíme, je délka cesty.





## Řešení problému

Algoritmus **korektně řeší** daný problém, když:

- 1 Se pro libovolný vstup daného problému (libovolnou vstupní instanci) po konečném počtu kroků zastaví.
- 2 Vyprodukuje výstup z množiny možných výstupů, který vyhovuje podmínkám uvedeným v zadání problému.

Pro jeden problém může existovat celá řada algoritmů, které jej korektně řeší.

**Poznámka:** Množina vstupních instancí bývá typicky nekonečná.

# Kódování vstupu a výstupu

Většinou předpokládáme, že vstupy i výstupy jsou kódovány jako slova v nějaké abecedě  $\Sigma$ .

**Příklad:** Například u problému „Třídění“ bychom mohli zvolit jako abecedu  $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, , \}$ .

Vstupem by pak mohlo být například slovo

826,13,3901,101,128,562

a výstupem slovo

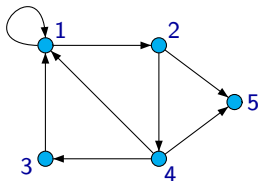
13,101,128,562,826,3901

**Poznámka:** Ne každé slovo ze  $\Sigma^*$  musí reprezentovat nějaký vstup. Kódování bychom ale měli zvolit tak, abychom byli schopni snadno poznat ta slova, která nějaký vstup reprezentují.

# Kódování vstupu a výstupu

**Příklad:** Pokud je vstupem nějakého problému například graf, můžeme ho reprezentovat jako seznam vrcholů a hran:

Například následující graf



můžeme reprezentovat jako slovo

$(1, 2, 3, 4, 5), ((1, 2), (2, 4), (4, 3), (3, 1), (1, 1), (2, 5), (4, 5), (4, 1))$

v abecedě  $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ,, (, )\}$ .

# Kódování vstupu a výstupu

Můžeme se omezit na případ, kdy jsou vstupy i výstupy kódovány jako slova v abecedě  $\{0, 1\}$  (tj. jako sekvence bitů).

Symbole jakékoli jiné abecedy lze reprezentovat jako sekvence bitů.

**Příklad:** Abeceda  $\{a, b, c, d, e, f, g\}$

a ↔ 001

b ↔ 010

c ↔ 011

d ↔ 100

e ↔ 101

f ↔ 110

g ↔ 111

Slovo 'defb' můžeme reprezentovat jako '100101110010'.

Můžeme tedy říct, že každý algoritmus realizuje výpočet hodnot nějaké funkce

$$f : \Sigma^* \rightarrow \Sigma^*$$

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

Alternativně se také na algoritmy můžeme dívat tak, že realizují výpočet nějaké funkce

$$f : \mathbb{N} \rightarrow \mathbb{N}$$

kde  $\mathbb{N} = \{0, 1, 2, 3, \dots\}$  je množina přirozených čísel, neboť na každou sekvenci bitů se můžeme dívat jako na zápis čísla v binární soustavě.

## Funkce

$$f : \Sigma^* \rightarrow \Sigma^* \quad \text{resp.} \quad f : \mathbb{N} \rightarrow \mathbb{N}$$

realizovaná nějakým algoritmem nemusí být nutně totální, může být částečná. Hodnota  $f(x)$  není definovaná například v případě, že se daný algoritmus pro vstup  $x$  nikdy nezastaví.

**Poznámka:** Funkce  $f : A \rightarrow B$  je **totální**, jestliže hodnota  $f(x)$  je definovaná pro libovolné  $x \in A$ , a **částečná**, jestliže pro některá  $x \in A$  nemusí být hodnota  $f(x)$  definovaná.

Stejně jako na algoritmy i na problémy se můžeme dívat jako na funkce typu:

$$f : \Sigma^* \rightarrow \Sigma^* \quad \text{resp.} \quad f : \mathbb{N} \rightarrow \mathbb{N}$$

Jestliže algoritmus realizuje nějakou funkci  $f'$ , řekneme, že tento algoritmus **řeší** problém popsáný funkcí  $f$ , jestliže se pro libovolný vstup  $x$  zastaví s tím, že

$$f(x) = f'(x)$$



Na rozhodovací problémy můžeme pohlížet jako na jazyky.

Jazyk odpovídající danému rozhodovacímu problému je množina těch slov ze  $\Sigma^*$ , která reprezentují ty vstupy, pro něž je odpověď **ANO**.

**Příklad:** Jazyk  $L$  tvořený těmi slovy ze  $\{0, 1\}^*$ , která jsou binárním zápisem nějakého prvočísla.

Například  $101 \in L$ , ale  $110 \notin L$ .

Často se při zkoumání algoritmů a problému omezujeme jen na rozhodovací problémy.

Není to však na úkor obecnosti, neboť libovolný obecný problém možné vhodným způsobem přeformulovat jako rozhodovací problém, s tím, že když najdeme algoritmus, který by řešil tento rozhodovací problém, tak bychom snadno sestrojili algoritmus, který by řešil původní problém, a naopak.

Pokud například máme problém  $P$ , kde:

- vstupy jsou prvky z nějaké množiny  $X$
- výstupy jsou slova z  $\{0, 1\}^*$

můžeme tento problém přeformulovat jako následující rozhodovací problém:

## Problém

**Vstup:** Prvek  $x \in X$  a číslo  $k$ .

**Otázka:** Když  $z$  je výstup, který odpovídá vstupu  $x$  problému  $P$ , má  $k$ -tý bit slova  $z$  hodnotu  $1$ ?

Předpokládejme, že máme dán nějaký problém  $P$ .

Jestliže existuje nějaký algoritmus, který řeší problém  $P$ , pak říkáme, že problém  $P$  je **algoritmicky řešitelný**.

Jestliže  $P$  je rozhodovací problém a jestliže existuje nějaký algoritmus, který problém  $P$  řeší, pak říkáme, že problém  $P$  je **rozhodnutelný**.

Když chceme ukázat, že problém  $P$  je algoritmicky řešitelný, stačí ukázat nějaký algoritmus, který ho řeší (a případně ukázat, že daný algoritmus problém  $P$  skutečně řeší).

# Algoritmicky řešitelné problémy

U mnohých problémů je na první pohled zřejmé, že jsou algoritmicky řešitelné, jako třeba:

- Třídění
- Hledání nejkratší cesty v grafu
- Prvočíselnost

kde stačí probrat všechny možnosti (kterých je ve všech těchto případech konečně mnoho), i když takový triviální algoritmus založený na řešení **hrubou silou** nemusí být zrovna efektivní.

Na druhou stranu existuje celá řada problémů, u kterých to tak jasné není.

- Najít algoritmus a dokázat, že řeší daný problém, může být velmi netriviální úkol.
- Algoritmus, který by řešil daný problém, nemusí vůbec existovat.

# Nutnost upřesnění pojmu „algorithmus“

Dosavadní definice pojmu algorithmus byla poněkud vágní.

Pokud bychom pro nějaký problém chtěli ukázat, že neexistuje algorithmus, který by daný problém řešil, tak by to s takovouto neurčitou definicí pojmu algorithmus asi nešlo.

Intuitivně chápeme, co by měl mít algorithmus za vlastnosti:

- Měl by se skládat z jednoduchých kroků, které je možno vykonávat „mechanicky“, bez porozumění problému.
- Objekty, se kterými algorithmus pracuje, i prováděné operace by měly být konečné.

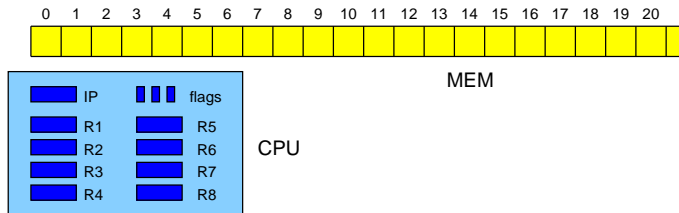
# Algoritmus realizovaný počítačem

Například program v libovolném programovacím jazyce dané vlastnosti zcela jistě má.

Ať už je program napsán v jakémkoliv programovacím jazyce, jsou jeho instrukce nakonec prováděny na hardwaru nějakého konkrétního počítače na úrovni instrukcí procesoru.

Je tedy jasné, že **každý** program v **každém** programovacím jazyce bychom mohli zapsat jako program tvořený pouze instrukcemi strojového kódu nějakého procesoru.

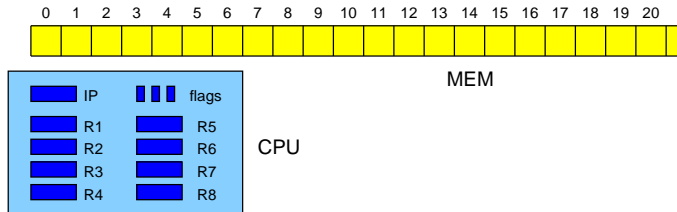
Typická architektura naprosté většiny počítačů vypadá následovně:



- Počítač má **paměť** skládající se z velkého množství paměťových buněk.
- Každá buňka může obsahovat číslo určité velikosti, typicky 1 byte (8 bitů), tj. číslo v rozsahu **0..255**.
- Buňky jsou očíslovány. Číslo buňky se nazývá její **adresa**.

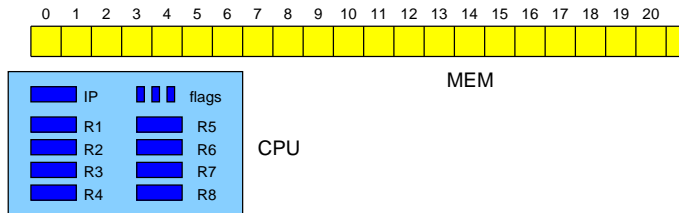


Typická architektura naprosté většiny počítačů vypadá následovně:



- Instrukce jsou uloženy v paměti (každá instrukce má svůj číselný **kód**) a jsou sekvenčně vykonávány **procesorem**.
- Procesor udržuje tzv. **čítač instrukcí IP**, který obsahuje adresu aktuálně prováděné instrukce.
- Procesor načte instrukci z adresy určené **IP**, zvětší **IP** o délku načtené instrukce a provede danou instrukci.

Typická architektura naprosté většiny počítačů vypadá následovně:



- Procesor obsahuje několik **registru** pevné délky (např. 32 nebo 64 bitů).
- Většina operací je prováděna na registrech.
- Procesor obsahuje **příznaky (flags)**, které umožňují testovat výsledek poslední operace (např. přetečení, jestli je výsledek nula apod.)

## Typické instrukce:

- Načtení obsahu paměťové buňky (resp. několika po sobě jdoucích buněk) do některého registru (**LOAD**).
- Uložení obsahu registru do některé paměťové buňky (resp. několika po sobě jdoucích buněk) (**STORE**).

**Poznámka:** Adresa buňky je buď přímá (tj. je přímo součástí instrukce) nebo nepřímá (uložená v některém registru, případně spočítaná z obsahu jednoho nebo několika registrů).

- Načtení obsahu jednoho registru do jiného registru (**MOV**).
- Aritmetické instrukce (**ADD, SUB, MUL, DIV, NEG, CMP, INC, DEC, ...**).
- Logické instrukce (**AND, OR, XOR, NOT, ...**).
- Bitové posuny a rotace (**SHL, SHR, ...**)

Typické instrukce (pokračování):

- Nepodmíněný skok (**JMP**).

**Poznámka:** Adresa skoku může být přímá nebo nepřímá.

- Podmíněné skoky (**JZERO**, **JGTZ**, ...).
- Volání podprogramů (**CALL**, **RET**).
- Různé speciální instrukce – práce se vstupem a výstupem, obsluha přerušení, mody činnosti procesoru, řízení přístupu do paměti (stránkování) apod.

**Příklad:** Instrukce zapsaná ve vyšším programovacím jazyce jako

$$x = y + 2$$

může být realizována následující sekvencí instrukcí určitého (hypotetického) procesoru:

```
LOAD    0x001b7c42,R5
ADD     $2,R5
STORE   R5,0x001b7c38
```

**Poznámka:** Předpokládáme, že proměnná  $x$  je uložena na adrese `0x001b7c38` a proměnná  $y$  na adrese `0x001b7c42`.

**Stroj RAM (Random Access Machine)** je idealizovaný model počítače.

Rozdíly oproti skutečnému počítači:

- Velikost paměti není omezena (adresa může být libovolné přirozené číslo).
- Velikost obsahu jednotlivých buněk není omezena (buňka může obsahovat libovolné celé číslo).
- Čte data sekvenčně ze vstupu, který je tvořen sekvencí celých čísel. Ze vstupu lze pouze číst.
- Zapisuje data sekvenčně na výstup, který je tvořen sekvencí celých čísel. Na výstup je možné pouze zapisovat.

Každý program v každém jazyce by mohl být realizován jako program stroje RAM.

Není složité (i když je to trochu pracné) si rozmyslet, že libovolný algoritmus prováděný strojem RAM je možné realizovat také Turingovým strojem.

Turingův stroj je schopen realizovat libovolný algoritmus, který by bylo možné zapsat jako program v nějakém programovacím jazyce.

**Poznámka:** Turingův stroj pracuje se slovy nad nějakou abecedou, zatímco stroj RAM s čísly. Čísla ale můžeme zapisovat jako sekvence symbolů a naopak symboly nějaké abecedy můžeme zapisovat jako čísla.

## Churchova-Turingova teze

Každý algoritmus je možné realizovat nějakým Turingovým strojem.

Není to věta, kterou by bylo možno dokázat v matematickém smyslu – není formálně definováno, co je to algoritmus.

Tezi formulovali nezávisle na sobě v polovině 30. let 20. století Alan Turing a Alonzo Church.



Ve stejné době bylo navrženo několik různých formalismů zachycujících pojem algoritmus:

- Turingovy stroje (Alan Turing)
- lambda kalkulus (Alonzo Church)
- rekurzivní funkce (Stephan Kleene)
- produkční systémy (Emil Post)
- ...

Dále můžeme uvést:

- Libovolný (obecný) programovací jazyk.

Všechny tyto modely jsou ekvivalentní z hlediska algoritmů, které jsou schopny realizovat.

Problém, který není algoritmicky řešitelný je **algoritmicky neřešitelný**.

Rozhodovací problém, který není rozhodnutelný je **nerozhodnutelný**.

**Příklad:** Následující problém zvaný **Problém zastavení (Halting problem)** je nerozhodnutelný:

## Halting problem

**Vstup:** Popis Turingova stroje  $M$  a slovo  $w$ .

**Otázka:** Zastaví se stroj  $M$  po nějakém konečném počtu kroků, pokud dostane jako svůj vstup slovo  $w$ ?

Alternativně bychom ho mohli formulovat třeba takto:

## Halting problem

**Vstup:** Zdrojový kód programu  $P$  v jazyce  $C$ , vstupní data  $x$ .

**Otázka:** Zastaví se program  $P$  po nějakém konečném počtu kroků, pokud dostane jako vstup data  $x$ ?

# Halting problem

Předpokládejme, že by existoval nějaký program, který by rozhodoval Halting problem.

Mohli bychom tedy vytvořit podprogram  $H$ , deklarovaný jako

boolean  $H(\text{String kod}, \text{String vstup})$

kde  $H(P, x)$  vrátí:

- true pokud se program  $P$  zastaví pro vstup  $x$ ,
- false pokud se program  $P$  nezastaví pro vstup  $x$ .

**Poznámka:** Řekněme, že podprogram  $H(P, x)$  by vracel false v případě, že  $P$  není syntakticky správný kód programu.

# Halting problem

S použitím podprogramu  $H$  bychom vytvořili program  $D$ , který bude provádět následující kroky:

- Načte svůj vstup do proměnné  $x$  typu `String`.
- Zavolá podprogram  $H(x, x)$ .
- Pokud podprogram  $H$  vrátil `true`, skočí do nekonečné smyčky

loop: goto loop

V případě, že  $H$  vrátil `false`, program  $D$  se ukončí.

Co udělá program  $D$ , pokud mu předložíme jako vstup jeho vlastní kód?

# Halting problem

Pokud  $D$  dostane jako vstup svůj vlastní kód, tak se buď zastaví nebo nezastaví.

- Pokud se  $D$  zastaví, tak  $H(D, D)$  vrátí `true` a  $D$  skočí do nekonečné smyčky. Spor!
- Pokud se  $D$  nezastaví, tak  $H(D, D)$  vrátí `false` a  $D$  se zastaví. Spor!

V obou případech dospějeme ke sporu a další možnost není. Nemůže tedy platit předpoklad, že  $H$  řeší Halting problem.

# Bonusový příklad

V předchozím příkladě jsme měli případ, kdy program zpracovával svůj vlastní kód. To není nic neobvyklého, například překladač může překládat svůj vlastní kód.

Uvažujme ale opačný případ: Chceme vytvořit program, který naopak svůj vlastní kód vydá jako výstup.

**Úkol:** Napište program ve vašem oblíbeném programovacím jazyce, který vypíše (například na standardní výstup) svůj vlastní zdrojový kód, přičemž ale:

- Nesmí číst data ze žádného externího zdroje (disku, klávesnice apod.).
- Nesmí být prázdný.

# Další nerozhodnutelné problémy

S jedním příkladem nerozhodnutelného problému už jsme se setkali:

## Problém

**Vstup:** Bezkontextové gramatiky  $G_1$  a  $G_2$ .

**Otázka:** Je  $L(G_1) = L(G_2)$ ?

případně

## Problém

**Vstup:** Bezkontextová gramatika  $G$  generující jazyk nad abecedou  $\Sigma$ .

**Otázka:** Je  $L(G) = \Sigma^*$ ?



# Redukce mezi problémy

Pokud máme o nějakém (rozhodovacím) problému dokázáno, že je nerozhodnutelný, můžeme ukázat nerozhodnutelnost dalších problémů pomocí tzv. redukci.

Řekněme, že  $A$  a  $B$  jsou rozhodovací problémy.

**Redukce** problému  $A$  na problém  $B$  je algoritmus  $P$ , který:

- Dostane jako vstup instanci problému  $A$  (označme ji  $x$ ).
- Jako svůj výstup (označme jej  $P(x)$ ) vyprodukuje instanci problému  $B$ .
- Platí

$$x \in A \quad \Leftrightarrow \quad P(x) \in B$$

tj. pro vstup  $x$  je v problému  $A$  odpověď **ANO** právě tehdy, když pro vstup  $P(x)$  je v problému  $B$  odpověď **ANO**.

Řekněme, že existuje redukce  $P$  problému  $A$  na problém  $B$ .

Pokud by problém  $B$  byl rozhodnutelný, pak i problém  $A$  je rozhodnutelný.

Řešení problému  $A$  pro vstup  $x$ :

- Zavoláme  $P$  se vstupem  $x$ , vrátí nám hodnotu  $P(x)$ .
- Zavoláme algoritmus řešící problém  $B$  se vstupem  $P(x)$ .  
Hodnotu, kterou nám vrátí vypíšeme jako výsledek.

Je zřejmé, že pokud  $A$  je nerozhodnutelný, tak  $B$  nemůže být rozhodnutelný.

Redukcí z Halting problému se dá ukázat nerozhodnutelnost celé řady problémů, které se týkají ověřování chování programů:

- Vydá daný program pro nějaký vstup odpověď **ANO**?
- Zastaví se daný program pro libovolný vstup?
- Dávají dva dané programy pro stejné vstupy stejný výstup?
- ...

# Další nerozhodnutelné problémy

Vstupem je množina typů kachliček, jako třeba:

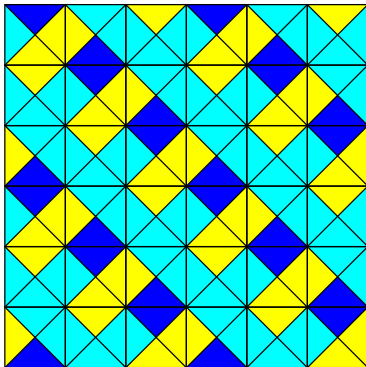


Otázka je, zda je možné použitím daných typů kachliček pokrýt každou libovolně velkou konečnou plochu tak, aby všechny kachličky spolu sousedily stejnými barvami.

**Poznámka:** Můžeme předpokládat, že máme v zásobě neomezené množství kachliček všech typů.

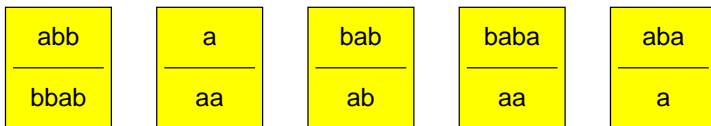
Kachličky není dovoleno otáčet.

# Další nerozhodnutelné problémy

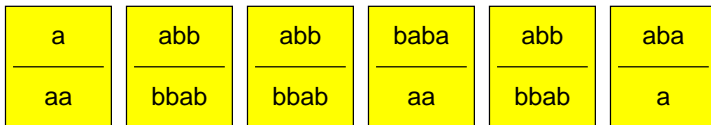


# Další nerozhodnutelné problémy

Vstupem je množina typů kartiček, jako třeba:



Otázka je, zda je možné z těchto typů kartiček vytvořit neprázdnou konečnou posloupnost, kde zřetěžením slov nahoře i dole vznikne totéž slovo. Každý typ kartičky je možné používat opakovaně.



Nahoře i dole vznikne slovo `aabbabbbabaabbaba`.

# Další nerozhodnutelné problémy

Redukcí z předchozího problému se dá snadno ukázat nerozhodnutelnost některých dalších problémů z oblasti bezkontextových gramatik:

## Problém

**Vstup:** Bezkontextové gramatiky  $G_1$  a  $G_2$ .

**Otázka:** Je  $L(G_1) \cap L(G_2) = \emptyset$ ?

## Problém

**Vstup:** Bezkontextová gramatika  $G$ .

**Otázka:** Je  $G$  nejednoznačná?

## Problém

**Vstup:** Uzavřená aritmetická formule vytvořená ze symbolů  $+$ ,  $\times$ ,  $=$ ,  $\wedge$ ,  $\vee$ ,  $\neg$ ,  $\forall$ ,  $\exists$ ,  $($ ,  $)$ , proměnných a celočíselných konstant.

**Otázka:** Je tato formule pravdivá v oboru přirozených čísel?

Příklad vstupu:

$$\forall x \exists y \forall z ((x \times y = z) \wedge (y + 5 = x))$$

**Poznámka:** Úzce souvisí s Gödelovou větou o neúplnosti.



Následující na první pohled velice podobný problém je rozhodnutelný:

## Problém

**Vstup:** Uzavřená aritmetická formule vytvořená ze symbolů  $+$ ,  $=$ ,  $\wedge$ ,  $\vee$ ,  $\neg$ ,  $\forall$ ,  $\exists$ ,  $($ ,  $)$ , proměnných a celočíselných konstant.

**Otázka:** Je tato formule pravdivá v oboru přirozených čísel?

Příklad vstupu:

$$\forall x \exists y \forall z ((x + y = z) \wedge (y + 5 = x))$$

- Stejný problém může řešit více různých algoritmů
- Potřebovali bychom rozhodnout, který algoritmus je lepší
- Nestačí algoritmy naprogramovat a změřit čas výpočtu
- Potřebovali bychom nějakou kvantitativní charakteristiku (složitost), podle které budeme algoritmy srovnávat
- Nejvhodnější je složitost definovat jako funkci velikosti vstupu

## Problém „Vyhledávání“

**Vstup:** Sekvence prvků  $x, a_1, a_2, \dots, a_n$ .

**Výstup:** Pokud  $a_i = x$  je výstupem  $i$  (pokud jich je více, tak nejmenší), jinak vypíše 0.

```

                                READ   R3
                                LOAD   $0,R1
cyklus:                        READ   R2
                                JZERO  R2,nenasel
                                ADD    $1,R1
                                SUB    R3,R2
                                JZERO  R2,nasel
                                JUMP   cyklus
nenasel:                        LOAD   $0,R1
nasel:                          WRITE R1
                                END
```

	READ	R3
	LOAD	\$0,R1
cyklus:	READ	R2
	JZERO	R2,nenasel
	ADD	\$1,R1
	SUB	R3,R2
	JZERO	R2,nasel
	JUMP	cyklus
nenasel:	LOAD	\$0,R1
nasel:	WRITE	R1
	END	

Pro sekvenci

9, 1, 8, 9, 7, 2, 0

R1

R2

R3

Výstup:

Instrukcí: 0

	READ	R3
	LOAD	\$0,R1
cyklus:	READ	R2
	JZERO	R2,nenasel
	ADD	\$1,R1
	SUB	R3,R2
	JZERO	R2,nasel
	JUMP	cyklus
nenasel:	LOAD	\$0,R1
nasel:	WRITE	R1
	END	

Pro sekvenci  
9, 1, 8, 9, 7, 2, 0

R1  
R2  
R3 9

Výstup:

Instrukcí: 1

```
cyklus:  READ  R3
         LOAD  $0,R1
         READ  R2
         JZERO R2,nenasel
         ADD   $1,R1
         SUB   R3,R2
         JZERO R2,nasel
         JUMP  cyklus
nenasel: LOAD  $0,R1
nasel:   WRITE R1
        END
```

Pro sekvenci  
9, 1, 8, 9, 7, 2, 0

R1 0  
R2  
R3 9

Výstup:

Instrukcí: 2

	READ	R3
	LOAD	\$0,R1
cyklus:	READ	R2
	<b>JZERO</b>	R2,nenasel
	ADD	\$1,R1
	SUB	R3,R2
	JZERO	R2,nasel
	JUMP	cyklus
nenasel:	LOAD	\$0,R1
nasel:	WRITE	R1
	END	

Pro sekvenci  
9, 1, 8, 9, 7, 2, 0

R1	0
R2	1
R3	9

Výstup:

Instrukcí: 3

	READ	R3
	LOAD	\$0,R1
cyklus:	READ	R2
	JZERO	R2,nenasel
	ADD	\$1,R1
	SUB	R3,R2
	JZERO	R2,nasel
	JUMP	cyklus
nenasel:	LOAD	\$0,R1
nasel:	WRITE	R1
	END	

Pro sekvenci

9, 1, 8, 9, 7, 2, 0

R1 0

R2 1

R3 9

Výstup:

Instrukcí: 4



	READ	R3
	LOAD	\$0,R1
cyklus:	READ	R2
	JZERO	R2,nenasel
	ADD	\$1,R1
	<b>SUB</b>	R3,R2
	JZERO	R2,nasel
	JUMP	cyklus
nenasel:	LOAD	\$0,R1
nasel:	WRITE	R1
	END	

Pro sekvenci

9, 1, 8, 9, 7, 2, 0

R1 1

R2 1

R3 9

Výstup:

Instrukcí: 5

	READ	R3
	LOAD	\$0,R1
cyklus:	READ	R2
	JZERO	R2,nenasel
	ADD	\$1,R1
	SUB	R3,R2
	<b>JZERO</b>	R2,nasel
	JUMP	cyklus
nenasel:	LOAD	\$0,R1
nasel:	WRITE	R1
	END	

Pro sekvenci

9, 1, 8, 9, 7, 2, 0

R1	1
R2	-8
R3	9

Výstup:

Instrukcí: 6

	READ	R3
	LOAD	\$0,R1
cyklus:	READ	R2
	JZERO	R2,nenasel
	ADD	\$1,R1
	SUB	R3,R2
	JZERO	R2,nasel
	<b>JUMP</b>	cyklus
nenasel:	LOAD	\$0,R1
nasel:	WRITE	R1
	END	

Pro sekvenci

9, 1, 8, 9, 7, 2, 0

R1 1

R2 -8

R3 9

Výstup:

Instrukcí: 7

	READ	R3
	LOAD	\$0,R1
cyklus:	READ	R2
	JZERO	R2,nenasel
	ADD	\$1,R1
	SUB	R3,R2
	JZERO	R2,nasel
	JUMP	cyklus
nenasel:	LOAD	\$0,R1
nasel:	WRITE	R1
	END	

Pro sekvenci  
9, 1, 8, 9, 7, 2, 0

R1	1
R2	-8
R3	9

Výstup:

Instrukcí: 8

	READ	R3
	LOAD	\$0,R1
cyklus:	READ	R2
	JZERO	R2,nenasel
	ADD	\$1,R1
	SUB	R3,R2
	JZERO	R2,nasel
	JUMP	cyklus
nenasel:	LOAD	\$0,R1
nasel:	WRITE	R1
	END	

Pro sekvenci  
9, 1, 8, 9, 7, 2, 0

R1	1
R2	8
R3	9

Výstup:

Instrukcí: 9

```
cyklus:  READ  R3
         LOAD  $0,R1
         READ  R2
         JZERO R2,nenasel
         ADD   $1,R1
         SUB   R3,R2
         JZERO R2,nasel
         JUMP  cyklus
nenasel: LOAD  $0,R1
nasel:   WRITE R1
        END
```

Pro sekvenci  
9, 1, 8, 9, 7, 2, 0

R1 1  
R2 8  
R3 9

Výstup:

Instrukcí: 10

	READ	R3
	LOAD	\$0,R1
cyklus:	READ	R2
	JZERO	R2,nenasel
	ADD	\$1,R1
	<b>SUB</b>	R3,R2
	JZERO	R2,nasel
	JUMP	cyklus
nenasel:	LOAD	\$0,R1
nasel:	WRITE	R1
	END	

Pro sekvenci

9, 1, 8, 9, 7, 2, 0

R1 2

R2 8

R3 9

Výstup:

Instrukcí: 11

	READ	R3
	LOAD	\$0,R1
cyklus:	READ	R2
	JZERO	R2,nenasel
	ADD	\$1,R1
	SUB	R3,R2
	<b>JZERO</b>	R2,nasel
	JUMP	cyklus
nenasel:	LOAD	\$0,R1
nasel:	WRITE	R1
	END	

Pro sekvenci  
9, 1, 8, **9**, **7**, **2**, **0**

R1	2
R2	-1
R3	9

Výstup:

Instrukcí: 12



```
cyklus:  READ  R3
         LOAD  $0,R1
         READ  R2
         JZERO R2,nenasel
         ADD   $1,R1
         SUB   R3,R2
         JZERO R2,nasel
         JUMP  cyklus
nenasel: LOAD  $0,R1
nasel:   WRITE R1
        END
```

Pro sekvenci  
9, 1, 8, 9, 7, 2, 0

R1 2  
R2 -1  
R3 9

Výstup:

Instrukcí: 13

	READ	R3
	LOAD	\$0,R1
cyklus:	READ	R2
	JZERO	R2,nenasel
	ADD	\$1,R1
	SUB	R3,R2
	JZERO	R2,nasel
	JUMP	cyklus
nenasel:	LOAD	\$0,R1
nasel:	WRITE	R1
	END	

Pro sekvenci  
9, 1, 8, 9, 7, 2, 0

R1	2
R2	-1
R3	9

Výstup:

Instrukcí: 14

	READ	R3
	LOAD	\$0,R1
cyklus:	READ	R2
	JZERO	R2,nenasel
	ADD	\$1,R1
	SUB	R3,R2
	JZERO	R2,nasel
	JUMP	cyklus
nenasel:	LOAD	\$0,R1
nasel:	WRITE	R1
	END	

Pro sekvenci  
9, 1, 8, 9, 7, 2, 0

R1	2
R2	9
R3	9

Výstup:

Instrukcí: 15

```

      READ   R3
      LOAD   $0,R1
cyklus: READ   R2
      JZERO  R2,nenasel
      ADD    $1,R1
      SUB    R3,R2
      JZERO  R2,nasel
      JUMP   cyklus
nenasel: LOAD   $0,R1
nasel:  WRITE  R1
      END
```

Pro sekvenci  
9, 1, 8, 9, 7, 2, 0

R1 2  
R2 9  
R3 9

Výstup:

Instrukcí: 16

	READ	R3
	LOAD	\$0,R1
cyklus:	READ	R2
	JZERO	R2,nenasel
	ADD	\$1,R1
	<b>SUB</b>	R3,R2
	JZERO	R2,nasel
	JUMP	cyklus
nenasel:	LOAD	\$0,R1
nasel:	WRITE	R1
	END	

Pro sekvenci

9, 1, 8, 9, 7, 2, 0

R1 3

R2 9

R3 9

Výstup:

Instrukcí: 17

	READ	R3
	LOAD	\$0,R1
cyklus:	READ	R2
	JZERO	R2,nenasel
	ADD	\$1,R1
	SUB	R3,R2
	<b>JZERO</b>	R2,nasel
	JUMP	cyklus
nenasel:	LOAD	\$0,R1
nasel:	WRITE	R1
	END	

Pro sekvenci

9, 1, 8, 9, **7, 2, 0**

R1 3

R2 0

R3 9

Výstup:

Instrukcí: 18

```

cyklus:  READ   R3
         LOAD   $0,R1
         READ   R2
         JZERO  R2,nenasel
         ADD    $1,R1
         SUB    R3,R2
         JZERO  R2,nasel
         JUMP   cyklus
nenasel: LOAD   $0,R1
nasel:   WRITE  R1
        END
```

Pro sekvenci  
9, 1, 8, 9, 7, 2, 0

R1 3  
R2 0  
R3 9

Výstup:

Instrukcí: 19

```

      READ   R3
      LOAD   $0,R1
cyklus: READ   R2
      JZERO  R2,nenasel
      ADD    $1,R1
      SUB    R3,R2
      JZERO  R2,nasel
      JUMP   cyklus
nenasel: LOAD   $0,R1
nasel:  WRITE  R1
      END
```

Pro sekvenci  
9, 1, 8, 9, 7, 2, 0

R1 3  
R2 0  
R3 9

Výstup: 3

Instrukcí: 20



	READ	R3	
	LOAD	\$0,R1	Pro sekvenci
cyklus:	READ	R2	9, 1, 8, 9, 7, 2, 0
	JZERO	R2, nenasel	
	ADD	\$1,R1	R1 3
	SUB	R3,R2	R2 0
	JZERO	R2, nasel	R3 9
	JUMP	cyklus	
nenasel:	LOAD	\$0,R1	Výstup: 3
nasel:	WRITE	R1	
	END		Instrukcí: 21

- Počet instrukcí v uvedeném příkladě:  $2 + 2 * 6 + 5 + 2 = 21$
- Pokud najde  $i$ , tž.  $a_i = x$ :  $2 + (i - 1) * 6 + 5 + 2 = 6i + 3$
- Pokud nenajde takové  $i$ :  $2 + n * 6 + 2 + 3 = 6n + 7$

## Definice

**Délka výpočtu** algoritmu  $\mathcal{A}$  na vstupu  $x$  rozumíme počet kroků stroje RAM implementujícího algoritmus  $\mathcal{A}$ , které vykoná na vstupu  $x$  až do svého zastavení.

## Definice

**Časová složitost** algoritmu  $\mathcal{A}$  (v nejhorším případě) je funkce  $t_{\mathcal{A}} : \mathbb{N} \rightarrow \mathbb{N}$ , kde  $t_{\mathcal{A}}(n)$  udává maximální délku výpočtu mezi všemi vstupy  $x$  délky  $n$ .

**Poznámka:** Předpokládá se, že výpočet stroje RAM vždy skončí. Jako velikost vstupu uvažujeme počet znaků, kterými je vstup zapsán.

**Příklad:** V příkladu s vyhledáváním uvažujme, že na vstupu jsou jen jednobytová čísla. Abeceda k zapsání vstupu je  $\{0, 1, \dots, 255\}$ . Nejhorší případ nastává, pokud není nalezeno hledané číslo.

Složitost je  $t_{\mathcal{A}}(n) = (n - 2) * 6 + 7 = 6n - 5$ .

## Definice

**Časová složitost** algoritmu  $\mathcal{A}$  **v nejlepším případě** je funkce  $t_{\mathcal{A}} : \mathbb{N} \rightarrow \mathbb{N}$ , kde  $t_{\mathcal{A}}(n)$  udává minimální délku výpočtu mezi všemi vstupy  $x$  délky  $n$ .

**Příklad:** V našem případě je časová složitost v nejlepším případě konstantní funkce  $t_{\mathcal{A}}(n) = 9$ . Nejrychleji totiž výpočet skončí, jestliže hned první číslo prohledávané posloupnosti je ono hledané.

**Poznámka:** Časovou složitost v nejlepším případě většinou nemá smysl uvažovat. Jde často o konstantní funkci nebo je dána nějakými triviálními vstupy.

## Definice

**Časová složitost** algoritmu  $\mathcal{A}$  v **průměrném případě** je funkce  $t_{\mathcal{A}} : \mathbb{N} \rightarrow \mathbb{N}$ , kde  $t_{\mathcal{A}}(n)$  udává průměrnou délku výpočtu mezi všemi vstupy  $x$  délky  $n$  (neboli střední hodnotu délky výpočtu na pravděpodobnostním prostoru všech vstupů).

**Příklad:** Uvažujme v našem vyhledávání jednobytová čísla. Spočítáme průměrnou složitost pro  $n = 5$  (uvažujeme jen správné vstupy délky 5).

- $x,x,y,z,0$ : možností:  $255^3$ , délka výpočtu:  $6 \cdot 1 + 3 = 9$
- $x,y,x,z,0$ : možností:  $255^2 \cdot 254$ , délka výpočtu:  $6 \cdot 2 + 3 = 15$
- $x,y,z,x,0$ : možností:  $255 \cdot 254^2$ , délka výpočtu:  $6 \cdot 3 + 3 = 21$
- $x,y,z,u,0$ : možností:  $255^4 - (255^3 + 255^2 \cdot 254 + 255 \cdot 254^2)$ , délka výpočtu:  $6 \cdot 3 + 7 = 25$
- Průměr tedy je  $t_{\mathcal{A}}(5) = (255^3 \cdot 9 + 255^2 \cdot 254 \cdot 15 + 255 \cdot 254^2 \cdot 21 + (255^4 - (255^3 + 255^2 \cdot 254 + 255 \cdot 254^2)) \cdot 25) / 255^4 = 24.88$

Obecně by se dala složitost vyjádřit:

$$t_{\mathcal{A}}(n) = \frac{(255^{n-1} - \sum_{i=1}^{n-2} 255^i 254^{n-2-i})(6n-5)}{255^{n-1}} + \\ + \frac{\sum_{i=1}^{n-2} (255^i 254^{n-2-i} (6n-6i-3))}{255^{n-1}}$$

## Poznámka:

- Časová složitost v průměrném případě většinou nejvíce vypovídá o reálné složitosti algoritmu v praxi.
- Je obvykle nejtěžší ji vyjádřit.
- Často není podstatný rozdíl mezi složitostí v průměrném a nejhorším případě. Někdy je ale rozdíl významný.

# Rychlost růstu funkcí

Program zpracovává vstup velikosti  $n$ .

Předpokládejme, že pro vstup velikosti  $n$  provede  $f(n)$  operací, a že provedení jedné operace trvá  $1 \mu\text{s}$  ( $10^{-6}$  s).

	$n$							
$f(n)$	20	40	60	80	100	200	500	1000
$n$	$20 \mu\text{s}$	$40 \mu\text{s}$	$60 \mu\text{s}$	$80 \mu\text{s}$	0.1 ms	0.2 ms	0.5 ms	1 ms
$n \log n$	$86 \mu\text{s}$	0.213 ms	0.354 ms	0.506 ms	0.664 ms	1.528 ms	4.48 ms	9.96 ms
$n^2$	0.4 ms	1.6 ms	3.6 ms	6.4 ms	10 ms	40 ms	0.25 s	1 s
$n^3$	8 ms	64 ms	0.216 s	0.512 s	1 s	8 s	125 s	16.7 min.
$n^4$	0.16 s	2.56 s	12.96 s	42 s	100 s	26.6 min.	17.36 hod.	11.57 dní
$2^n$	1.05 s	12.75 dní	36560 let	$38.3 \cdot 10^9$ let	$40.1 \cdot 10^{15}$ let	$50 \cdot 10^{45}$ let	$10.4 \cdot 10^{136}$ let	–
$n!$	77147 let	$2.59 \cdot 10^{34}$ let	$2.64 \cdot 10^{68}$ let	$2.27 \cdot 10^{105}$ let	$2.96 \cdot 10^{144}$ let	–	–	–

# Rychlost růstu funkcí

Uvažujme 3 algoritmy se složitostmi  $t_1(n) = n$ ,  $t_2(n) = n^3$ ,  $t_3(n) = 2^n$ . Náš počítač zvládne v reálném čase (kolik jsme ochotni počkat)  $10^{12}$  kroků.

Složitost	Velikost vstupu
$t_1(n) = n$	$10^{12}$
$t_2(n) = n^3$	$10^4$
$t_3(n) = 2^n$	40

# Rychlost růstu funkcí

Uvažujme 3 algoritmy se složitostmi  $t_1(n) = n$ ,  $t_2(n) = n^3$ ,  $t_3(n) = 2^n$ . Náš počítač zvládne v reálném čase (kolik jsme ochotni počkat)  $10^{12}$  kroků.

Složitost	Velikost vstupu
$t_1(n) = n$	$10^{12}$
$t_2(n) = n^3$	$10^4$
$t_3(n) = 2^n$	40

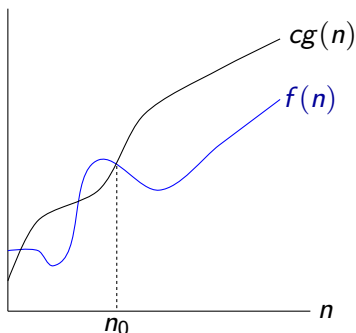
Nyní počítač 1000 násobně zrychlíme. Zvládne tedy  $10^{15}$  kroků.

Složitost	Velikost vstupu	Nárůst
$t_1(n) = n$	$10^{15}$	$1000\times$
$t_2(n) = n^3$	$10^5$	$10\times$
$t_3(n) = 2^n$	50	+10



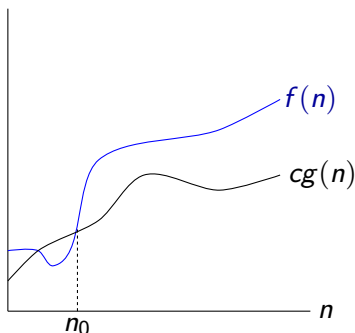
- Přesnou složitost bývá problém vyjádřit
- Přesná složitost je silně závislá na konkrétním zvoleném modelu a konkrétní implementaci
- Složitost nás většinou zajímá hlavně pro velké vstupy. Pro malé vstupy obvykle i neefektivní algoritmus proběhne rychle
- Ve většině případů pro představu o složitosti stačí nejvýznamnější člen složitostní funkce
- Proto zavádíme tzv. asymptotickou notaci, která nám umožní zanedbat konstanty a méně významné členy funkcí

# Asymptotická notace



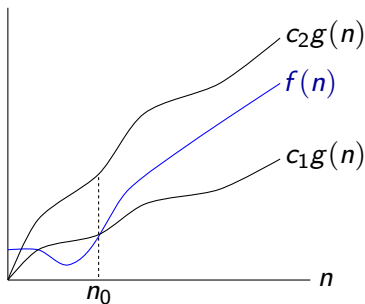
## Definice

Pro libovolné dvě funkce  $f, g : \mathbb{N} \rightarrow \mathbb{N}$  řekneme, že  $f \in O(g)$ , označujeme také jako  $f(n) \in O(g(n))$ , právě tehdy, když platí  $(\exists c > 0)(\exists n_0 \geq 0)(\forall n \geq n_0) : f(n) \leq cg(n)$ .



## Definice

Pro libovolné dvě funkce  $f, g : \mathbb{N} \rightarrow \mathbb{N}$  řekneme, že  $f \in \Omega(g)$ , označujeme také jako  $f(n) \in \Omega(g(n))$ , právě tehdy, když platí  $g \in O(f)$  neboli  $(\exists c > 0)(\exists n_0 \geq 0)(\forall n \geq n_0) : 0 \leq c g(n) \leq f(n)$ .



## Definice

Pro libovolné dvě funkce  $f, g : \mathbb{N} \rightarrow \mathbb{N}$  řekneme, že  $f \in \Theta(g)$ , označujeme také jako  $f(n) \in \Theta(g(n))$ , právě tehdy, když platí  $g \in O(f)$  a  $f \in O(g)$  neboli

$$(\exists c_1 > 0)(\exists c_2 > 0)(\exists n_0 \geq 0)(\forall n \geq n_0) : 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n).$$

## Definice

Pro libovolné dvě funkce  $f, g : \mathbb{N} \rightarrow \mathbb{N}$  řekneme, že  $f \in o(g)$ , označujeme také jako  $f(n) \in o(g(n))$ , právě tehdy, když platí

$$(\forall c > 0)(\exists n_0 > 0)(\forall n \geq n_0) : 0 \leq f(n) < c g(n).$$

## Definice

Pro libovolné dvě funkce  $f, g : \mathbb{N} \rightarrow \mathbb{N}$  řekneme, že  $f \in \omega(g)$ , označujeme také jako  $f(n) \in \omega(g(n))$ , právě tehdy, když platí

$$(\forall c > 0)(\exists n_0 > 0)(\forall n \geq n_0) : 0 \leq c g(n) < f(n).$$

**Poznámka:** Často se píše  $f(n) = O(g(n))$  místo  $f(n) \in O(g(n))$

**Příklad:** Při analýze složitosti vyhledávání čísla v posloupnosti jsme dostali  $t_A(n) = 6n - 5$ . Platí  $t_A \in O(n)$ . Takže můžeme říct, že složitost našeho algoritmu je v  $O(n)$ .

**Příklad:**

$$\begin{array}{ll} n \in O(n^2) & n^3 \in O(n^4) \\ 1000n \in O(n) & 0.00001n^2 - 10^{10}n \in \Theta(10^{10}n^2) \\ 2^{\log_2 n} \in \Theta(n) & n^3 - n^2 \log^3 n + 1000n - 10^{100} \in \Theta(n^3) \\ n \in o(n^2) & n^3 + 1000n - 10^{100} \notin o(n^3) \\ n^2 \in \omega(n) & n^3 + n^2 \in \Omega(n^2) \\ n^3 + 2^n \in \Omega(n^2) & n! \in \Omega(2^n) \end{array}$$

- Ne vždy je algoritmus s lepší asymptotickou složitostí výhodnější v praxi
- Můžeme si to ilustrovat na algoritmech pro třídění

Algoritmus	Nejhorší	Průměrný
Bubblesort	$O(n^2)$	$O(n^2)$
Heapsort	$O(n \log n)$	$O(n \log n)$
Quicksort	$O(n^2)$	$O(n \log n)$

- Quicksort má horší asymptotickou složitost v nejhorším případě než heapsort, stejnou asymptotickou složitost v průměrném případě a přesto je v praxi nejrychlejší

- Ne vždy je vhodné uvažovat složitost jako funkci počtu znaků abecedy potřebných pro zápis vstupu
- Pro grafy často bývá složitost vyjádřena jako funkce dvou parametrů:  $m$  (počet hran) a  $n$  (počet vrcholů)
- Pro operace s posloupnostmi čísel (třídění, vyhledávání,...) bývá  $n$  počet čísel a ne cifer nebo bitů
- Pro rozhodování o prvočíselnosti nemůžeme brát  $n = 1$  (vstupem je jedno číslo), ale uvažujeme počet bitů (popřípadě cifer)
- Při práci s maticemi uvažujeme většinou  $n$  jako počet řádků a  $m$  počet sloupců (popř. jen  $n$  je-li sloupců a řádku přibližně stejně) a ne přímo počet prvků matice



- O algoritmu s časovou složitostí  $t_{\mathcal{A}}(n)$  řekneme, že je:
  - logaritmický**, pokud  $t_{\mathcal{A}}(n) \in \Theta(\log n)$
  - lineární**, pokud  $t_{\mathcal{A}}(n) \in \Theta(n)$
  - kvadratický**, pokud  $t_{\mathcal{A}}(n) \in \Theta(n^2)$
  - kubický**, pokud  $t_{\mathcal{A}}(n) \in \Theta(n^3)$
  - polynomiální**, pokud  $t_{\mathcal{A}}(n) \in \Theta(n^k)$  pro nějaké  $k \in \mathbb{N}$
  - exponenciální**, pokud  $t_{\mathcal{A}}(n) \in \Theta(k^n)$  pro nějaké  $k \in \mathbb{N}$
- Pro exponenciální složitost se používá také  $2^{\Theta(n)}$ , protože potom již nemusíme uvažovat různé základy mocniny.
- Pro logaritmickou složitost nehraje roli základ logaritmu, protože  $\log_k n \in \Theta(\log_l n)$  pro všechna  $k, l > 0$ .

- Zatím jsme uvažovali, že provedení všech instrukcí trvá stejně dlouho
- V praxi mohou být některé instrukce časově náročnější
- Místo počtu instrukcí v nejhorším případě nás může zajímat počet provedení jedné konkrétní instrukce v nejhorším případě
- Pokud známe časy provedení jednotlivých instrukcí, můžeme si spočítat počty jejich provedení a čas běhu potom dostaneme jako  $\sum_i n_i t_i$ , kde  $n_i$  je počet provedení instrukce  $i$  a  $t_i$  je čas potřebný k vykonání této instrukce

# Prostorová (paměťová) složitost algoritmů

- Zatím jsme se zajímali o čas, který potřebujeme k výpočtu
- Často bývá kritickou velikost paměti

## Definice

**Velikost paměti** stroje RAM  $\mathcal{M}$  na vstupu  $x$  rozumíme počet buněk paměti, které stroj  $\mathcal{M}$  během svého výpočtu navštíví.

## Definice

**Prostorová složitost** stroje RAM  $\mathcal{M}$  (v nejhorším případě) je funkce  $s_{\mathcal{M}} : \mathbb{N} \rightarrow \mathbb{N}$ , kde  $s_{\mathcal{M}}(n)$  udává maximální velikost paměti ze všech výpočtů nad vstupy  $x$  délky  $n$ .

# Prostorová (paměťová) složitost algoritmů

- Pro konkrétní problém můžeme mít dva algoritmy takové, že jeden má menší prostorovou složitost a druhý zase časovou složitost
- Prostorová složitost je někdy kritičtější než časová. Když dojde paměť, ve výpočtu se nedá pokračovat
- Je-li časová složitost algoritmu v  $O(f(n))$  je i prostorová v  $O(f(n))$  (počet buněk navštívených RAMem nemůže být větší než počet kroků, protože v každém kroku navšíví maximálně jednu buňku)

# Složitost problémů

- Zřejmě jsou některé problémy „složitější“ než jiné
- Jestliže pro nějaký problém známe algoritmus v  $\Theta(n^3)$ , který jej řeší, máme horní odhad pro složitost problému. Ale může existovat algoritmus v  $\Theta(n^2)$ , který daný problém také řeší.

## Definice

Pro funkci  $f : \mathbb{N} \rightarrow \mathbb{N}$  rozumíme **třídou časové složitosti**  $\mathcal{T}(f)$  (nebo  $\mathcal{T}(f(n))$ ) množinu těch problémů, které jsou řešeny RAMy s časovou složitostí v  $O(f(n))$ .

## Definice

Pro funkci  $f : \mathbb{N} \rightarrow \mathbb{N}$  rozumíme **třídou prostorové složitosti**  $\mathcal{S}(f)$  (nebo  $\mathcal{S}(f(n))$ ) množinu těch problémů, které jsou řešeny RAMy s prostorovou složitostí v  $O(f(n))$ .

- Pokud chceme ukázat, že problém je v  $\mathcal{T}(f(n))$  stačí najít algoritmus se složitostí v  $O(f(n))$  řešící daný problém.
- Pokud chceme ukázat, že problém není v  $\mathcal{T}(f(n))$ , musí ukázat, že neexistuje žádný algoritmus se složitostí v  $O(f(n))$  řešící daný problém. To je obvykle dost obtížné.
- Třídy složitosti jsou závislé na zvoleném modelu (v našem případě stroj RAM). Pro jiný model bychom mohli dostat jiné třídy.

## Definice

$$\text{PTIME} = \bigcup_{k=0}^{\infty} \mathcal{I}(n^k)$$

- Třída PTIME se ukazuje jako vhodná aproximace zvládnutelných problémů
- Není to ale přesné, algoritmus v  $\Theta(n^{1000000})$  se moc za rychlý považovat nedá. Naopak i algoritmy s exponenciální složitostí mohou být použitelné, pokud nejsou určeny pro velké vstupy.
- Pokud je znám pro nějaký problém polynomiální algoritmus, často se najde i polynomiální algoritmus s nízkým stupněm polynomu (řekněme menším než 6)
- Třída PTIME není závislá na zvoleném (rozumném) modelu

# Složitosti konkrétních problémů

Mějme dáno pole  $A$  []

- Nalezení konkrétního prvku v  $A$  lineárním prohledáváním je v  $O(n)$ .
- Časová složitost setřídění  $A$  je  $O(n \log n)$ .
- V setříděném poli  $A$  lze nalézt prvek binárním vyhledáváním v čase  $O(\log n)$ .

Vezměme aritmetiku dlouhých  $n$ -místných čísel.

- Sečíst dvě taková čísla lze v čase  $O(n)$  a vynásobit v čase  $O(n^2)$  běžnými “školními” postupy.
- Sofistikovaný Strassenův algoritmus vynásobí dvě  $n$ -místná čísla v čase  $O(n \log n \log \log n)$ .

Mějme daný graf  $G$  s  $n$  vrcholy a  $m$  hranami ( $m = O(n^2)$ ).

- Komponenty souvislosti  $G$  najdeme v čase  $O(n + m)$ .
- Nejkratší cestu mezi dvěma vrcholy vypočteme v čase  $O(n + m \log n)$ .
- Minimální kostru nalezneme hladově v čase  $O(n + m \log n)$ .
- Rovinné nakreslení grafu lze nalézt (pokud existuje) v čase  $O(n)$ .

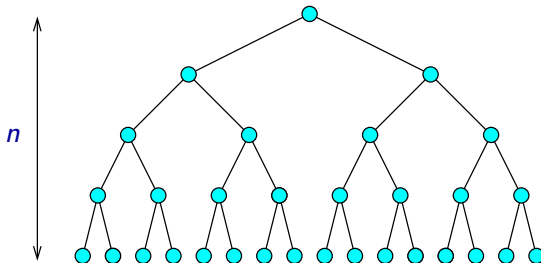


# Rekurentní vztahy

Někdy je vhodné vyjadřovat hodnotu funkce pomocí **rekurentních vztahů**.

Rekurentní vztah je rovnice nebo nerovnice, kde je hodnota funkce pro větší argumenty vyjádřena pomocí hodnot, kterých funkce nabývá pro menší argumenty.

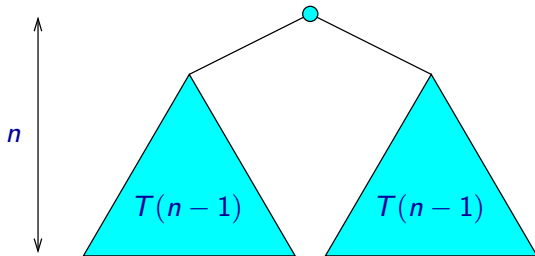
**Příklad:** Kolik vrcholů má úplný binární strom výšky  $n$ ?



# Rekurentní vztahy

$T(n)$  – počet vrcholů úplného binárního stromu výšky  $n$

$$T(n) = \begin{cases} 1 & \text{pro } n = 0 \\ 2 \cdot T(n-1) + 1 & \text{pro } n > 0 \end{cases}$$



$$T(n) = \begin{cases} 1 & \text{pro } n = 0 \\ 2 \cdot T(n-1) + 1 & \text{pro } n > 0 \end{cases}$$

$$T(0) = 1$$

$$T(1) = 2 \cdot T(0) + 1 = 2 \cdot 1 + 1 = 3$$

$$T(2) = 2 \cdot T(1) + 1 = 2 \cdot 3 + 1 = 7$$

$$T(3) = 2 \cdot T(2) + 1 = 2 \cdot 7 + 1 = 15$$

$$T(4) = 2 \cdot T(3) + 1 = 2 \cdot 15 + 1 = 31$$

$$T(5) = 2 \cdot T(4) + 1 = 2 \cdot 31 + 1 = 63$$

$$T(6) = 2 \cdot T(5) + 1 = 2 \cdot 63 + 1 = 127$$

$$T(7) = 2 \cdot T(6) + 1 = 2 \cdot 127 + 1 = 255$$

$$T(8) = 2 \cdot T(7) + 1 = 2 \cdot 255 + 1 = 511$$

$$T(9) = 2 \cdot T(8) + 1 = 2 \cdot 511 + 1 = 1023$$

$$T(10) = 2 \cdot T(9) + 1 = 2 \cdot 1023 + 1 = 2047$$

...

Pokud chceme vyjádřit hodnotu  $T(n)$  nerekurentně, jednou z možností je použít tzv. **substituční metodu**:

- 1 Uhodnout správné řešení.
- 2 Ověřit jeho správnost pomocí matematické indukce.

## Příklad:

$$T(n) = \begin{cases} 1 & \text{pro } n = 0 \\ 2 \cdot T(n-1) + 1 & \text{pro } n > 0 \end{cases}$$

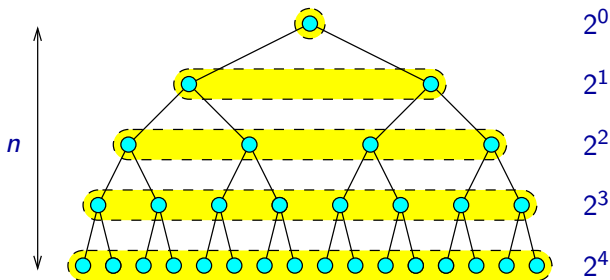
Uhodneme, že  $T(n) = 2^{n+1} - 1$ .

Indukcí ověříme, že tomu tak skutečně je:

- Báze ( $i = 0$ ):  $T(0) = 2^{0+1} - 1 = 1$
- Indukční krok ( $i > 0$ ):

$$\begin{aligned} T(i) &= 2 \cdot T(i-1) + 1 \\ &= 2 \cdot (2^{(i-1)+1} - 1) + 1 \\ &= 2 \cdot 2^i - 2 + 1 \\ &= 2^{i+1} - 1 \end{aligned}$$

**Poznámka:** Alternativní možností je spočítat počty vrcholů v jednotlivých „vrstvách“ stromu.



$$T(n) = \sum_{i=0}^n 2^i = \frac{2^{n+1} - 1}{2 - 1} = 2^{n+1} - 1$$

Rekurentní vztahy se často používají při zkoumání časové složitosti rekurzivních algoritmů.

**Příklad:** Algoritmus **MERGE-SORT**.

Hlavní myšlenka algoritmu: Dvě setříděné posloupnosti snadno spojíme do jediné setříděné posloupnosti.

Pokud mají obě posloupnosti dohromady  $n$  prvků, vyžaduje tato operace  $n$  kroků.

34	42	58	61
----	----	----	----



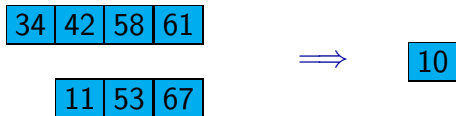
10	11	53	67
----	----	----	----

Rekurentní vztahy se často používají při zkoumání časové složitosti rekurzivních algoritmů.

**Příklad:** Algoritmus **MERGE-SORT**.

Hlavní myšlenka algoritmu: Dvě setříděné posloupnosti snadno spojíme do jediné setříděné posloupnosti.

Pokud mají obě posloupnosti dohromady  $n$  prvků, vyžaduje tato operace  $n$  kroků.



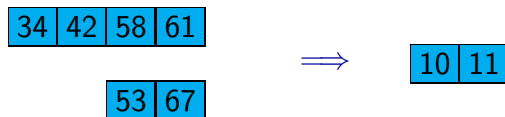


Rekurentní vztahy se často používají při zkoumání časové složitosti rekurzivních algoritmů.

**Příklad:** Algoritmus **MERGE-SORT**.

Hlavní myšlenka algoritmu: Dvě setříděné posloupnosti snadno spojíme do jediné setříděné posloupnosti.

Pokud mají obě posloupnosti dohromady  $n$  prvků, vyžaduje tato operace  $n$  kroků.



Rekurentní vztahy se často používají při zkoumání časové složitosti rekurzivních algoritmů.

**Příklad:** Algoritmus **MERGE-SORT**.

Hlavní myšlenka algoritmu: Dvě setříděné posloupnosti snadno spojíme do jediné setříděné posloupnosti.

Pokud mají obě posloupnosti dohromady  $n$  prvků, vyžaduje tato operace  $n$  kroků.

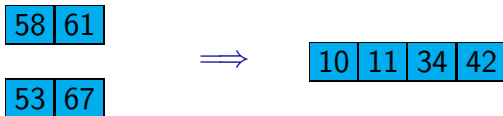


Rekurentní vztahy se často používají při zkoumání časové složitosti rekurzivních algoritmů.

**Příklad:** Algoritmus **MERGE-SORT**.

Hlavní myšlenka algoritmu: Dvě setříděné posloupnosti snadno spojíme do jediné setříděné posloupnosti.

Pokud mají obě posloupnosti dohromady  $n$  prvků, vyžaduje tato operace  $n$  kroků.

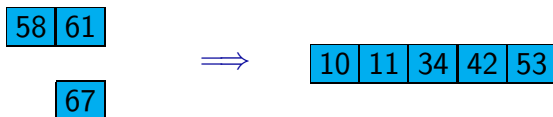


Rekurentní vztahy se často používají při zkoumání časové složitosti rekurzivních algoritmů.

**Příklad:** Algoritmus **MERGE-SORT**.

Hlavní myšlenka algoritmu: Dvě setříděné posloupnosti snadno spojíme do jediné setříděné posloupnosti.

Pokud mají obě posloupnosti dohromady  $n$  prvků, vyžaduje tato operace  $n$  kroků.

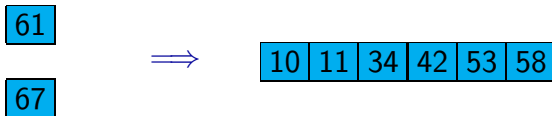


Rekurentní vztahy se často používají při zkoumání časové složitosti rekurzivních algoritmů.

**Příklad:** Algoritmus **MERGE-SORT**.

Hlavní myšlenka algoritmu: Dvě setříděné posloupnosti snadno spojíme do jediné setříděné posloupnosti.

Pokud mají obě posloupnosti dohromady  $n$  prvků, vyžaduje tato operace  $n$  kroků.



Rekurentní vztahy se často používají při zkoumání časové složitosti rekurzivních algoritmů.

**Příklad:** Algoritmus **MERGE-SORT**.

Hlavní myšlenka algoritmu: Dvě setříděné posloupnosti snadno spojíme do jediné setříděné posloupnosti.

Pokud mají obě posloupnosti dohromady  $n$  prvků, vyžaduje tato operace  $n$  kroků.

67



10	11	34	42	53	58	61
----	----	----	----	----	----	----

Rekurentní vztahy se často používají při zkoumání časové složitosti rekurzivních algoritmů.

**Příklad:** Algoritmus **MERGE-SORT**.

Hlavní myšlenka algoritmu: Dvě setříděné posloupnosti snadno spojíme do jediné setříděné posloupnosti.

Pokud mají obě posloupnosti dohromady  $n$  prvků, vyžaduje tato operace  $n$  kroků.



10	11	34	42	53	58	61	67
----	----	----	----	----	----	----	----

MERGE-SORT( $A, p, r$ )

```
1  if  $r - p > 1$ 
2      then  $q \leftarrow \lfloor (p + r) / 2 \rfloor$ 
3          MERGE-SORT( $A, p, q$ )
4          MERGE-SORT( $A, q, r$ )
5          MERGE( $A, p, q, r$ )
```

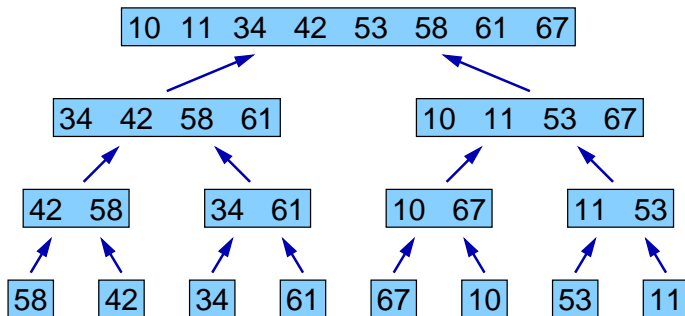
Pro setřídění pole  $A$ , které obsahuje prvky  $A[0], A[1], \dots, A[n-1]$ , zavoláme MERGE-SORT( $A, 0, n$ ).

**Poznámka:** Procedura MERGE( $A, p, q, r$ ) spojí setříděné posloupnosti uložené v  $A[p..q-1]$  a  $A[q..r-1]$  do jedné posloupnosti uložené v  $A[p..r-1]$ .



# Rekurentní vztahy

**Vstup:** 58, 42, 34, 61, 67, 10, 53, 11



Dobu výpočtu  $T(n)$  algoritmu **MERGE-SORT** pro vstup velikosti  $n$  můžeme vyjádřit jako:

$$T(n) = \begin{cases} \Theta(1) & \text{pro } n = 1 \\ 2T(n/2) + \Theta(n) & \text{pro } n > 1 \end{cases}$$

**Poznámka:**  $\Theta(1)$  označuje množinu všech funkcí jejichž hodnota je omezena konstantou.

**Poznámka:** Přesněji můžeme  $T(n)$  vyjádřit jako

$$T(n) = \begin{cases} \Theta(1) & \text{pro } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{pro } n > 1 \end{cases}$$

# Rekurentní vztahy

Pokud bychom znali hodnoty všech konstant a přesné hodnoty funkcí, mohli bychom spočítat přesnou hodnotu  $T(n)$  pro libovolné  $n$ .

## Příklad:

$$T(n) = \begin{cases} 4 & \text{pro } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + 5n + 11 & \text{pro } n > 1 \end{cases}$$

$$T(1) = 4$$

$$T(2) = T(\lceil 2/2 \rceil) + T(\lfloor 2/2 \rfloor) + 5 \cdot 2 + 11 = T(1) + T(1) + 21 = 29$$

$$T(3) = T(\lceil 3/2 \rceil) + T(\lfloor 3/2 \rfloor) + 5 \cdot 3 + 11 = T(2) + T(1) + 26 = 59$$

$$T(4) = T(\lceil 4/2 \rceil) + T(\lfloor 4/2 \rfloor) + 5 \cdot 4 + 11 = T(2) + T(2) + 31 = 89$$

$$T(5) = T(\lceil 5/2 \rceil) + T(\lfloor 5/2 \rfloor) + 5 \cdot 5 + 11 = T(3) + T(2) + 36 = 124$$

$$T(6) = T(\lceil 6/2 \rceil) + T(\lfloor 6/2 \rfloor) + 5 \cdot 6 + 11 = T(3) + T(3) + 41 = 159$$

$$T(7) = T(\lceil 7/2 \rceil) + T(\lfloor 7/2 \rfloor) + 5 \cdot 7 + 11 = T(4) + T(3) + 46 = 194$$

...

- Přesné hodnoty funkcí většinou neznáme, místo nich známe jen asymptotické odhady.
- V tom případě bude výsledkem také jen asymptotický odhad.
- I když všechny přesné hodnoty známe, může nám stačit jen asymptotický odhad, pokud by přesné odvození bylo příliš komplikované.

**Příklad:**  $T(n) = \begin{cases} \Theta(1) & \text{pro } n = 1 \\ 2T(\lfloor n/2 \rfloor) + n & \text{pro } n > 1 \end{cases}$

Uhodneme, že řešením je  $T(n) = O(n \lg n)$ , a dokážeme, že  $T(n) \leq cn \lg n$  pro nějakou vhodně zvolenou konstantu  $c$ :

$$\begin{aligned} T(n) &\leq 2(c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + n \\ &\leq cn \lg(n/2) + n \\ &= cn \lg n - cn \lg 2 + n \\ &= cn \lg n - cn + n \\ &\leq cn \lg n \end{aligned}$$

kde poslední nerovnost platí pro libovolné  $c \geq 1$ .

**Poznámka:**  $\lg n$  označuje  $\log_2 n$ .

Zbývá ověřit, že platí báze indukce.

Předpokládejme na chvíli pro jednoduchost, že  $T(1) = 1$ .

Všimněte si, že vztah  $T(n) \leq cn \lg n$  neplatí pro  $n = 1$ , ať už zvolíme jakoukoliv hodnotu  $c \geq 1$  (protože  $\lg 1 = 0$ ), a báze indukce tedy neplatí!

**Řešení:** Pokud chceme odvodit pouze asymptotický odhad, stačí ukázat, že  $T(n) \leq cn \lg n$  platí pro  $n \geq n_0$ , kde  $n_0$  je nějaká vhodně zvolená konstanta.

Zvolíme-li například  $n_0 = 2$ , stačí ukázat, že pro nějaké  $c$  platí pro libovolné  $n \geq 2$  vztah  $T(n) \leq cn \lg n$ :

- Báze:  $T(2) \leq c2 \lg 2$ ,  $T(3) \leq c3 \lg 3$   
Platí pro  $c = 2$ , neboť  $T(2) = 4$  a  $T(3) = 5$
- Indukční krok: pro  $n \geq 4$  hodnota  $T(n)$  nezávisí (přímo) na  $T(1)$ .

Při používání asymptotické notace je třeba dávat pozor, protože snadno můžeme udělat chybu při zanedbávání konstant.

**Příklad:**

$$T(n) = \begin{cases} \Theta(1) & \text{pro } n = 1 \\ 2T(\lfloor n/2 \rfloor) + n & \text{pro } n > 1 \end{cases}$$

Uhodneme, že  $T(n) = O(n)$  a tedy  $T(n) \leq cn$  pro nějakou konstantu  $c$ .

$$\begin{aligned} T(n) &\leq 2(c\lfloor n/2 \rfloor) + n \\ &\leq cn + n \\ &= O(n) \quad \leftarrow \text{Chyba!} \end{aligned}$$

Výše uvedené odvození je špatně, protože jsme nedokázali, že  $T(n) \leq cn$  platí pro tutéž konstantu  $c$ , která byla použita v indukčním předpokladu.

## Poznámka:

Nadále budeme předpokládat, že pro „malé“ hodnoty  $n$  je funkce  $T(n)$  omezena nějakou konstantou, tj. že existují nějaké konstanty  $c, d$  takové, že

$$n \leq d \quad \Rightarrow \quad T(n) \leq c$$

Tento předpoklad nebudeme nadále explicitně uvádět.

Pokud se například hodnota  $T(1)$  změní, výsledná hodnota  $T(n)$  se většinou změní nanejvýš o konstantu, ale asymptotický růst funkce  $T(n)$  se nezmění.



## Lemma

Nechť  $a_1, \dots, a_k, c$  jsou kladné konstanty takové, že  $a_1 + \dots + a_k < 1$  a pro funkci  $T : \mathbb{N} \rightarrow \mathbb{N}$  platí

$$T(n) \leq T(\lceil a_1 n \rceil) + T(\lceil a_2 n \rceil) + \dots + T(\lceil a_k n \rceil) + cn.$$

Pak  $T(n) = O(n)$ .

**Důkaz:** Zvolme  $\varepsilon > 0$  takové, že  $a_1 + \dots + a_k < 1 - 2\varepsilon$ . Pak pro nějaké dostatečně velké  $n_0$  platí pro všechna  $n \geq n_0$

$$\lceil a_1 n \rceil + \dots + \lceil a_k n \rceil \leq (1 - \varepsilon)n$$

Chceme ukázat, že pro nějaké vhodně zvolené  $d$  platí pro všechna  $n \geq 1$

$$T(n) \leq dn$$

- Báze: Pro  $n \leq n_0$  stačí, aby pro  $d$  platilo

$$d > \max\{T(n)/n \mid 1 \leq n \leq n_0\}$$

- Indukční krok:

$$\begin{aligned}T(n) &\leq T(\lceil a_1 n \rceil) + \dots + T(\lceil a_k n \rceil) + cn \\ &\leq d \cdot \lceil a_1 n \rceil + \dots + d \cdot \lceil a_k n \rceil + cn \\ &= d \cdot (\lceil a_1 n \rceil + \dots + \lceil a_k n \rceil) + cn \\ &\leq d \cdot (1 - \varepsilon)n + cn \\ &= dn - d\varepsilon n + cn \\ &= dn - (d\varepsilon - c)n \\ &\leq dn\end{aligned}$$

Poslední nerovnost platí, pokud  $d$  zvolíme tak, že  $d\varepsilon > c$ .

## Lemma

*Nechť  $k \geq 2$  a  $a_1, \dots, a_k, c$  jsou kladné konstanty takové, že  $a_1 + \dots + a_k = 1$  a pro funkci  $T : \mathbb{N} \rightarrow \mathbb{N}$  platí*

$$T(n) \leq T(\lceil a_1 n \rceil) + T(\lceil a_2 n \rceil) + \dots + T(\lceil a_k n \rceil) + cn.$$

*Pak  $T(n) = O(n \log n)$ .*

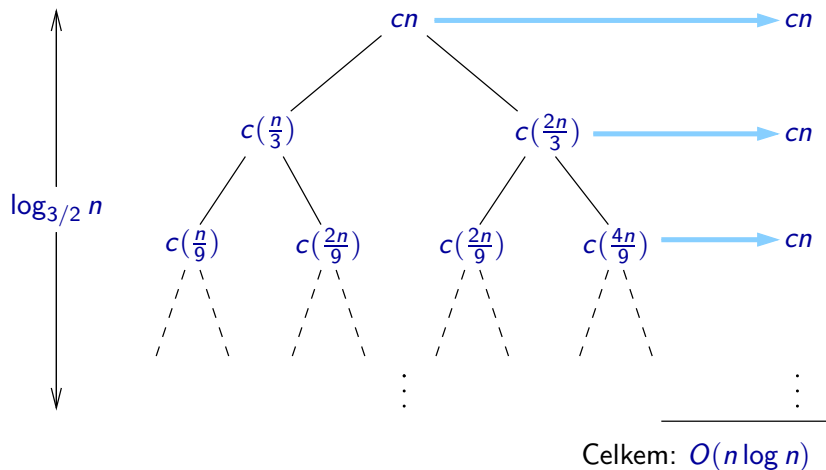
Důkaz byl podobný jako v předchozím případě (o něco komplikovanější).

Ideu budeme ilustrovat na příkladě

$$T(n) = T(n/3) + T(2n/3) + cn$$

Ukážeme, že  $T(n) = O(n \log n)$ . Pro jednoduchost zanedbáme zaokrouhlování.

# Rekurentní vztahy



## Věta (Master theorem)

Nechť  $a \geq 1$  a  $b > 1$  jsou konstanty,  $f : \mathbb{N} \rightarrow \mathbb{N}$  je funkce a funkce  $T : \mathbb{N} \rightarrow \mathbb{N}$  je definována vztahem

$$T(n) = aT(n/b) + f(n)$$

Pak platí:

- Je-li  $f(n) = O(n^c)$ , kde  $c < \log_b a$ , pak  $T(n) = \Theta(n^{\log_b a})$ .
- Je-li  $f(n) = \Theta(n^{\log_b a})$ , pak  $T(n) = \Theta(n^{\log_b a} \log n)$ .
- Je-li  $f(n) = \Theta(n^c)$ , kde  $c > \log_b a$ , pak  $T(n) = \Theta(n^c)$ .

**Poznámka:** Zápis  $n/b$  ve výše uvedené větě, může znamenat jak  $\lfloor n/b \rfloor$ , tak  $\lceil n/b \rceil$ .

- Ukazuje se, že různé (algoritmické) problémy jsou různě těžké.
- Obtížnější jsou ty problémy, k jejichž řešení potřebujeme více času a paměti.
- Obtížnost problémů chceme nějak posuzovat, a to jak
  - absolutně – kolik času a kolik paměti potřebujeme k jejich řešení, tak
  - relativně – o kolik je jejich řešení obtížnější nebo naopak jednodušší oproti jiným problémům.
- Proč se u některých problémů nedaří nalézt efektivní algoritmy?  
Může vůbec nějaký efektivní algoritmus pro daný problém existovat?
- Kde přesně jsou limity toho, co můžeme prakticky zvládnout?

Je potřeba odlišovat **složitost algoritmu** a **složitost problému**.

Pokud například zkoumáme časovou složitost v nejhorším případě, mohli bychom neformálně říct:

- **složitost algoritmu** – funkce, která vyjadřuje, kolik kroků maximálně udělá daný algoritmus pro vstup velikosti  $n$
- **složitost problému** – jaká je časová složitost „nejoptimálnějšího“ algoritmu, který řeší daný problém

Zavedení pojmu „složitost problému“ ve výše uvedeném smyslu naráží na značné technické obtíže. Pojem „složitost problému“ se tedy jako takový nedefinuje, ale obchází se zavedením tzv. **tříd složitosti**.

Pokud pak hovoříme o složitosti problému, máme tím na mysli to, do kterých tříd složitosti daný problém patří resp. nepatří.

Třídy složitosti jsou podmnožiny množiny všech (algoritmických) **problémů**.

Daná konkrétní třída složitosti je vždy charakterizována nějakou vlastností, kterou mají problémy do ní patřící.

Typickým příkladem takové vlastnosti je vlastnost, že pro daný problém existuje nějaký algoritmus s určitým omezením (např. časové nebo prostorové složitosti):

- Do dané třídy pak patří všechny problémy, pro které takovýto algoritmus existuje.
- Naopak do ní nepatří problémy, pro které žádný takový algoritmus neexistuje.



## Definice

Pro libovolnou funkci  $f : \mathbb{N} \rightarrow \mathbb{N}$  definujeme třídu  $\mathcal{T}(f(n))$  jako třídu obsahující právě ty problémy, pro něž existuje algoritmus s časovou složitostí  $O(f(n))$ .

## Příklad:

- $\mathcal{T}(n)$  – třída všech problémů pro něž existuje algoritmus s časovou složitostí  $O(n)$
- $\mathcal{T}(n^2)$  – třída všech problémů pro něž existuje algoritmus s časovou složitostí  $O(n^2)$
- $\mathcal{T}(n \log n)$  – třída všech problémů pro něž existuje algoritmus s časovou složitostí  $O(n \log n)$

## Definice

Pro libovolnou funkci  $f : \mathbb{N} \rightarrow \mathbb{N}$  definujeme třídu  $\mathcal{S}(f(n))$  jako třídu obsahující právě ty problémy, pro něž existuje algoritmus s prostorovou složitostí  $O(f(n))$ .

## Příklad:

- $\mathcal{S}(n)$  – třída všech problémů pro něž existuje algoritmus s prostorovou složitostí  $O(n)$
- $\mathcal{S}(n^2)$  – třída všech problémů pro něž existuje algoritmus s prostorovou složitostí  $O(n^2)$
- $\mathcal{S}(n \log n)$  – třída všech problémů pro něž existuje algoritmus s prostorovou složitostí  $O(n \log n)$

## Poznámka:

Všimněte si, že u tříd  $\mathcal{T}(f)$  a  $\mathcal{S}(f)$  může to, které problémy do dané třídy patří, záviset na použitém výpočetním modelu (zda je to stroj RAM, jednopáskový Turingův stroj, více páskový Turingův stroj, ...).

Pomocí tříd  $\mathcal{T}(f(n))$  a  $\mathcal{S}(f(n))$  můžeme definovat třídy **PTIME** a **PSPACE** jako

$$\text{PTIME} = \bigcup_{k \geq 0} \mathcal{T}(n^k)$$

$$\text{PSPACE} = \bigcup_{k \geq 0} \mathcal{S}(n^k)$$

- **PTIME** je třída všech problémů, pro které existuje algoritmus s polynomiální časovou složitostí, tj. s časovou složitostí  $O(n^k)$ , kde  $k$  je nějaká konstanta.
- **PSPACE** je třída všech problémů, pro které existuje algoritmus s polynomiální prostorovou složitostí, tj. s prostorovou složitostí  $O(n^k)$ , kde  $k$  je nějaká konstanta.

**Poznámka:** Vzhledem k tomu, že všechny (rozumné) výpočetní modely jsou schopné se navzájem simulovat tak, že při dané simulaci nevzroste počet kroků ani množství použité paměti víc než polynomiálně, není definice tříd **PTIME** a **PSPACE** závislá na použitém výpočetním modelu. Pro jejich zadefinování můžeme použít kterýkoliv výpočetní model.

Říkáme, že tyto třídy jsou **robustní** – jejich definice nezávisí na použitém výpočetním modelu.

Analogicky můžeme zavést další třídy:

**EXPTIME** – množina všech problémů, pro které existuje algoritmus s časovou složitostí  $2^{O(n^k)}$ , kde  $k$  je nějaká konstanta

**EXPSPACE** – množina všech problémů, pro které existuje algoritmus s prostorovou složitostí  $2^{O(n^k)}$ , kde  $k$  je nějaká konstanta

**LOGSPACE** – množina všech problémů, pro které existuje algoritmus s prostorovou složitostí  $O(\log n)$

**Poznámka:** Místo  $2^{O(n^k)}$  bychom mohli psát také  $O(c^{n^k})$ , kde  $c$  a  $k$  jsou nějaké konstanty.

Při definici třídy **LOGSPACE** musíme přesněji specifikovat, co považujeme za prostorovou složitost algoritmu.

Uvažujeme například Turingův stroj, který pracuje se třemi páskami:

- **Vstupní páskou**, na které je na začátku výpočtu zapsán vstup. Z této pásky je možno pouze číst.
- **Pracovní páskou**, která je na začátku výpočtu prázdná. Z této pásky je možno číst i na ni zapisovat.
- **Výstupní páskou**, která je také na začátku výpočtu prázdná a na kterou je možno pouze zapisovat.

Množství použité paměti je pak definováno, jako počet použitých políček na pracovní pásce.

Další příklady tříd složitosti:

**2-EXPTIME** – množina všech problémů, pro které existuje algoritmus s časovou složitostí  $2^{2^{O(n^k)}}$ , kde  $k$  je nějaká konstanta

**2-EXPSPACE** – množina všech problémů, pro které existuje algoritmus s prostorovou složitostí  $2^{2^{O(n^k)}}$ , kde  $k$  je nějaká konstanta

**ELEMENTARY** – množina všech problémů, pro které existuje algoritmus s časovou (či prostorovou) složitostí

$$2^{2^{2^{\dots^{2^{2^{O(n^k)}}}}}}$$

kde  $k$  je konstanta a počet exponentů je omezen konstantou.



# Vztahy mezi třídami složitosti

Pokud Turingův stroj provede  $m$  kroků, tak použije maximálně  $m$  políček na pásce.

Pokud tedy existuje pro nějaký problém algoritmus s časovou složitostí  $O(f(n))$ , má tento algoritmus paměťovou složitost (nejvýše)  $O(f(n))$ .

Je tedy zřejmé, že platí následující vztah.

## Pozorování

Pro libovolnou funkci  $f : \mathbb{N} \rightarrow \mathbb{N}$  platí  $\mathcal{T}(f(n)) \subseteq \mathcal{S}(f(n))$ .

**Poznámka:** Analogicky bychom mohli argumentovat například pro stroj RAM.

# Vztahy mezi třídami složitosti

Z předchozího okamžitě plyne:

$$\begin{aligned} \text{PTIME} &\subseteq \text{PSPACE} \\ \text{EXPTIME} &\subseteq \text{EXPSPACE} \\ \text{2-EXPTIME} &\subseteq \text{2-EXPSPACE} \\ &\vdots \end{aligned}$$

Vzhledem k tomu, že polynomiální funkce rostou pomaleji než exponenciální, zjevně platí:

$$\text{PTIME} \subseteq \text{EXPTIME} \subseteq \text{2-EXPTIME} \subseteq \dots$$

$$\text{LOGSPACE} \subseteq \text{PSPACE} \subseteq \text{EXPSPACE} \subseteq \text{2-EXPSPACE} \subseteq \dots$$

## Věta

Pro libovolnou prostorově zkonstruovatelnou funkci  $f : \mathbb{N} \rightarrow \mathbb{N}$  existuje problém, který je možné řešit algoritmem s prostorovou složitostí  $O(f(n))$ , ale není možné řešit žádným algoritmem s prostorovou složitostí  $o(f(n))$ .

Důkaz tohoto tvrzení je netriviální a značně přesahuje rozsah tohoto předmětu.

**Poznámka:** Funkce  $f : \mathbb{N} \rightarrow \mathbb{N}$ , kde  $f(n) \geq \log n$ , se nazývá prostorově zkonstruovatelná, jestliže existuje algoritmus s prostorovou složitostí  $O(f(n))$ , který pro vstup  $w$ , kde  $|w| = n$ , vytvoří binární reprezentaci hodnoty  $f(n)$ .

Prakticky všechny „rozumné“ funkce mají tuto vlastnost.

## Důsledek

Pro libovolné dvě funkce  $f_1, f_2 : \mathbb{N} \rightarrow \mathbb{N}$  takové, že  $f_1(n)$  je v  $o(f_2(n))$  a  $f_2(n)$  je prostorově zkonstruovatelná, platí, že  $\mathcal{S}(f_1(n)) \subsetneq \mathcal{S}(f_2(n))$ .

Z tohoto tvrzení okamžitě plynou následující důsledky:

- Pro libovolná dvě reálná čísla  $0 \leq \epsilon_1 < \epsilon_2$  platí

$$\mathcal{S}(n^{\epsilon_1}) \subsetneq \mathcal{S}(n^{\epsilon_2})$$

- $\text{LOGSPACE} \subsetneq \text{PSPACE}$
- $\text{PSPACE} \subsetneq \text{EXPSPACE}$
- $\text{EXPSPACE} \subsetneq \text{2-EXPSPACE}$

## Věta

Pro libovolnou časově zkonstruovatelnou funkci  $t : \mathbb{N} \rightarrow \mathbb{N}$  existuje problém, který je možné řešit algoritmem s časovou složitostí  $O(t(n))$ , ale není možné řešit žádným algoritmem s časovou složitostí  $o(t(n)/\log t(n))$ .

**Poznámka:** Funkce  $t : \mathbb{N} \rightarrow \mathbb{N}$ , kde  $t(n) \geq \log n$ , se nazývá časově zkonstruovatelná, jestliže existuje algoritmus s časovou složitostí  $O(t(n))$ , který pro vstup  $w$ , kde  $|w| = n$ , vytvoří binární reprezentaci hodnoty  $t(n)$ .

## Důsledek

Pro libovolné dvě funkce  $t_1, t_2 : \mathbb{N} \rightarrow \mathbb{N}$  takové, že  $t_1(n)$  je v  $o(t_2(n)/\log t_2(n))$  a  $t_2(n)$  je časově zkonstruovatelná, platí, že  $\mathcal{T}(f_1(n)) \subsetneq \mathcal{T}(f_2(n))$ .

Z předchozího plynou následující důsledky:

- Pro libovolná dvě reálná čísla  $0 \leq \epsilon_1 < \epsilon_2$  platí

$$\mathcal{T}(n^{\epsilon_1}) \subsetneq \mathcal{T}(n^{\epsilon_2})$$

- $\text{PTIME} \subsetneq \text{EXPTIME}$
- $\text{EXPTIME} \subsetneq 2\text{-EXPTIME}$

Při zkoumání vztahů mezi třídami složitosti se ukazuje jako užitečný pojem **konfigurace**.

Konfigurací budeme rozumět celkový stav, ve kterém se během jednoho kroku nachází stroj, provádějící nějaký daný algoritmus.

- U Turingova stroje je konfigurace dána stavem jeho řídicí jednotky, obsahem pásky (resp. pásek) a pozicí hlavy (resp. hlav).
- U stroje RAM je konfigurace dána obsahem paměti, obsahem všech registrů (včetně IP) a pozicemi čtecí a zapisovací hlavy.

# Vztahy mezi třídami složitosti

Mělo by být jasné, že konfigurace (resp. jejich popisy) můžeme zapisovat jako slova v nějaké abecedě.

Navíc můžeme konfigurace zapisovat tak, že délka těchto slov bude zhruba stejná jako množství paměti použité algoritmem (tj. počet políček na pásce použitých Turingovým strojem, počet bitů paměti použitých strojem RAM apod.).

**Poznámka:** Pokud máme abecedu  $\Sigma$ , kde  $|\Sigma| = c$  tak:

- Počet slov délky  $n$  je  $c^n$ , tj.  $2^{\Theta(n)}$ .
- Počet slov délky nejvýše  $n$  je

$$\sum_{i=0}^n c^i = \frac{c^{n+1} - 1}{c - 1}$$

tj. také  $2^{\Theta(n)}$ .



# Vztahy mezi třídami složitosti

Je jasné, že během výpočtu korektního algoritmu se žádná konfigurace nemůže zopakovat, protože jinak by se algoritmus zacyklil a běžel by donekonečna.

Pokud tedy víme, že paměťová složitost nějakého algoritmu je  $O(f(n))$ , znamená to, že počet různých konfigurací dosažitelných během výpočtu je  $2^{O(f(n))}$ .

Protože se konfigurace během žádného výpočtu neopakují, je i časová složitost daného algoritmu maximálně  $2^{O(f(n))}$ .

## Pozorování

Pro libovolnou funkci  $f : \mathbb{N} \rightarrow \mathbb{N}$  platí, že  $\mathcal{S}(f(n)) \subseteq \mathcal{T}(2^{f(n)})$ .

Z předchozího plynou následující důsledky:

$$\text{LOGSPACE} \subseteq \text{PTIME}$$

$$\text{PSPACE} \subseteq \text{EXPTIME}$$

$$\text{EXSPACE} \subseteq 2\text{-EXPTIME}$$

⋮

Shrnutí:

$$\text{LOGSPACE} \subseteq \text{PTIME} \subseteq \text{PSPACE} \subseteq \text{EXPTIME} \subseteq \text{EXPSPACE} \subseteq \\ \subseteq \text{2-EXPTIME} \subseteq \text{2-EXPSPACE} \subseteq \dots \subseteq \text{ELEMENTARY}$$

Navíc je známo, že:

- $\text{PTIME} \subsetneq \text{EXPTIME} \subsetneq \text{2-EXPTIME} \subsetneq \dots$
- $\text{LOGSPACE} \subsetneq \text{PSPACE} \subsetneq \text{EXPSPACE} \subsetneq \text{2-EXPSPACE} \subsetneq \dots$

**Horním odhadem** složitosti problému rozumíme to, že složitost problému není vyšší než nějaká uvedená.

Většinou je to formulováno tak, že daný problém patří do nějaké určité třídy složitosti.

Příklady tvrzení, které se týkají horních odhadů složitosti:

- Problém nalezení nejkratší cesty v grafu je v **P**TIME.
- Problém ekvivalence dvou regulárních výrazů je v **EXPSPACE**.

Pokud chceme zjistit nějaký horní odhad složitosti problému, stačí ukázat, že existuje algoritmus s danou složitostí.

**Dolním odhadem** složitosti problému rozumíme to, že složitost problému je alespoň taková jako nějaká uvedená.

Většinou je to formulováno tak, že daný problém nepatří do nějaké určité třídy složitosti.

Obecně je zjišťování (netriviálních) dolních odhadů složitosti problémů mnohem obtížnější než zjišťování horních odhadů.

Pro odvození dolního odhadu musíme totiž ukázat, že neexistuje žádný algoritmus, který by řešil daný problém a přitom měl danou složitost.

## Problém „Třídění“

**Vstup:** Posloupnost prvků  $a_1, a_2, \dots, a_n$ .

**Výstup:** Prvky  $a_1, a_2, \dots, a_n$  seříděné od nejmenšího po největší.

Ukážeme, že každý algoritmus, který řeší problém třídění a na prvcích tříděné posloupnosti používá pouze operaci porovnávání (tj. nezkoumá obsah těchto prvků), má složitost  $\Omega(n \log n)$ .

# Dolní odhad pro problém třídění

Jestliže algoritmus nezkoumá obsah jednotlivých prvků jinak než jejich porovnáváním, je zřejmé, že pro činnost algoritmu je důležité pouze relativní pořadí prvků na vstupu a nikoliv jejich konkrétní hodnoty.

Můžeme se tedy omezit na případ, kdy vstupem jsou pouze permutace množiny  $\{1, 2, \dots, n\}$ .

Budeme sledovat během výpočtu hodnotu registru IP stroje RAM, tj. která instrukce se v kterém kroku provádí.

Pokud předložíme stroji RAM dvě různé permutace, je jasné, že od nějakého okamžiku se budou hodnoty registru IP lišit (v důsledku porovnání dvou prvků a následného provedení nebo naopak neprovedení podmíněného skoku), protože pokud by se v obou výpočtech prováděly přesně stejné operace, v jednom z nich bychom dostali chybný výstup.

# Dolní odhad pro problém třídění

Všechny možné výpočty nad vstupy velikosti  $n$  si můžeme představit jako strom, kde jednotlivé větve odpovídají jednotlivým výpočtům.

- Tento strom musí mít nejméně  $n!$  listů, protože existuje  $n!$  různých permutací  $n$  prvků.
- Výška tohoto stromu  $t$  (tj. délka jeho nejdelší větve) odpovídá časové složitosti algoritmu.
- Předpokládáme-li, že každý vrchol má nanejvýš  $c$  potomků, pak počet listů stromu je nanejvýš  $c^t$ .

Z toho plyne, že

$$c^t \geq n!$$

neboli

$$t \geq \log_c n!$$



Vzhledem k tomu, že  $\log_c n! = \Theta(n \log n)$ , dostáváme požadovaný výsledek.

**Poznámka:** K dokázání toho, že  $\log_c n! = \Theta(n \log n)$ , stačí využít faktu, že  $n! \leq n^n \leq (n!)^2$ , z čehož plyne

$$\log n! \leq \log n^n \leq \log(n!)^2$$

$$\log n! \leq n \log n \leq 2 \log n!$$

$$\frac{1}{2} n \log n \leq \log n! \leq n \log n$$

## Definice

Rozhodovací problém je takový, kde je množina možných výstupů dvouprvková  $\{ANO, NE\}$ .

- V této a následující přednášce se budeme zabývat výhradně rozhodovacími problémy

# Nedeterministický Turingův stroj

Podobný princip jako byl použit pro nedeterministické konečné (resp. zásobníkové) automaty můžeme použít i na Turingovy stroje.

## Definice

**Nedeterministický Turingův stroj** je definován jako šestice

$\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{ANO}}, q_{\text{NE}})$  kde:

- $Q$  je konečná množina **stavů**
- $\Gamma$  je konečná množina **páskových symbolů**
- $\Sigma \subseteq \Gamma, \Sigma \neq \emptyset$  je konečná množina **vstupních symbolů**
- $\delta : (Q - F) \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma \times \{-1, 0, +1\})$  je **přechodová funkce**
- $q_0 \in Q$  je **počáteční stav**
- $q_{\text{ANO}}, q_{\text{NE}} \in Q$  jsou **koncové stavy** (přijímající a nepřijímající)

# Nedeterministický Turingův stroj

- Požadujeme, aby každý výpočet skončil buď ve stavu  $q_{ANO}$  nebo ve stavu  $q_{NE}$ .
- Odpověď nedeterministického TS na vstup  $w$  je ANO, jestliže alespoň jeden z možných výpočtů skončí v  $q_{ANO}$ .
- Složitost takového nedeterministického stroje můžeme definovat jako počet kroků nejdelšího možného výpočtu stroje nad daným vstupem

# Nedeterministický stroj RAM

- Je definován stejně jako deterministický RAM
- Navíc je jen jedna instrukce `CHOOSE GOTO x OR GOTO y`, která umožňuje stroji vybrat si jedno z možných pokračování
- Pokud by ze všech možných výpočtů takového stroje nad zadaným vstupem alespoň jeden skončil s odpovědí `ANO`, je odpověď `ANO` (bez ohledu na další možné výpočty)
- Pokud všechny možné výpočty končí odpovědí `NE`, je odpověď `NE`
- Složitost nedeterministického stroje RAM definujeme jako délku nejdelšího možného výpočtu nad zadaným vstupem
- Podobně můžeme definovat nedeterministické verze jiných výpočetních modelů

Na nedeterminismus můžeme nahlížet dvěma způsoby:

- 1 Ve chvíli, kdy dochází k nejednoznačnému pokračování, se stroj rozhodne pro tu nejlepší vedoucí nejrychleji k odpovědi ANO (pokud taková možnost existuje). Odpověď je ANO, právě když tento jediný výpočet skončí odpovědí ANO.
  - 2 Ve chvíli, kdy dochází k nejednoznačnosti se stroj rozdvojí (případně rozdělí na více identických kopií) a každá kopie pokračuje jedním možným pokračováním. Odpověď je ANO právě tehdy, když alespoň jedna z kopií stroje odpoví ANO.
- Oba přístupy jsou ekvivalentní.
  - Možnost 1. uhádne to nejlepší pokračování zatímco možnost 2. jej najde mezi všemi možnými. Pokud je odpověď ne, tak ani při výběru nejlepší varianty ani prozkoumání všech možných variant nikdy nedostaneme odpověď ANO.

- Z hlediska rozhodnutelnosti nemá nedeterminismus význam. Pokud nějaký problém řeší nedeterministický RAM nebo TS, tak ho dokáže řešit i deterministický, který postupně vyzkouší všechny možné výpočty.
- Význam je při rozdělování problémů do složitostních tříd, kdy nám díky nedeterminismu vznikají třídy mezi těmi deterministickými.

## Definice

Pro funkci  $f : \mathbb{N} \rightarrow \mathbb{N}$  rozumíme **třídou časové složitosti**  $\mathcal{NT}(f)$  množinu těch problémů, které jsou řešeny nedeterministickými RAMy s časovou složitostí v  $O(f(n))$ .

## Definice

Pro funkci  $f : \mathbb{N} \rightarrow \mathbb{N}$  rozumíme **třídou prostorové složitosti**  $\mathcal{NS}(f)$  množinu těch problémů, které jsou řešeny nedeterministickými RAMy s prostorovou složitostí v  $O(f(n))$ .



## Definice

$$\text{NPTIME} = \bigcup_{k=0}^{\infty} \mathcal{NT}(n^k)$$

- Existují rozhodovací problémy, pro které známe exponenciální algoritmus (s polynomiální prostorovou složitostí), ale nevíme, jestli existuje polynomiální algoritmus.
- Mnoho takových problémů patří právě do třídy NPTIME
- Každý problém patřící do PTIME patří i do NPTIME– algoritmus řešící problém z PTIME můžeme prohlásit za nedeterministický (který má 0 instrukcí CHOOSE) a složitost takového nedeterministického algoritmu je stejná jako původního deterministického, protože nejdelší větev výpočtu je ta jediná možná, která určovala i složitost deterministického.

## Problém „Barvení grafu $k$ barvami“

**Vstup:** Neorientovaný graf  $G$  a přirozené číslo  $k$

**Výstup:** ANO, pokud lze vrcholy grafu obarvit  $k$  barvami tak, aby žádné dva vrcholy spojené hranou neměly stejnou barvu. NE pokud to takto obarvit nelze.

- RAM prochází postupně všechny vrcholy a u každého má volbu z  $k$  barev. Po přiřazení barvy zkontroluje sousední vrcholy, jestli již některý nemá přiřazenu stejnou barvu.
- Pokud má být odpověď ANO, tak existuje správné obarvení. Jestliže se při každé volbě mezi více barvami stroj rozdělí na několik kopií, tak ta kopie, kde se přiřadily barvy odpovídající správnému řešení odpoví ANO a tedy i celý nedeterministický RAM.
- Pokud má být odpověď NE, tak žádné možné přiřazení nesplňuje požadavky na barvení a všechny kopie RAMu musí odpovědět NE, tedy i nedeterministický RAM odpoví NE.

## Problém „Barvení grafu $k$ barvami“

**Vstup:** Neorientovaný graf  $G$  a přirozené číslo  $k$

**Výstup:** ANO, pokud lze vrcholy grafu obarvit  $k$  barvami tak, aby žádné dva vrcholy spojené hranou neměly stejnou barvu. NE pokud to takto obarvit nelze.

- Počet kroků kterékoliv kopie algoritmu odpovídá počtu vrcholů a hran v grafu. Složitost tohoto algoritmu je tedy polynomiální.
- Problém barvení grafu tedy patří do třídy NPTIME.
- Nejlepší známý deterministický algoritmus pracuje v exponenciálním čase a polynomiálním prostoru.

- Podobně můžeme definovat třídu **NPSPACE**.
- Obdobně jako v deterministickém případě zřejmě platí  **$NPTIME \subseteq NPSPACE$** .
- Již dříve jsme zdůvodnili  **$PTIME \subseteq NPTIME$** .
- Nedeterministický RAM můžeme simulovat deterministickým, který zkusí všechny možné výpočty. Každý z možných výpočtů použije maximálně polynomiálně mnoho paměti a jednotlivé výpočty si nic nepředávají, takže můžeme používat pro všechny stejnou oblast paměti. Platí tedy  **$NPTIME \subseteq PSPACE$** .
- Savitch ukázal, že pokud je problém rozhodován nedeterministickým Turingovým strojem s prostorovou složitostí  $f(n)$ , tak je rozhodován i nějakým deterministickým Turingovým strojem s prostorovou složitostí  $f^2(n)$ . Z toho plyne  **$NPSPACE \subseteq PSPACE$** .

$$PTIME \subseteq NPTIME \subseteq PSPACE = NPSPACE$$

# Třída NPTIME

Třída NPTIME je nejdůležitější z tříd definovaných s pomocí nedeterminismu, protože obsahuje mnoho praktických problémů. Tuto třídu můžeme alternativně definovat následujícím způsobem.

## Definice

NPTIME je třídou všech rozhodovacích problémů  $\mathcal{P}$  se vstupem  $x$  takových, že existuje algoritmus  $\mathcal{A}$  splňující:

- Složitost  $\mathcal{A}$  je v  $O(p(|x|))$ .
  - Vstupem  $\mathcal{A}$  je  $(x, y)$ , kde  $|y| \in O(p(|x|))$ .
  - Odpověď  $\mathcal{P}$  na  $x$  je ANO právě tehdy, když existuje  $y$  takové, že  $\mathcal{A}$  vrací ANO pro vstup  $(x, y)$ .
- 
- $y$  je jakási “nápopvěda”, která se často nazývá **svědek**.
  - Není důležité, jak je těžké svědka najít. Jen když je znám, musí běžet algoritmus v polynomiálním čase.

## Problém „Isomorfismus grafů“

**Vstup:** Neorientované grafy  $G_1 = (V_1, E_1)$ ,  $G_2 = (V_2, E_2)$

**Výstup:** ANO, jestliže jsou grafy isomorfní, tedy existuje-li zobrazení  $\rho : V_1 \rightarrow V_2$  takové, že  $(x, y) \in E_1 \Leftrightarrow (\rho(x), \rho(y)) \in E_2$ . NE pokud takové zobrazení neexistuje.

Svědkiem je v tomto případě zobrazení  $\rho$ . Algoritmus pouze ověří, jestli hrany mezi vrcholy v  $G_1$  odpovídají příslušným hranám v  $G_2$ .

## Problém „loupežníka“

**Vstup:** Množina přirozených čísel  $A = \{a_1, a_2, \dots, a_n\}$

**Výstup:** ANO, jestliže je možné najít množinu  $X$  tak, že  $X \subset A$ ,  
$$\sum_{a_i \in X} a_i = \sum_{a_j \in A-X} a_j.$$

Svědkiem je v tomto případě množina  $X$  a algoritmus kontroluje, že je podmnožinou  $A$  a součty se rovnají.

- Obě definice třídy NPTIME (deterministický algoritmus s využitím svědků a nedeterministický algoritmus) jsou ekvivalentní
- Převod z nedeterministického algoritmu na deterministický se svědky
  - Pokud máme nedeterministický algoritmus, tak se může rozhodovat mezi více variantami a vybere vždy tu vedoucí nejrychleji k řešení.
  - Rozhoduje se maximálně polynomiálně často (více kroků ani neudělá) a tak můžeme jako svědka zvolit posloupnost voleb algoritmu.
  - Potom nám již stačí deterministický algoritmus, který díky svědkovi vždy ví, jak pokračovat.
- Převod z deterministického algoritmu se svědky na nedeterministický
  - Nedeterministický algoritmus si na začátku může nedeterministicky nagenarovat svědka (v polynomiálním čase)
  - Dále již nedeterministický algoritmus pracuje s vygenerovaným svědkem stejně, jako deterministický



## Definice

Problém  $P_1$  je **polynomiálně převeditelný** na problém  $P_2$ , jestliže existuje algoritmus s polynomiální časovou složitostí, který pro libovolný vstup  $w$  problému  $P_1$  sestrojí vstup  $w'$  problému  $P_2$ , přičemž platí, že odpověď na otázku problému  $P_1$  pro vstup  $w$  je stejná jako odpověď na otázku problému  $P_2$  pro vstup  $w'$ .

- Pokud převedeme polynomiálním algoritmem  $w$  na  $w'$ , kde  $|w| = n$  a  $|w'| = m$ , tak  $m \in O(n^k)$ . Pokud je  $P_2 \in \text{PTIME}$ , tak jeho složitost je v  $O(m^l)$ . Pokud převedeme  $w$  na  $w'$  a na to spustíme algoritmus pro  $P_2$  vyřešíme v  $O((n^k)^l) = O(n^{kl}) = O(n^r)$  problém  $P_1$ . Tedy  $P_1 \in \text{PTIME}$ .
- Podobně, převedeme-li  $P_1$  na  $P_2$  a  $P_2 \in \text{NPTIME}$ , tak i  $P_1 \in \text{NPTIME}$

S několika příklady převodu jste se setkali v předmětu Diskrétní matematika.

## Tok v síti

**Vstup:** Orientovaný ohodnocený graf s vyznačenou dvojicí vrcholů zdroj a stok a konstanta  $l$

**Výstup:** ANO, pokud existuje v grafu tok o velikosti  $l$ , NE jinak

## Párování v bipartitním grafu

**Vstup:** Neorientovaný bipartitní graf a konstanta  $k$

**Výstup:** ANO, pokud existuje párování o velikosti  $k$  ( $k$  hran takových, že žádné dvě z nich nemají stejný koncový vrchol)

- Je znám polynomiální algoritmus řešící problém toku v síti
- Problém párování dokážeme převést na problém toku v síti v polynomiálním čase (dokonce lineárním) a tedy složením dvou algoritmů (převodu a řešení problému toku) dostaneme algoritmus řešící problém párování v polynomiálním čase.
- Při převodu si označíme partity grafu, který je vstupem pro párování, jako  $V_1$  a  $V_2$ . Přidáme nové dva vrcholy  $z$  a  $s$ . Přidáme orientované hrany ze  $z$  do každého vrcholu z  $V_1$ . Hrany mezi vrcholy z  $V_1$  a  $V_2$  budou mít orientaci ve směru z  $V_1$  do  $V_2$ . Z každého vrcholu z  $V_2$  přidáme orientovanou hranu do  $s$ .

## Definice

Problém  $Q$  je **NP-těžký**, pokud každý problém ve třídě NP (neboli NPTIME) lze na  $Q$  převést polynomiálním převodem.

## Definice

Problém  $Q$  nazveme **NP-úplným**, pokud je NP-těžký a současně patří do třídy NP.

- Pokud bychom znali polynomiální algoritmus pro kterýkoliv NP-úplný problém  $P$ , tak dokážeme polynomiálně řešit všechny.
- Každý NP-úplný problém  $P'$  totiž můžeme převést na  $P$  a složením dvou polynomiálních algoritmů (převodu a řešení  $P$ ) dostaneme zase polynomiální algoritmus řešící  $P'$ .

## Věta

Nechť problém  $Q$  je NP-těžký. Pokud existuje polynomiální převod problému  $Q$  na nějaký problém  $P$ , pak také  $P$  je NP-těžký.

**Důkaz:** Protože je  $Q$  NP-těžký, je možné na něj převést všechny problémy z třídy NP. Spojení algoritmů převodu na  $Q$  a z  $Q$  na  $P$  dostaneme pro jakýkoliv problém z NP algoritmus převodu na  $P$ . Protože tedy jakýkoliv problém z NP umíme převést na  $P$ , je  $P$  NP-těžký.

- Dokazovat NP-těžkost hledáním převodů všech problémů z NP není možné
- Na základě této věty nám stačí ukázat převod z jednoho problému, o kterém již dříve bylo dokázáno, že je NP-těžký.

- NP-těžké problémy jsou vlastně nejtěžší problémy ze třídy NP
- Navržení polynomiálního algoritmu pro jeden NP-úplný problém znamená, že známe polynomiální algoritmus pro každý problém v NP
- Navržení polynomiálního algoritmu pro problém z NP, který není NP-těžký, neznámá nic pro složitost NP-těžkých problémů
- Otázka jestli  $P = NP$  je jednou z neznámějších dlouhodobě otevřených otázek teoretické informatiky
- Předpokládá se  $P \neq NP$ , tedy že NP-úplné problémy nepatří mezi efektivně řešitelné
- Otázka ale zní, jestli vůbec nějaký NP-úplný problém existuje. Jak ukázat, že všechny problémy z NP jdou na něj převést, když jich je nekonečně mnoho?

## SAT - Splnitelnost logických formulí

**Vstup:** Logická formule  $\Phi$  v konjunktivní normální formě

**Výstup:** ANO, pokud existuje ohodnocení  $\mu$  proměnných takové, aby pravdivostní hodnota formule  $\mu(\Phi) = 1$

**Příklad:** Mějme formuli

$$\Phi = (a_1 \vee \neg a_2 \vee a_4) \wedge (\neg a_1 \vee a_3) \wedge (\neg a_1 \vee \neg a_3 \vee \neg a_4)$$

**Řešení:** Pokud ohodnotíme např.  $\mu(a_1) = \mu(a_3) = 1$ ,  $\mu(a_2) = \mu(a_4) = 0$  dostaneme  $\mu(\Phi) = 1$ , tedy odpověď je, že formule je splnitelná.

- $a_1, a_2, a_3, a_4$  nazýváme **atomické proměnné**.
- Proměnné i jejich negace jsou **literály**.
- Disjunkce, např.  $(a_1 \vee \neg a_2 \vee a_4)$ , nazýváme **klauzule**.
- Aby byla formule splněna, musí ohodnocení přiřazovat **1** alespoň jednomu literálu v každé klauzuli.

## Věta (Cook)

Existuje NP-úplný problém.

- V důkazu se ukáže, že SAT je NP-úplný
- Příslušnost SATu do NP je zřejmá. Nedeterministický algoritmus hádá ohodnocení proměnných a jen ověří pravdivostní hodnotu formule (popř. využijeme svědka v podobě ohodnocení proměnných a algoritmus jen kontroluje, jestli je ohodnocení správné).
- Přesný důkaz NP-obtížnosti není úplně snadný, musí se ošetřit mnoho drobností



- Každý problém z NP má být převoditelný na SAT
- Pro jeden konkrétní problém, pro který chceme převoditelnost zrovna ukázat, existuje nějaký nedeterministický Turingův stroj, který jej rozhoduje
- Ukážeme konstrukci, která pro tento Turingův stroj  $\mathcal{M}$  a vstupní slovo  $w$  sestrojí formuli  $\phi_{\mathcal{M},w}$
- Formule  $\phi_{\mathcal{M},w}$  bude splnitelná právě tehdy, když TS přijme slovo  $w$
- Následující konstrukce funguje pro všechny Turingovy stroje, tedy bychom mohli postupně ukázat převod všech problémů z NP na SAT (což nemůžeme udělat explicitně, protože jich je nekonečně mnoho)

# Cookova věta

Výpočet Turingova stroje  $\mathcal{M}$  na vstupním slově  $w$  ( $|w| = n$ ) můžeme reprezentovat tabulkou

	0	1	2	...	$w_n$	# ... #	$n^k$
0	#	$(q_0, w_1)$	$w_2$	...	$w_n$	# ... #	#
$i$	#	$\alpha_1$	$\alpha_2$	...	$\alpha_n$	$\alpha_{n+1} \dots$	#
$n^k$	#	$\beta_1$	$\beta_2$	...	$\beta_n$	$\beta_{n+1} \dots$	#

- Zavedeme proměnné  $x_{i,j,s}$  pro každé  $0 \leq i \leq n^k$ ,  $0 \leq j \leq n^k$ ,  $s \in \Gamma$ . Hodnota **1** takové proměnné znamená, že v  $i$ -tém řádku a  $j$ -tém sloupci se nachází symbol  $s$ , tedy stroj má v  $i$ -té konfiguraci na  $j$ -té pozici pásky symbol  $s$ .
- Zavedeme proměnné  $x_{i,j,(q,s)}$  pro každé  $0 \leq i \leq n^k$ ,  $0 \leq j \leq n^k$ ,  $s \in \Gamma$  a  $q \in Q$ . Hodnota **1** takové proměnné znamená, že v  $i$ -tém řádku a  $j$ -tém sloupci se nachází symbol  $(q, s)$ , tedy stroj má v  $i$ -té konfiguraci na  $j$ -té pozici pásky symbol  $s$ , na tento symbol ukazuje hlava a stroj se nachází ve stavu  $q$ .
- Formulí  $\phi_{\mathcal{M},w}$  vytvoříme jako

$$\phi_{\mathcal{M},w} = \phi_{\text{cell}} \wedge \phi_{\text{start}} \wedge \phi_{\text{move}} \wedge \phi_{\text{accept}}$$

$\phi_{cell}$  je formule, která musí zajistit, že v každé buňce tabulky je právě jeden symbol.

$$\phi_{cell} = \bigwedge_{0 \leq i, j \leq n^k} \left[ \left( \bigvee_{s \in \Gamma \cup (Q \times \Gamma)} x_{i,j,s} \right) \wedge \left( \bigwedge_{r \neq t \in \Gamma \cup (Q \times \Gamma)} (\neg x_{i,j,r} \vee \neg x_{i,j,t}) \right) \right]$$

$\bigvee_{s \in \Gamma \cup (Q \times \Gamma)} x_{i,j,s}$  zajistí, že každá buňka obsahuje alespoň jeden symbol.  
 $\bigwedge_{r \neq t \in \Gamma \cup (Q \times \Gamma)} (\neg x_{i,j,r} \vee \neg x_{i,j,t})$  zajistí, že žádná buňka neobsahuje více různých symbolů.

$\phi_{start}$  musí zajistit, že první řádek odpovídá počáteční konfiguraci. Tedy na pásce je slovo  $w$ , hlava ukazuje na první symbol tohoto slova a stav je  $q_0$ .

$$\phi_{start} = x_{0,0,\#} \wedge x_{0,1,(q_0,w_1)} \wedge x_{0,2,w_2} \wedge \dots \wedge x_{0,n,w_n} \wedge x_{0,n+1,\#} \wedge \dots \wedge x_{0,n^k,\#}$$

$\phi_{accept}$  musí zajistit, že na posledním řádku tabulky se vyskytuje stav  $q_{ANO}$ . Předpokládáme, že pokud Turingův stroj skončí výpočet dříve než po  $n^k$  krocích, všechny následující konfigurace (řádky tabulky) jsou stejné.

$$\phi_{accept} = \bigvee_{1 \leq j \leq n^k, w \in \Gamma} x_{n^k, j, (q_{ANO}, w)}$$

$\phi_{move}$  musí zajistit, že každá následující konfigurace vznikne z předchozí správným způsobem.

- Musíme prozkoumat všechna “okna” o dvou řádcích a třech sloupcích.
- Pokud horní řádek neobsahuje stav, může v dolním přibýt na kraji stav, ale symboly musí zůstat stejné.
- Pokud horní řádek obsahuje stav, může na spodním být stav posunutý a symbol na pozici, kde stav byl, může být jiný.
- Konkrétní správná okna záleží jen na přechodové funkci stroje  $\mathcal{M}$ . Vůbec nezáleží na délce vstupu  $w$ , takže počet správných oken je konstantní vzhledem k velikosti vstupu.
- Označme  $SO$  množinu správných oken, tedy šestic symbolů z  $\Gamma \cup (Q \times \Gamma)$  takových, že tvoří správné okno

$$\phi_{move} = \bigwedge_{1 \leq i, j \leq n^k} \bigvee_{(a_1, \dots, a_6) \in SO} (x_{i-1, j-1, a_1} \wedge \dots \wedge x_{i, j+1, a_6})$$

- Formule  $\phi_{\mathcal{M},w} = \phi_{cell} \wedge \phi_{start} \wedge \phi_{move} \wedge \phi_{accept}$  je polynomiální vzhledem k  $n$ .
- Jestliže je splnitelná tato formule, dokážeme vyplnit tabulku tak, že reprezentuje přijímající výpočet Turingova stroje  $\mathcal{M}$  na slově  $w$ .
- Jestliže je formule nespíitelná, tak neexistuje přijímající výpočet Turingova stroje  $\mathcal{M}$  na slově  $w$ , protože se nám nepodaří vyplnit tabulku reprezentující takový výpočet.
- Ukázali jsme tedy polynomiální převod libovolného problému z NP na problém SAT



Důkazy NP-úplnosti problémů můžeme rozdělit na:

- 1 Přímé - podobně jako pro SAT se ukáže přímo převoditelnost všech možných problémů z NP na zadaný problém. Nejčastěji se využívá posloupnosti konfigurací Turingova stroje - odpověď na problém pro který dokazujeme NP-úplnost bude ANO právě tehdy, když existuje přijímající výpočet daného Turingova stroje na daném slově.
- 2 Nepřímé - převodem z nějakého již známého NP-úplného problému. Vzniká vlastně sekvence převodů začínající nějakým s přímo dokázanou NP-úplností a končící problémem, pro který NP-úplnost právě dokazujeme.

Všechny důkazy, které budeme dále dělat, budou vedeny způsobem 2. Ten je obecně běžnější a častěji používaný.

Důkazy NP-úplnosti problémů můžeme rozdělit na:

- 1 Přímé - podobně jako pro SAT se ukáže přímo převoditelnost všech možných problémů z NP na zadaný problém. Nejčastěji se využívá posloupnosti konfigurací Turingova stroje - odpověď na problém pro který dokazujeme NP-úplnost bude ANO právě tehdy, když existuje přijímající výpočet daného Turingova stroje na daném slově.
- 2 Nepřímé - převodem z nějakého již známého NP-úplného problému. Vzniká vlastně sekvence převodů začínající nějakým s přímo dokázanou NP-úplností a končící problémem, pro který NP-úplnost právě dokazujeme.

Všechny důkazy, které budeme dále dělat, budou vedeny způsobem 2. Ten je obecně častěji používaný.

## Definice

Problém  $P_1$  je **polynomiálně převoditelný** na problém  $P_2$ , jestliže existuje algoritmus s polynomiální časovou složitostí, který pro libovolný vstup  $w$  problému  $P_1$  sestrojí vstup  $w'$  problému  $P_2$ , přičemž platí, že odpověď na otázku problému  $P_1$  pro vstup  $w$  je stejná jako odpověď na otázku problému  $P_2$  pro vstup  $w'$ .

- Algoritmus převodu musí být obecný, tedy ke každému možnému vstupu problému  $P_1$  umět sestrojít vstup problému  $P_2$
- Musíme ověřit, že z každého vstupu s odpovědí ANO vznikne vstup s odpovědí ANO (nebo ekvivalentně, že pokud vznikne vstup s odpovědí NE, tak i původní měl odpověď NE).
- Musíme ověřit, že z každého vstupu s odpovědí NE vznikne vstup s odpovědí NE (nebo ekvivalentně, že pokud vznikne vstup s odpovědí ANO, tak i původní měl odpověď ANO).

## SAT - Splnitelnost logických formulí

**Vstup:** Logická formule  $\Phi$  v konjunktivní normální formě

**Výstup:** ANO, pokud existuje ohodnocení  $\mu$  proměnných takové, aby pravdivostní hodnota formule  $\mu(\Phi) = 1$

**Příklad:** Mějme formuli

$$\Phi = (a_1 \vee \neg a_2 \vee a_4) \wedge (\neg a_1 \vee a_3) \wedge (\neg a_1 \vee \neg a_3 \vee \neg a_4)$$

**Řešení:** Pokud ohodnotíme např.  $\mu(a_1) = \mu(a_3) = 1$ ,  $\mu(a_2) = \mu(a_4) = 0$  dostaneme  $\mu(\Phi) = 1$ , tedy odpověď je, že formule je splnitelná.

**Příklad:** Mějme formuli  $\Phi = (a_1 \vee a_2) \wedge \neg a_1 \wedge \neg a_2$

**Řešení:** Formule není splnitelná.

## 3-SAT - Splnitelnost logických formulí

**Vstup:** Logická formule  $\Phi$  v konjunktivní normální formě taková, že každá klauzule obsahuje právě 3 literály

**Výstup:** ANO, pokud existuje ohodnocení  $\mu$  proměnných takové, aby pravdivostní hodnota formule  $\mu(\Phi) = 1$

**Příklad:** Mějme formuli

$$\Phi = (a_1 \vee \neg a_2 \vee a_4) \wedge (\neg a_1 \vee a_3 \vee a_3) \wedge (\neg a_1 \vee \neg a_3 \vee \neg a_4)$$

**Řešení:** Pokud ohodnotíme např.  $\mu(a_1) = \mu(a_3) = 1$ ,  $\mu(a_2) = \mu(a_4) = 0$  dostaneme  $\mu(\Phi) = 1$ , tedy odpověď je, že formule je splnitelná.

Problém 3-SAT je NP-úplný.

- Příslušnost do třídy NP ukážeme stejně jako v případě SAT. Stačí jako svědka vzít ohodnocení všech proměnných a ověřit, zda pro dané ohodnocení je formule pravdivá.
- NP-obtížnost ukážeme převodem z problému SAT

Algoritmus převodu musí k obecné formuli v konjunktivní normální formě sestrojít formuli, která bude mít v každé klauzuli právě 3 literály.

Problém působí pouze klauzule, kde jsou méně než 3 nebo více než 3 literály.

Algoritmus převodu:

- Ke klauzuli s 1 literálem  $x$  (resp  $\neg x$ ) sestrojíme klauzuli  $(x \vee x \vee x)$  (resp.  $(\neg x \vee \neg x \vee \neg x)$ )
- Ke klauzuli se 2 literály  $(x \vee y)$  (kde  $x, y$  jsou buď proměnné nebo jejich negace) sestrojíme klauzuli  $(x \vee x \vee y)$
- Ke klauzuli s  $n > 3$  literály  $(x_1 \vee x_2 \vee \dots \vee x_n)$  sestrojíme  $n - 2$  klauzulí:

$$(x_1 \vee x_2 \vee y_1) \wedge (\neg y_1 \vee x_3 \vee y_2) \wedge (\neg y_2 \vee x_4 \vee y_3) \wedge \dots \wedge (\neg y_{n-3} \vee x_{n-1} \vee x_n),$$

kde  $x_1, x_2, \dots, x_n$  jsou literály (proměnné nebo negace proměnných) a  $y_1, y_2, \dots, y_{n-3}$  jsou nově přidané proměnné.

**Příklad:** Mějme formuli

$$\Phi = (a_1 \vee \neg a_2 \vee a_4) \wedge (\neg a_1 \vee a_3) \wedge (\neg a_1 \vee \neg a_3 \vee \neg a_4 \vee a_5 \vee \neg a_5) \wedge a_5.$$

*Řešení:*

- Klauzule  $(a_1 \vee \neg a_2 \vee a_4)$  může zůstat
- Klauzuli  $(\neg a_1 \vee a_3)$  nahradíme  $(\neg a_1 \vee \neg a_1 \vee a_3)$
- Klauzuli  $a_5$  nahradíme  $(a_5 \vee a_5 \vee a_5)$
- Klauzuli  $(\neg a_1 \vee \neg a_3 \vee \neg a_4 \vee a_5 \vee \neg a_5)$  nahradíme  
 $(\neg a_1 \vee \neg a_3 \vee y_1) \wedge (\neg y_1 \vee \neg a_4 \vee y_2) \wedge (\neg y_2 \vee a_5 \vee \neg a_5)$

Výsledkem je  $(a_1 \vee \neg a_2 \vee a_4) \wedge (\neg a_1 \vee \neg a_1 \vee a_3) \wedge (\neg a_1 \vee \neg a_3 \vee y_1) \wedge (\neg y_1 \vee \neg a_4 \vee y_2) \wedge (\neg y_2 \vee a_5 \vee \neg a_5) \wedge (a_5 \vee a_5 \vee a_5)$



- $x$ ,  $(x \vee x)$  i  $(x \vee x \vee x)$  mají stejnou hodnotu při stejném ohodnocení literálu  $x$ . Rozšíření klauzule o literál, který se v ní vyskytuje nemá tedy vliv na splnitelnost.
- Pokud byla formule splnitelná vznikne splnitelná formule:
  - Ohodnocení proměnných muselo alespoň jednomu literálu z klauzule  $(x_1 \vee x_2 \vee \dots \vee x_n)$  přiřadit **1**.
  - Stejně ohodnocení znamená, že v  $(x_1 \vee x_2 \vee y_1) \wedge (\neg y_1 \vee x_3 \vee y_2) \wedge (\neg y_2 \vee x_4 \vee y_3) \wedge \dots \wedge (\neg y_{n-3} \vee x_{n-1} \vee x_n)$  bude alespoň jedna klauzule mít hodnotu **1** bez ohledu na hodnoty  $y_1, \dots, y_{n-3}$
  - Pokud to není první klauzule z nově vzniklých, je v ní je literál  $\neg y_i$ , který můžeme ohodnotit **0** a v předchozí klauzuli je  $y_i$  ohodnocena **1** a hodnota této klauzule je tedy také **1**. Obdobně nejde-li o poslední klauzuli, je v ní je literál  $\neg y_{i+1}$ , který můžeme ohodnotit **0** a v následující klauzuli je  $y_{i+1}$  ohodnocena **1** a hodnota klauzule je **1**.
  - Podobnou úvahu můžeme induktivně opakovat až k první a poslední klauzuli. Pokud v nějakém ohodnocení byla hodnota původní klauzule **1**, bude existovat ohodnocení takové, že hodnota všech nových klauzulí bude **1**.

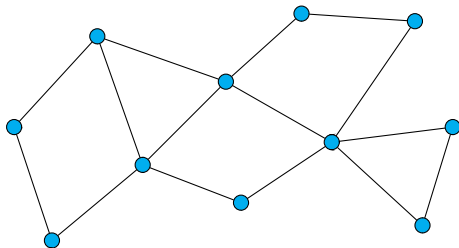
- Pokud vznikne splnitelná formule byla i původní splnitelná:
  - Ohodnocení, které přiřadí formuli  $(x_1 \vee x_2 \vee y_1) \wedge (\neg y_1 \vee x_3 \vee y_2) \wedge (\neg y_2 \vee x_4 \vee y_3) \wedge \dots \wedge (\neg y_{n-3} \vee x_{n-1} \vee x_n)$  hodnotu **1** musí  $n - 2$  výskytům literálů přiřadit **1**. Proměnných  $y$  je jen  $n - 3$ , jen jedna z  $y$  nebo  $\neg y$  může být **1** a každá má ve formuli jen jeden výskyt v negované a jeden v nenegované verzi. Tedy alespoň jeden z literálů  $x$  musí být ohodnocen **1**.
  - Uvažujeme-li na původní formuli stejné ohodnocení, tak musí mít stejný literál  $x$  také hodnotu **1**.
- Ukázali jsme tedy, že po převodu z odpovědi ANO pro SAT plyne odpověď ANO pro 3-SAT a opačně
- Délka vytvořené formule nebude více než trojnásobně delší než původní
- Problém 3-SAT je tedy NP-úplný

## 3-COL - Problém „Barvení grafu 3 barvami“

**Vstup:** Neorientovaný graf  $G$

**Výstup:** Lze vrcholy grafu obarvit **3** barvami tak, aby žádné dva vrcholy spojené hranou neměly stejnou barvu?

**Příklad:**



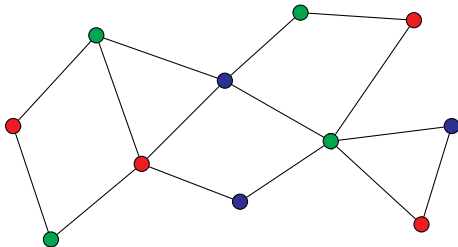
# Barvení grafu 3 barvami

## 3-COL - Problém „Barvení grafu 3 barvami“

**Vstup:** Neorientovaný graf  $G$

**Výstup:** Lze vrcholy grafu obarvit 3 barvami tak, aby žádné dva vrcholy spojené hranou neměly stejnou barvu?

**Příklad:**



**Odpověď:** ANO

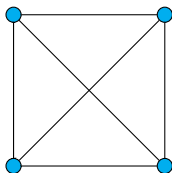
# Barvení grafu 3 barvami

## 3-COL - Problém „Barvení grafu 3 barvami“

**Vstup:** Neorientovaný graf  $G$

**Výstup:** Lze vrcholy grafu obarvit 3 barvami tak, aby žádné dva vrcholy spojené hranou neměly stejnou barvu?

**Příklad:**



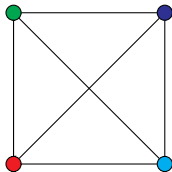
# Barvení grafu 3 barvami

## 3-COL - Problém „Barvení grafu 3 barvami“

**Vstup:** Neorientovaný graf  $G$

**Výstup:** Lze vrcholy grafu obarvit 3 barvami tak, aby žádné dva vrcholy spojené hranou neměly stejnou barvu?

**Příklad:**

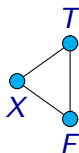


Odpověď: NE

Hledáme algoritmus, který sestrojí k zadané formuli v konjunktivní normální formě s právě 3 literály v každé klauzuli graf tak, že formule bude splnitelná právě tehdy, když půjde graf obarvit korektně 3 barvami.

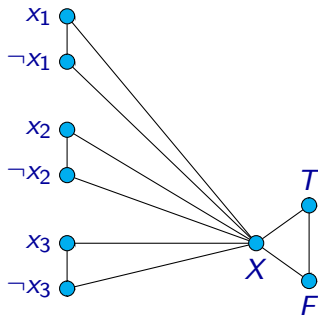
- Graf bude obsahovat vrcholy  $X, F, T$  propojené do trojúhelníku
- Pro každou proměnnou bude graf mít dvojici vrcholů  $x, \neg x$  popojené hranou. Každý z nich bude propojen hranou s  $X$ .
- Pro každou klauzuli přidáme 6 vrcholů  $c_1, c'_1, c_2, c'_2, c_3, c'_3$  a hrany  $(c_1, c_2), (c_1, c_3), (c_2, c_3), (c_1, c'_1), (c_2, c'_2), (c_3, c'_3)$
- Přidáme hrany mezi  $c'_i$  a  $x_j$  nebo  $\neg x_j$  podle toho, jaký literál se nachází na  $i$ -té pozici v dané klauzuli

**Příklad:** Uvažujme formuli  $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_3)$

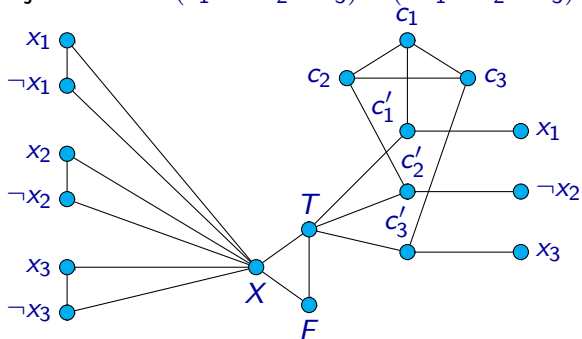




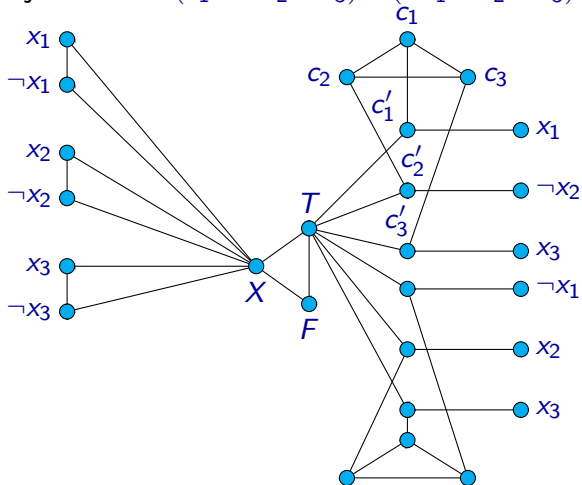
**Příklad:** Uvažujme formuli  $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_3)$



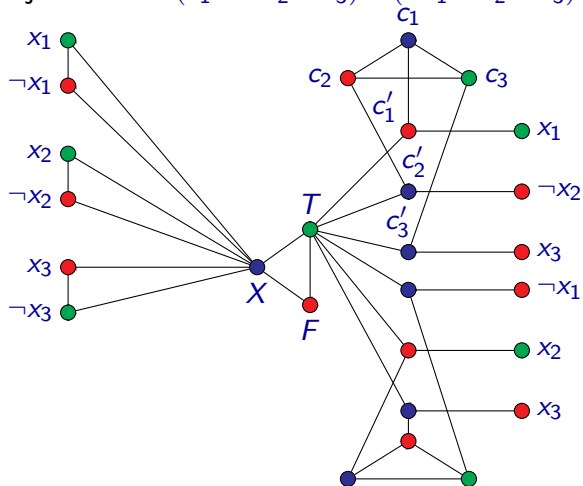
**Příklad:** Uvažujme formuli  $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_3)$



**Příklad:** Uvažujme formuli  $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_3)$



**Příklad:** Uvažujme formuli  $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_3)$

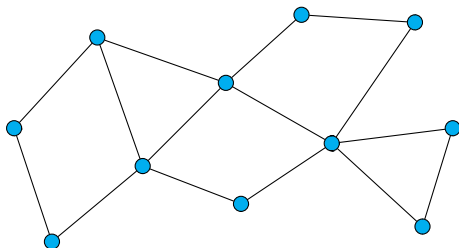


## IS - Problém „Nezávislá množina“

**Vstup:** Neorientovaný graf  $G$  a přirozené číslo  $k$

**Výstup:** Existuje v grafu nezávislá množina velikosti  $k$  (množina  $k$  vrcholů, které vzájemně nejsou propojeny hranou)?

**Příklad:**  $k = 5$

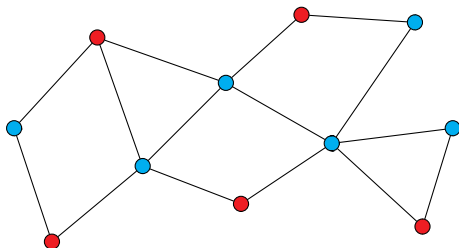


## IS - Problém „Nezávislá množina“

**Vstup:** Neorientovaný graf  $G$  a přirozené číslo  $k$

**Výstup:** Existuje v grafu nezávislá množina velikosti  $k$  (množina  $k$  vrcholů, které vzájemně nejsou propojeny hranou)?

**Příklad:**  $k = 5$



**Odpověď:** ANO

Hledáme algoritmus, který sestrojí ke grafu  $G$  graf  $G'$  a číslo  $k$  tak, že  $G$  bude obarvitelný 3 barvami právě tehdy, když v  $G'$  bude existovat nezávislá množina velikosti  $k$ .

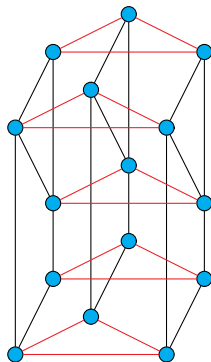
- Graf  $G'$  bude obsahovat 3 kopie grafu  $G$
- Všechny tři kopie každého vrcholu budou propojeny hranami do trojúhelníku
- Číslo  $k$  položíme rovno  $|G|$

**Příklad:**



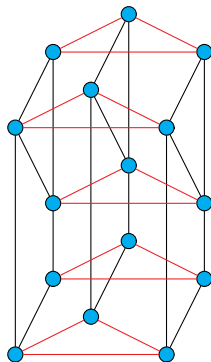


**Příklad:**



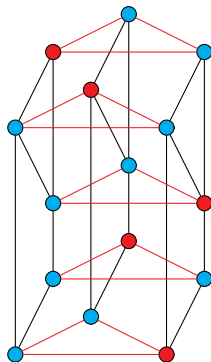
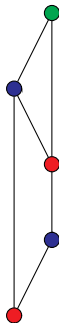
$k = 5$

**Příklad:**



$k = 5$

**Příklad:**



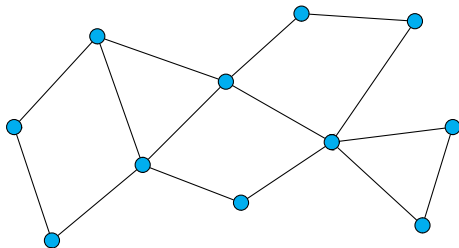
$k = 5$

## VC - Problém „Vrcholové pokrytí“

**Vstup:** Neorientovaný graf  $G$  a přirozené číslo  $k$

**Výstup:** Existuje v grafu množina vrcholů velikosti  $k$  taková, že každá hrana má alespoň jeden svůj vrchol v této množině?

**Příklad:**  $k = 6$

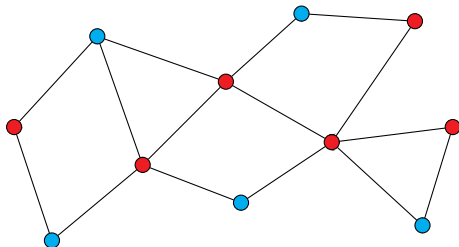


## VC - Problém „Vrcholové pokrytí“

**Vstup:** Neorientovaný graf  $G$  a přirozené číslo  $k$

**Výstup:** Existuje v grafu množina vrcholů velikosti  $k$  taková, že každá hrana má alespoň jeden svůj vrchol v této množině?

**Příklad:**  $k = 6$



**Odpověď:** ANO

# Vrcholové pokrytí

- Vrcholové pokrytí patří do NP. Stačí jako svědka vzít množinu  $k$  vrcholů a ověřit, že tato množina tvoří pokrytí.
- NP-obtížnost lehce ukážeme převodem z problému IS
- Graf nemusíme měnit, jen limit  $k$  pro pokrytí bude  $|G| - n$ , kde  $n$  je limit pro nezávislou množinu
- Pokud existovala nezávislá množina, tak žádná hrana v ní nemohla mít oba krajní vrcholy. Tedy alespoň jeden krajní vrchol každé hrany je v doplňku nezávislé množiny a ta tedy tvoří pokrytí požadované velikosti.
- Pokud bude existovat pokrytí, každá hrana v něm má alespoň jeden vrchol. Tedy v doplňku nejsou oba krajní vrcholy žádné hrany (žádné dva vrcholy nejsou spojeny hranou) a tedy tento doplněk tvoří nezávislou množinu velikosti  $n = |G| - k$

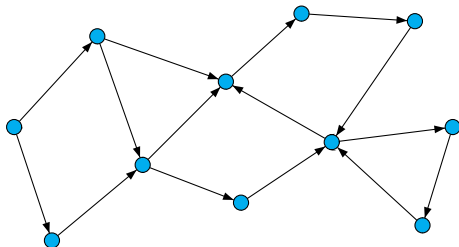
# Hamiltonovský cyklus

## HC - Problém „Hamiltonovský cyklus“

**Vstup:** Orientovaný graf  $G$

**Výstup:** Existuje v grafu Hamiltonovský cyklus (orientovaná kružnice procházející každý vrchol právě jednou)?

**Příklad:**



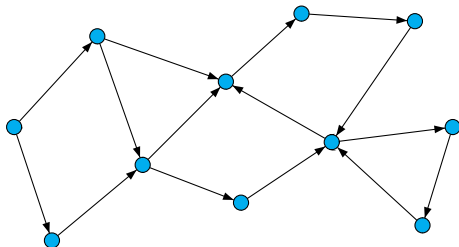
# Hamiltonovský cyklus

## HC - Problém „Hamiltonovský cyklus“

**Vstup:** Orientovaný graf  $G$

**Výstup:** Existuje v grafu Hamiltonovský cyklus (orientovaná kružnice procházející každý vrchol právě jednou)?

**Příklad:**



Odpověď: NE



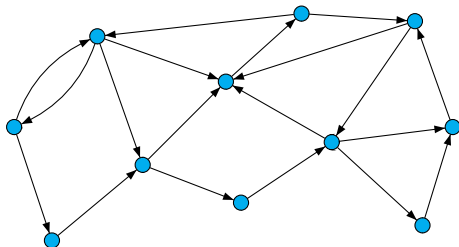
# Hamiltonovský cyklus

## HC - Problém „Hamiltonovský cyklus“

**Vstup:** Orientovaný graf  $G$

**Výstup:** Existuje v grafu Hamiltonovský cyklus (orientovaná kružnice procházející každý vrchol právě jednou)?

**Příklad:**



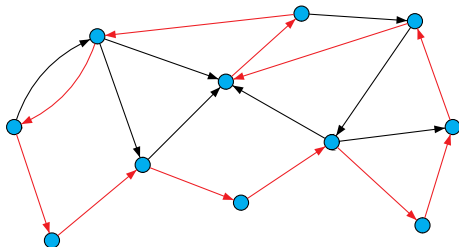
# Hamiltonovský cyklus

## HC - Problém „Hamiltonovský cyklus“

**Vstup:** Orientovaný graf  $G$

**Výstup:** Existuje v grafu Hamiltonovský cyklus (orientovaná kružnice procházející každý vrchol právě jednou)?

**Příklad:**



Odpověď: ANO

- Problém patří do třídy NP. Stačí vzít jako svědka množinu hran a ověřit, že tyto hrany tvoří cyklus a každý vrchol má právě jednu vstupní a výstupní hranu.
- NP-obtížnost můžeme ukázat například převodem z problému VC nebo SAT
- Oba důkazy jsou složitější na popsání, proto je vynecháme

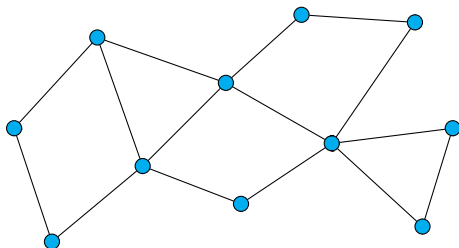
# Hamiltonovská kružnice

## HK - Problém „Hamiltonovská kružnice“

**Vstup:** Neorientovaný graf  $G$

**Výstup:** Existuje v grafu Hamiltonovská kružnice (neorientovaná kružnice procházející každý vrchol právě jednou)?

**Příklad:**

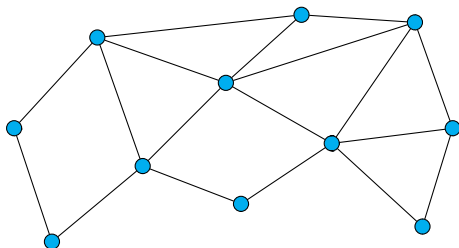


## HK - Problém „Hamiltonovská kružnice“

**Vstup:** Neorientovaný graf  $G$

**Výstup:** Existuje v grafu Hamiltonovská kružnice (neorientovaná kružnice procházející každý vrchol právě jednou)?

**Příklad:**



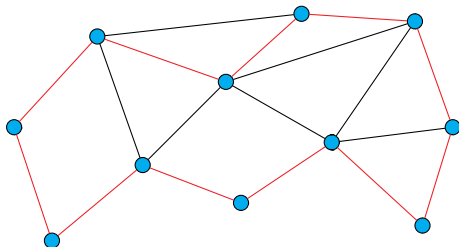
# Hamiltonovská kružnice

## HK - Problém „Hamiltonovská kružnice“

**Vstup:** Neorientovaný graf  $G$

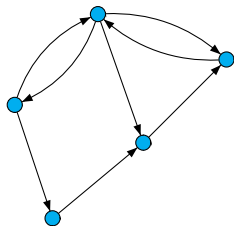
**Výstup:** Existuje v grafu Hamiltonovská kružnice (neorientovaná kružnice procházející každý vrchol právě jednou)?

**Příklad:**



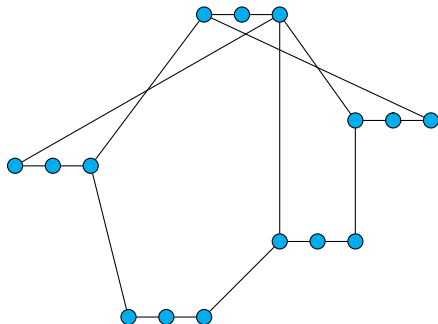
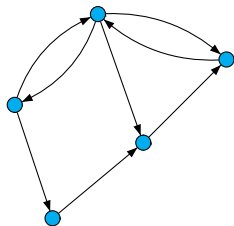
**Odpověď:** ANO

- Problém patří do třídy NP. Stačí vzít jako svědka množinu hran a ověřit, že tyto hrany tvoří kružnici a každý vrchol má právě dvě hrany.
- NP-obtížnost můžeme ukázat převodem z problému HC
- Ke každému vrcholu  $x$  orientovaného grafu dáme do neorientovaného tři vrcholy  $x_1, x_2, x_3$ , které spojíme hranami  $(x_1, x_2), (x_2, x_3)$ .
- Každá orientovaná hrana  $(x, y)$  bude reprezentována hranou  $(x_3, y_1)$

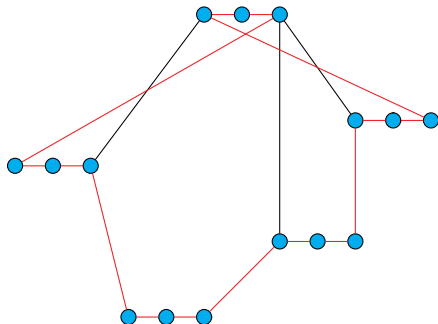
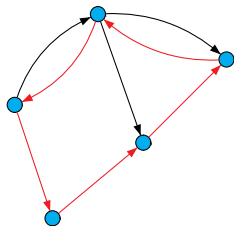




# Převod HC na HK



# Převod HC na HK



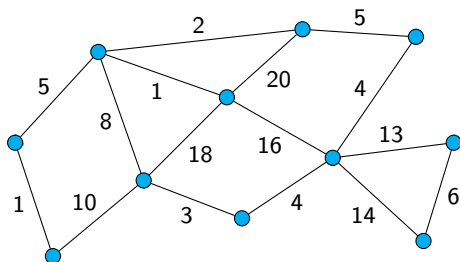
# Problém obchodního cestujícího

## TSP - Problém „obchodního cestujícího“

**Vstup:** Neorientovaný graf  $G$  s hranami ohodnocenými přirozenými čísly a číslo  $k$

**Výstup:** Existuje v grafu uzavřený sled procházející všemi vrcholy a mající součet délek hran (včetně opakovaných) maximálně  $k$ ?

**Příklad:**  $k = 70$



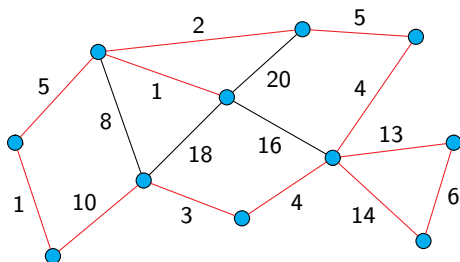
# Problém obchodního cestujícího

## TSP - Problém „obchodního cestujícího“

**Vstup:** Neorientovaný graf  $G$  s hranami ohodnocenými přirozenými čísly a číslo  $k$

**Výstup:** Existuje v grafu uzavřený sled procházející všemi vrcholy a mající součet délek hran (včetně opakovaných) maximálně  $k$ ?

**Příklad:**  $k = 70$



**Odpověď:** ANO, protože byl nalezen sled se součtem 69

- Problém patří do třídy NP. Stačí vzít jako svědka množinu hran a ověřit, že tyto hrany tvoří uzavřený sled procházející každým vrcholem a součet je menší nebo roven  $k$ .
- NP-obtížnost můžeme ukázat převodem z problému HK
- Vstupní graf do problému HK  $G$  převedeme na  $G'$  (vstup do TSP) tak, že jen ohodnotíme všechny hrany 1.
- Jako limit  $k$  pro délku cesty obchodního cestujícího vezmeme  $|G|$

## PART - Problém „loupežníka“

**Vstup:** Množina přirozených čísel  $A = \{a_1, a_2, \dots, a_n\}$

**Výstup:** ANO, jestliže je možné najít množinu  $X$  tak, že  $X \subset A$ ,  
$$\sum_{a_i \in X} a_i = \sum_{a_j \in A-X} a_j.$$

**Příklad:**  $A = \{10, 40, 60, 70, 100, 120, 140, 160, 300\}$

*Řešení:*

$X = \{10, 70, 120, 140, 160\}, \sum_{a_i \in X} a_i = 500$

$A - X = \{40, 60, 100, 300\}, \sum_{a_j \in A-X} a_j = 500$

# Převod 3-SAT na problém loupežníka

- Budeme reprezentovat proměnné a klauzule jako úseky bitů v dlouhých binárních číslech
- Pro každou proměnnou je v každém čísle vyhrazen 1 bit
- Pro každou klauzuli jsou vyhrazeny 3 bity
- Pro každý možný literál přidáme jedno číslo s 1 na místě reprezentujícím proměnnou tohoto literálu a dále počtem výskytů literálu v klauzuli v místě vyhrazeném každé klauzuli
- Přidáme jedno číslo F (reprezentující hodnotu false) s 1 na místě všech klauzulí
- Pro každou klauzuli přidáme 2 čísla s jedničkou jen na místě klauzule, pro kterou jsou určena

# Převod 3-SAT na problém loupežníka

**Příklad:** Uvažujme formuli  $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_3)$

		$c_1$	$c_2$
$x_1$	100	001	000
$\neg x_1$	100	000	001
$x_2$	010	000	001
$\neg x_2$	010	001	000
$x_3$	001	001	001
$\neg x_3$	001	000	000
$F$	000	001	001
$c_1$	000	001	000
$c_1$	000	001	000
$c_2$	000	000	001
$c_2$	000	000	001



# Převod 3-SAT na problém loupežníka

**Příklad:** Uvažujme formuli  $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_3)$

		$c_1$	$c_2$
$x_1$	100	001	000
$\neg x_1$	100	000	001
$x_2$	010	000	001
$\neg x_2$	010	001	000
$x_3$	001	001	001
$\neg x_3$	001	000	000
$F$	000	001	001
$c_1$	000	001	000
$c_1$	000	001	000
$c_2$	000	000	001
$c_2$	000	000	001

		$c_1$	$c_2$
$x_1$	100	001	000
$x_2$	010	000	001
$\neg x_3$	001	000	000
$c_2$	000	000	001
$c_2$	000	000	001
$c_1$	000	001	000
$c_1$	000	001	000
<hr/>		111	011
$\neg x_1$	100	000	001
$\neg x_2$	010	001	000
$x_3$	001	001	001
$F$	000	001	001
<hr/>		111	011

## ILP - Problém „Celočíselné lineární programování“

**Vstup:** Matice  $\mathbf{A}$  a vektor  $\vec{b}$  určující soustavu lineárních nerovnic s proměnnými  $z_i$ .

$$\mathbf{A} \cdot \vec{z} \leq \vec{b}$$

**Výstup:** Existuje vektor  $\vec{z} \in \{0, 1\}^*$  takový, že je soustava nerovnic  $\mathbf{A} \cdot \vec{z} \leq \vec{b}$  splněná?

### Příklad:

$$3z_1 - 2z_2 + 4z_3 \leq 1$$

$$-4z_1 - 3z_2 \leq -6$$

$$2z_1 - z_3 \leq 2$$

**Řešení:**

Pro  $z_1 = 1$ ,  $z_2 = 1$ ,  $z_3 = 0$  platí všechny nerovnice, takže odpověď je ANO.

- Uvažujme  $A = \{a_1, a_2, \dots, a_n\}$  množinu přirozených čísel, která je vstupem problému loupežníka
- Necht'  $a = \sum_{a_i \in A} a_i$ .
- Sestavíme sestavu dvou nerovnic:

$$a_1 z_1 + a_2 z_2 + \dots + a_n z_n \leq a/2$$

$$a_1(1 - z_1) + a_2(1 - z_2) + \dots + a_n(1 - z_n) \leq a/2$$

- Hodnota  $z_i = 1$  nám říká, že číslo  $a_i$  dáme do první množiny rozkladu
- Hodnota  $(1 - z_i) = 1$  nám říká, že číslo  $a_i$  dáme do druhé množiny rozkladu

- Pro mnoho důležitých problémů nejsou známy efektivní algoritmy. Řada těchto problémů je například NP-úplných.
- Je možné, že pro tyto problémy ani polynomiální algoritmy neexistují a my to zatím jenom neumíme dokázat.
- Přesto potřebujeme tyto problémy řešit.

Pokud chceme řešit nějaký problém, tak v ideálním případě chceme použít k jeho řešení algoritmus, který:

- Bude **polynomiální**, tj. rychle skončí pro libovolnou vstupní instanci.
- Bude **korektní**, tj. pro libovolnou vstupní instanci najde správnou odpověď.

Pokud nevíme, jak takový algoritmus najít, musíme trochu slevit ze svých požadavků.

- **Obtížné instance versus typické instance**

Při studiu složitosti problémů zkoumáme většinou složitost v nejhorším případě.

Instance, kvůli kterým je problém těžký, mohou být dost odlišné od „typických“ instancí, pro které chceme problém řešit.

Při podrobnějším zkoumání problému můžeme nalézt určitou podmnožinu instancí, pro které je možné problém rychle řešit.

## Problém batohu

**Vstup:** Čísla  $a_1, a_2, \dots, a_m$  a číslo  $s$ .

**Otázka:** Existuje podmnožina množiny čísel  $a_1, a_2, \dots, a_m$  taková, že součet čísel v této podmnožině je  $s$ ?

**Poznámka:** Jako velikost instance bereme celkový počet bitů v zápisu čísel  $a_1, a_2, \dots, a_m$  a  $s$ .

Tento problém je NP-úplný. Neumíme ho v rozumném čase řešit, pokud čísla v instanci budou „velká“ (např. 1000-bitová).

Pokud je však hodnota  $s$  malá (např. do 1000000, tj. asi 20 bitů), je možné tento problém vyřešit během několika sekund i pro relativně velké hodnoty  $m$  (např.  $m = 10000$ ).

## Problém „SAT“

**Vstup:** Booleovská formule  $\phi$  v KNF.

**Otázka:** Je  $\phi$  splnitelná?

Pokud  $\phi$  obsahuje pouze Hornovy klauzule, je možné SAT vyřešit v polynomiálním čase.

**Poznámka:** Hornova klauzule je klauzule, ve které se nachází nejvýše jeden pozitivní literál.

Například  $(\neg x_1 \vee \neg x_2 \vee \neg x_3 \vee \neg x_4 \vee x_5)$  je Hornova klauzule.

Všimněte si, že tato klauzule je ekvivalentní formuli

$$(x_1 \wedge x_2 \wedge x_3 \wedge x_4) \Rightarrow x_5$$



**SAT** je možné vyřešit v polynomiálním čase také v případě, kdy se omezíme na formule, kde každá klauzule obsahuje nanejvýš dva literály.

**Příklad:**

$$(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge (x_3 \vee x_4) \wedge (x_2 \vee \neg x_4)$$

## Exponenciální algoritmy

I exponenciální algoritmus může být prakticky použitelný, pokud:

- Instance, pro které problém chceme řešit, jsou poměrně malé.
- Složitost je sice exponenciální, ale roste pomaleji než  $2^n$ .
- Algoritmus je co nejoptimálněji naprogramován.

### Příklad:

- $f(n) = (1.2)^n$  pro  $n = 100$  je  $f(n) \approx 86 \cdot 10^6$ .
- $f(n) = 10 \cdot 2^{\sqrt{n}}$  pro  $n = 300$  je  $f(n) \approx 1.64 \cdot 10^6$ .

## Menší požadavky na korektnost

- **Randomizované algoritmy** – algoritmy, které využívají při výpočtu generátor náhodných čísel.

Existuje určitá nenulová pravděpodobnost, že algoritmus vrátí chybný výsledek.

Pro libovolně malé  $\varepsilon > 0$  jsme však schopni zaručit, že pravděpodobnost chyby není větší než  $\varepsilon$ .

- **Aproximační algoritmy** – používají se pro řešení optimalizačních problémů.

U těchto algoritmů není zaručeno, že naleznou optimální řešení, ale je zaručeno, že naleznou řešení, které nebude o moc horší než optimální řešení (např. nanejvýš  $2\times$  horší).

## Prvočíslnost

**Vstup:** Přirozené číslo  $p$ .

**Otázka:** Je  $p$  prvočíslo?

Pro „malá“  $p$  (např. do  $10^{12}$ ) stačí vyzkoušet čísla od 2 do  $\lfloor \sqrt{p} \rfloor$ , zda některé z nich dělí  $p$ .

Tento problém má velký význam v kryptografii, kde ho potřebujeme řešit pro čísla, která mají délku několik tisíc bitů.

# Testování prvočíslnosti

To, zda je možné testovat prvočíslnost v polynomiálním čase, byl dlouho otevřený problém.

V roce 2002 se podařilo najít polynomiální algoritmus se složitostí  $O(n^{12})$  (N. Saxona, M. Agrawal, N. Kayal).

Později se ho podařilo zlepšit na  $O(n^8)$ , ale pro praktické účely je stále nepoužitelný.

V praxi se používají **randomizované** algoritmy se složitostí  $O(n^3)$ :

- Miller-Rabin (1976)
- Solovay-Strassen (1977)

**Poznámka:**  $n$  zde označuje počet bitů čísla  $p$ .

S problémem testování prvočíslnosti úzce souvisí následující problém.

## Faktorizace

**Vstup:** Přirozené číslo  $p$ .

**Výstup:** Rozklad čísla  $p$  na prvočísla.

Na rozdíl od prvočíslnosti není pro problém faktorizace znám žádný efektivní algoritmus, a to ani randomizovaný.

Jedna z používaných šifer je tzv. RSA kryptosystém, který je založen na tom, že:

- Umíme rychle najít velká prvočísla (a rychle je mezi sebou vynásobit).
- Není známo, jak v rozumném čase z tohoto součinu zjistit původní prvočísla.

# Testování prvočíselnosti

**Poznámka:** Pokud umíme rychle testovat, zda je dané číslo prvočíslem, není problém velké prvočíslo náhodně vygenerovat, neboť je známo, že prvočísla jsou mezi ostatními čísly rozmístěna poměrně „hustě“.

$\pi(n)$  – počet prvočísel v intervalu  $1, 2, \dots, n$

Je dokázáno, že

$$\lim_{n \rightarrow \infty} \frac{\pi(n)}{n / \ln n} = 1$$

neboli jinak řečeno, mezi čísly od  $1$  do  $n$  je zhruba  $n / \ln n$  prvočísel.

Pokud tedy chceme náhodně vygenerovat  $k$ -bitové prvočíslo, stačí zhruba  $k$  pokusů.

Randomizovaný algoritmus:

- používá během výpočtu generátor náhodných čísel
- může pro tentýž vstup skončit s různým výsledkem
- existuje určitá pravděpodobnost, že vrátí chybný výsledek

Aby měl randomizovaný algoritmus praktický smysl, musíme mít možnost regulovat pravděpodobnost chyby:

*Pro libovolně malé  $\epsilon > 0$  musíme být schopni zaručit, že pravděpodobnost chyby nebude větší než  $\epsilon$ .*



Vezměme si například algoritmus Miller-Rabin pro testování prvočíselnosti.

Pro libovolné  $n$  vrací buď odpověď „Prvočíslo“ nebo odpověď „Složené“.

- Pokud  $n$  je prvočíslo, vrátí vždy odpověď „Prvočíslo“.
- Pokud  $n$  je číslo složené, je pravděpodobnost toho, že vrátí odpověď „Složené“ nejméně 50%.

Může však vrátit (chybnou) odpověď „Prvočíslo“.

Algoritmus můžeme spouštět opakovaně. Výsledek při dalším spuštění je nezávislý na výsledcích z předchozích spuštění.

Řekněme, že algoritmus spustíme  $m$  krát (pro tentýž vstup  $n$ ).

- Pokud alespoň jednou dostaneme odpověď „Složené“, pak víme s jistotou, že  $n$  je číslo složené.
- Pokud pokaždé dostaneme odpověď „Prvočíslo“, pak pravděpodobnost toho, že  $n$  není prvočíslo je maximálně

$$\left(\frac{1}{2}\right)^m = 2^{-m}$$

Například pro  $m = 100$  je pravděpodobnost chyby zanedbatelně malá.

**Poznámka:** Lze například odvodit, že pravděpodobnost toho, že počítač bude zasažen během dané mikrosekundy meteoritem, je nejméně  $2^{-100}$  za předpokladu, že každých 1000 let je meteoritem zničeno alespoň  $100 \text{ m}^2$  zemského povrchu.

Randomizované algoritmy jsou typicky založeny na hledání **svědků** (witness), kteří „dovědí“ určitou odpověď vydanou algoritmem.

Tyto svědky vybíráme z množiny **potenciálních svědků**:

- Náhodně vybereme nějakého potenciálního svědka.
- Ověříme, zda se jedná o skutečného svědka, a podle toho vydáme odpověď.

Například v algoritmu Miller-Rabin hledáme svědky složenosti čísla  $n$ .

Vybíráme je z množiny čísel  $\{1, 2, \dots, n-1\}$ :

- Pokud  $n$  je prvočíslo, žádní svědci složenosti neexistují.
- Pokud  $n$  není prvočíslo, je zaručeno, že alespoň polovina čísel v množině  $\{1, 2, \dots, n-1\}$  jsou svědci složenosti  $n$ .

## Malá Fermatova věta:

Pokud  $n$  je prvočíslo, pak pro každé  $a$  z množiny  $\{1, 2, \dots, n-1\}$  platí

$$a^{n-1} \equiv 1 \pmod{n}$$

Malou Fermatovu větu lze použít k testování prvočíslnosti (tzv. Fermatův test):

- Pro dané  $n$  zvolíme nějaké  $a \in \{1, 2, \dots, n-1\}$ .
- Pokud  $a^{n-1} \not\equiv 1 \pmod{n}$ , pak  $n$  určitě není prvočíslo.
- Pokud  $a^{n-1} \equiv 1 \pmod{n}$ , pak  $n$  je možná prvočíslo.

Je překvapivé, že tato procedura vrací chybný výsledek jen poměrně zřídka:

- Pro  $a = 2$  a  $n < 10000$  je jen 22 hodnot, pro které dostaneme chybnou odpověď.
- Pro  $a = 2$  a  $n \rightarrow \infty$  se pravděpodobnost toho, že pro náhodně vybrané  $n$  dostaneme chybnou odpověď, blíží nule.

# Testování prvočíselnosti

Fermatův test není 100% spolehlivý. Existují složená čísla, která splňují podmínku  $a^{n-1} \equiv 1 \pmod{n}$  pro všechna  $a \in \{1, 2, \dots, n-1\}$  nesoudělná s  $n$ .

Tato čísla se nazývají **Carmichaelova čísla** a jsou extrémně vzácná. Například mezi čísly do  $10^8$  existuje jen 255 Carmichaelových čísel.

Prvních 15 Carmichaelových čísel:

561, 1105, 1729, 2465, 2821, 6601, 8911, 10585, 15841, 29341, 41041, 46657, 52633, 62745, 63973

## Problém

Vstup: Přirozená čísla  $a, b, n$ .

Výstup: Hodnota  $a^b \bmod n$ .

Využijeme toho, že  $a^{2i} = a^i \cdot a^i$  a  $a^{2i+1} = a^i \cdot a^i \cdot a$ .

## MODULAR-EXPONENTIATION( $a, b, n$ )

```
1  $d \leftarrow 1$ 
2  $\langle b_k, b_{k-1}, \dots, b_0 \rangle$  je binární reprezentace  $b$ 
3 for  $i \leftarrow k$  downto 0
4     do  $d \leftarrow (d \cdot d) \bmod n$ 
5         if  $b_i = 1$ 
6             then  $d \leftarrow (d \cdot a) \bmod n$ 
7 return  $d$ 
```



# Testování prvočíselnosti (Miller-Rabin)

Na umocňování je založen i následující algoritmus pro testování prvočíselnosti:

MILLER-RABIN( $n,s$ )

```
1  for  $j \leftarrow 1$  to  $s$ 
2      do  $a \leftarrow \text{RANDOM}(2, n - 1)$ 
3         if WITNESS( $a, n$ )
4            then return „Složené“      ▷  $n$  je zcela jistě složené.
5  return „Prvočíslo“      ▷  $n$  je s velkou pravděpodobností prvočíslo.
```

Využívá malou Fermatovu větu a toho, že platí následující tvrzení:

Jestliže  $p$  je liché prvočíslo, pak rovnice  $x^2 \equiv 1 \pmod{p}$  má právě dvě řešení:  $x = 1$  a  $x = p - 1$ .

# Testování prvočíselnosti (Miller-Rabin)

WITNESS( $a, n$ )

```
1   $d \leftarrow 1$ 
2   $\langle b_k, b_{k-1}, \dots, b_0 \rangle$  je binární reprezentace  $n - 1$ 
3  for  $i \leftarrow k$  downto 0
4      do  $x \leftarrow d$ 
5           $d \leftarrow (d \cdot d) \bmod n$ 
6          if  $d = 1$  and  $x \neq 1$  and  $x \neq n - 1$ 
7              then return TRUE
8          if  $b_i = 1$ 
9              then  $d \leftarrow (d \cdot a) \bmod n$ 
10 if  $d \neq 1$ 
11     then return TRUE
12 return FALSE
```

- Množina potenciálních svědků bývá obrovská – většinou exponenciálně velká

**Poznámka:** Kdyby byla polynomiálně velká mohli bychom systematicky projít všechny potenciální svědky a nepotřebovali bychom randomizovaný algoritmus.

- Musí být zaručeno, že pokud nějakí skuteční svědci existují, pak jich je dostatečně mnoho, čímž je zaručeno, že pravděpodobnost, že na nějakého svědka narazíme, je dostatečně vysoká.

# Randomizovaný komunikační protokol

Máme dvojici počítačů  $C_1$  a  $C_2$ , které jsou velmi daleko od sebe (např. jeden v Evropě a druhý v Americe).

Oba počítače obsahují tutéž databázi, která by měla obsahovat přesně tatáž data.

Naším cílem je navrhnout vhodný komunikační protokol, pomocí kterého ověříme, že obě databáze obsahují přesně tatáž data.

Označme  $n$  počet bitů dat v databázi. Řekněme, že  $n = 10^{16}$ , a že tedy není možné přenášet celý obsah databáze, nebo by to bylo příliš časově náročné.

**Poznámka:** Není těžké ukázat, že každý deterministický protokol řešící tento problém musí přenést minimálně  $n$  bitů.

## Počáteční situace:

$C_1$  má  $n$ -bitovou sekvenci  $x = x_1 \cdots x_n$ ,

$C_2$  má  $n$ -bitovou sekvenci  $y = y_1 \cdots y_n$ .

**Cíl:** Zjistit, zda  $x = y$ .

- 1  $C_1$  zvolí náhodně prvočíslo  $p$  z množiny  $\{2, 3, \dots, n^2\}$ .
- 2  $C_1$  spočte číslo  $s = \text{Num}(x) \bmod p$  a pošle  $C_2$  dvojici  $(s, p)$ .
- 3  $C_2$  přijme  $(s, p)$  a spočte číslo  $t = \text{Num}(y) \bmod p$ .  
Pokud  $s \neq t$ , vydá odpověď „ $x \neq y$ “, jinak vydá odpověď „ $x = y$ “.

**Objem přenášených dat:**  $s$  i  $p$  mají maximálně  $2 \cdot \lceil \lg n \rceil$  bitů, takže se celkem přenáší maximálně  $4 \cdot \lceil \lg n \rceil$  bitů.

**Příklad:** Pro  $n = 10^{16}$  se bude přenášet maximálně  $4 \cdot 16 \cdot \lceil \lg 10 \rceil = 256$  bitů.

**Pravděpodobnost chyby:**

- Pokud  $x = y$ , pak pro libovolné  $p$  platí

$$\text{Num}(x) \bmod p = \text{Num}(y) \bmod p$$

takže dostaneme vždy odpověď „ $x = y$ “  
(pravděpodobnost chyby je 0).

# Randomizovaný komunikační protokol

- Pokud  $x \neq y$  a dostaneme chybnou odpověď „ $x = y$ “, znamená to, že

$$z = \text{Num}(x) \bmod p = \text{Num}(y) \bmod p$$

neboli existují čísla  $x', y'$  taková, že

$$\text{Num}(x) = x' \cdot p + z \qquad \text{Num}(y) = y' \cdot p + z$$

takže  $p$  je dělitelem čísla  $w = |\text{Num}(x) - \text{Num}(y)|$ , neboť

$$\text{Num}(x) - \text{Num}(y) = x' \cdot p - y' \cdot p = (x' - y') \cdot p$$

Prvočíslo  $p$  bylo vybíráno náhodně z množiny  $\{2, 3, \dots, n^2\}$ , která obsahuje

$$\pi(n^2) \approx \frac{n^2}{\ln n^2}$$

prvočísel. Musíme zjistit, kolik z těchto prvočísel je dělitelem  $w$ .

# Randomizovaný komunikační protokol

Zjevně je  $w = |\text{Num}(x) - \text{Num}(y)| < 2^n$ . Číslo  $w$  můžeme rozložit na prvočísla

$$w = p_1^{i_1} p_2^{i_2} \cdots p_k^{i_k}$$

Chceme ukázat, že  $k \leq n - 1$ .

Předpokládejme, že  $k \geq n$ . Pak

$$w = p_1^{i_1} p_2^{i_2} \cdots p_k^{i_k} \geq p_1 p_2 \cdots p_n > 1 \cdot 2 \cdot 3 \cdots n = n! > 2^n$$

což je spor s tím, že  $w < 2^n$ .

Pravděpodobnost, že z množiny  $\{2, 3, \dots, n^2\}$  vybereme náhodně prvočísla, které je dělitelem  $w$  je maximálně

$$\frac{n-1}{\pi(n^2)} \leq \frac{n-1}{n^2 / \ln n^2} \leq \frac{\ln n^2}{n}$$

Například pro  $n = 10^{16}$  je to maximálně  $0.36892 \cdot 10^{-14}$ .



Modifikace protokolu:

- $C_1$  vygeneruje náhodně 10 prvočísel  $p_1, p_2, \dots, p_{10}$  a pošle

$$p_1, s_1, p_2, s_2, \dots, p_{10}, s_{10}$$

kde  $s_i = \text{Num}(x) \bmod p_i$ .

- $C_2$  spočítá  $t_i = \text{Num}(y) \bmod p_i$  pro  $i \in \{1, \dots, 10\}$ .

Pokud  $s_i \neq t_i$  pro nějaké  $i$ , vydá odpověď „ $x \neq y$ “, jinak vydá odpověď „ $x = y$ “.

Pro  $n = 10^{16}$  je třeba přenést maximálně 2560 bitů.

Protože 10 prvočísel je vybíráno náhodně, je pravděpodobnost chyby maximálně

$$\left(\frac{n-1}{\pi(n^2)}\right)^{10} \leq \left(\frac{\ln n^2}{n}\right)^{10} = \frac{2^{10} \cdot (\ln n)^{10}}{n^{10}}$$

Pro  $n = 10^{16}$  je pravděpodobnost chyby menší než  $0.4717 \cdot 10^{-141}$ .

Mnoho důležitých problémů je NP-úplných. U optimalizačních problémů často nepotřebujeme nalézt nejobtímnější řešení, ale stačí nám řešení, které je „dostatečně dobré“.

Algoritmům, které vrací takováto „dostatečně dobrá“ řešení, se říká **aproximační algoritmy**.

**Poznámka:** **Optimalizační problém** je problém, kde pro každý vstup  $w$  máme definovanou množinu  $S_w$  všech **přípustných řešení** a funkci  $c : S_w \rightarrow \mathbb{R}^+$ .

Úkolem je najít takové  $x \in S_w$ , pro které je hodnota  $c(x)$  minimální (resp. maximální).

**Příklady:** Hledání nejkratší cesty v grafu, určování maximálního toku v síti.

Řekněme, že řešíme problém, kde je úkolem hodnotu  $c(x)$  minimalizovat.

Řekněme, že pro vstup  $w$  vydá algoritmus řešení  $x$ , přičemž optimálním řešením je  $x^*$ . Zjevně platí  $c(x) \geq c(x^*)$ .

Jako **aproximační poměr** daného algoritmu označujeme funkci  $\rho(n)$ , která každému  $n$  přiřazuje maximální hodnotu poměru  $c(x)/c(x^*)$  pro všechny vstupy velikosti  $n$ .

**Poznámka:** Naopak u algoritmu, kde je cíle hodnotu  $c(x)$  maximalizovat, bychom uvažovali poměr  $c(x^*)/c(x)$ .

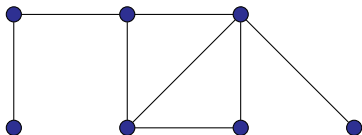
O algoritmu, který dosahuje aproximačního poměru  $\rho(n)$ , říkáme, že je to  $\rho(n)$ -**aproximační algoritmus**.

Pro řadu problémů jsou známy polynomiální algoritmy, kde je hodnota  $\rho(n)$  omezena:

- malou konstantou, např. 2-aproximační algoritmy, nebo
- pomalu rostoucí funkcí, např.  $\Theta(\log n)$ -aproximační algoritmy.

# Vrcholové pokrytí grafu

Mějme neorientovaný graf  $G = (V, E)$ . **Vrcholové pokrytí** (vertex-cover) grafu  $G$  je množina  $C \subseteq V$  taková, že pro každou hranu  $(u, v) \in E$  platí, že buď  $u \in C$  nebo  $v \in C$ .

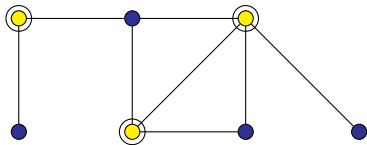


Úkolem je najít pro zadaný graf minimální vrcholové pokrytí.

**Poznámka:** Je známo, že se jedná o NP-úplný problém.

# Vrcholové pokrytí grafu

Mějme neorientovaný graf  $G = (V, E)$ . **Vrcholové pokrytí** (vertex-cover) grafu  $G$  je množina  $C \subseteq V$  taková, že pro každou hranu  $(u, v) \in E$  platí, že buď  $u \in C$  nebo  $v \in C$ .



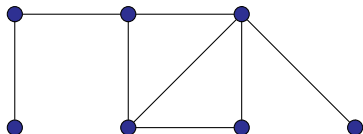
Úkolem je najít pro zadaný graf minimální vrcholové pokrytí.

**Poznámka:** Je známo, že se jedná o NP-úplný problém.

# Vrcholové pokrytí grafu

## APPROX-VERTEX-COVER( $G$ )

```
1  $C \leftarrow \emptyset$ 
2  $E' \leftarrow E$ 
3 while  $E' \neq \emptyset$ 
4     do vyber libovolnou hranu  $(u, v)$  z  $E'$ 
5          $C \leftarrow C \cup \{u, v\}$ 
6         odstraň z  $E'$  hrany incidentní s  $u$  nebo  $v$ 
7 return  $C$ 
```

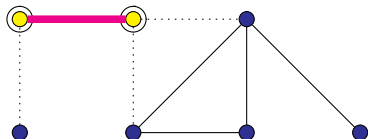




# Vrcholové pokrytí grafu

## APPROX-VERTEX-COVER( $G$ )

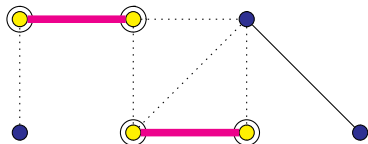
```
1  $C \leftarrow \emptyset$ 
2  $E' \leftarrow E$ 
3 while  $E' \neq \emptyset$ 
4     do vyber libovolnou hranu  $(u, v)$  z  $E'$ 
5          $C \leftarrow C \cup \{u, v\}$ 
6         odstraň z  $E'$  hrany incidentní s  $u$  nebo  $v$ 
7 return  $C$ 
```



# Vrcholové pokrytí grafu

## APPROX-VERTEX-COVER( $G$ )

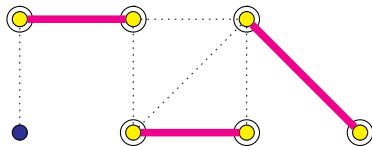
```
1  $C \leftarrow \emptyset$ 
2  $E' \leftarrow E$ 
3 while  $E' \neq \emptyset$ 
4     do vyber libovolnou hranu  $(u, v)$  z  $E'$ 
5          $C \leftarrow C \cup \{u, v\}$ 
6         odstraň z  $E'$  hrany incidentní s  $u$  nebo  $v$ 
7 return  $C$ 
```



# Vrcholové pokrytí grafu

## APPROX-VERTEX-COVER( $G$ )

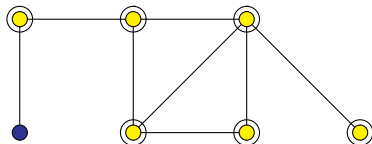
```
1  $C \leftarrow \emptyset$ 
2  $E' \leftarrow E$ 
3 while  $E' \neq \emptyset$ 
4     do vyber libovolnou hranu  $(u, v)$  z  $E'$ 
5          $C \leftarrow C \cup \{u, v\}$ 
6         odstraň z  $E'$  hrany incidentní s  $u$  nebo  $v$ 
7 return  $C$ 
```



# Vrcholové pokrytí grafu

## APPROX-VERTEX-COVER( $G$ )

```
1  $C \leftarrow \emptyset$ 
2  $E' \leftarrow E$ 
3 while  $E' \neq \emptyset$ 
4     do vyber libovolnou hranu  $(u, v)$  z  $E'$ 
5          $C \leftarrow C \cup \{u, v\}$ 
6         odstraň z  $E'$  hrany incidentní s  $u$  nebo  $v$ 
7 return  $C$ 
```



## Tvrzení

**APPROX-VERTEX-COVER** je polynomiální 2-aproximační algoritmus.

**Důkaz:** Označme  $A$  množinu všech hran vybraných v kroku 4. Jakékoliv vrcholové pokrytí musí obsahovat alespoň jeden z koncových vrcholů každé hrany z  $A$ .

Pokud je  $C^*$  minimální vrcholové pokrytí, pak  $|C^*| \geq |A|$ .

Na druhou stranu, zjevně platí  $|C| = 2 \cdot |A|$ , a tedy  $|C| \leq 2 \cdot |C^*|$ .

Je zjevné, že **APPROX-VERTEX-COVER** má polynomiální časovou složitost.

# Problém obchodního cestujícího

V **problému obchodního cestujícího** (traveling-salesman problem) je vstupem úplný neorientovaný graf s ohodnocením hran.

Úkolem je najít co nejkratší okružní cestu, která prochází každým vrcholem právě jednou.

(Délku cesty počítáme jako součet ohodnocení hran na této cestě.)

Problém obchodního cestujícího je **NP**-úplný.

(Jednoduchá redukce z problému Hamiltonovské kružnice.)

**Poznámka:** V problému Hamiltonovské kružnice je vstupem neorientovaný graf  $G$  a otázka je, zda v grafu  $G$  existuje cyklus procházející každým vrcholem právě jednou.

## Věta

Pokud  $P \neq NP$ , pak pro žádnou konstantu  $\rho \geq 1$  neexistuje polynomiální  $\rho$ -aproximační algoritmus řešící problém obchodního cestujícího.

**Důkaz:** Předpokládejme, že by pro nějaké  $\rho$  takový algoritmus existoval. Ukážeme, že pak by existoval polynomiální algoritmus pro problém Hamiltonovské kružnice.

Problém Hamiltonovské kružnice je  $NP$ -úplný, nalezení polynomiálního algoritmu by znamenalo, že  $P = NP$ .

# Problém obchodního cestujícího

Nechť  $G = (V, E)$  je instance problému Hamiltonovské kružnice.

Instanci problému obchodního cestujícího  $G' = (V', E')$  sestrojíme tak, že  $V' = V$  a  $E'$  obsahuje všechny možné hrany, kterým přiřadíme ohodnocení

$$c(u, v) = \begin{cases} 1 & \text{pokud } (u, v) \in E \\ \rho \cdot |V| + 1 & \text{jinak} \end{cases}$$

Jestliže graf  $G$  obsahuje Hamiltonovskou kružnici, pak v  $G'$  existuje okružní cesta délky  $|V|$ .

Jakákoliv okružní cesta v  $G'$ , která obsahuje hranu, která není v  $G$ , má délku větší než  $\rho \cdot |V|$ .

Pokud v  $G$  Hamiltonovská kružnice existuje,  $\rho$ -aproximační algoritmus musí nějakou takovou kružnici vrátit jako výslednou okružní cestu.



# Problém obchodního cestujícího

Uvažujme nyní variantu TSP, kde musí každá trojice vrcholů  $u, v, w$  splňovat **trojúhelníkovou nerovnost**

$$c(u, w) \leq c(u, v) + c(v, w).$$

V praxi je tato podmínka často splněna.

Problém obchodního cestujícího zůstává NP-úplný i za této podmínky.

Existuje však pro něj polynomiální 1.5-aproximační algoritmus.

My si ukážeme jednodušší 2-aproximační algoritmus.

# Problém obchodního cestujícího

Předpokládejme, že instancí TSP je graf  $G$ , a že optimálním řešením (které neznáme) je cesta  $H^*$ .

- Nejprve najdeme v grafu  $G$  minimální kostru  $T$ .

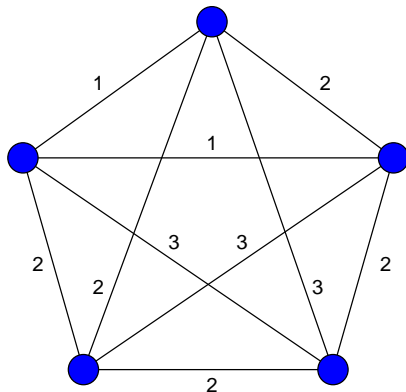
**Poznámka:** Zjevně platí  $c(T) \leq c(H^*)$ , neboť odstraněním libovolné hrany z  $H^*$  vznikne kostra grafu  $G$ .

- Vytvoříme okružní cestu  $W$ , která odpovídá průchodu stromem  $T$  do hloubky.

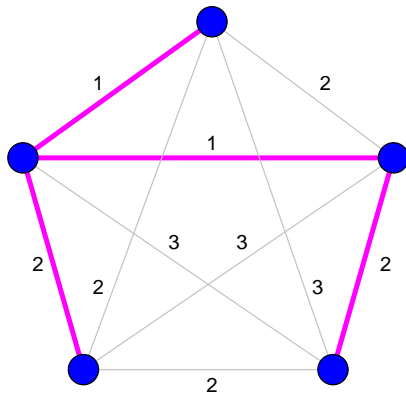
**Poznámka:** Každou hranu  $T$  procházíme dvakrát. Platí tedy  $c(W) = 2 \cdot c(T) \leq 2 \cdot |H^*|$ .

- Z cesty  $W$  vypustíme vrcholy, které jsme již navštívili. Díky trojúhelníkové nerovnosti se tím může délka cesty pouze snížit.

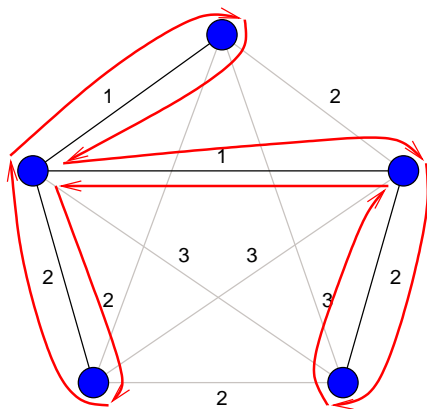
# Problém obchodního cestujícího



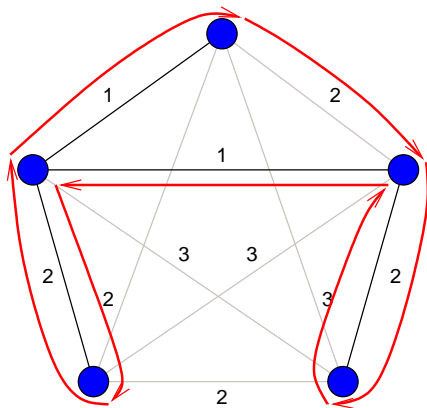
# Problém obchodního cestujícího



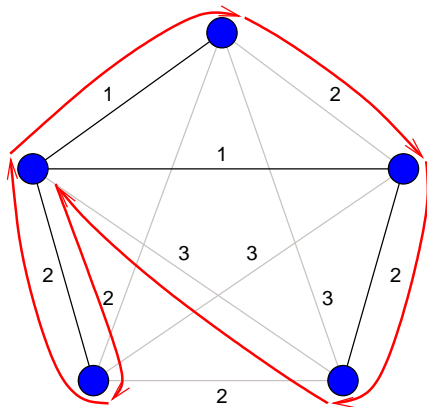
# Problém obchodního cestujícího



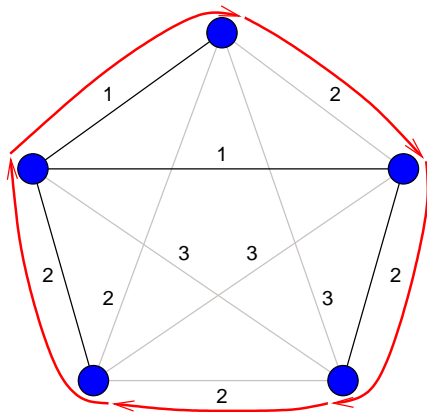
# Problém obchodního cestujícího



# Problém obchodního cestujícího



# Problém obchodního cestujícího





# Problém obchodního cestujícího

