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

30. května 2007

Přednášející

Jméno: Zdeněk Sawa

E-mail: zdenek.sawa@vsb.cz

Místnost: A1024

Tel.: 4437

Jméno: Martin Kot

E-mail: martin.kot@vsb.cz

Místnost: A1024

Tel.: 4437

Požadavky

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)

WWW stránky

Webové stránky k předmětu naleznete na adrese:

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

Na těchto stránkách najdete:

- Informace o předmětu
- Učební text
- Slidy z přednášek
- Zadání příkladů na cvičení
- Zadání zápočtových příkladů
- Zadání prémiových příkladů
- Aktuální informace

Poznámky

- 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.

Prémiové příklady

- Body dostane vždy první student v ročníku, který daný příklad vyřeší.
- Řešení posílejte e-mailem přednášejícím, ne cvičícím:
  - Zdeněk Sawa (zdenek.sawa@vsb.cz)
  - Martin Kot (martin.kot@vsb.cz)
- Každý student může tímto způsobem získat 15 bodů navíc nejvýše jednou.

Náplň předmětu

V tomto předmětu se budeme zabývat dvěma oblastmi teoretické informatiky:

- **Teorií formálních jazyků**
- **Výpočetní složitostí**

Plán přednášek

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

Plán přednášek

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

Formální jazyky

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)

Motivace 2: Lexikální a syntaktická analýza

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.

Motivace 2: Lexikální a syntaktická analýza

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.?

Motivace 2: Lexikální a syntaktická analýza

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.

Základní pojmy

Co mají všechny dosud zmíněné problémy společného?

- Pracujeme se sekvencemi **znaků** (říkáme též **symbolů**).
- Znaky 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**.

Abeceda a slovo

Definice

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

Poznámka: Abeceda se často označuje řeckým písmenem Σ (velké sigma).

Definice

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

Příklad 1:

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

Slova v abecedě Σ: $\mathtt{AHOJ}$ $\mathtt{ABRACADABRA}$ $\mathtt{ERROR}$

Abeceda a slovo

Příklad 2:

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

Slovo v abecedě Σ: $\mathtt{HELLO}\sqcup\mathtt{WORLD}$

Příklad 3:

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

Slova v abecedě Σ: 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*

Abeceda a slovo

Příklad 6:

Abeceda Σ je množina všech ASCII znaků.

Příklad slova:

```
class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

```
class␣HelloWorld␣{ ↩ ␣␣␣␣public␣static␣void␣main(Str···
```

Délka slova

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 ε (epsilon).

(Pozn.: Někteří autoři používají pro označení prázdného slova místo ε řecké písmeno λ (lambda).)

$$|\varepsilon| = 0$$

Zřetězení slov

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.

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

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:

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

Zjevně pro libovolná slova v a w platí:

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

Pro libovolné slovo w také platí:

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

Jazyk

Množinu všech slov tvořených symboly z abecedy Σ označujeme Σ^*.

Definice

(Formální) jazyk v abecedě Σ je nějaká libovolná podmnožina množiny Σ^*.

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

Příklad 2: Množina všech syntakticky správných programů v jazyce Java je jazyk v abecedě tvořené množinou všech Unicode znaků.

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

Jazyk

Příklad 4: Uvažujme abecedu Σ 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.

Doplněk – $\overline{L}$ je jazyk tvořený těmi slovy ze Σ^*, 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 Σ.

Množinové operace na jazycích

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`.

Zřetězení jazyků

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:

$$\begin{aligned} L_1 &= \{abb, ba\} \\ L_2 &= \{a, ab, bbb\} \end{aligned}$$

Jazyk $L_1 \cdot L_2$ obsahuje slova:

$$abba \qquad abbab \qquad abbbbb \qquad baa \qquad baab \qquad babbb$$

Zřetězení jazyků

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

Zřetězení jazyků

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

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:

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

Zřetězení jazyků

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

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

kde

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

Iterace jazyka

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:

$$aaaaaa \quad aaaab \quad aabaa \quad aabb \quad baaaa \quad baab \quad bbaa \quad bbb$$

Definujeme

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

Iterace jazyka

Induktivní definice pro libovolné $k \geq 0$:

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

Definice

Iterace jazyka L je jazyk

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

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

Příklad: $L = \{aa, b\}$

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

Iterace jazyka

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:

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

Řešení předchozího příkladu bychom tedy také mohli zapsat stručněji:

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

Zrcadlový obraz

Zrcadlový obraz slova w je slovo w zapsané „pozpátku“.

Zrcadlový obraz slova w značíme w^R.

Příklad: $w = \texttt{AHOJ}$ $\quad$ $w^R = \texttt{JOHA}$

Zrcadlový obraz jazyka L je jazyk tvořený zrcadlovými obrazy všech slov z jazyka L.

Zrcadlový obraz jazyka L značíme L^R.

$$L^R = \{w^R \mid w \in L\}$$

Příklad: $L = \{ab, baaba, aaab\}$

$L^R = \{ba, abaab, baaa\}$

Práce s formálními jazyky

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.

Práce s formálními jazyky

V teorii formálních jazyků se používají především následující dva přístupy:

- Popsat (idealizovaný) stoj, zařízení, algoritmus, který rozpozná slova patřící do daného jazyka – vede k použití tzv. **automatů**.

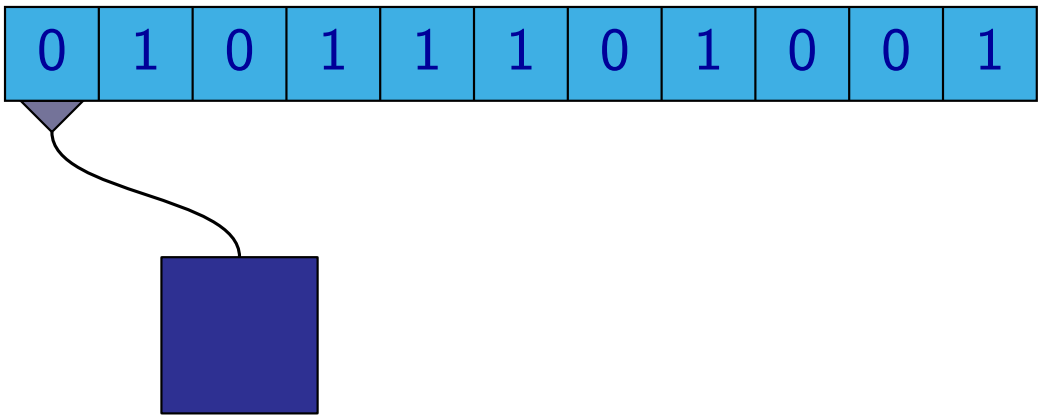
- Popsat postup, jak mechanicky generovat všechna možná slova patřící do daného jazyka – vede k tzv. **gramatikám** a **regulárním výrazům**. (budeme se jim věnovat později)

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.

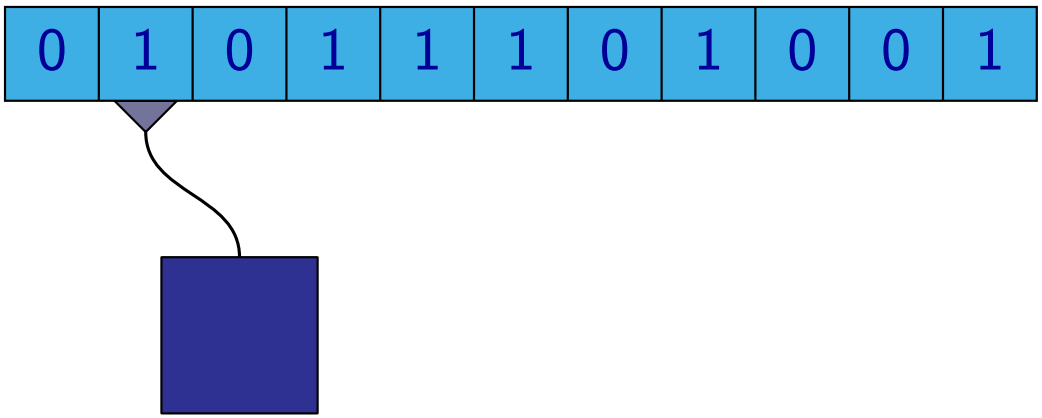


Rozpoznávání jazyka

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.

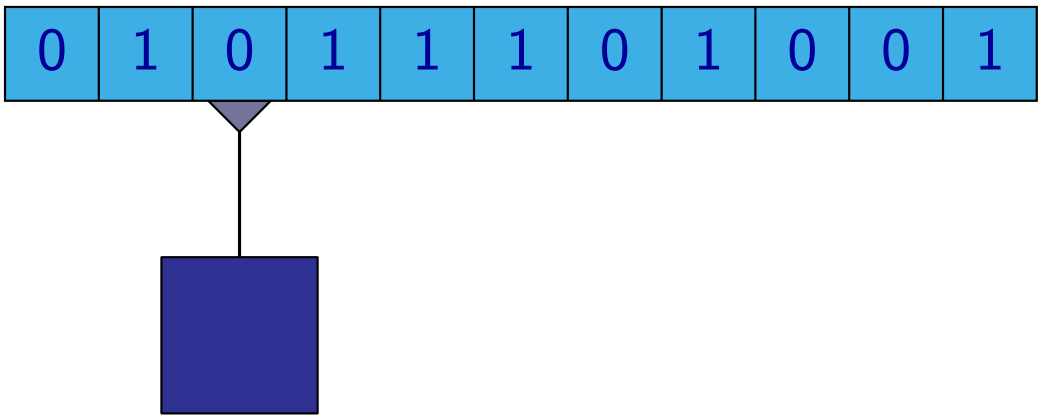


Rozpoznávání jazyka

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.

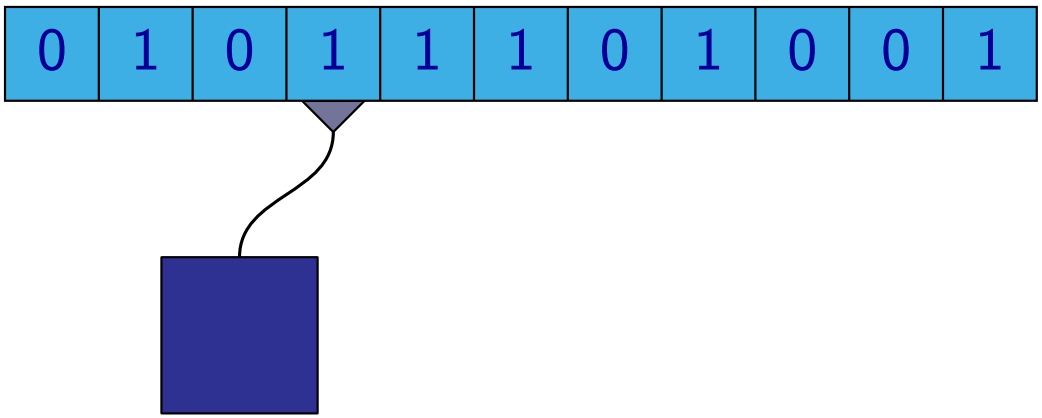


Rozpoznávání jazyka

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.

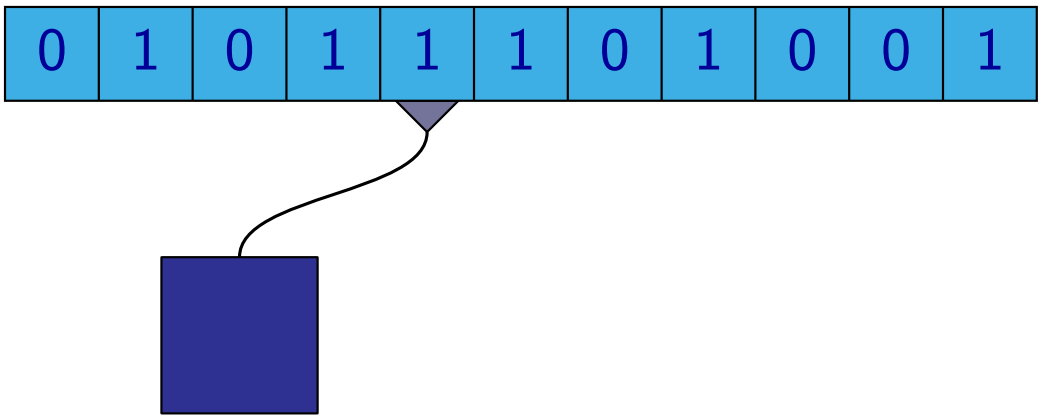


Rozpoznávání jazyka

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.

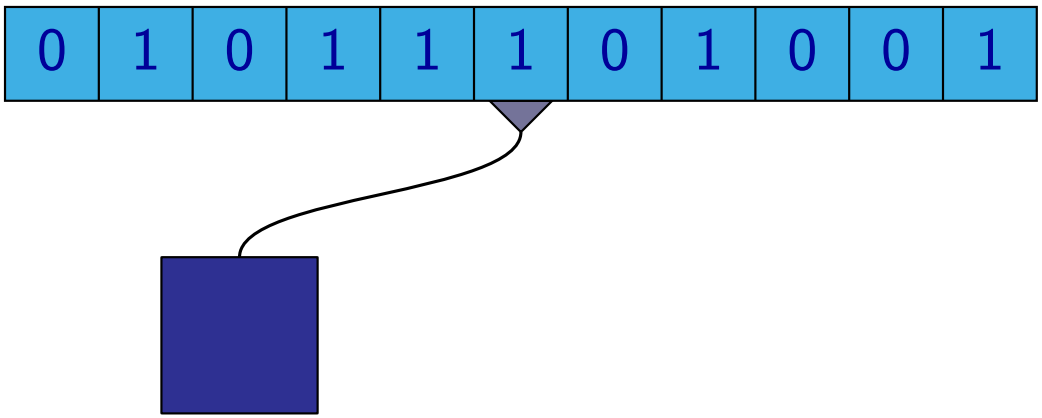


Rozpoznávání jazyka

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.

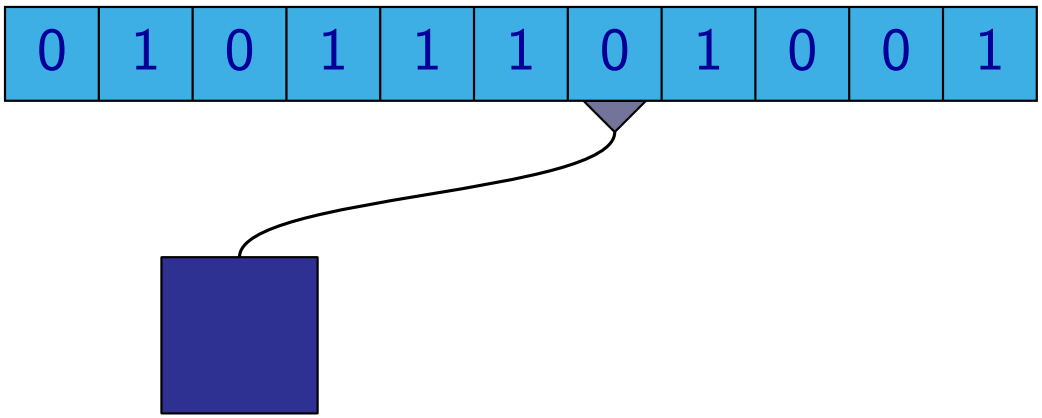


Rozpoznávání jazyka

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.

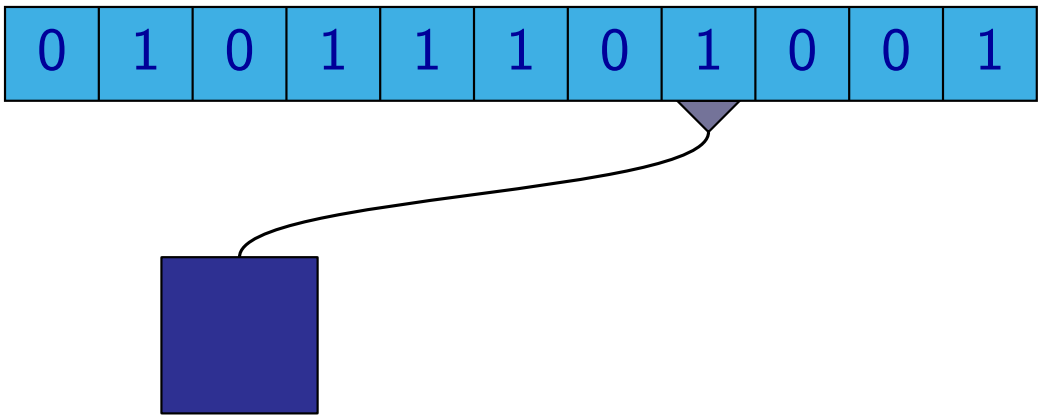


Rozpoznávání jazyka

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.

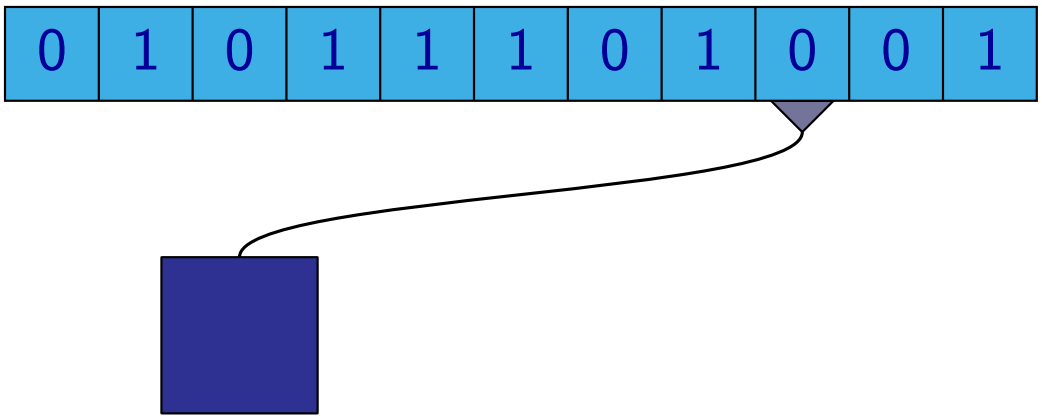


Rozpoznávání jazyka

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.

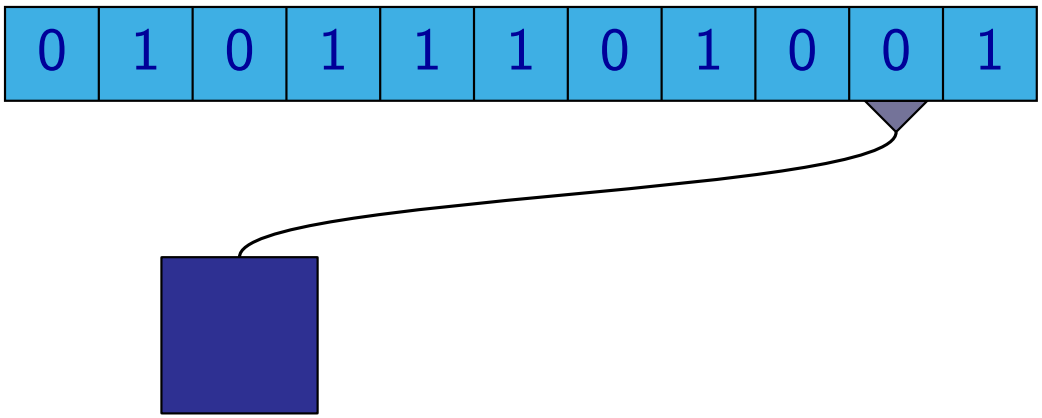


Rozpoznávání jazyka

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.

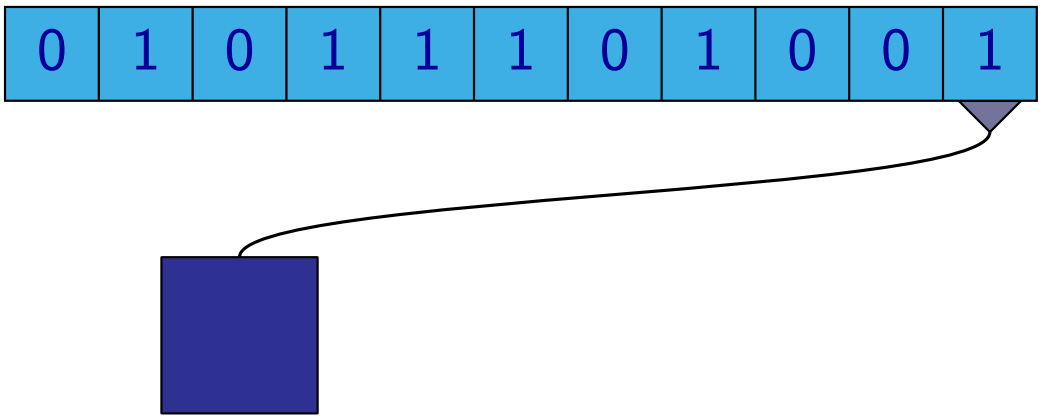


Rozpoznávání jazyka

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.

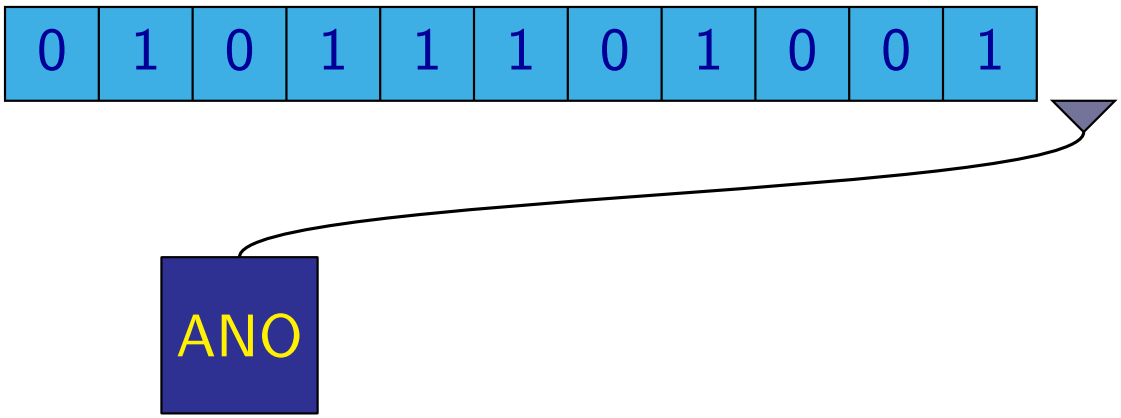


Rozpoznávání jazyka

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

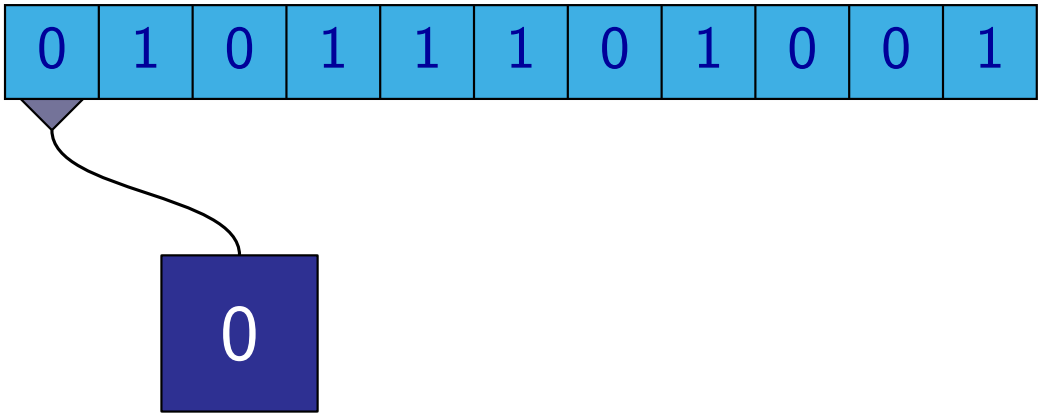
Chtěli bychom rozpoznávat jazyk L, který je tvořen slovy, ve kterých se vyskytuje sudý počet symbolů 1.

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



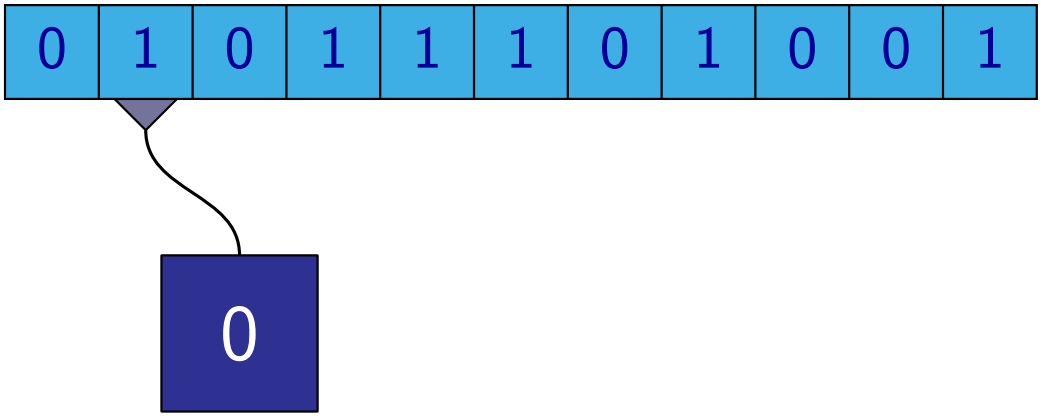
Rozpoznávání jazyka

První nápad: Počítat počet výskytů symbolů 1.



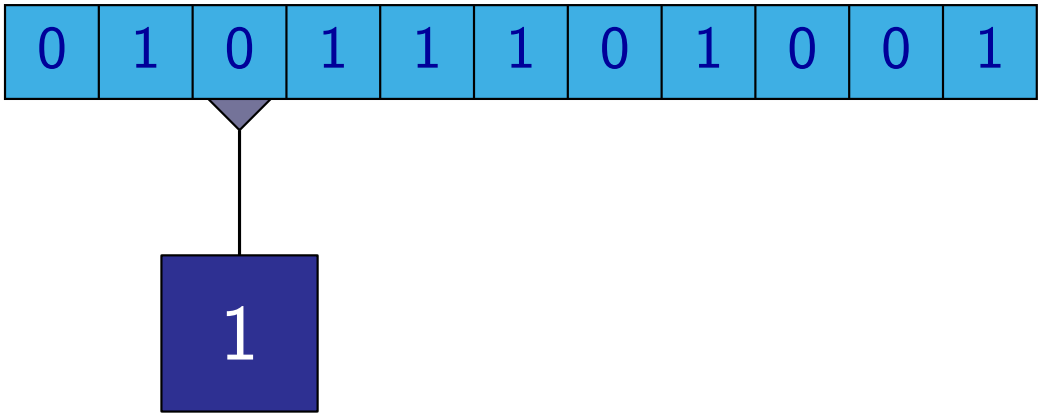
Rozpoznávání jazyka

První nápad: Počítat počet výskytů symbolů 1.



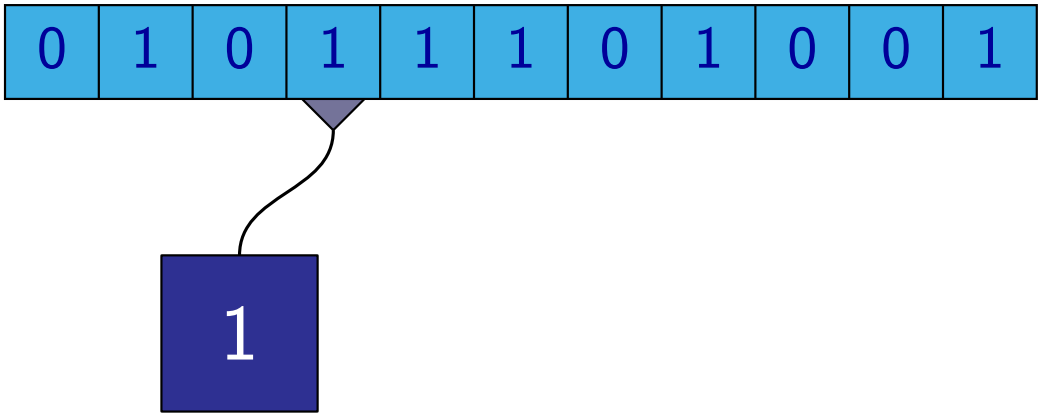
Rozpoznávání jazyka

První nápad: Počítat počet výskytů symbolů 1.



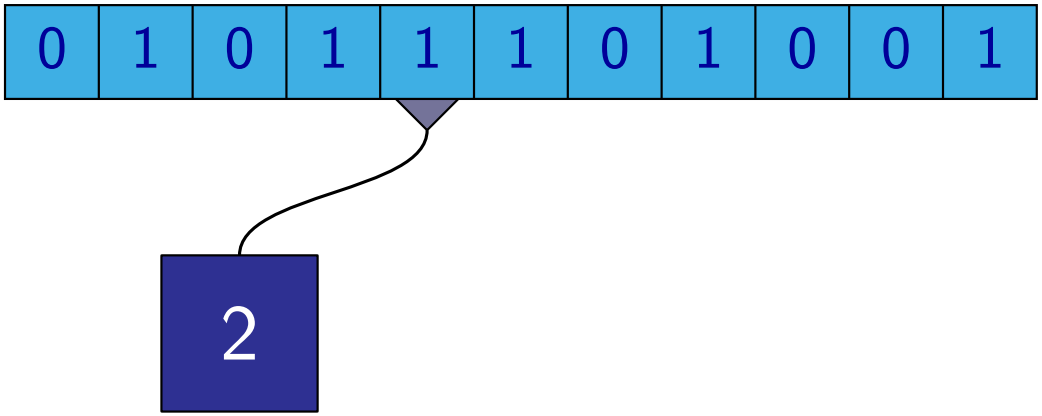
Rozpoznávání jazyka

První nápad: Počítat počet výskytů symbolů 1.



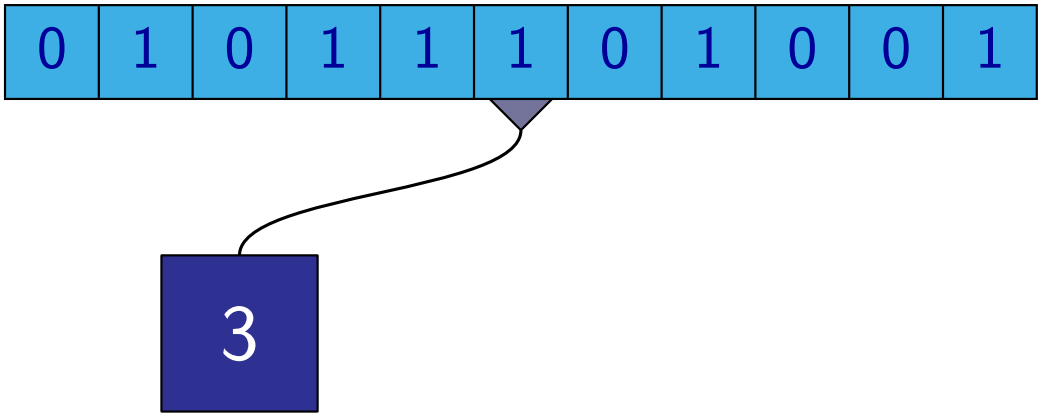
Rozpoznávání jazyka

První nápad: Počítat počet výskytů symbolů 1.



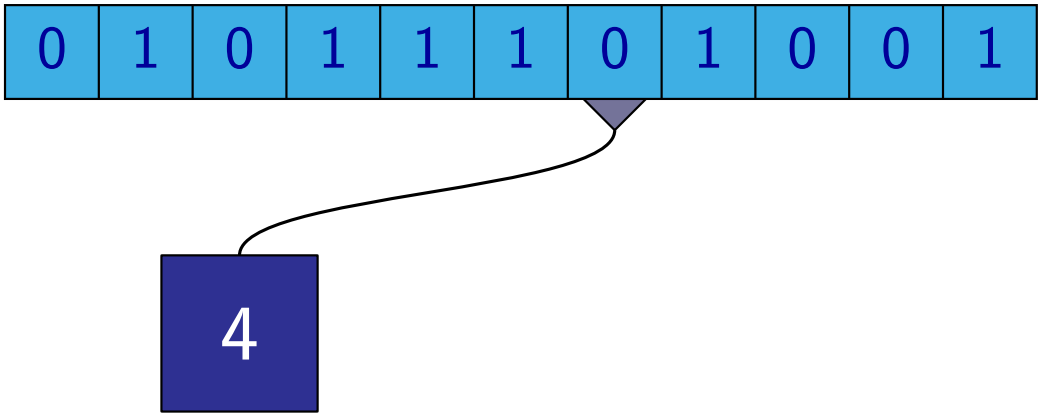
Rozpoznávání jazyka

První nápad: Počítat počet výskytů symbolů 1.



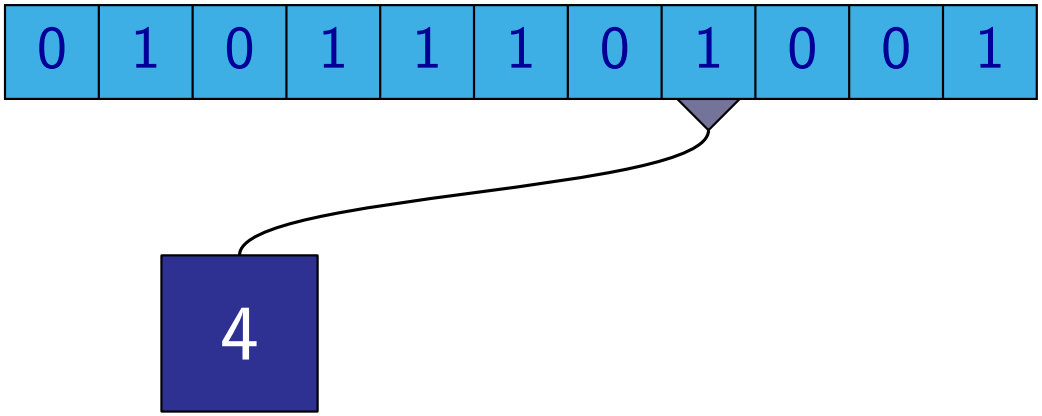
Rozpoznávání jazyka

První nápad: Počítat počet výskytů symbolů 1.



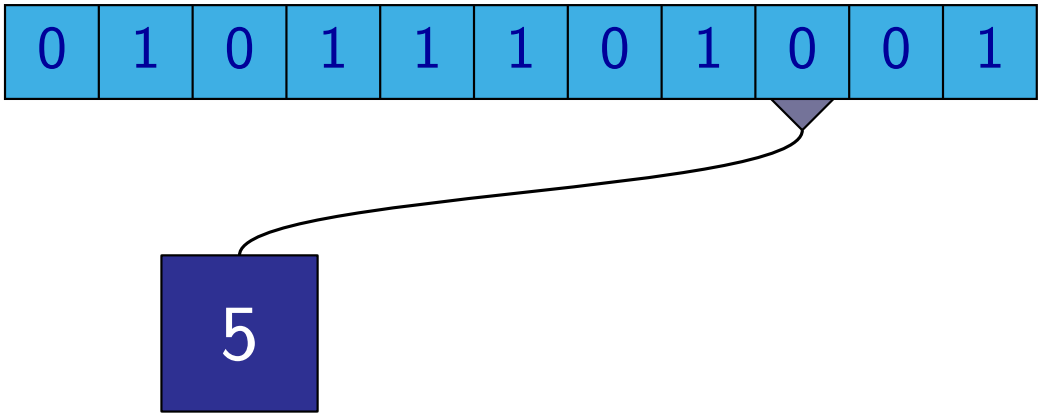
Rozpoznávání jazyka

První nápad: Počítat počet výskytů symbolů 1.



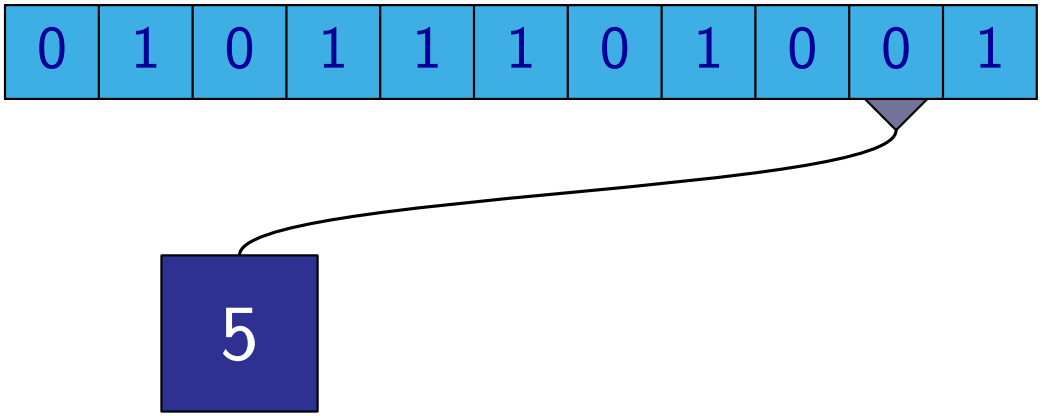
Rozpoznávání jazyka

První nápad: Počítat počet výskytů symbolů 1.



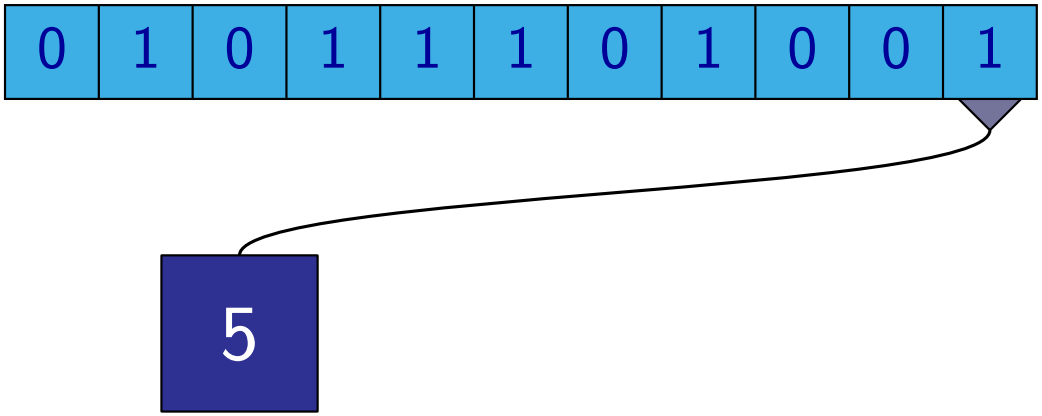
Rozpoznávání jazyka

První nápad: Počítat počet výskytů symbolů 1.



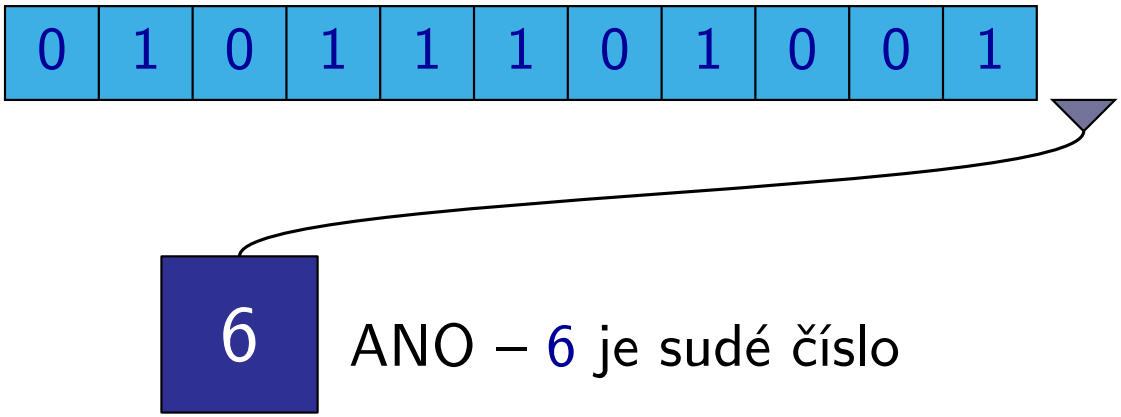
Rozpoznávání jazyka

První nápad: Počítat počet výskytů symbolů 1.



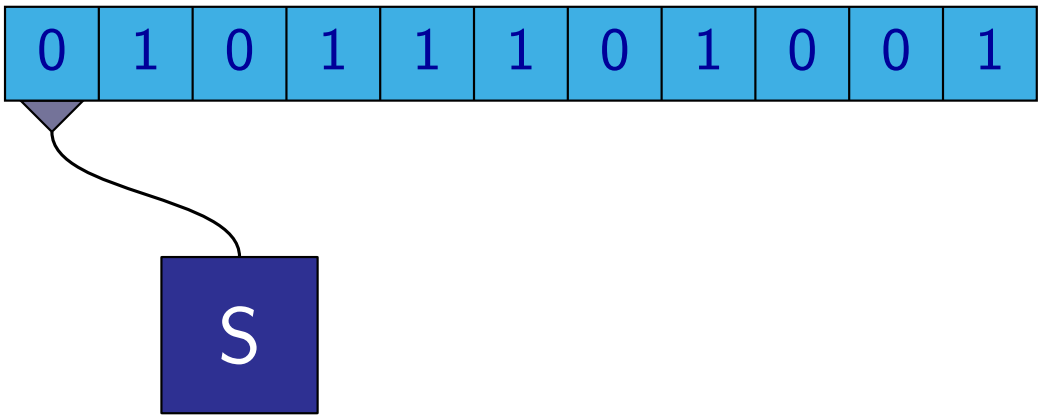
Rozpoznávání jazyka

První nápad: Počítat počet výskytů symbolů 1.



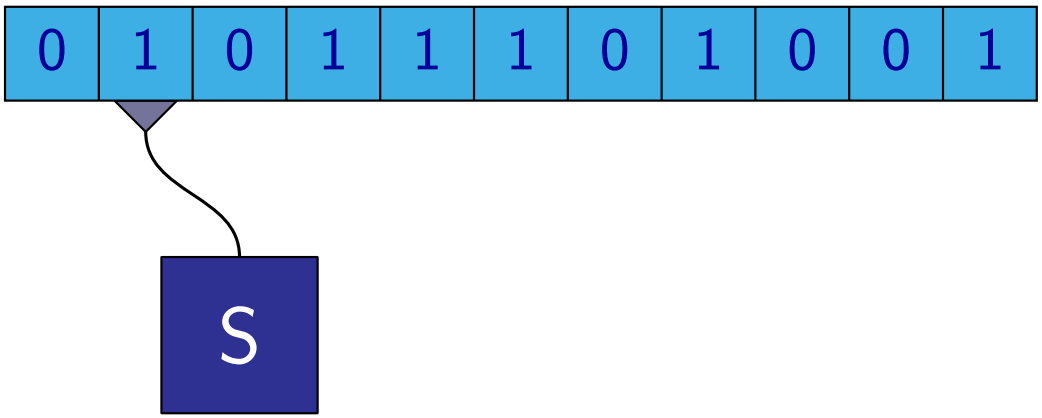
Rozpoznávání jazyka

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).



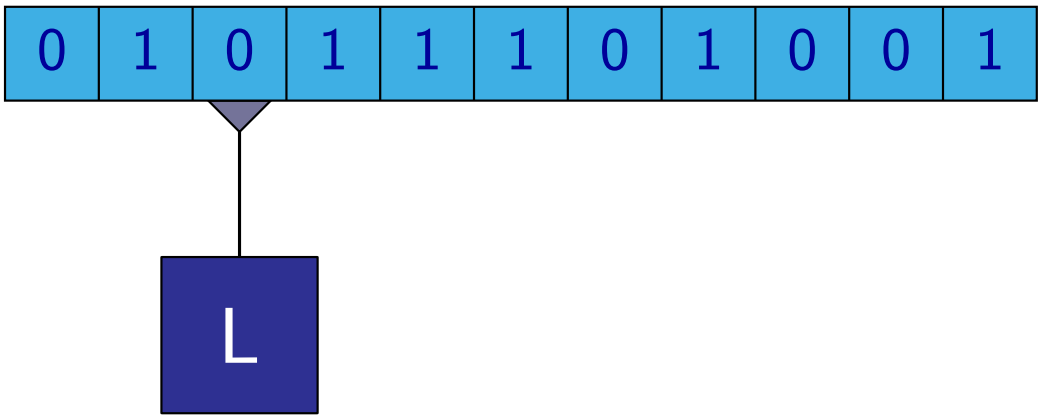
Rozpoznávání jazyka

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).



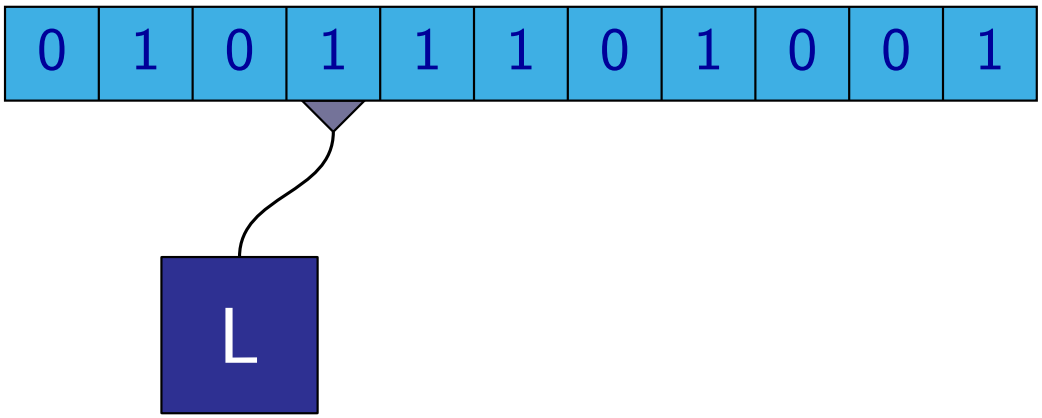
Rozpoznávání jazyka

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).



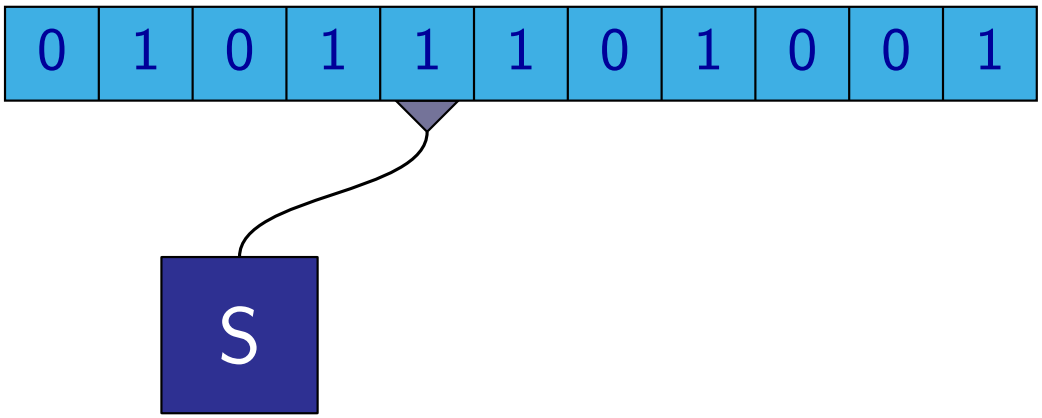
Rozpoznávání jazyka

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).



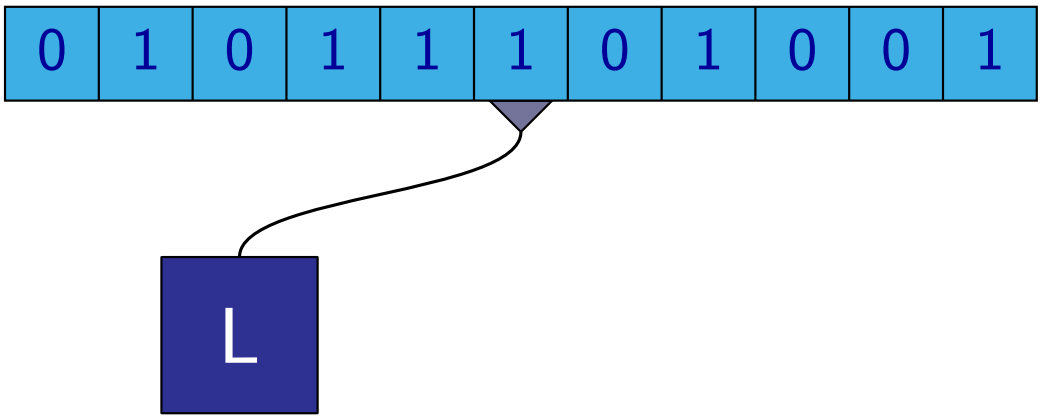
Rozpoznávání jazyka

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).



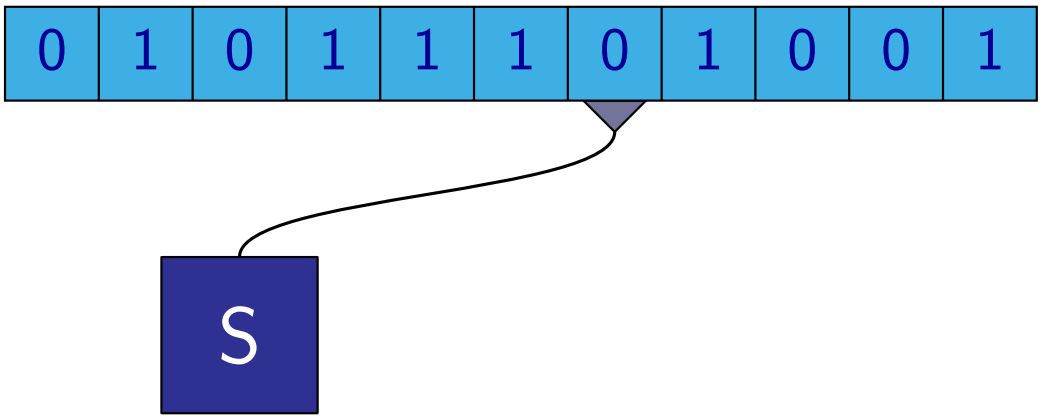
Rozpoznávání jazyka

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).



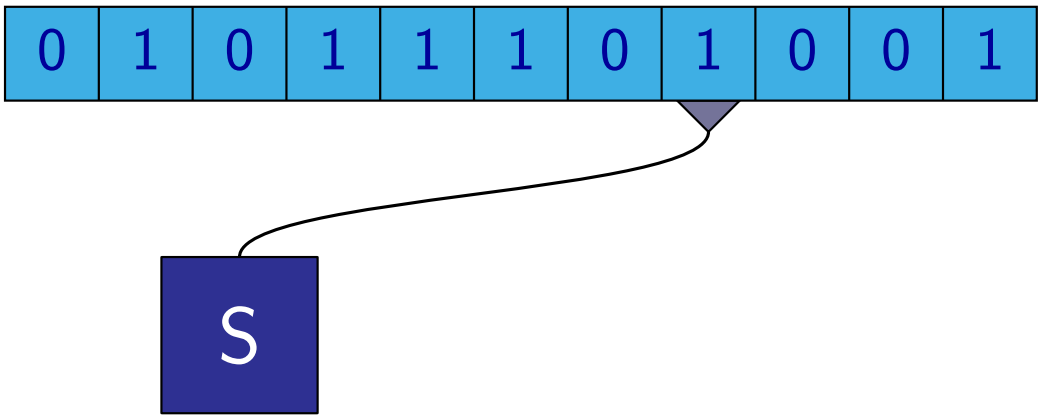
Rozpoznávání jazyka

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).



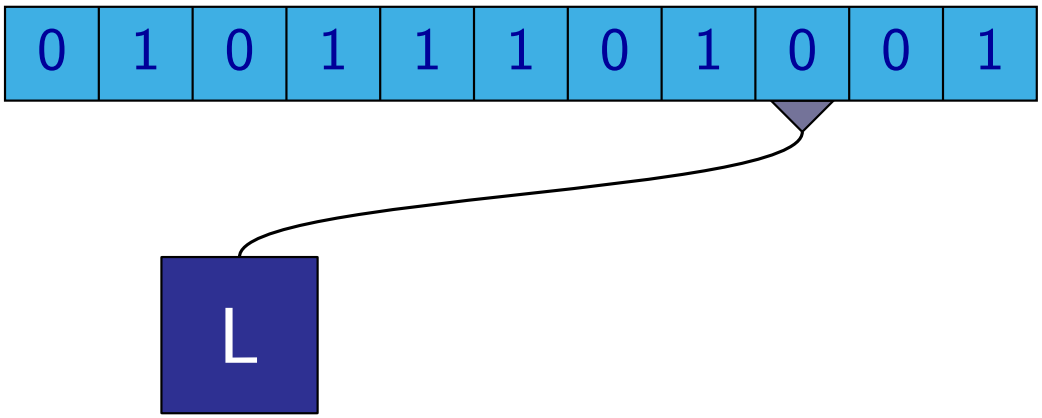
Rozpoznávání jazyka

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).



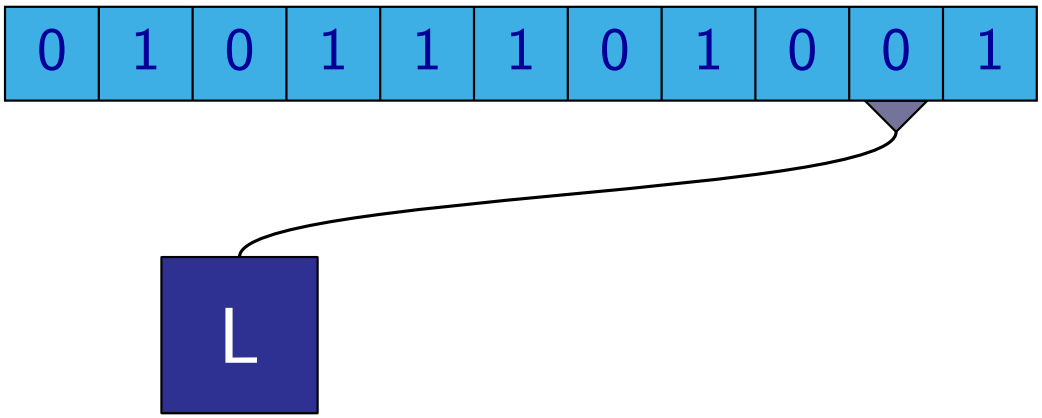
Rozpoznávání jazyka

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).



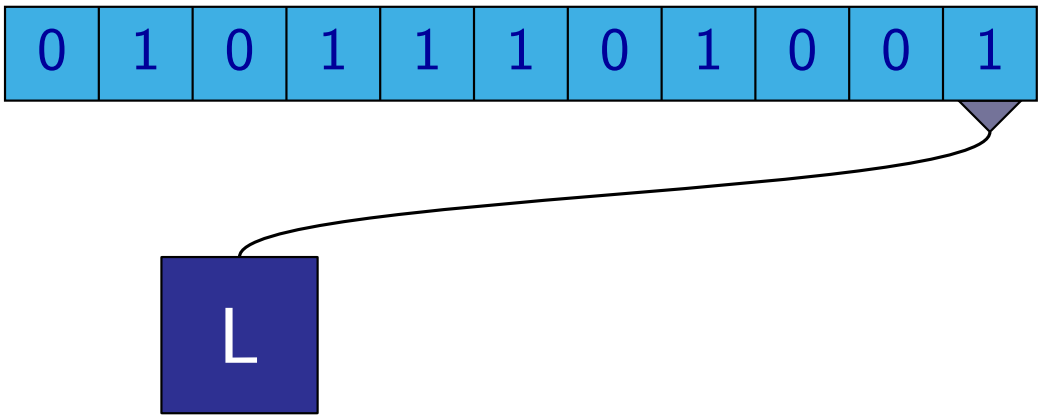
Rozpoznávání jazyka

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).



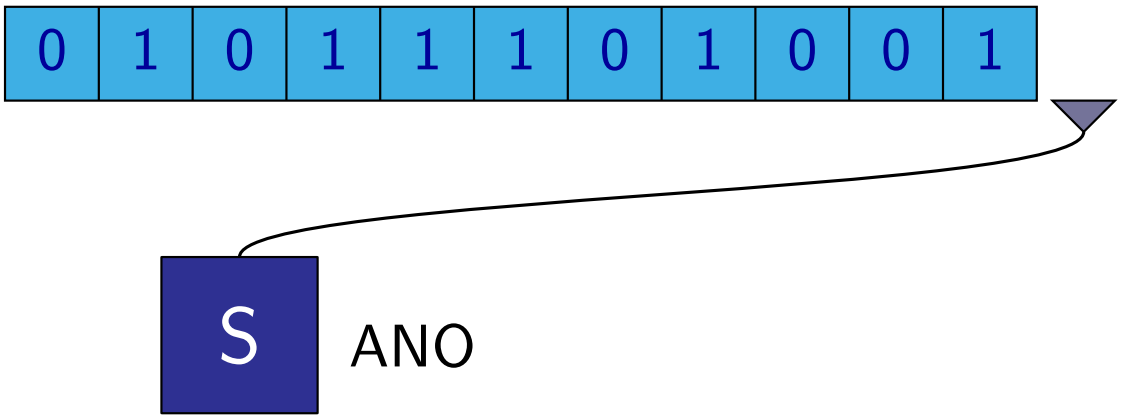
Rozpoznávání jazyka

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).



Rozpoznávání jazyka

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).



Rozpoznávání jazyka

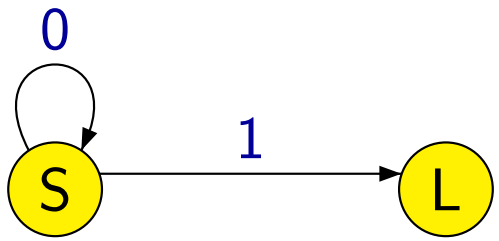
Chovaní tohoto zařízení můžeme popsat grafem:

Rozpoznávání jazyka

Chovaní tohoto zařízení můžeme popsat grafem:

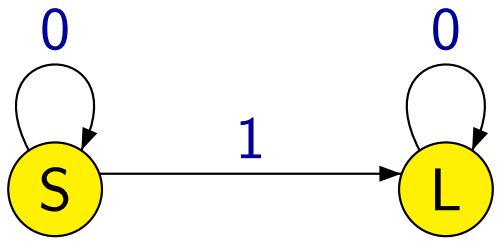
Rozpoznávání jazyka

Chovaní tohoto zařízení můžeme popsat grafem:



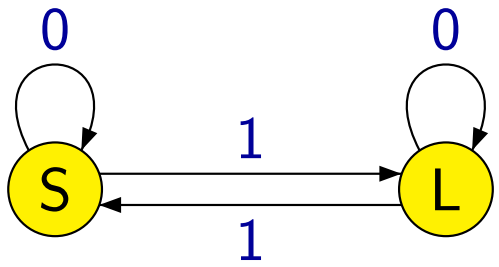
Rozpoznávání jazyka

Chovaní tohoto zařízení můžeme popsat grafem:



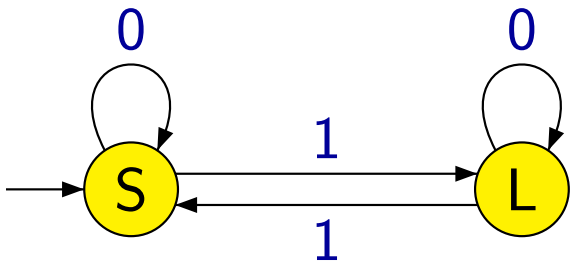
Rozpoznávání jazyka

Chovaní tohoto zařízení můžeme popsat grafem:



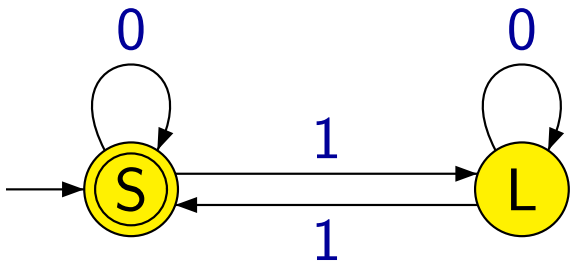
Rozpoznávání jazyka

Chovaní tohoto zařízení můžeme popsat grafem:



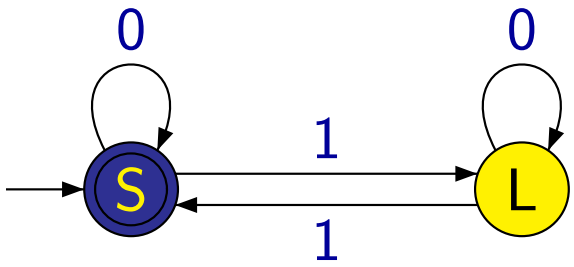
Rozpoznávání jazyka

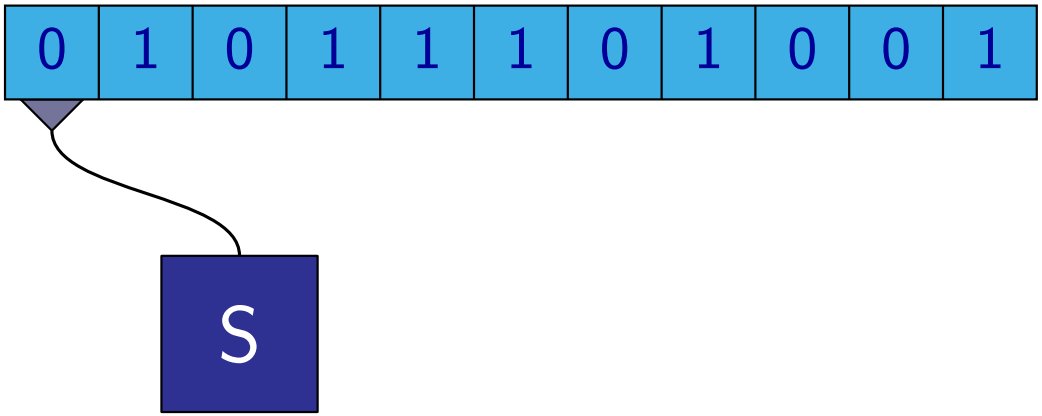
Chovaní tohoto zařízení můžeme popsat grafem:



Rozpoznávání jazyka

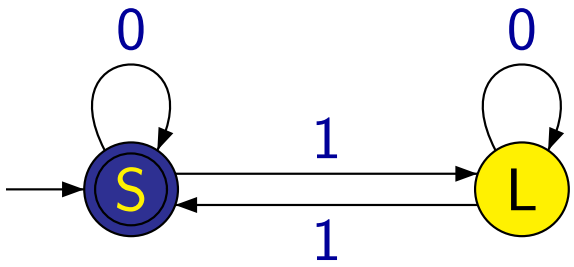
Chovaní tohoto zařízení můžeme popsat grafem:

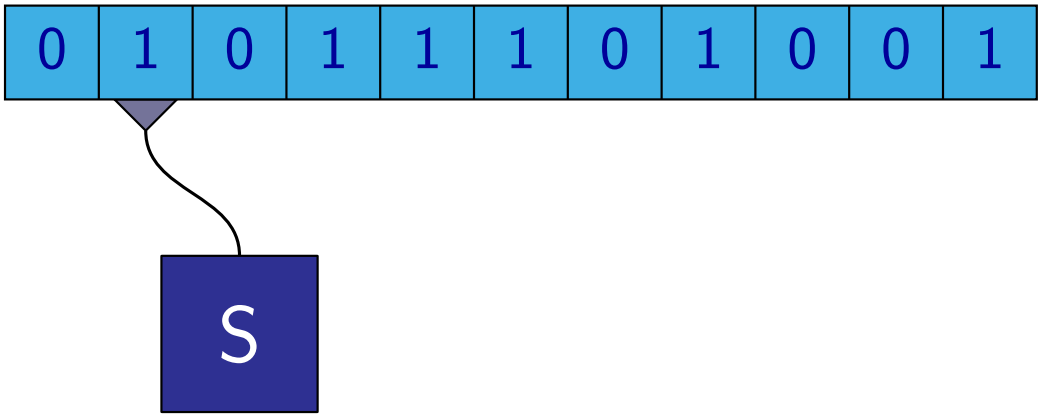




Rozpoznávání jazyka

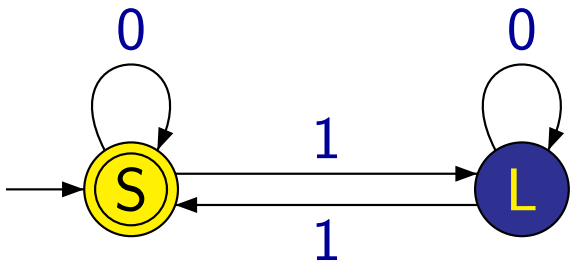
Chovaní tohoto zařízení můžeme popsat grafem:

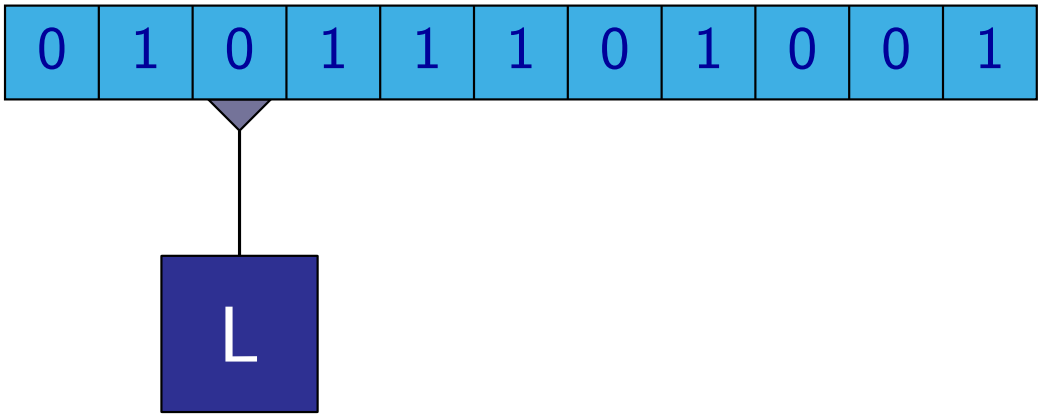




Rozpoznávání jazyka

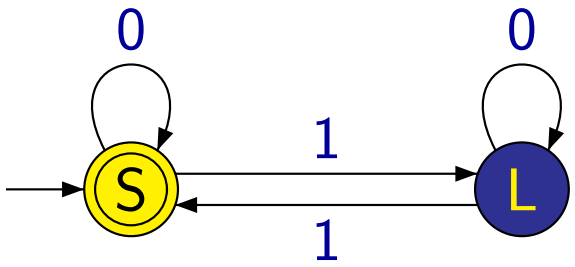
Chovaní tohoto zařízení můžeme popsat grafem:

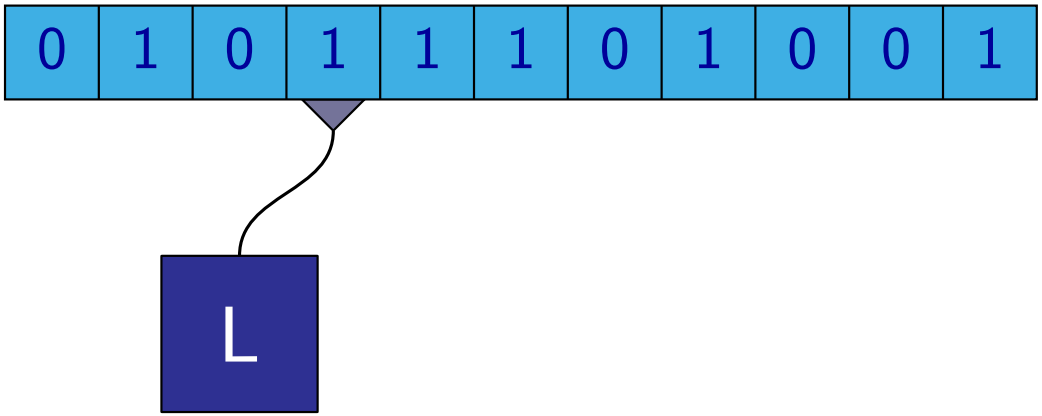




Rozpoznávání jazyka

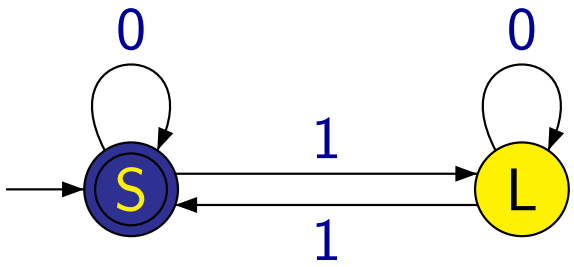
Chovaní tohoto zařízení můžeme popsat grafem:

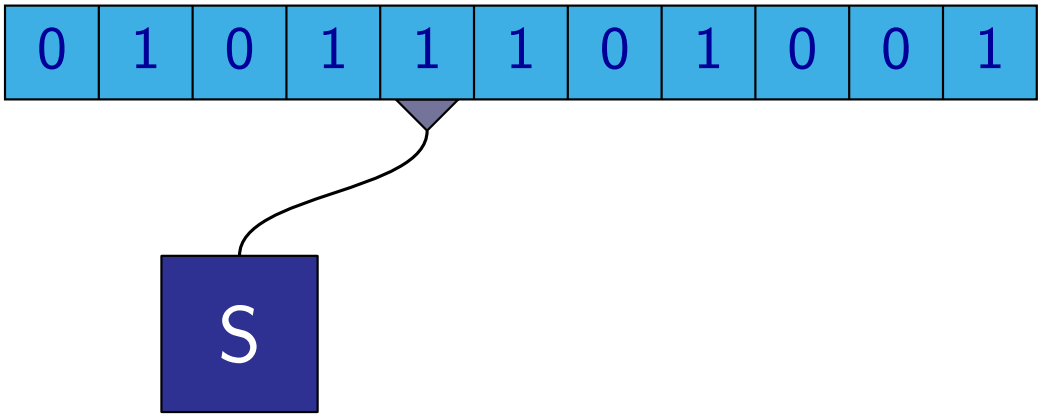




Rozpoznávání jazyka

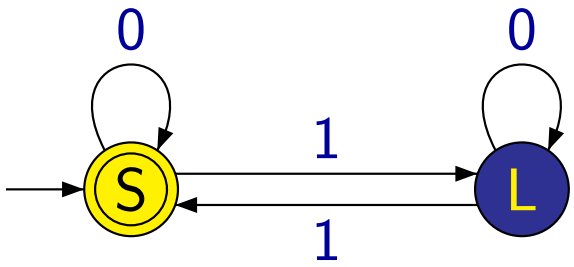
Chovaní tohoto zařízení můžeme popsat grafem:

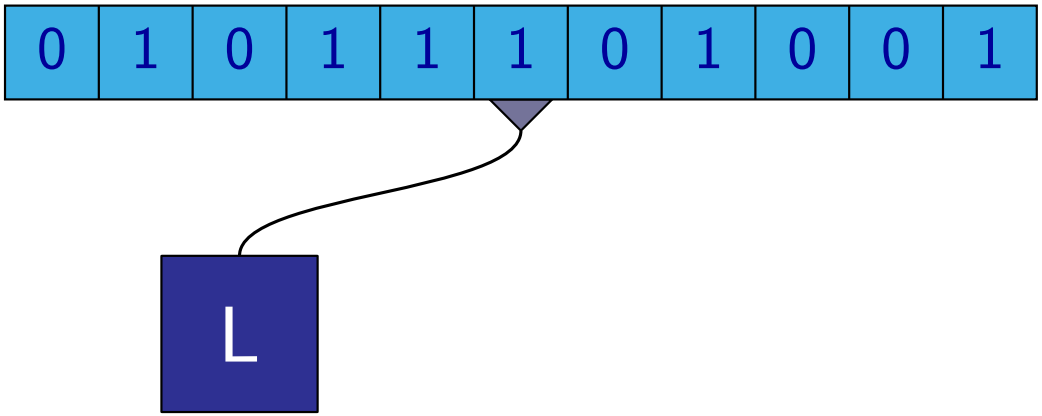




Rozpoznávání jazyka

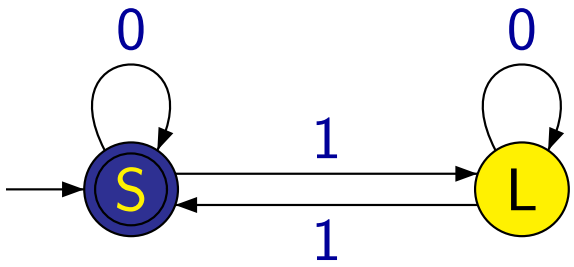
Chovaní tohoto zařízení můžeme popsat grafem:

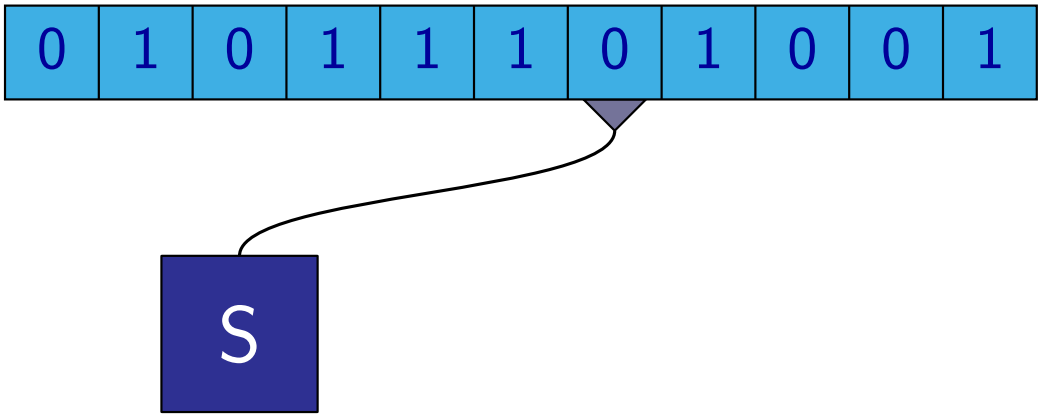




Rozpoznávání jazyka

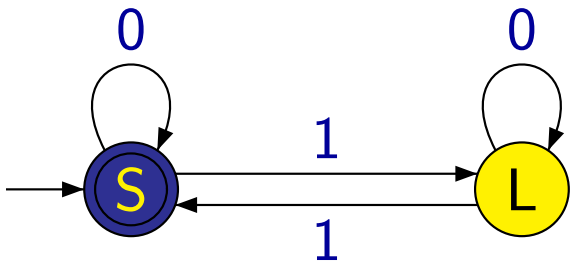
Chovaní tohoto zařízení můžeme popsat grafem:

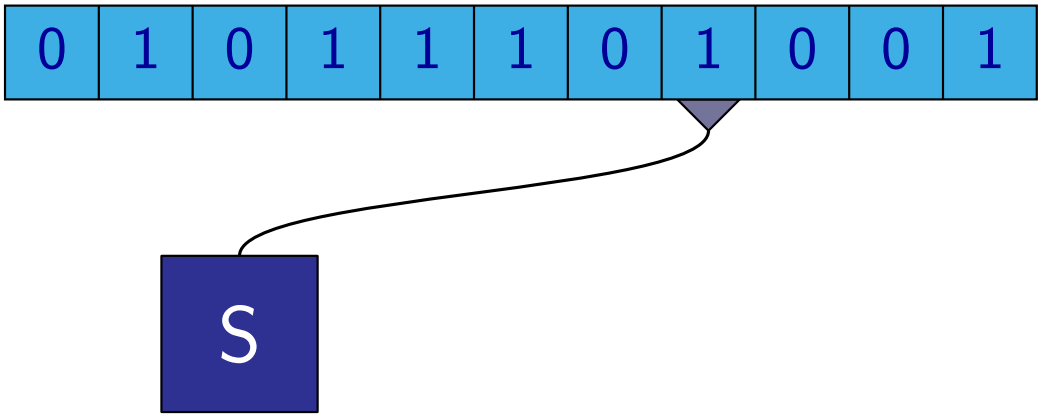




Rozpoznávání jazyka

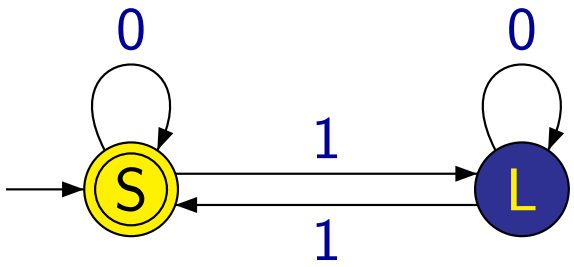
Chovaní tohoto zařízení můžeme popsat grafem:

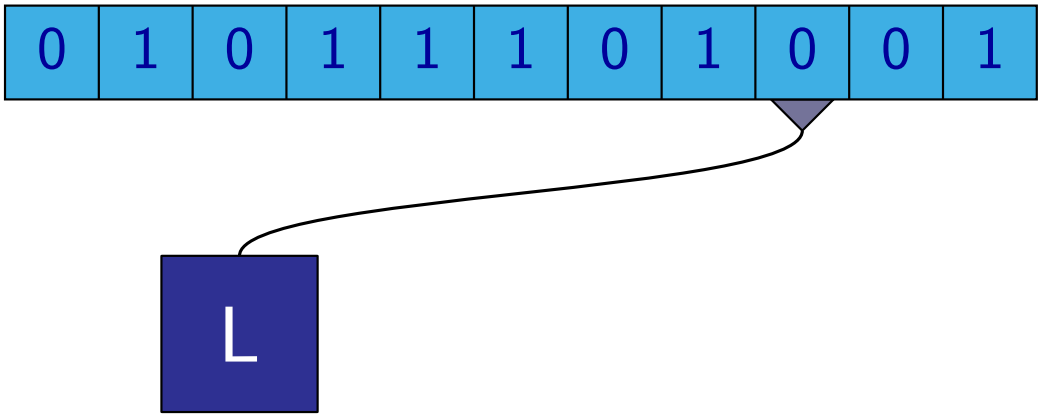




Rozpoznávání jazyka

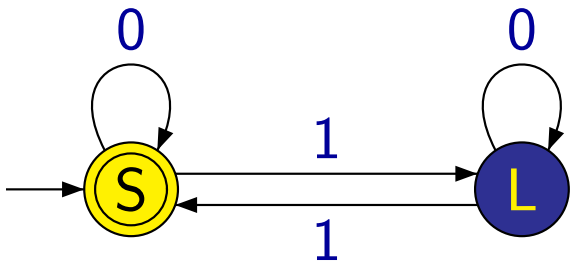
Chovaní tohoto zařízení můžeme popsat grafem:

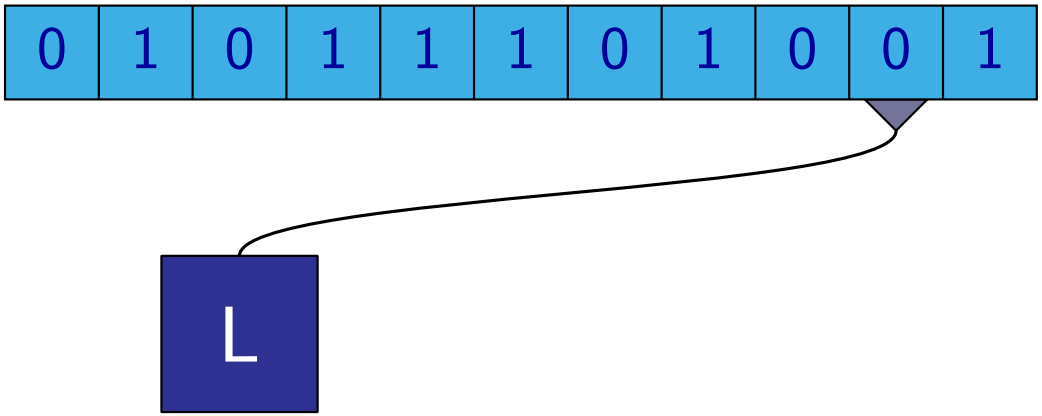




Rozpoznávání jazyka

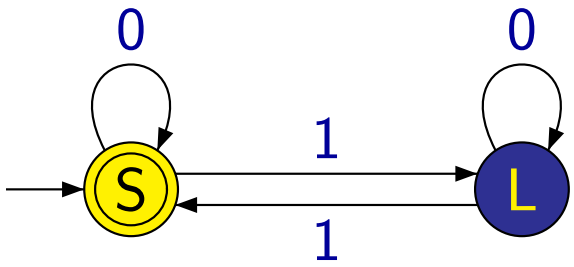
Chovaní tohoto zařízení můžeme popsat grafem:

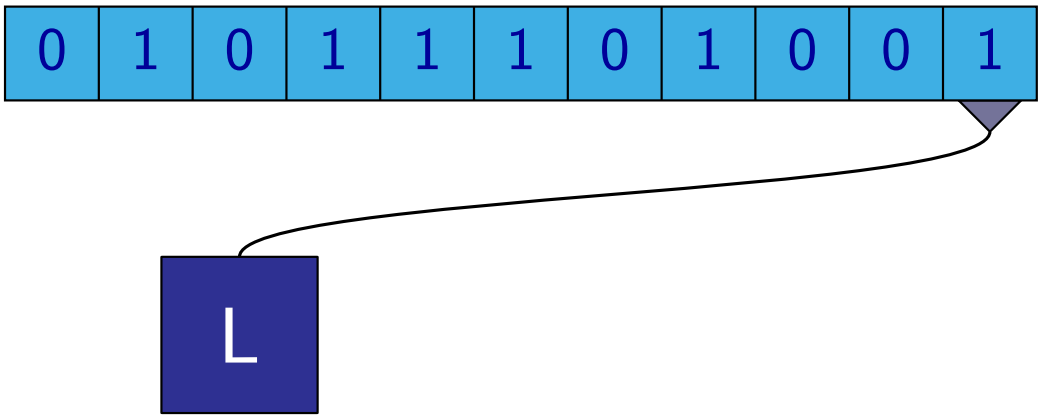




Rozpoznávání jazyka

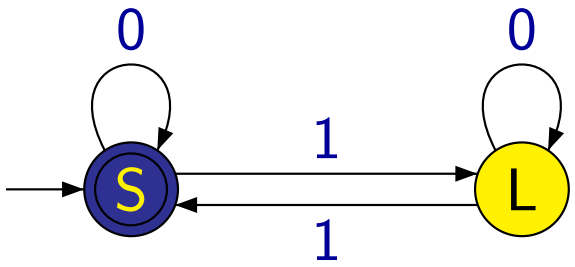
Chovaní tohoto zařízení můžeme popsat grafem:

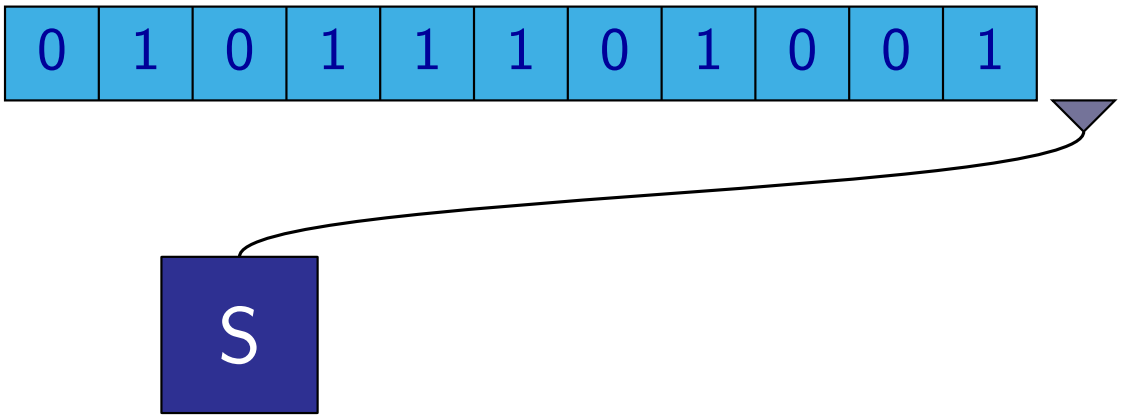




Rozpoznávání jazyka

Chovaní tohoto zařízení můžeme popsat grafem:





Vyhledávání v textu

Problém

Vstup: t – dlouhý text, s – hledaný 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`.

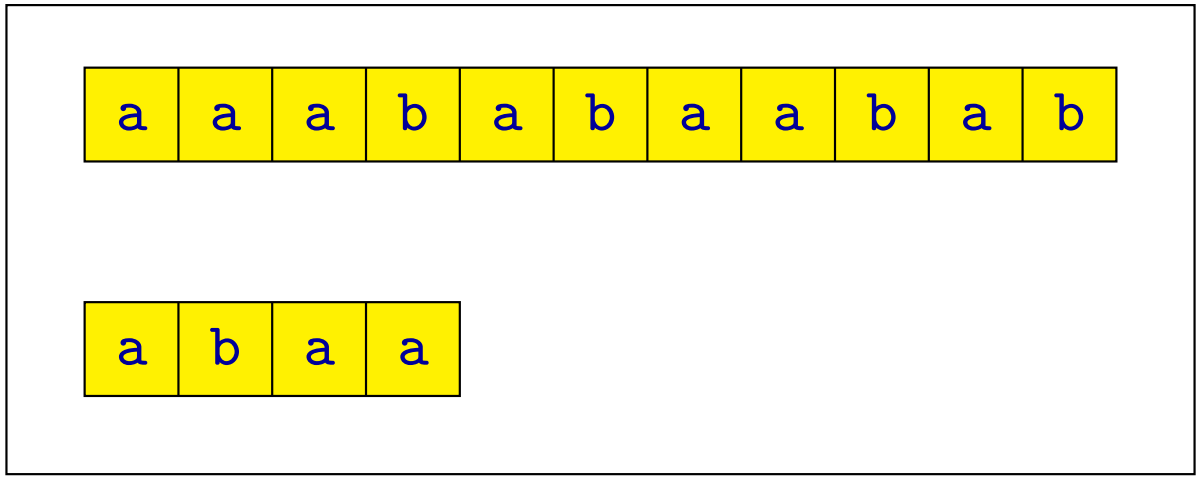
Vyhledávání v textu

Jednoduchý algoritmus, který nás asi napadne v první chvíli:

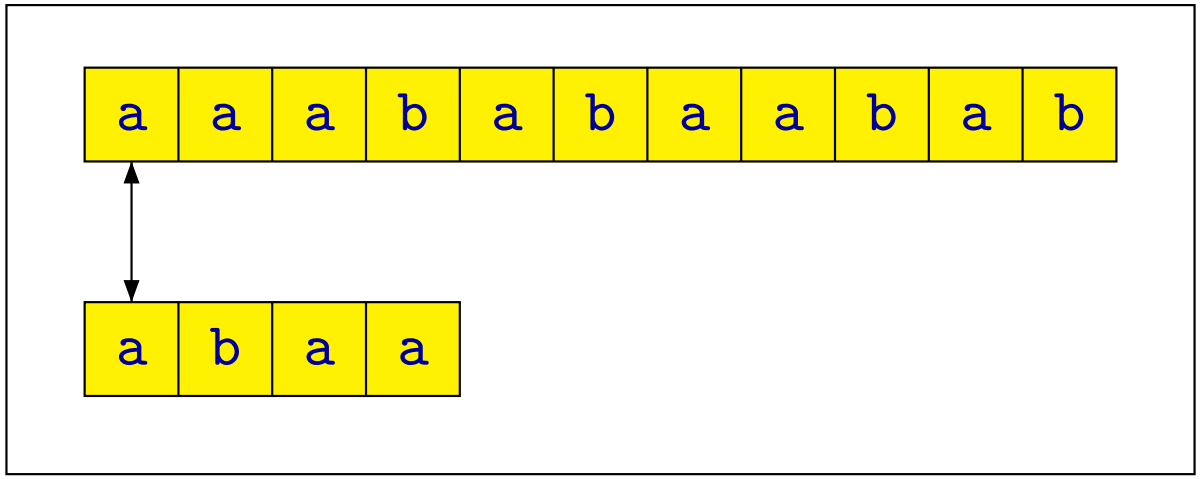
NAIVNÍ-VYHLEDÁVÁNÍ(t, s)

```
1  n ← length(t)
2  m ← length(s)
3  for i ← 0 to n − m
4        do nalezen ← TRUE
5           for j ← 0 to m − 1
6                 do if s[j] ≠ t[i + j]
7                       then nalezen ← FALSE; break
8           if nalezen
9              then return TRUE
10 return FALSE
```

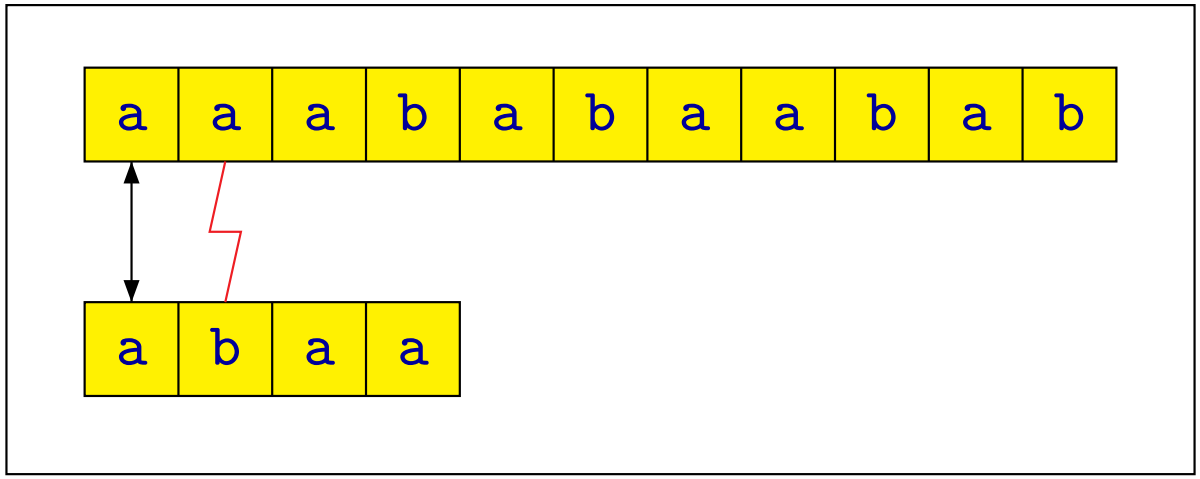
Vyhledávání v textu



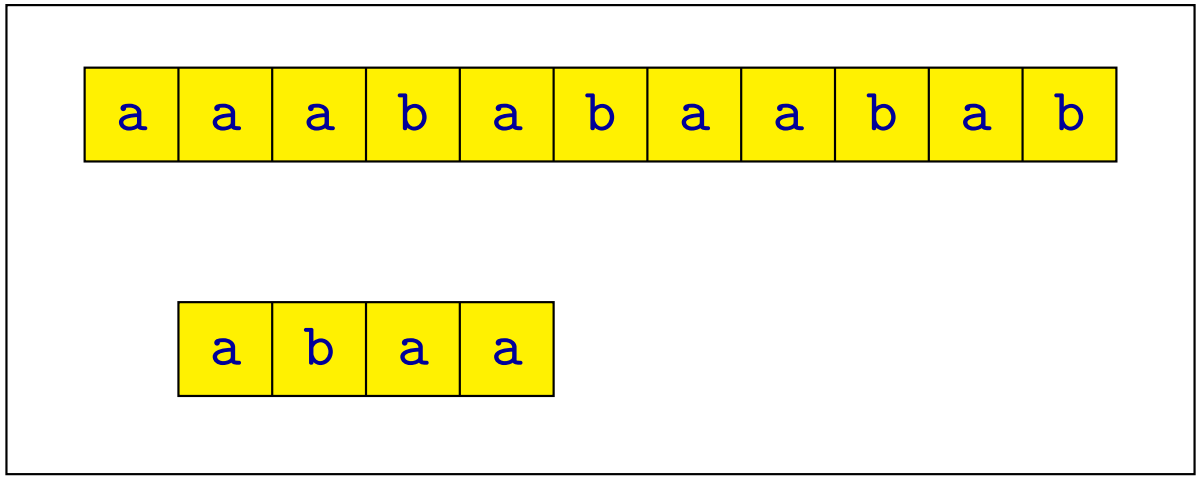
Vyhledávání v textu



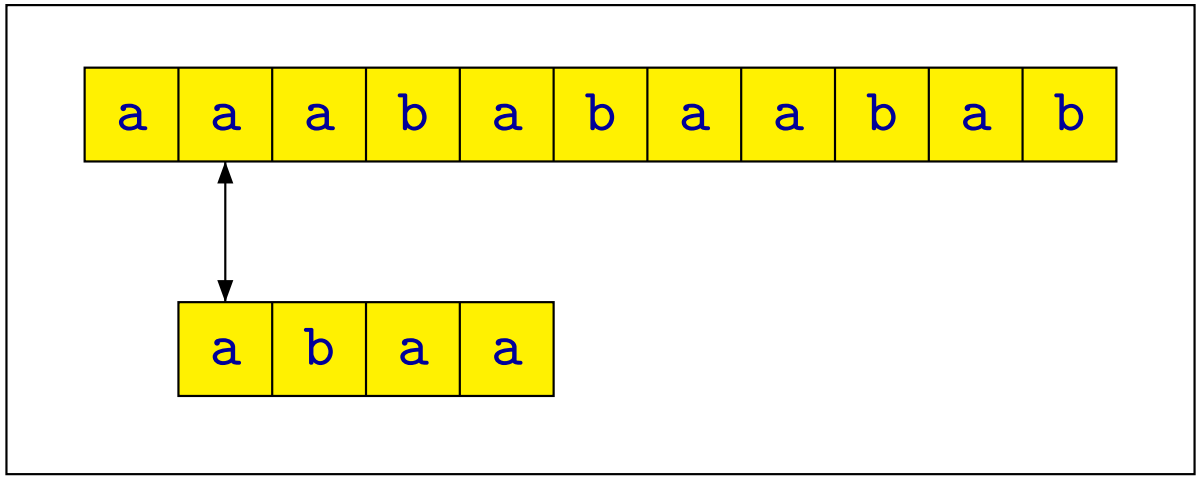
Vyhledávání v textu



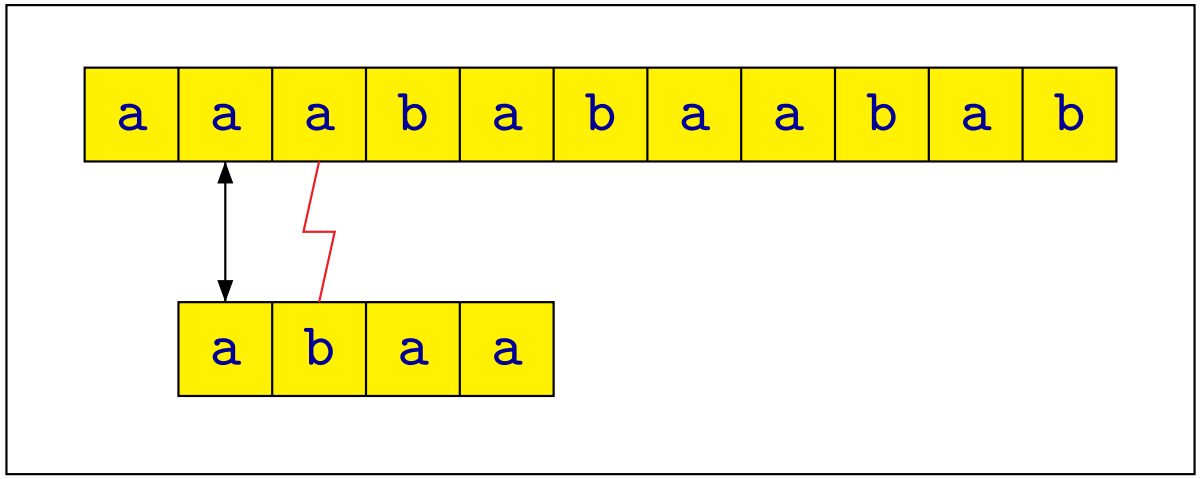
Vyhledávání v textu



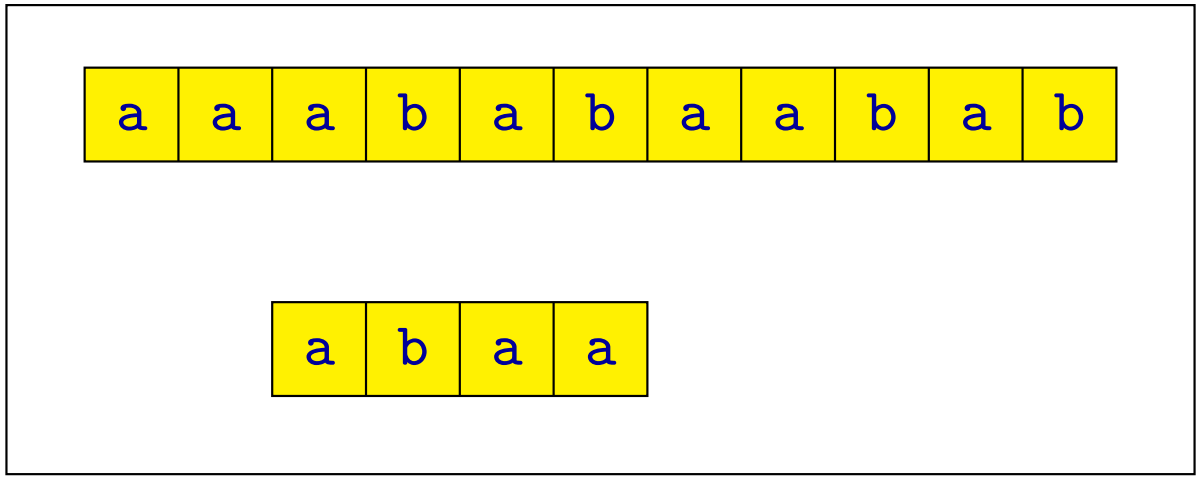
Vyhledávání v textu



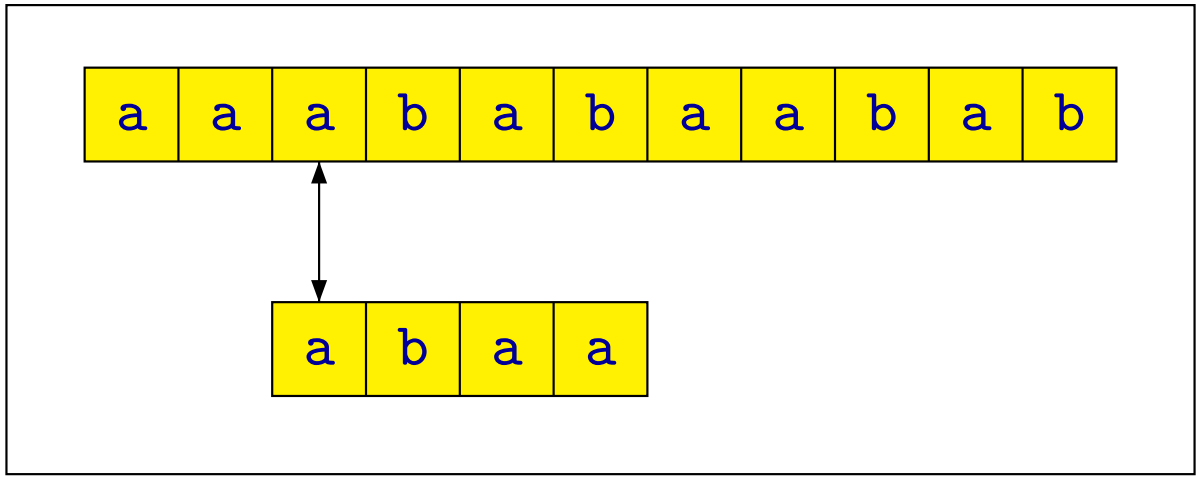
Vyhledávání v textu



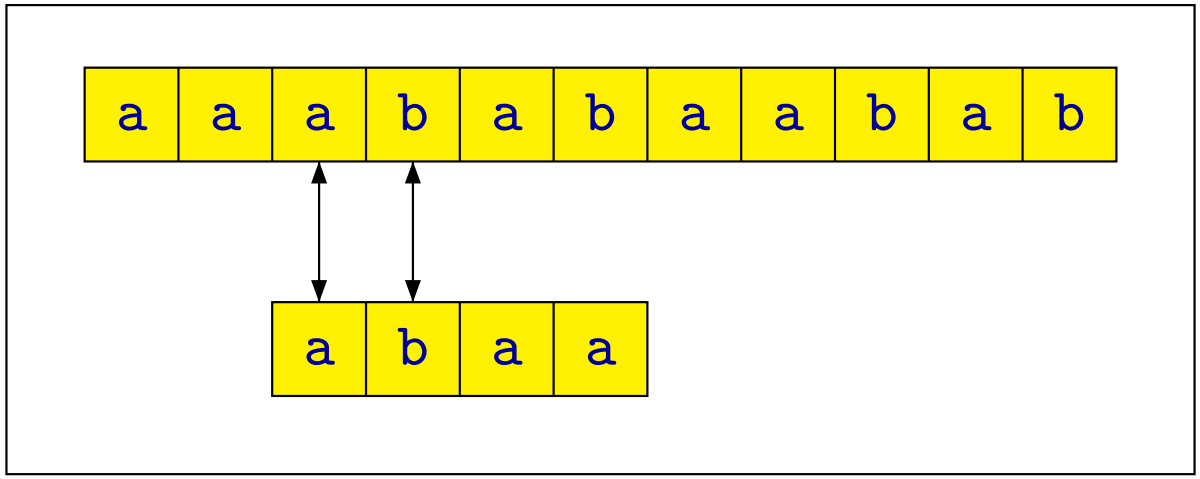
Vyhledávání v textu



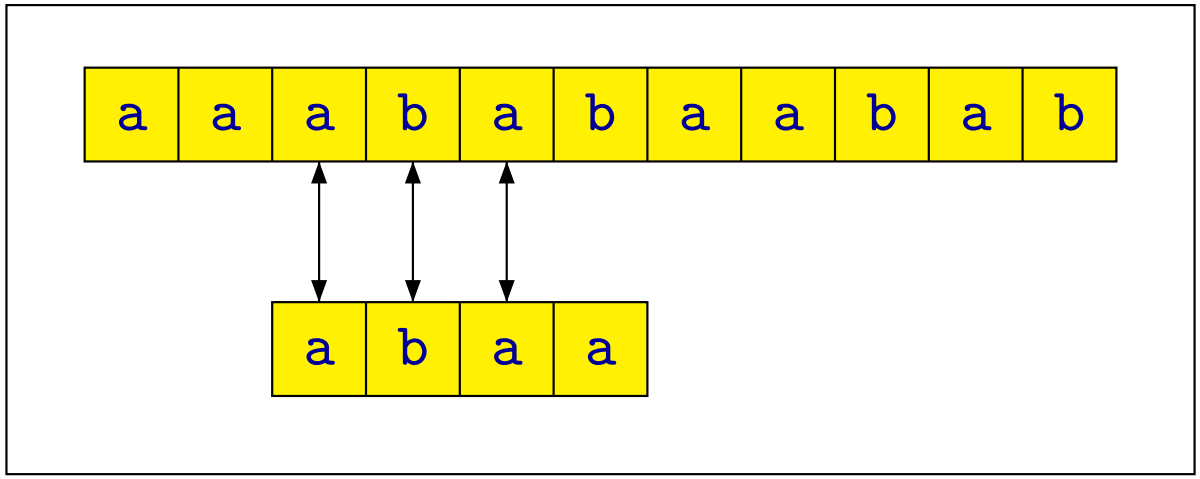
Vyhledávání v textu



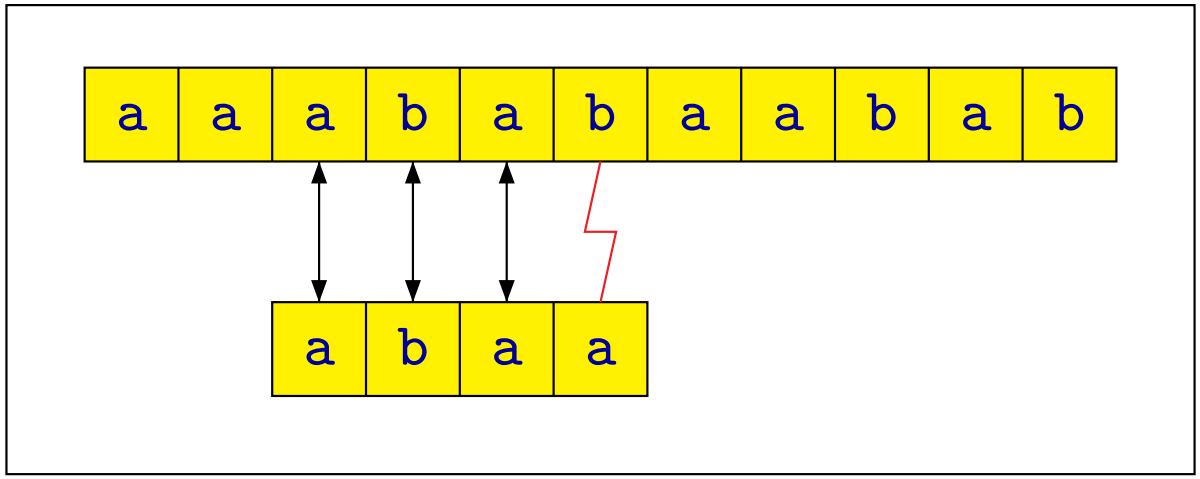
Vyhledávání v textu



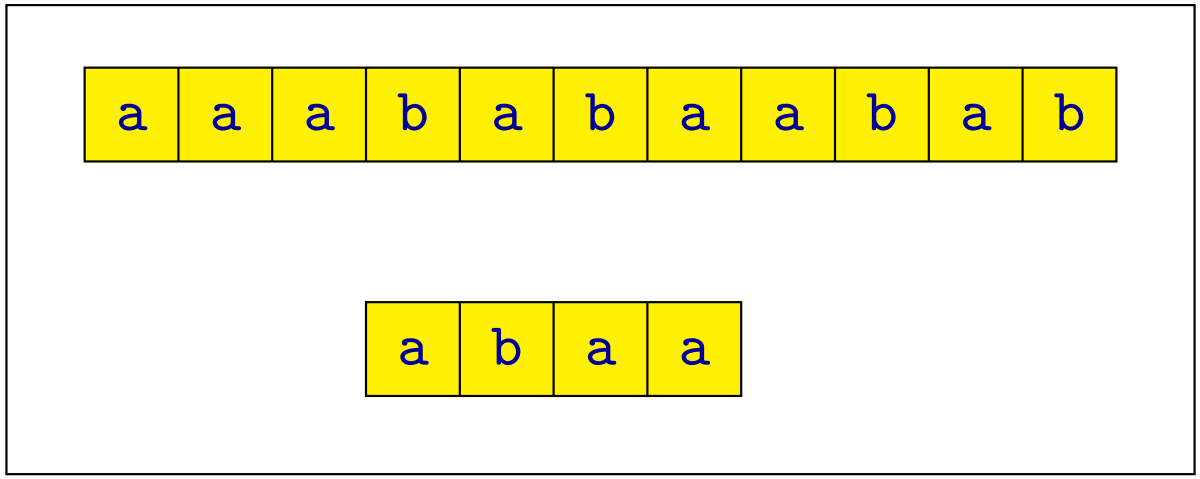
Vyhledávání v textu



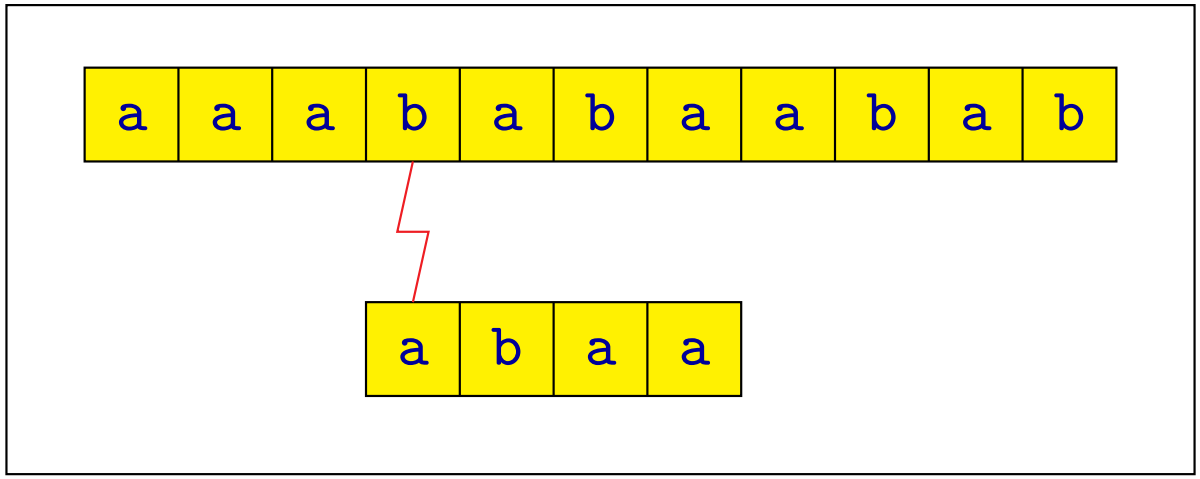
Vyhledávání v textu



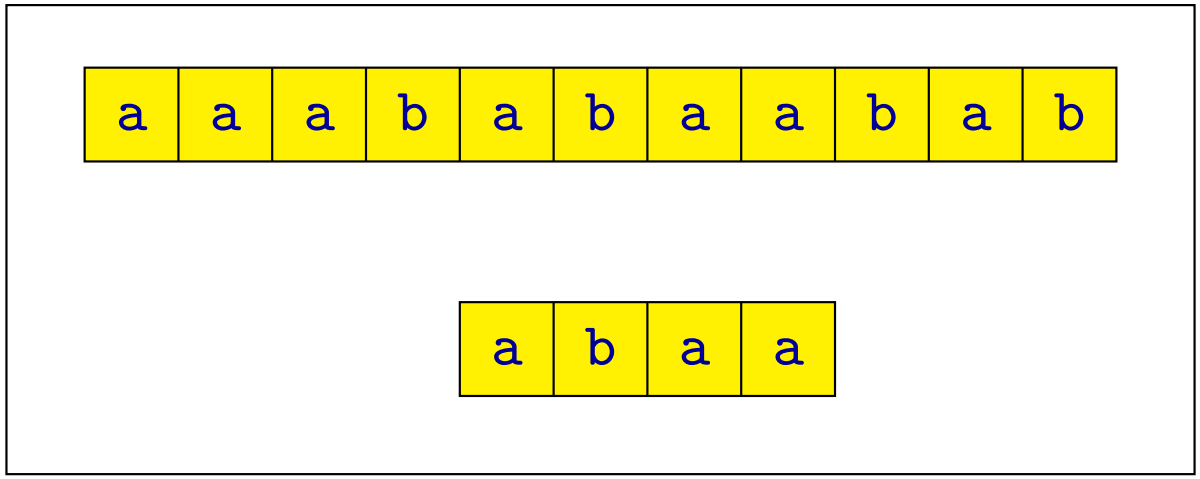
Vyhledávání v textu



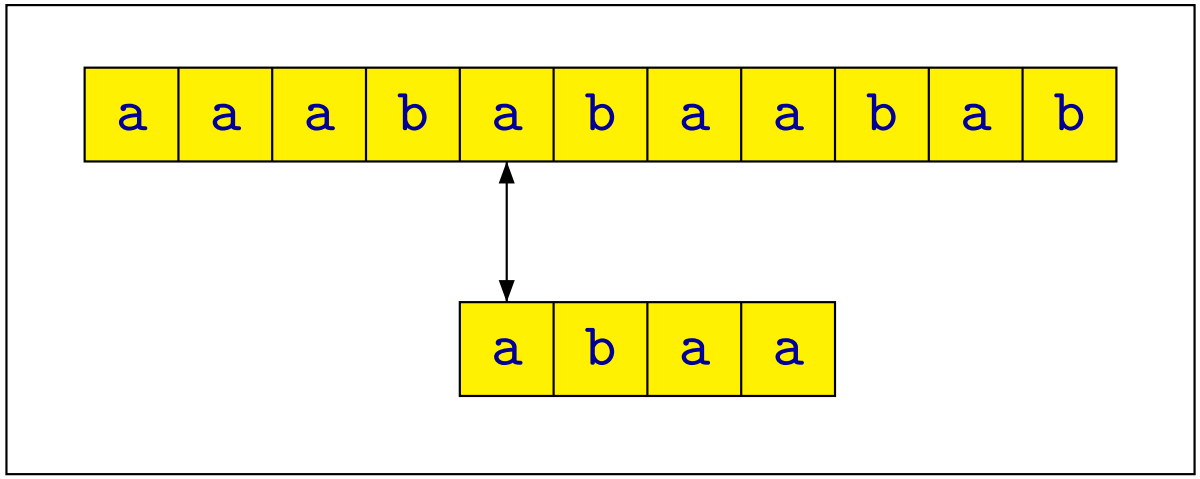
Vyhledávání v textu



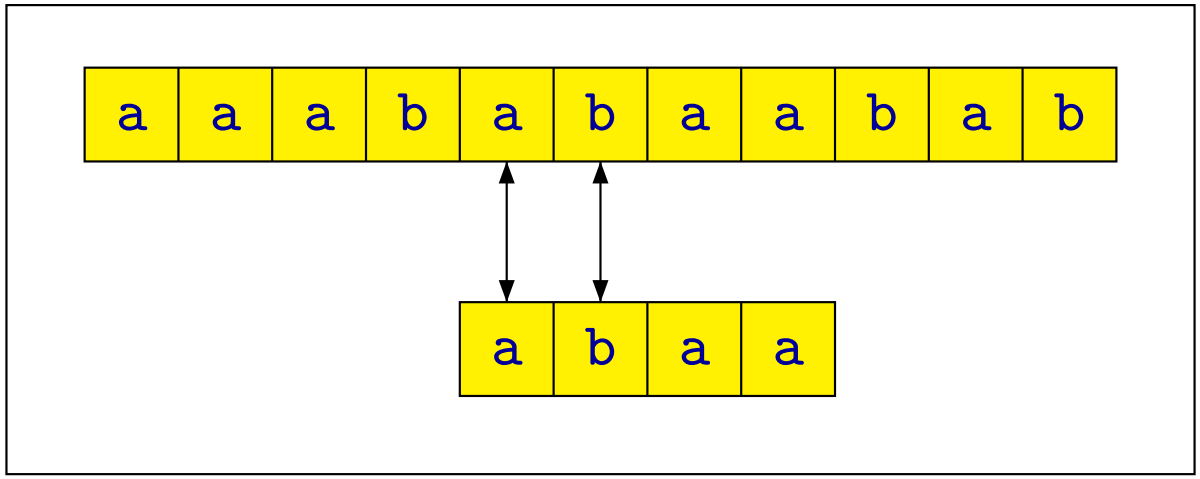
Vyhledávání v textu



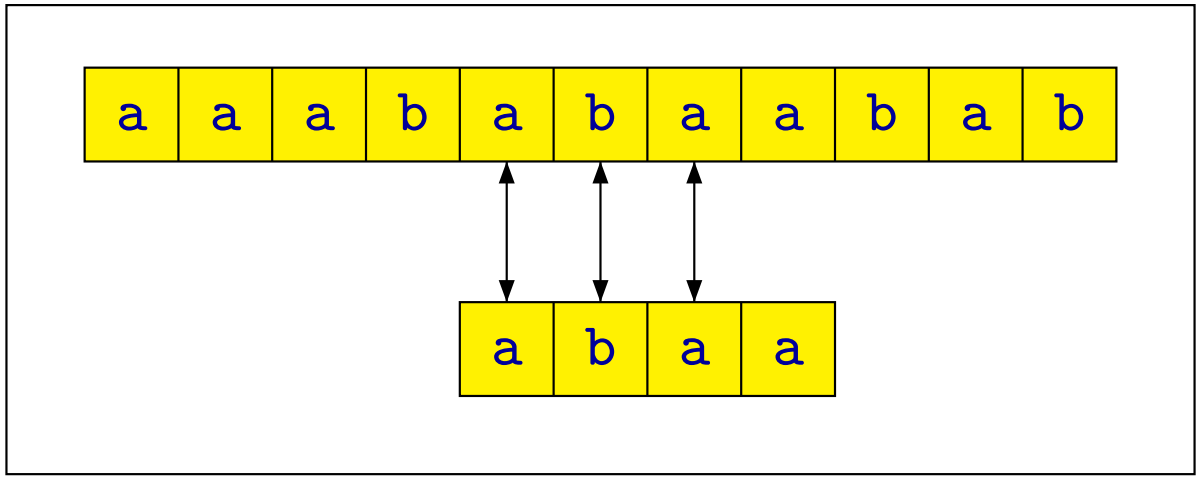
Vyhledávání v textu



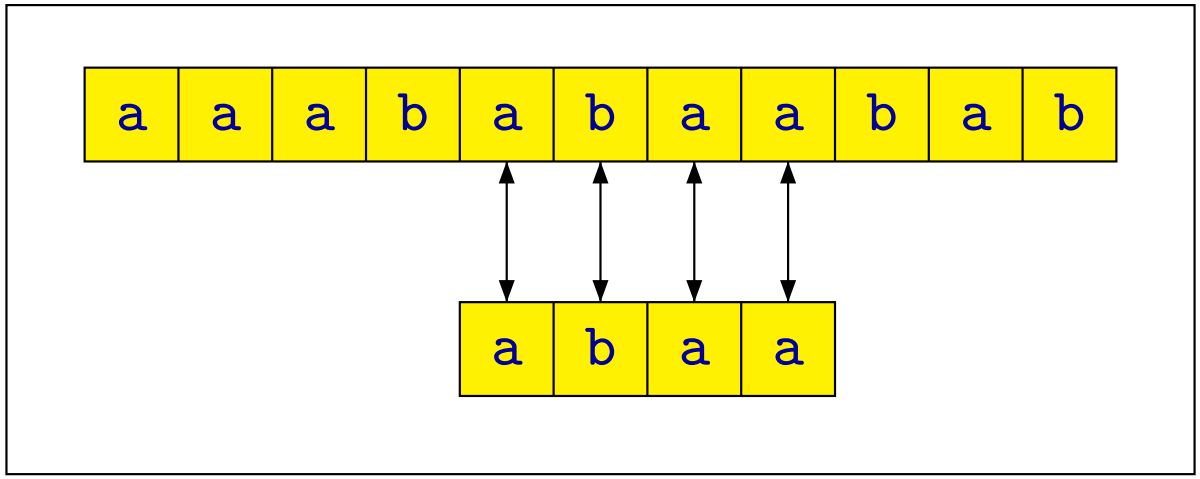
Vyhledávání v textu



Vyhledávání v textu



Vyhledávání v textu



Vyhledávání v textu

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 je shodná s koncem dosud načteného textu. Upravovat tento údaj vždy jen na základě dalšího jednoho načteného znaku:

LEPŠÍ-VYHLEDÁVÁNÍ(t, s)

```
1  n ← length(t)
2  m ← length(s)
3  q ← 0
4  for i ← 0 to n − 1
5        do q ← δ(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 ε, `a`, `ab`, `aba`, `abaa`, `abaab`.
- Sufixy slova `abaab` jsou ε, `b`, `ab`, `aab`, `baab`, `abaab`.
- Podslova slova `abaab` jsou ε, `a`, `b`, `ab`, `ba`, `ab`, `aba`, `baa`, `abb`, `abaa`, `baab`, `abaab`.

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 $\{0, 1, \ldots, m\}$. Hodnoty m nabývá jen v případě, že bylo nalezeno slovo s.

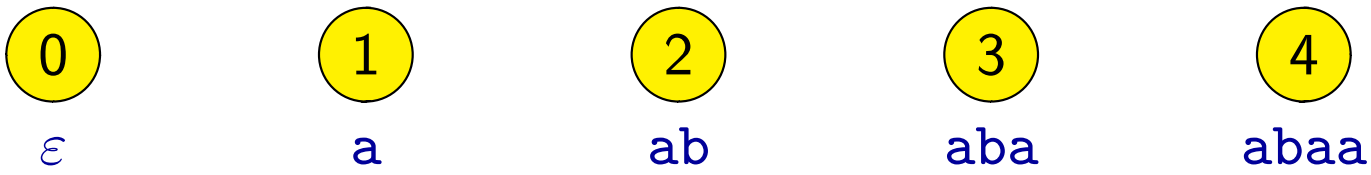
Vyhledávání v textu

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 $\{0, 1, \ldots, m\}$. Hodnoty m nabývá jen v případě, že bylo nalezeno slovo s.

Chování tohoto systému si opět můžeme znázornit jako graf:



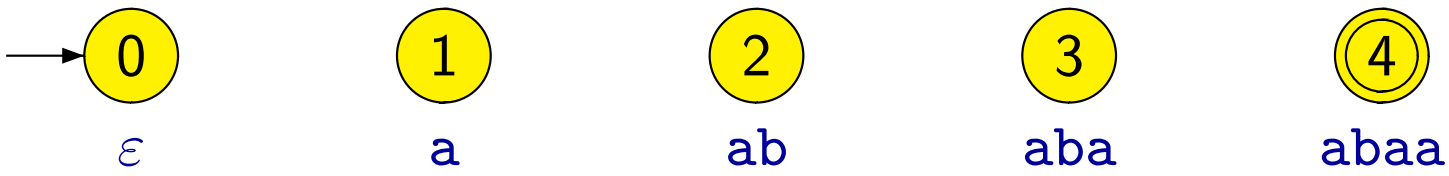
Vyhledávání v textu

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 $\{0, 1, \ldots, m\}$. Hodnoty m nabývá jen v případě, že bylo nalezeno slovo s.

Chování tohoto systému si opět můžeme znázornit jako graf:



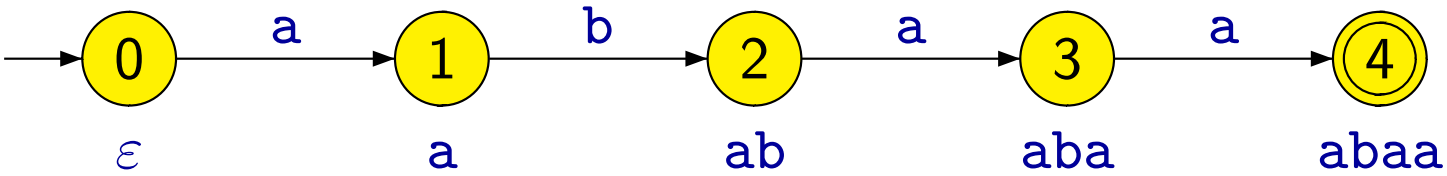
Vyhledávání v textu

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 $\{0, 1, \ldots, m\}$. Hodnoty m nabývá jen v případě, že bylo nalezeno slovo s.

Chování tohoto systému si opět můžeme znázornit jako graf:



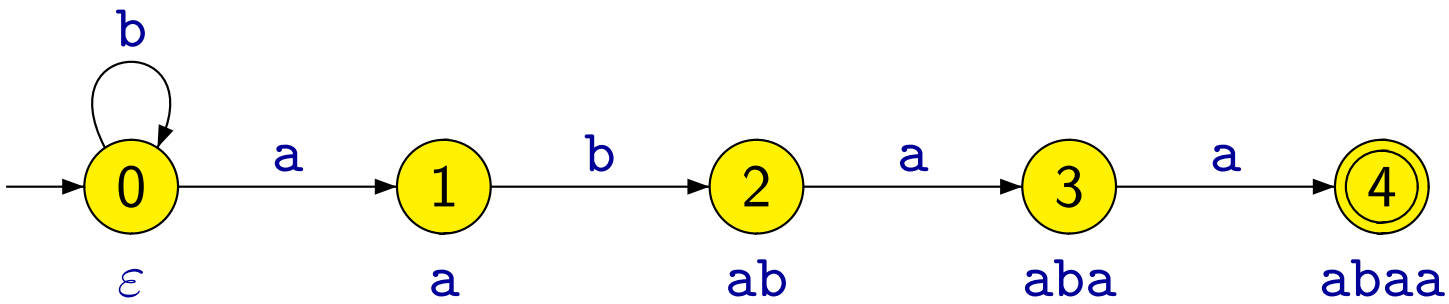
Vyhledávání v textu

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 $\{0, 1, \ldots, m\}$. Hodnoty m nabývá jen v případě, že bylo nalezeno slovo s.

Chování tohoto systému si opět můžeme znázornit jako graf:



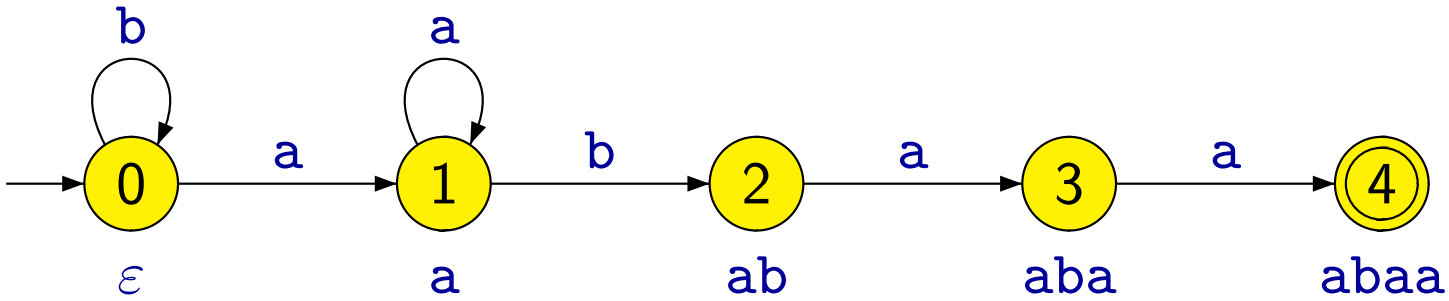
Vyhledávání v textu

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 $\{0, 1, \ldots, m\}$. Hodnoty m nabývá jen v případě, že bylo nalezeno slovo s.

Chování tohoto systému si opět můžeme znázornit jako graf:



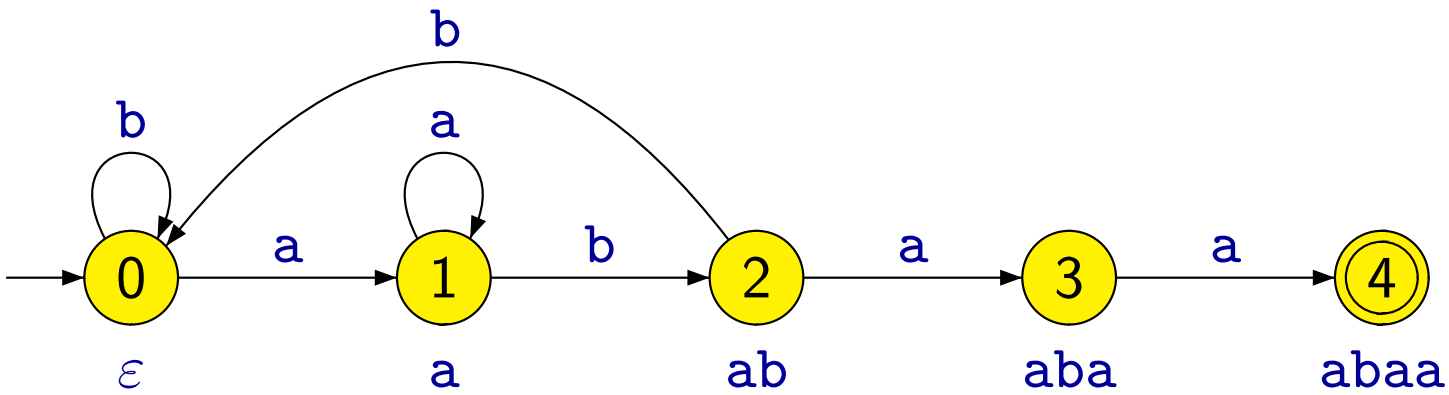
Vyhledávání v textu

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 $\{0, 1, \ldots, m\}$. Hodnoty m nabývá jen v případě, že bylo nalezeno slovo s.

Chování tohoto systému si opět můžeme znázornit jako graf:



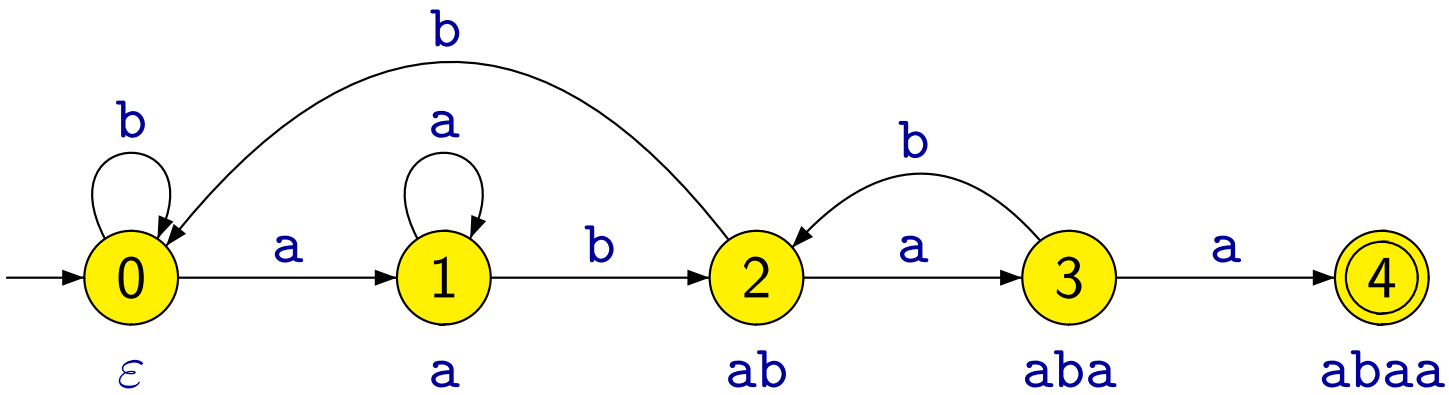
Vyhledávání v textu

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 $\{0, 1, \ldots, m\}$. Hodnoty m nabývá jen v případě, že bylo nalezeno slovo s.

Chování tohoto systému si opět můžeme znázornit jako graf:



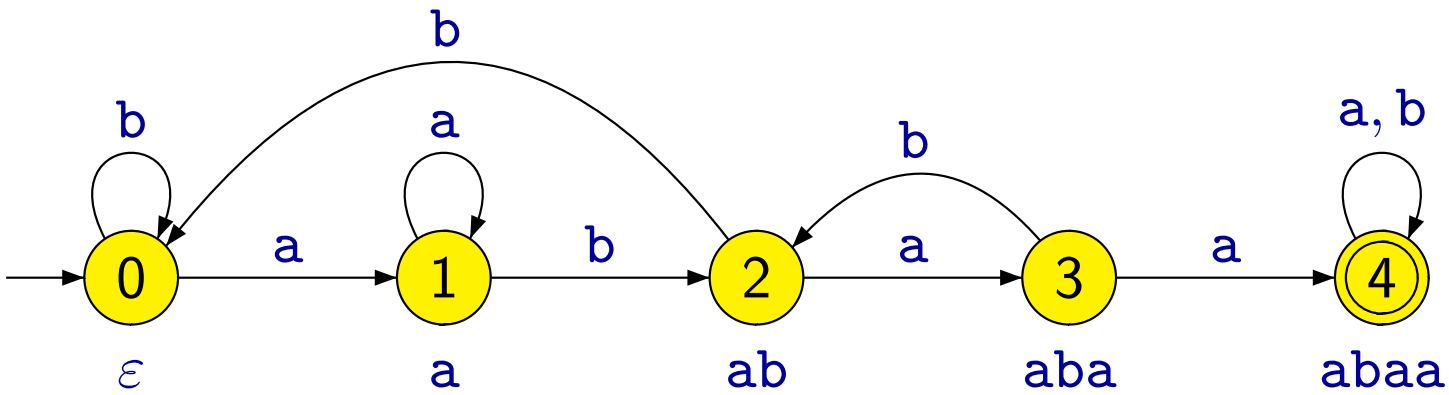
Vyhledávání v textu

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

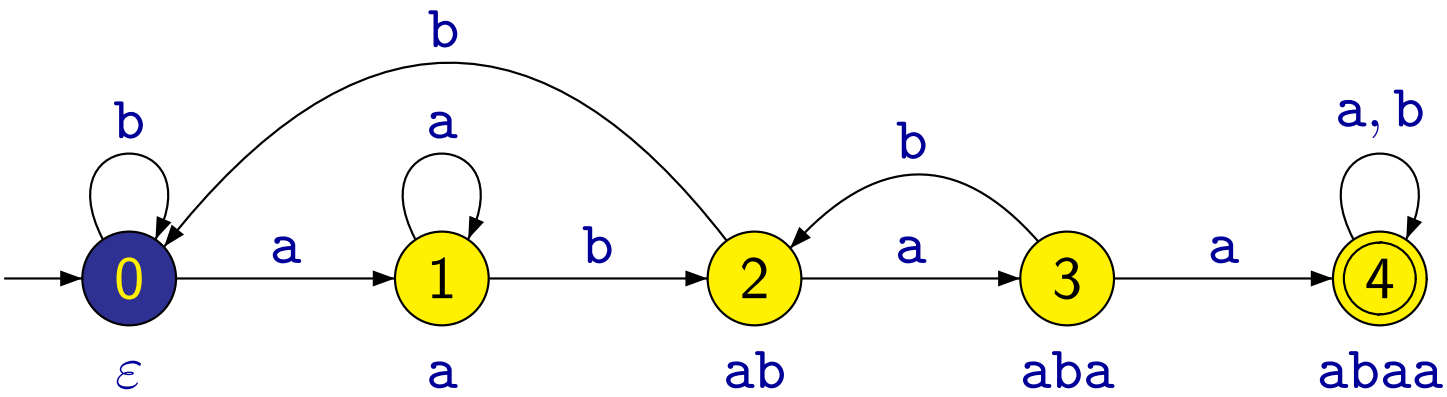
Poznámka: Pokud je takových prefixů víc, pamatujeme si délku nejdelšího z nich.

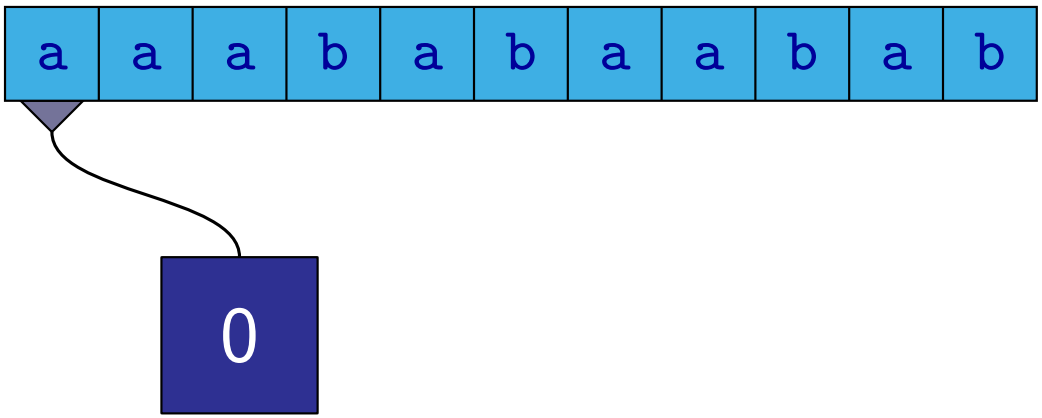
Proměnná zjevně může nabývat jen hodnot $\{0, 1, \ldots, m\}$. Hodnoty m nabývá jen v případě, že bylo nalezeno slovo s.

Chování tohoto systému si opět můžeme znázornit jako graf:

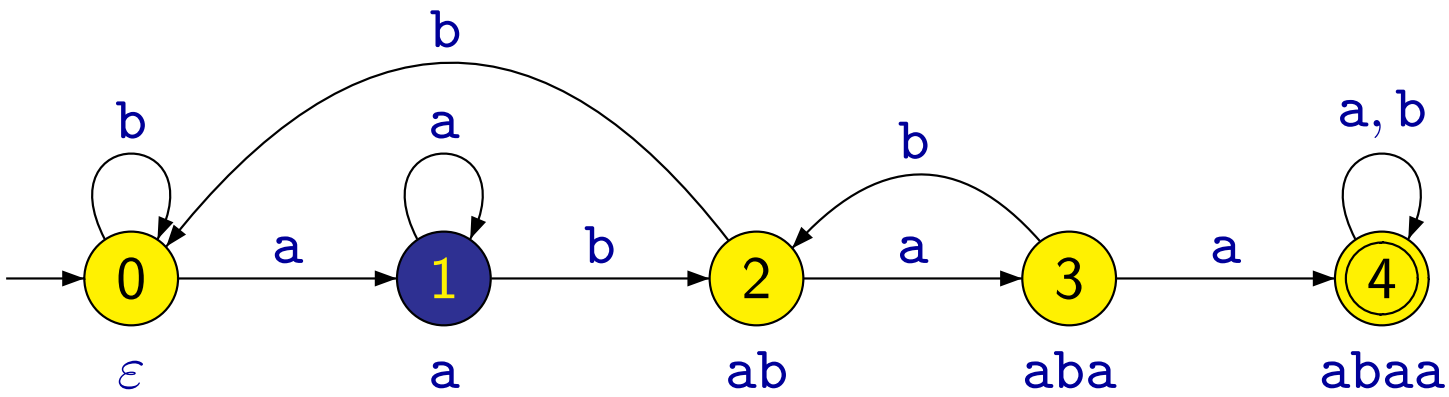


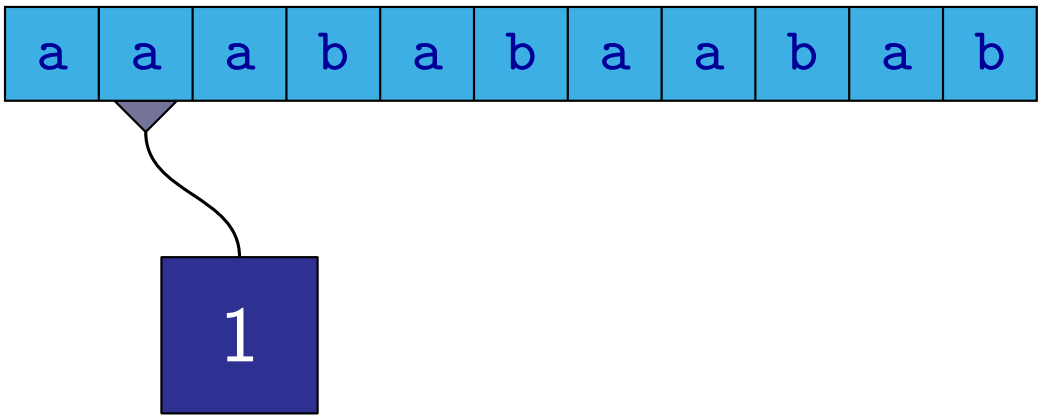
Vyhledávání v textu



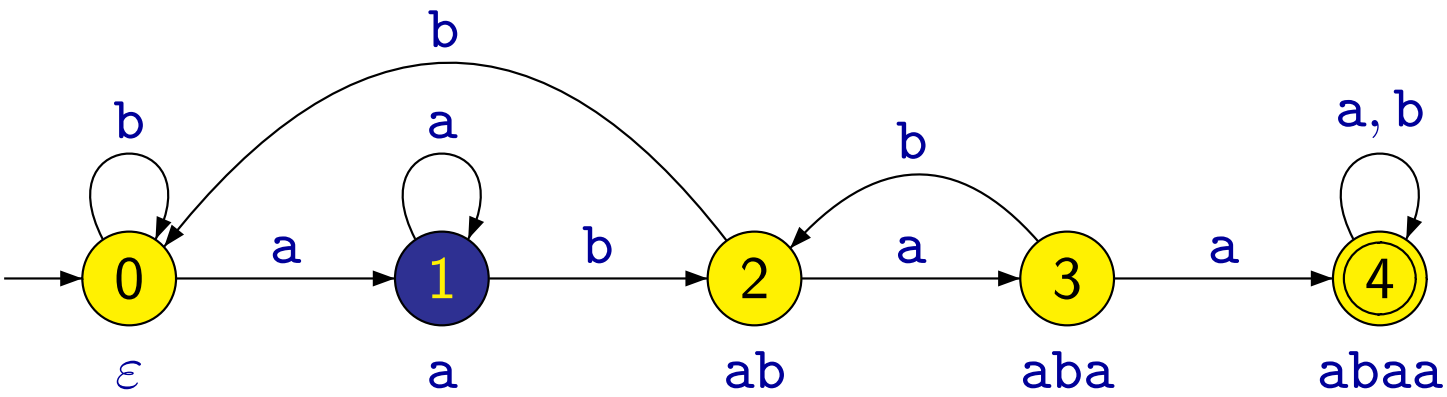


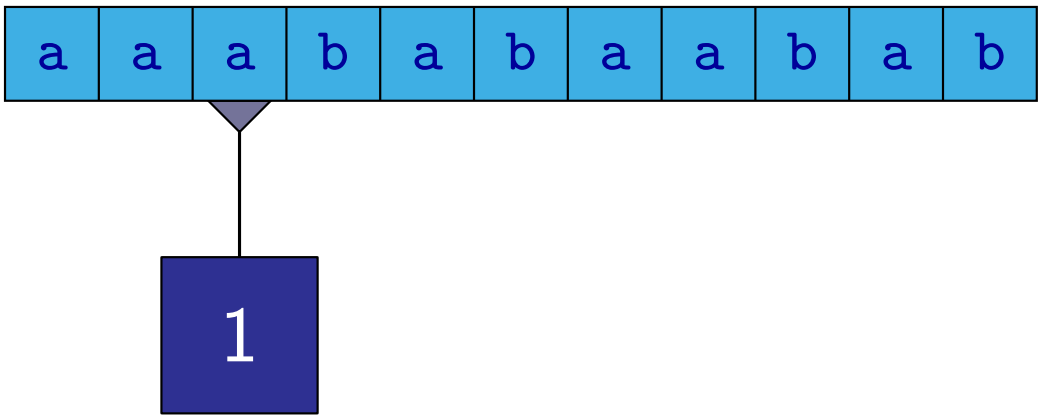
Vyhledávání v textu



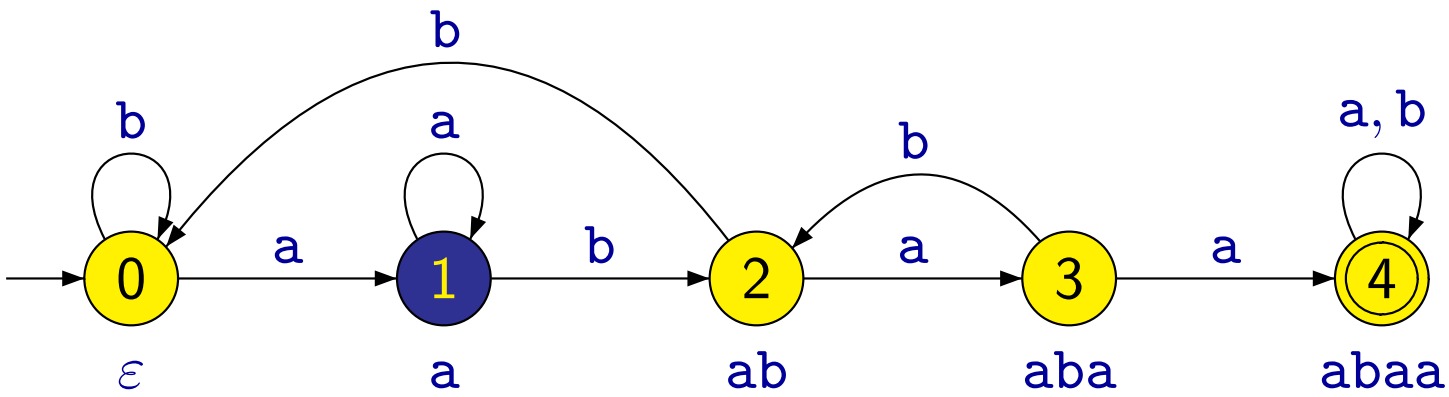


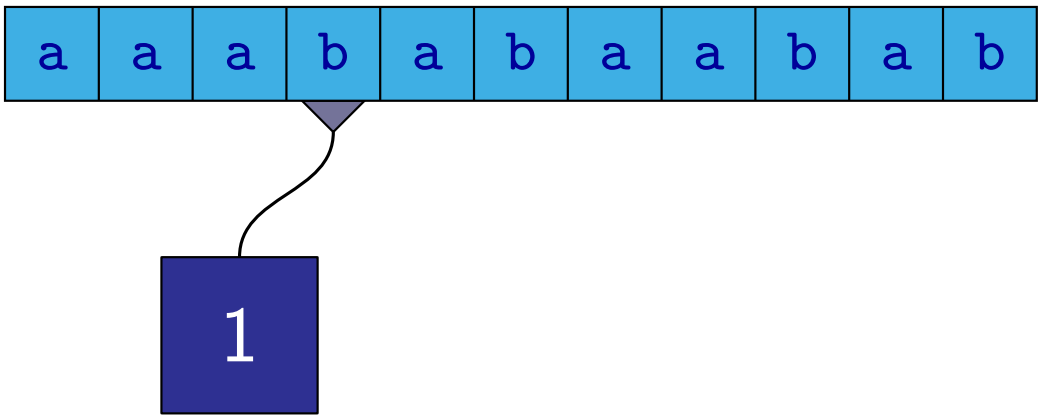
Vyhledávání v textu



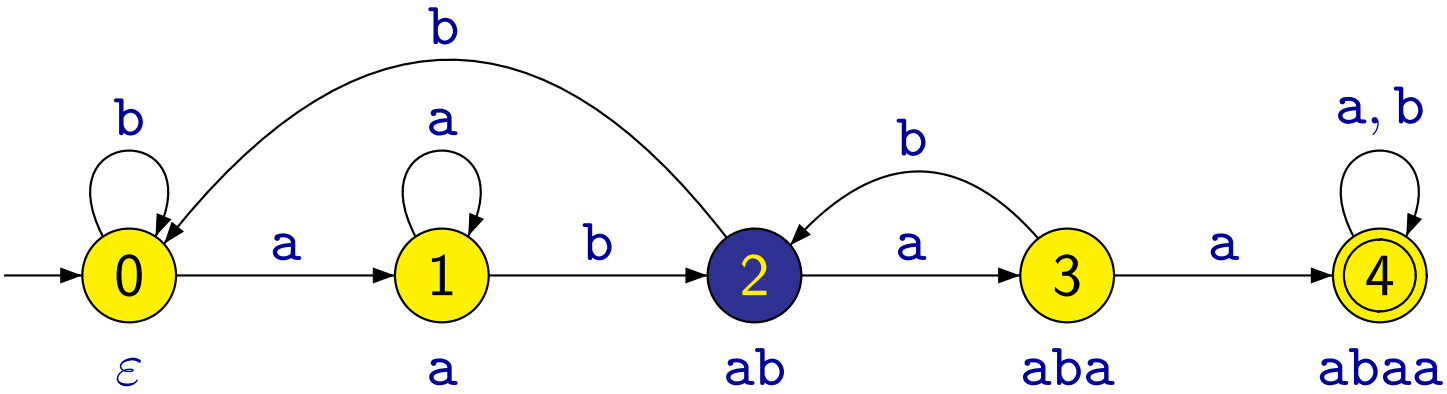


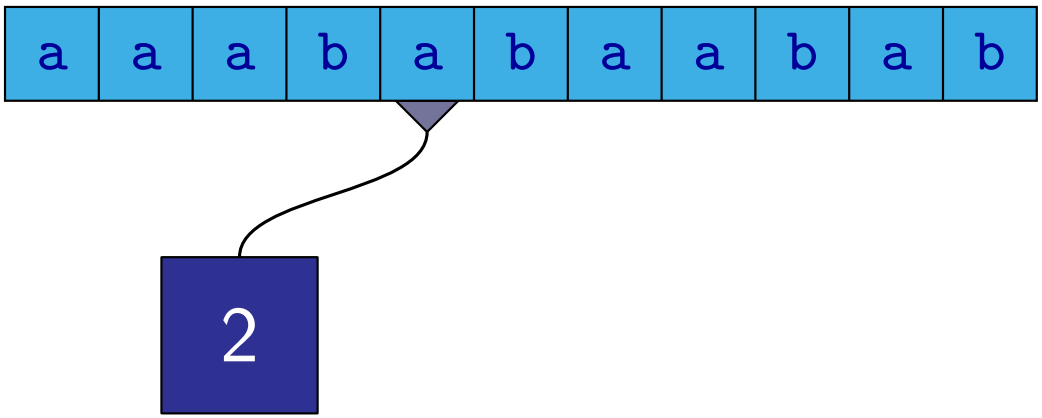
Vyhledávání v textu



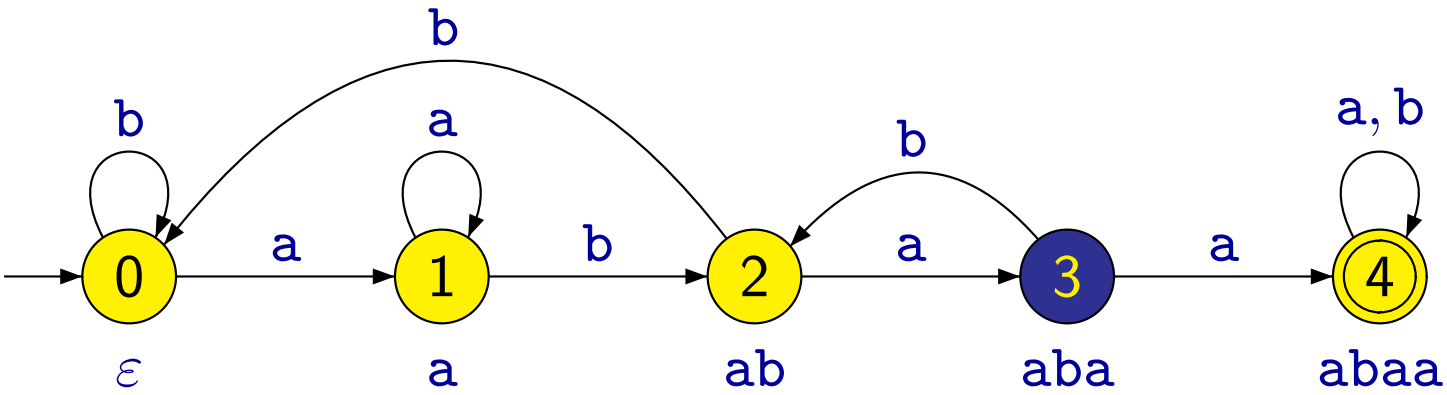


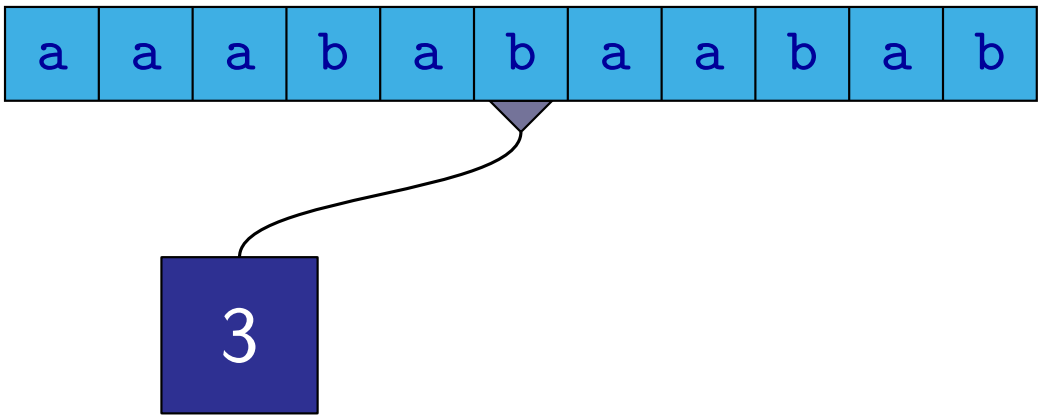
Vyhledávání v textu



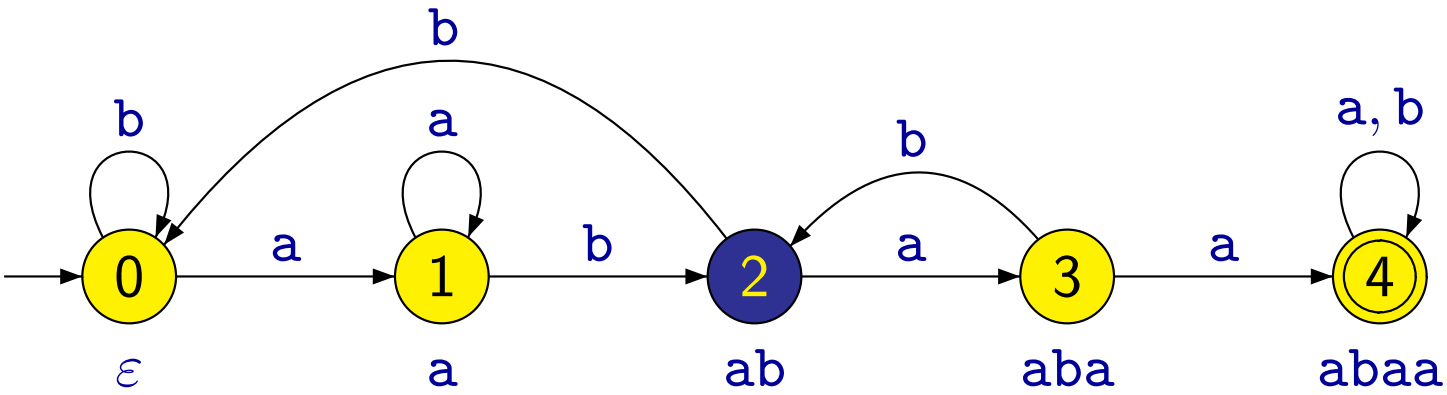


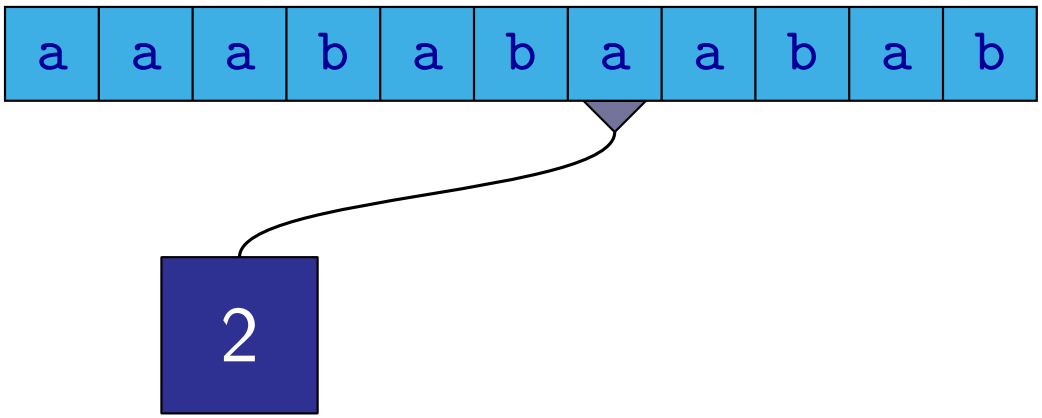
Vyhledávání v textu



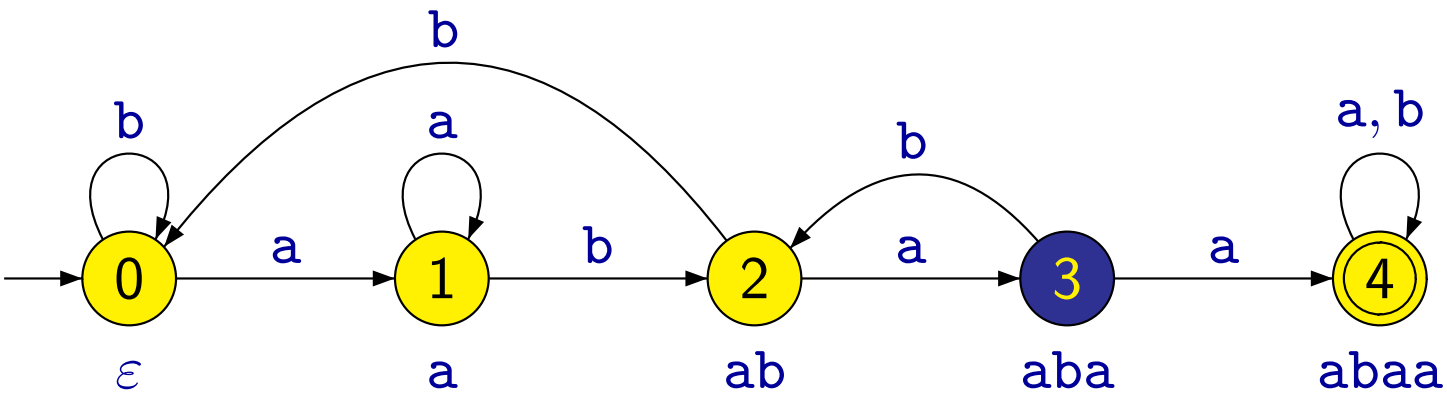


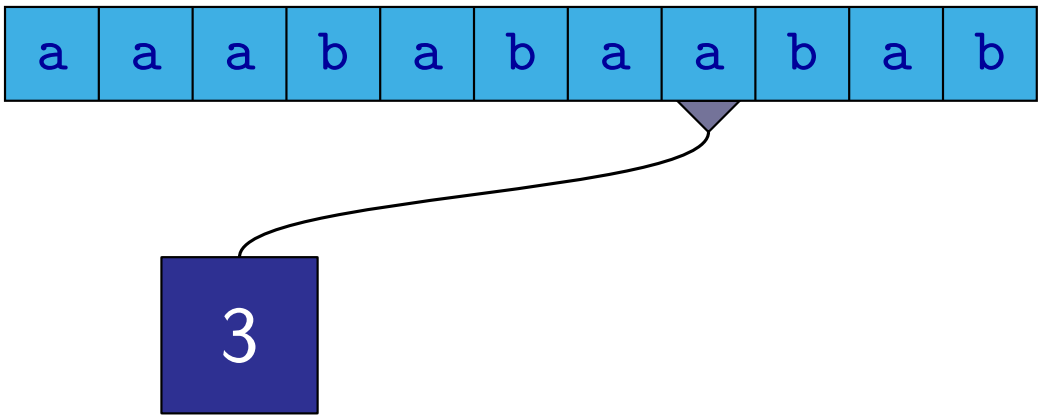
Vyhledávání v textu



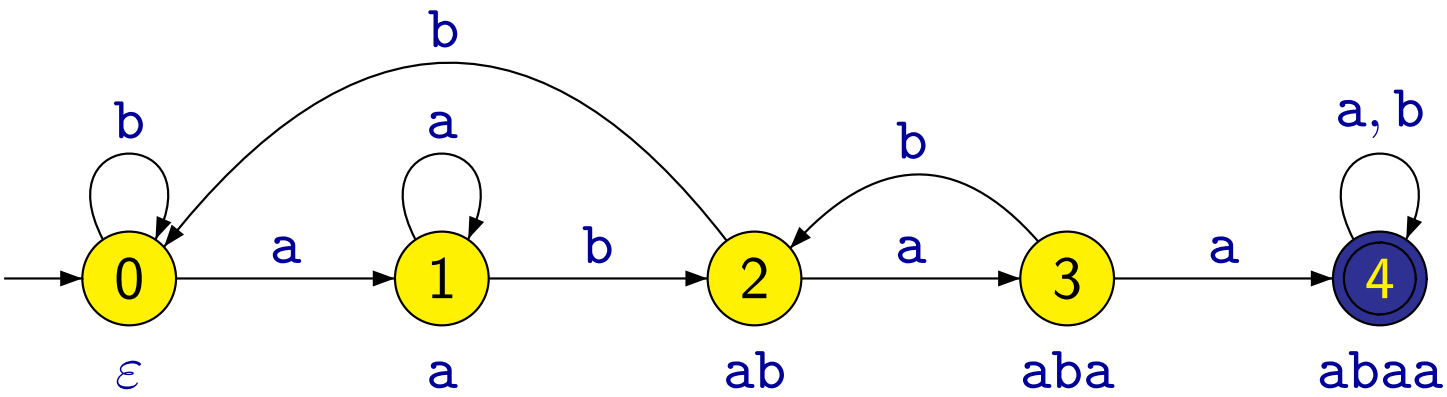


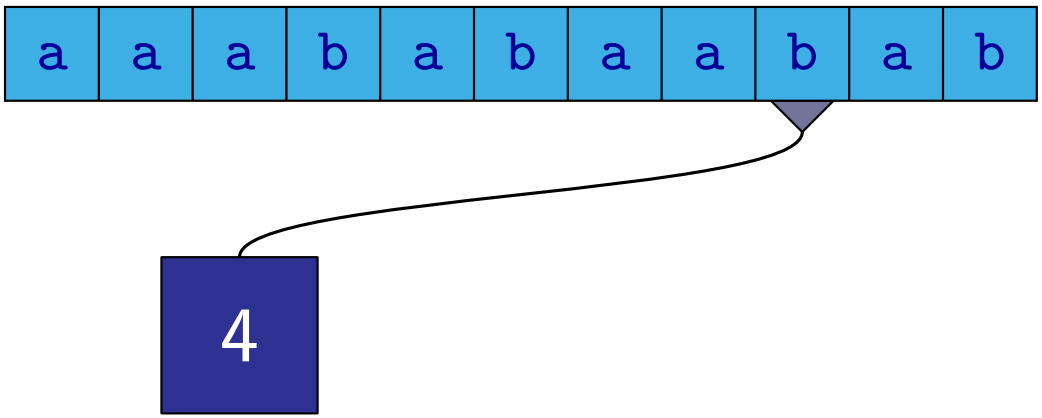
Vyhledávání v textu



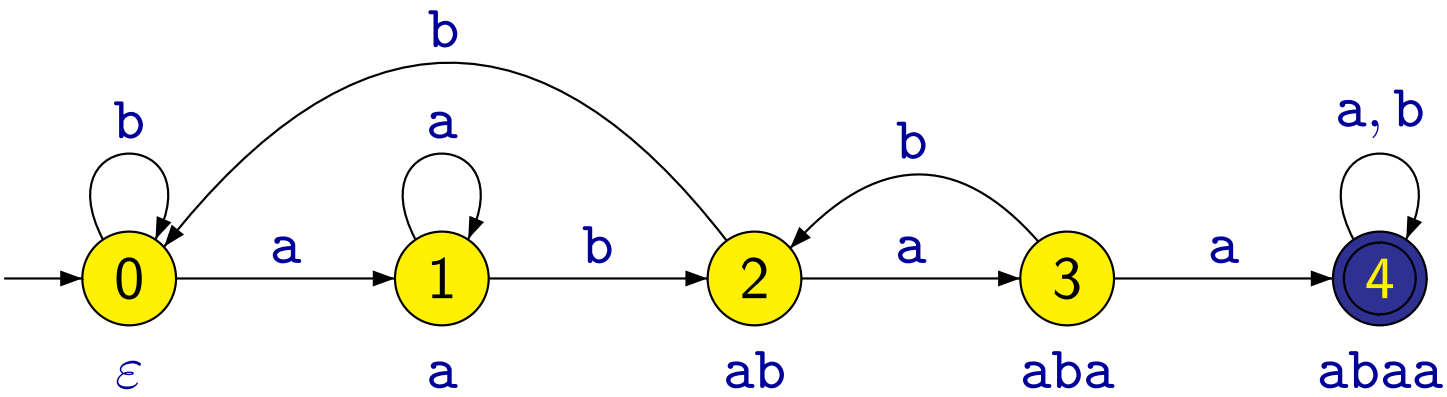


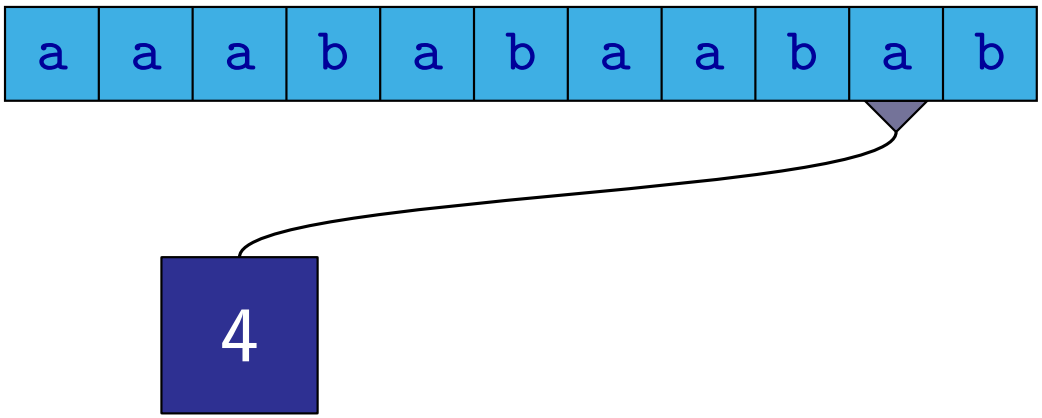
Vyhledávání v textu



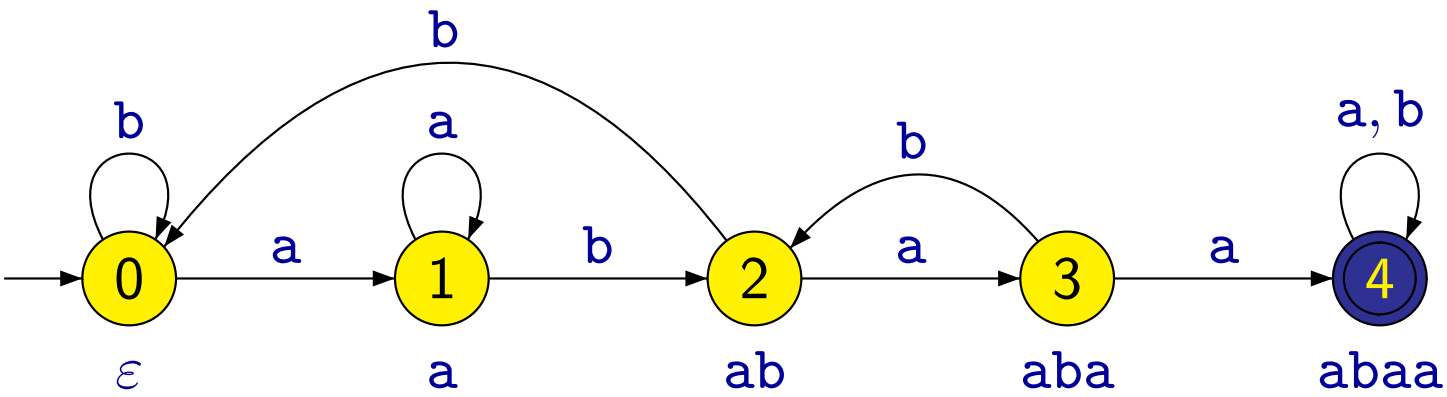


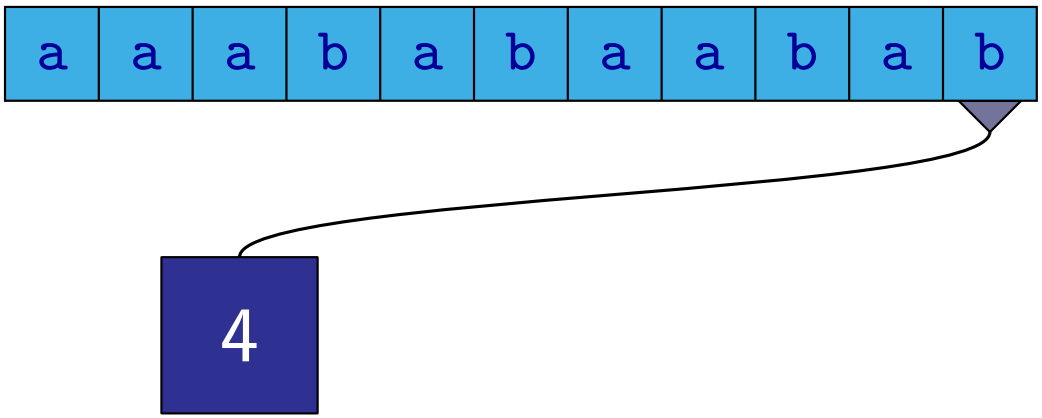
Vyhledávání v textu



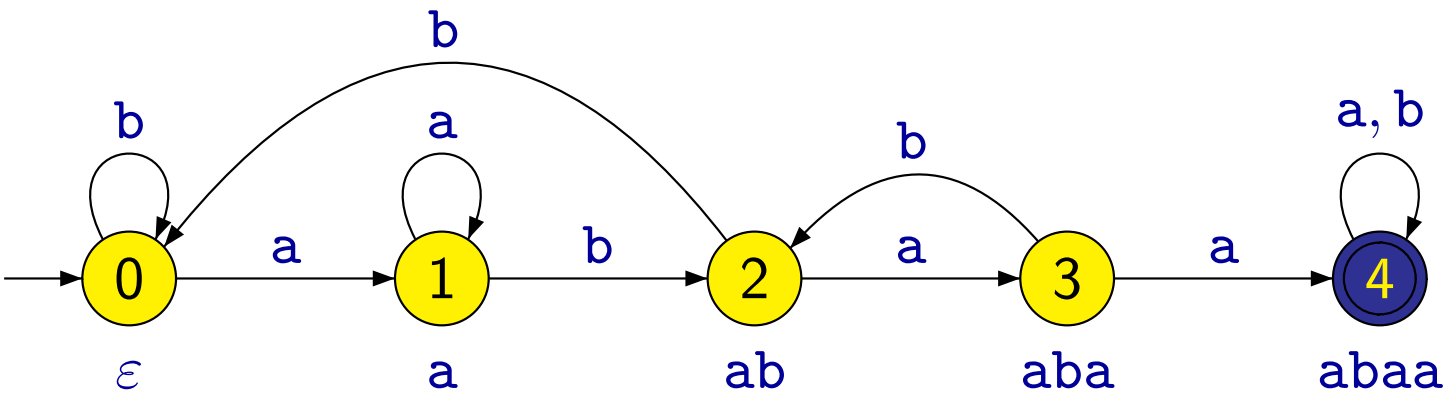


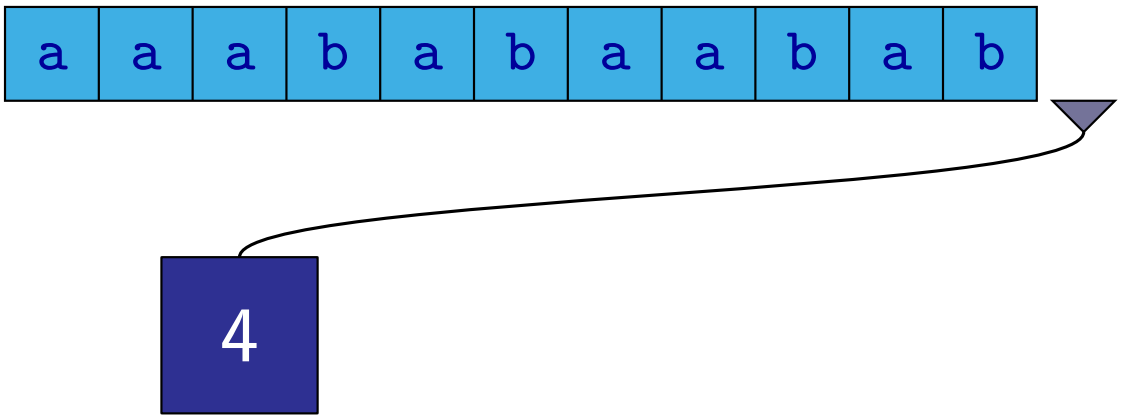
Vyhledávání v textu



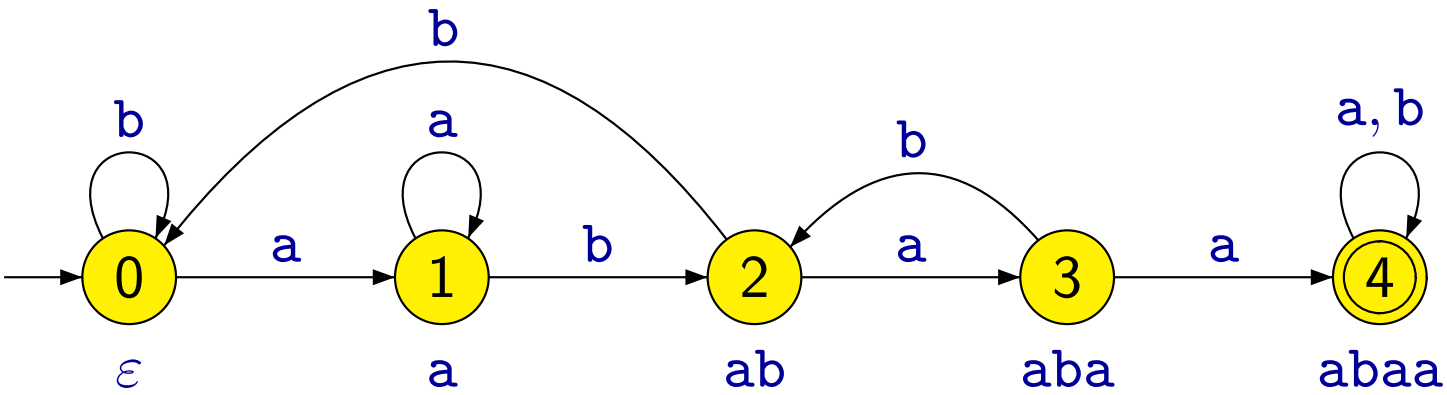


Vyhledávání v textu





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

Vyhledávání v textu

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

Vyhledávání v textu

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 $s_0 s_1 \cdots s_{q'-1}$ je nejdelším prefixem slova s takovým, že je současně suffixem slova $s_0 s_1 \cdots s_{q-1} t_i$.

Poznámka: Předpokládáme, že $s = s_0 s_1 \cdots s_{m-1}$.

Vyhledávání v textu

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 $s_0 s_1 \cdots s_{q'-1}$ je nejdelším prefixem slova s takovým, že je současně suffixem slova $s_0 s_1 \cdots s_{q-1} t_i$.

Poznámka: Předpokládáme, že $s = s_0 s_1 \cdots s_{m-1}$.

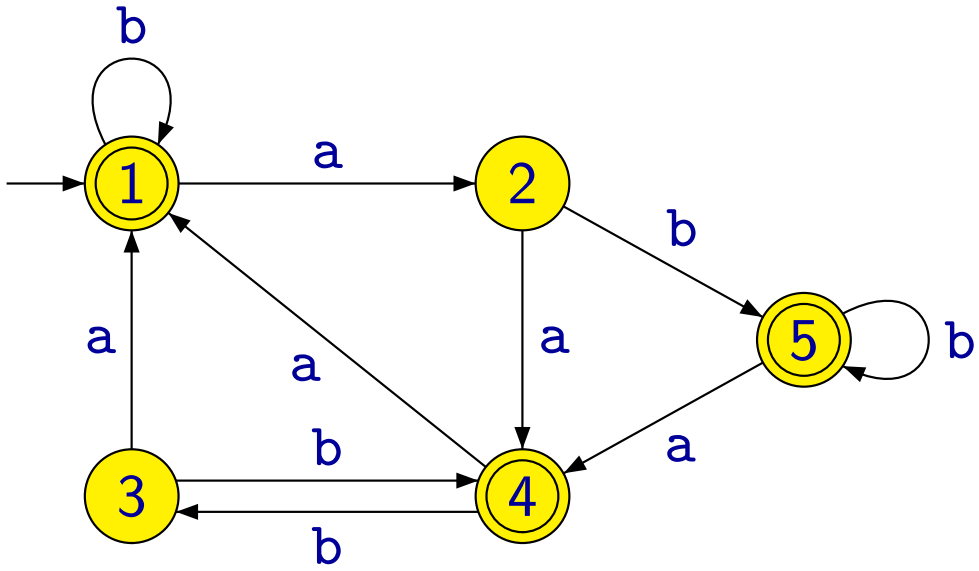
Poznámka: Výše uvedené úvahy vedou k algoritmu nazývanému podle jeho autorů Knuth-Morris-Pratt.

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.

Konečné automaty

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

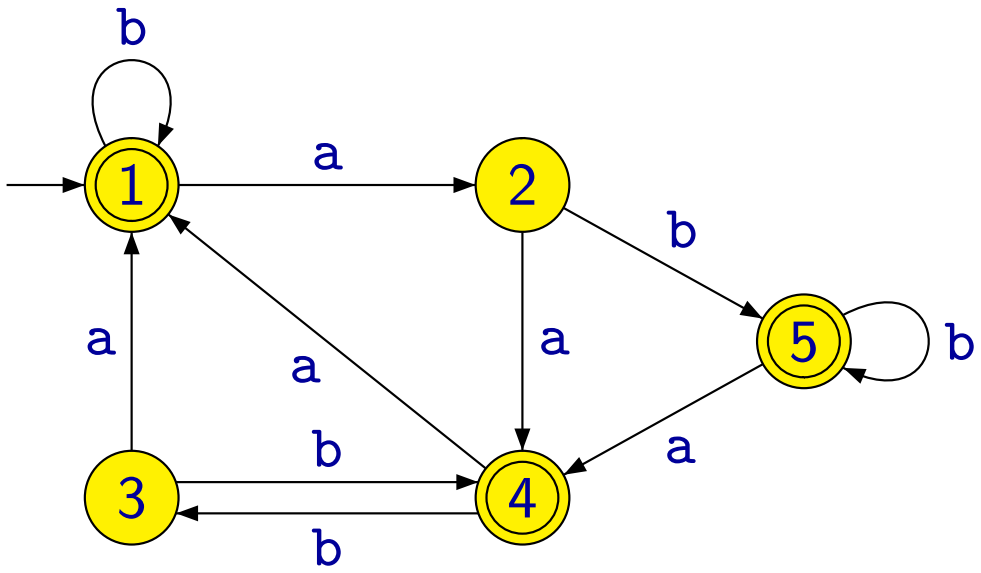
Formálně je **deterministický konečný automat** definován jako pětice

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

kde:

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

Deterministický konečný automat



- $Q = \{1, 2, 3, 4, 5\}$
- $\Sigma = \{\mathtt{a}, \mathtt{b}\}$
- $q_0 = 1$
- $F = \{1, 4, 5\}$

$$\delta(1, \mathtt{a}) = 2 \qquad \delta(1, \mathtt{b}) = 1$$
$$\delta(2, \mathtt{a}) = 4 \qquad \delta(2, \mathtt{b}) = 5$$
$$\delta(3, \mathtt{a}) = 1 \qquad \delta(3, \mathtt{b}) = 4$$
$$\delta(4, \mathtt{a}) = 1 \qquad \delta(4, \mathtt{b}) = 3$$
$$\delta(5, \mathtt{a}) = 4 \qquad \delta(5, \mathtt{b}) = 5$$

Deterministický konečný automat

Místo zápisu

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

budeme raději používat stručnější tabulku nebo grafické znázornění:

δ	a	b
1	2	1
2	4	5
3	1	4
4	1	3
5	4	5

Deterministický konečný automat

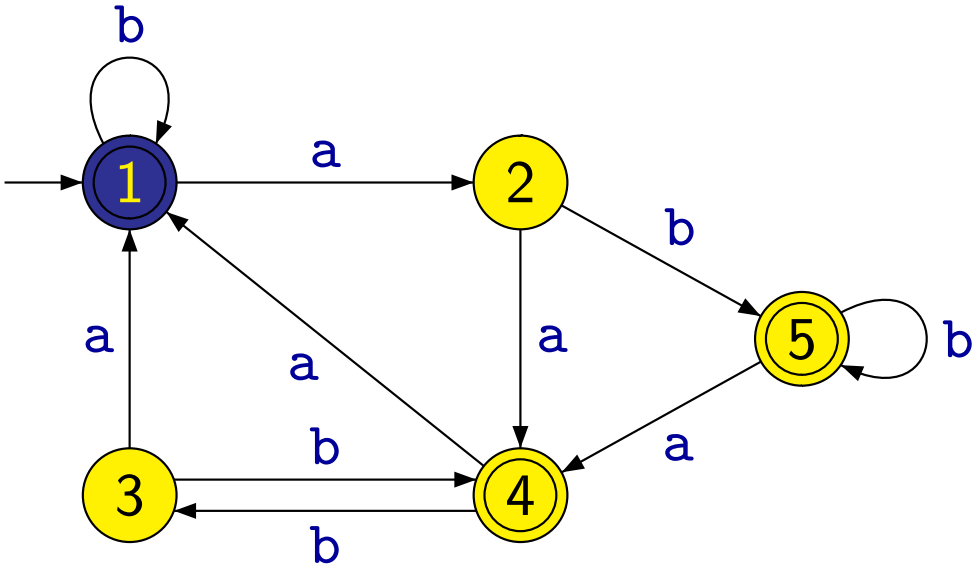
Místo zápisu

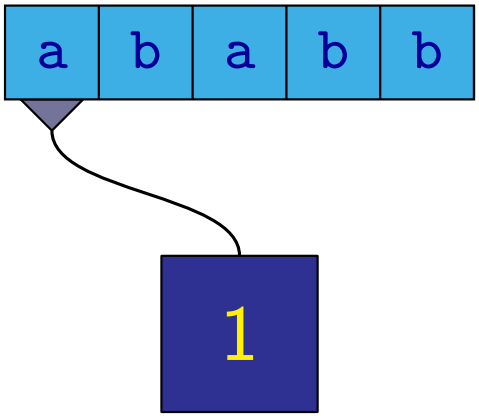
$$\begin{array}{ll} \delta(1,\mathtt{a}) = 2 & \delta(1,\mathtt{b}) = 1 \\ \delta(2,\mathtt{a}) = 4 & \delta(2,\mathtt{b}) = 5 \\ \delta(3,\mathtt{a}) = 1 & \delta(3,\mathtt{b}) = 4 \\ \delta(4,\mathtt{a}) = 1 & \delta(4,\mathtt{b}) = 3 \\ \delta(5,\mathtt{a}) = 4 & \delta(5,\mathtt{b}) = 5 \end{array}$$

budeme raději používat stručnější tabulku nebo grafické znázornění:

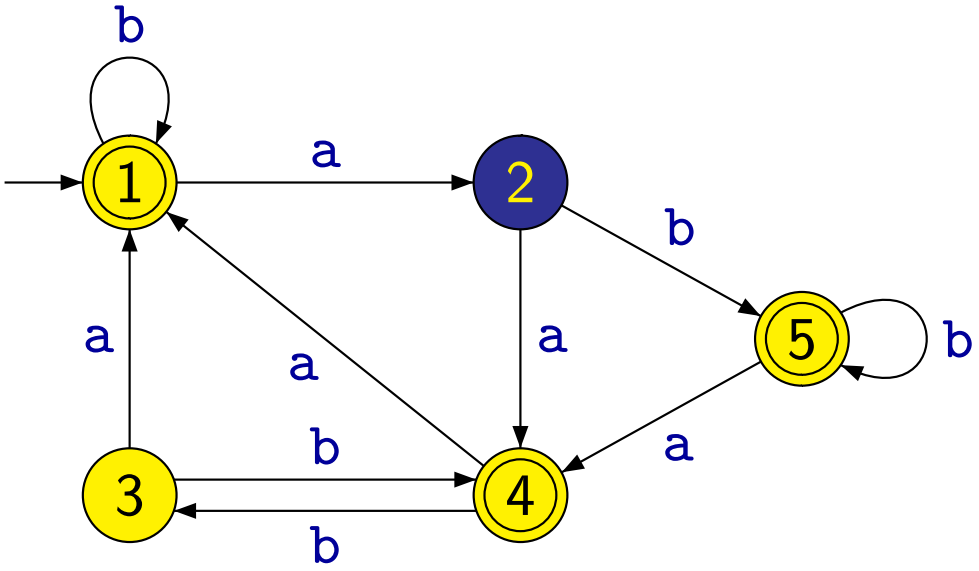
δ	a	b
$\leftrightarrow 1$	2	1
2	4	5
3	1	4
$\leftarrow 4$	1	3
$\leftarrow 5$	4	5

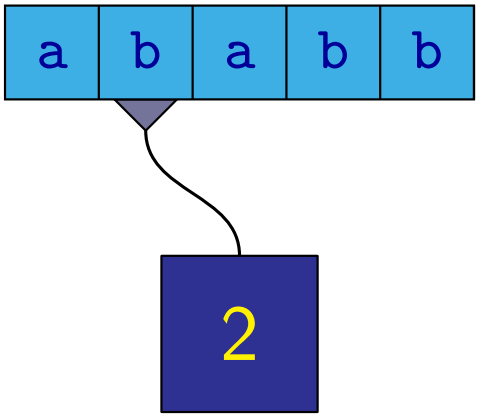
Deterministický konečný automat



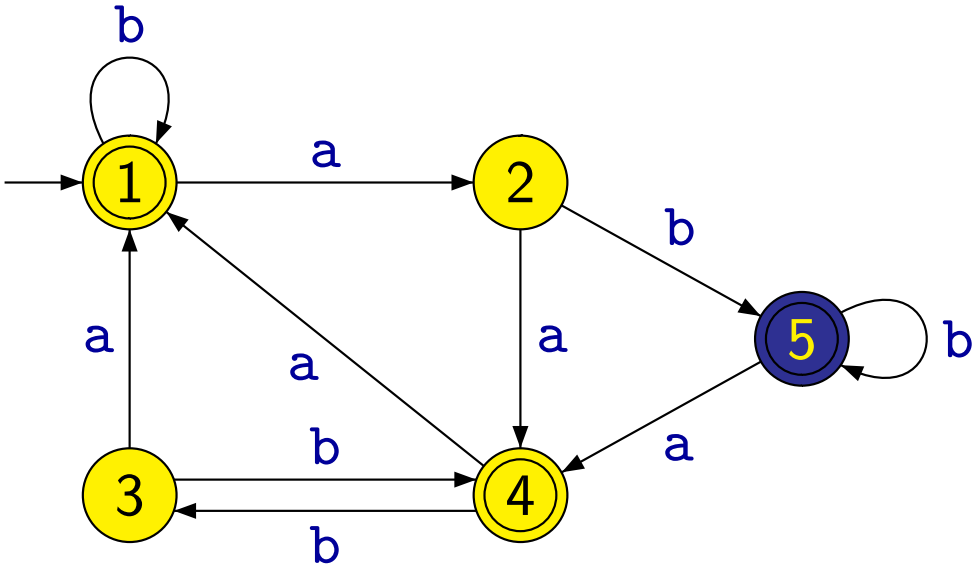


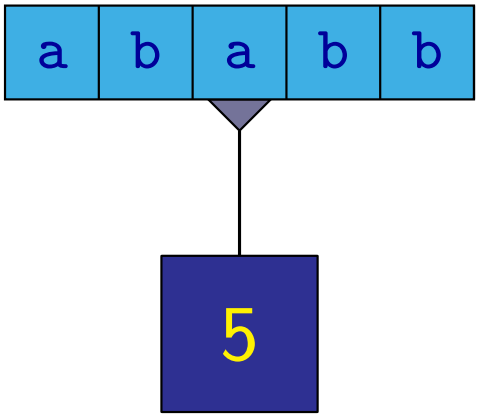
Deterministický konečný automat



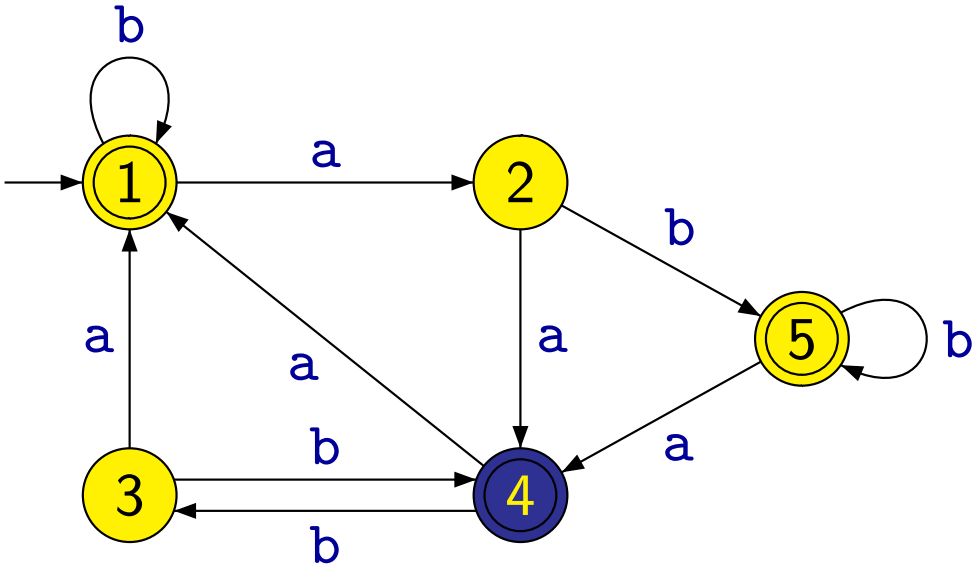


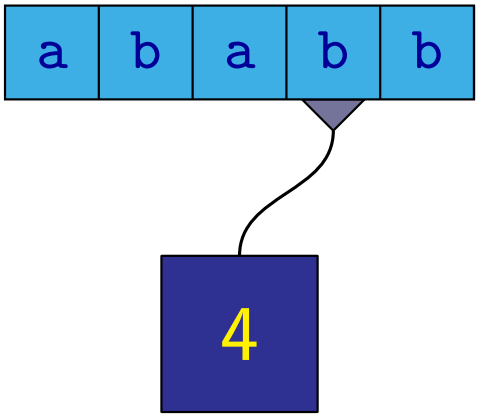
Deterministický konečný automat



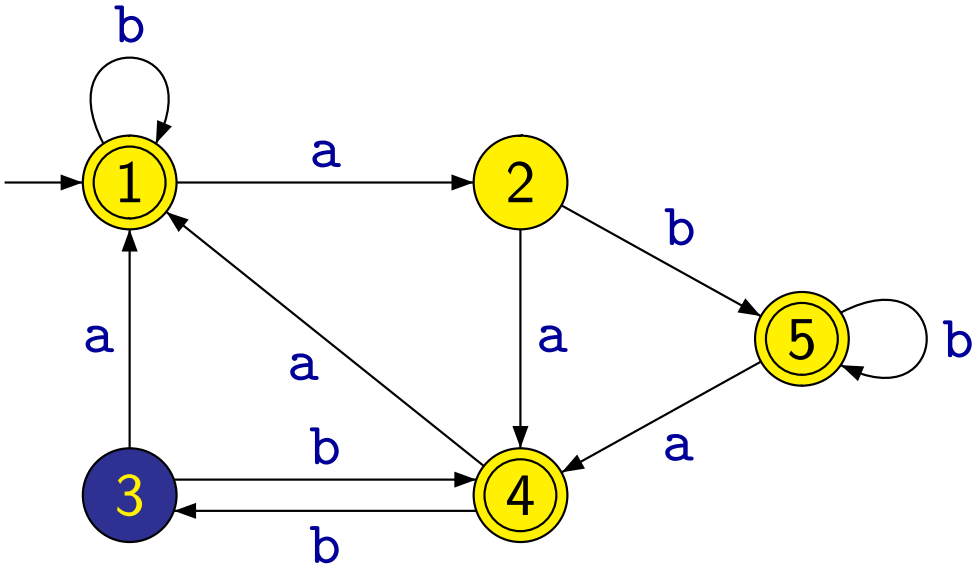


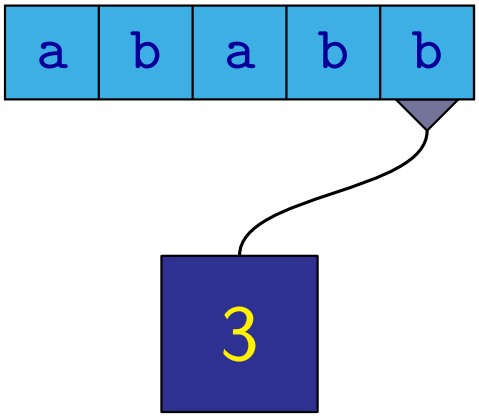
Deterministický konečný automat



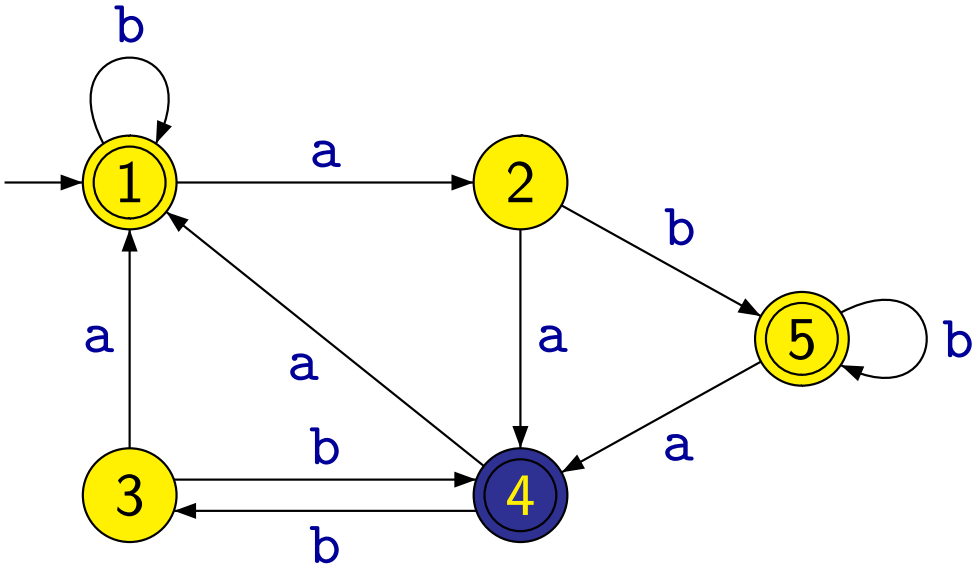


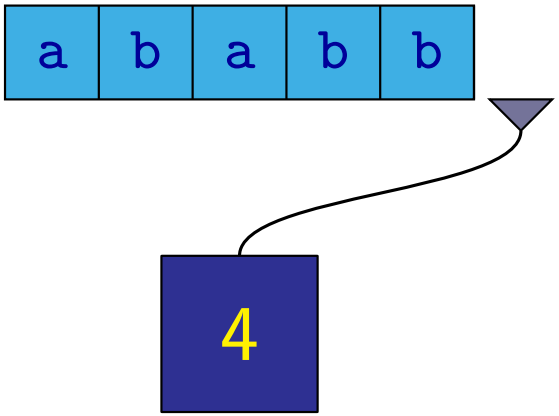
Deterministický konečný automat





Deterministický konečný automat





Deterministický konečný automat

Konfigurace konečného automatu je dána stavem jeho řídící jednotky a dosud nepřečteným obsahem pásky.

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

Příklad: $(2, \mathtt{babb})$ 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:

$$(2, \mathtt{babb}) \vdash (5, \mathtt{abb})$$

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: $(1, \texttt{ababb})$ je počáteční konfigurace.

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

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

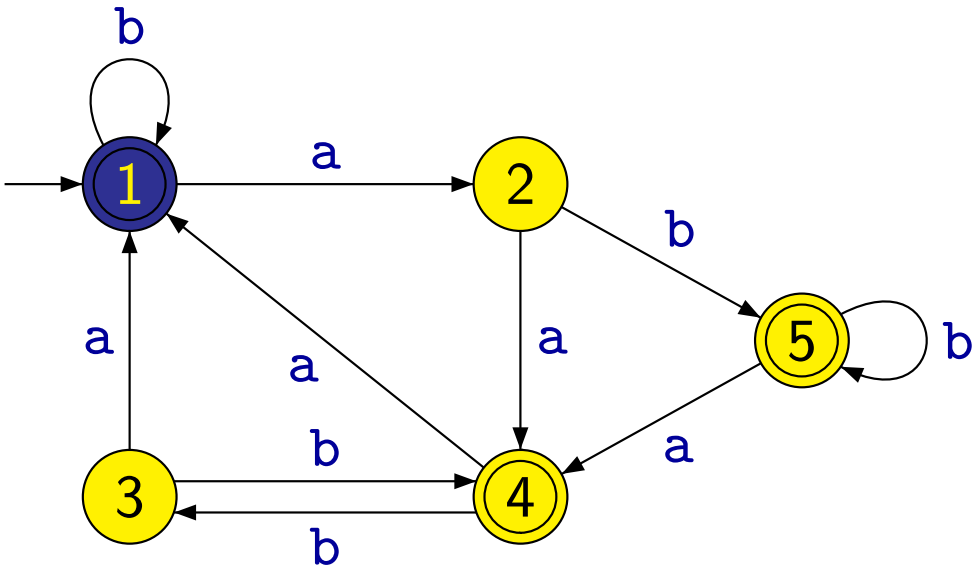
Definice

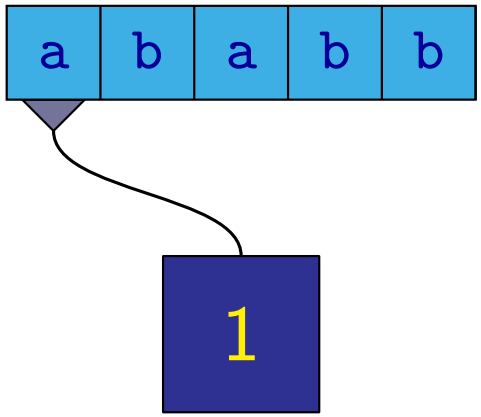
Výpočet automatu je posloupnost konfigurací

$$C_0, C_1, C_2, \cdots, C_k$$

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

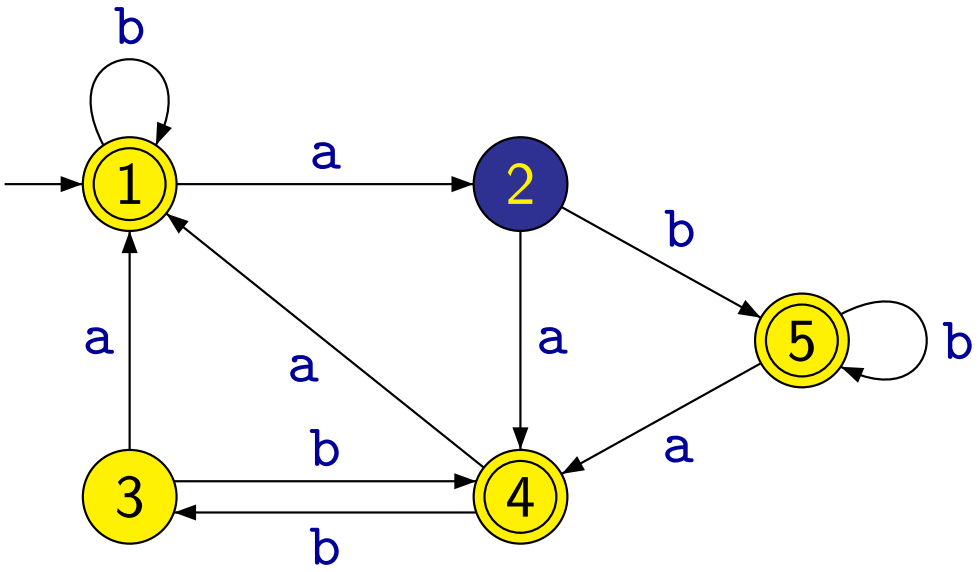
Deterministický konečný automat

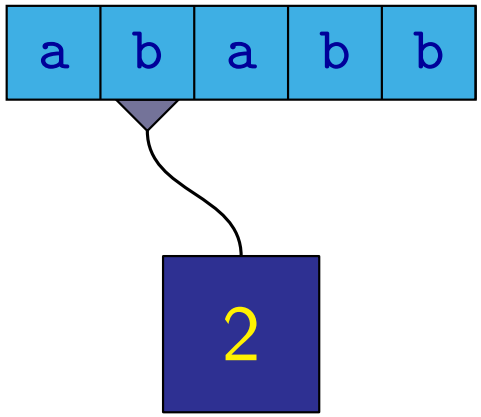




$(1, \mathtt{ababb})$

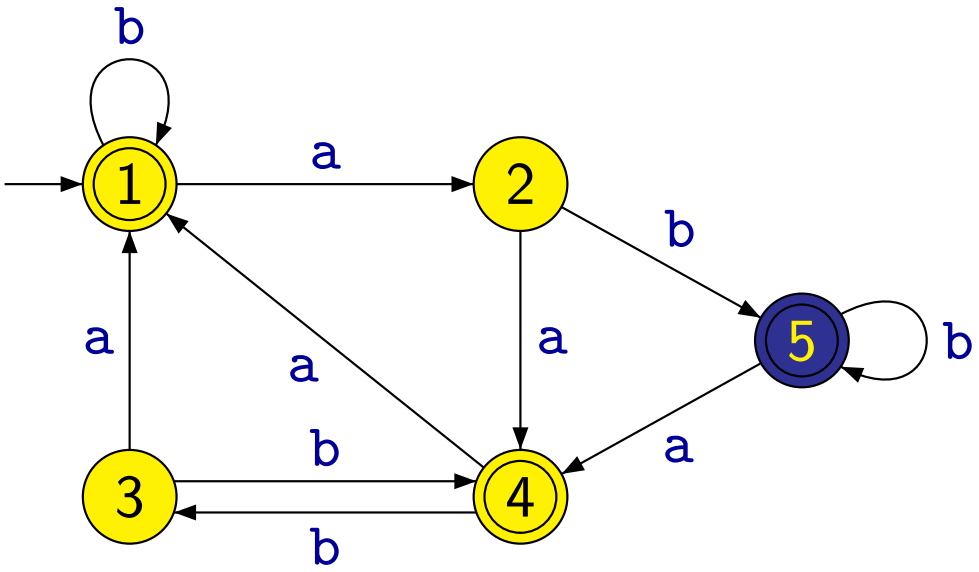
Deterministický konečný automat

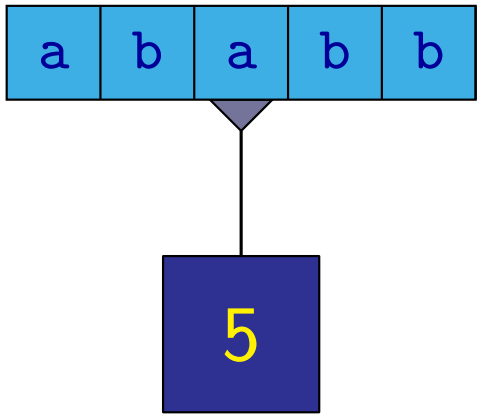




$(1, \mathtt{ababb}) \vdash (2, \mathtt{babb})$

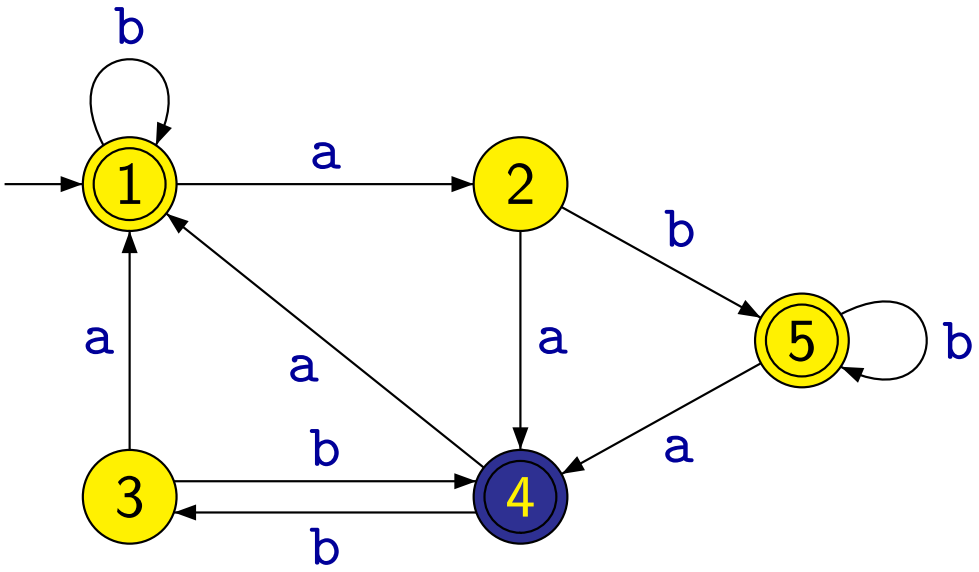
Deterministický konečný automat

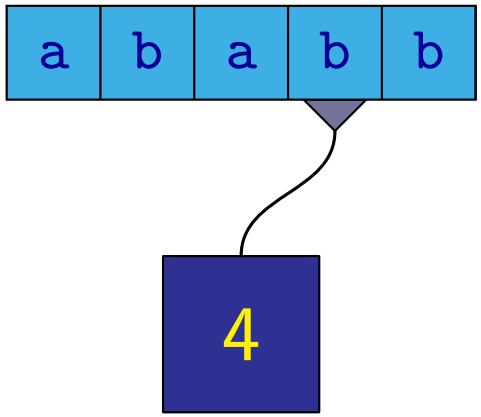




$(1, \mathtt{ababb}) \vdash (2, \mathtt{babb}) \vdash (5, \mathtt{abb})$

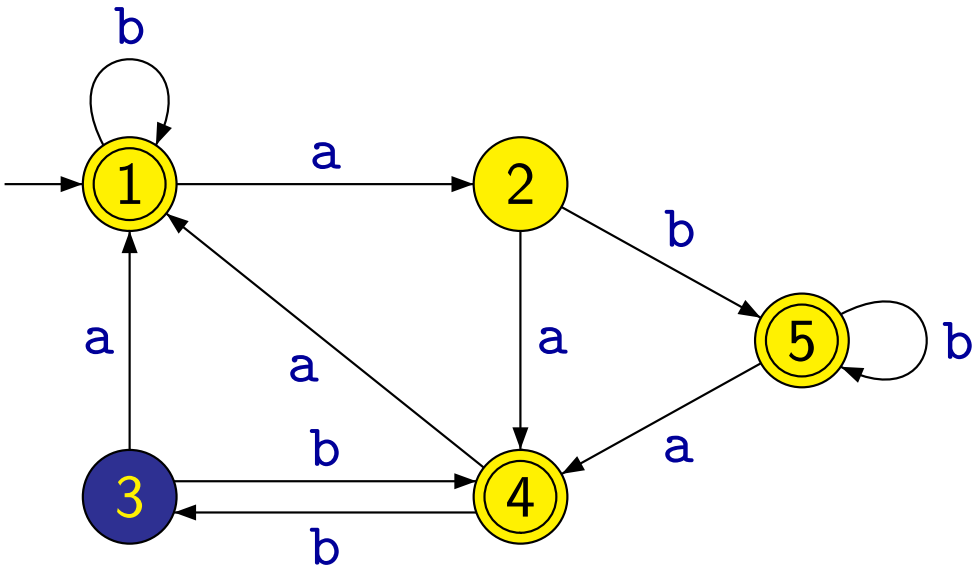
Deterministický konečný automat

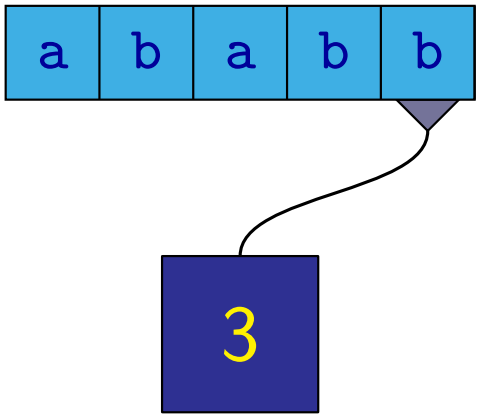




$(1, \mathtt{ababb}) \vdash (2, \mathtt{babb}) \vdash$
$(5, \mathtt{abb}) \vdash (4, \mathtt{bb})$

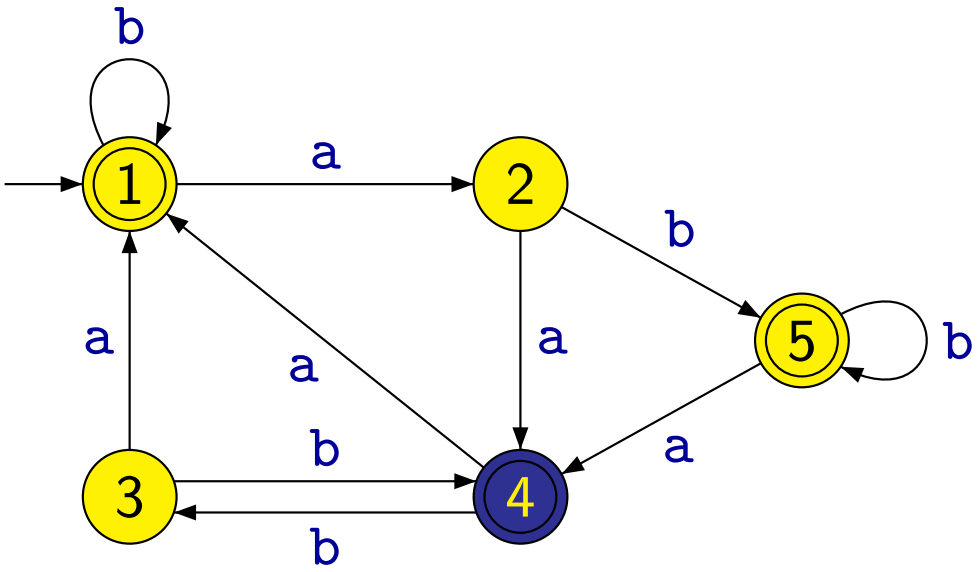
Deterministický konečný automat

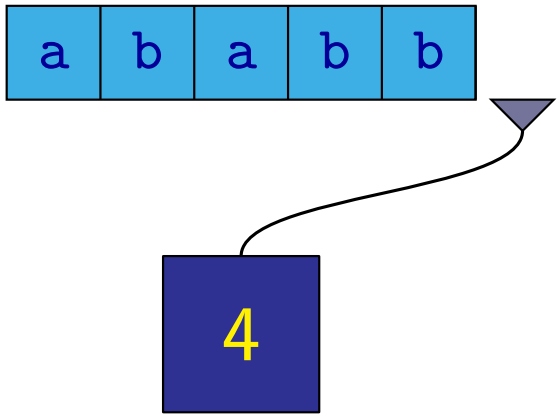




$(1, \mathtt{ababb}) \vdash (2, \mathtt{babb}) \vdash$
$(5, \mathtt{abb}) \vdash (4, \mathtt{bb}) \vdash$
$(3, \mathtt{b})$

Deterministický konečný automat





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

Deterministický konečný automat

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

$$C_0 \vdash C_1 \vdash C_2 \vdash \cdots \vdash C_k$$

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

Deterministický konečný automat

Definice

Koncová konfigurace (q, ε) 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 $q \in F$.

Deterministický konečný automat

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.

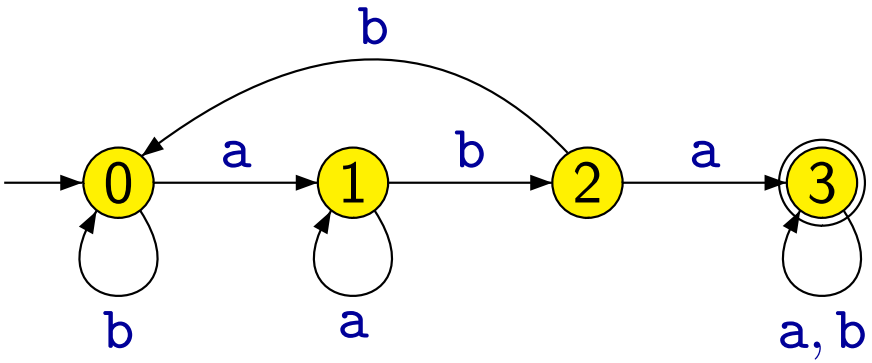
$$L(A) = \{w \in \Sigma^* \mid (q_0, w) \vdash^* (q, \varepsilon),\ q \in F\}$$

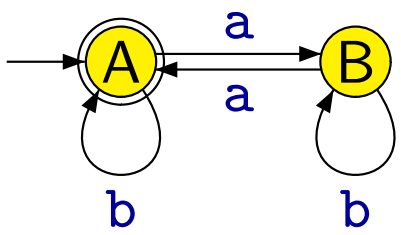
Definice

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

Automat pro průnik jazyků

Máme následující dva automaty:

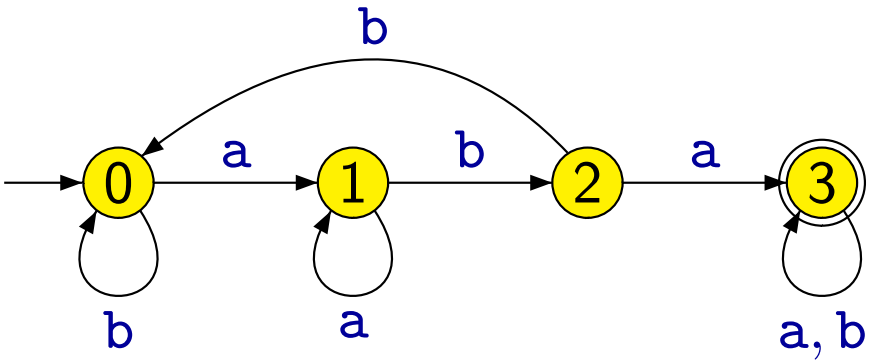


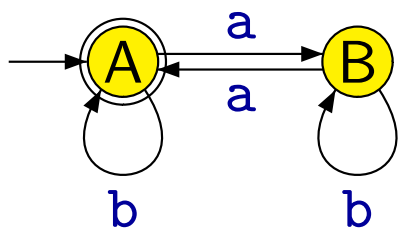


Příjmou oba slovo ababb?

Automat pro průnik jazyků

Máme následující dva automaty:



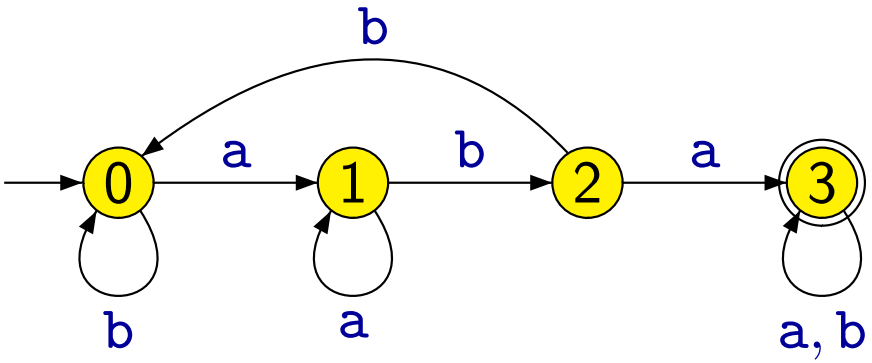


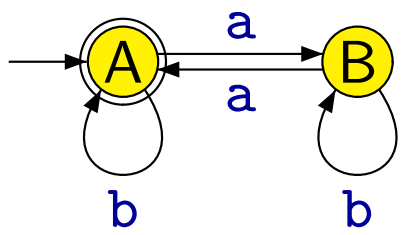
Příjmou oba slovo `ababb`?

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

Automat pro průnik jazyků

Máme následující dva automaty:



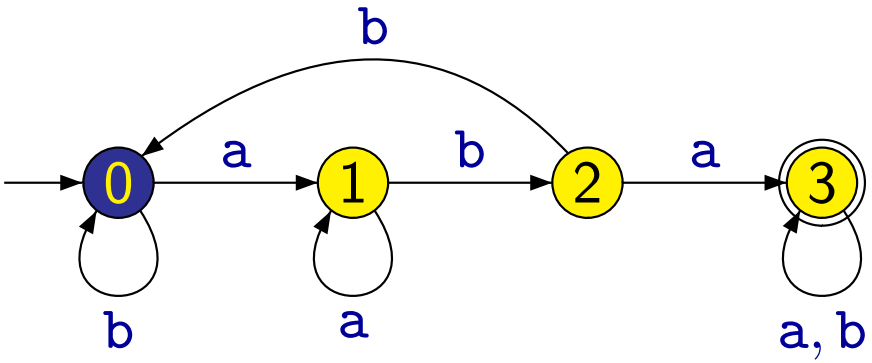


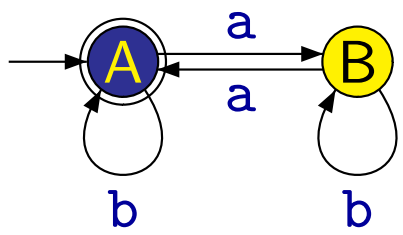
Příjmou oba slovo ababb?

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:



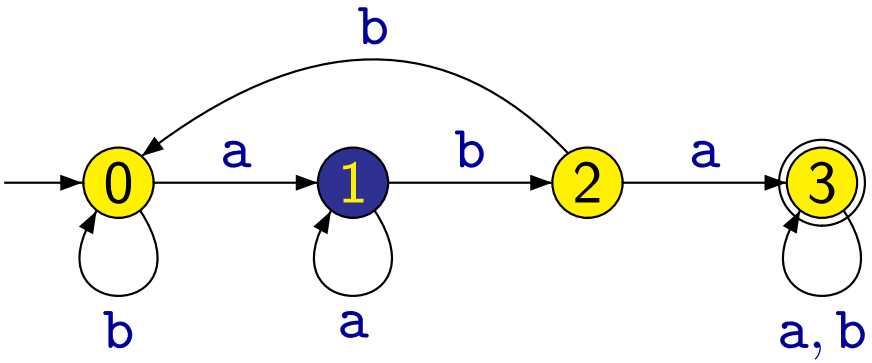


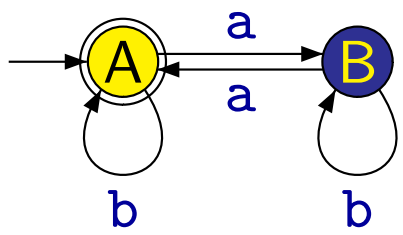
Příjmou oba slovo ababb?

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:



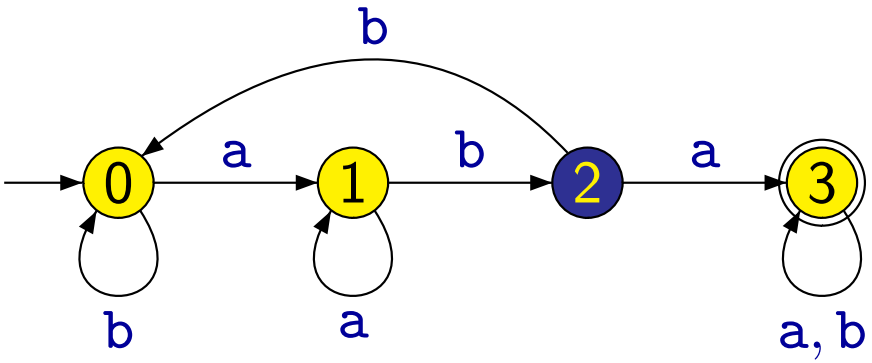


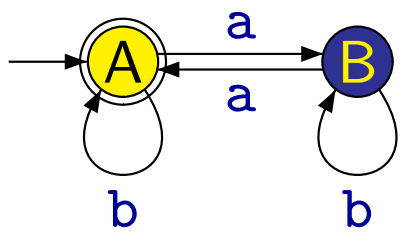
Příjmou oba slovo ababb?

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:



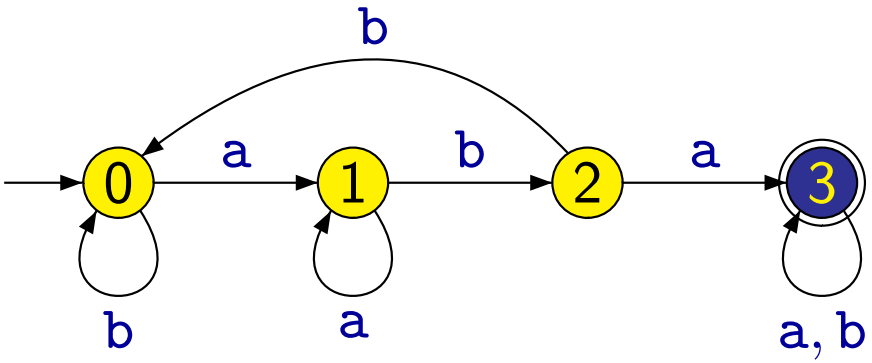


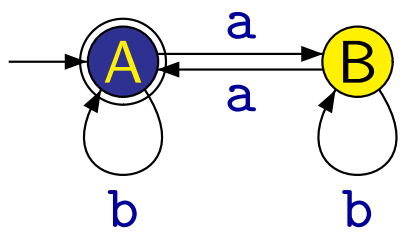
Příjmou oba slovo ababb?

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:



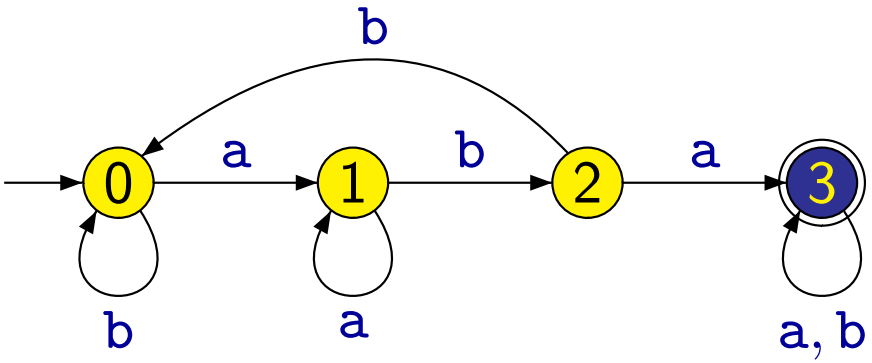


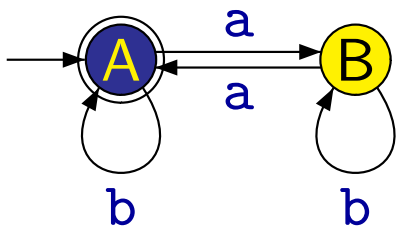
Příjmou oba slovo ababb?

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:



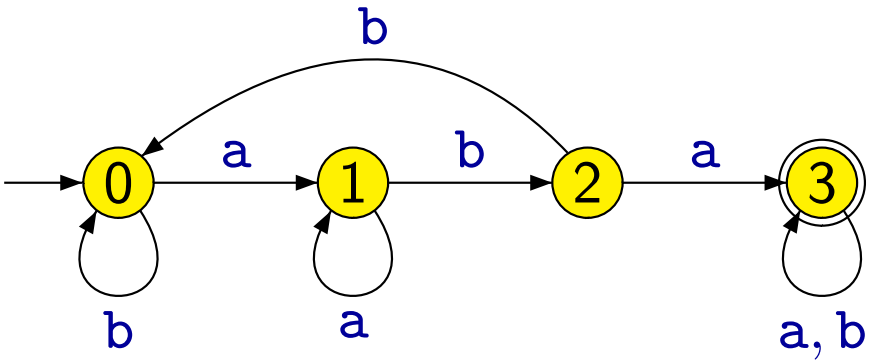


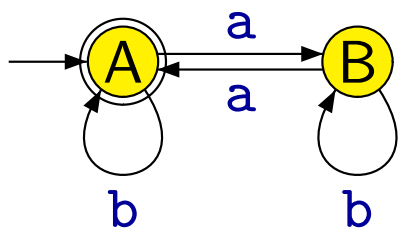
Příjmou oba slovo ababb?

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:

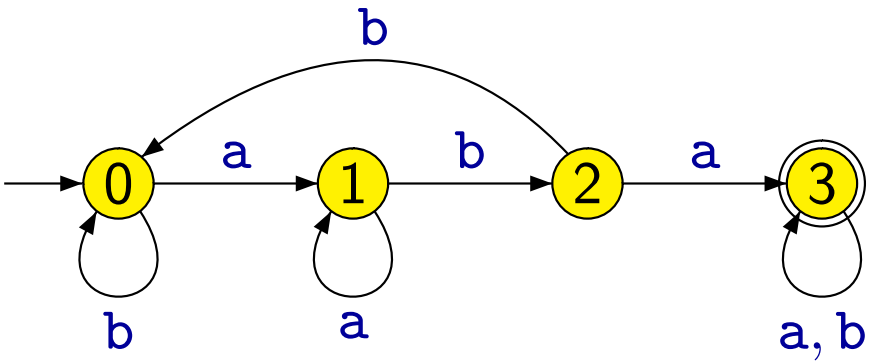


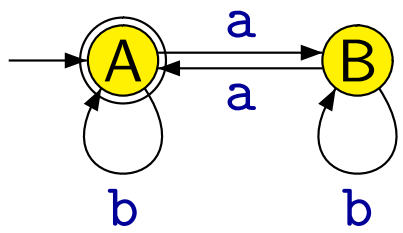


Příjmou oba slovo `ababb`?

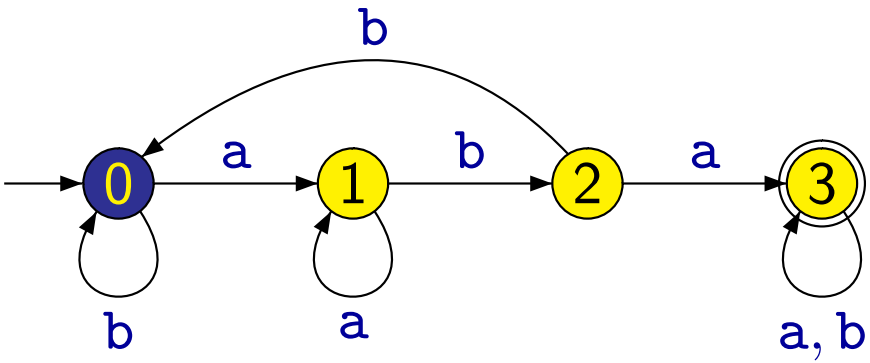
Můžeme postupně nechat přečíst slovo oběma automatům. Odpověď bude ano, pokud oba odpoví ano.
Lepší je číst slovo současně oběma automaty.
Situace při tomto postupu je dána nepřečtenou částí slova a aktuálními stavy obou automatů. Zkusíme vytvořit automat, který toto má jako své konfigurace.

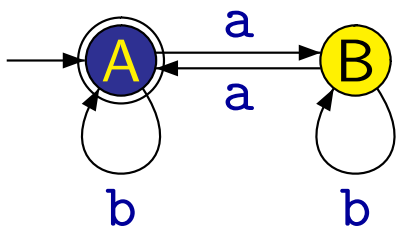
Automat pro průnik jazyků

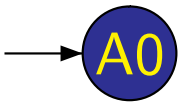




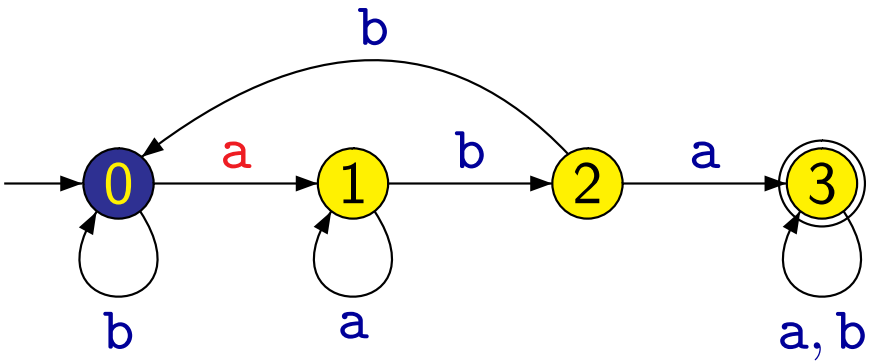
Automat pro průnik jazyků

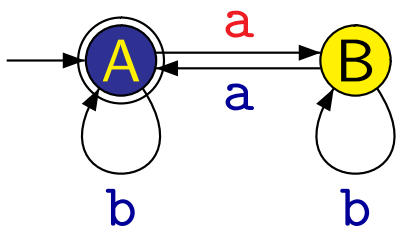




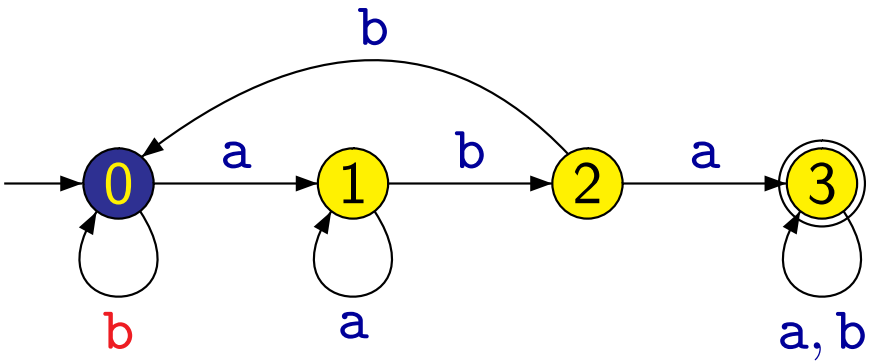


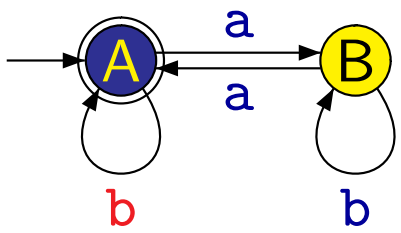
Automat pro průnik jazyků

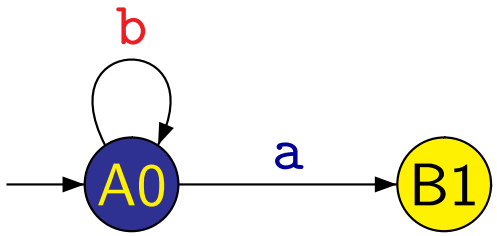




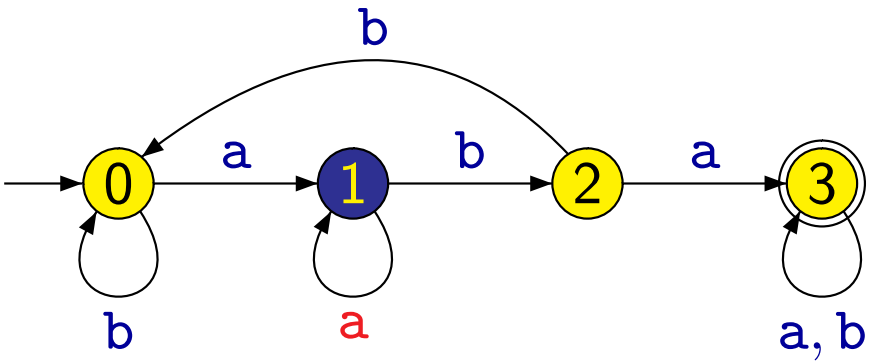
Automat pro průnik jazyků

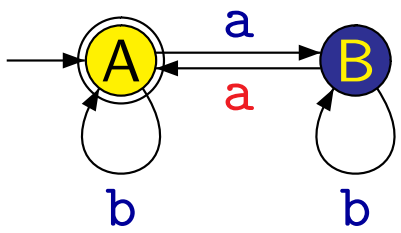


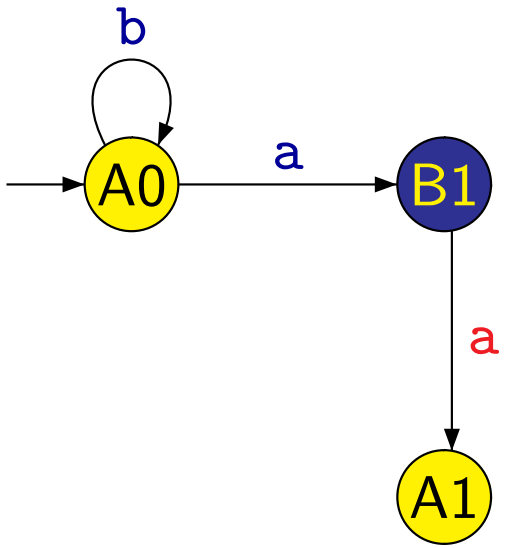




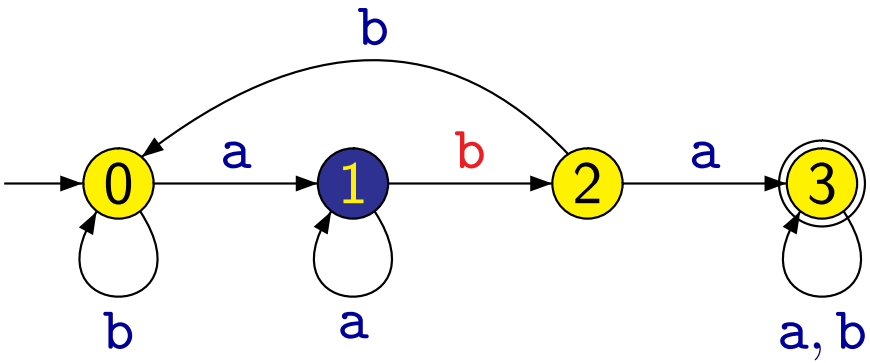
Automat pro průnik jazyků

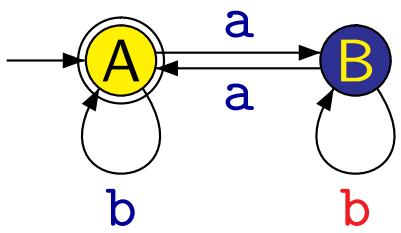


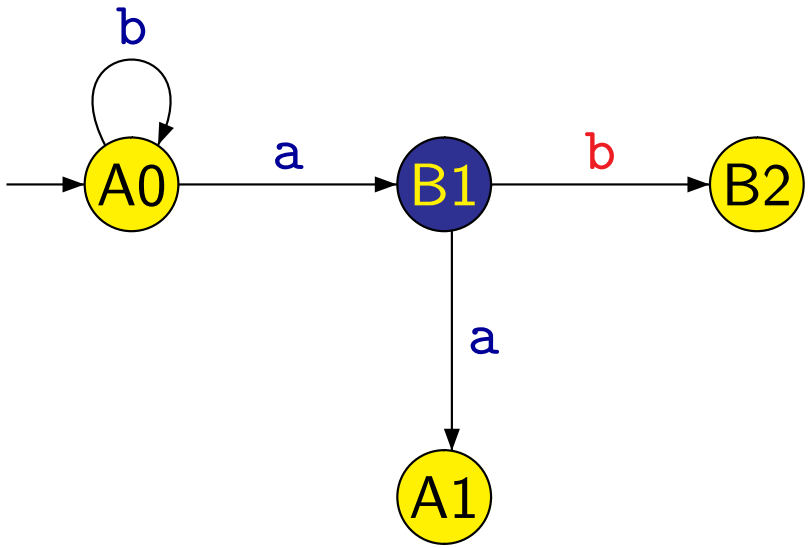




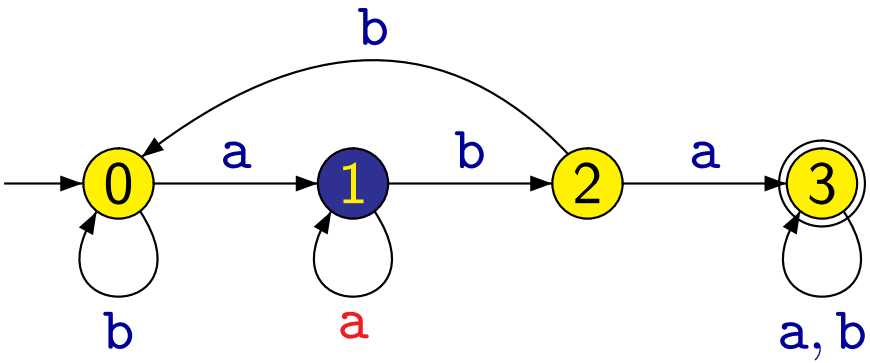
Automat pro průnik jazyků

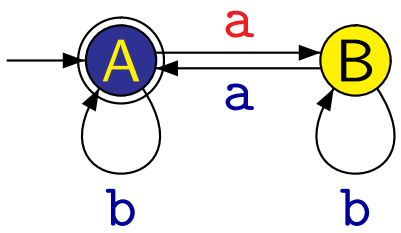


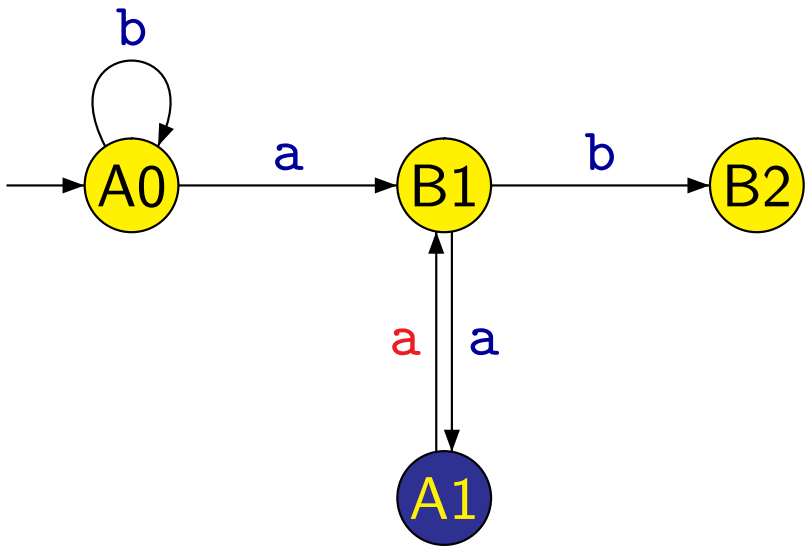




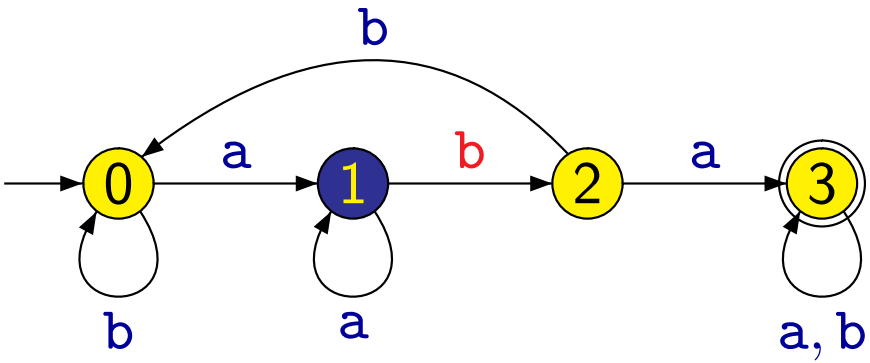
Automat pro průnik jazyků

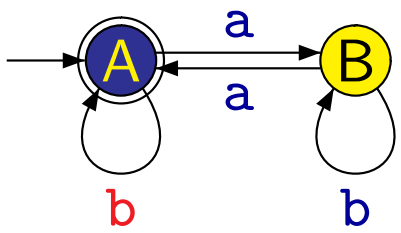


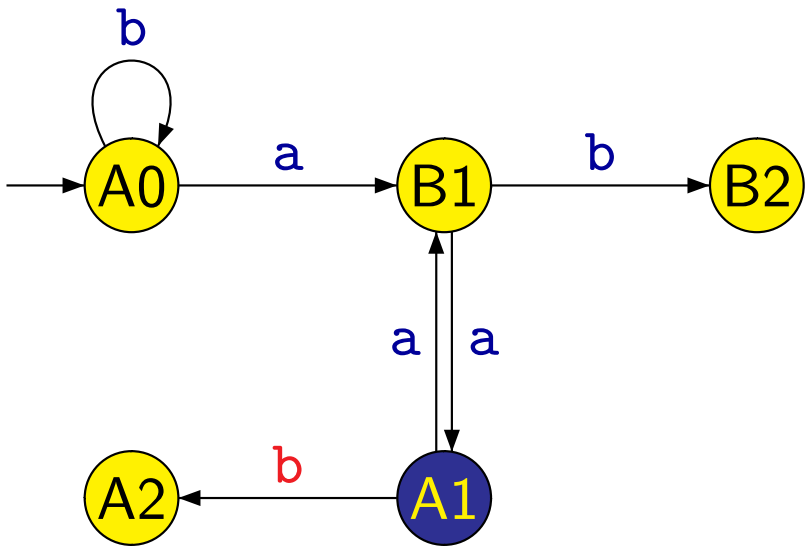




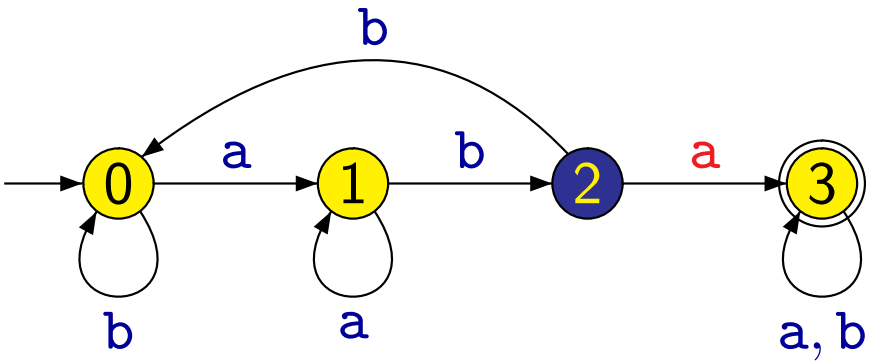
Automat pro průnik jazyků

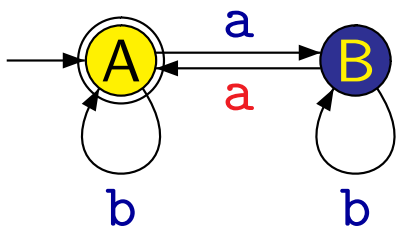


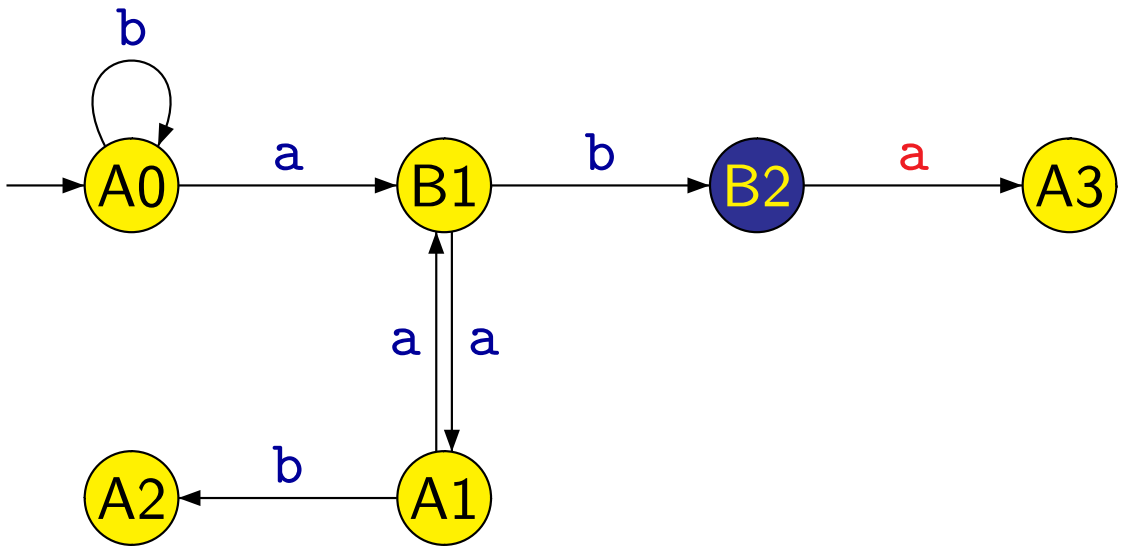




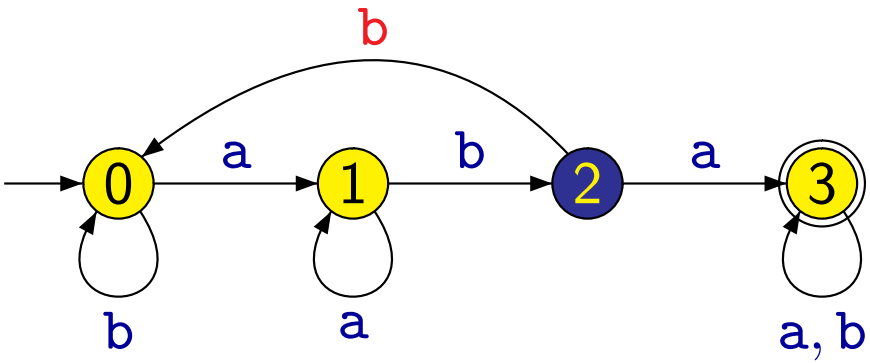
Automat pro průnik jazyků

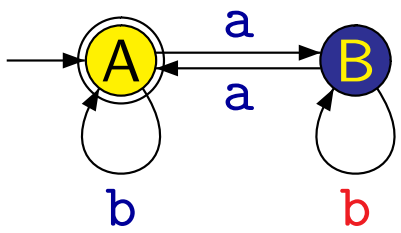


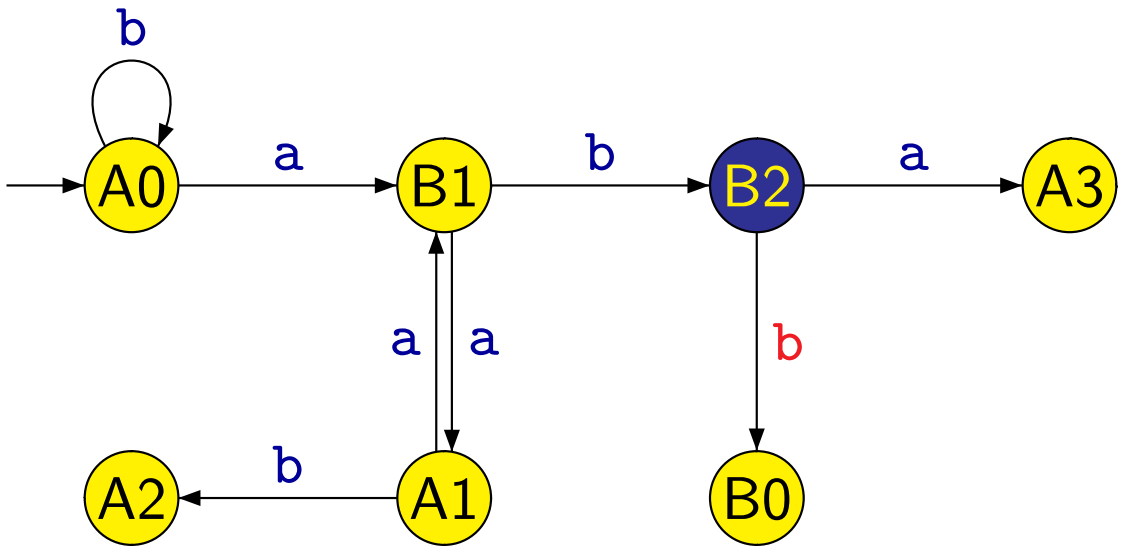




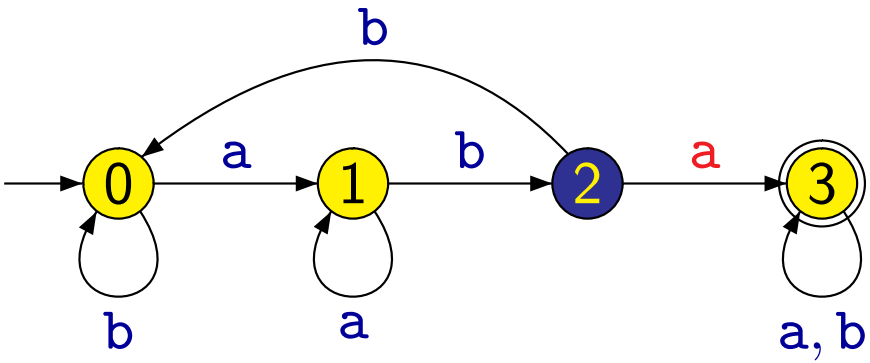
Automat pro průnik jazyků

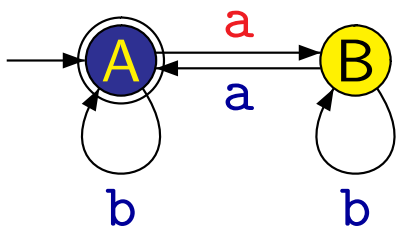


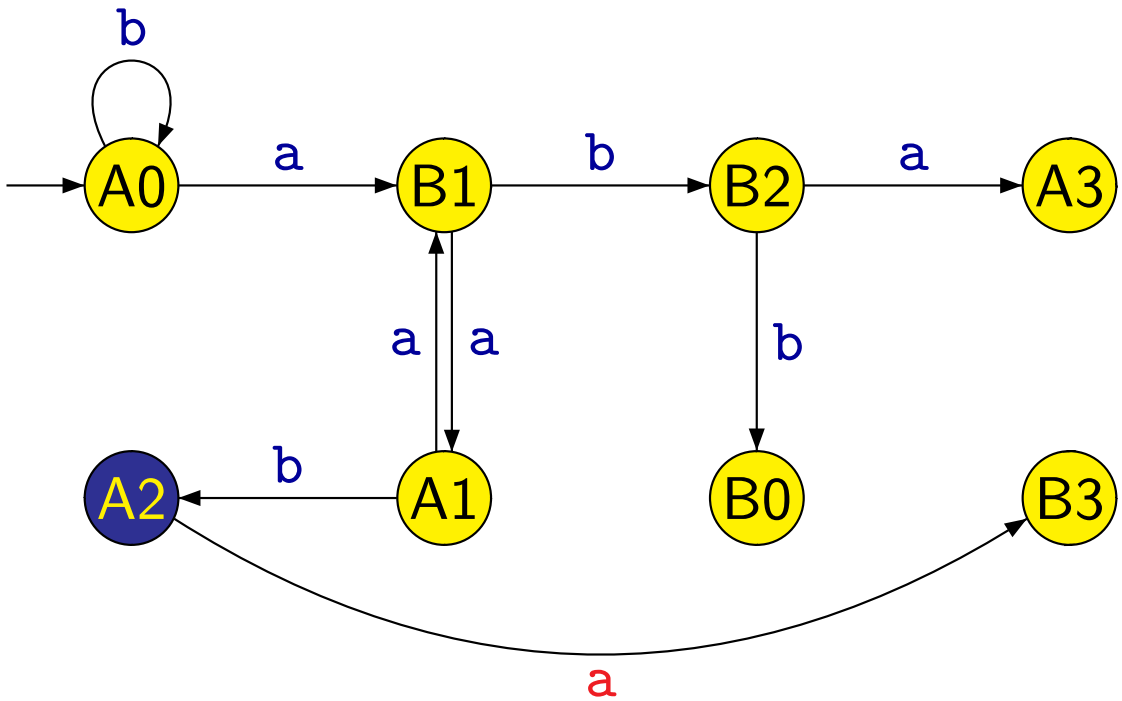




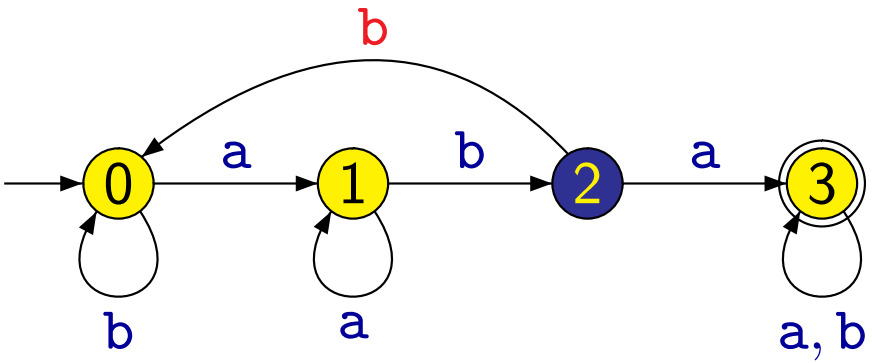
Automat pro průnik jazyků

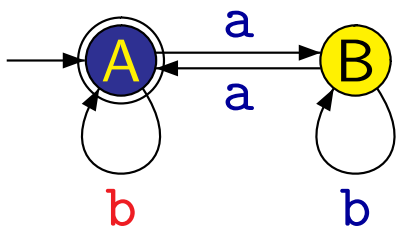


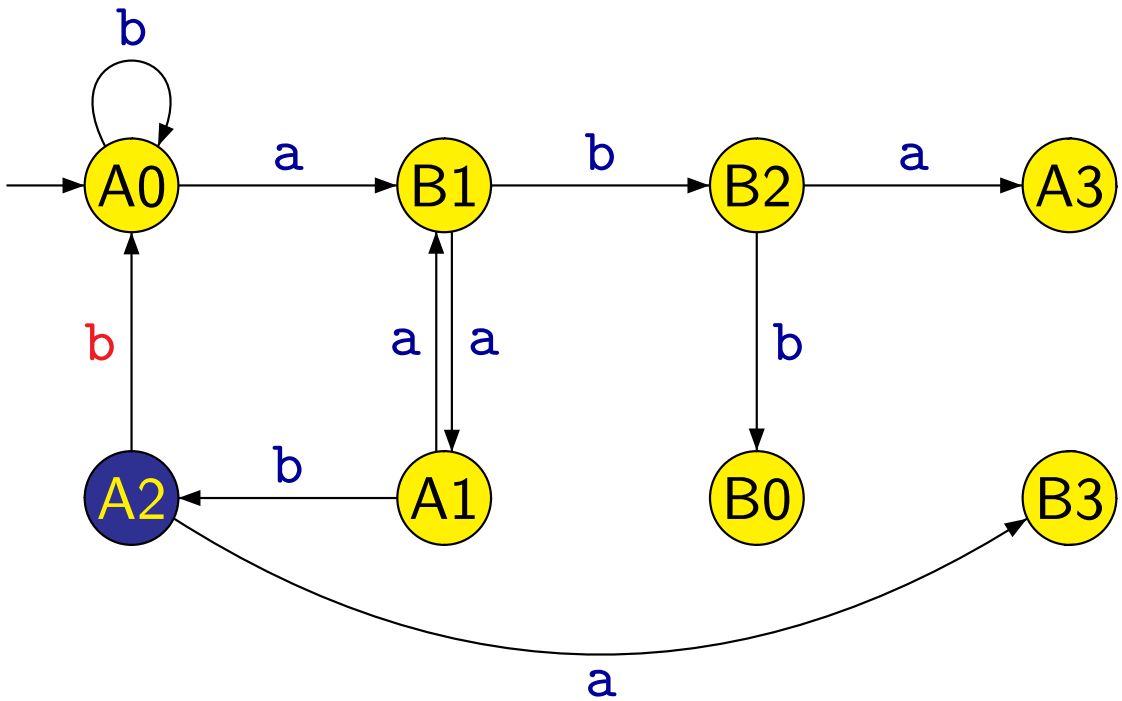




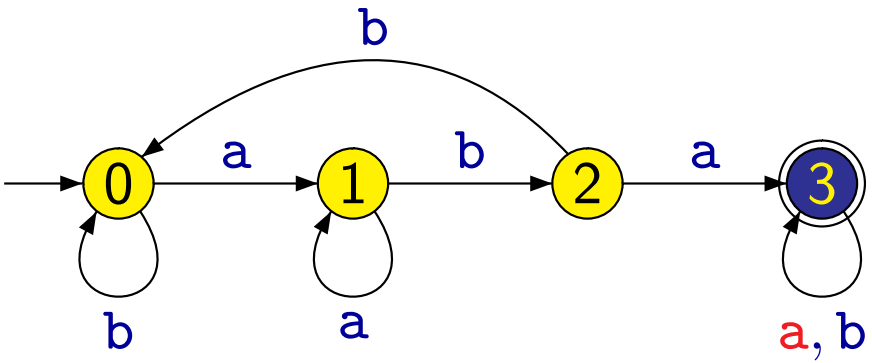
Automat pro průnik jazyků

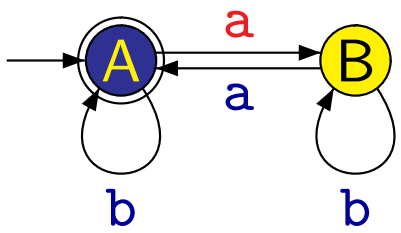


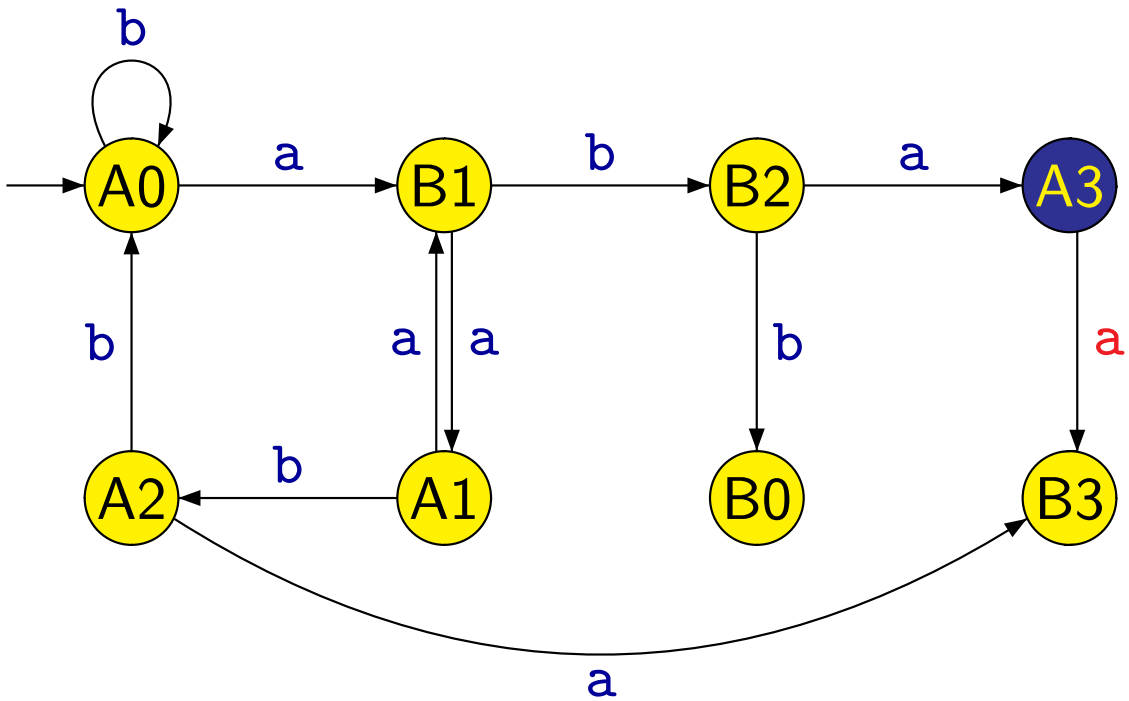




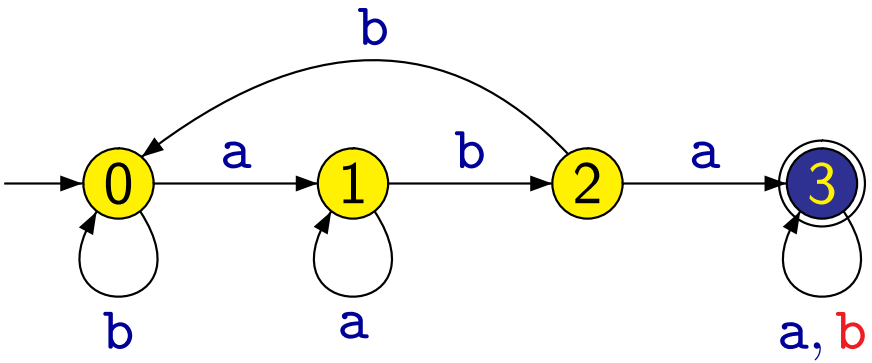
Automat pro průnik jazyků

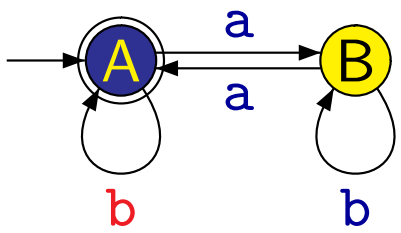


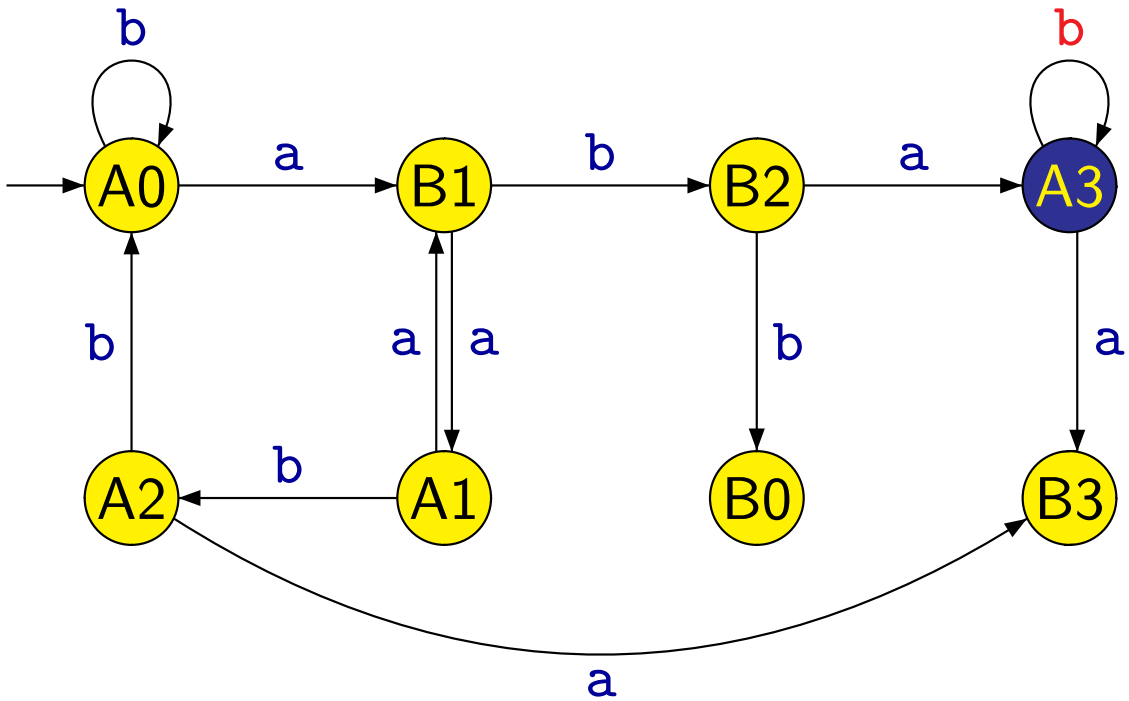




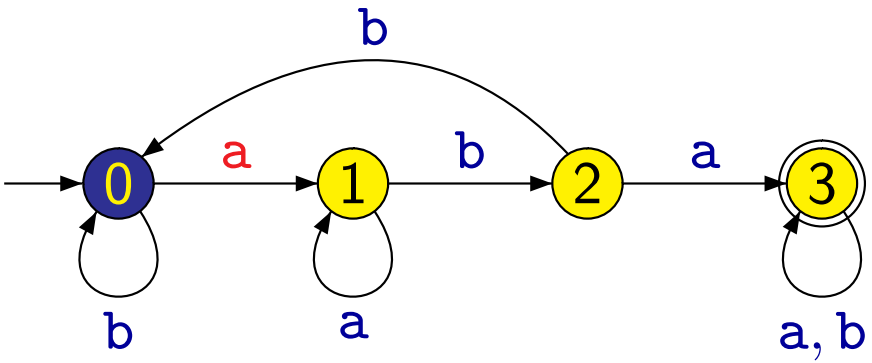
Automat pro průnik jazyků

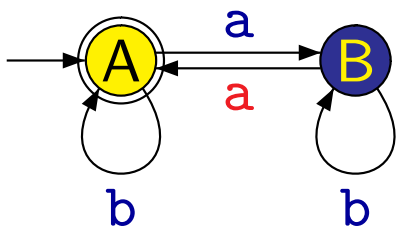


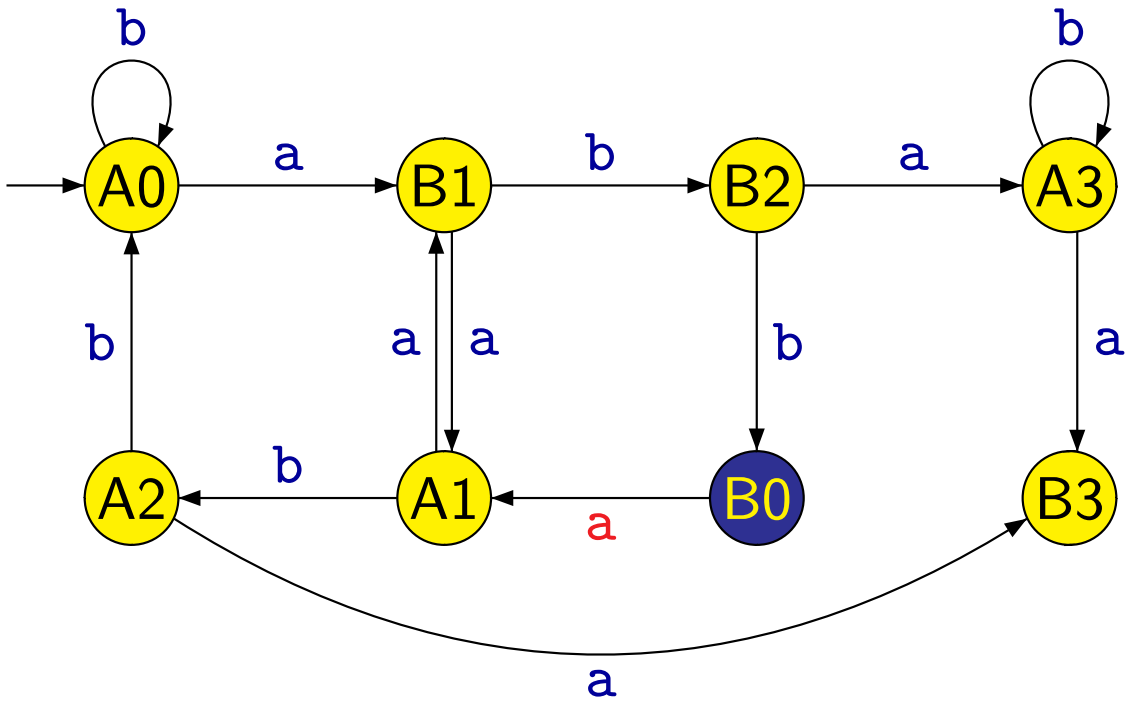




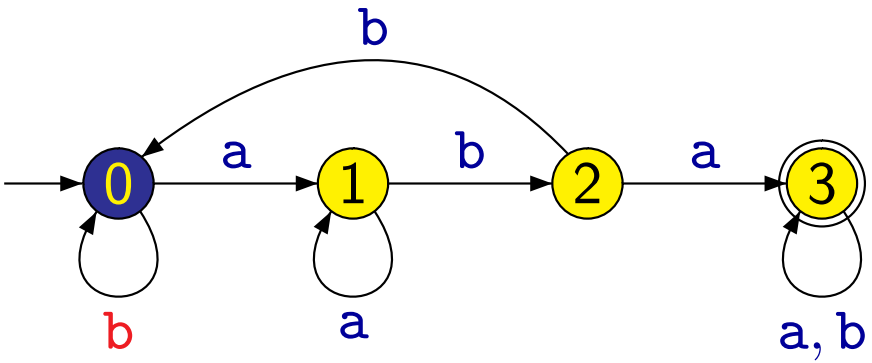
Automat pro průnik jazyků

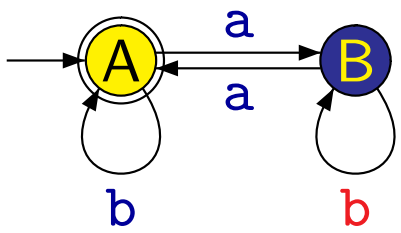


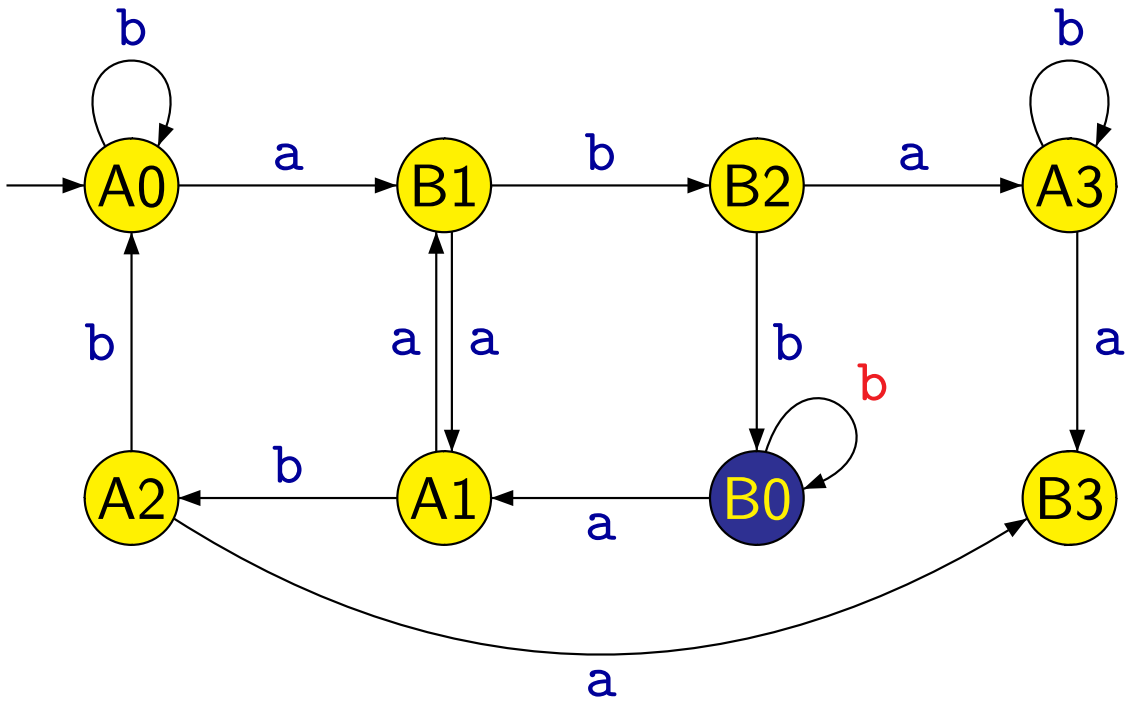




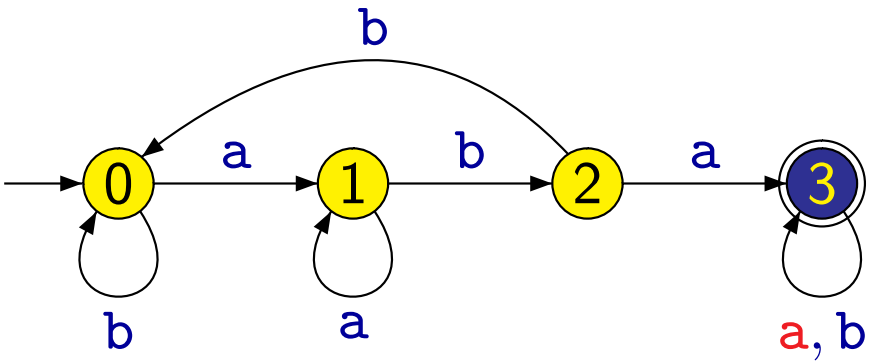
Automat pro průnik jazyků

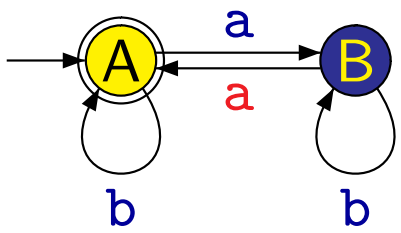


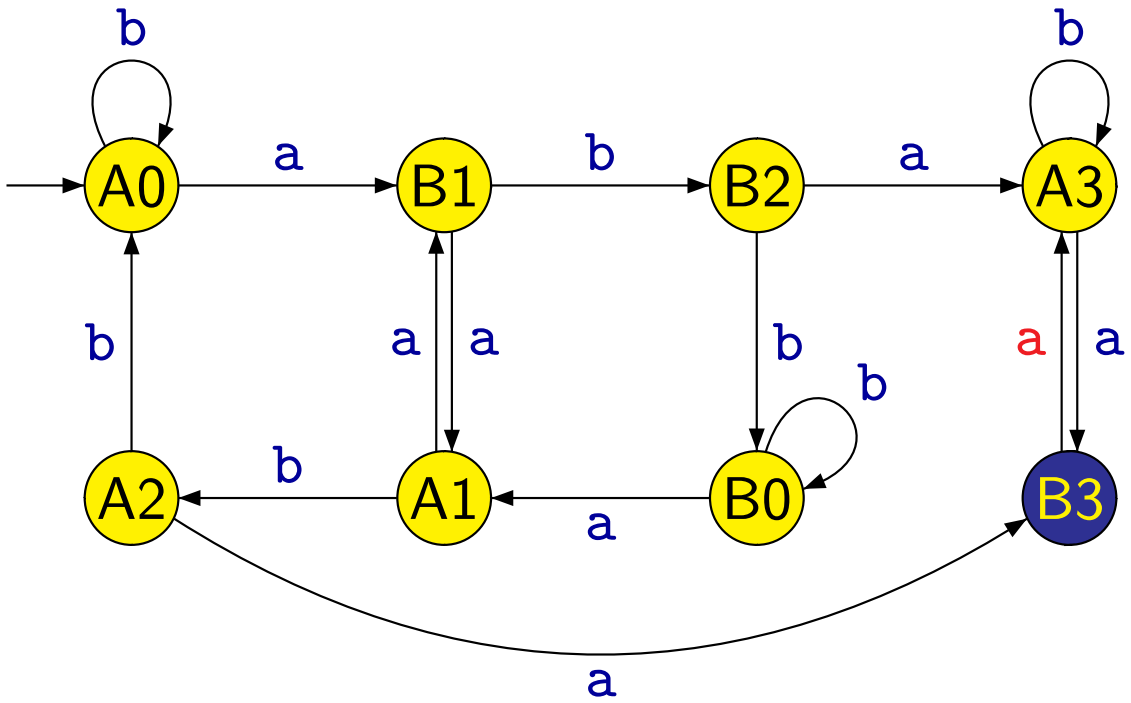




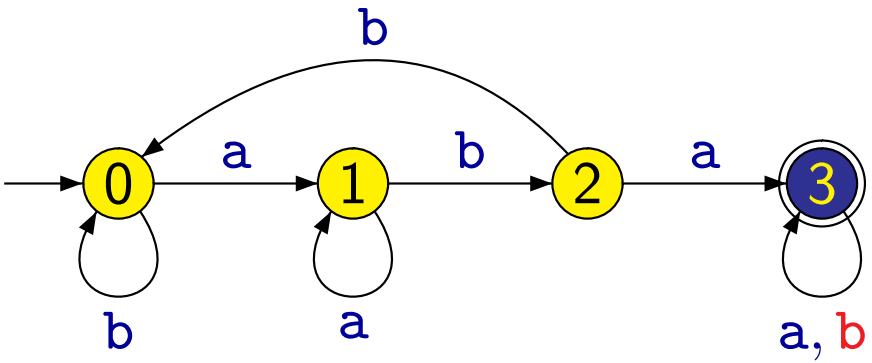
Automat pro průnik jazyků

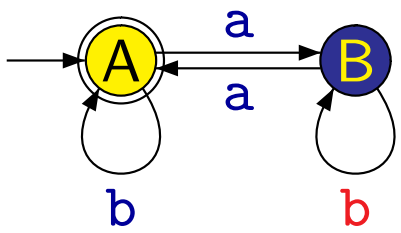


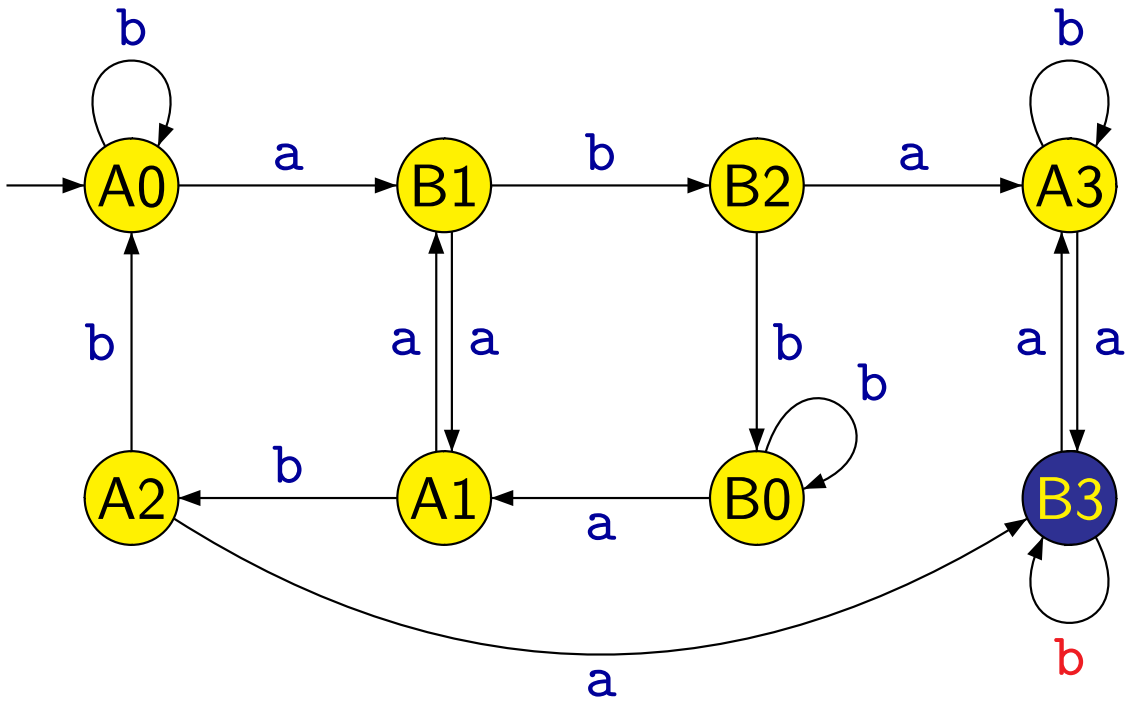




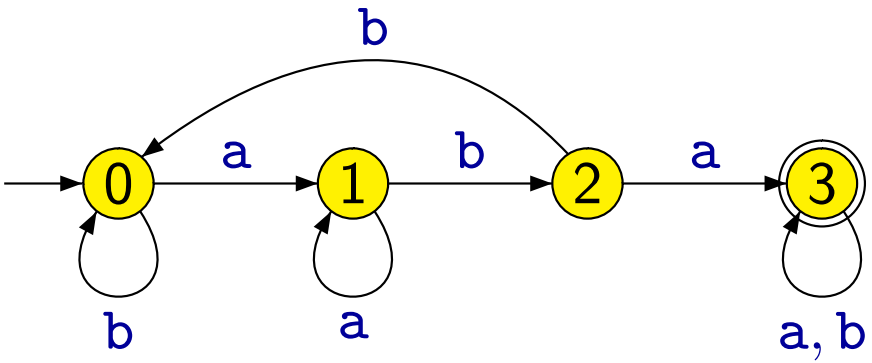
Automat pro průnik jazyků

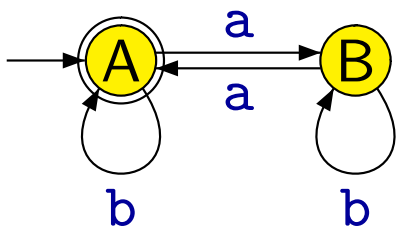


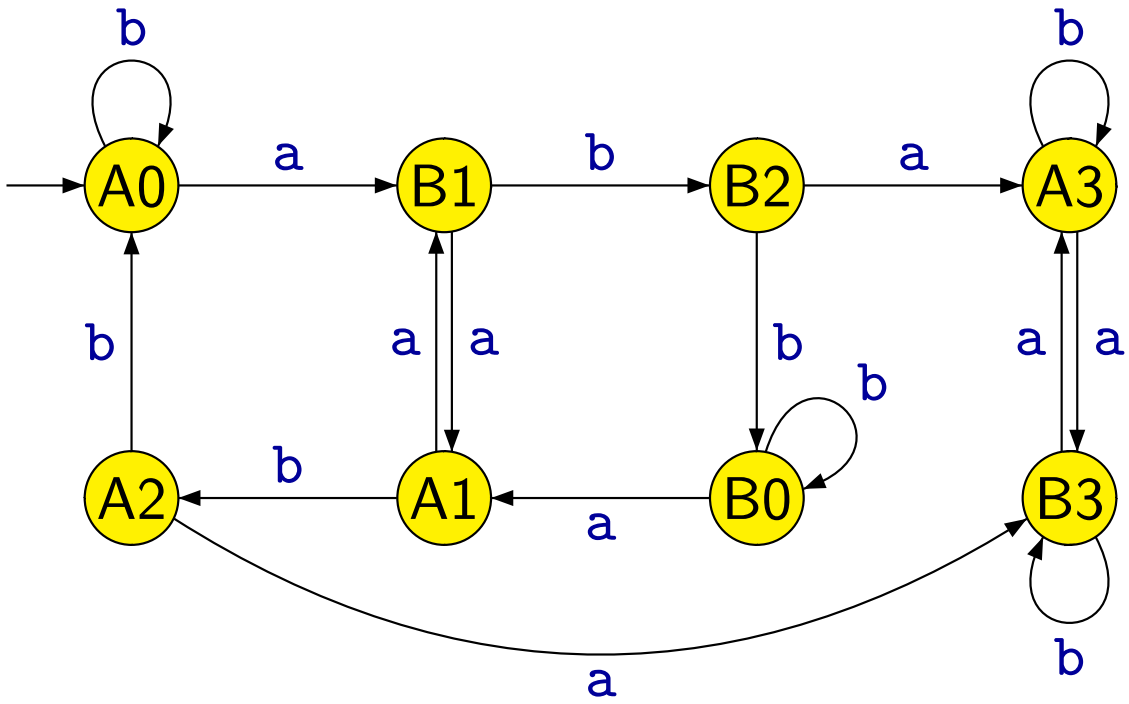




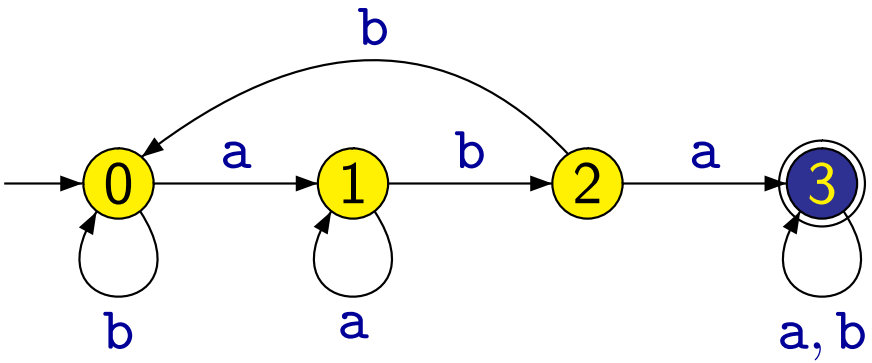
Automat pro průnik jazyků

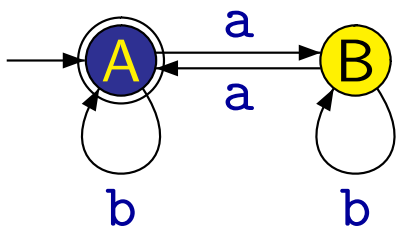


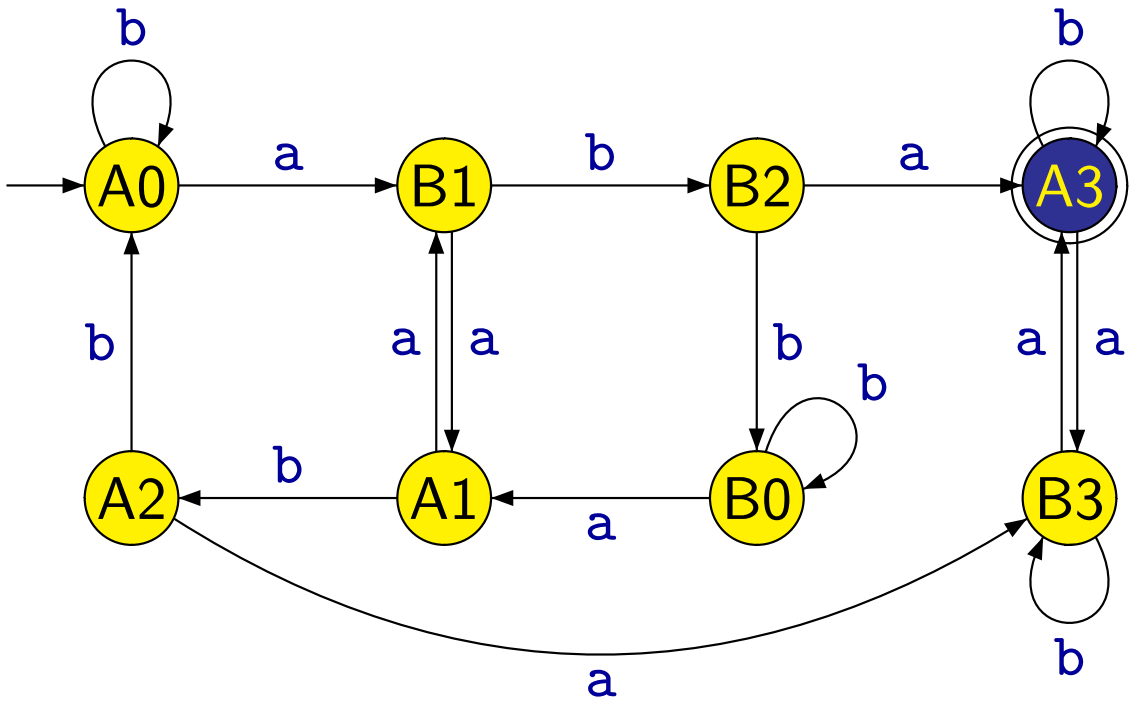




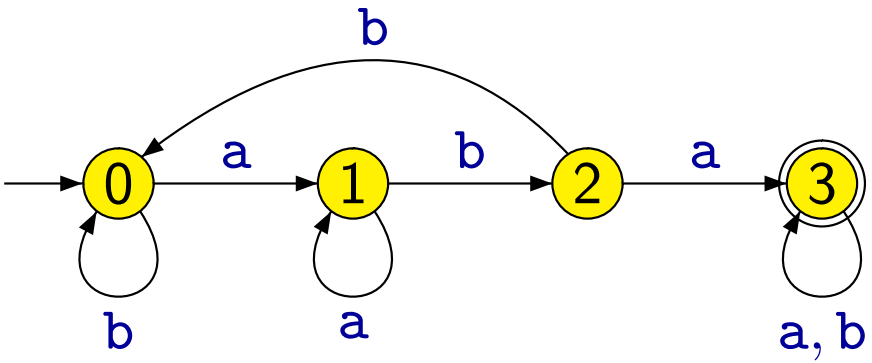
Automat pro průnik jazyků

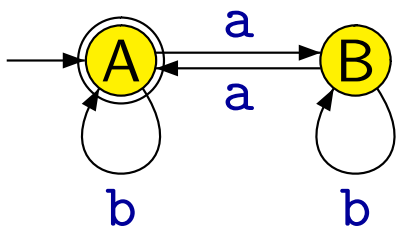


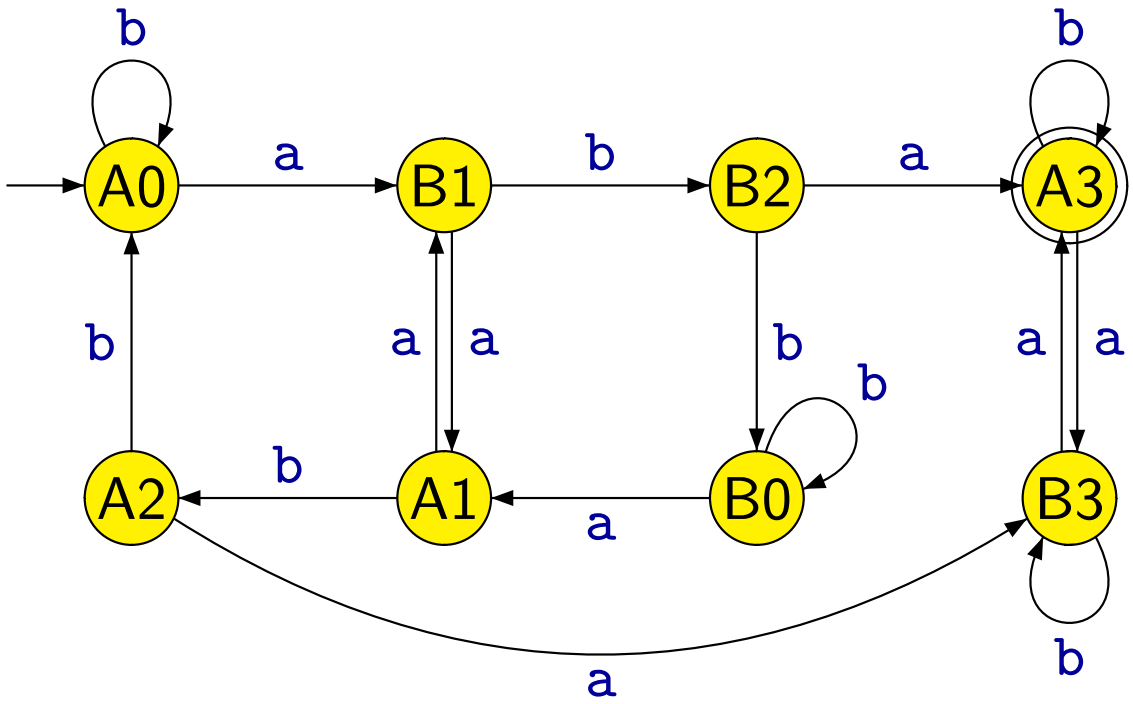




Automat pro průnik jazyků







Průnik regulárních jazyků

Věta

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

Průnik regulárních jazyků

Věta

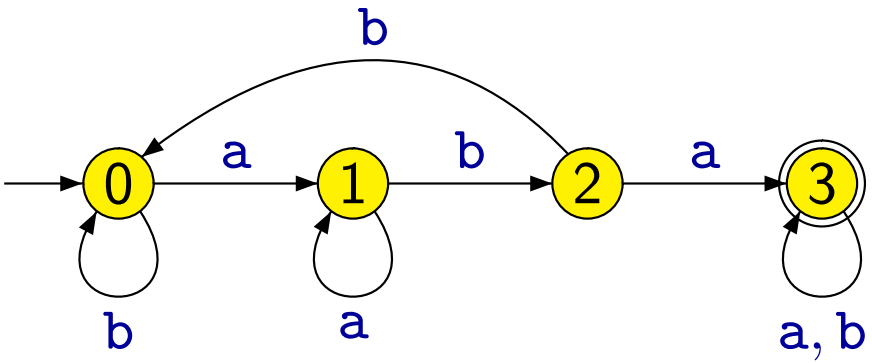
Jestliže jazyky $L_1, L_2 \subseteq \Sigma^*$ jsou regulární, pak také jazyk $L_1 \cap L_2$ je regulární.

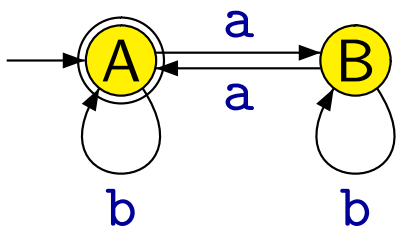
Důkaz: Nechť $L_1 = L(\mathcal{A}_1)$, $L_2 = L(\mathcal{A}_2)$ pro konečné automaty $\mathcal{A}_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1)$, $\mathcal{A}_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$. Definujeme automat $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ tž.

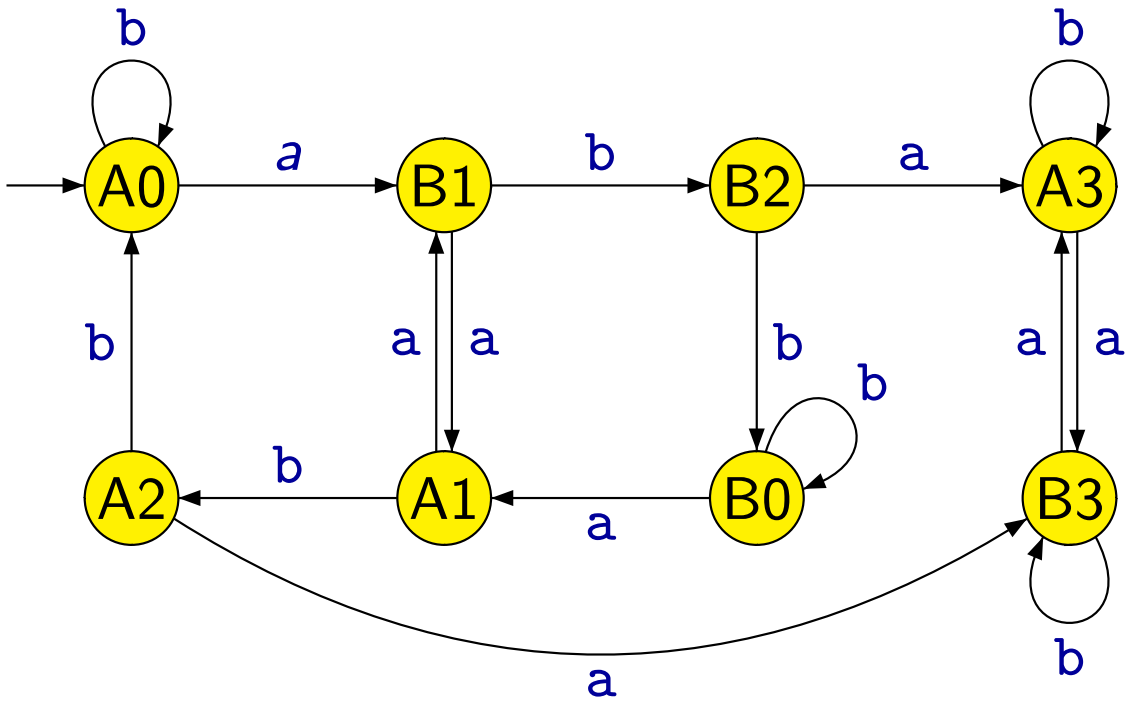
- $Q = Q_1 \times Q_2$,
- $\delta(\,(q_1, q_2), a\,) = (\,\delta_1(q_1, a), \delta_2(q_2, a)\,)$ pro všechna $q_1 \in Q_1$, $q_2 \in Q_2$, $a \in \Sigma$,
- $q_0 = (q_{01}, q_{02})$,
- $F = (F_1 \times F_2)$.

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

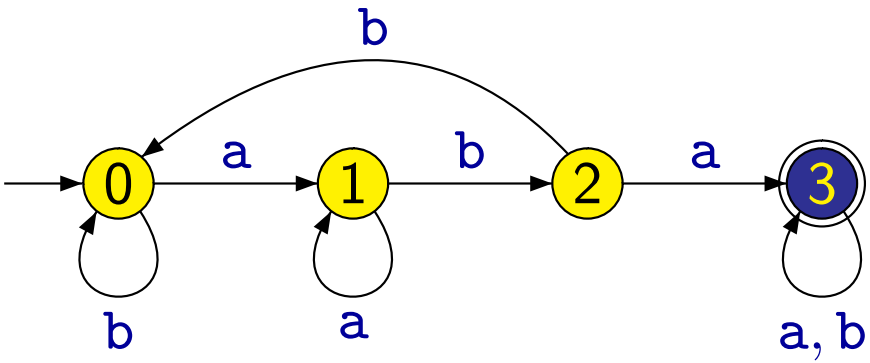
Automat pro sjednocení jazyků

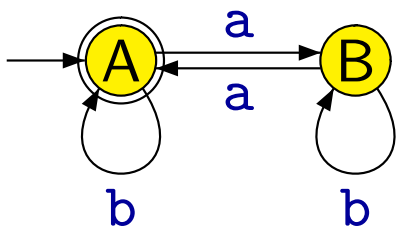


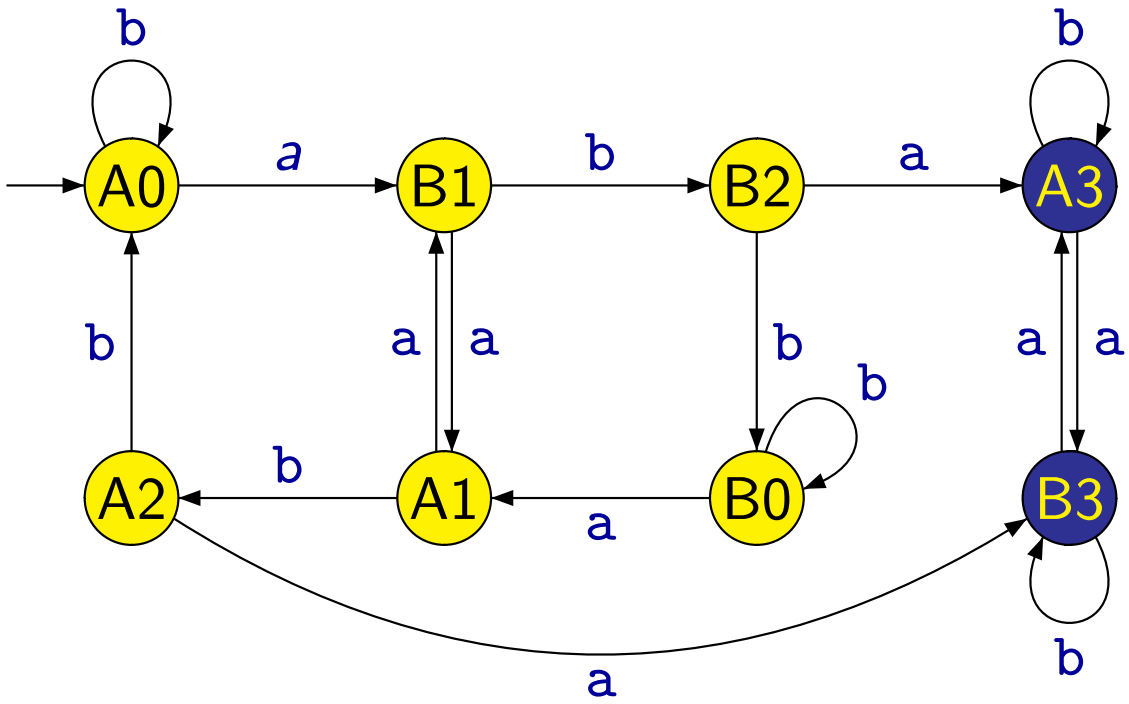




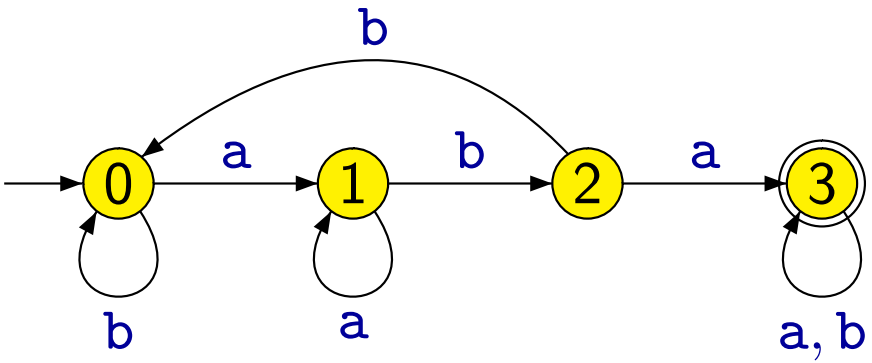
Automat pro sjednocení jazyků

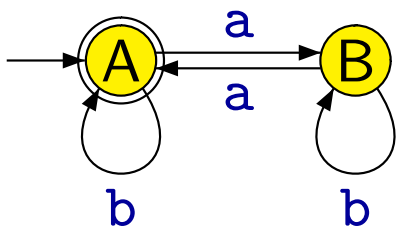


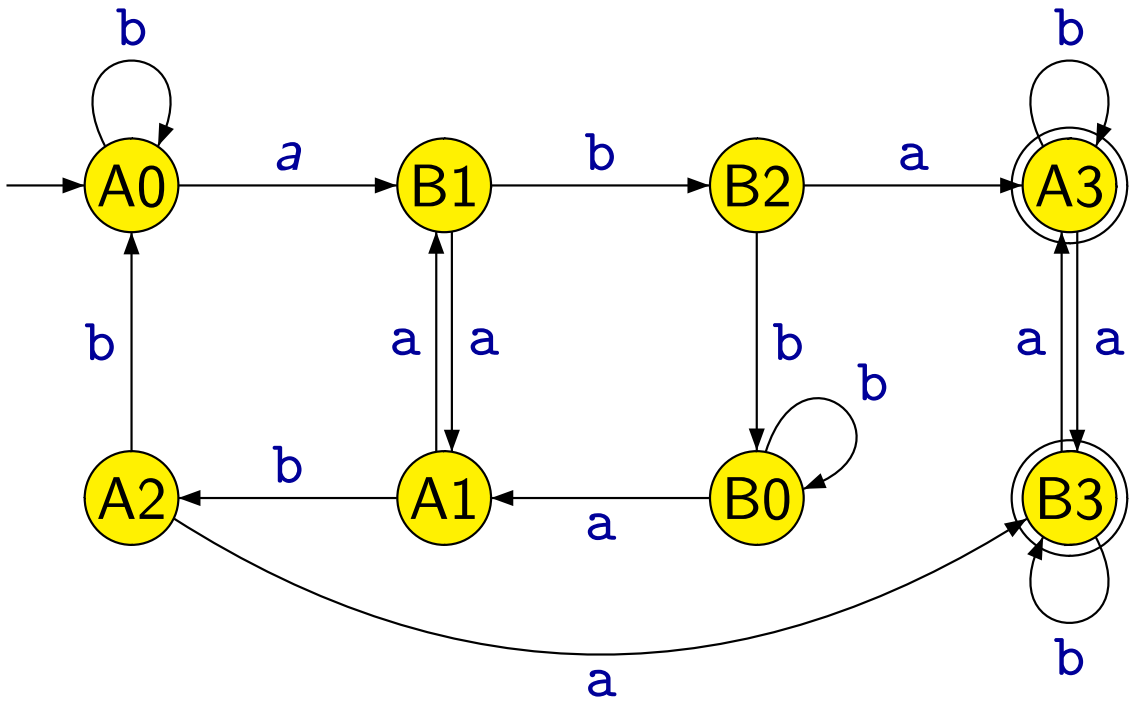




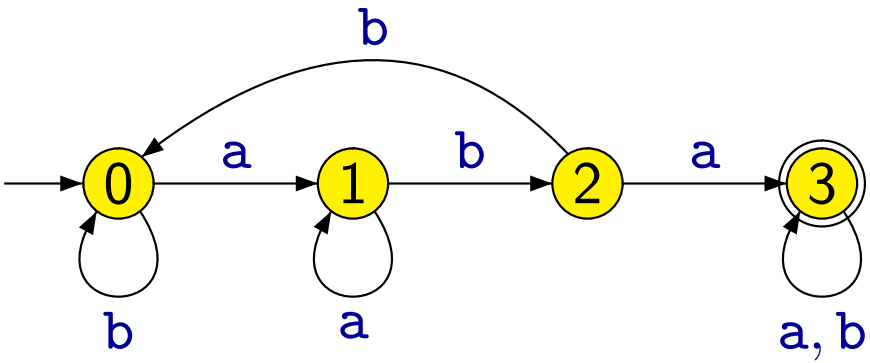
Automat pro sjednocení jazyků

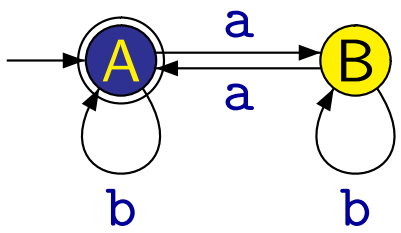


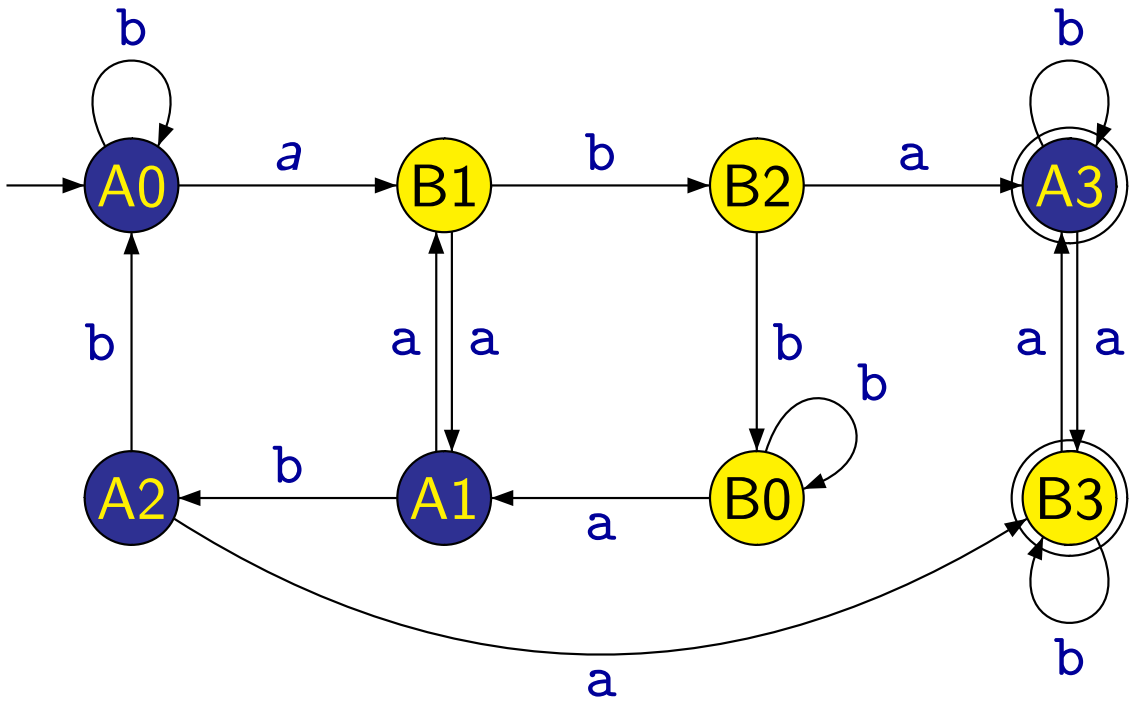




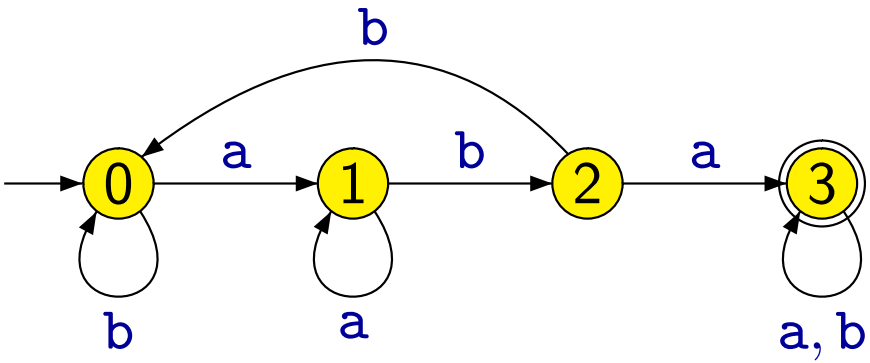
Automat pro sjednocení jazyků

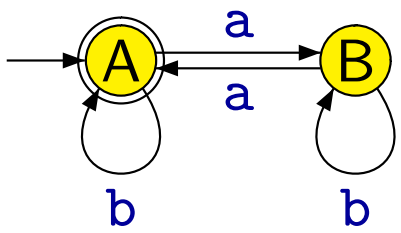


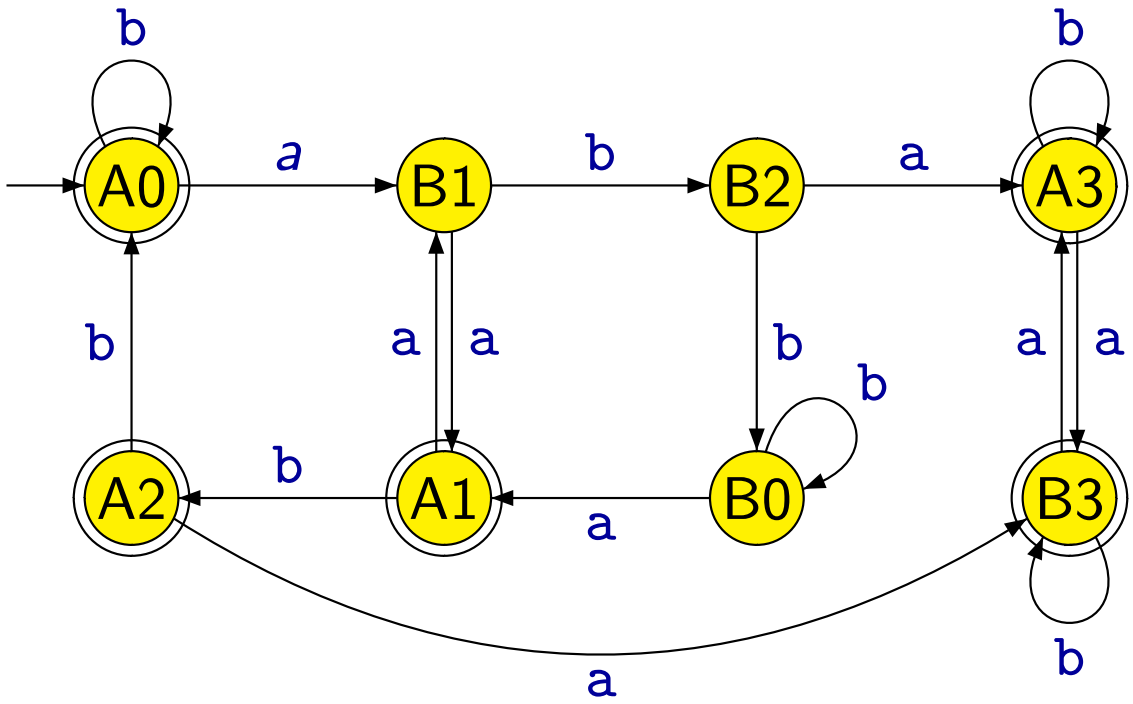




Automat pro sjednocení jazyků







Sjednocení regulárních jazyků

Věta

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

Sjednocení regulárních jazyků

Věta

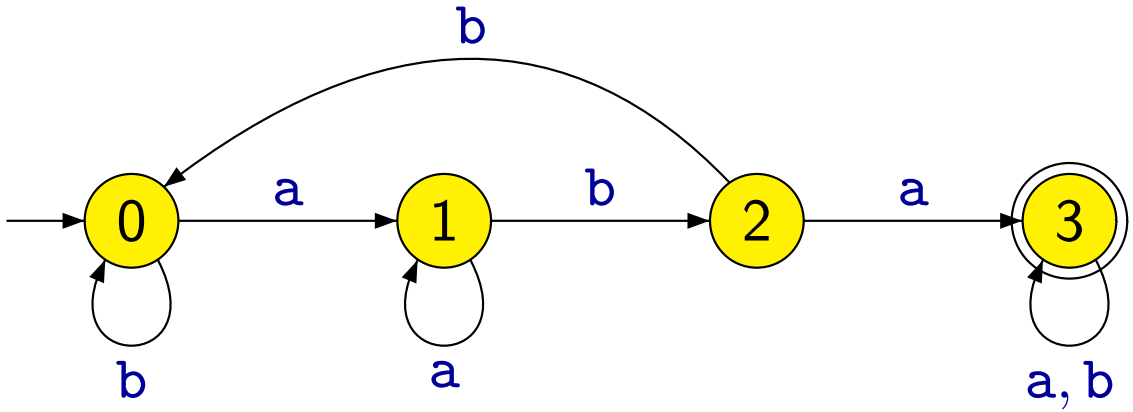
Jestliže jazyky $L_1, L_2 \subseteq \Sigma^*$ jsou regulární, pak také jazyk $L_1 \cup L_2$ je regulární.

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.

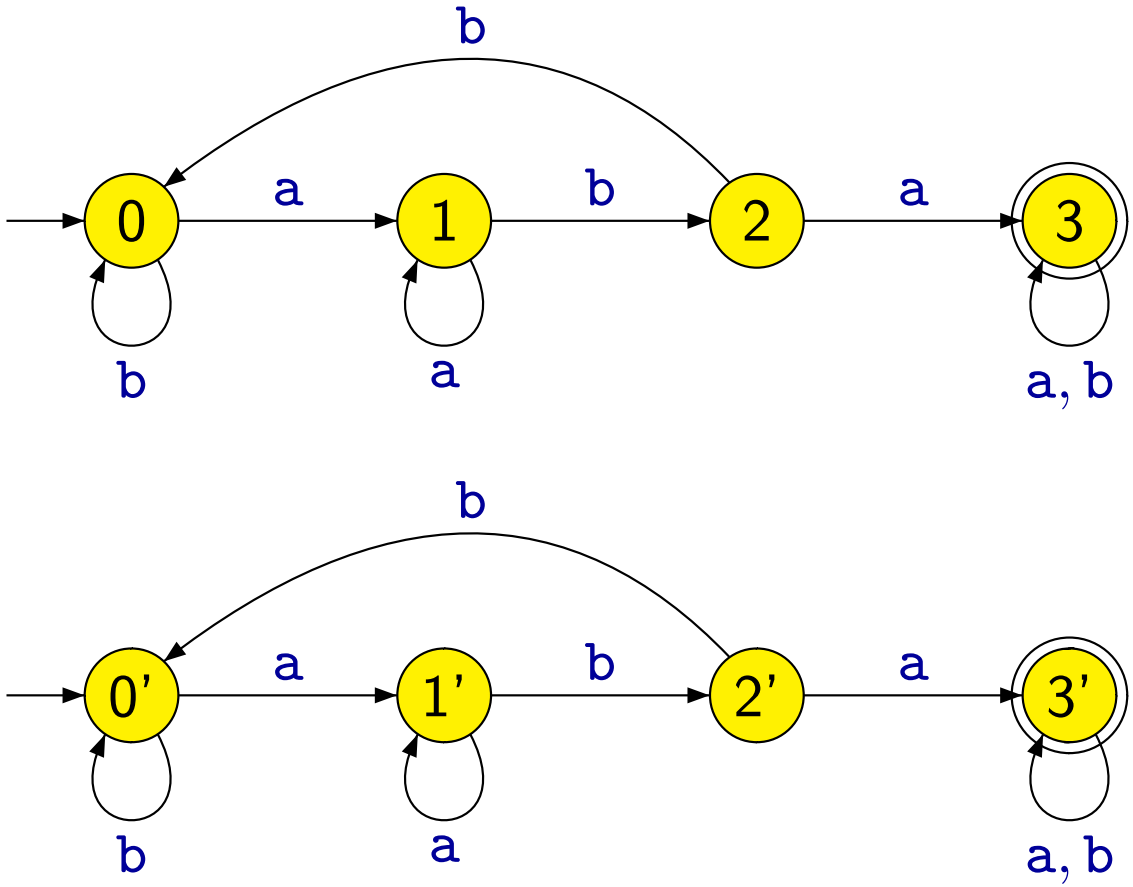
- $Q = Q_1 \times Q_2$,
- $\delta(\,(q_1, q_2), a\,) = (\,\delta_1(q_1, a), \delta_2(q_2, a)\,)$ pro všechna $q_1 \in Q_1$, $q_2 \in Q_2$, $a \in \Sigma$,
- $q_0 = (q_{01}, q_{02})$,
- $F = (F_1 \times Q_2) \cup (Q_1 \times F_2)$.

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 $\delta^*(\,(q_1, q_2), w\,) = (\,\delta_1^*(q_1, w), \delta_2^*(q_2, w)\,)$.

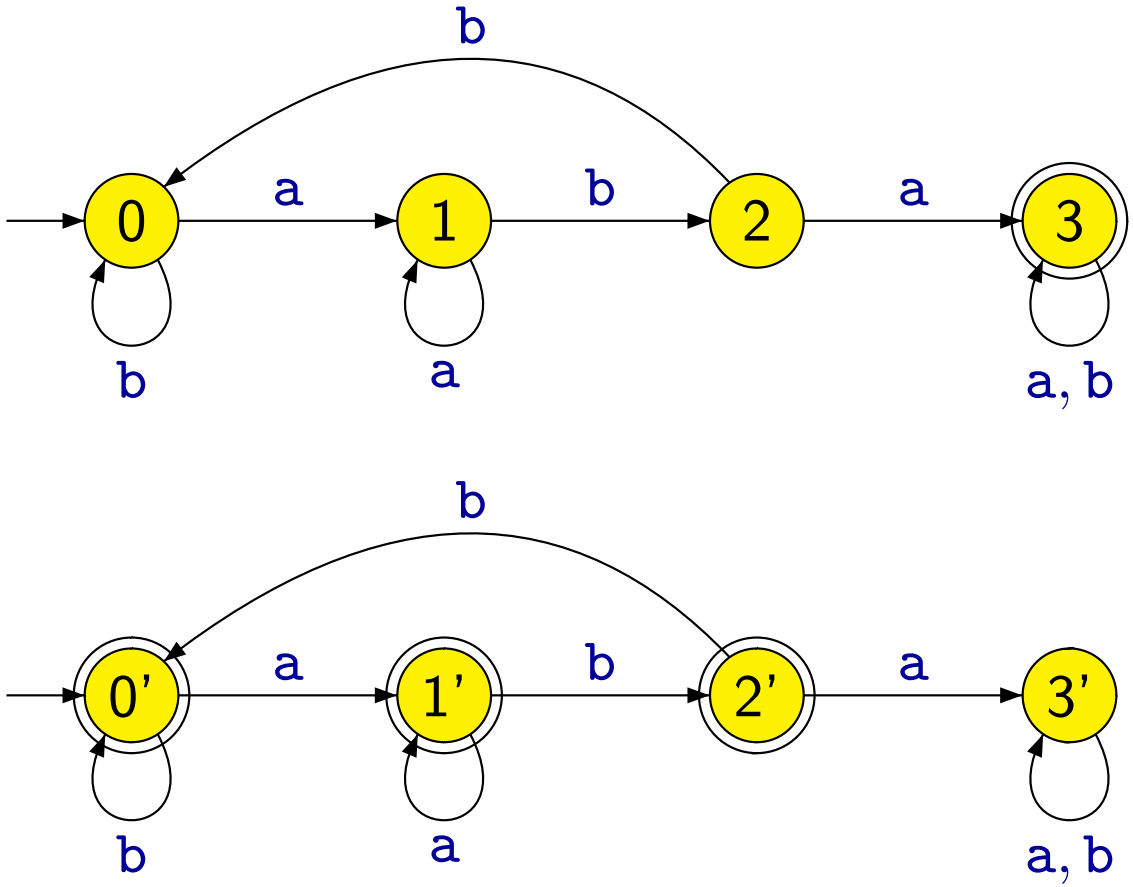
Automat pro doplněk jazyka



Automat pro doplněk jazyka



Automat pro doplněk jazyka



Doplněk regulárního jazyka

Věta

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

Doplněk regulárního jazyka

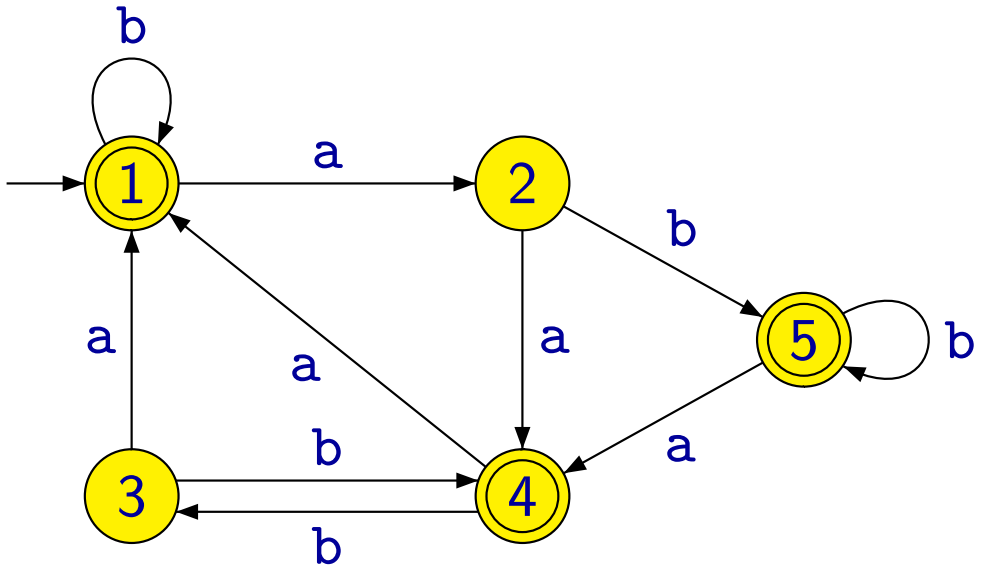
Věta

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

Důkaz: Nechť $L = L(\mathcal{A})$ 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 $\mathcal{A}$

Jazyk rozpoznávaný automatem

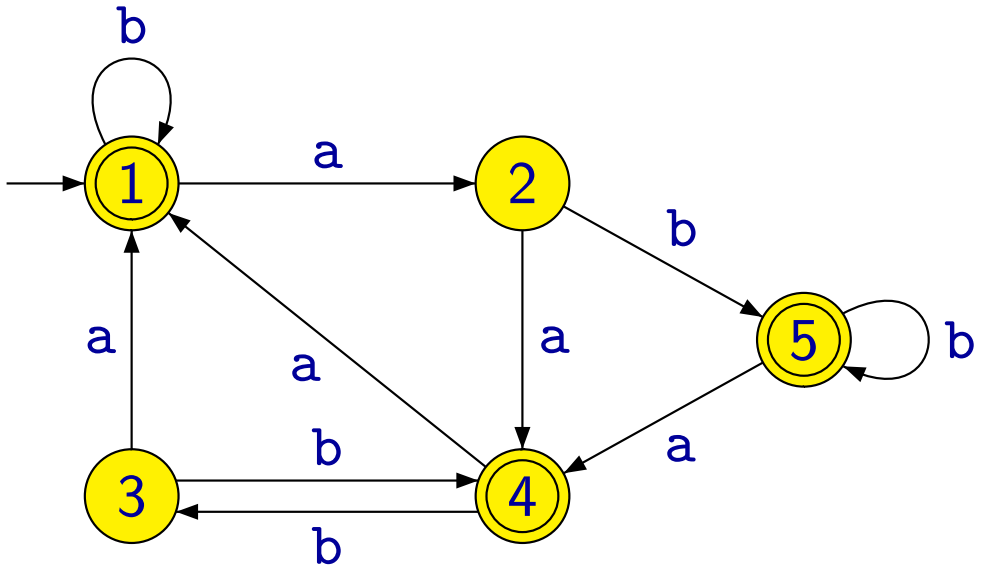


Uvažujme libovolný **tah** v grafu takový, že:

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

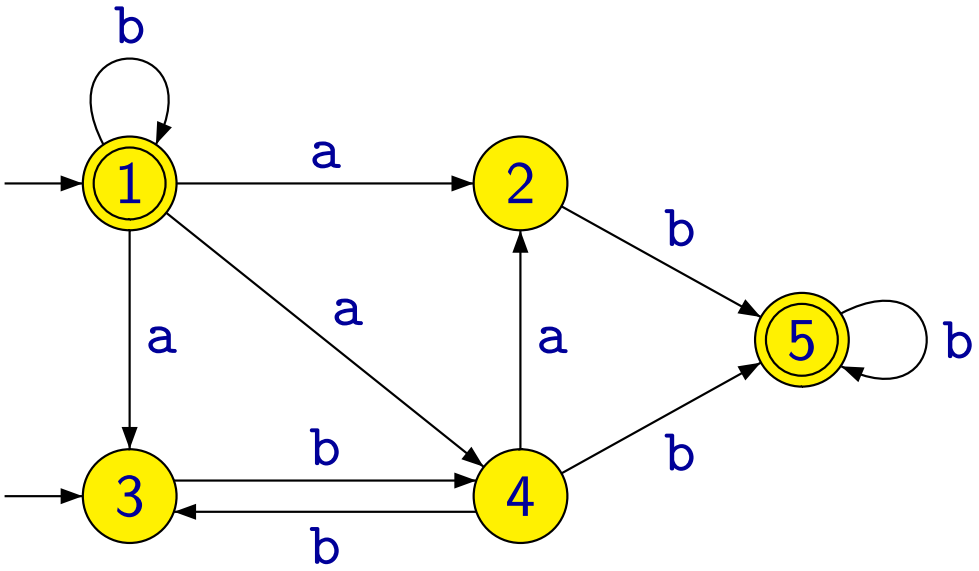
Symboly, jimiž jsou ohodnoceny hrany (tj. přechody) v tomto tahu, tvoří **slovo**, které je přijímáno daným automatem.

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

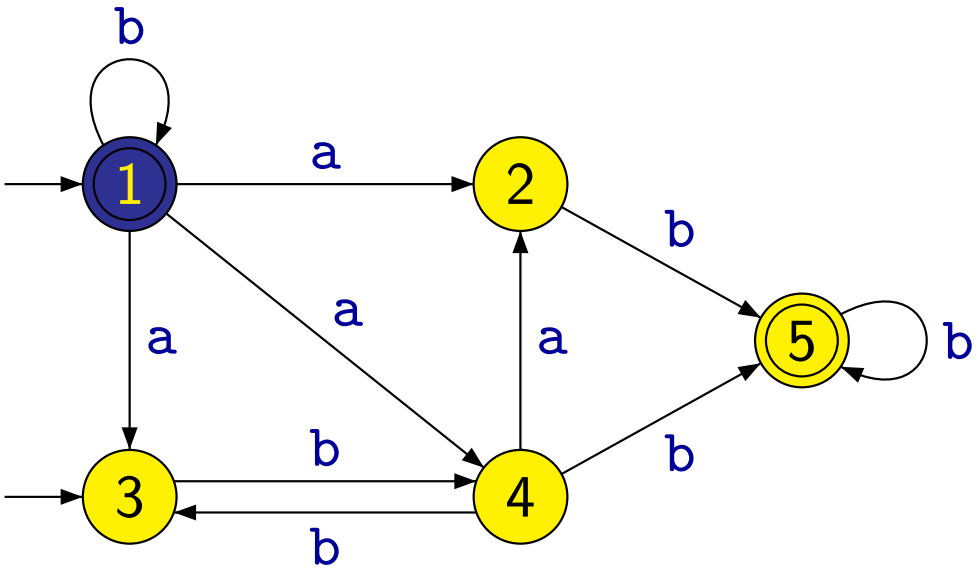
Takto obecněji definovaný automat se nazývá **nedeterministický konečný automat**.

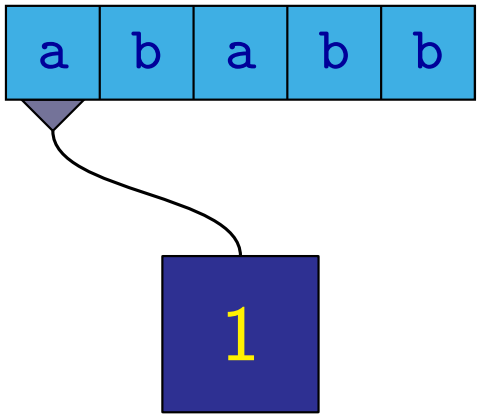
Rozdíly oproti deterministickým konečným automatům:

- 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).

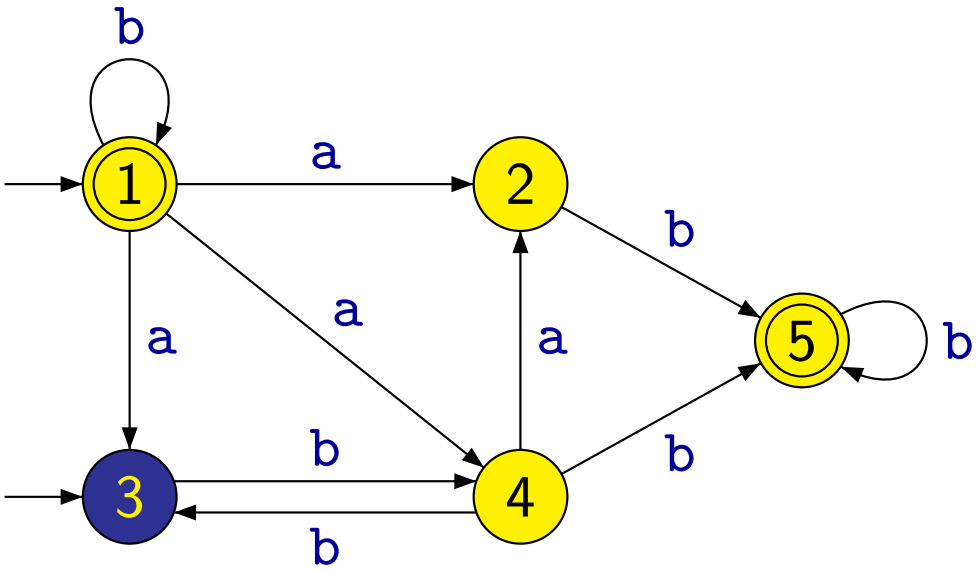
Nedeterministický konečný automat

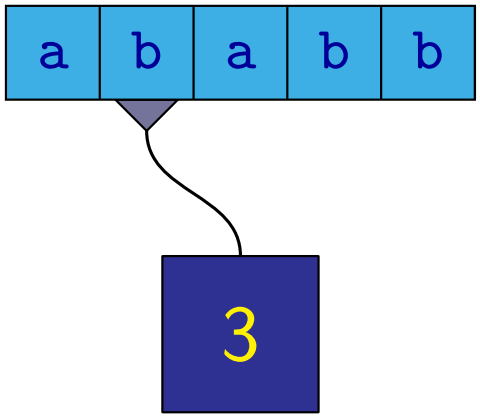




$(1, \mathtt{ababb})$

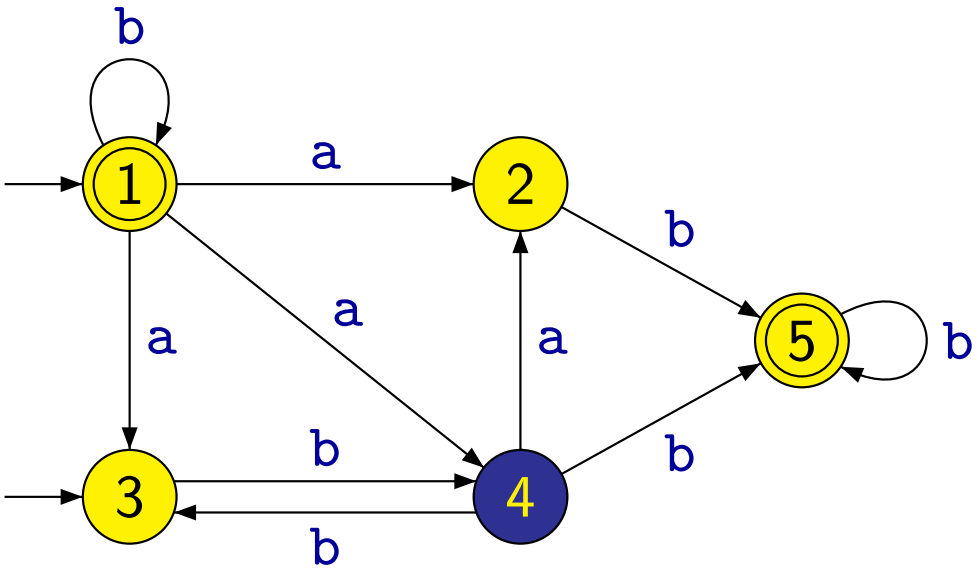
Nedeterministický konečný automat

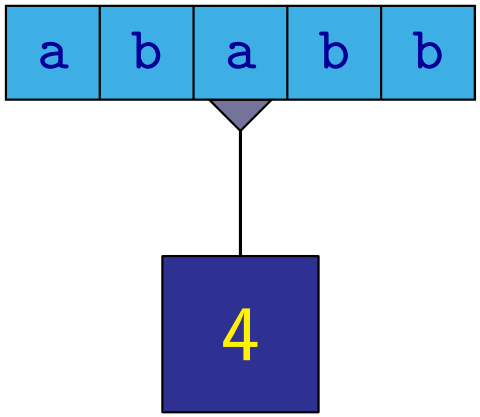




$(1, \mathtt{ababb})$
$\vdash (3, \mathtt{babb})$

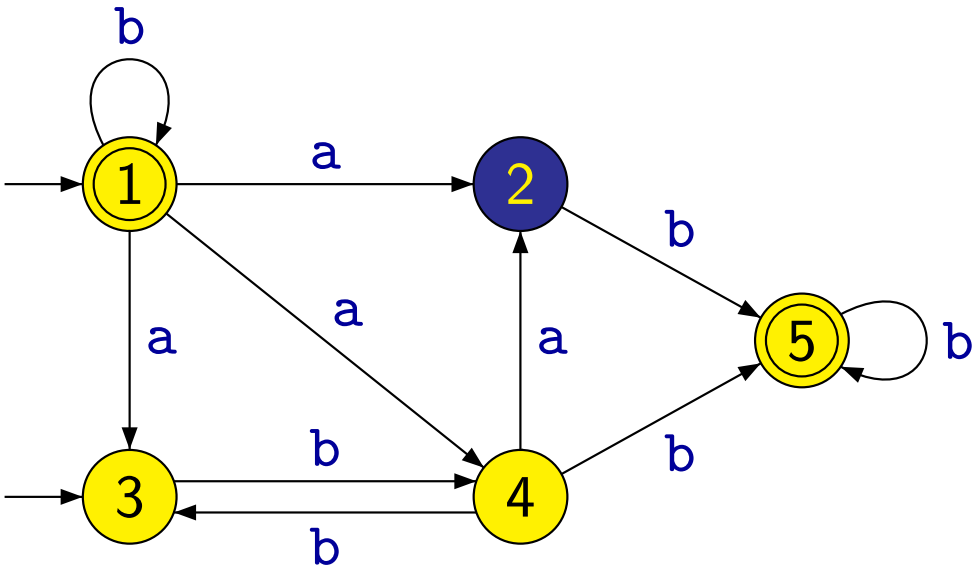
Nedeterministický konečný automat

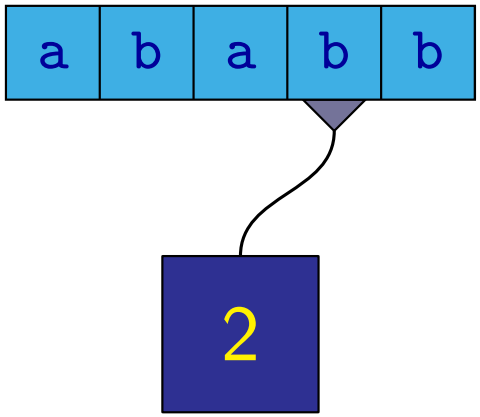




$(1, \mathtt{ababb})$
$\vdash (3, \mathtt{babb})$
$\vdash (4, \mathtt{abb})$

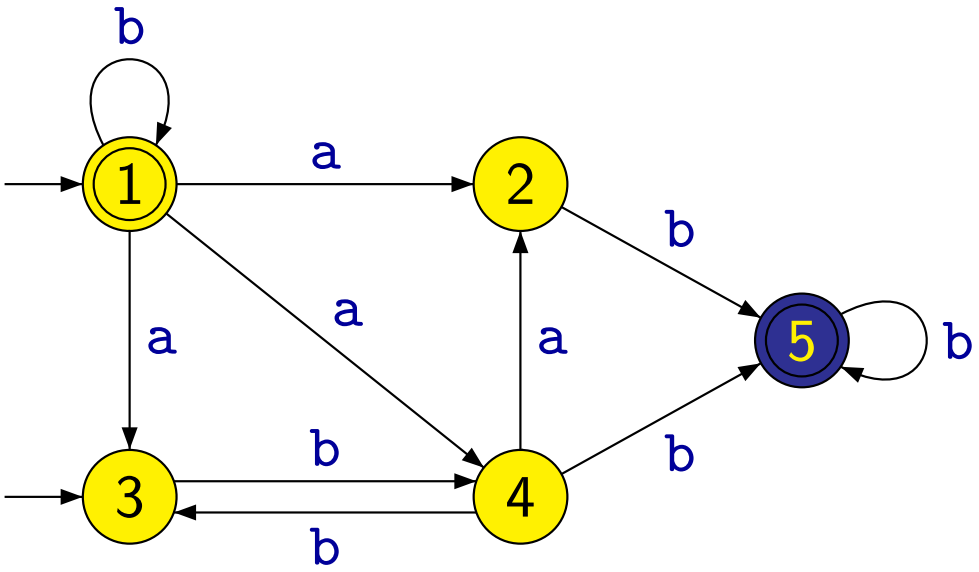
Nedeterministický konečný automat

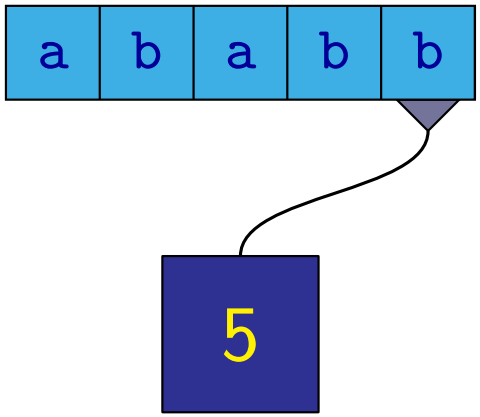




$(1, \mathtt{ababb})$
$\vdash (3, \mathtt{babb})$
$\vdash (4, \mathtt{abb})$
$\vdash (2, \mathtt{bb})$

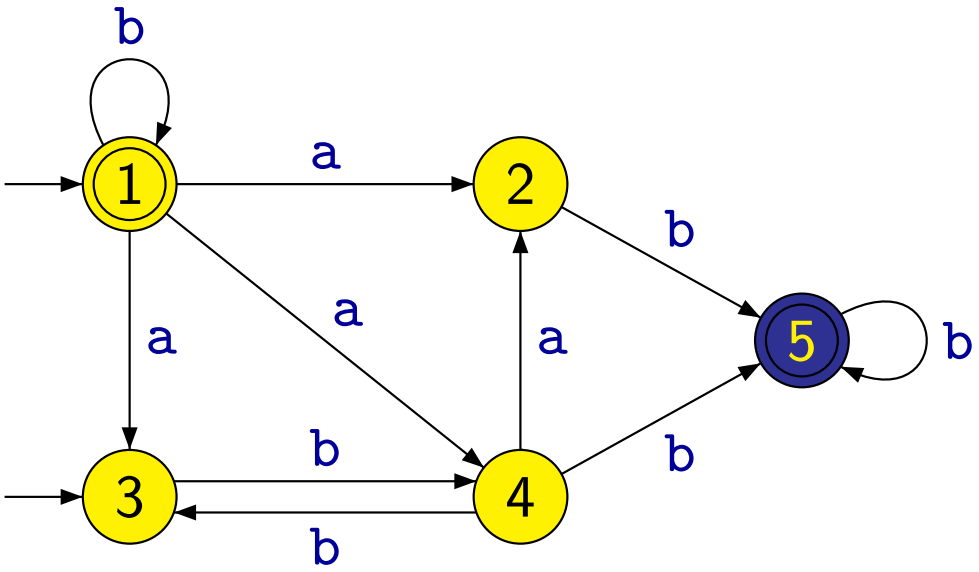
Nedeterministický konečný automat

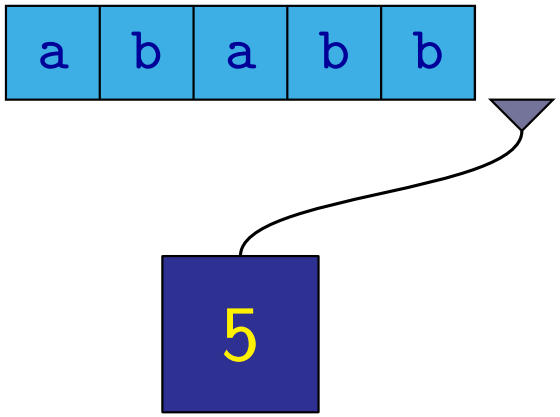




$(1, \mathtt{ababb})$
$\vdash (3, \mathtt{babb})$
$\vdash (4, \mathtt{abb})$
$\vdash (2, \mathtt{bb})$
$\vdash (5, \mathtt{b})$

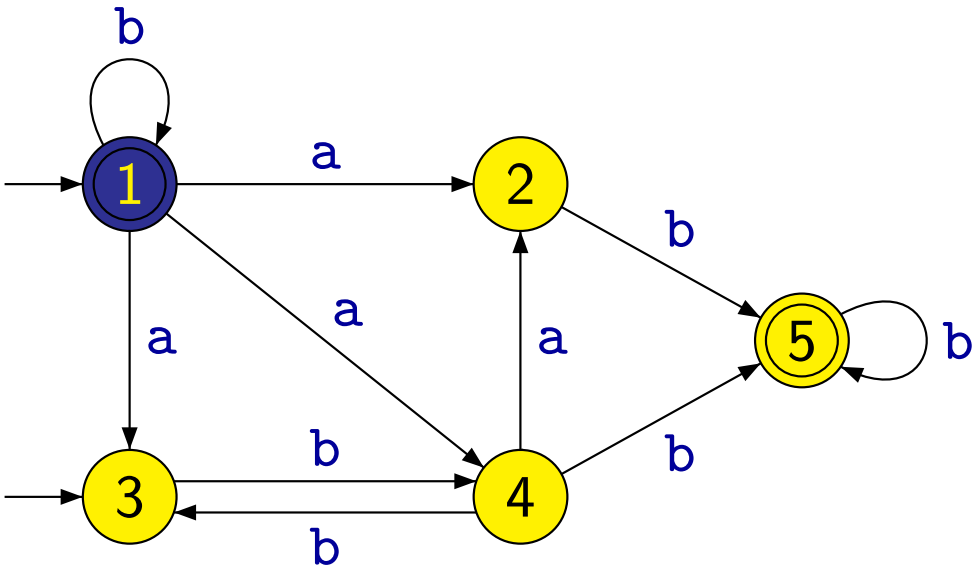
Nedeterministický konečný automat

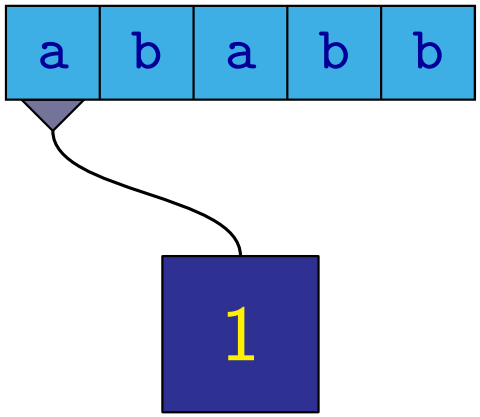




$$(1, \mathtt{ababb}) \vdash (3, \mathtt{babb}) \vdash (4, \mathtt{abb}) \vdash (2, \mathtt{bb}) \vdash (5, \mathtt{b}) \vdash (5, \varepsilon)$$

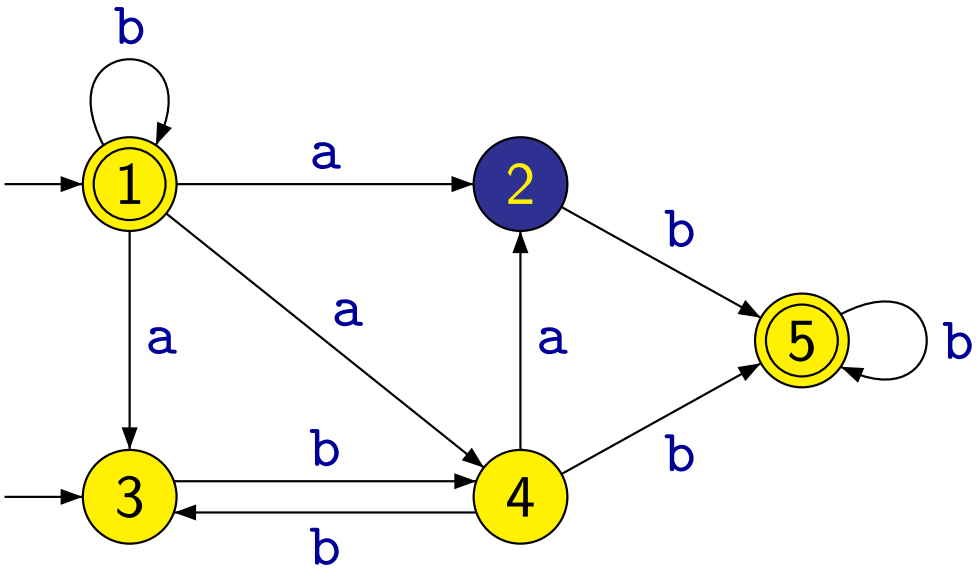
Nedeterministický konečný automat

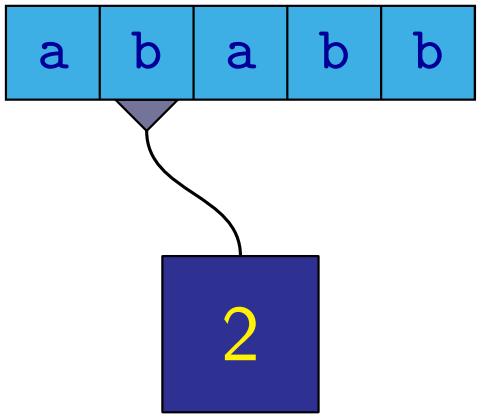




$(1, ababb)$

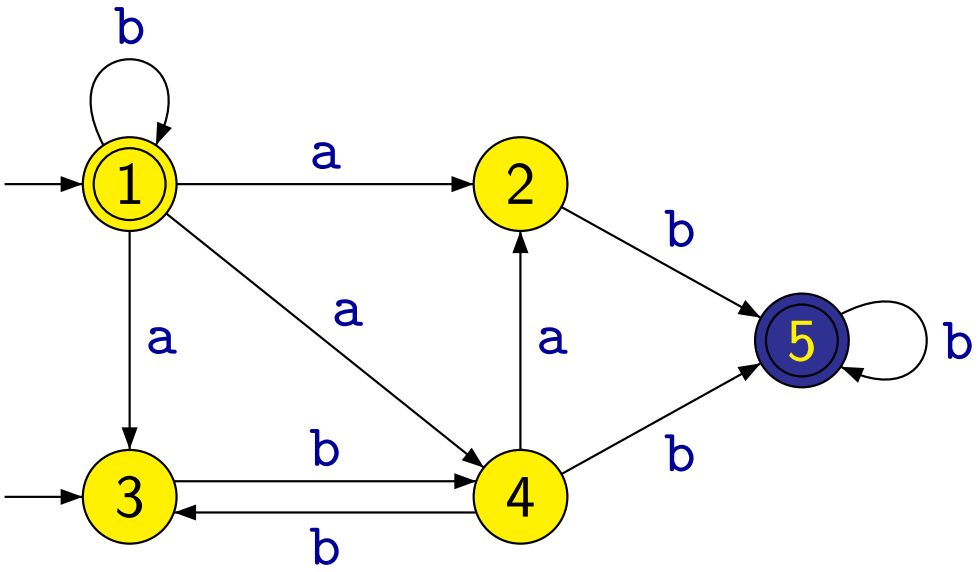
Nedeterministický konečný automat

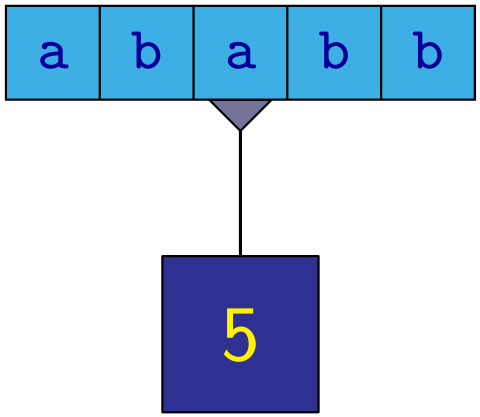




$(1, \mathtt{ababb})$
$\vdash (2, \mathtt{babb})$

Nedeterministický konečný automat

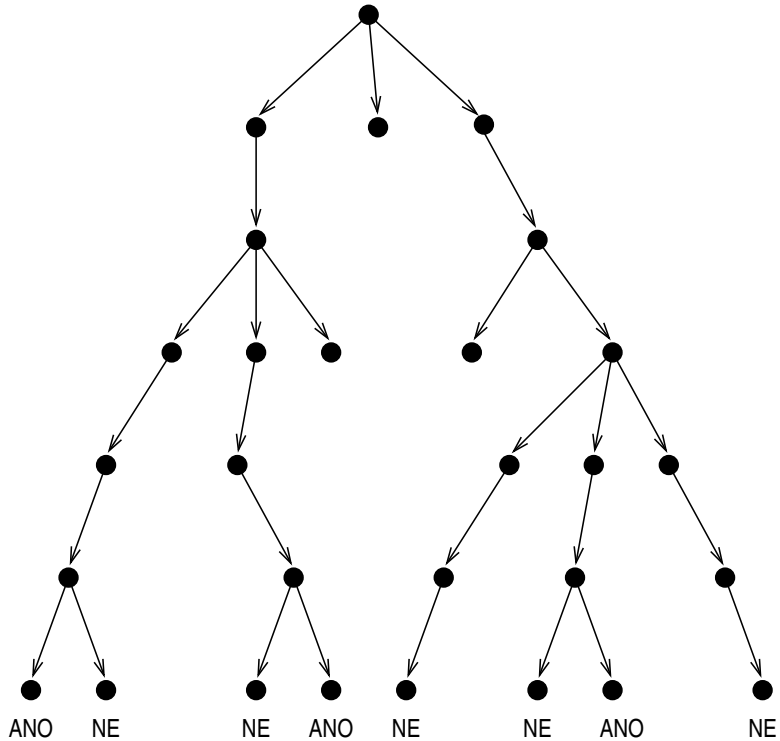




$$(1, \mathtt{ababb}) \vdash (2, \mathtt{babb}) \vdash (5, \mathtt{abb})$$

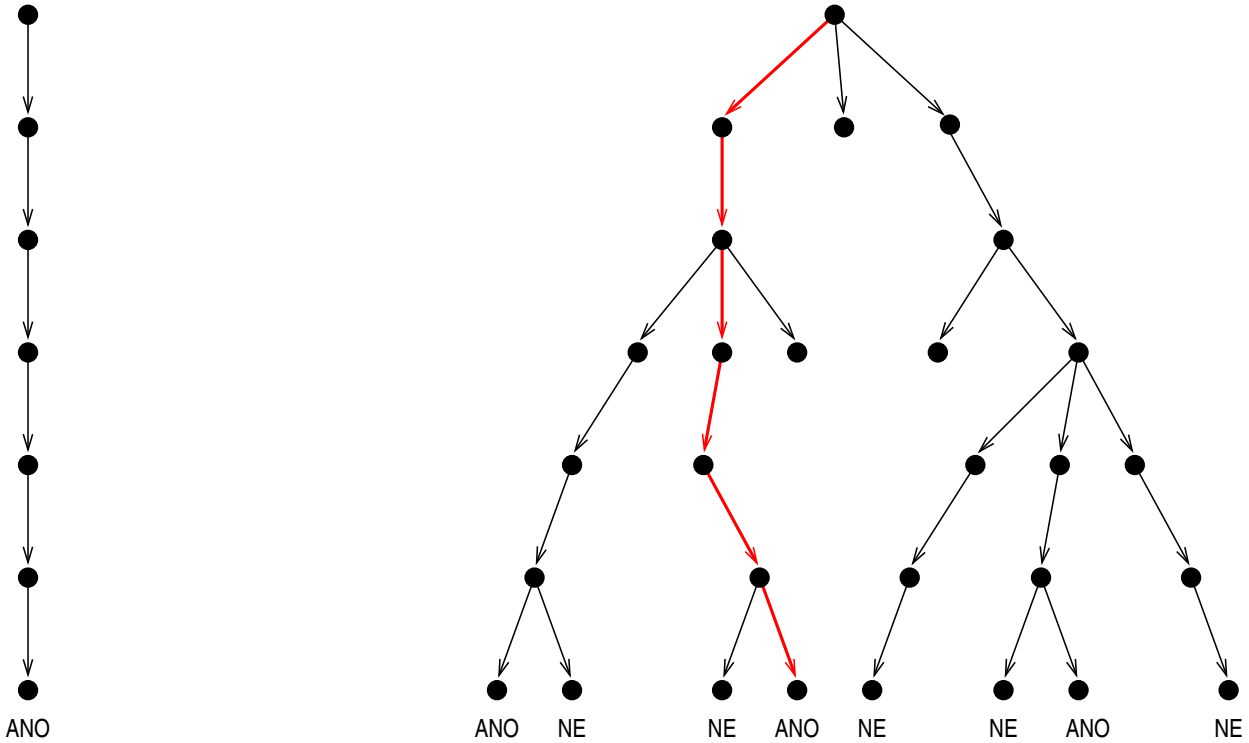
Nedeterministický konečný automat

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



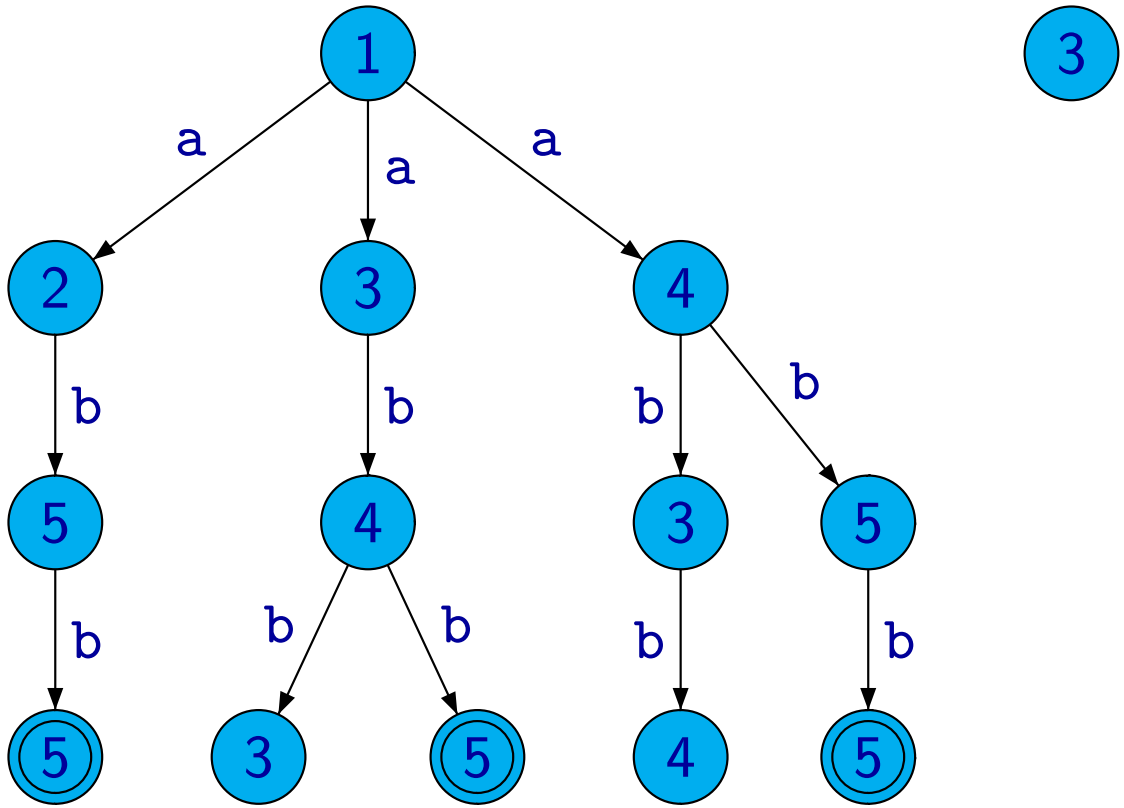
Nedeterministický konečný automat

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



Nedeterministický konečný automat

	a	b
$\leftrightarrow 1$	$2, 3, 4$	1
2	$-$	5
$\rightarrow 3$	$-$	4
4	2	$3, 5$
$\leftarrow 5$	$-$	5



Příklad: Les reprezentující všechny možné výpočty nad slovem abb.

Nedeterministický konečný automat

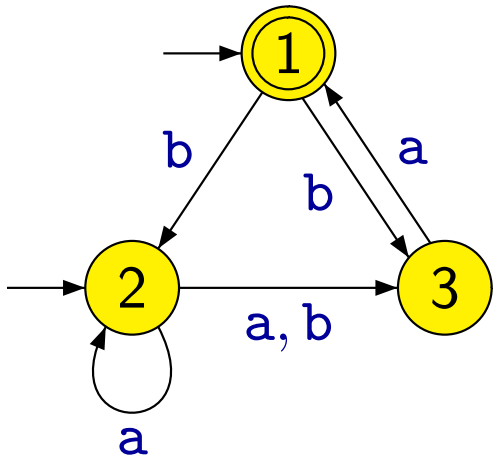
Formálně je **nedeterministický konečný automat** definován jako pětice

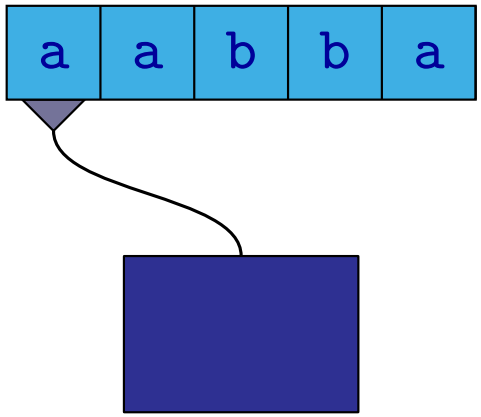
$$(Q, \Sigma, \delta, I, F)$$

kde:

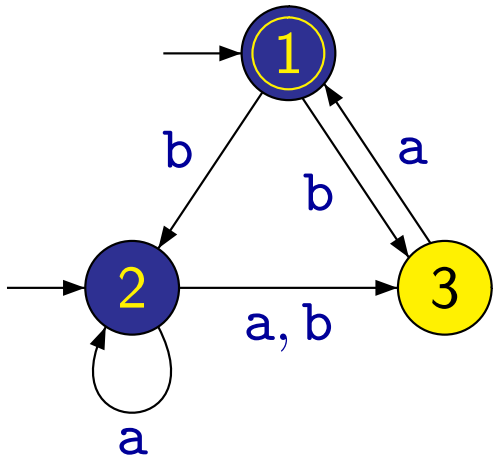
- Q je konečná množina **stavů**
- Σ je konečná **abeceda**
- $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ je **přechodová funkce**
- $I \subseteq Q$ je množina **počátečních stavů**
- $F \subseteq Q$ je množina **přijímajících stavů**

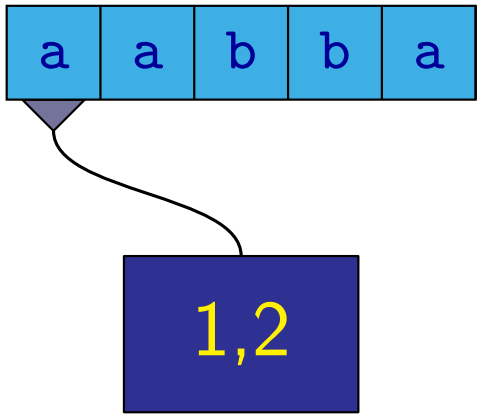
Převod nedeterministického automatu na deterministický



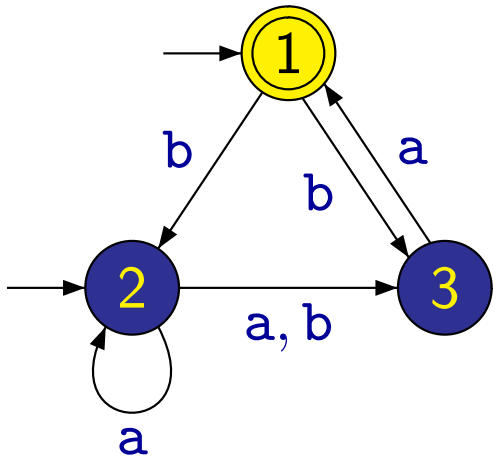


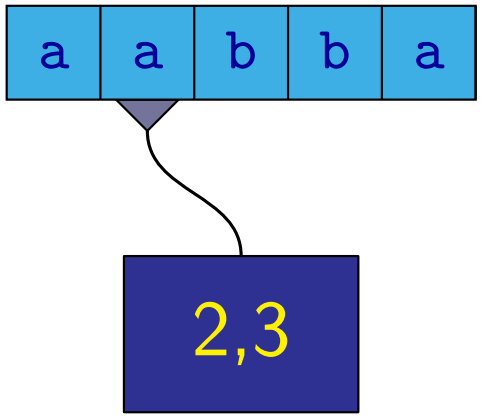
Převod nedeterministického automatu na deterministický



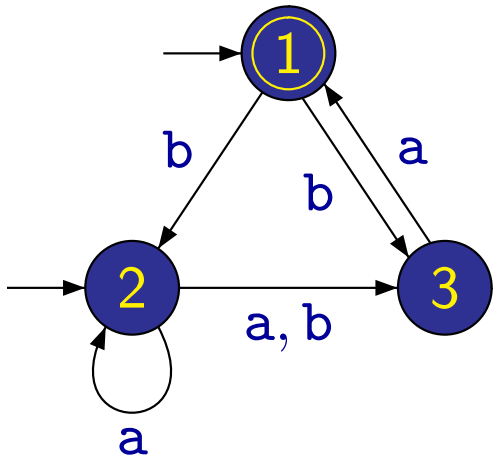


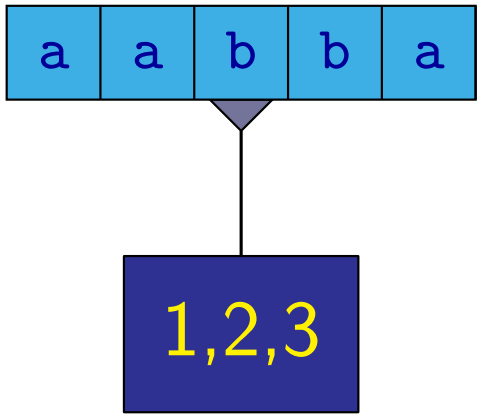
Převod nedeterministického automatu na deterministický



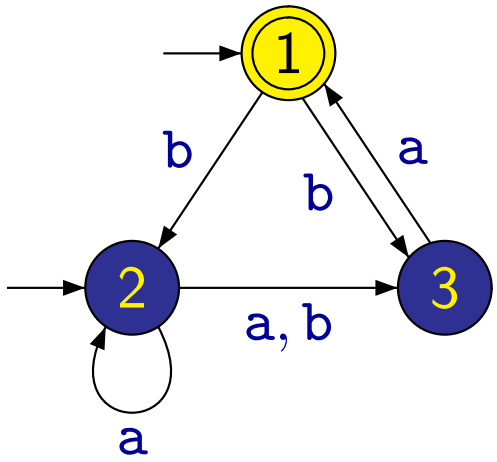


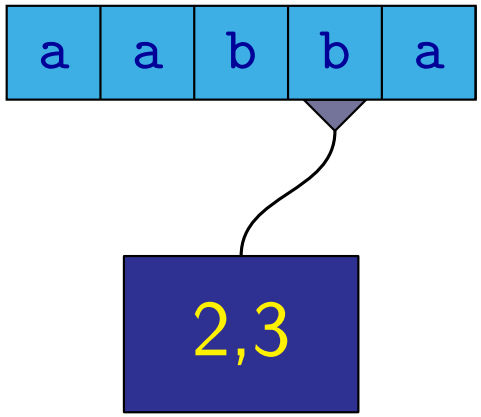
Převod nedeterministického automatu na deterministický



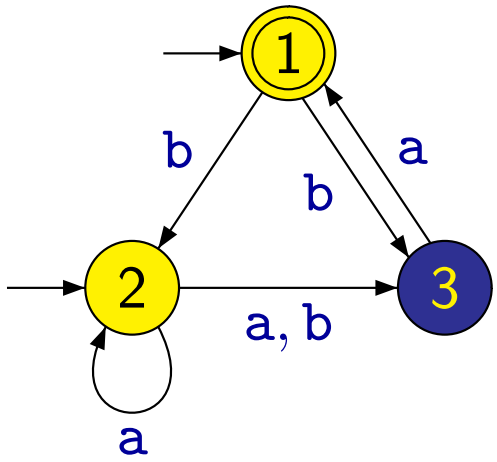


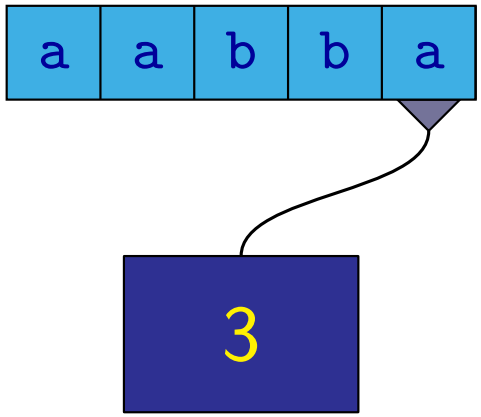
Převod nedeterministického automatu na deterministický



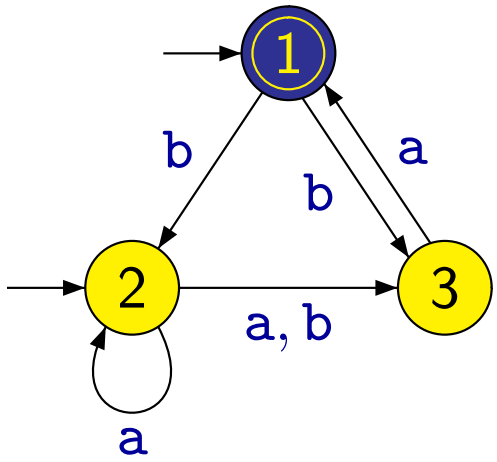


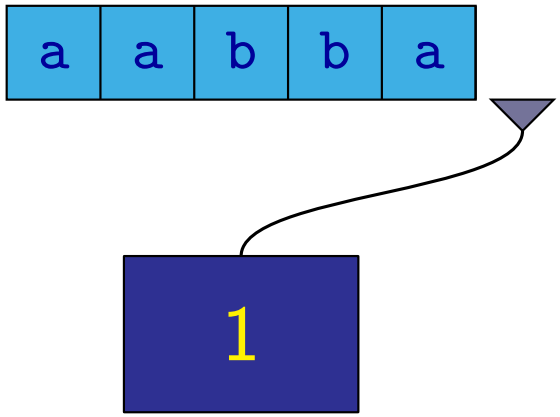
Převod nedeterministického automatu na deterministický



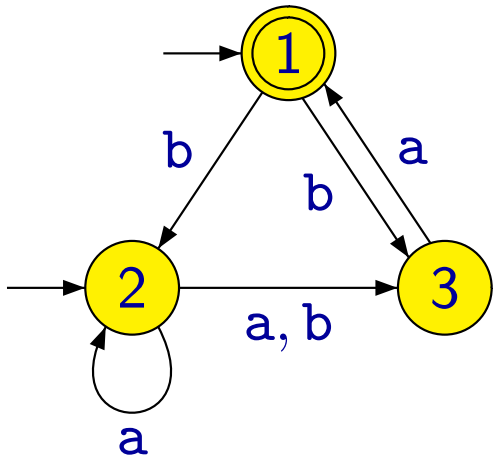


Převod nedeterministického automatu na deterministický

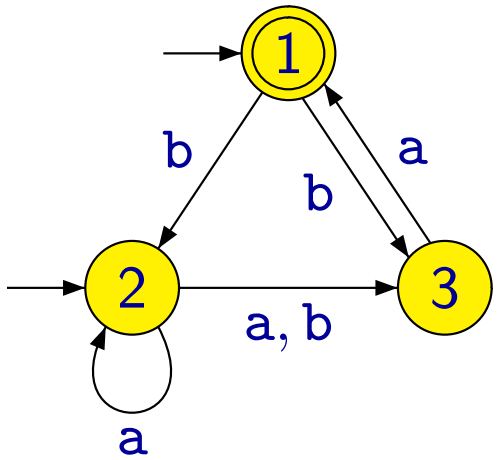




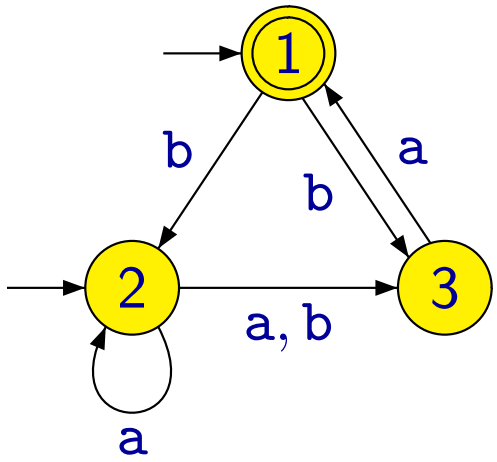
Převod nedeterministického automatu na deterministický

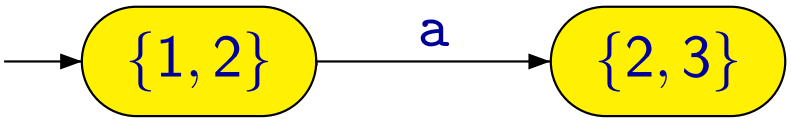


Převod nedeterministického automatu na deterministický

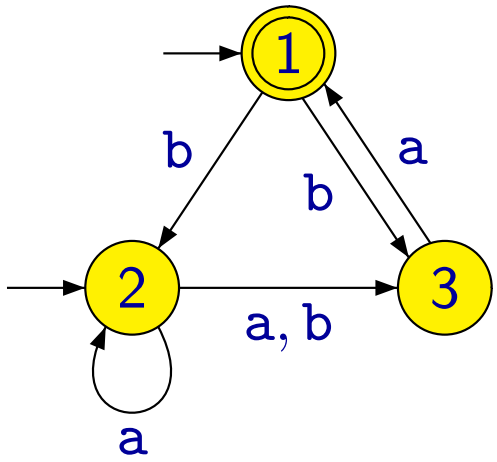


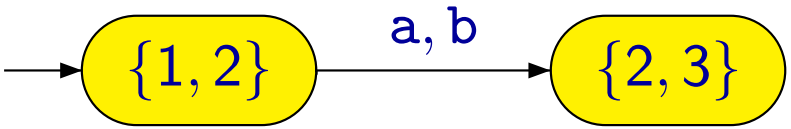
Převod nedeterministického automatu na deterministický



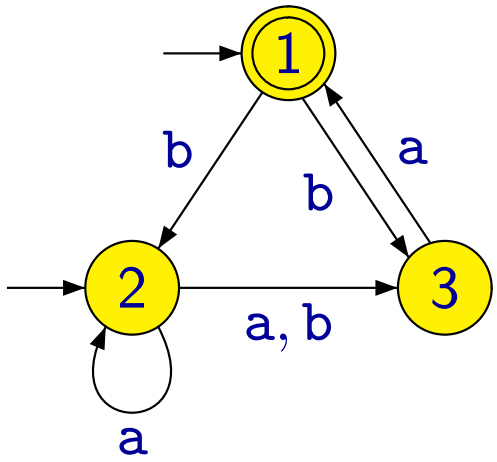


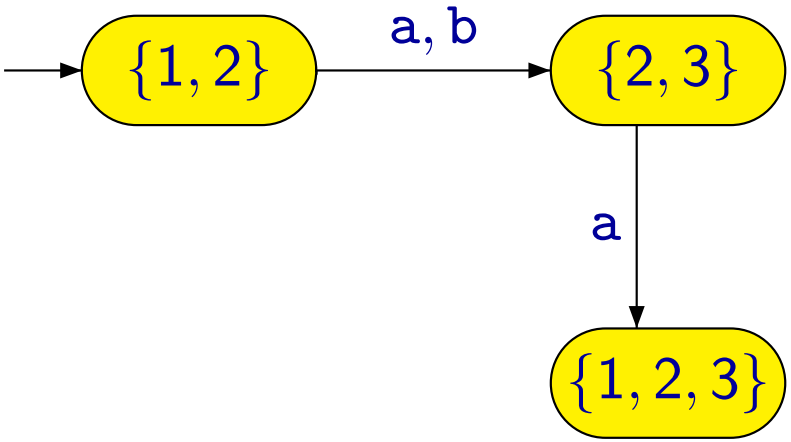
Převod nedeterministického automatu na deterministický



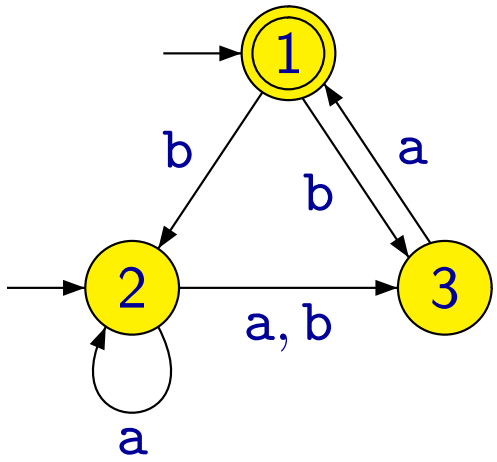


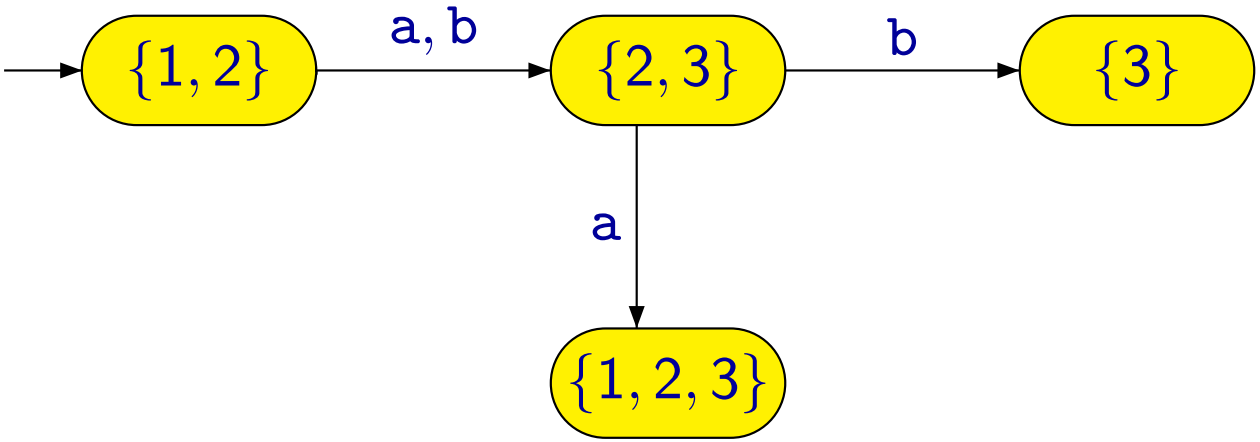
Převod nedeterministického automatu na deterministický



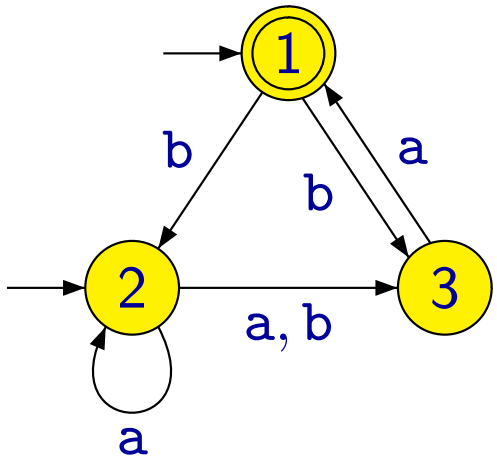


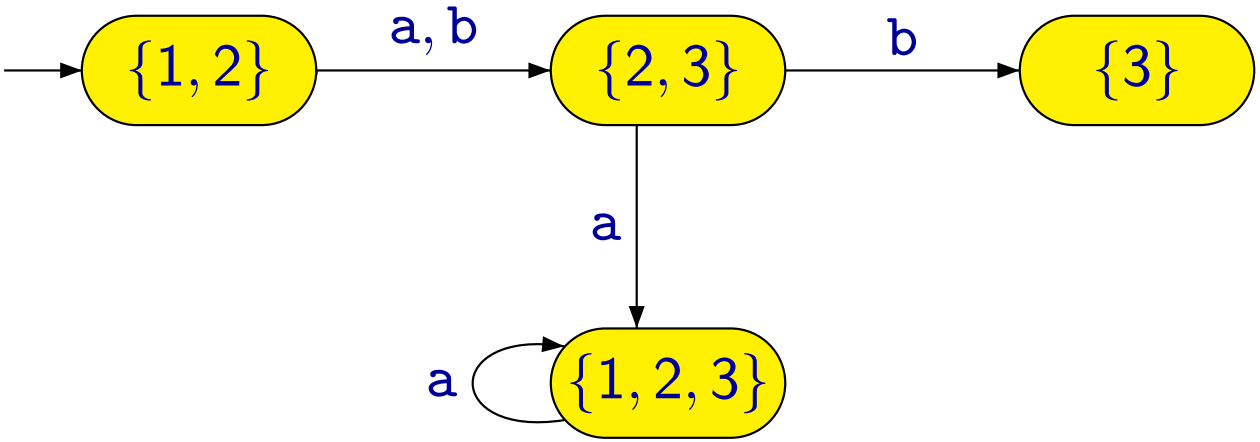
Převod nedeterministického automatu na deterministický



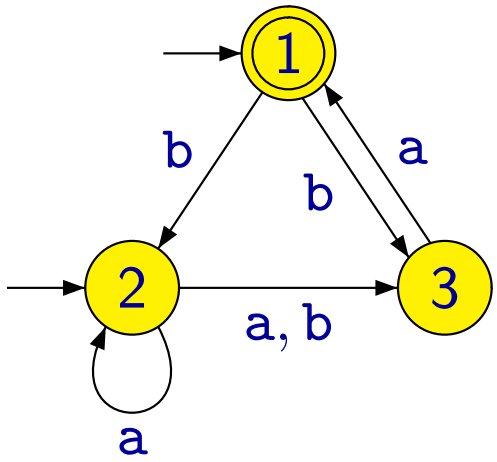


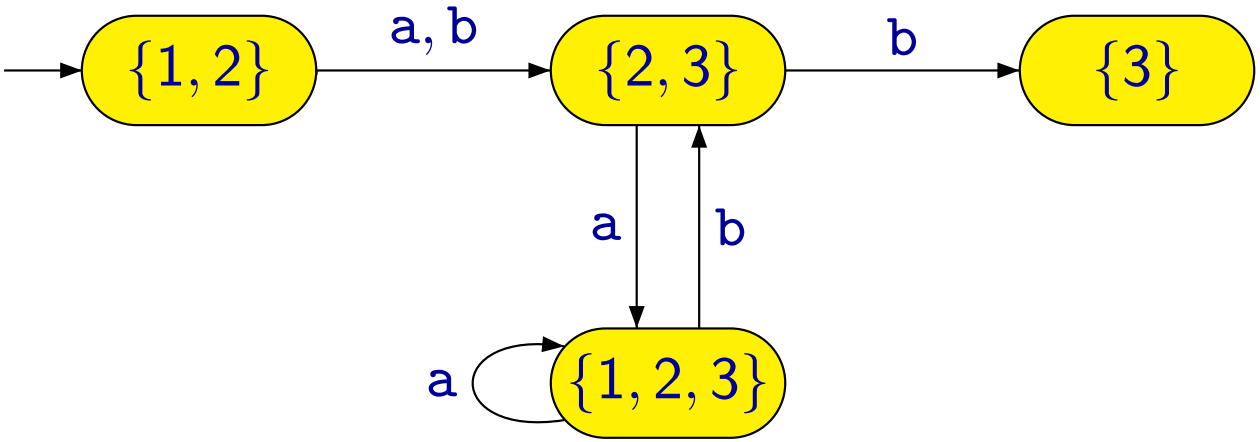
Převod nedeterministického automatu na deterministický



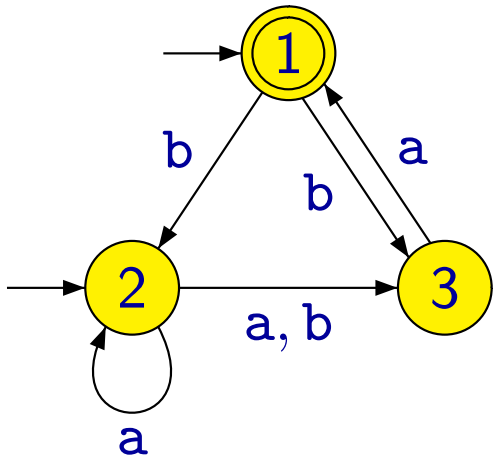


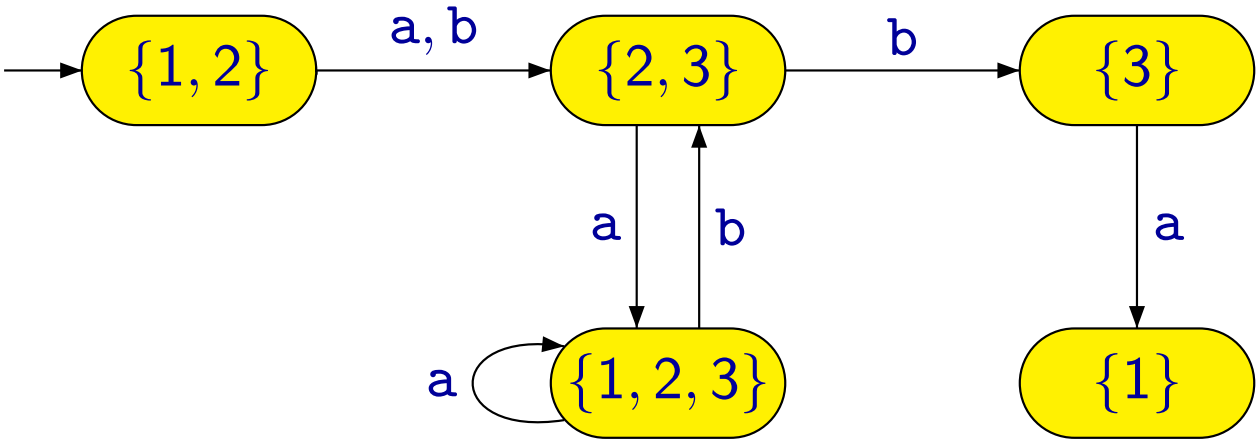
Převod nedeterministického automatu na deterministický



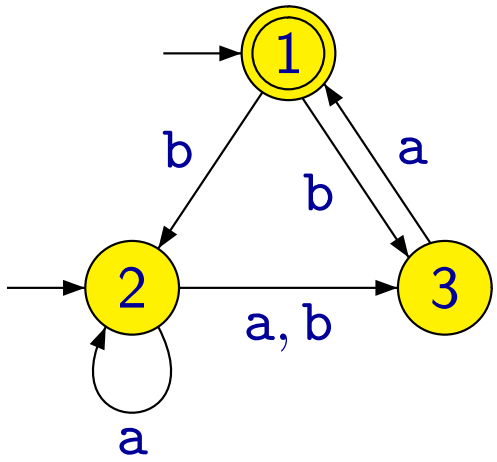


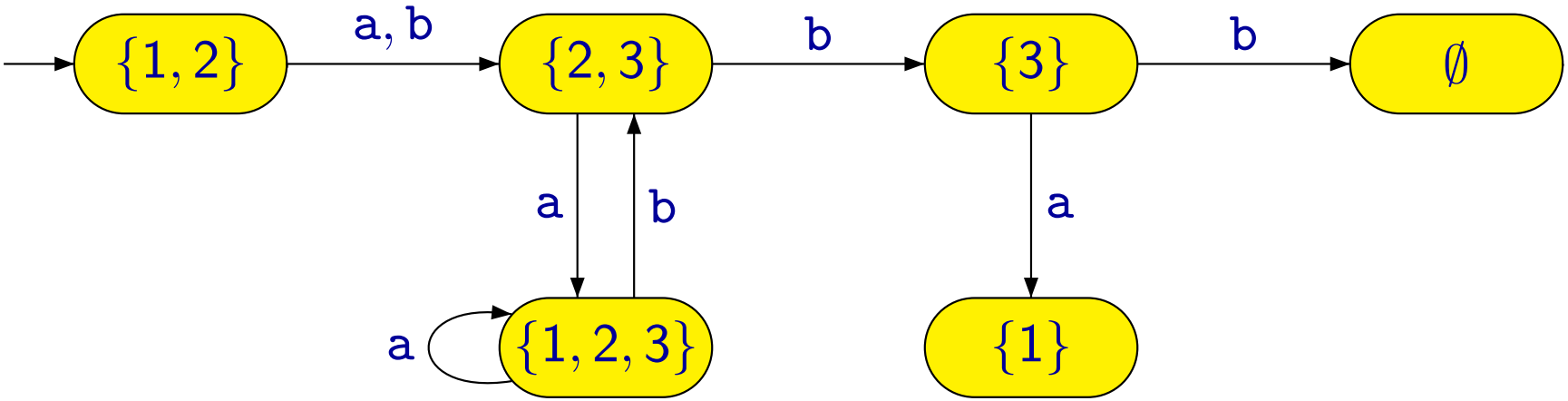
Převod nedeterministického automatu na deterministický



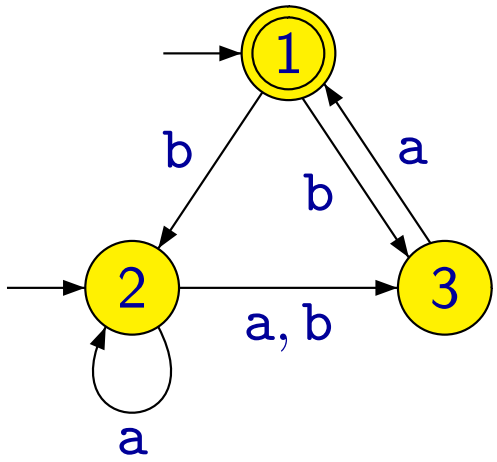


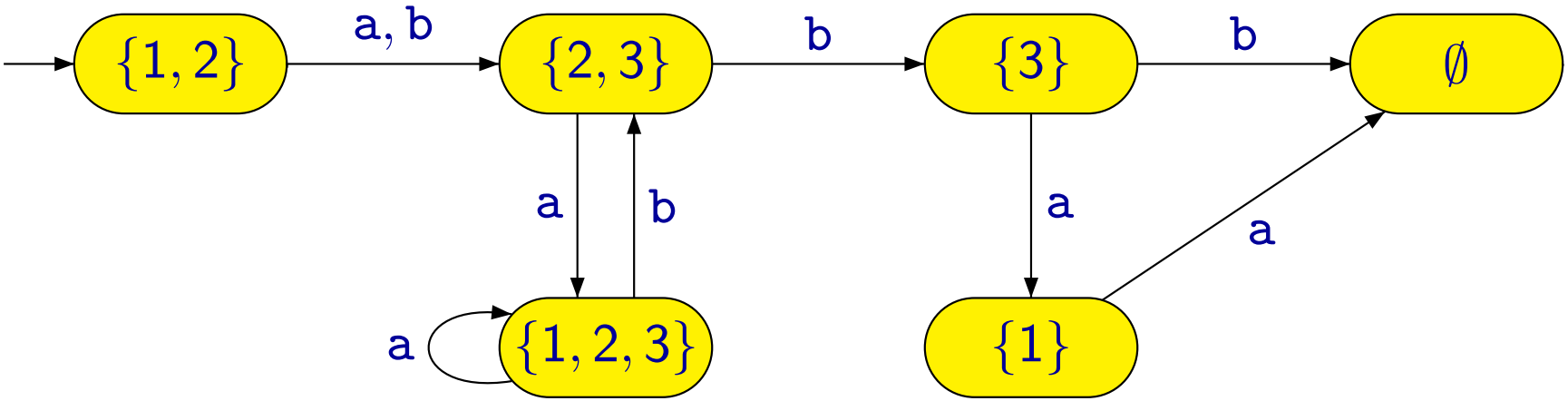
Převod nedeterministického automatu na deterministický



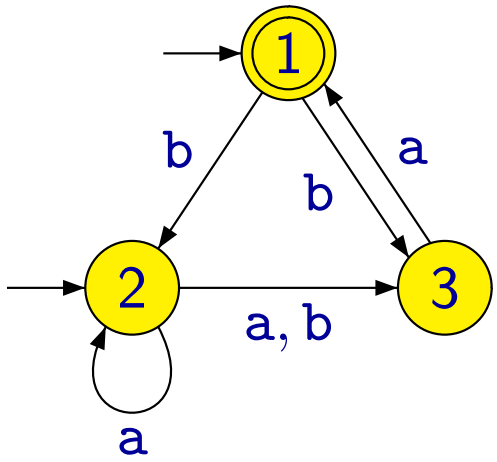


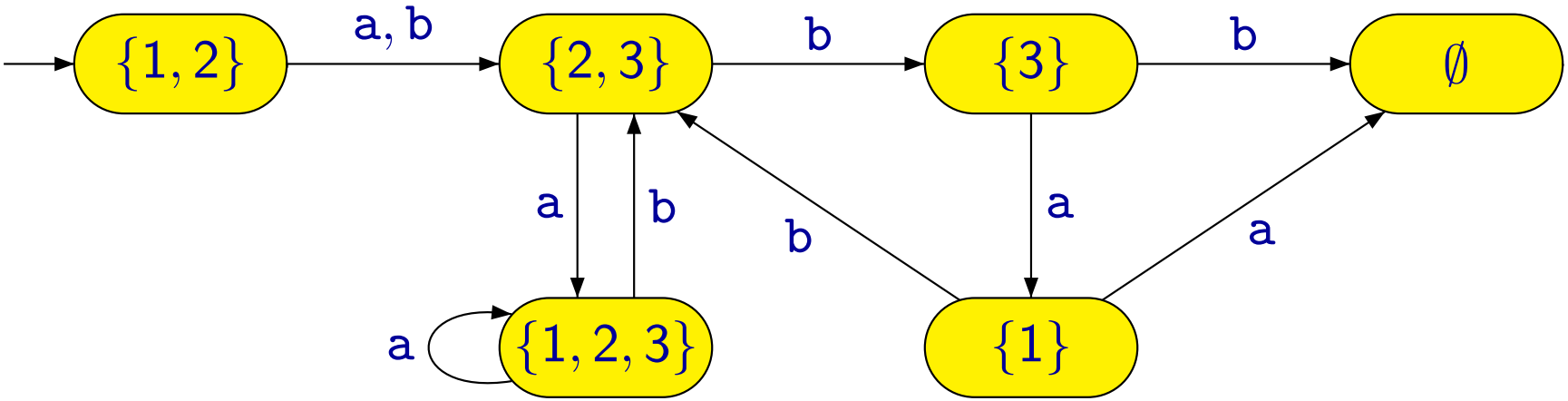
Převod nedeterministického automatu na deterministický



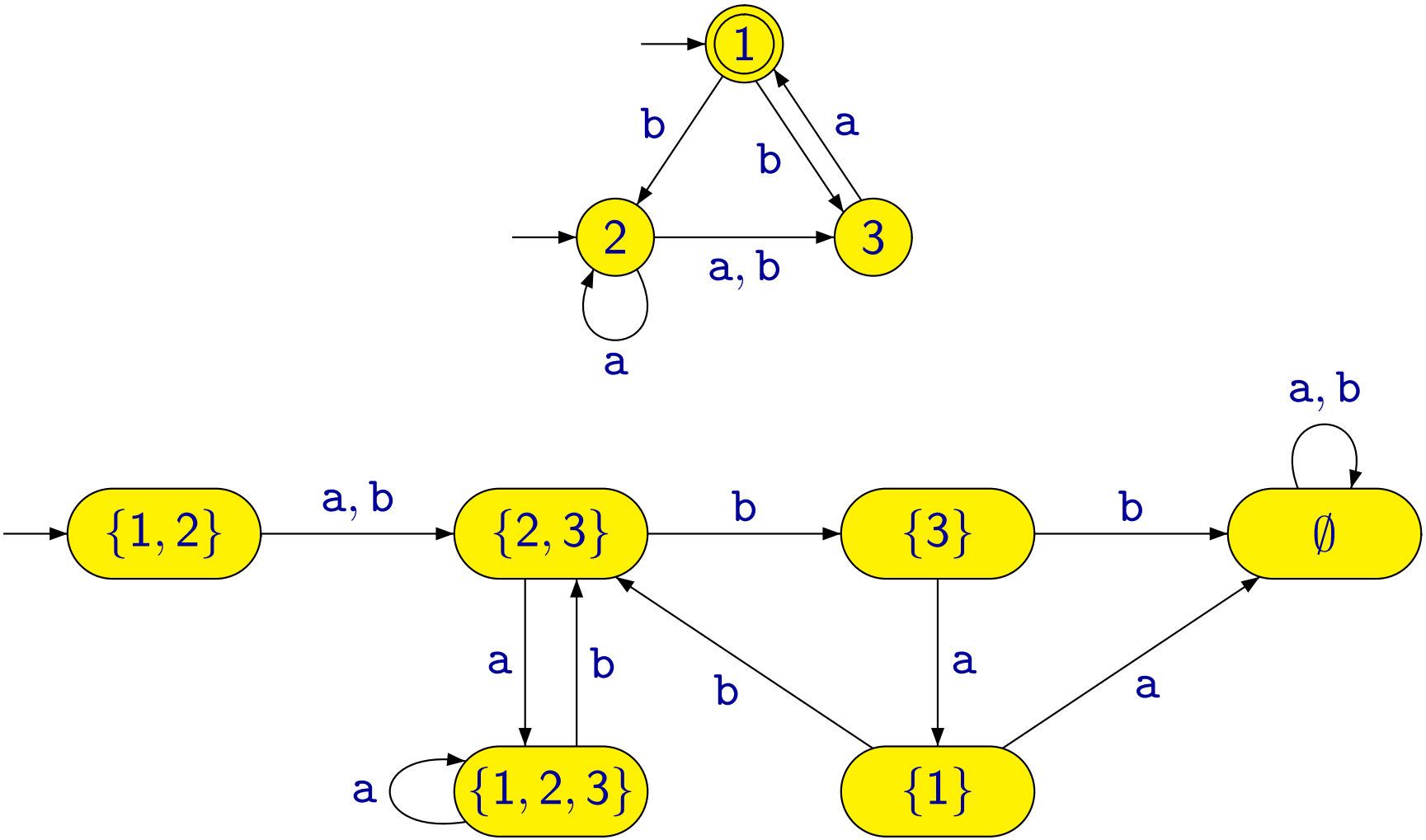


Převod nedeterministického automatu na deterministický

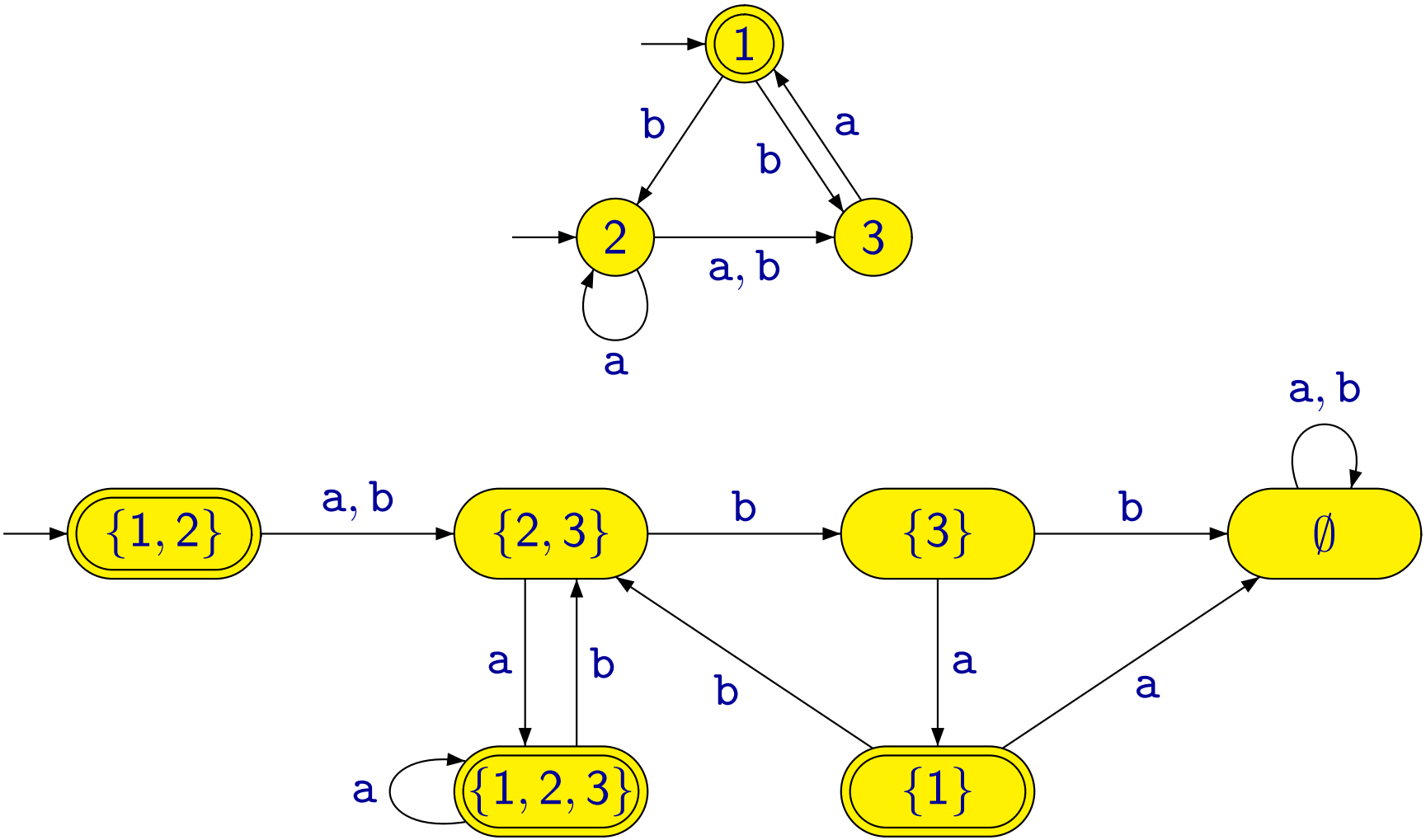




Převod nedeterministického automatu na deterministický



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

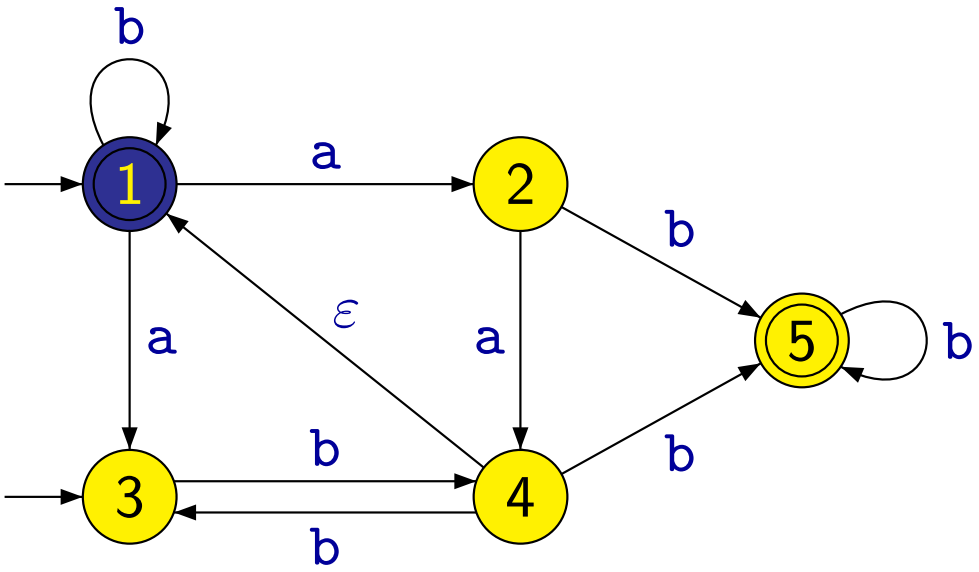
Převod nedeterministického automatu na deterministický

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

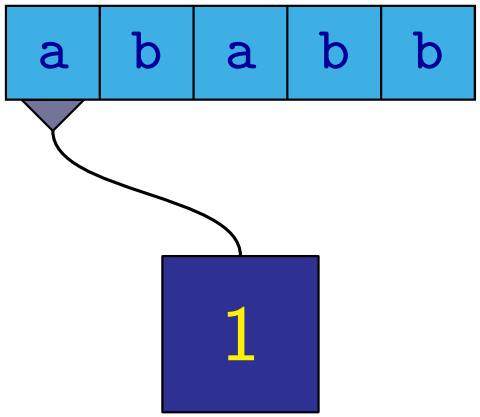
Například při převodu automatu, který má 20 stavů, může vzniknout automat, který má $2^{20} = 1048576$ stavů.

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

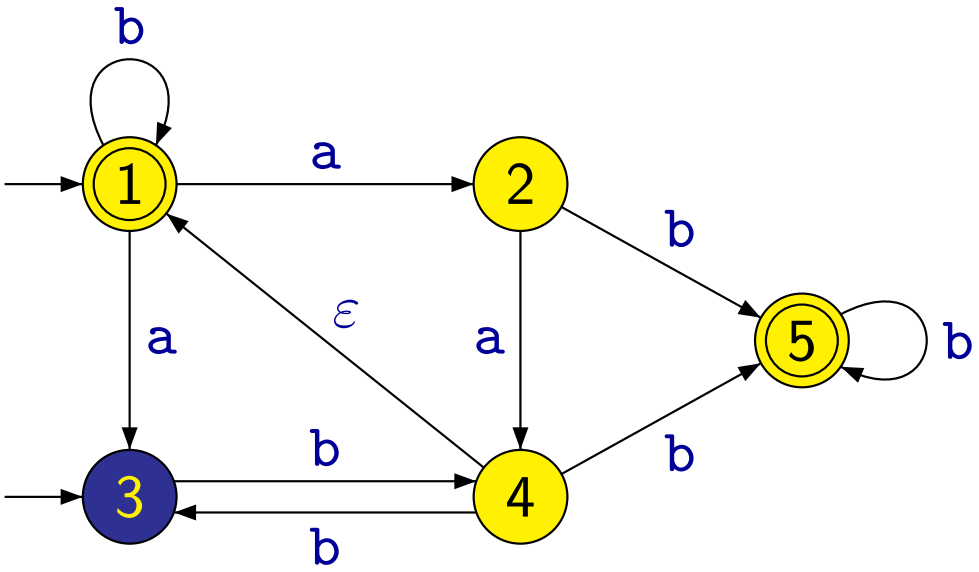
Zobecněný nedeterministický konečný automat

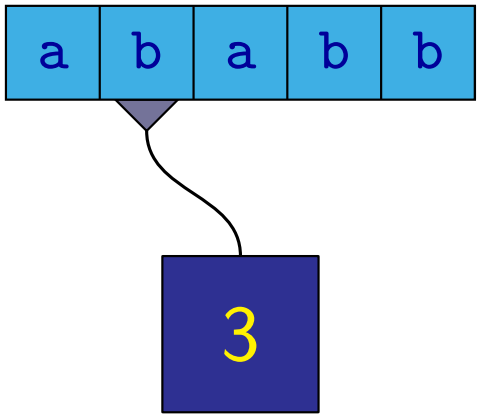


$(1, ababb)$



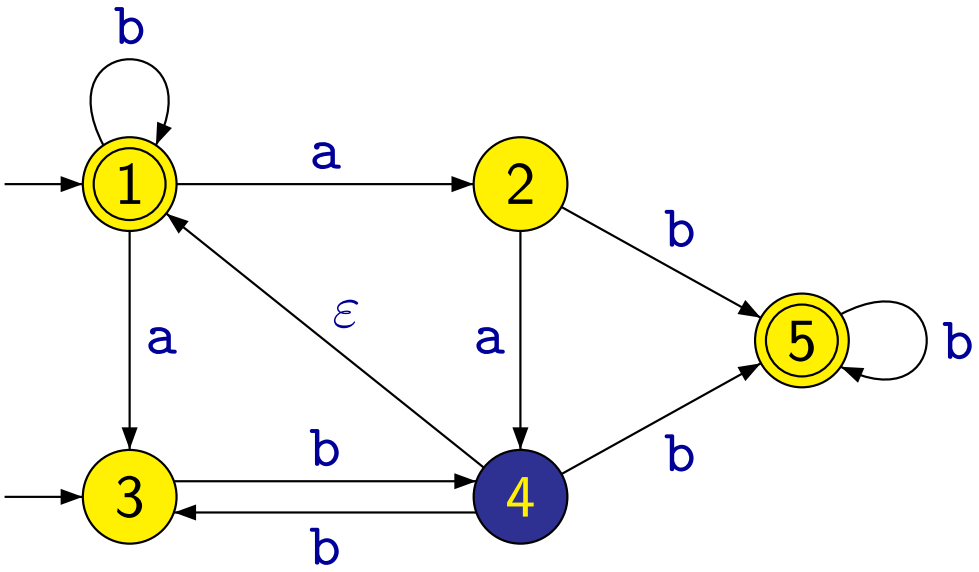
Zobecněný nedeterministický konečný automat

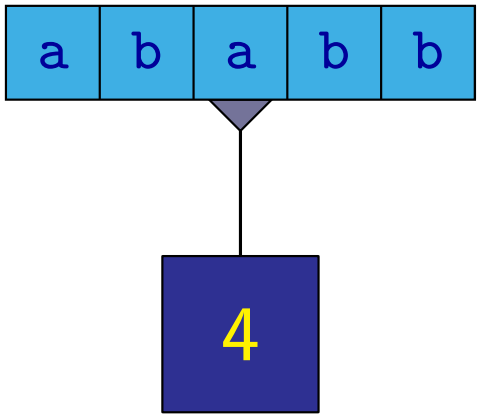




$(1, \mathtt{ababb})$
$\vdash (3, \mathtt{babb})$

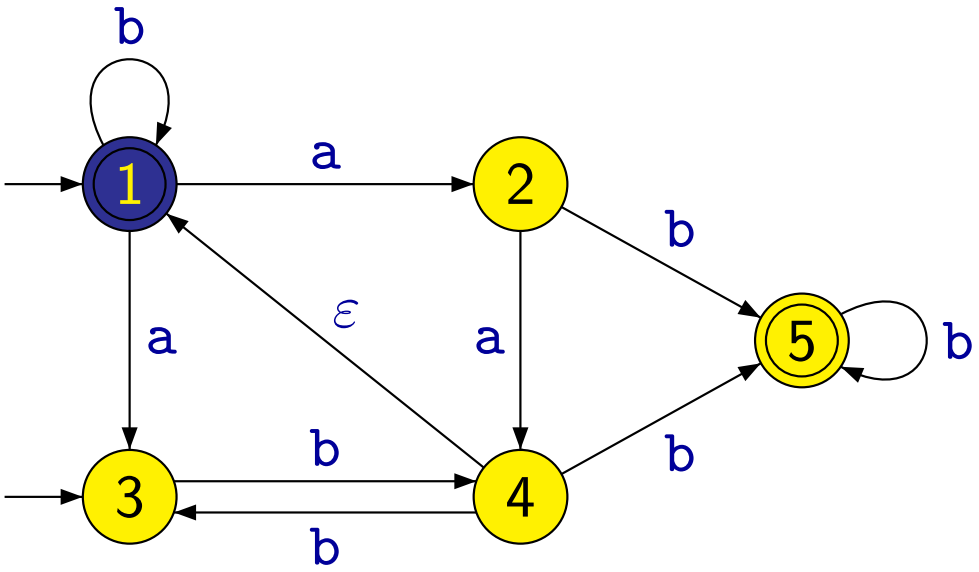
Zobecněný nedeterministický konečný automat

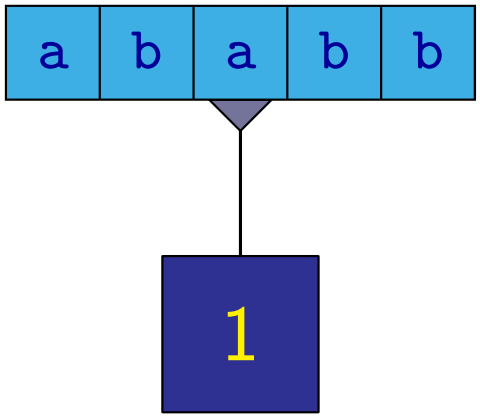




$$(1, \mathtt{ababb}) \vdash (3, \mathtt{babb}) \vdash (4, \mathtt{abb})$$

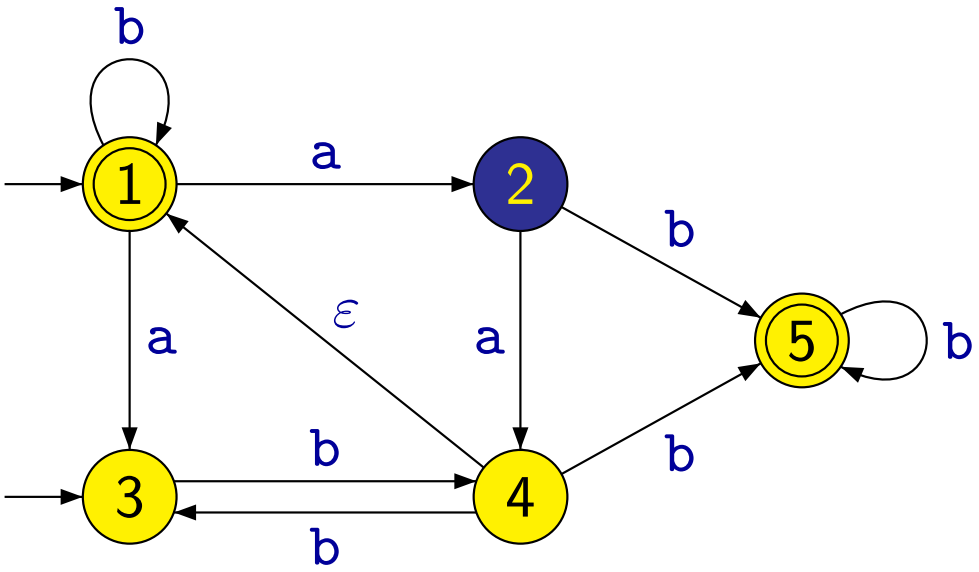
Zobecněný nedeterministický konečný automat

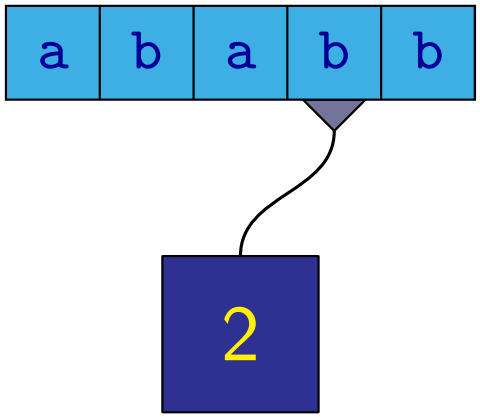




$$(1, \mathtt{ababb}) \vdash (3, \mathtt{babb}) \vdash (4, \mathtt{abb}) \vdash (1, \mathtt{abb})$$

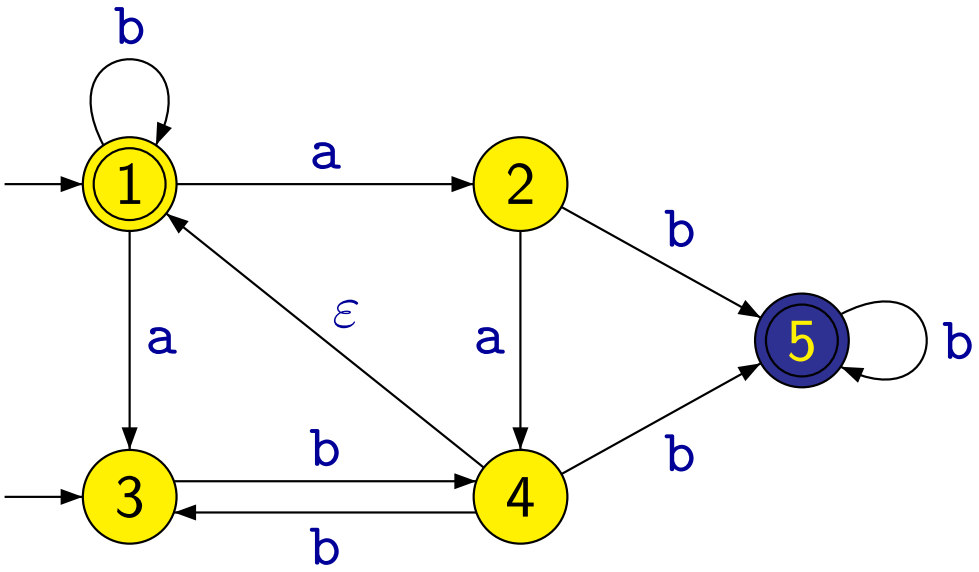
Zobecněný nedeterministický konečný automat

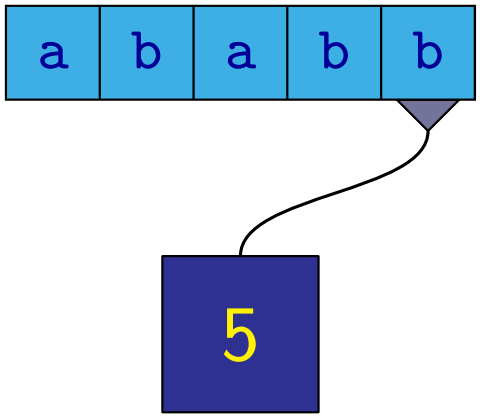




$$(1, \mathtt{ababb}) \vdash (3, \mathtt{babb}) \vdash (4, \mathtt{abb}) \vdash (1, \mathtt{abb}) \vdash (2, \mathtt{bb})$$

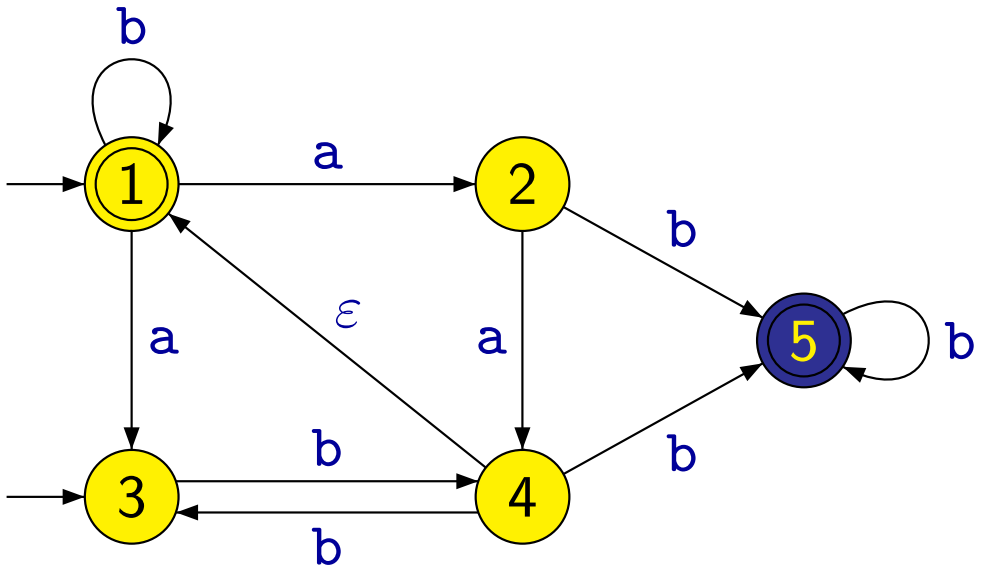
Zobecněný nedeterministický konečný automat

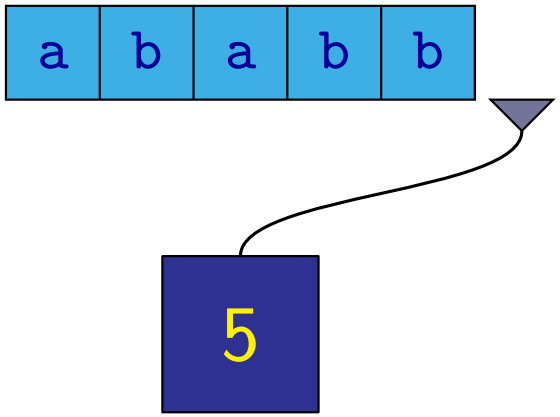




$$
\begin{aligned}
&(1, \mathtt{ababb}) \\
&\vdash (3, \mathtt{babb}) \\
&\vdash (4, \mathtt{abb}) \\
&\vdash (1, \mathtt{abb}) \\
&\vdash (2, \mathtt{bb}) \\
&\vdash (5, \mathtt{b})
\end{aligned}
$$

Zobecněný nedeterministický konečný automat





$$
\begin{aligned}
&(1, \mathtt{ababb}) \\
&\vdash (3, \mathtt{babb}) \\
&\vdash (4, \mathtt{abb}) \\
&\vdash (1, \mathtt{abb}) \\
&\vdash (2, \mathtt{bb}) \\
&\vdash (5, \mathtt{b}) \\
&\vdash (5, \varepsilon)
\end{aligned}
$$

Zobecněný nedeterministický konečný automat

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

Při provádění ε-přechodu se mění pouze stav řídící 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 ε-přechody) bez ohledu na délku slova na pásce.

Zobecněný nedeterministický konečný automat

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

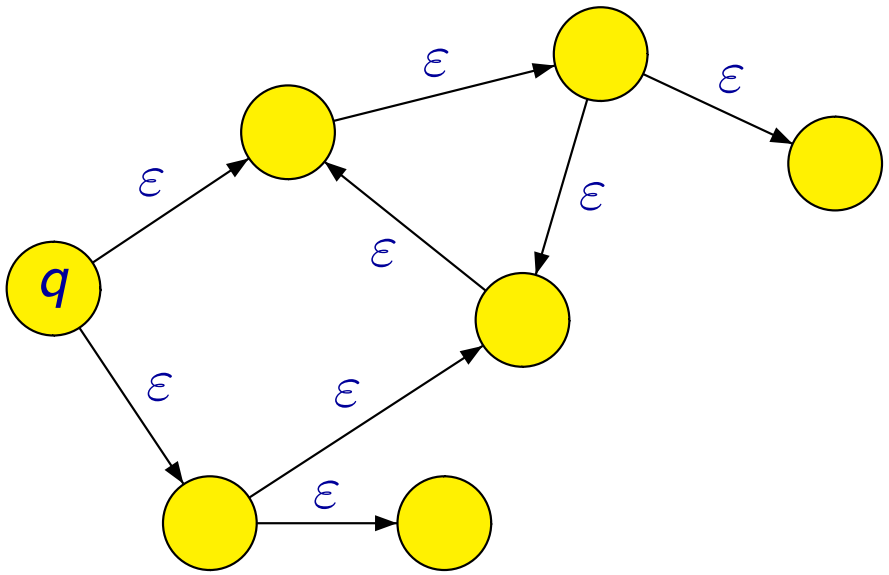
$$(Q, \Sigma, \delta, I, F)$$

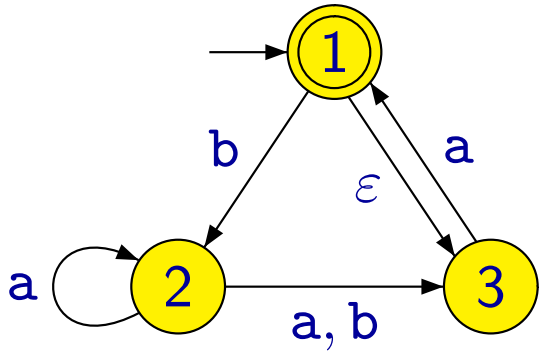
kde:

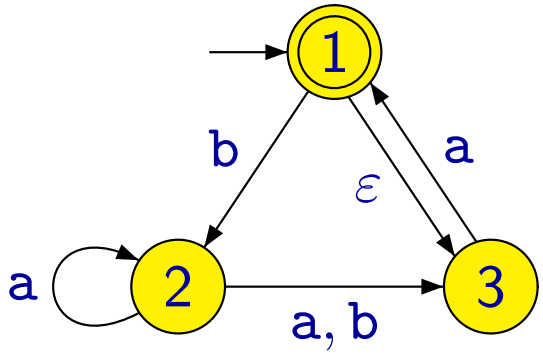
- Q je konečná množina **stavů**
- Σ je konečná **abeceda**
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$ je **přechodová funkce**
- $I \subseteq Q$ je množina **počátečních stavů**
- $F \subseteq Q$ je množina **přijímajících stavů**

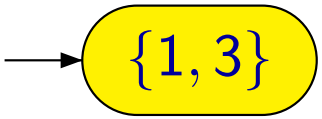
Převod na deterministický konečný automat

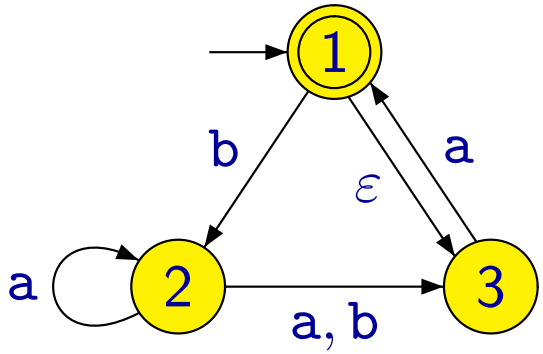
Zobecněný nedeterministický konečný automat je možné převést na deterministický podobnou konstrukcí jako nedeterministický konečný automat, s tím rozdílem, že do množin stavů musíme vždy přidat navíc i všechny stavy dosažitelné daných stavů nějakou sekvencí ε-přechodů.

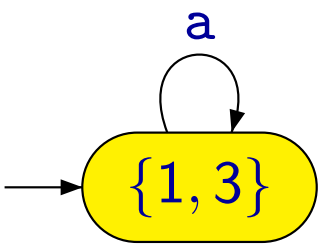


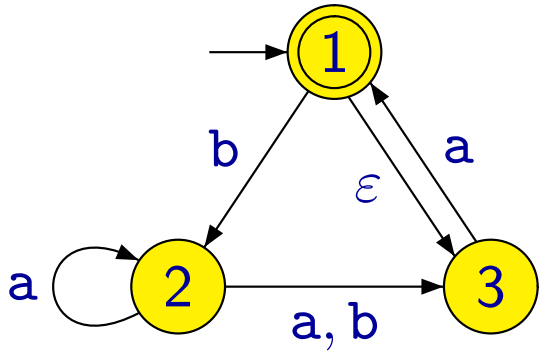

1
b
a
ε
a
2
3
a, b

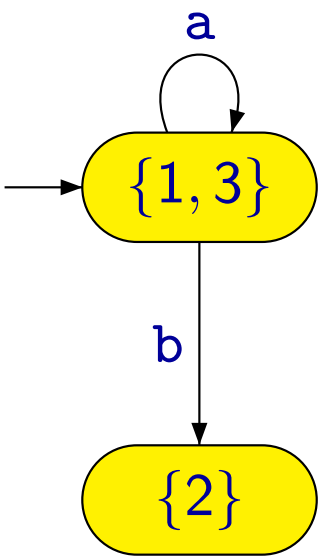

1
b
a
ε
a
2
3
a, b

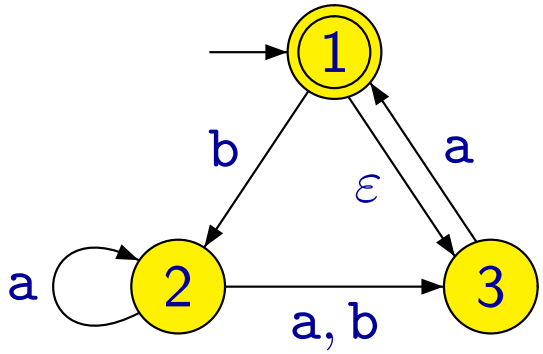

{1, 3}

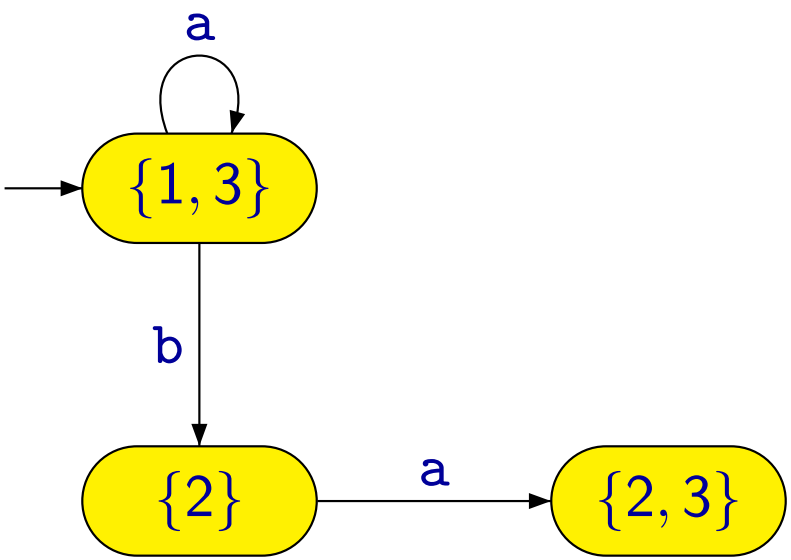

1
b
a
ε
a
2
3
a, b

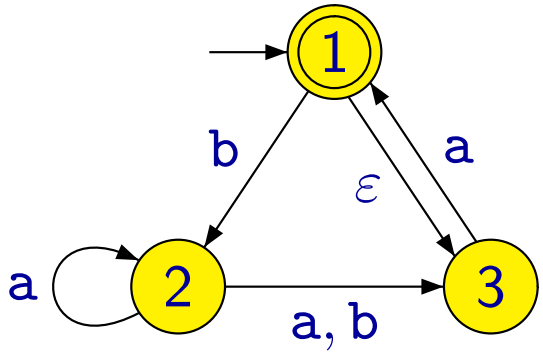

a
{1, 3}

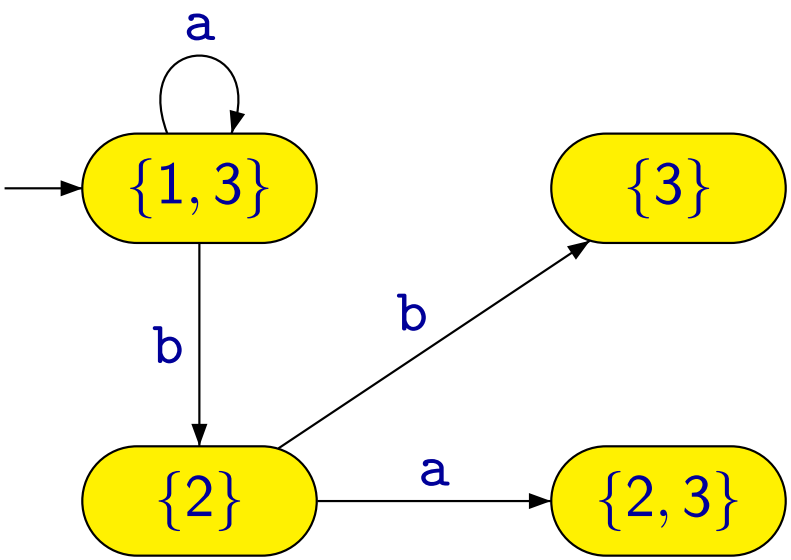

1
b
a
ε
a
2
3
a, b

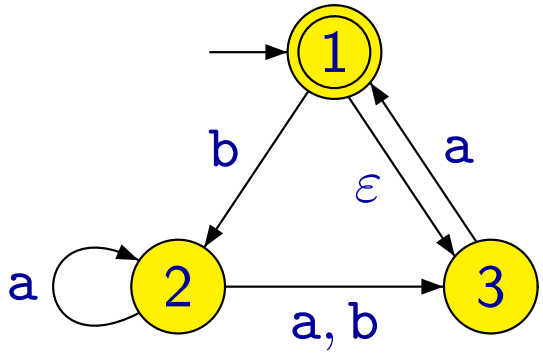

a
{1, 3}
b
{2}

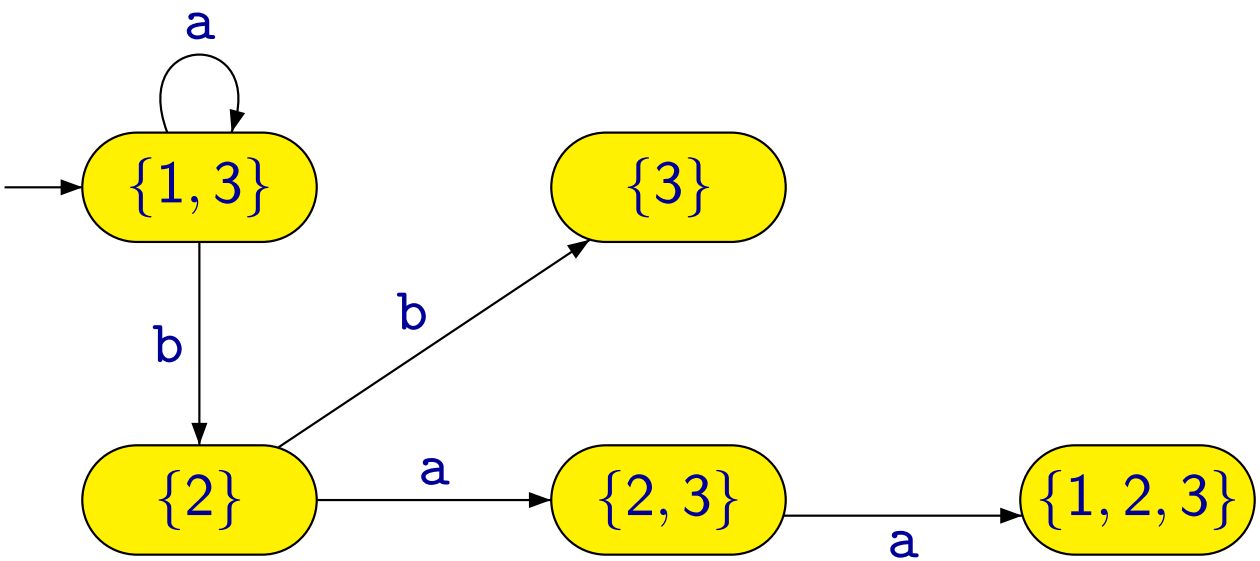

1
b
a
ε
a
2
3
a, b

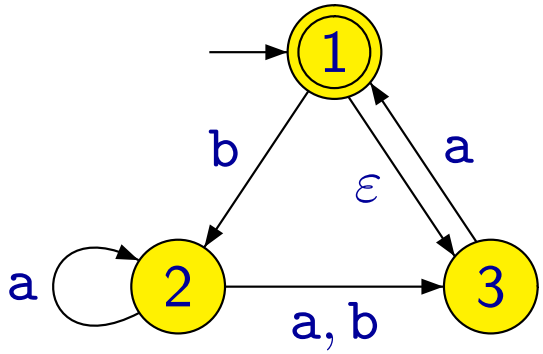

a
{1, 3}
b
{2}
a
{2, 3}

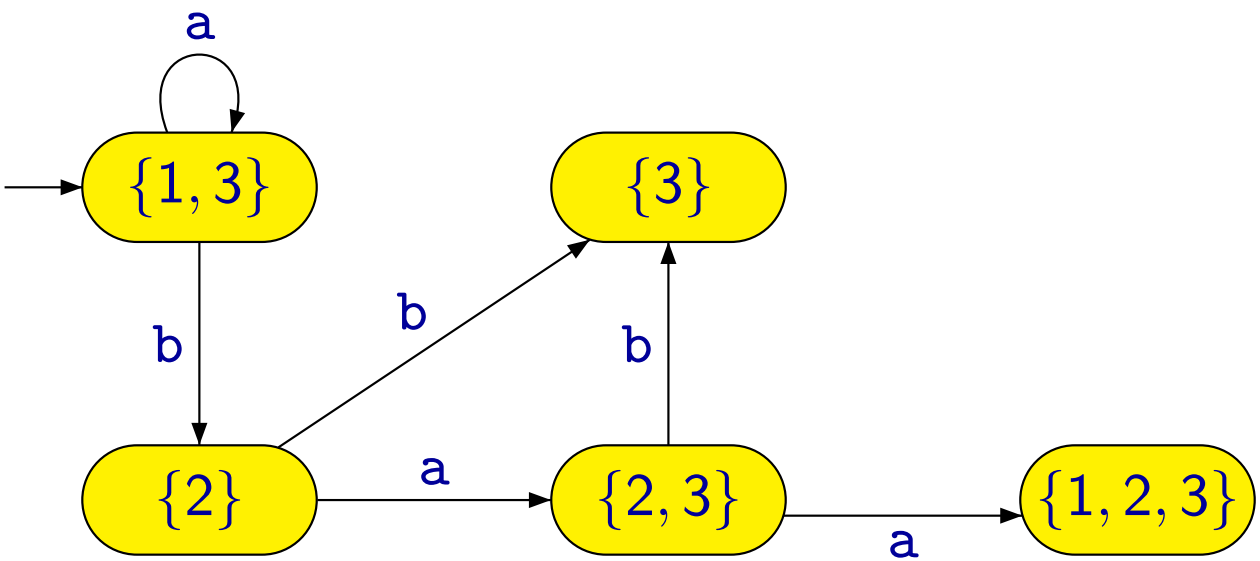

1
b
a
ε
a
2
3
a, b

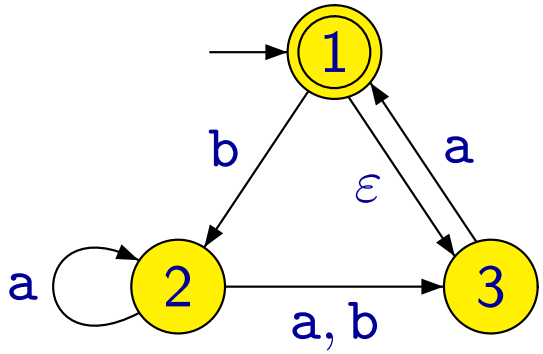

a
{1, 3}
{3}
b
b
{2}
a
{2, 3}

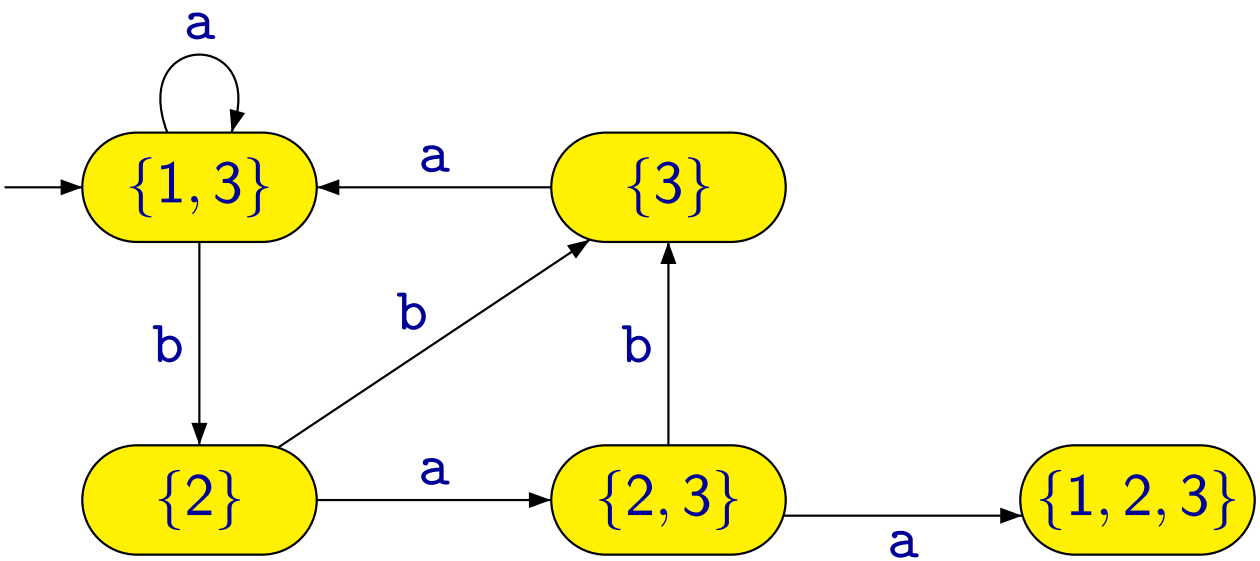

1
2
3
b
a
ε
a
a, b

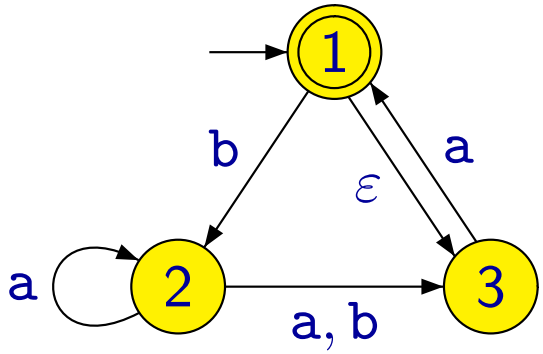

a
{1, 3}
{3}
b
b
{2}
a
{2, 3}
{1, 2, 3}
a

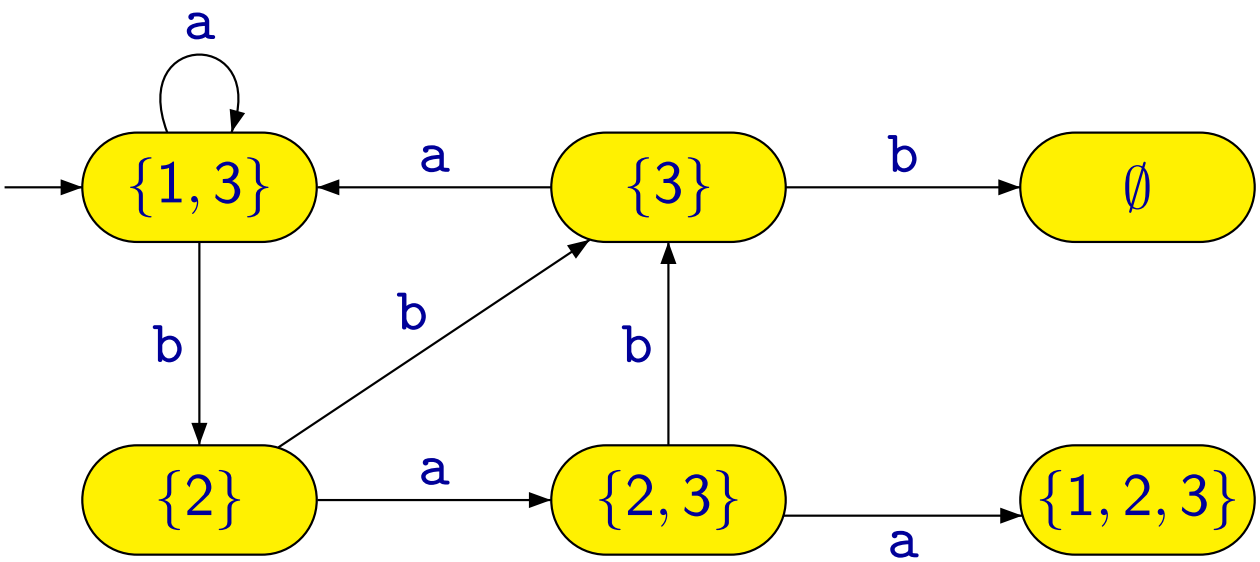

1
b
a
ε
a
2
3
a, b

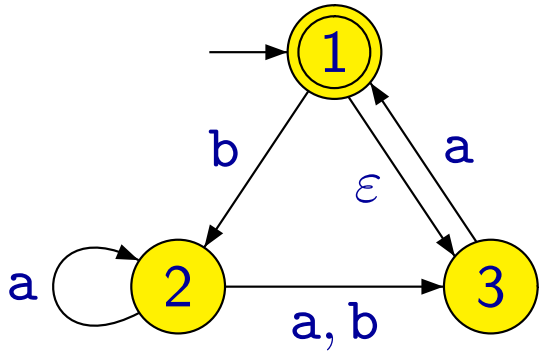

a
{1, 3}
{3}
b
b
b
{2}
a
{2, 3}
{1, 2, 3}
a

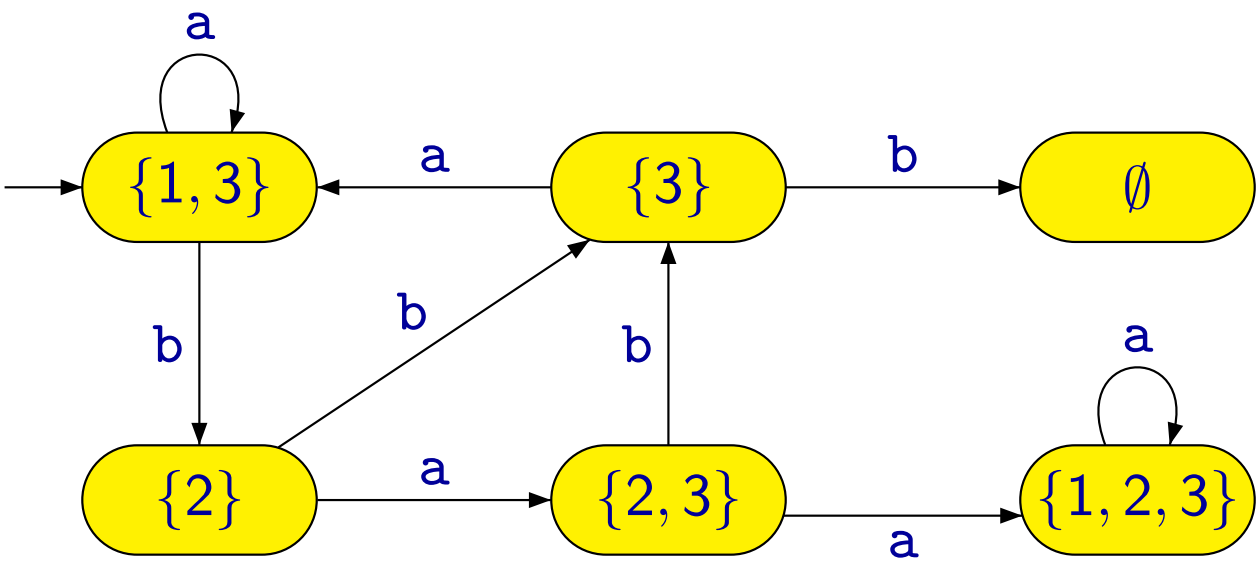

1
b
a
ε
a
2
3
a, b

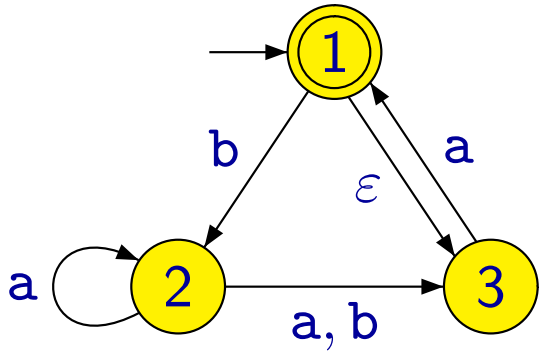

a
{1, 3}
a
{3}
b
b
b
{2}
a
{2, 3}
a
{1, 2, 3}

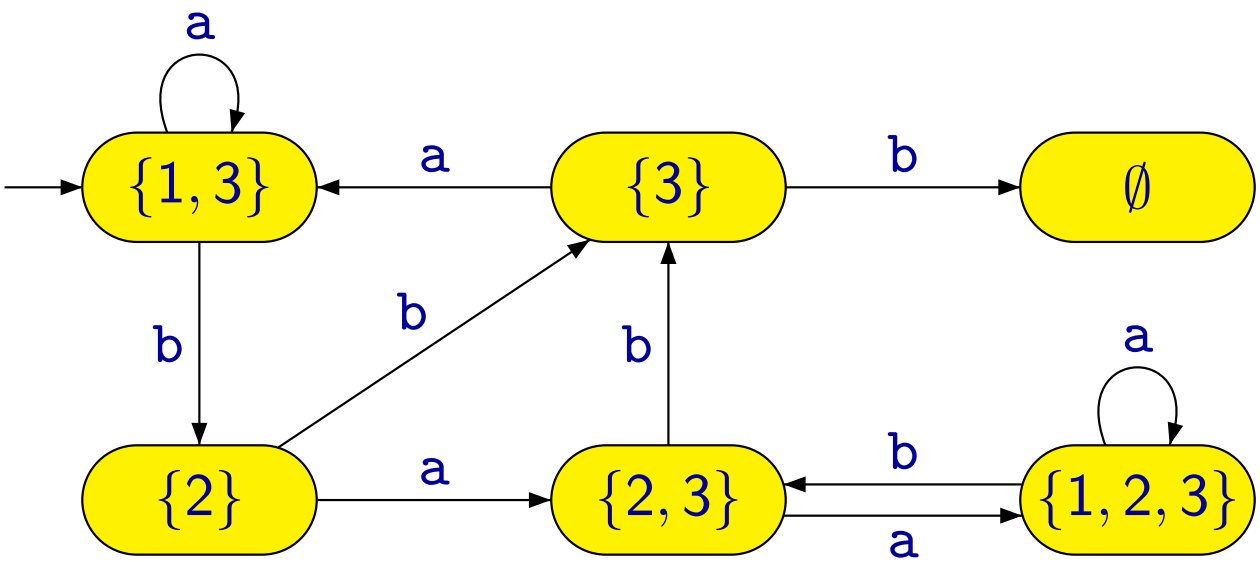

1
b
a
ε
a
2
3
a, b

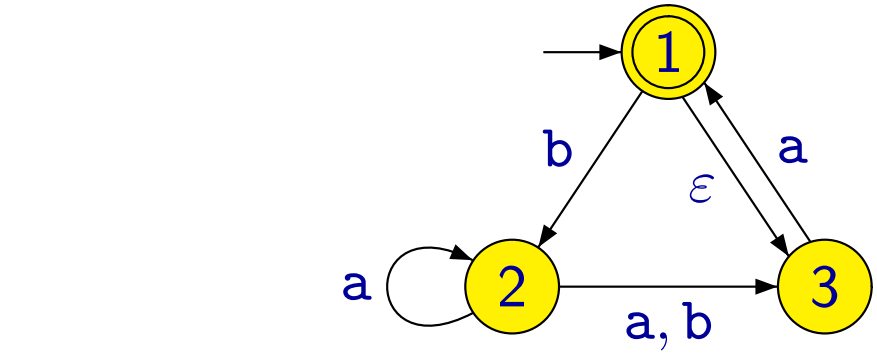

a
{1, 3}
a
{3}
b
∅
b
b
b
{2}
a
{2, 3}
{1, 2, 3}
a

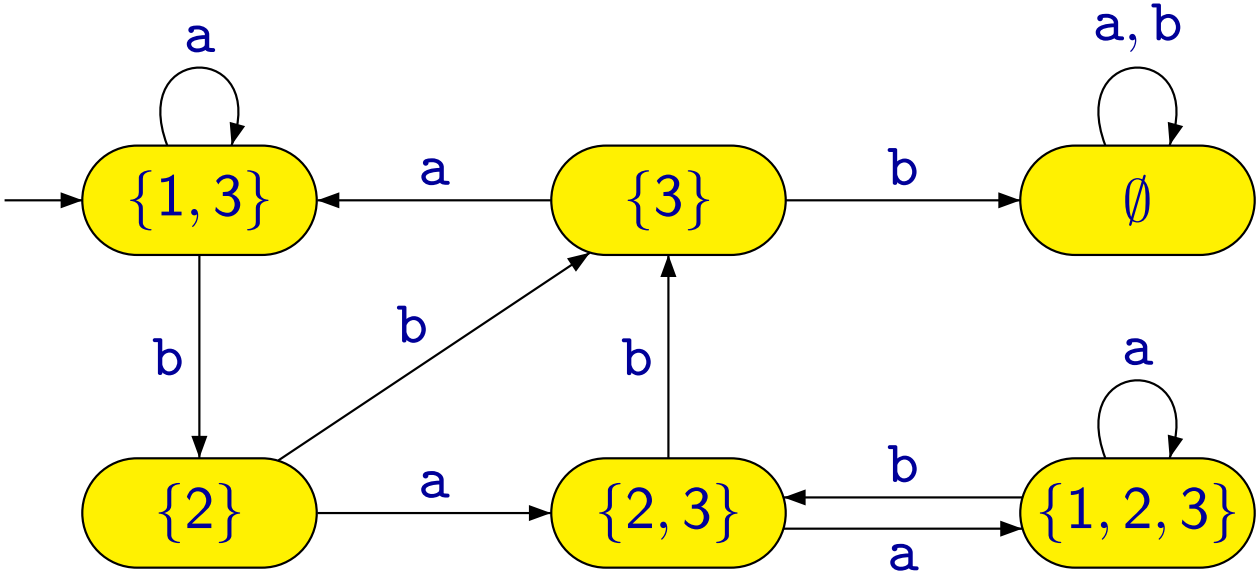

1
b
a
ε
a
2
3
a, b

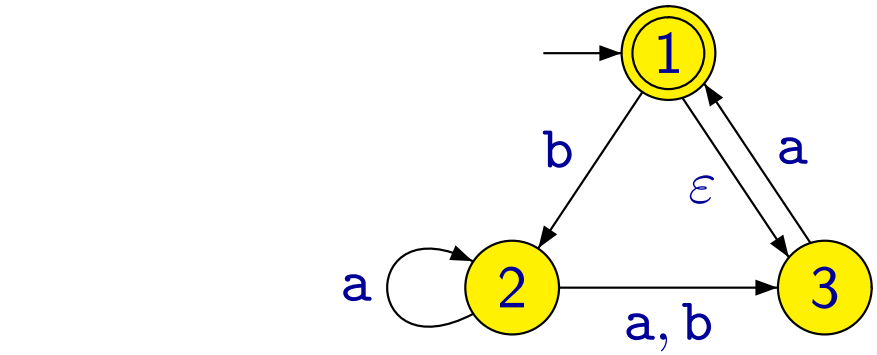

a
{1, 3}
a
{3}
b
∅
b
b
b
a
{2}
a
{2, 3}
{1, 2, 3}
a

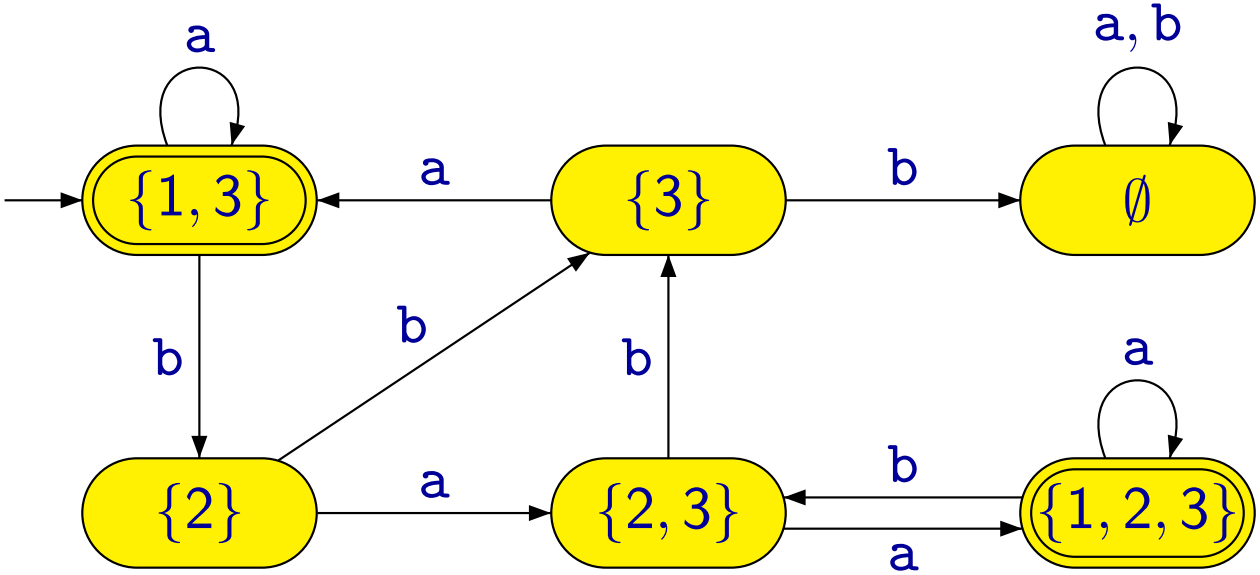

1
b
a
ε
a
2
3
a, b


a
{1, 3}
a
{3}
b
∅
b
b
b
a
{2}
a
{2, 3}
b
{1, 2, 3}
a


1
2
3
b
a
ε
a
a, b


a
a, b
{1, 3}
a
{3}
b
∅
b
b
b
a
{2}
a
{2, 3}
b
{1, 2, 3}
a

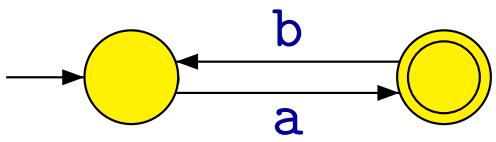

1
2
3
b
a
ε
a
a, b


{1, 3}
{3}
∅
{2}
{2, 3}
{1, 2, 3}
a
a, b
a
b
b
b
b
a
b
a

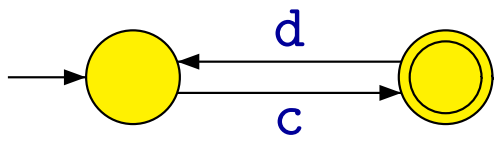
Zřetězení jazyků

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

A_1:



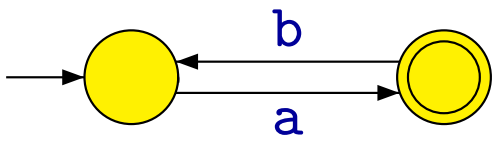
A_2:



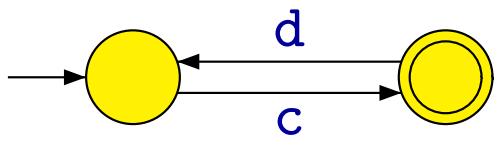
Zřetězení jazyků

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

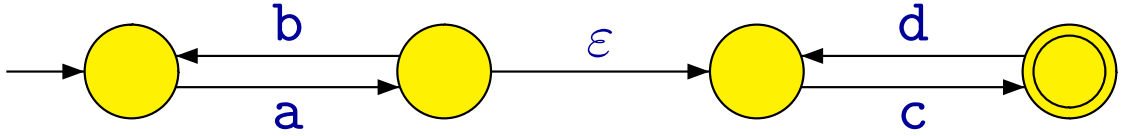
A_1:



A_2:



A:

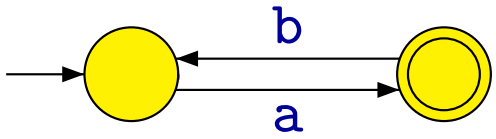


$L(A) = L(A_1) \cdot L(A_2)$

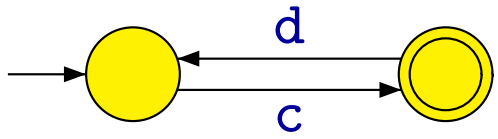
Zřetězení jazyků

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

A_1:

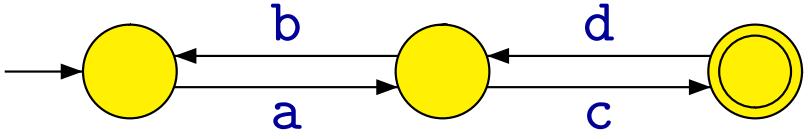


A_2:



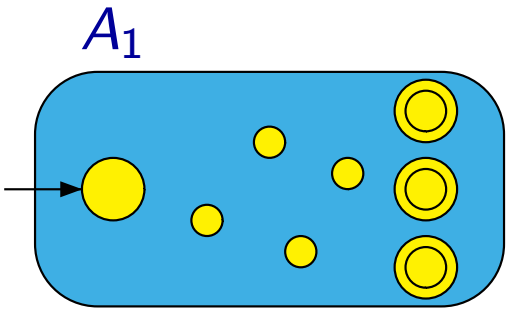
Chybná konstrukce:

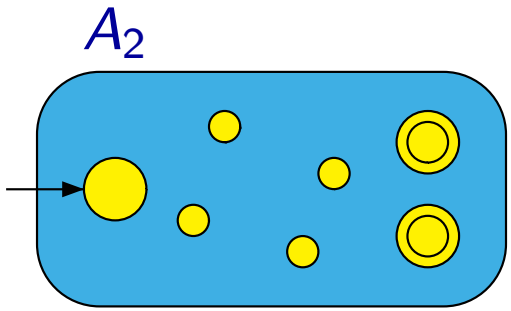
A:



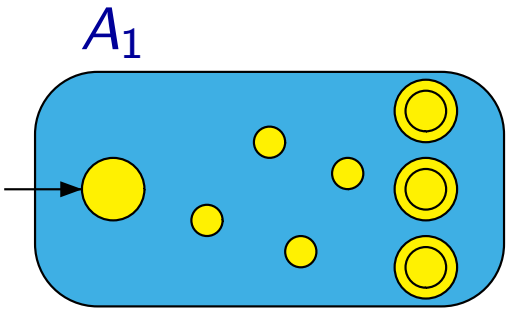
$\mathtt{acdbac} \in L(A)$, ale $\mathtt{acdbac} \notin L(A_1) \cdot L(A_2)$

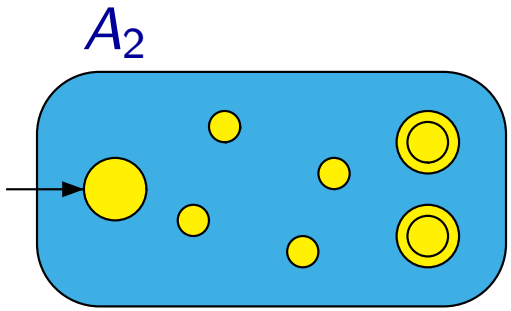
Zřetězení jazyků

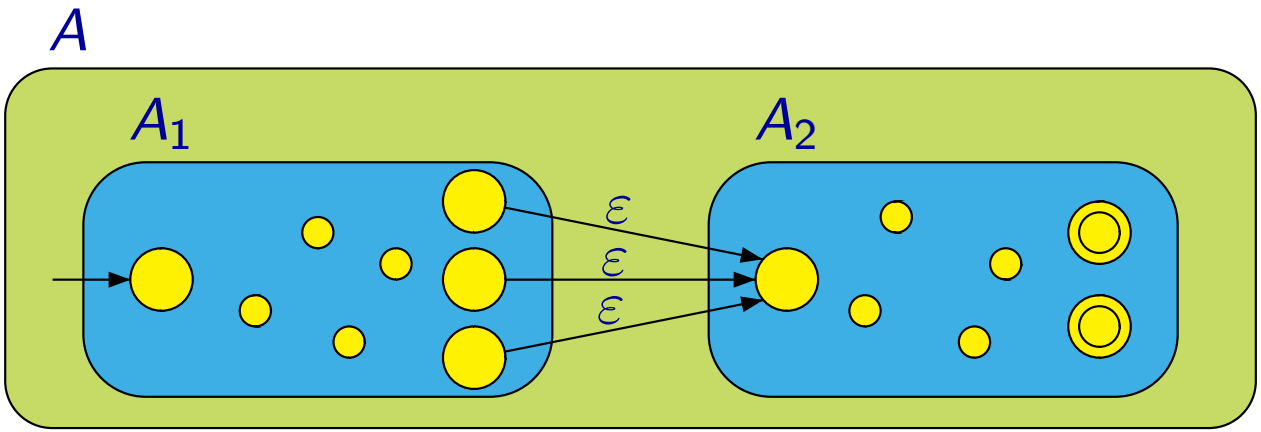




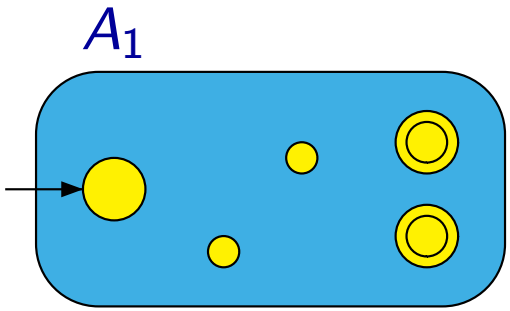
Zřetězení jazyků



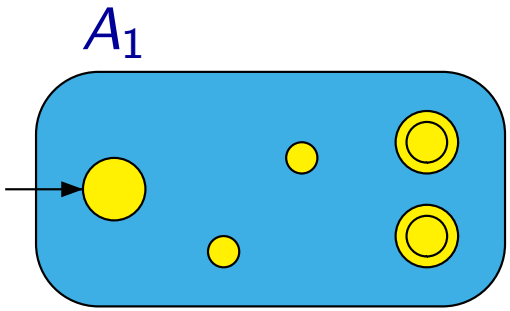


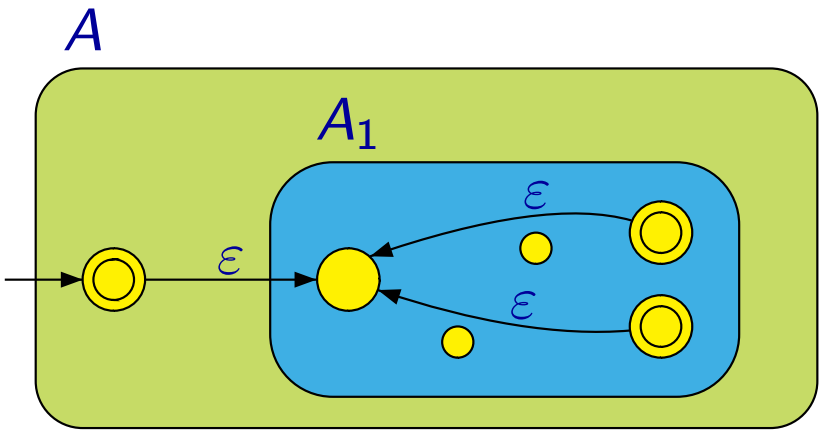


Iterace jazyka



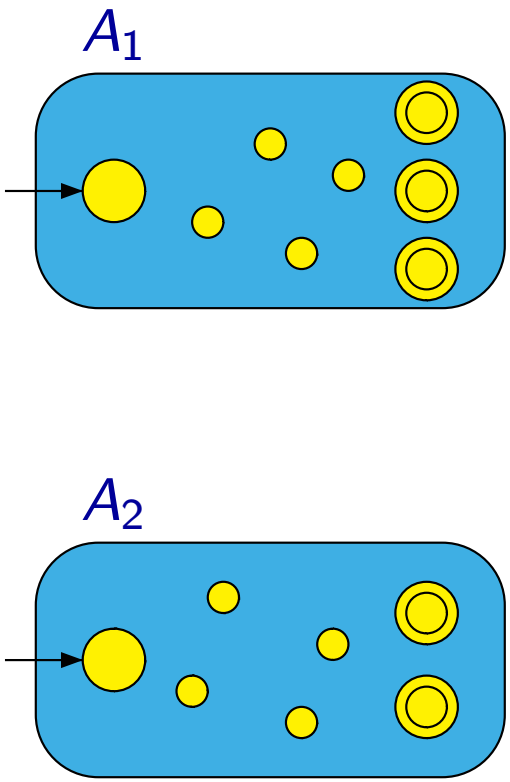
Iterace jazyka





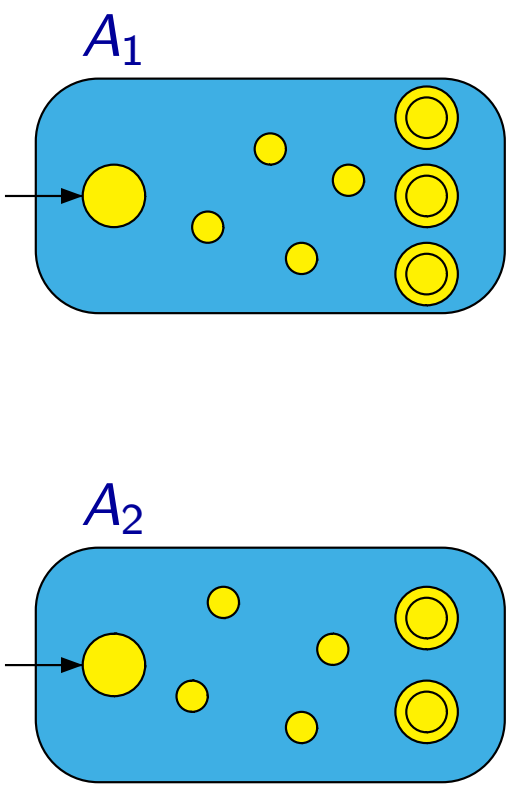
Sjednocení jazyků

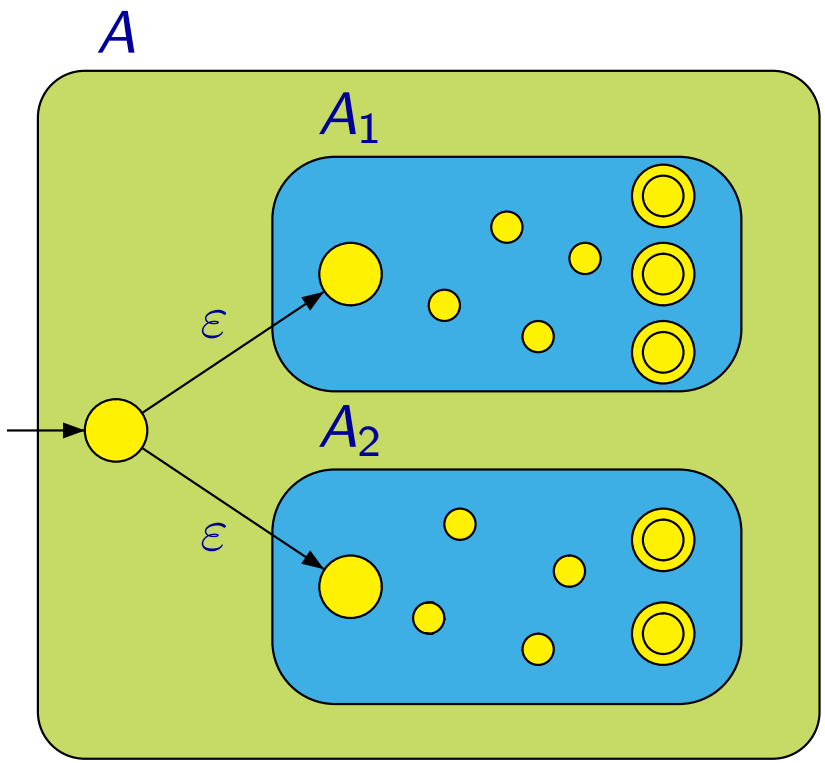
Alternativní konstrukce pro sjednocení jazyků:



Sjednocení jazyků

Alternativní konstrukce pro sjednocení jazyků:





Uzavřenost množiny vůči operacím

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

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

Příklad: Uvažujme množinu přirozených čísel $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$ 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 $\mathbb{R} - \{0\}$ je uzavřená vůči operaci $/$.

Uzavřenost třídy regulárních jazyků

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

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

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

Regulární výrazy

Induktivní definice regulárních výrazů nad abecedou Σ:

- $\emptyset$, ε, a (kde $a \in \Sigma$) jsou regulární výrazy:

  $\emptyset$ ... označuje prázdný jazyk
  ε ... označuje jazyk $\{\varepsilon\}$
  a ... označuje jazyk $\{a\}$

- Jestliže α, β jsou regulární výrazy, pak i $(\alpha + \beta)$, $(\alpha \cdot \beta)$, (α^*) jsou regulární výrazy:

  $(\alpha + \beta)$ ... označuje sjednocení jazyků označených α a β
  $(\alpha \cdot \beta)$ ... označuje zřetězení jazyků označených α a β
  (α^*) ... označuje iteraci jazyka označeného α

- Neexistují žádné další regulární výrazy než ty definované podle předchozích dvou bodů.

Regulární výrazy

Příklad:

- Podle definice jsou 0 i 1 regulární výrazy.

Regulární výrazy

Příklad:

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

Regulární výrazy

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.

Regulární výrazy

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.

Regulární výrazy

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 α je regulární výraz, zápisem $[\alpha]$ označujeme jazyk definovaný regulárním výrazem α.

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

Regulární výrazy

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

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

Příklad: Místo

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

obvykle píšeme

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

Regulární výrazy

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

0 . . . jazyk tvořený jediným slovem 0

Regulární výrazy

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

0 . . . jazyk tvořený jediným slovem 0

01 . . . jazyk tvořený jediným slovem 01

Regulární výrazy

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

Regulární výrazy

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 ε, 0, 00, 000, ...

Regulární výrazy

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 ε, 0, 00, 000, ...

$(01)^*$ ... jazyk tvořený slovy ε, 01, 0101, 010101, ...

Regulární výrazy

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 ε, 0, 00, 000, ...

$(01)^*$ ... jazyk tvořený slovy ε, 01, 0101, 010101, ...

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

Regulární výrazy

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 ε, 0, 00, 000, ...

$(01)^*$ ... jazyk tvořený slovy ε, 01, 0101, 010101, ...

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

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

Regulární výrazy

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 ε, 0, 00, 000, ...

$(01)^*$ ... jazyk tvořený slovy ε, 01, 0101, 010101, ...

$(0 + 1)^*$ ... 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

Regulární výrazy

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

Regulární výrazy

$(0+1)^*00+(01)^*111(01)^*$ ... 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

Regulární výrazy

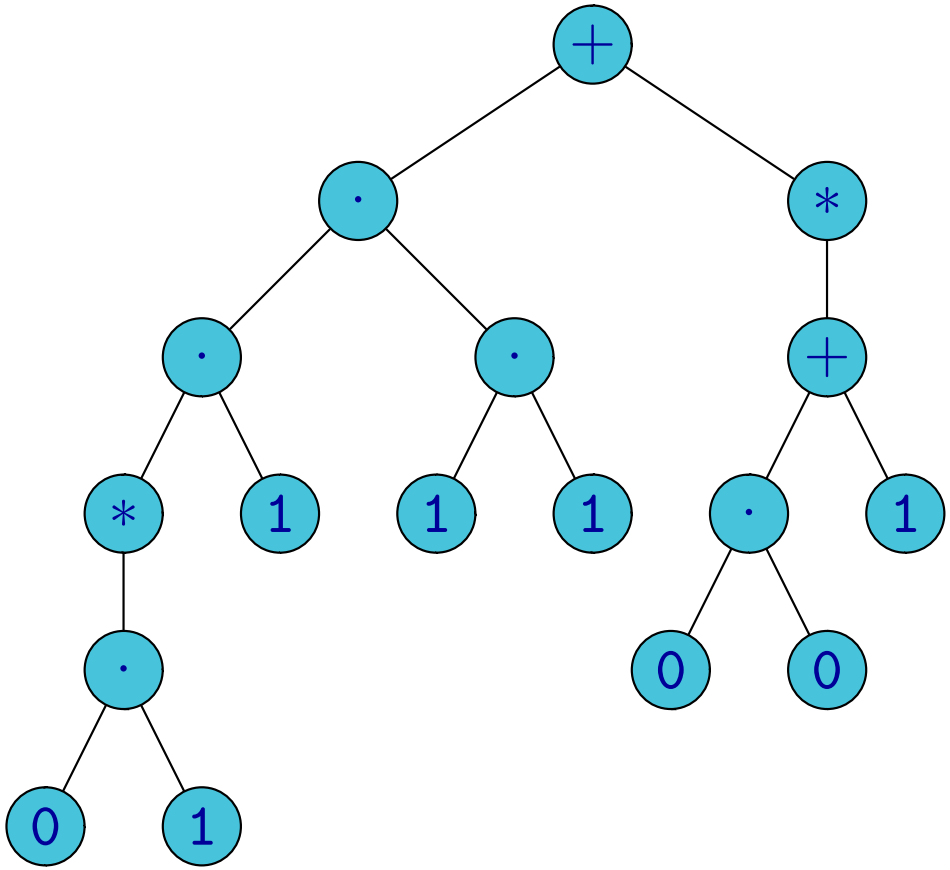
$(0+1)^*00+(01)^*111(01)^*$ ... jazyk tvořený všemi slovy, která buď končí 00 nebo obsahují podslovo 111 předcházené i následované libovolným počtem slov 01

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

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

Regulární výrazy

Strukturu regulárního výrazu si můžeme znázornit jako strom:



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

Převod regulárního výrazu na konečný automat

Tvrzení

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

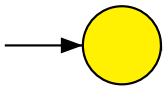
Důkaz: Stačí ukázat, jak k danému regulárnímu výrazu α zkonstruovat konečný automat, který rozpoznává jazyk $[\alpha]$.

Konstrukce je rekurzivní a postupuje podle struktury výrazu α:

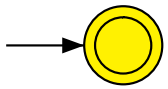
- Pokud je α elementární výraz (tj. $\emptyset$, ε nebo a):
  - Sestrojíme přímo odpovídající automat.
- Pokud je α tvaru $(\beta + \gamma)$, $(\beta \cdot \gamma)$ nebo (β^*):
  - Rekurzivně sestrojíme automaty rozpoznávající jazyky $[\beta]$ a $[\gamma]$.
  - Z nich sestrojíme automat rozpoznávající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

Převod regulárního výrazu na konečný automat

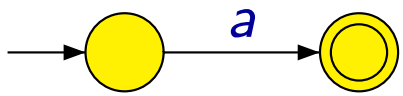
Automaty pro elementární výrazy:



$\emptyset$



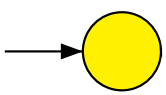
ε



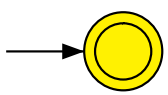
a

Převod regulárního výrazu na konečný automat

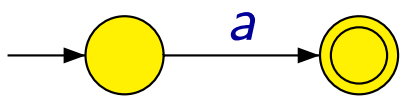
Automaty pro elementární výrazy:



$\emptyset$

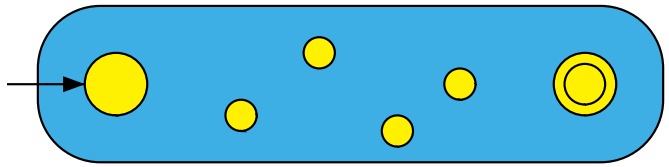


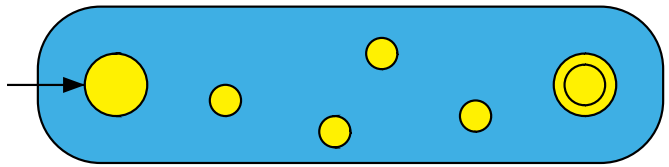
ε



a

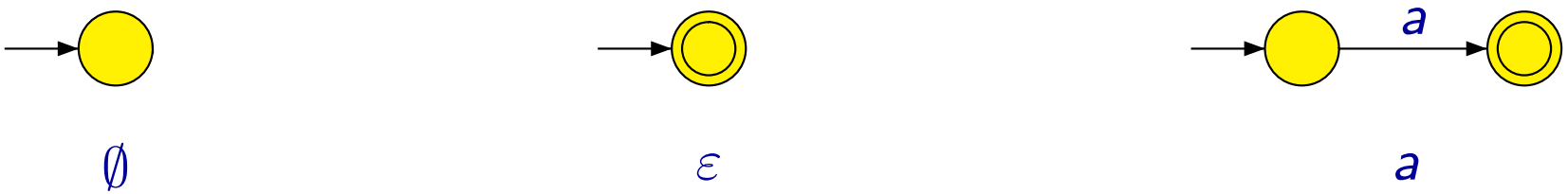
Konstrukce pro sjednocení:



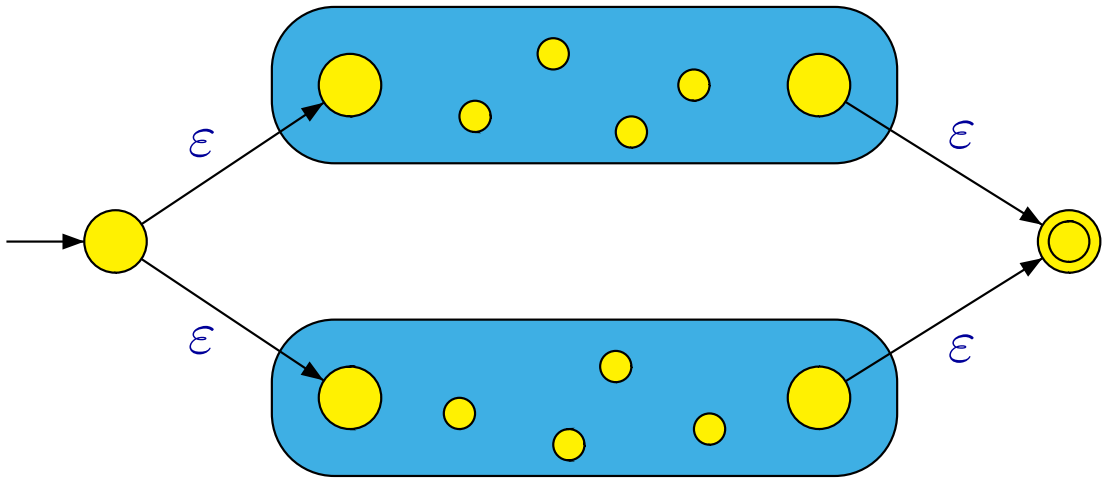


Převod regulárního výrazu na konečný automat

Automaty pro elementární výrazy:

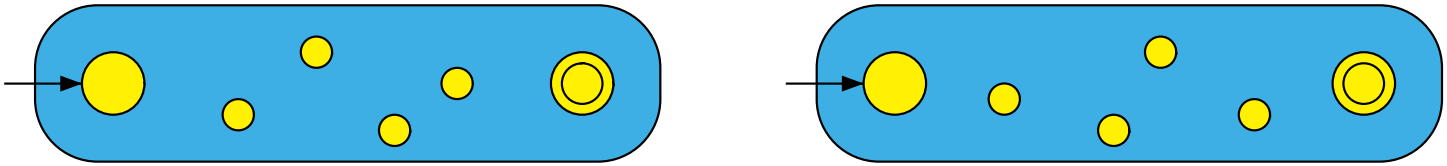


Konstrukce pro sjednocení:



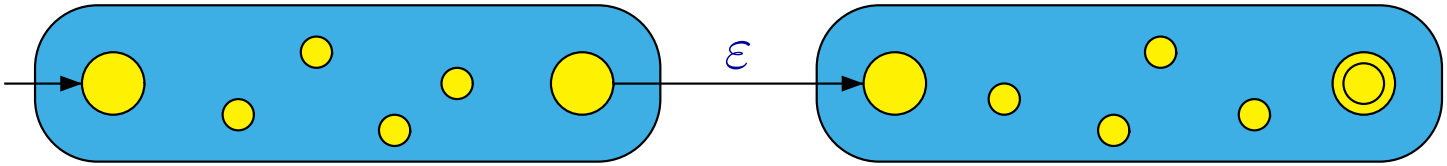
Převod regulárního výrazu na konečný automat

Konstrukce pro zřetězení:



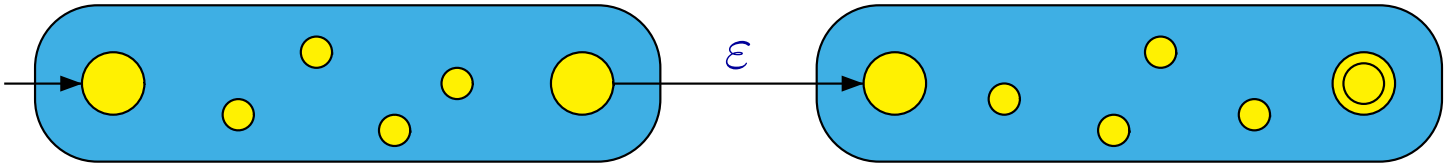
Převod regulárního výrazu na konečný automat

Konstrukce pro zřetězení:

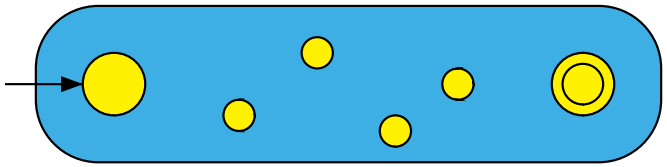


Převod regulárního výrazu na konečný automat

Konstrukce pro zřetězení:

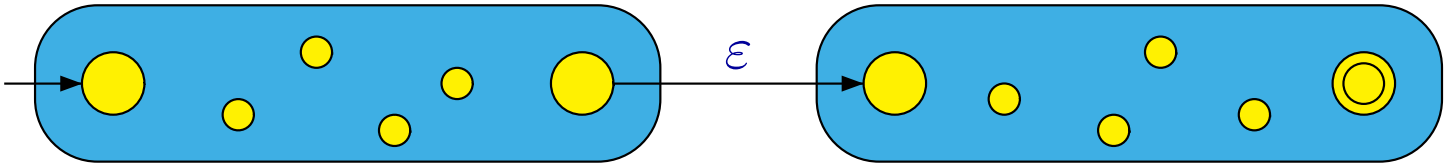


Konstrukce pro iteraci:

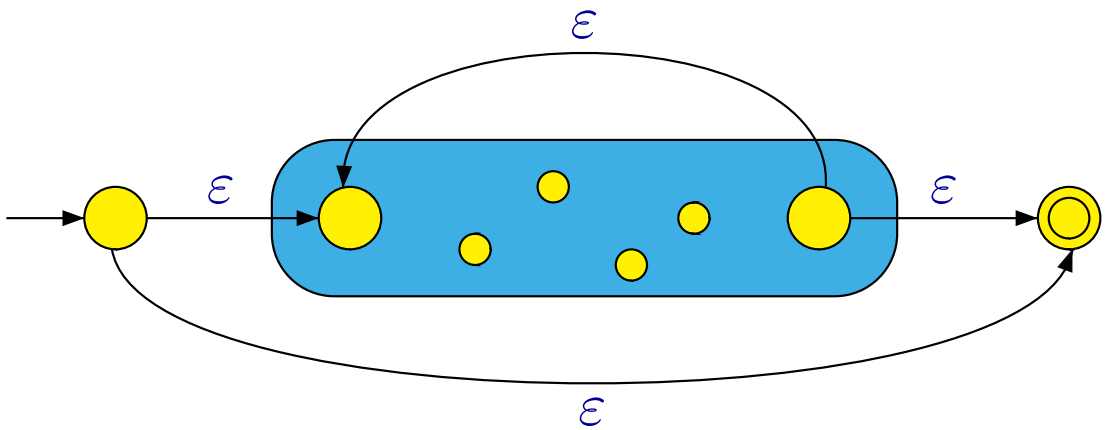


Převod regulárního výrazu na konečný automat

Konstrukce pro zřetězení:



Konstrukce pro iteraci:

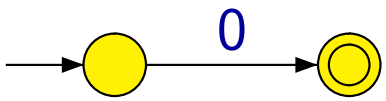


Převod regulárního výrazu na konečný automat

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

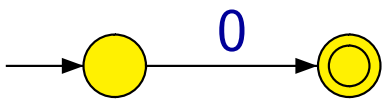
Převod regulárního výrazu na konečný automat

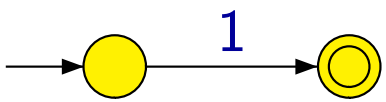
Příklad: Konstrukce automatu pro výraz $((0+1)\cdot 1)^*$:



Převod regulárního výrazu na konečný automat

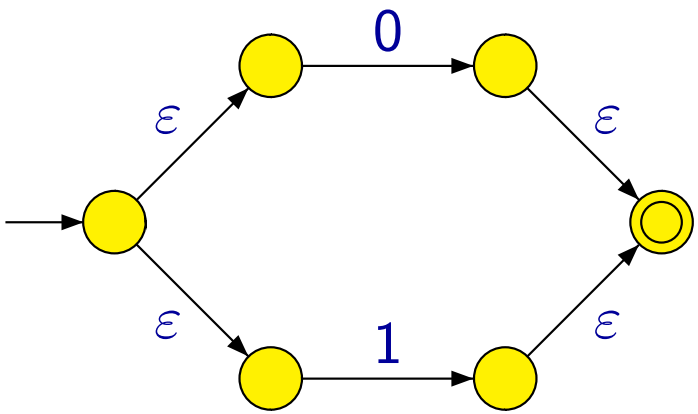
Příklad: Konstrukce automatu pro výraz $((0 + 1) \cdot 1)^*$:





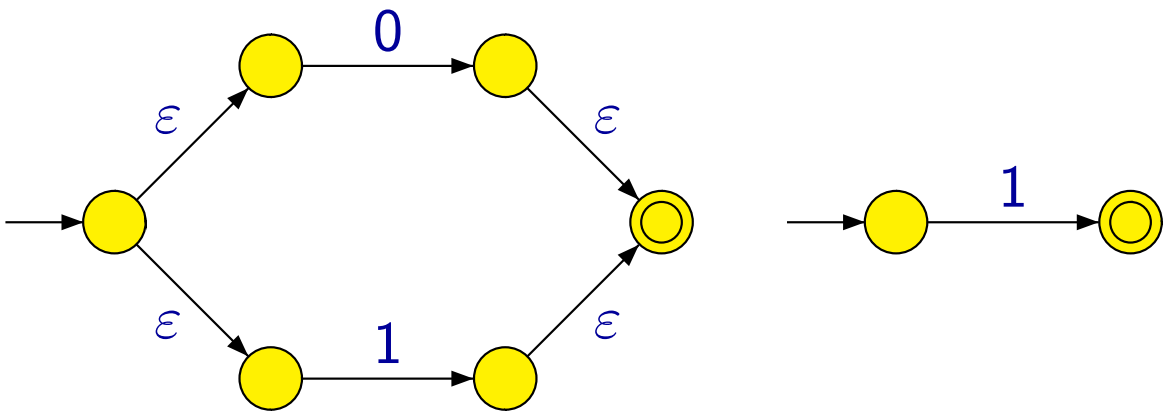
Převod regulárního výrazu na konečný automat

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



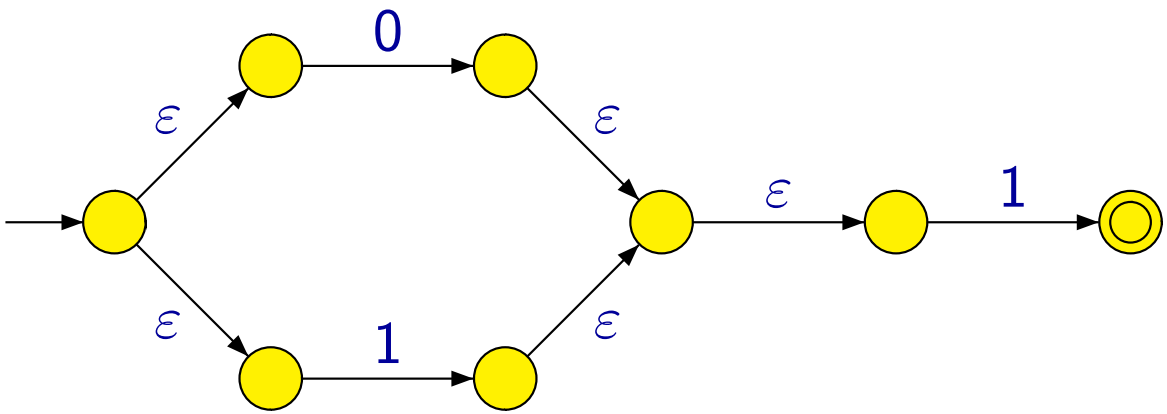
Převod regulárního výrazu na konečný automat

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



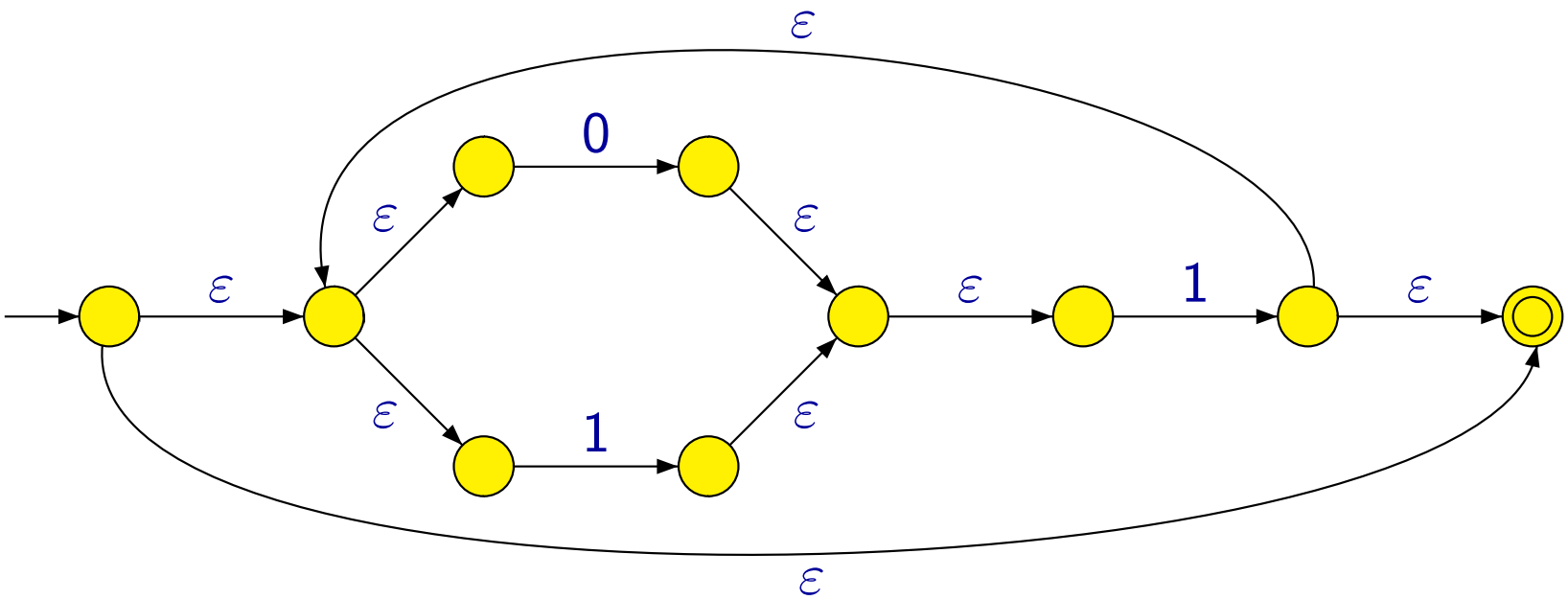
Převod regulárního výrazu na konečný automat

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



Převod regulárního výrazu na konečný automat

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



Převod regulárního výrazu na konečný automat

Pokud se výraz α 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.

Převod konečného automatu na regulární výraz

Tvrzení

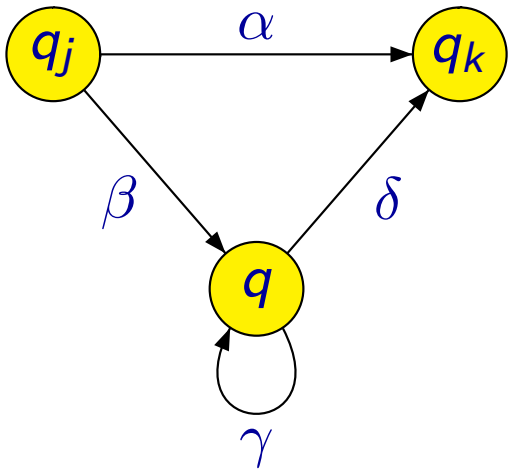
Každý regulární jazyk je možné popsat nějakým regulárním výrazem.

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

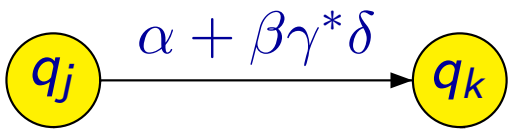
- A upravíme tak, aby měl právě jeden počáteční a právě jeden koncový stav.
- Budeme postupně odebírat jednotlivé stavy.
- Přechody budou označeny regulárními výrazy.
- Zbude automat se dvěma stavy – počátečním a koncovým, a jedním přechodem ohodnoceným výsledným regulárním výrazem.

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.

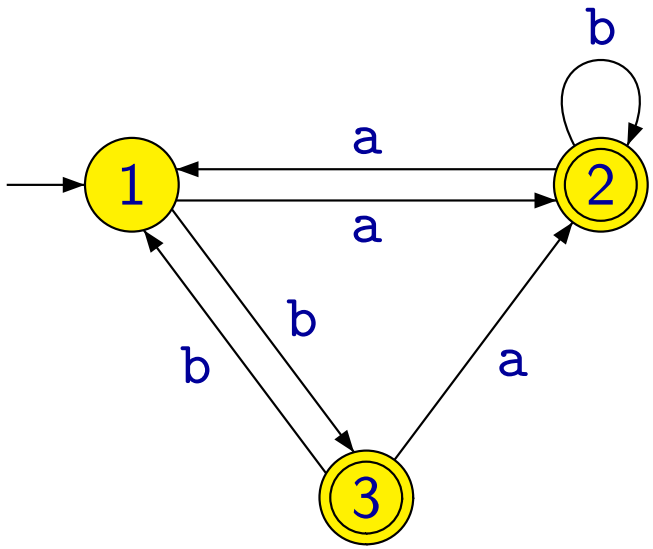


Po odstranění stavu q:



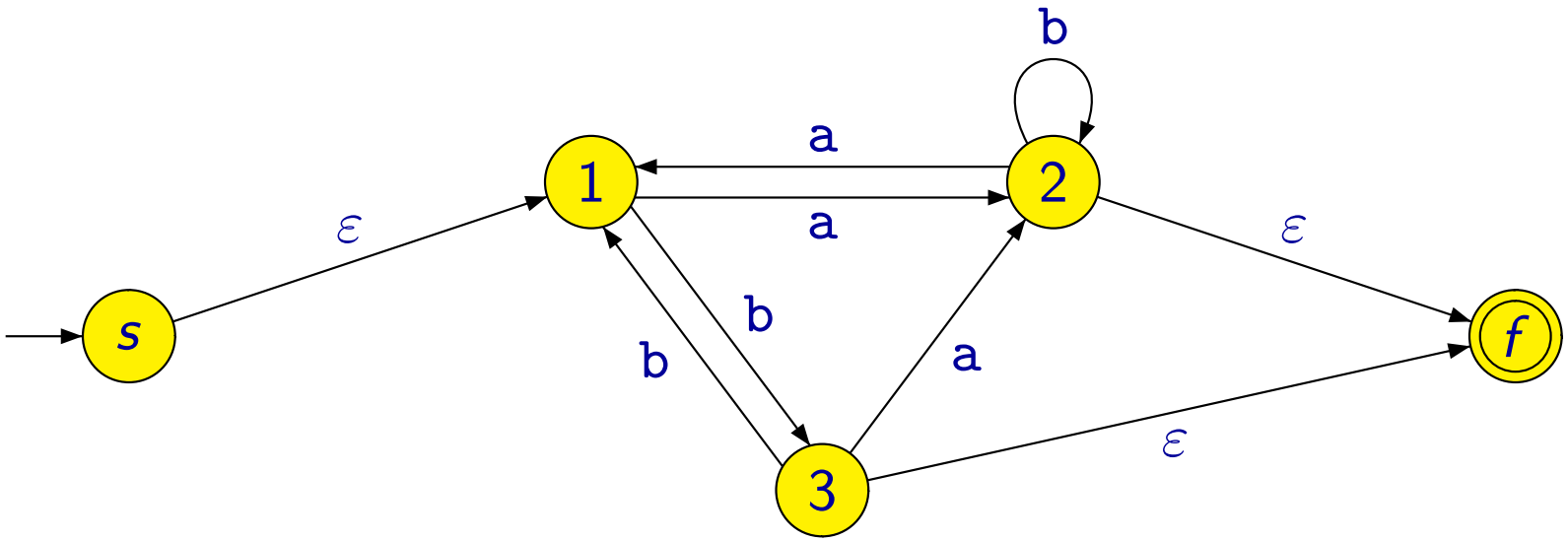
Převod konečného automatu na regulární výraz

Příklad:



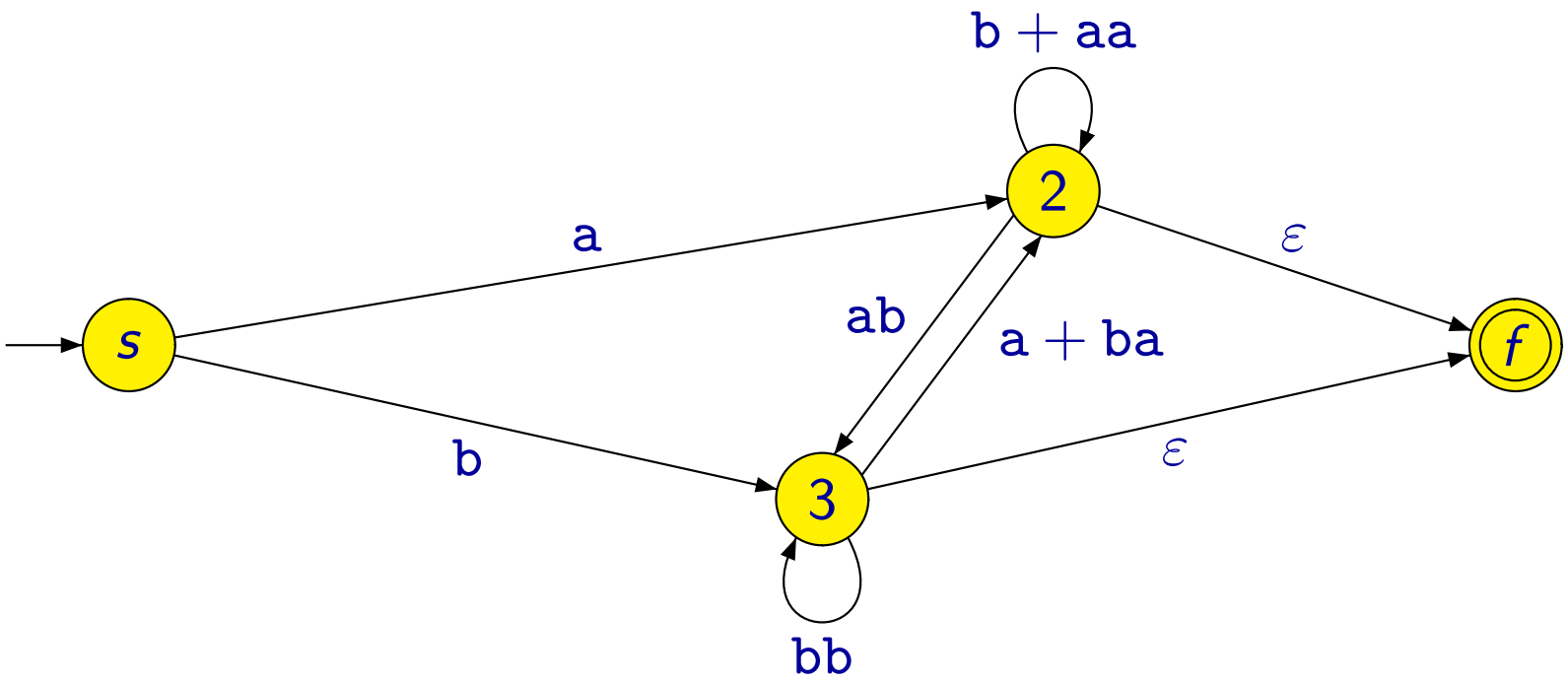
Převod konečného automatu na regulární výraz

Příklad:



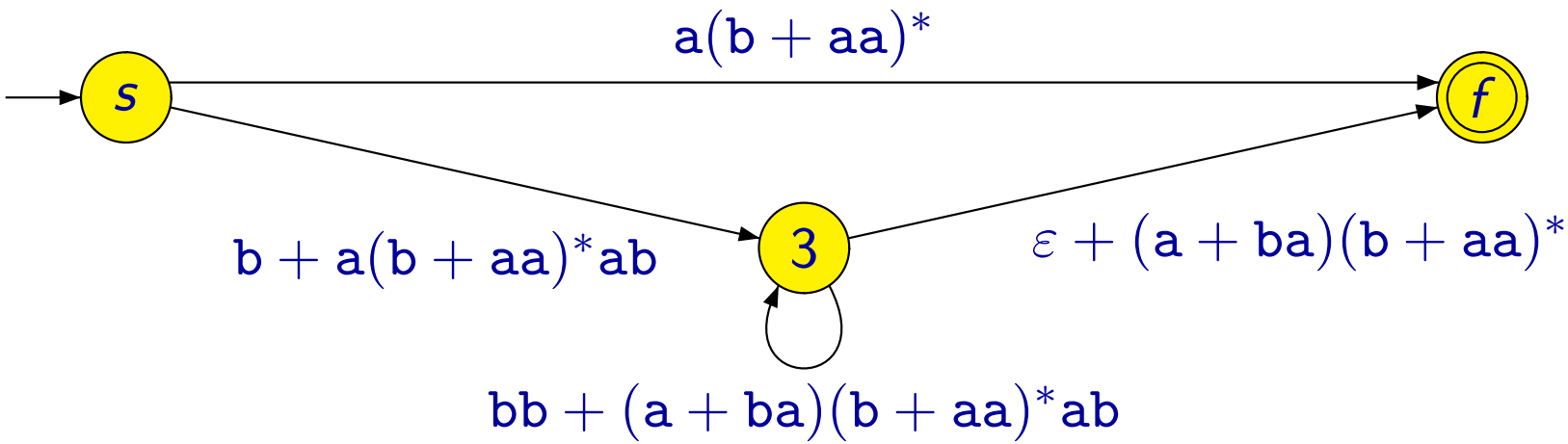
Převod konečného automatu na regulární výraz

Příklad:



Převod konečného automatu na regulární výraz

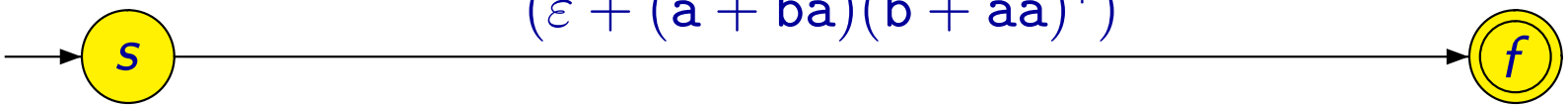
Příklad:



Převod konečného automatu na regulární výraz

Příklad:

$$\mathtt{a}(\mathtt{b}+\mathtt{aa})^*+$$
$$(\mathtt{b}+\mathtt{a}(\mathtt{b}+\mathtt{aa})^*\mathtt{ab})$$
$$(\mathtt{bb}+(\mathtt{a}+\mathtt{ba})(\mathtt{b}+\mathtt{aa})^*\mathtt{ab})^*$$
$$(\varepsilon+(\mathtt{a}+\mathtt{ba})(\mathtt{b}+\mathtt{aa})^*)$$



Ekvivalence konečných automatů a regulárních výrazů

Věta

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

V praxi používané regulární výrazy

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):

$.$	...	zastupuje libovolný znak
$\alpha\beta$	...	zřetězení α a β
$\alpha \mid \beta$	...	sjednocení α a β
$\alpha*$	...	iterace α
$\alpha+$	...	totéž, co $\alpha\alpha^*$
$\alpha?$	...	totéž, co $\alpha + \varepsilon$
$\alpha\{m\}$	...	totéž co m krát α
$\alpha\{m,n\}$	...	α minimálně m krát, maximálně n krát
(α)	...	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

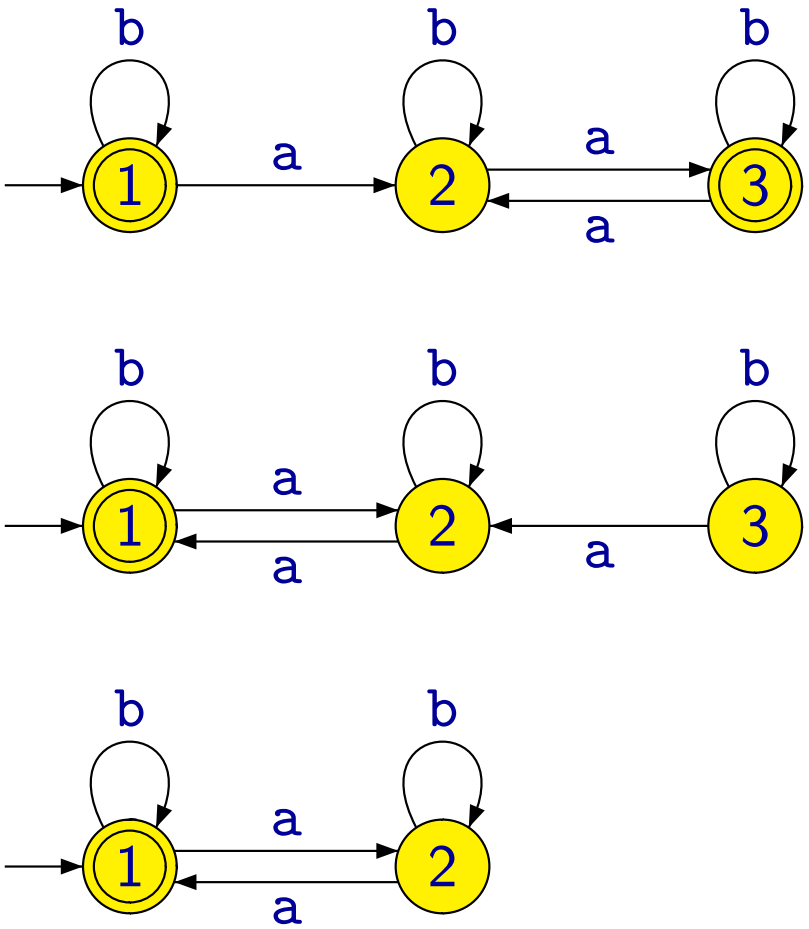
```
^[_a-zA-Z0-9\.\-]+@[_a-zA-Z0-9\.\-]+\.[a-zA-Z]{2,4}$
```

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

`http://www.root.cz/serialy/regularni-vyrazy/`

Minimalizace automatů

Ekvivalence automatů



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

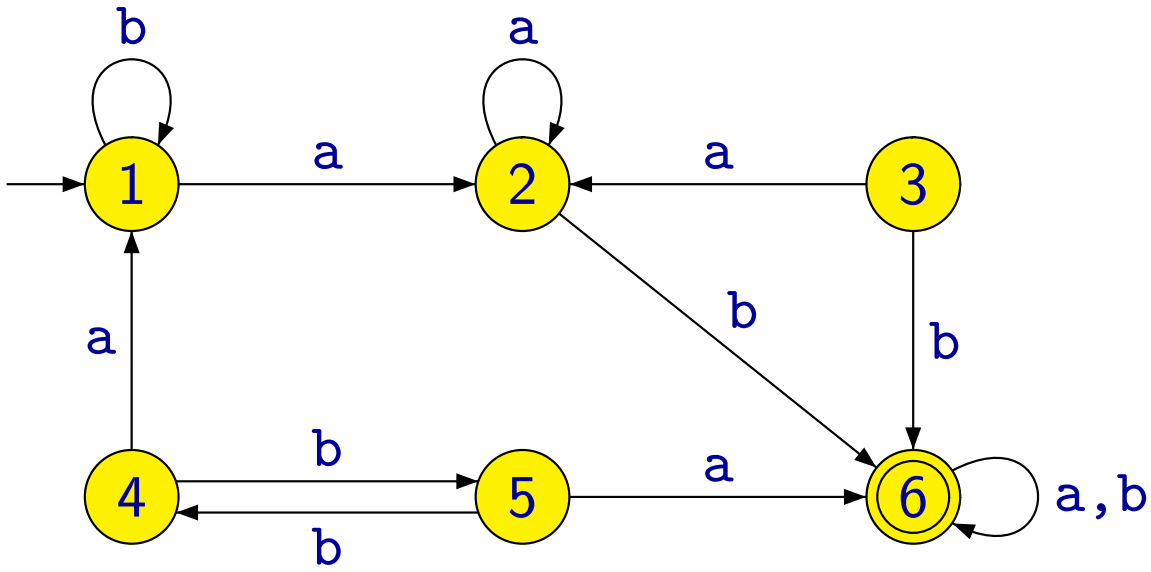
Ekvivalence automatů

Definice

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

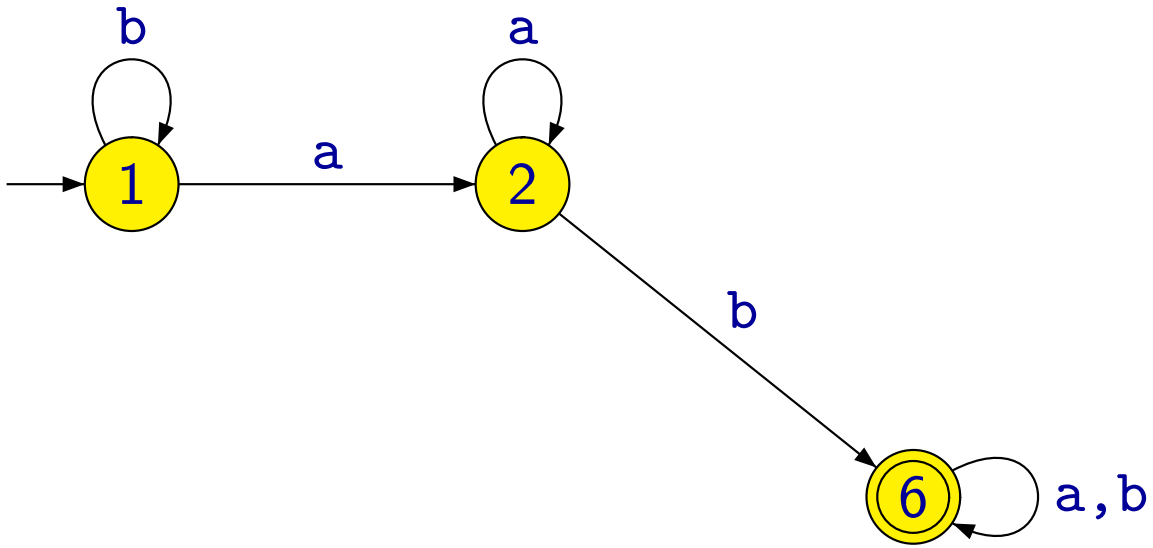
- Existuje nějaký vhodný algoritmus zjišťující, jestli jsou dva automaty ekvivalentní?
- Můžeme zjistit, jestli k danému automatu neexistuje nějaký ekvivalentní menší (s menším počtem stavů)?
- Když o nějakém automatu víme, že menší ekvivalentní už neexistuje, může existovat jiný stejně velký ekvivalentní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

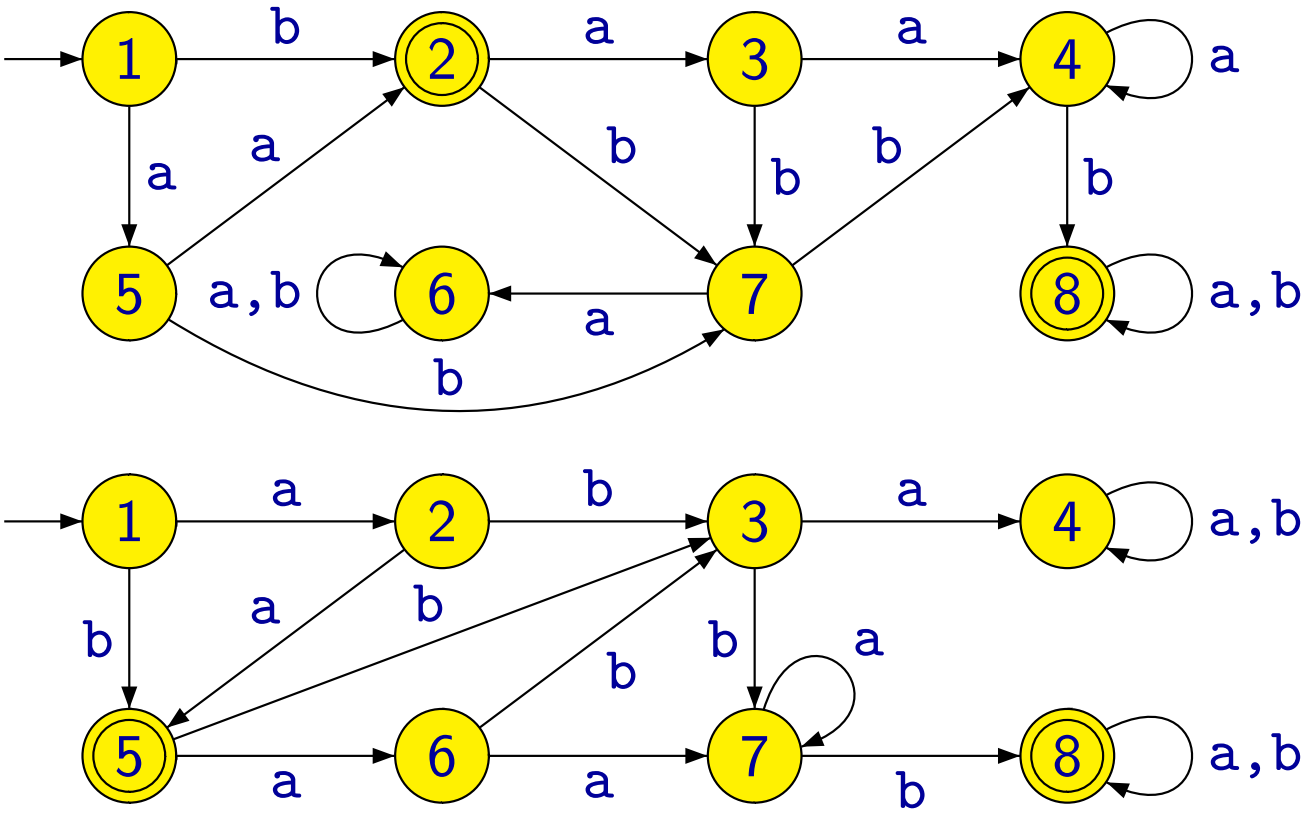
Stav q deterministického konečného automatu $\mathcal{A}$ je **dosažitelný slovem** w, pokud se výpočet $\mathcal{A}$ po přečtení celého slova w zastaví ve stavu q.

Definice

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

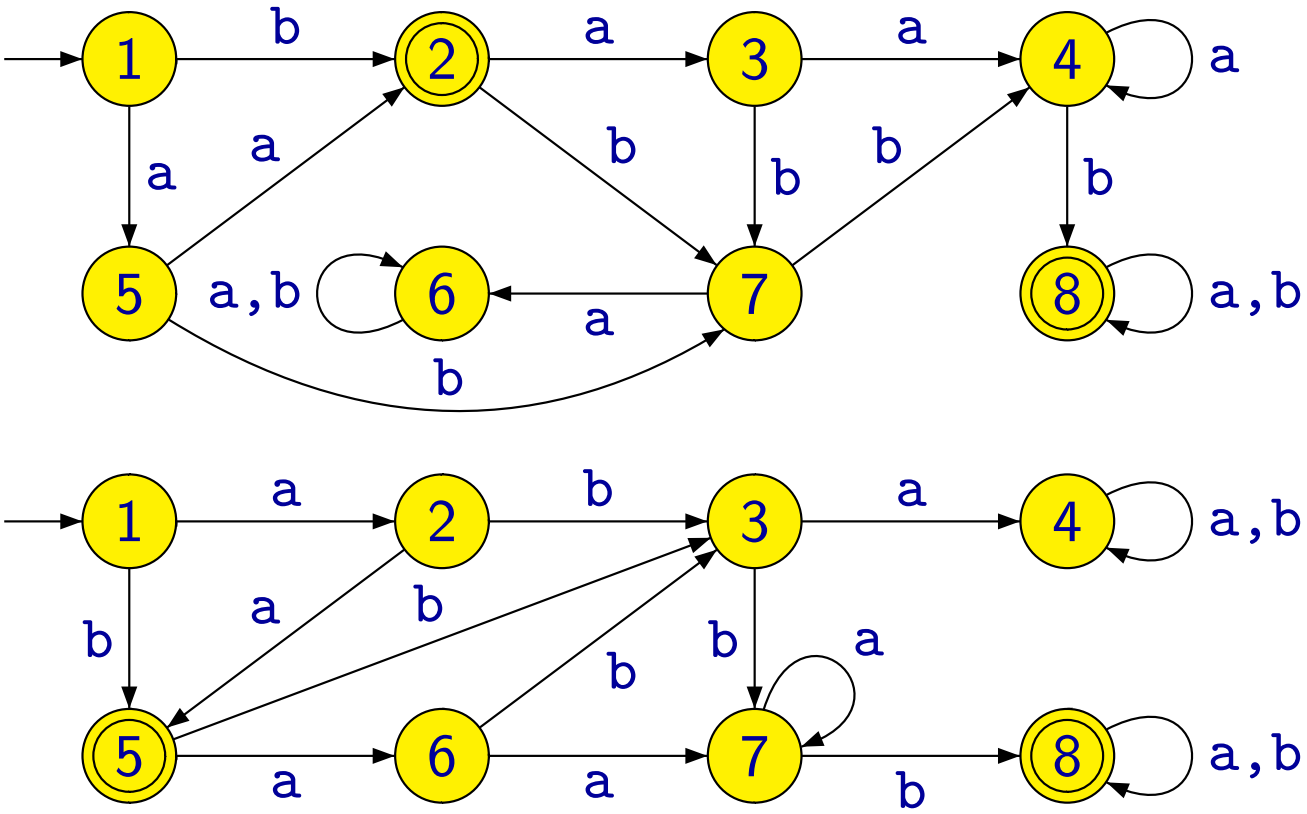
- Do nedosažitelných stavů nevede v grafu automatu žádná orientovaná cesta z počátečního stavu
- Nedosažitelné stavy můžeme z automatu odstranit (spolu se všemi přechody vedoucími z nich). Jazyk přijímaný automatem se nezmění.

Normovaný tvar automatu



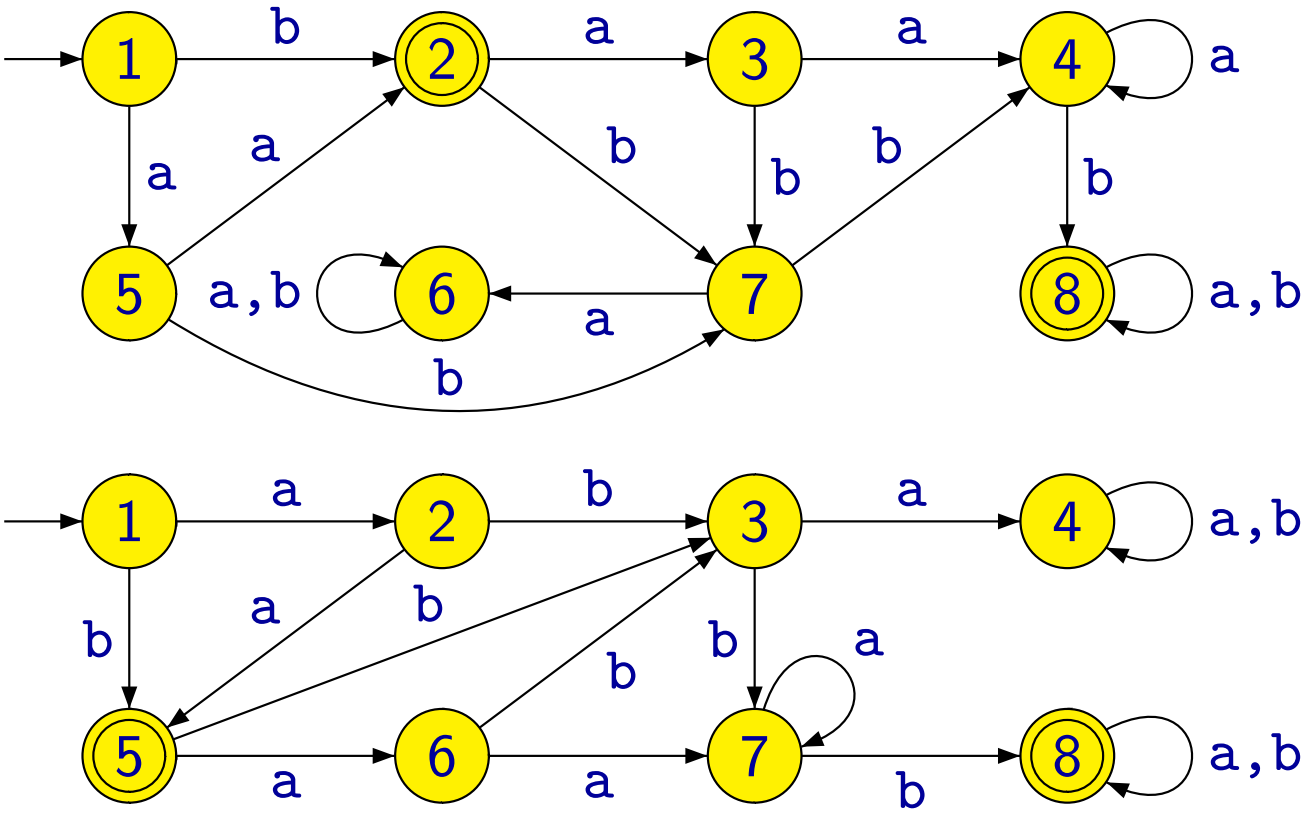
$\mathcal{A}_1$	a	b
$\rightarrow 1$	5	2
$\leftarrow 2$	3	7
3	4	7
4	4	8
5	2	7
6	6	6
7	6	4
$\leftarrow 8$	8	8

Normovaný tvar automatu



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

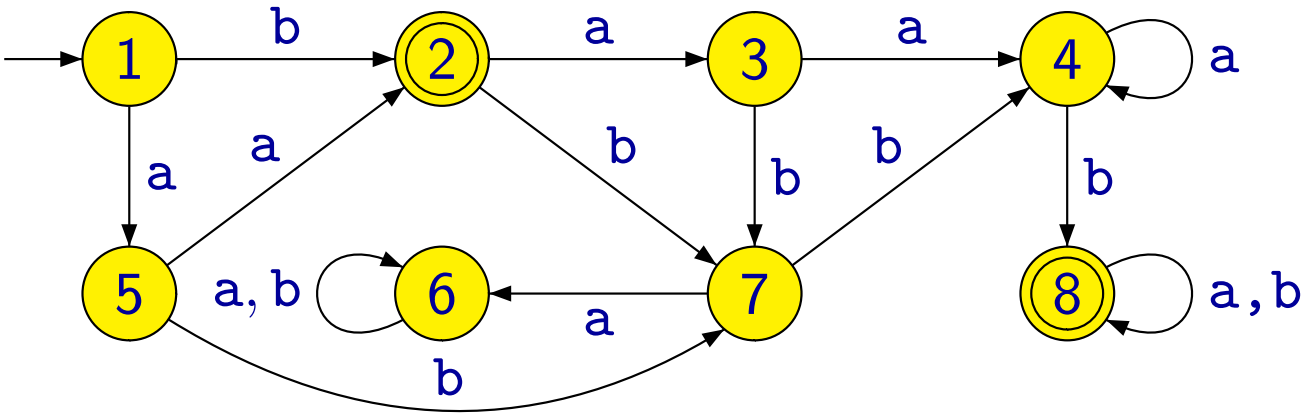
Normovaný tvar automatu



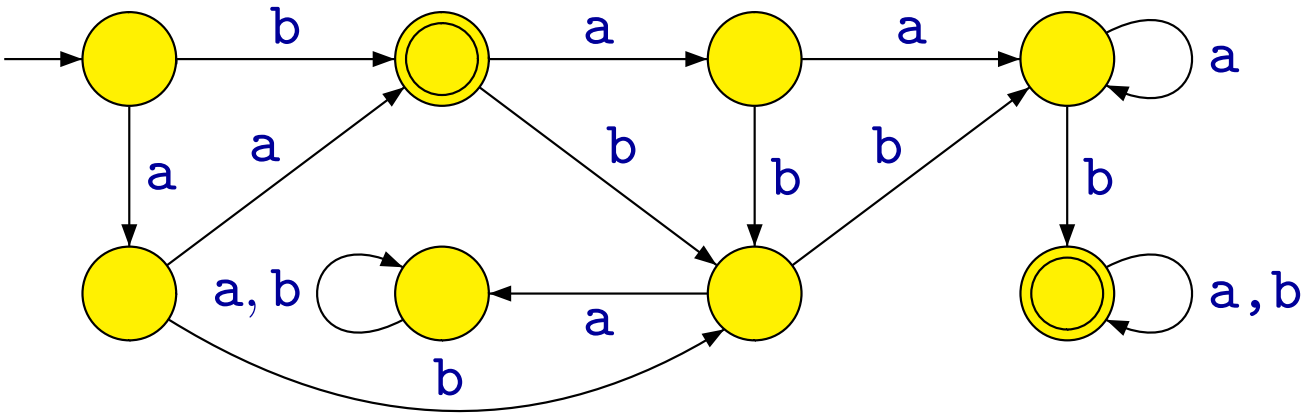
$\mathcal{A}_2$	a	b
$\rightarrow 1$	2	5
2	5	3
3	4	7
4	4	4
$\leftarrow 5$	6	3
6	7	3
7	7	8
$\leftarrow 8$	8	8

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

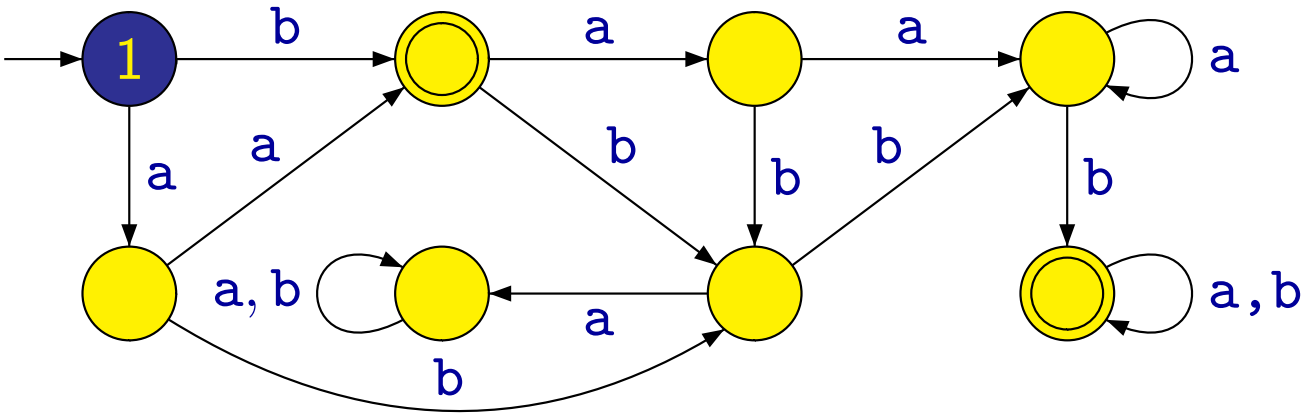
Normovaný tvar automatu



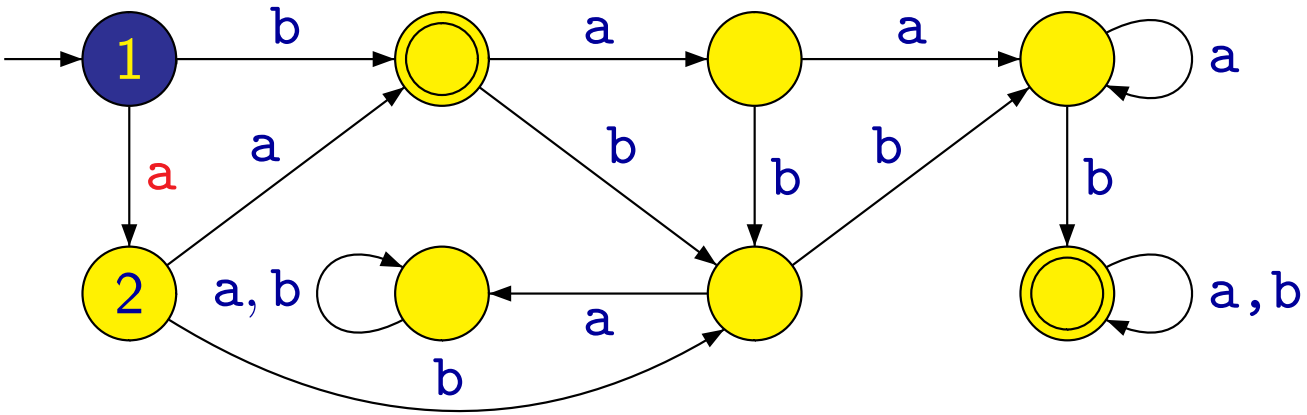
Normovaný tvar automatu



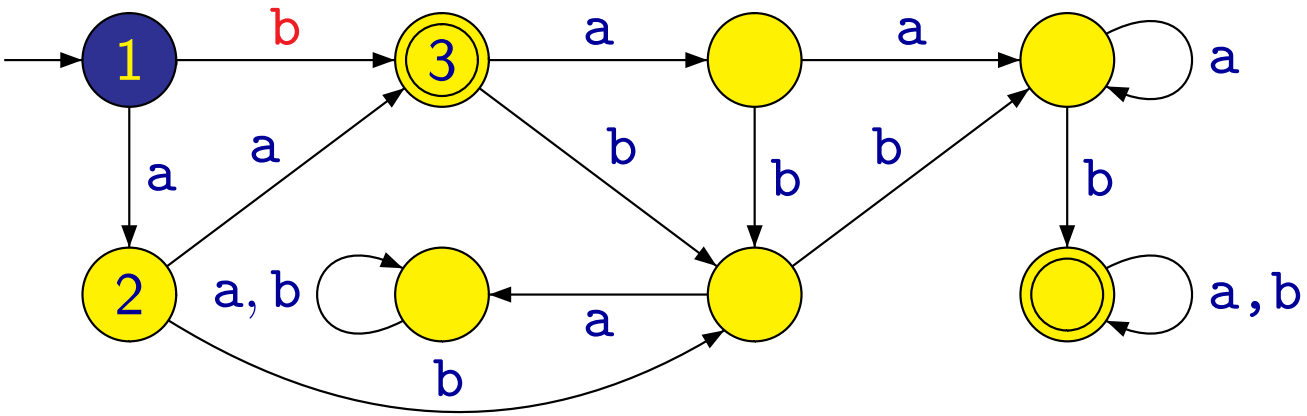
Normovaný tvar automatu



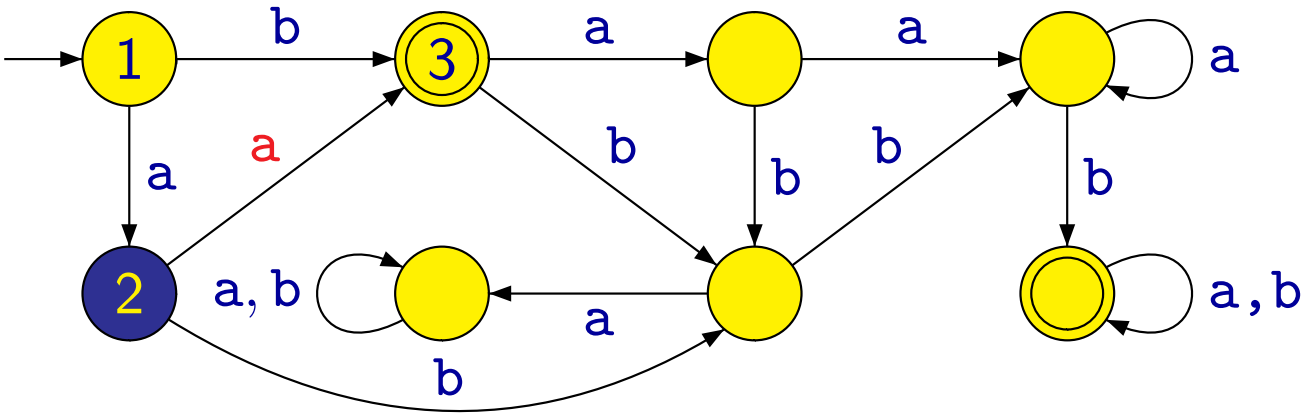
Normovaný tvar automatu



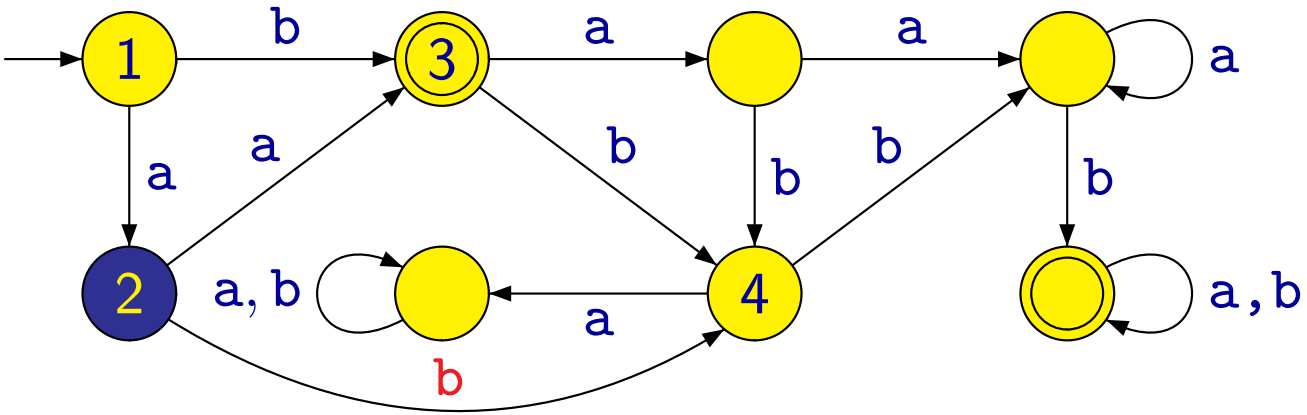
Normovaný tvar automatu



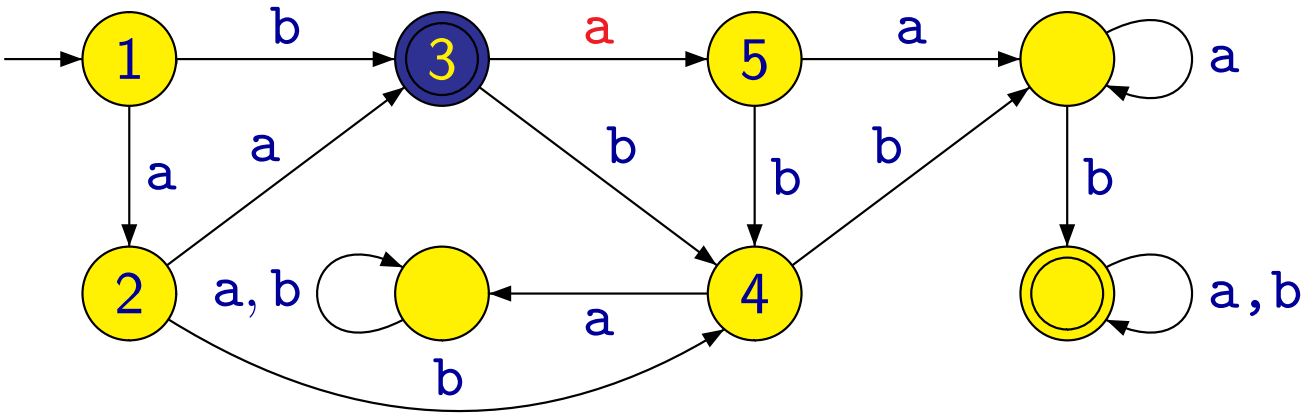
Normovaný tvar automatu



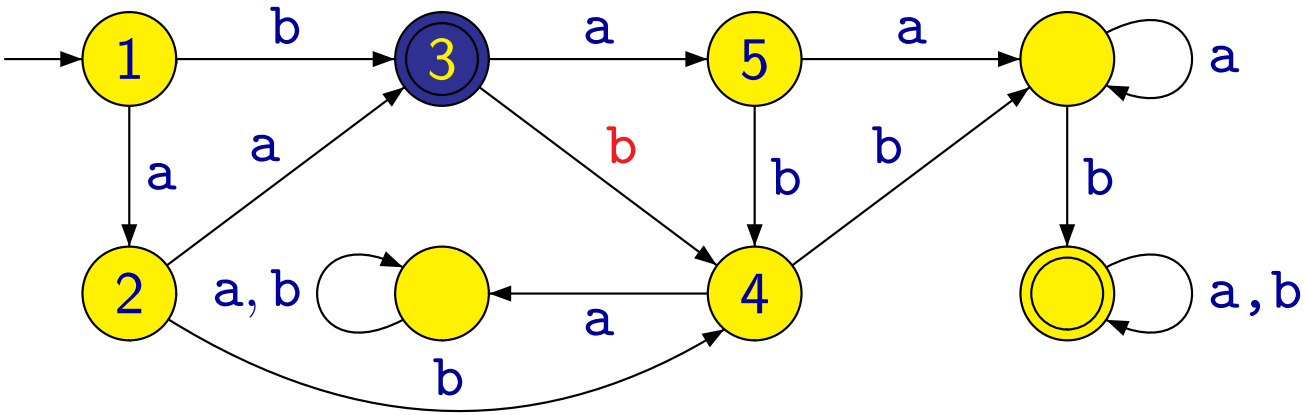
Normovaný tvar automatu



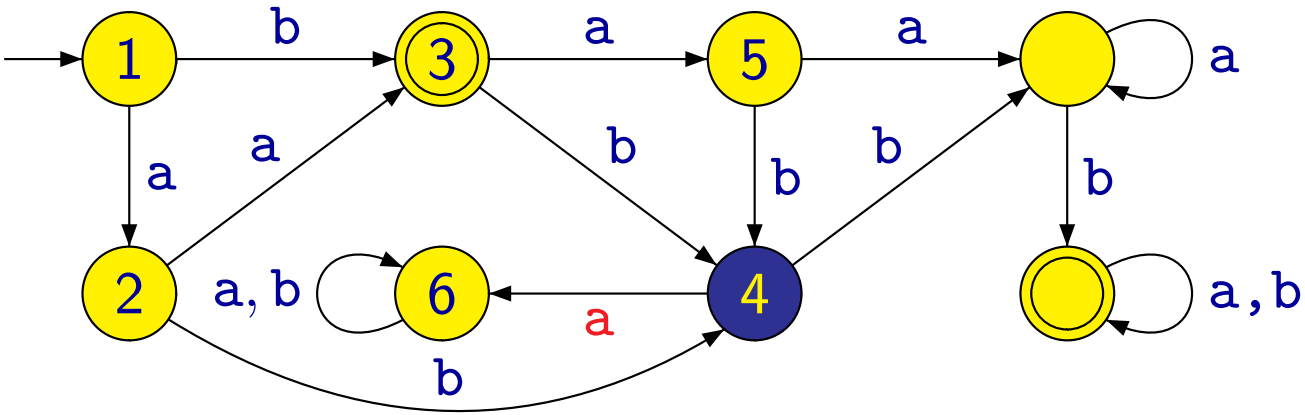
Normovaný tvar automatu



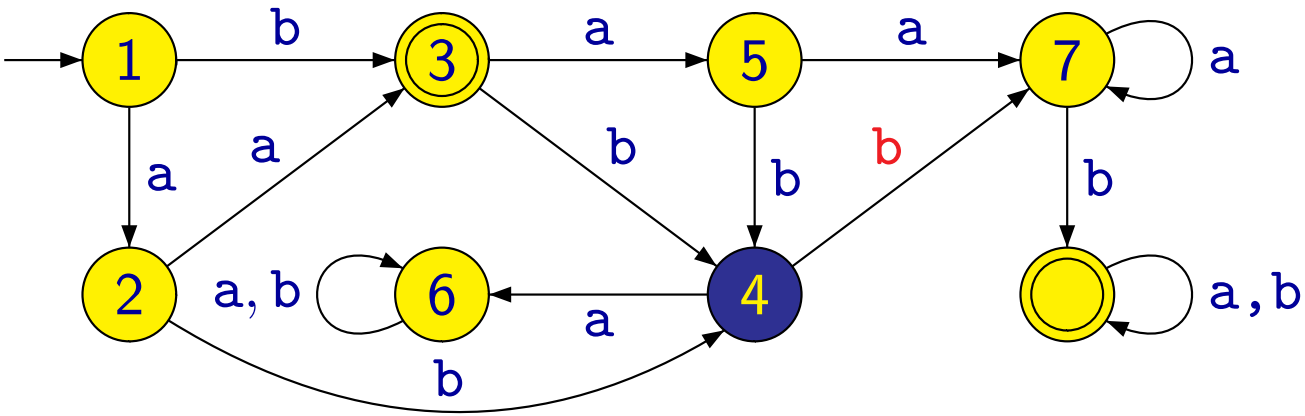
Normovaný tvar automatu



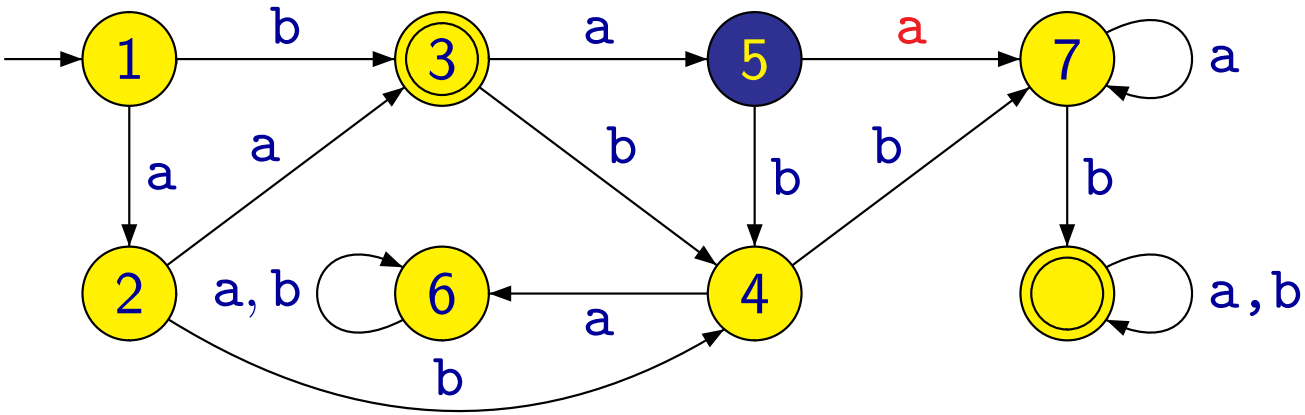
Normovaný tvar automatu



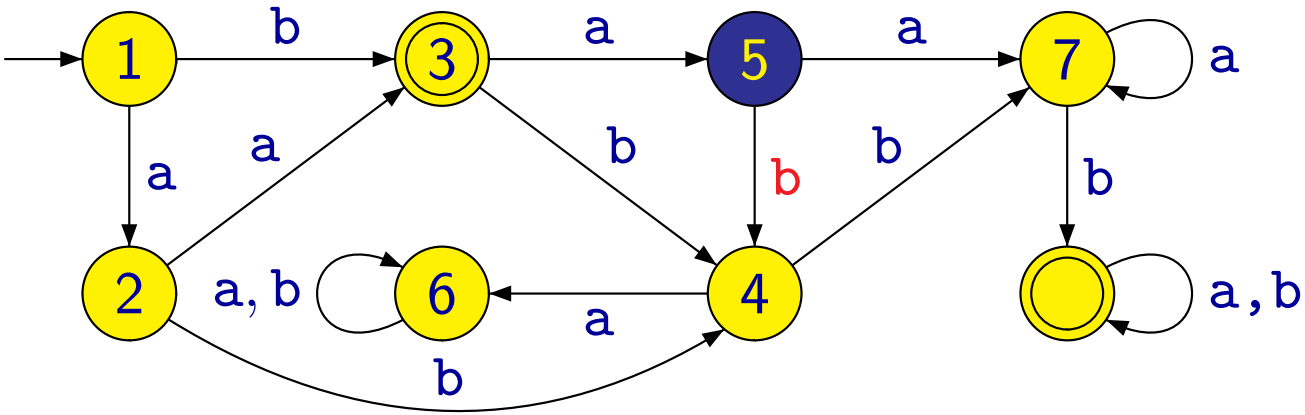
Normovaný tvar automatu



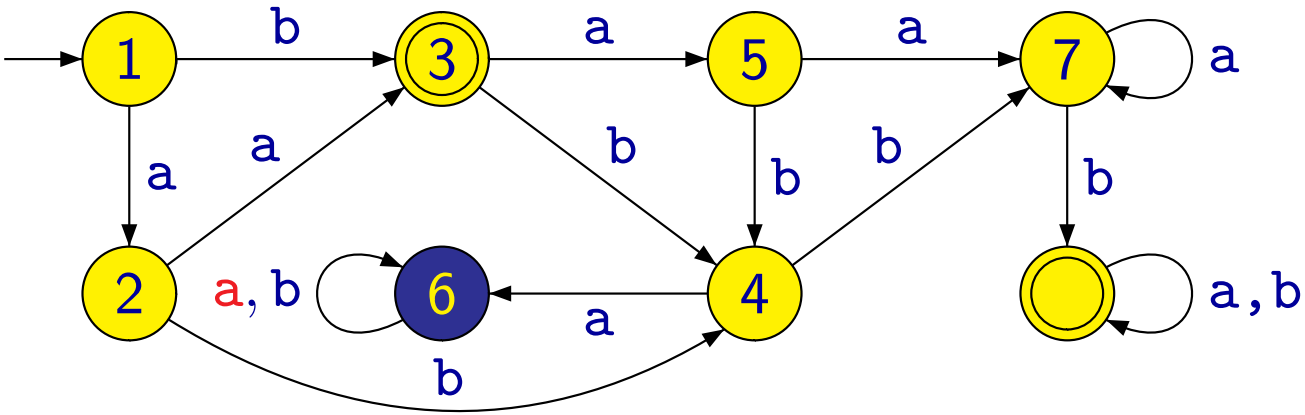
Normovaný tvar automatu



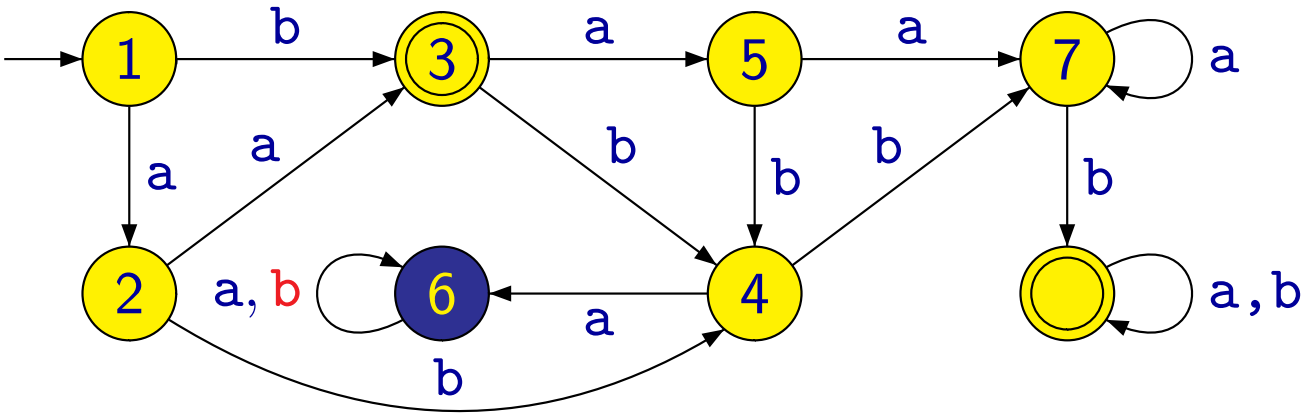
Normovaný tvar automatu



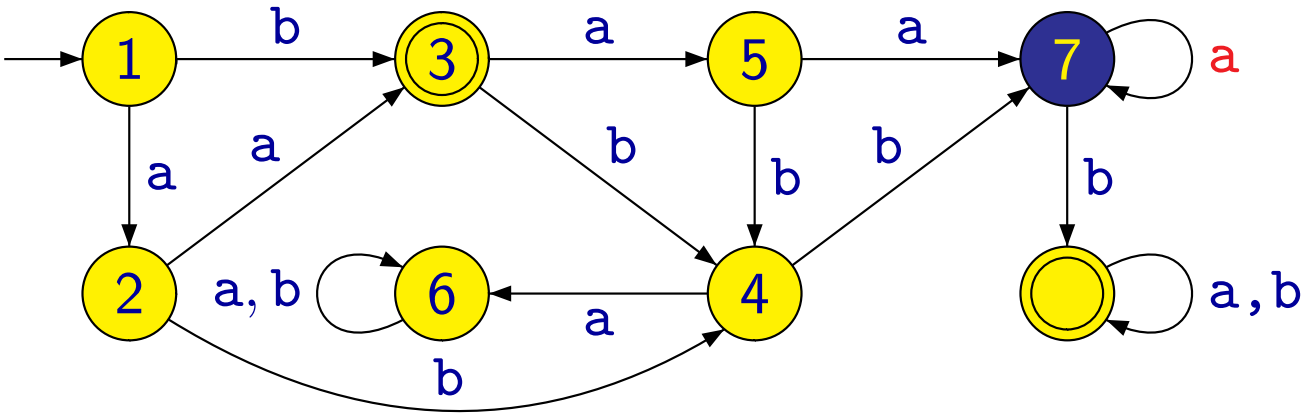
Normovaný tvar automatu



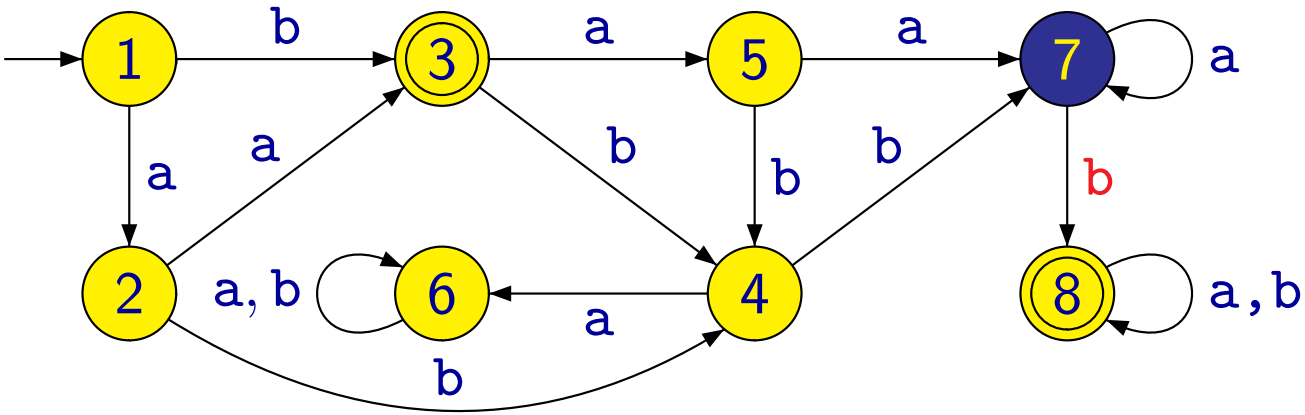
Normovaný tvar automatu



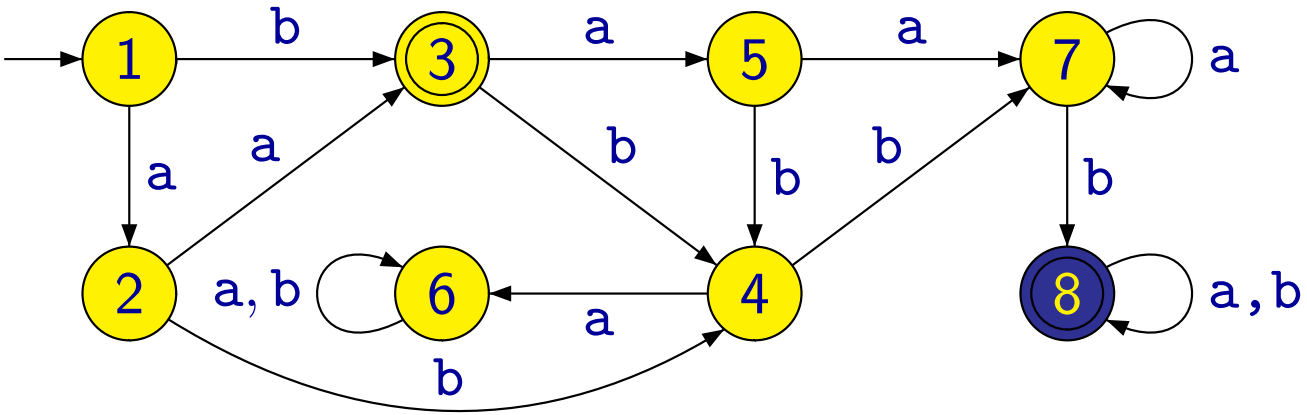
Normovaný tvar automatu



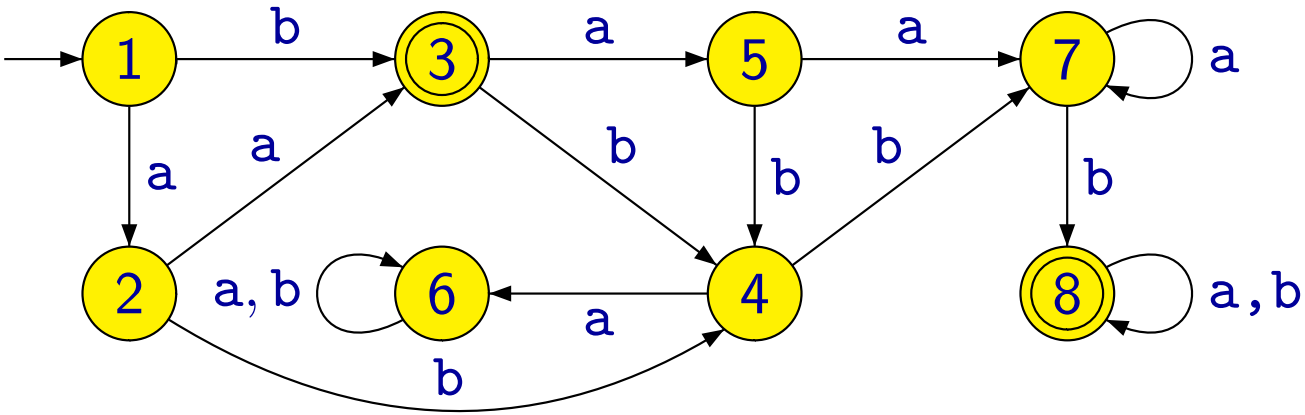
Normovaný tvar automatu



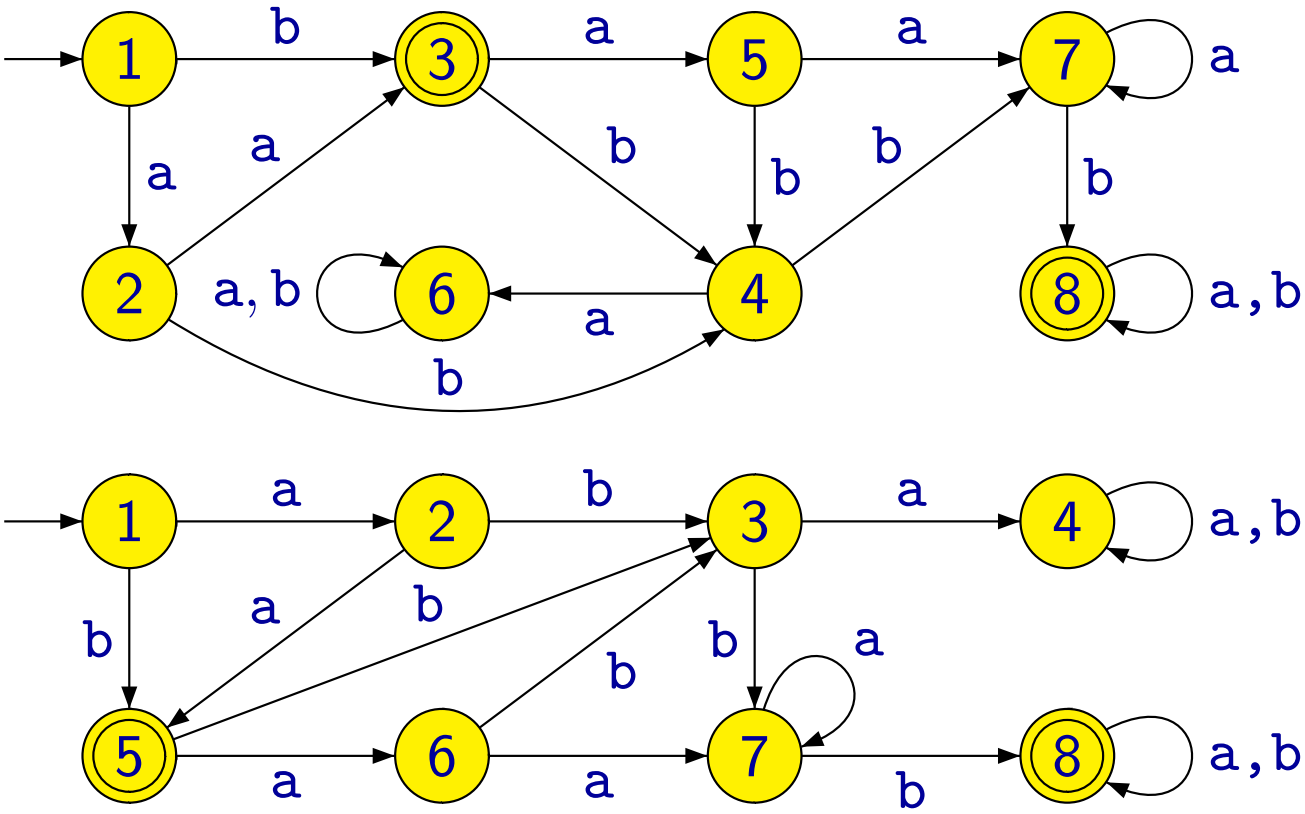
Normovaný tvar automatu



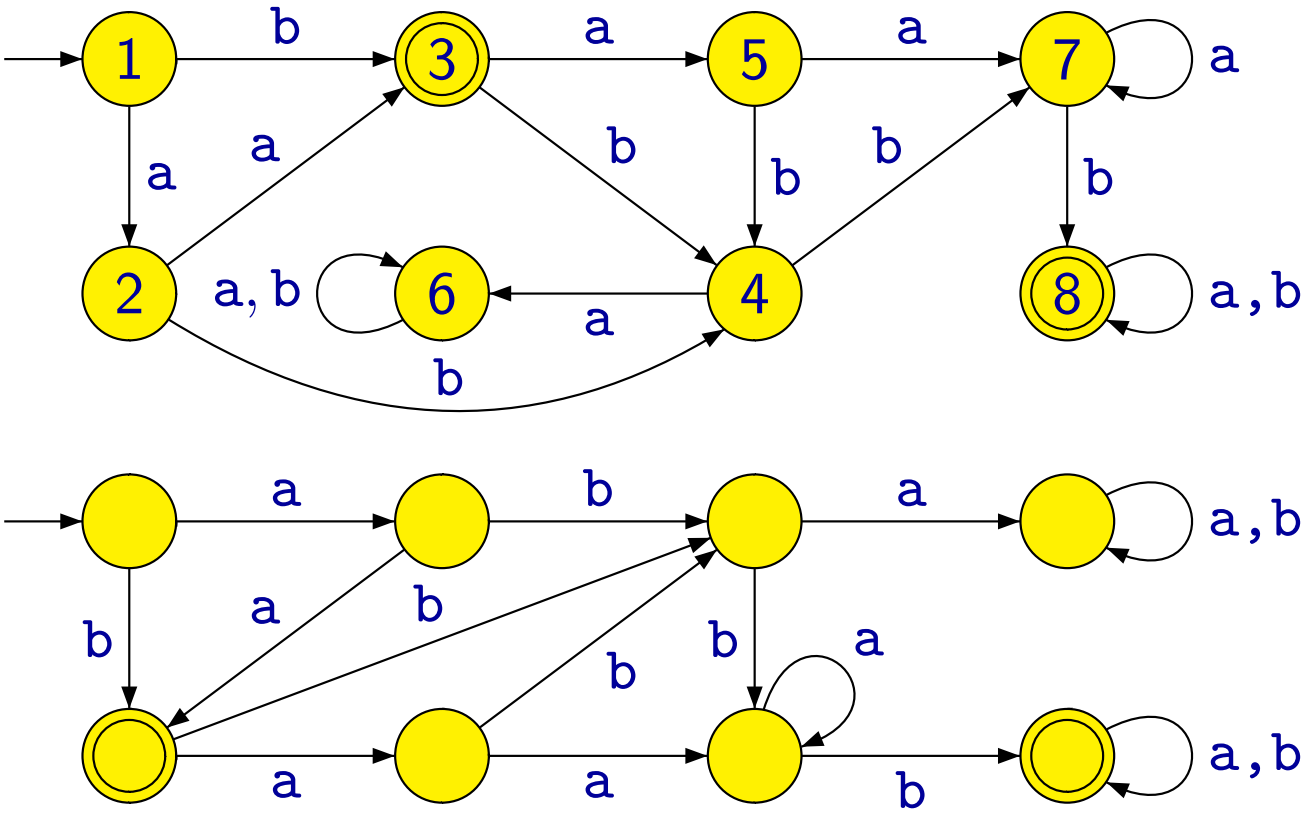
Normovaný tvar automatu



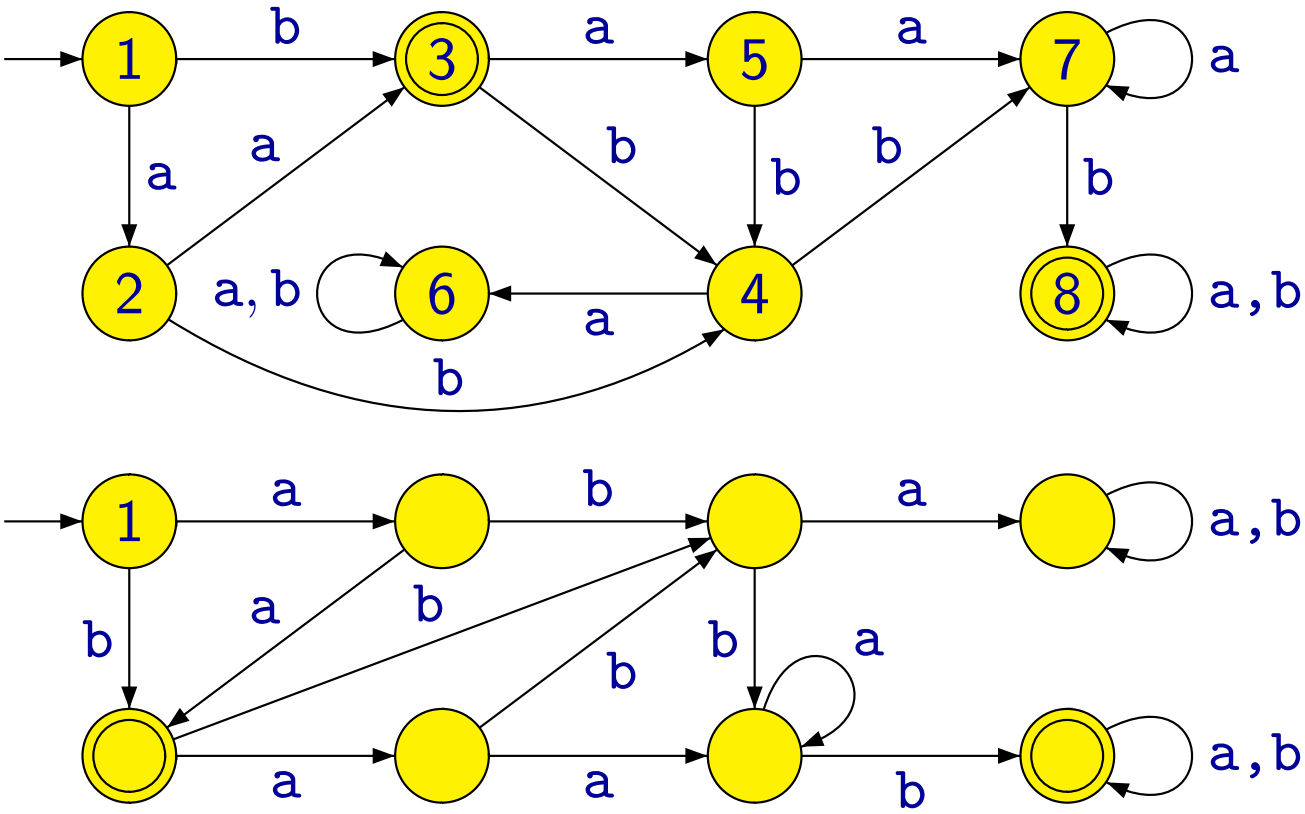
Normovaný tvar automatu



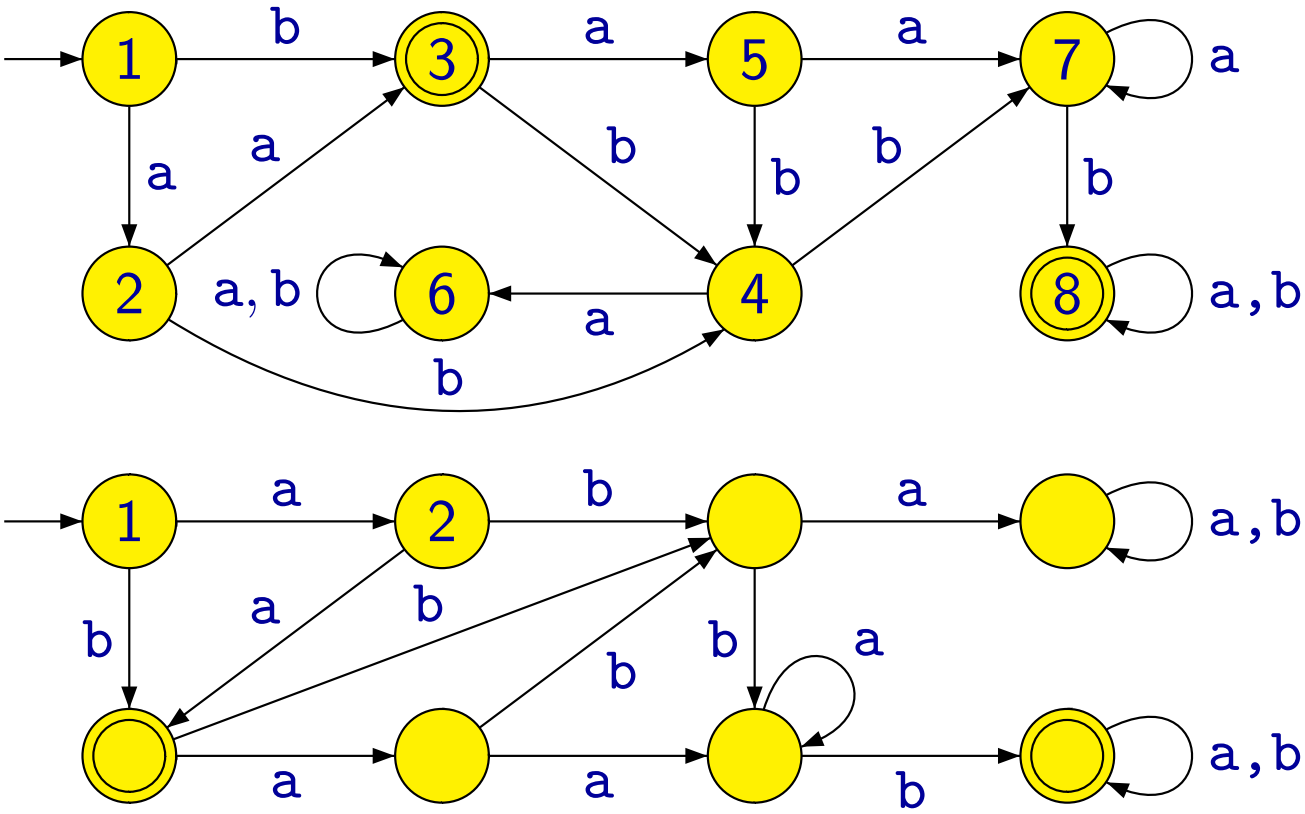
Normovaný tvar automatu



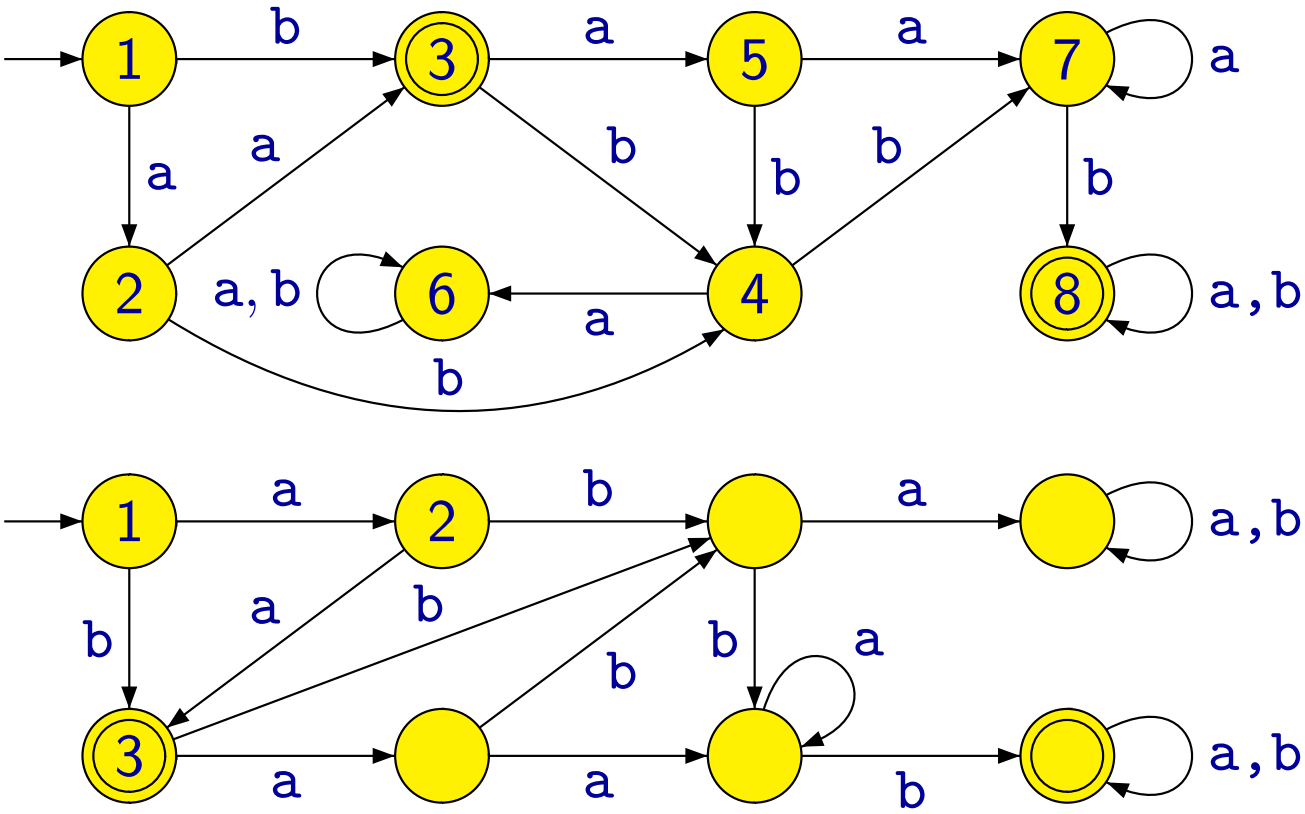
Normovaný tvar automatu



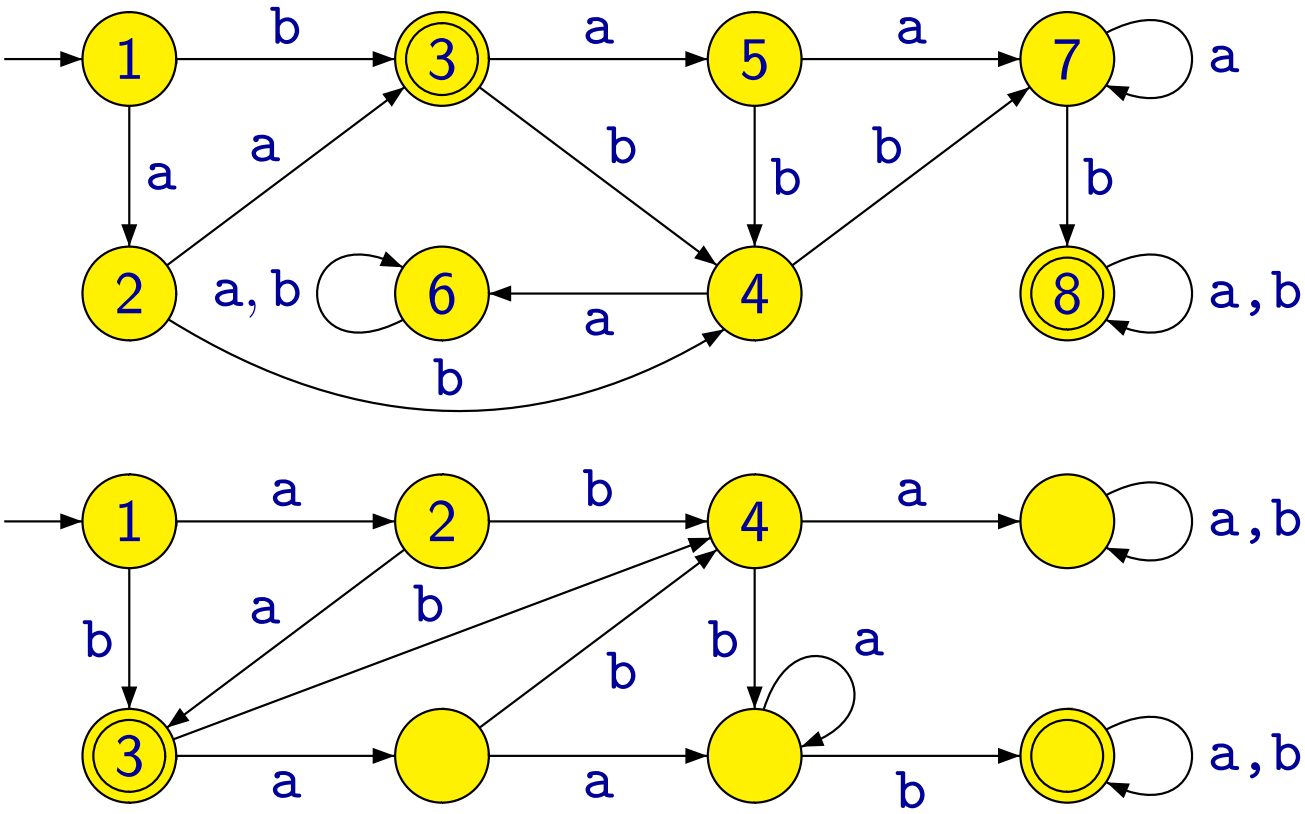
Normovaný tvar automatu



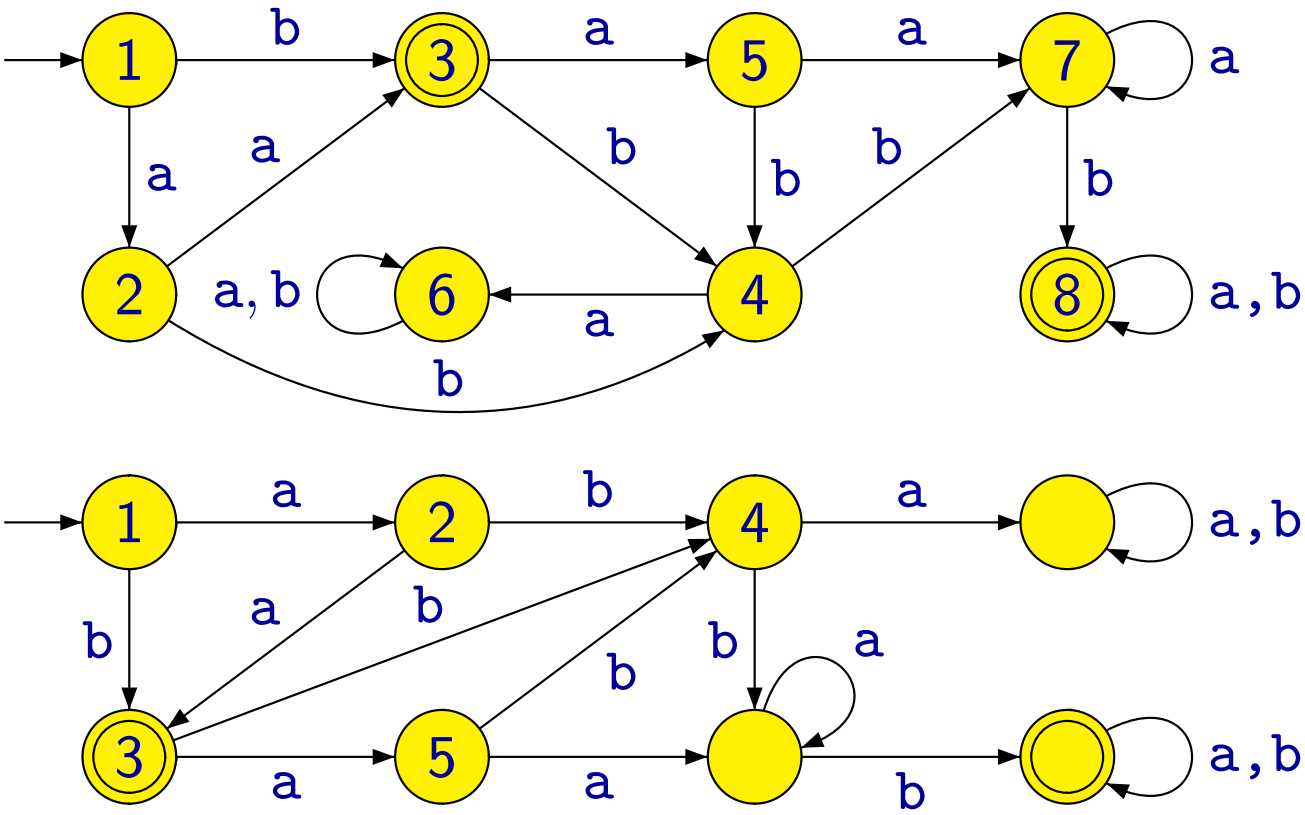
Normovaný tvar automatu



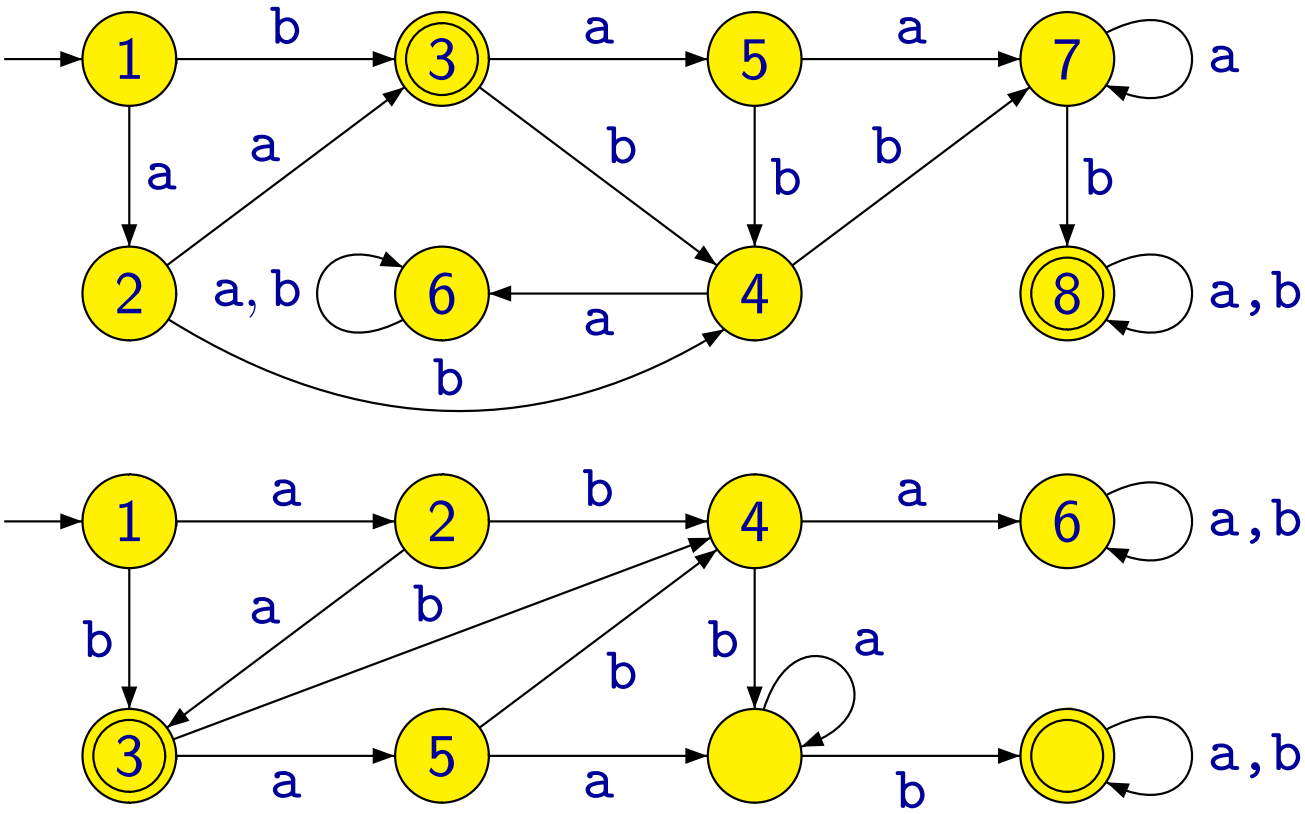
Normovaný tvar automatu



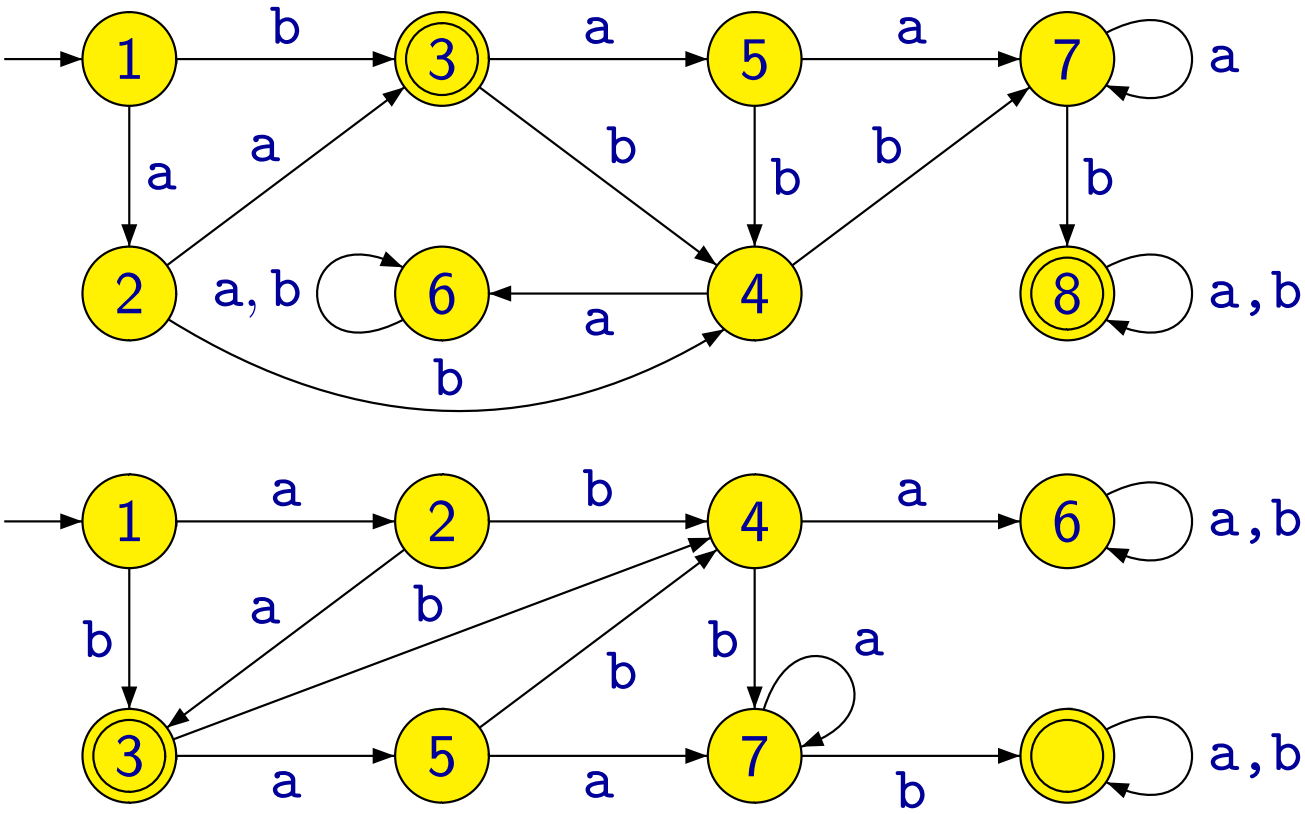
Normovaný tvar automatu



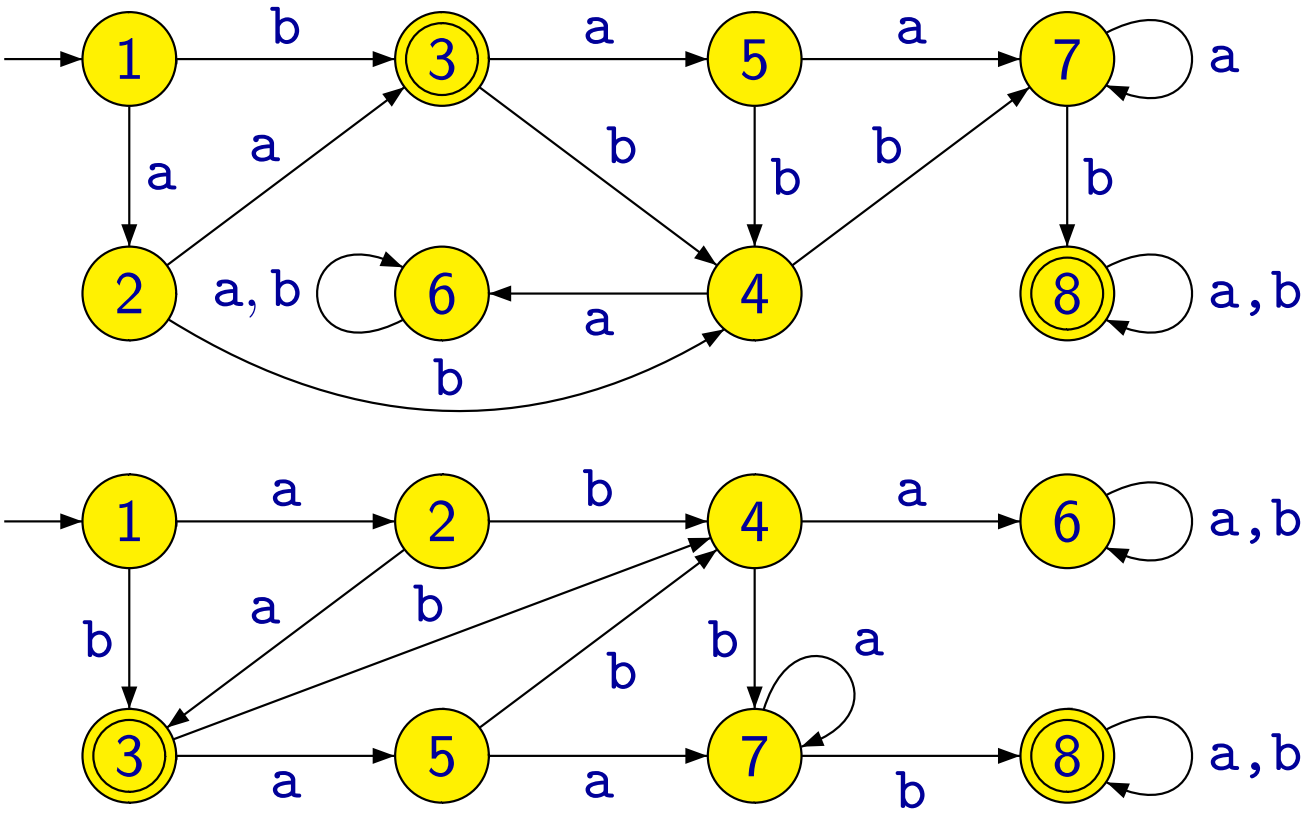
Normovaný tvar automatu



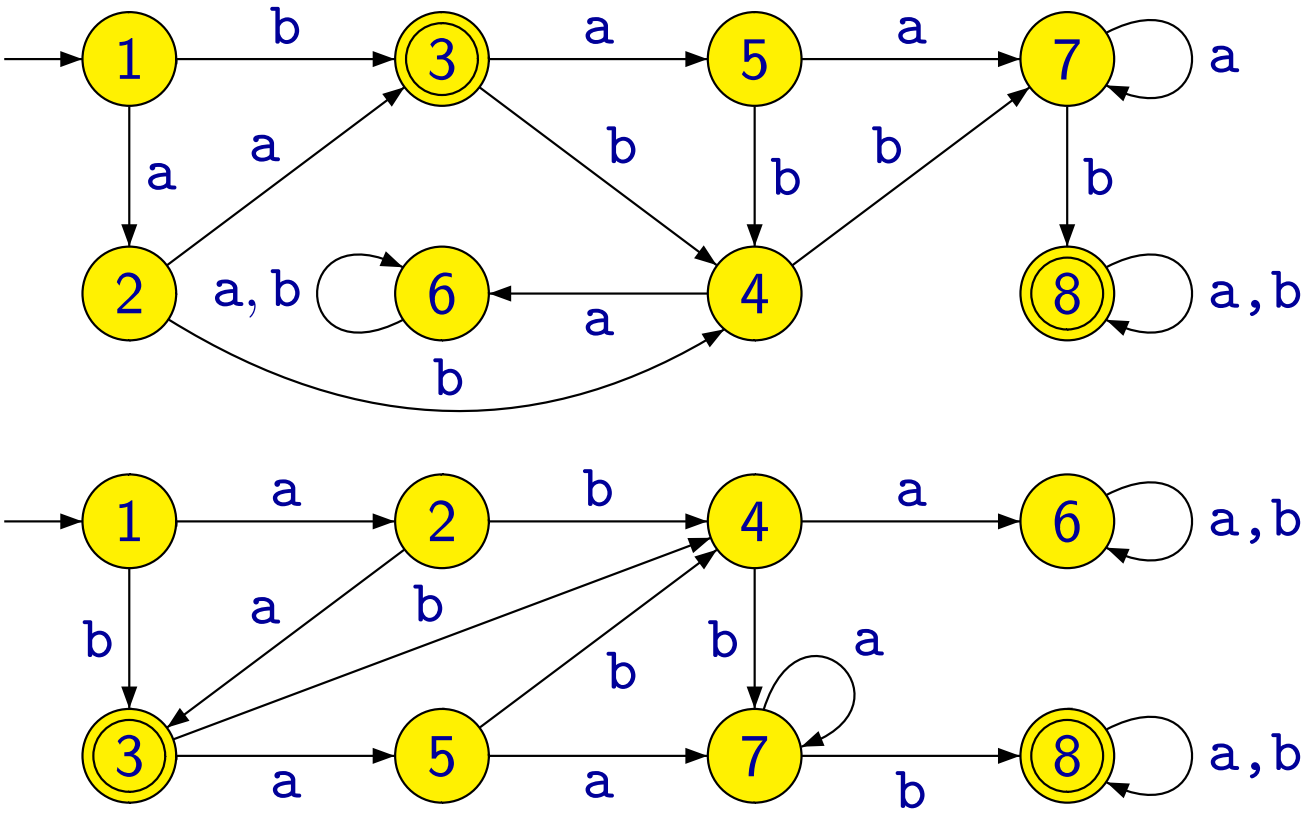
Normovaný tvar automatu



Normovaný tvar automatu

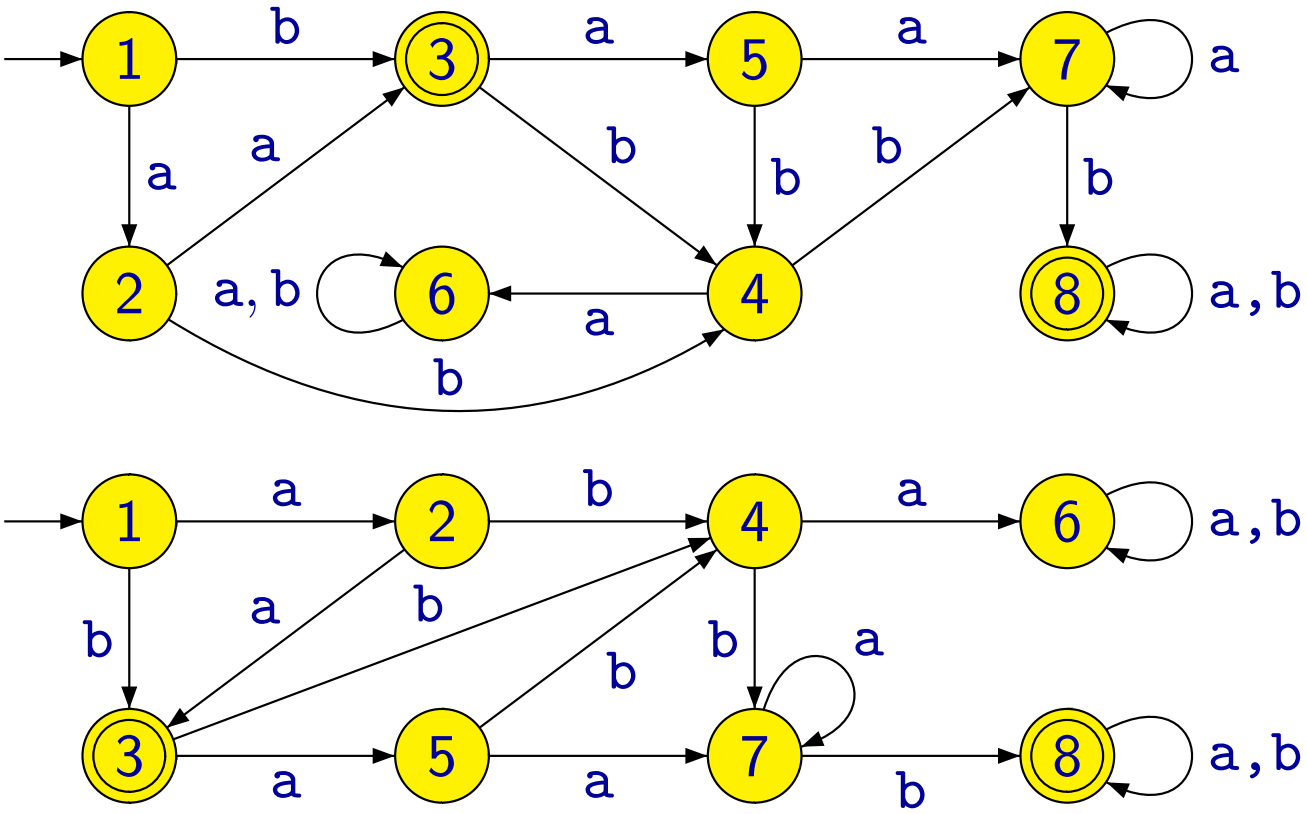


Normovaný tvar automatu



	a	b
$\rightarrow$ 1	2	3
2	3	4
$\leftarrow$ 3	5	4
4	6	7
5	7	4
6	6	6
7	7	8
$\leftarrow$ 8	8	8

Normovaný tvar automatu



	a	b
$\rightarrow 1$	2	3
2	3	4
$\leftarrow 3$	5	4
4	6	7
5	7	4
6	6	6
7	7	8
$\leftarrow 8$	8	8

- 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

Normovaný tvar automatu

Definice

Mějme abecedu Σ spolu s úplný ostrým uspořádáním $<$ nad znaky abecedy. Pro slova nad abecedou Σ 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 $u <_L v$

Normovaný tvar automatu

Definice

Mějme abecedu Σ spolu s úplný ostrým uspořádáním $<$ nad znaky abecedy. Pro slova nad abecedou Σ 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 $\varepsilon <_L a <_L b <_L aa <_L bb <_L aba <_L baa <_L aaaa <_L bbbbb$.

Normovaný tvar automatu

Definice

Mějme abecedu Σ spolu s úplný ostrým uspořádáním $<$ nad znaky abecedy. Pro slova nad abecedou Σ 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 $\varepsilon <_L 1 <_L 3 <_L 31 <_L 33 <_L 111 <_L 321$.

Normovaný tvar automatu

Definice

Mějme abecedu Σ spolu s úplný ostrým uspořádáním $<$ nad znaky abecedy. Pro slova nad abecedou Σ 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 $\varepsilon <_L 1 <_L 2 <_L 01 <_L 03 <_L 111 <_L 0021$.

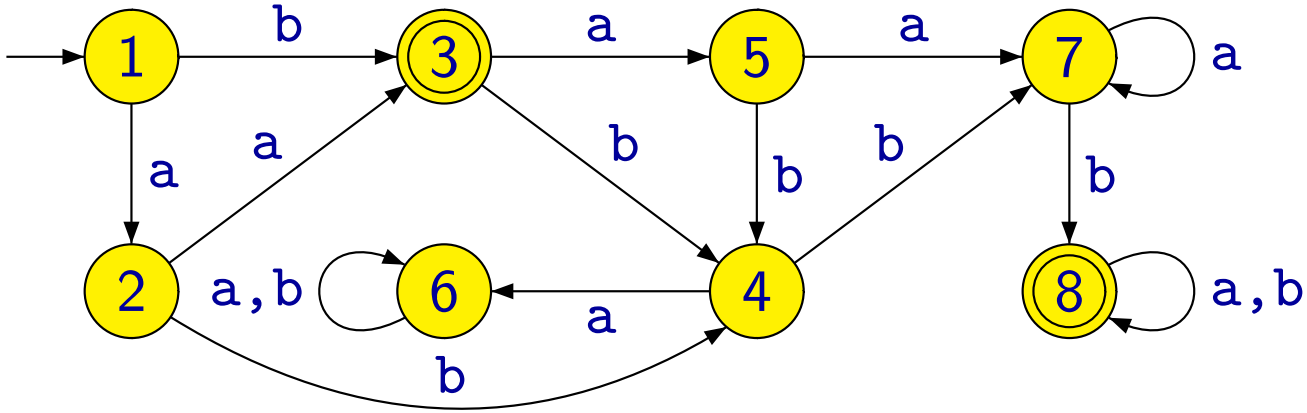
Normovaný tvar automatu

Definice

Mějme deterministický konečný automat $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ bez nedosažitelných stavů a uspořádání prvků abecedy Σ (které indukuje uspořádání $<_L$ na Σ^*).
Označme pro každý stav $i \in \{1, 2, \ldots, n\}$ 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

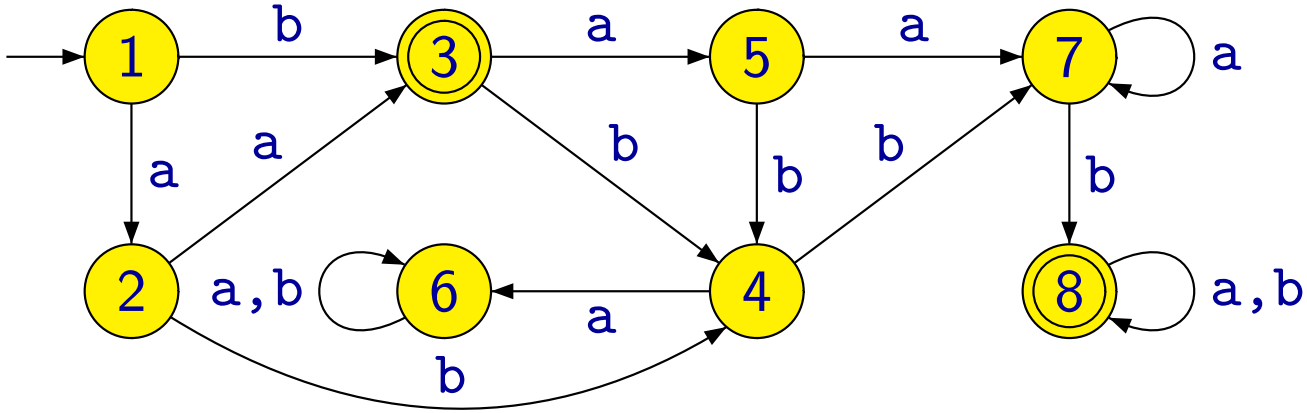
- $Q = \{1, 2, 3, \ldots, n\}$ pro nějaké $n \geq 1$
- 1 je počáteční stav
- Pro každou dvojici stavů $i, j \in \{1, 2, \ldots, n\}$ takovou, že $i < j$, platí $u_i <_L u_j$.

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

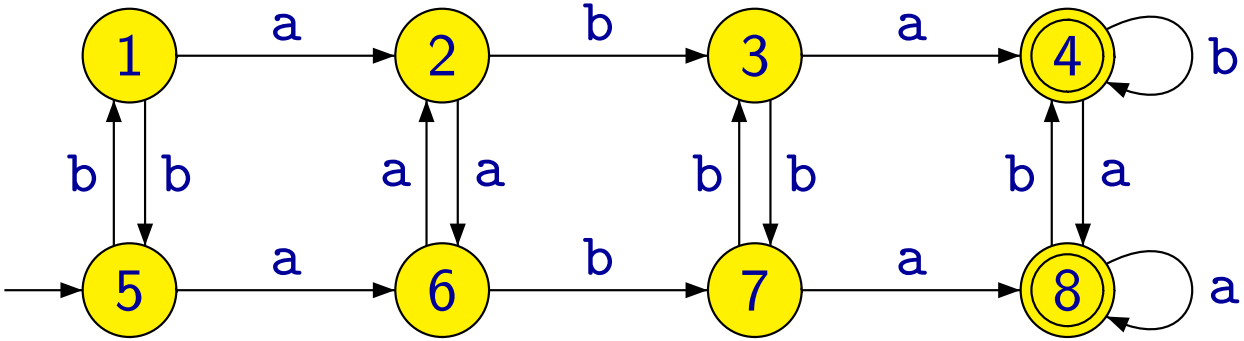
Normovaný tvar automatu



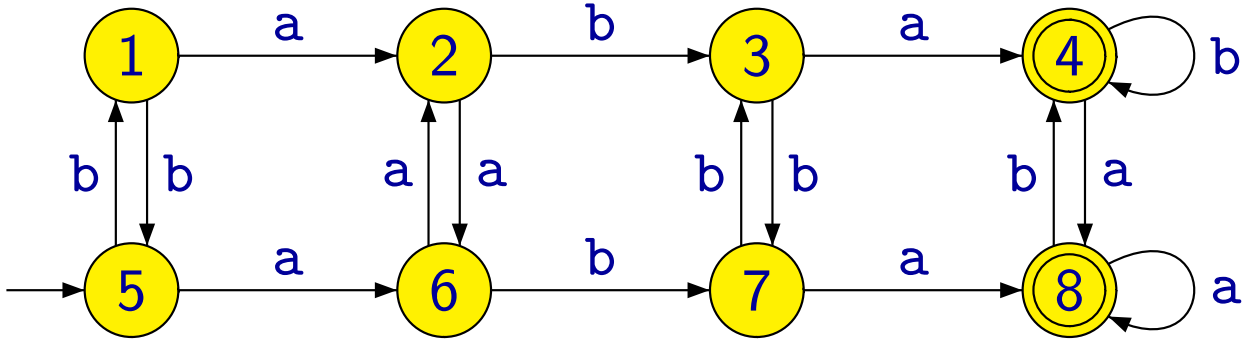
- Do stavu 4 se dostaneme slovy ab, bb, aab, bab tedy $u_4 = ab$
- Do stavu 5 se dostaneme slovy ba, aaa tedy $u_5 = ba$
- $ab <_L ba$ proto jsou stavy $4, 5$ označeny v tomto pořadí
- Pro ostatní stavy jsou nejmenší slova následující:

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

Minimalizace konečného automatu

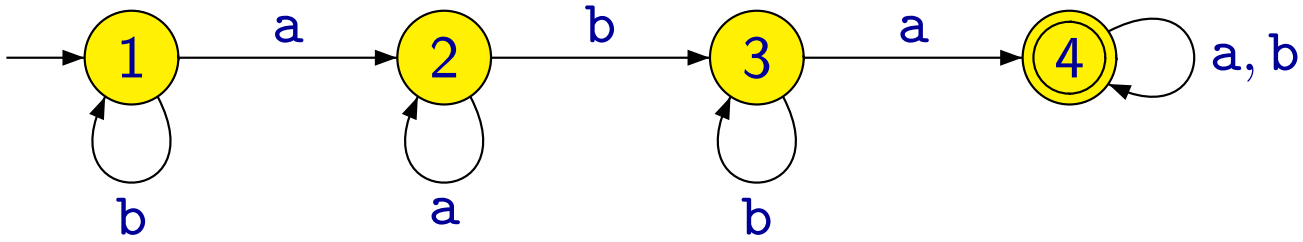


Minimalizace konečného automatu



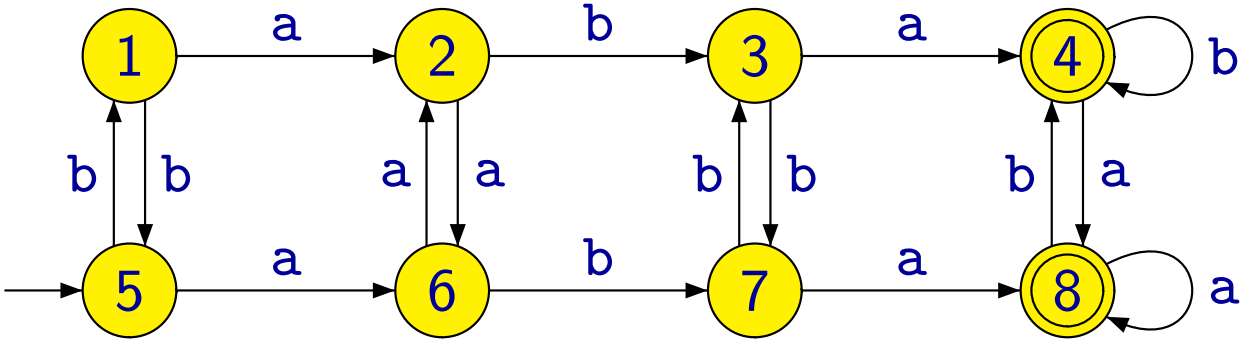
- Automat přijímá taková slova, z nichž by vypuštěním některých znaků zůstalo *aba*
- Existuje automat s méně stavy, který přijímá stejný jazyk?

Minimalizace konečného automatu



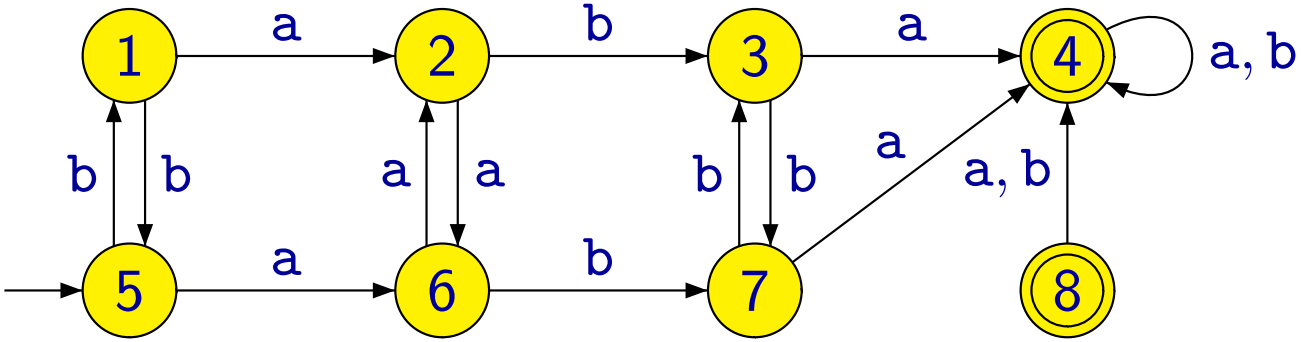
- Automat přijímá taková slova, z nichž by vypuštěním některých znaků zůstalo *aba*
- Existuje automat s méně stavy, který přijímá stejný jazyk?
- Existuje nějaký algoritmus, který by našel ekvivalentní automat s nejmenším možným počtem stavů?

Minimalizace konečného automatu



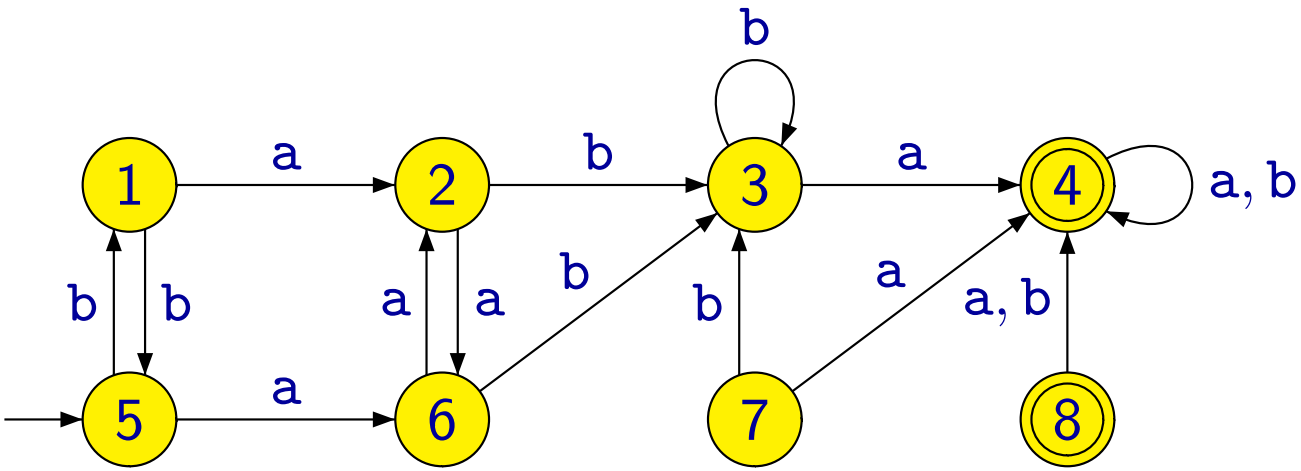
- Když se automat při běhu dostane do stavu 4, příjme už jistě dané slovo, ať už bude pokračovat jakkoliv
- Když se automat při běhu dostane do stavu 8, příjme také už jistě dané slovo
- Je tedy jedno, jestli jde automat některým přechodem do 4 nebo 8

Minimalizace konečného automatu



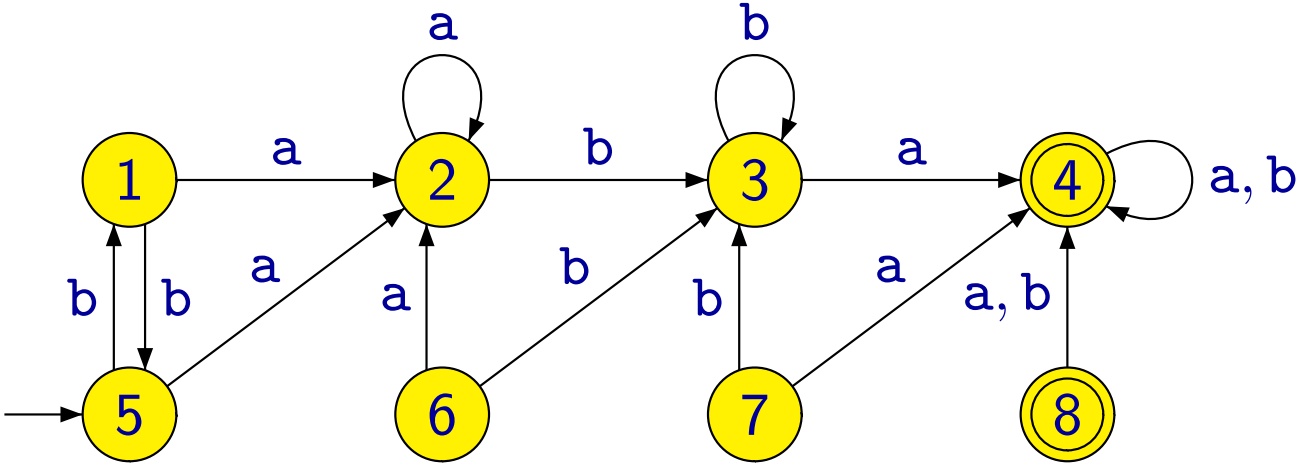
- Když se automat při běhu dostane do stavu 3, příjme každé slovo, které bude v pokračování obsahovat alespoň jedno *a*
- Když se automat při běhu dostane do stavu 7, příjme každé slovo, které bude v pokračování obsahovat alespoň jedno *a*
- Je tedy jedno, jestli jde automat některým přechodem do 3 nebo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

Minimalizace konečného automatu



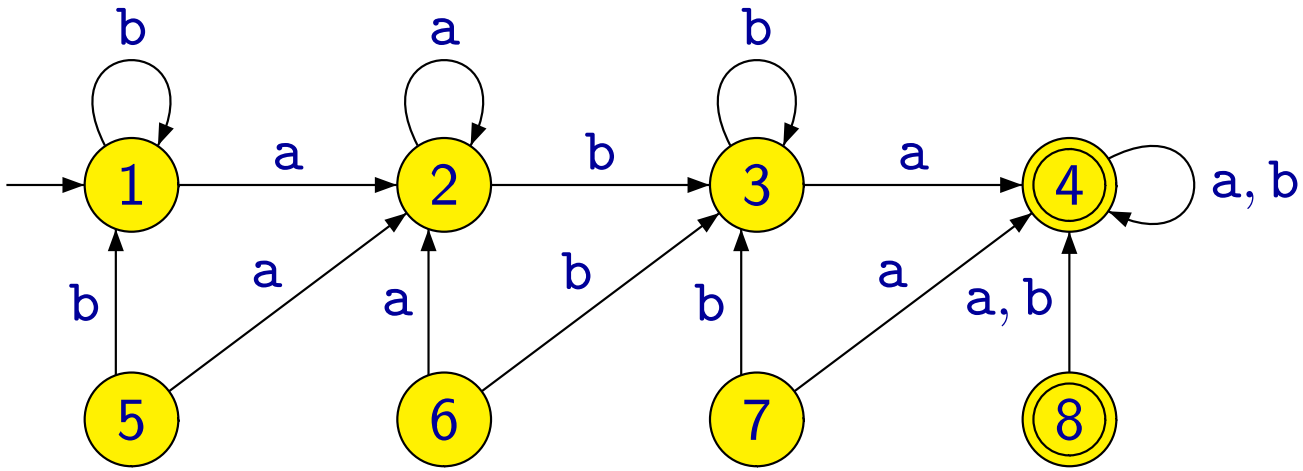
- Když se automat při běhu dostane do stavu 2, příjme každé slovo, které bude v pokračování obsahovat alespoň jedno b a jedno a v tomto pořadí
- Když se automat při běhu dostane do stavu 6, příjme stejná slova
- Je tedy jedno, jestli jde automat některým přechodem do 2 nebo 6

Minimalizace konečného automatu



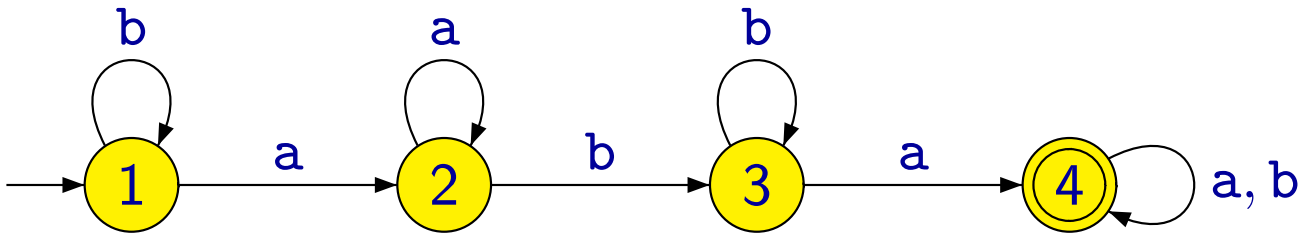
- Ve stavu 1 i 5 automat příjme jakékoliv slovo, které patří do jazyka takových slov, z nichž by vypuštěním některých znaků zůstalo *aba*
- Je tedy jedno, jestli jde automat některým přechodem do 1 nebo 5, a je jedno, ve kterém z nich začne

Minimalizace konečného automatu



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

Minimalizace konečného automatu



Minimalizace konečného automatu

Definice

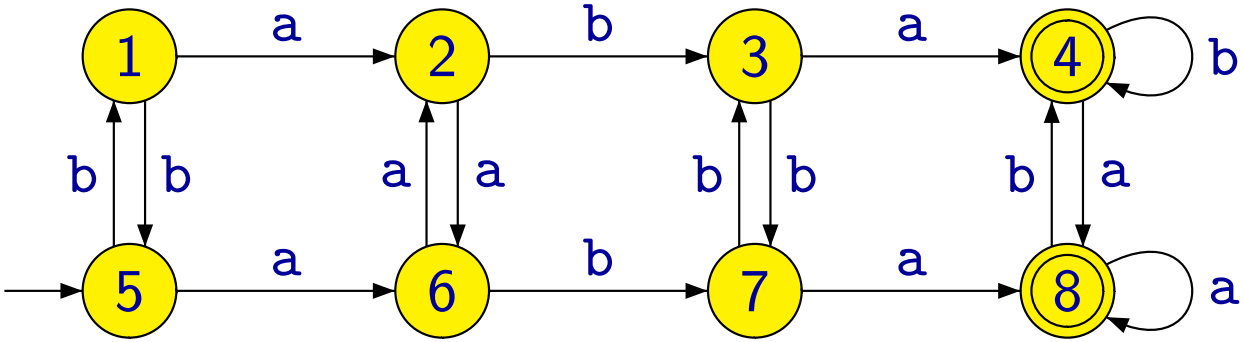
Pro každý stav q automatu $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ definujeme $L(q) = L(\mathcal{A}_q)$, kde $\mathcal{A}_q = (Q, \Sigma, \delta, q, F)$

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 přesměrovat 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
$\leftarrow$ 4	8	4
$\rightarrow$ 5	6	1
6	2	7
7	8	3
$\leftarrow$ 8	8	4

Minimalizace konečného automatu

	a	b
1	2	5
2	6	3
3	4	7
$\leftarrow$ 4	8	4
$\rightarrow$ 5	6	1
6	2	7
7	8	3
$\leftarrow$ 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
$\leftarrow$ 4	8	4
$\rightarrow$ 5	6	1
6	2	7
7	8	3
$\leftarrow$ 8	8	4

		a	b
I	1	2	5
	2	6	3
	3	4	7
	$\rightarrow$ 5	6	1
	6	2	7
	7	8	3
II	$\leftarrow$ 4	8	4
	$\leftarrow$ 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
I	1	2	5
	2	6	3
	3	4	7
	→ 5	6	1
	6	2	7
	7	8	3
II	← 4	8	4
	← 8	8	4

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

- 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
$\leftarrow$ 4	8	4
$\rightarrow$ 5	6	1
6	2	7
7	8	3
$\leftarrow$ 8	8	4

		a	b
I	1	I	I
	2	I	I
	3	II	I
	$\rightarrow$ 5	I	I
	6	I	I
	7	II	I
II	$\leftarrow$ 4	II	II
	$\leftarrow$ 8	II	II

- 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

	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	I
	3	II	I
	→ 5	I	I
	6	I	I
	7	II	I
II	← 4	II	II
	← 8	II	II

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

- 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

	a	b
1	2	5
2	6	3
3	4	7
$\leftarrow$ 4	8	4
$\rightarrow$ 5	6	1
6	2	7
7	8	3
$\leftarrow$ 8	8	4

		a	b
I	1	I	I
	2	I	II
	$\rightarrow$ 5	I	I
	6	I	II
II	3	III	II
	7	III	II
III	$\leftarrow$ 4	III	III
	$\leftarrow$ 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 příjme 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 příjme 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

	a	b
1	2	5
2	6	3
3	4	7
$\leftarrow$ 4	8	4
$\rightarrow$ 5	6	1
6	2	7
7	8	3
$\leftarrow$ 8	8	4

		a	b
I	1	II	I
	$\rightarrow$ 5	II	I
II	2	II	III
	6	II	III
III	3	IV	III
	7	IV	III
IV	$\leftarrow$ 4	IV	IV
	$\leftarrow$ 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

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.

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

Pro každý regulární jazyk L existuje právě jeden minimální automat A v normovaném tvaru takový, že $L = L(A)$.

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

Pro každý regulární jazyk L existuje právě jeden minimální automat A v normovaném tvaru takový, že $L = L(A)$.

Lemma

V každém konečném automatu $A = \{Q, \Sigma, \delta, q_0, F\}$, který není minimální, najdeme alespoň jednu dvojici stavů q_1, q_2 $(q_1 \neq q_2)$ takovou, že $L(q_1) \neq L(q_2)$.

Porovnání konečných automatů

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

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 Σ je jen spočetně mnoho, zatímco jazyků nad abecedou Σ je nespočetně mnoho.

Neregulární jazyky

Jak dokázat o nějakém jazyce L, že není regulární?

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

Jak ale dokázat, že něco neexistuje?

Neregulární jazyky

Jak dokázat o nějakém jazyce L, že není regulární?

Jazyk není regulární, jestliže neexistuje (tj. není možné sestrojit) 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.

Neregulární jazyky

Jak dokázat o nějakém jazyce L, že není regulární?

Jazyk není regulární, jestliže neexistuje (tj. není možné sestrojit) 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 (nebudeme se jí dále zabývat)

Tvrzení

Každý konečný jazyk je regulární.

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

Neregulární jazyky

Tvrzení

Každý konečný jazyk je regulární.

Důkaz: Konečný jazyk s n slovy můžeme popsat regulárním výrazem $w_1 + w_2 + \ldots + w_n$. 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:

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

délky $k+1$, kde

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

Pumping Lemma

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 $i < j$.

$$q_0, \cdots, q_i, \cdots, q_j, \cdots, q_k$$

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:

$$\underbrace{a_1 \cdots a_i}_{u} \quad \underbrace{a_{j+1} \cdots a_k}_{w}$$

$$\underbrace{a_1 \cdots a_i}_{u} \quad \underbrace{a_{i+1} \cdots a_j}_{v} \quad \underbrace{a_{i+1} \cdots a_j}_{v} \quad \underbrace{a_{j+1} \cdots a_k}_{w}$$

$$\underbrace{a_1 \cdots a_i}_{u} \quad \underbrace{a_{i+1} \cdots a_j}_{v} \quad \underbrace{a_{i+1} \cdots a_j}_{v} \quad \underbrace{a_{i+1} \cdots a_j}_{v} \quad \underbrace{a_{j+1} \cdots a_k}_{w}$$

$$\cdots$$

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

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

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

Pumping Lemma

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.

Pumping Lemma

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 $uv^i w \notin L$.

Pumping Lemma

Poznámka: Dokazování platnosti či neplatnosti formule prvého 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.

Pumping Lemma

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.

Bezkontextové gramatiky

Bezkontextové gramatiky

Příklad:

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

Bezkontextové gramatiky

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

Pravidla popisují, jaké řetězce může daný neterminál reprezentovat.

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

$$\textbf{while } x \leq y \textbf{ do begin } x := (x + 1)\,;\, y := (y - 1) \textbf{ end}$$

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

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

Bezkontextové gramatiky

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

$\langle \text{STMT} \rangle$

Bezkontextové gramatiky

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

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

Bezkontextové gramatiky

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** $\langle \text{STMT} \rangle$

Bezkontextové gramatiky

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$\rangle$

Bezkontextové gramatiky

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$\rangle$

Bezkontextové gramatiky

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** $\langle \text{STMT} \rangle$

Bezkontextové gramatiky

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** $\langle \text{STMT} \rangle$

Bezkontextové gramatiky

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** $\langle \text{STMT} \rangle$

Bezkontextové gramatiky

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$\rangle$

Bezkontextové gramatiky

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$\rangle$

Bezkontextové gramatiky

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**

Bezkontextové gramatiky

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**
...

Bezkontextové gramatiky

Formálně je **bezkontextová gramatika** definována jako čtveřice

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

kde:

- Π je konečná množina **neterminálních symbolů** (**neterminálů**)
- Σ 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, ...
- Pro označení terminálních symbolů budeme používat malá písmena a, b, c, ... nebo číslice 0, 1, 2, ...
- Pro označení řetězců z $(\Pi \cup \Sigma)^*$ budeme používat malá písmena řecké abecedy α, β, γ, ...
- Místo zápisu (A, α) budeme pro pravidla používat zápis

$$A \to \alpha$$

A – levá strana pravidla
α – pravá strana pravidla

Bezkontextové gramatiky

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

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

$$A \to aBBb$$
$$A \to AaA$$
$$B \to \varepsilon$$
$$B \to bCA$$
$$C \to AB$$
$$C \to a$$
$$C \to b$$

Bezkontextové gramatiky

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

$$A \to \alpha_1 \qquad A \to \alpha_2 \qquad A \to \alpha_3$$

můžeme je stručněji zapsat jako

$$A \to \alpha_1 \mid \alpha_2 \mid \alpha_3$$

Například pravidla dříve uvedené gramatiky můžeme zapsat jako

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

Bezkontextové gramatiky

Gramatiky slouží ke generování slov.

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

$$A \to aBBb \mid AaA$$
$$B \to \varepsilon \mid bCA$$
$$C \to AB \mid a \mid b$$

Například slovo $abbabb$ je možné v gramatice G vygenerovat následujícím způsobem:

Bezkontextové gramatiky

Gramatiky slouží ke generování slov.

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

$$A \to aBBb \mid AaA$$
$$B \to \varepsilon \mid bCA$$
$$C \to AB \mid a \mid b$$

Například slovo $abbabb$ je možné v gramatice G vygenerovat následujícím způsobem:

A

Bezkontextové gramatiky

Gramatiky slouží ke generování slov.

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

$$\underline{A} \to aBBb \mid AaA$$
$$B \to \varepsilon \mid bCA$$
$$C \to AB \mid a \mid b$$

Například slovo $abbabb$ je možné v gramatice G vygenerovat následujícím způsobem:

$\underline{A}$

Bezkontextové gramatiky

Gramatiky slouží ke generování slov.

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

$$\underline{A} \longrightarrow \underline{aBBb} \mid AaA$$
$$B \longrightarrow \varepsilon \mid bCA$$
$$C \longrightarrow AB \mid a \mid b$$

Například slovo $abbabb$ je možné v gramatice G vygenerovat následujícím způsobem:

$\underline{A} \Rightarrow \underline{aBBb}$

Bezkontextové gramatiky

Gramatiky slouží ke generování slov.

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

$$A \to aBBb \mid AaA$$
$$B \to \varepsilon \mid bCA$$
$$C \to AB \mid a \mid b$$

Například slovo $abbabb$ je možné v gramatice G vygenerovat následujícím způsobem:

$A \Rightarrow aBBb$

Bezkontextové gramatiky

Gramatiky slouží ke generování slov.

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

$$A \to aBBb \mid AaA$$
$$B \to \varepsilon \mid bCA$$
$$C \to AB \mid a \mid b$$

Například slovo $abbabb$ je možné v gramatice G vygenerovat následujícím způsobem:

$A \Rightarrow a\underline{B}Bb$

Bezkontextové gramatiky

Gramatiky slouží ke generování slov.

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

$$A \longrightarrow aBBb \mid AaA$$
$$B \longrightarrow \varepsilon \mid bCA$$
$$C \longrightarrow AB \mid a \mid b$$

Například slovo $abbabb$ je možné v gramatice G vygenerovat následujícím způsobem:

$A \Rightarrow aBBb \Rightarrow abCABb$

Bezkontextové gramatiky

Gramatiky slouží ke generování slov.

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

$$A \to aBBb \mid AaA$$
$$B \to \varepsilon \mid bCA$$
$$C \to AB \mid a \mid b$$

Například slovo $abbabb$ je možné v gramatice G vygenerovat následujícím způsobem:

$A \Rightarrow aBBb \Rightarrow abCABb$

Bezkontextové gramatiky

Gramatiky slouží ke generování slov.

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

$$\underline{A} \longrightarrow aBBb \mid AaA$$
$$B \longrightarrow \varepsilon \mid bCA$$
$$C \longrightarrow AB \mid a \mid b$$

Například slovo $abbabb$ je možné v gramatice G vygenerovat následujícím způsobem:

$A \Rightarrow aBBb \Rightarrow abC\underline{A}Bb$

Bezkontextové gramatiky

Gramatiky slouží ke generování slov.

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

$$\underline{A} \longrightarrow \underline{aBBb} \mid AaA$$
$$B \longrightarrow \varepsilon \mid bCA$$
$$C \longrightarrow AB \mid a \mid b$$

Například slovo $abbabb$ je možné v gramatice G vygenerovat následujícím způsobem:

$A \Rightarrow aBBb \Rightarrow abC\underline{A}Bb \Rightarrow abC\underline{aBBb}Bb$

Bezkontextové gramatiky

Gramatiky slouží ke generování slov.

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

$$A \longrightarrow aBBb \mid AaA$$
$$B \longrightarrow \varepsilon \mid bCA$$
$$C \longrightarrow AB \mid a \mid b$$

Například slovo $abbabb$ je možné v gramatice G vygenerovat následujícím způsobem:

$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb$

Bezkontextové gramatiky

Gramatiky slouží ke generování slov.

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

$$A \to aBBb \mid AaA$$
$$B \to \varepsilon \mid bCA$$
$$C \to AB \mid a \mid b$$

Například slovo $abbabb$ je možné v gramatice G vygenerovat následujícím způsobem:

$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaB\underline{B}bBb$

Bezkontextové gramatiky

Gramatiky slouží ke generování slov.

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

$$A \to aBBb \mid AaA$$
$$B \to \varepsilon \mid bCA$$
$$C \to AB \mid a \mid b$$

Například slovo $abbabb$ je možné v gramatice G vygenerovat následujícím způsobem:

$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb$

Bezkontextové gramatiky

Gramatiky slouží ke generování slov.

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

$$A \longrightarrow aBBb \mid AaA$$
$$B \longrightarrow \varepsilon \mid bCA$$
$$C \longrightarrow AB \mid a \mid b$$

Například slovo $abbabb$ je možné v gramatice G vygenerovat následujícím způsobem:

$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb$

Bezkontextové gramatiky

Gramatiky slouží ke generování slov.

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

$$A \longrightarrow aBBb \mid AaA$$
$$B \longrightarrow \varepsilon \mid bCA$$
$$\underline{C} \longrightarrow AB \mid a \mid b$$

Například slovo $abbabb$ je možné v gramatice G vygenerovat následujícím způsobem:

$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow ab\underline{C}aBbBb$

Bezkontextové gramatiky

Gramatiky slouží ke generování slov.

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

$$A \longrightarrow aBBb \mid AaA$$
$$B \longrightarrow \varepsilon \mid bCA$$
$$\underline{C} \longrightarrow AB \mid a \mid \underline{b}$$

Například slovo $abbabb$ je možné v gramatice G vygenerovat následujícím způsobem:

$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow ab\underline{C}aBbBb \Rightarrow ab\underline{b}aBbBb$

Bezkontextové gramatiky

Gramatiky slouží ke generování slov.

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

$$A \longrightarrow aBBb \mid AaA$$
$$B \longrightarrow \varepsilon \mid bCA$$
$$C \longrightarrow AB \mid a \mid b$$

Například slovo $abbabb$ je možné v gramatice G vygenerovat následujícím způsobem:

$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBbBb$

Bezkontextové gramatiky

Gramatiky slouží ke generování slov.

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

$$A \longrightarrow aBBb \mid AaA$$
$$B \longrightarrow \varepsilon \mid bCA$$
$$C \longrightarrow AB \mid a \mid b$$

Například slovo $abbabb$ je možné v gramatice G vygenerovat následujícím způsobem:

$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBbBb$

Bezkontextové gramatiky

Gramatiky slouží ke generování slov.

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

$$A \longrightarrow aBBb \mid AaA$$
$$B \longrightarrow \varepsilon \mid bCA$$
$$C \longrightarrow AB \mid a \mid b$$

Například slovo $abbabb$ je možné v gramatice G vygenerovat následujícím způsobem:

$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBbBb \Rightarrow abbaBbb$

Bezkontextové gramatiky

Gramatiky slouží ke generování slov.

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

$$A \to aBBb \mid AaA$$
$$B \to \varepsilon \mid bCA$$
$$C \to AB \mid a \mid b$$

Například slovo $abbabb$ je možné v gramatice G vygenerovat následujícím způsobem:

$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBbBb \Rightarrow abbaBbb$

Bezkontextové gramatiky

Gramatiky slouží ke generování slov.

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

$$A \longrightarrow aBBb \mid AaA$$
$$B \longrightarrow \varepsilon \mid bCA$$
$$C \longrightarrow AB \mid a \mid b$$

Například slovo $abbabb$ je možné v gramatice G vygenerovat následujícím způsobem:

$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBbBb \Rightarrow abbaBbb$

Bezkontextové gramatiky

Gramatiky slouží ke generování slov.

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

$$A \longrightarrow aBBb \mid AaA$$
$$B \longrightarrow \varepsilon \mid bCA$$
$$C \longrightarrow AB \mid a \mid b$$

Například slovo $abbabb$ je možné v gramatice G vygenerovat následujícím způsobem:

$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBbBb \Rightarrow abbaBbb \Rightarrow abbabb$

Bezkontextové gramatiky

Gramatiky slouží ke generování slov.

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

$$A \to aBBb \mid AaA$$
$$B \to \varepsilon \mid bCA$$
$$C \to AB \mid a \mid b$$

Například slovo $abbabb$ je možné v gramatice G vygenerovat následujícím způsobem:

$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBbBb \Rightarrow abbaBbb \Rightarrow abbabb$

Bezkontextové gramatiky

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 $(A \to \gamma) \in P$.

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

$$aCBbA \Rightarrow aCbCAbA$$

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

Bezkontextové gramatiky

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 $(A \to \gamma) \in P$.

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

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

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

Derivace délky n větné formy α' z větné formy α je posloupnost větných forem

$$\beta_0, \beta_1, \beta_2, \cdots, \beta_n$$

takových, že

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

což můžeme stručněji zapsat jako

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

Bezkontextové gramatiky

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

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

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

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

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

Bezkontextové gramatiky

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

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

Příklad: Chceme vytvořit gramatiku generující jazyk

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

Bezkontextové gramatiky

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

$$S \rightarrow aSb \mid \varepsilon$$

Bezkontextové gramatiky

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

$$S \to aSb \mid \varepsilon$$

$S \Rightarrow \varepsilon$
$S \Rightarrow aSb \Rightarrow ab$
$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$
$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabbb$
$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaaaSbbbb \Rightarrow aaaabbbb$
$\ldots$

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:

$$S \to aSa \mid bSb \mid a \mid b \mid \varepsilon$$

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:

$$S \to aSa \mid bSb \mid a \mid b \mid \varepsilon$$

$$S \Rightarrow aSa \Rightarrow abSba \Rightarrow abaSaba \Rightarrow abaaaba$$

Bezkontextové gramatiky

Příklad: Chceme vytvořit gramatiku generující jazyk L tvořený všemi dobře uzávorkovanými sekvencemi symbolů '(' a ')'.

Například $(()())(()) \in L$, ale $)()) \notin L$.

Bezkontextové gramatiky

Příklad: Chceme vytvořit gramatiku generující jazyk L tvořený všemi dobře uzávorkovanými sekvencemi symbolů '(' a ')'.

Například $(()())(()) \in L$, ale $)()) \notin L$.

Řešení:

$$S \to \varepsilon \mid (S) \mid SS$$

Bezkontextové gramatiky

Příklad: Chceme vytvořit gramatiku generující jazyk L tvořený všemi dobře uzávorkovanými sekvencemi symbolů '(' a ')'.

Například $(()())(()) \in L$, ale $)()) \notin L$.

Řešení:

$$S \to \varepsilon \mid (S) \mid SS$$

$S \Rightarrow SS \Rightarrow (S)S \Rightarrow (S)(S) \Rightarrow (SS)(S) \Rightarrow ((S)S)(S) \Rightarrow$
$(()S)(S) \Rightarrow (()(S))(S) \Rightarrow (()())(S) \Rightarrow (()())((S)) \Rightarrow$
$(()())(())$

Bezkontextové gramatiky

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 $(a + a) * a + (a * a) \in L$.

Bezkontextové gramatiky

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:

$$E \longrightarrow a \mid E + E \mid E * E \mid (E)$$

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:

$$E \rightarrow a \mid E + E \mid E * E \mid (E)$$

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

Derivační strom

$A \longrightarrow aBBb \mid AaA$
$B \longrightarrow \varepsilon \mid bCA$
$C \longrightarrow AB \mid a \mid b$

Derivační strom

A

$A \longrightarrow aBBb \mid AaA$
$B \longrightarrow \varepsilon \mid bCA$
$C \longrightarrow AB \mid a \mid b$

A

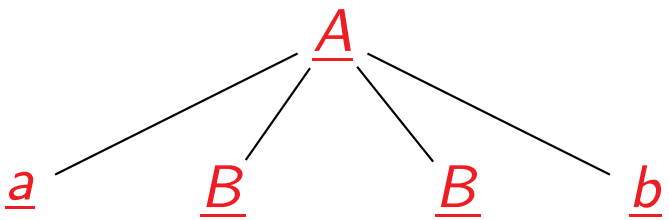
Derivační strom

$\underline{A}$

$\underline{A} \longrightarrow aBBb \mid AaA$
$B \longrightarrow \varepsilon \mid bCA$
$C \longrightarrow AB \mid a \mid b$

$\underline{A}$

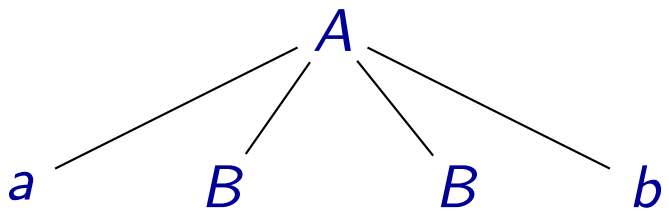
Derivační strom



$A \longrightarrow aBBb \mid AaA$
$B \longrightarrow \varepsilon \mid bCA$
$C \longrightarrow AB \mid a \mid b$

$A \Rightarrow aBBb$

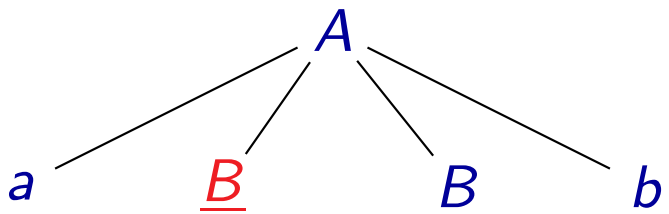
Derivační strom



$A \longrightarrow aBBb \mid AaA$
$B \longrightarrow \varepsilon \mid bCA$
$C \longrightarrow AB \mid a \mid b$

$A \Rightarrow aBBb$

Derivační strom

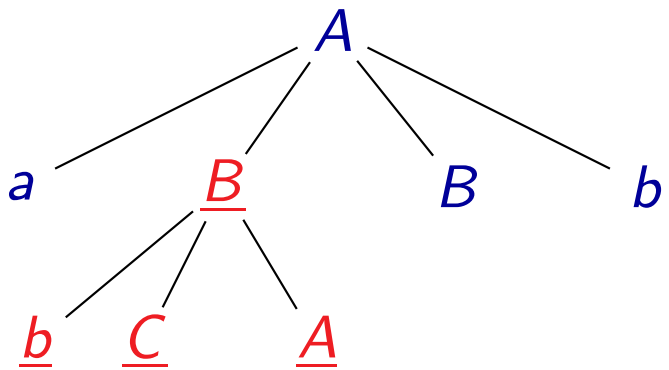


$A \longrightarrow aBBb \mid AaA$
$\underline{B} \longrightarrow \varepsilon \mid bCA$
$C \longrightarrow AB \mid a \mid b$

$A \Rightarrow a\underline{B}Bb$

Derivační strom

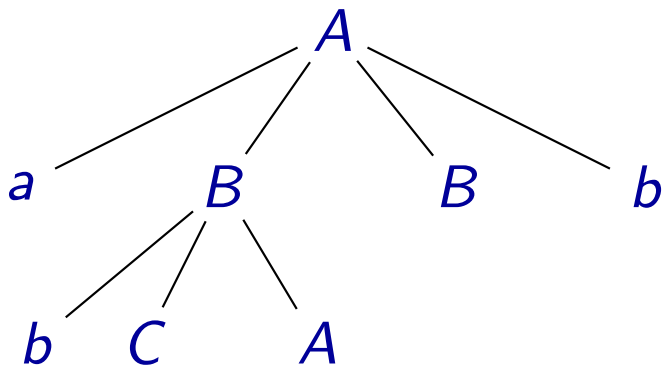
$A \longrightarrow aBBb \mid AaA$
$B \longrightarrow \varepsilon \mid bCA$
$C \longrightarrow AB \mid a \mid b$



$A \Rightarrow aBBb \Rightarrow abCABb$

Derivační strom

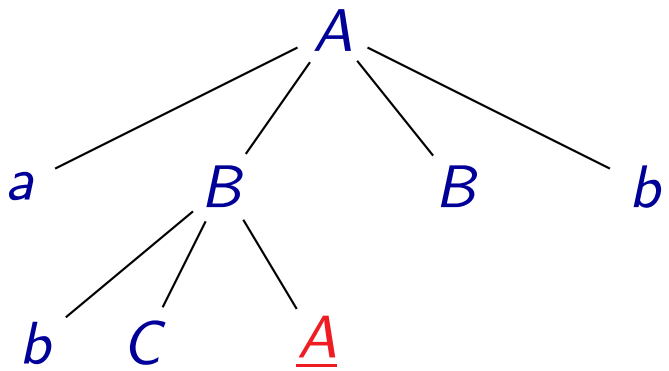
$A \longrightarrow aBBb \mid AaA$
$B \longrightarrow \varepsilon \mid bCA$
$C \longrightarrow AB \mid a \mid b$



$A \Rightarrow aBBb \Rightarrow abCABb$

Derivační strom

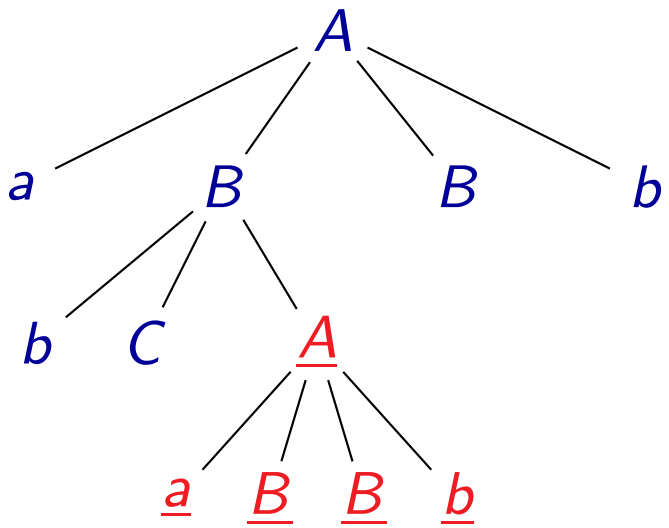
$\underline{A} \longrightarrow aBBb \mid AaA$
$B \longrightarrow \varepsilon \mid bCA$
$C \longrightarrow AB \mid a \mid b$



$A \Rightarrow aBBb \Rightarrow abC\underline{A}Bb$

Derivační strom

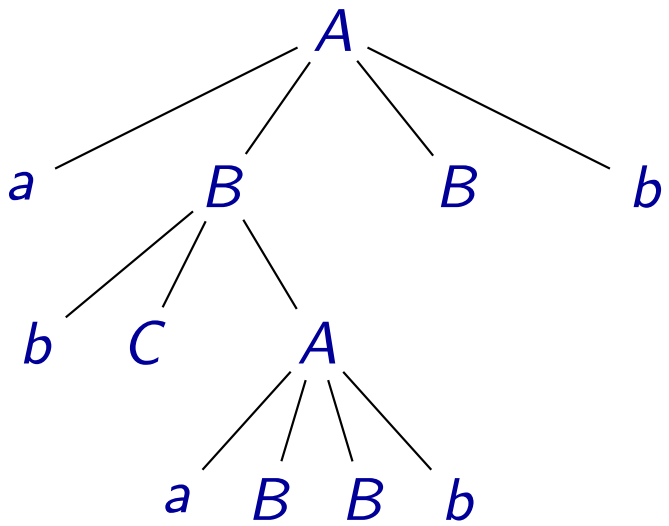
$\underline{A} \longrightarrow \underline{aBBb} \mid AaA$
$B \longrightarrow \varepsilon \mid bCA$
$C \longrightarrow AB \mid a \mid b$



$A \Rightarrow aBBb \Rightarrow abC\underline{A}Bb \Rightarrow abC\underline{aBBb}Bb$

Derivační strom

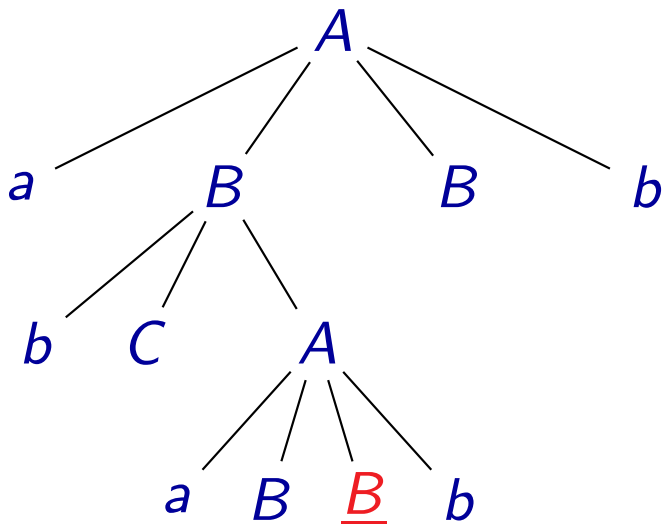
$A \to aBBb \mid AaA$
$B \to \varepsilon \mid bCA$
$C \to AB \mid a \mid b$



$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb$

Derivační strom

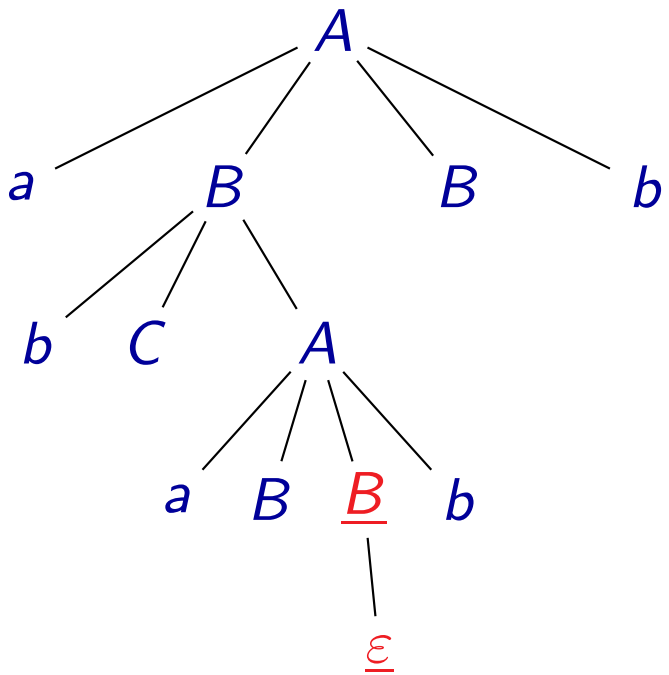
$A \longrightarrow aBBb \mid AaA$
$B \longrightarrow \varepsilon \mid bCA$
$C \longrightarrow AB \mid a \mid b$



$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaB\underline{B}bBb$

Derivační strom

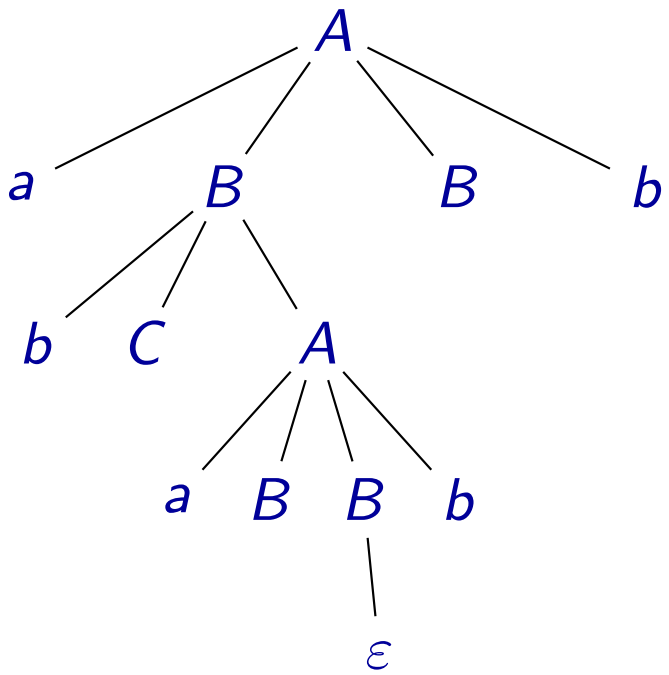
$A \longrightarrow aBBb \mid AaA$
$\underline{B} \longrightarrow \underline{\varepsilon} \mid bCA$
$C \longrightarrow AB \mid a \mid b$



$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaB\underline{B}bBb \Rightarrow abCaBbBb$

Derivační strom

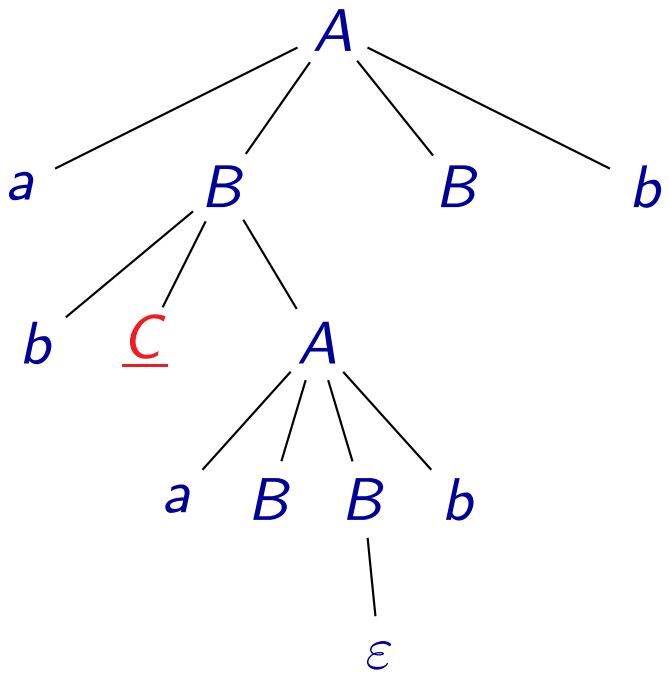
$A \longrightarrow aBBb \mid AaA$
$B \longrightarrow \varepsilon \mid bCA$
$C \longrightarrow AB \mid a \mid b$



$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb$

Derivační strom

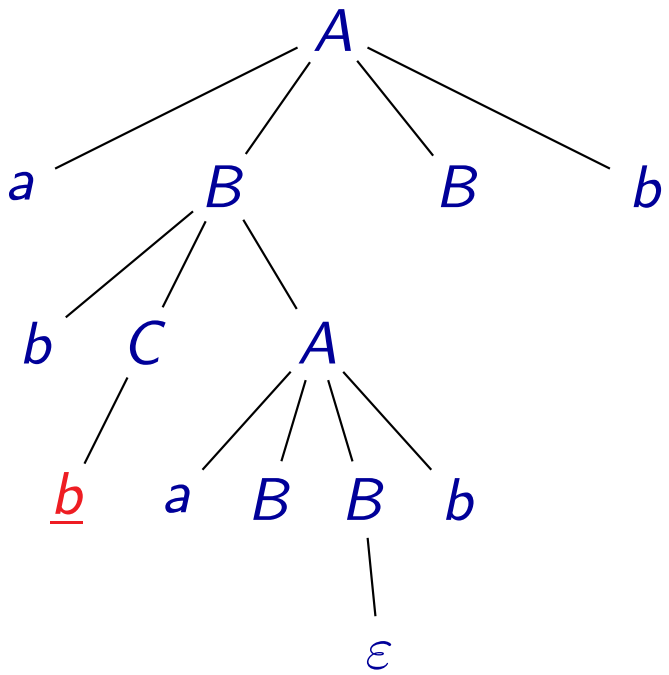
$A \longrightarrow aBBb \mid AaA$
$B \longrightarrow \varepsilon \mid bCA$
$\underline{C} \longrightarrow AB \mid a \mid b$



$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow ab\underline{C}aBbBb$

Derivační strom

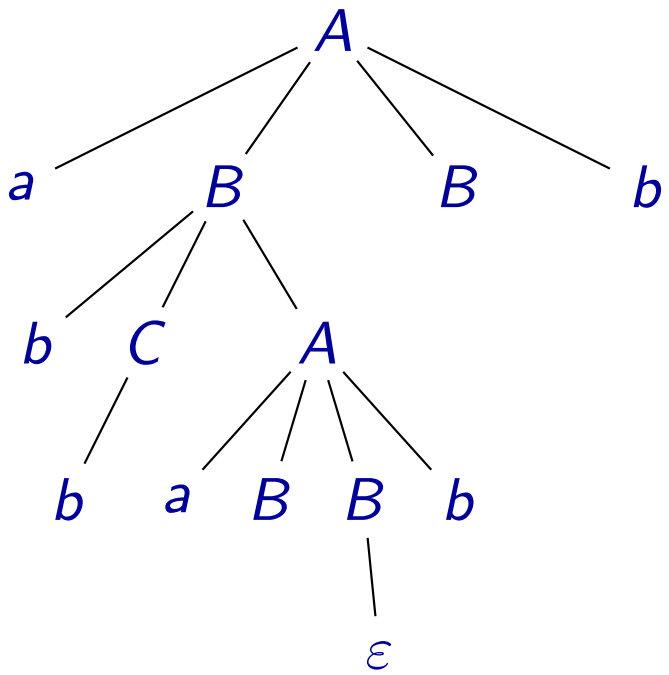
$A \longrightarrow aBBb \mid AaA$
$B \longrightarrow \varepsilon \mid bCA$
$\underline{C} \longrightarrow AB \mid a \mid \underline{b}$



$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow ab\underline{C}aBbBb \Rightarrow ab\underline{b}aBbBb$

Derivační strom

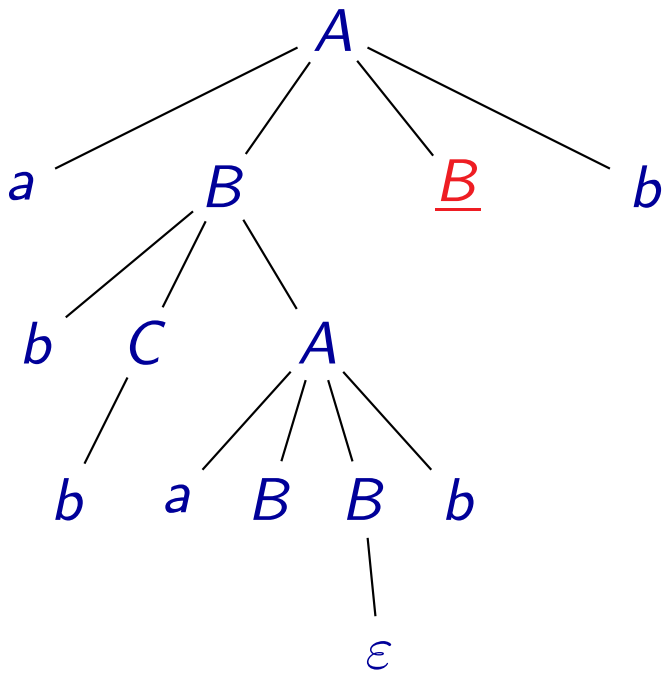
$A \longrightarrow aBBb \mid AaA$
$B \longrightarrow \varepsilon \mid bCA$
$C \longrightarrow AB \mid a \mid b$



$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBbBb$

Derivační strom

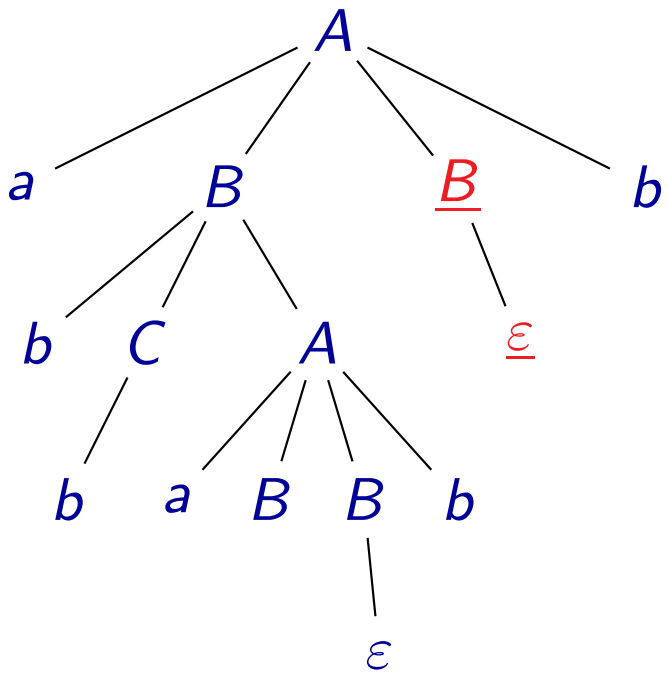
$A \longrightarrow aBBb \mid AaA$
$\underline{B} \longrightarrow \varepsilon \mid bCA$
$C \longrightarrow AB \mid a \mid b$



$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBb\underline{B}b$

Derivační strom

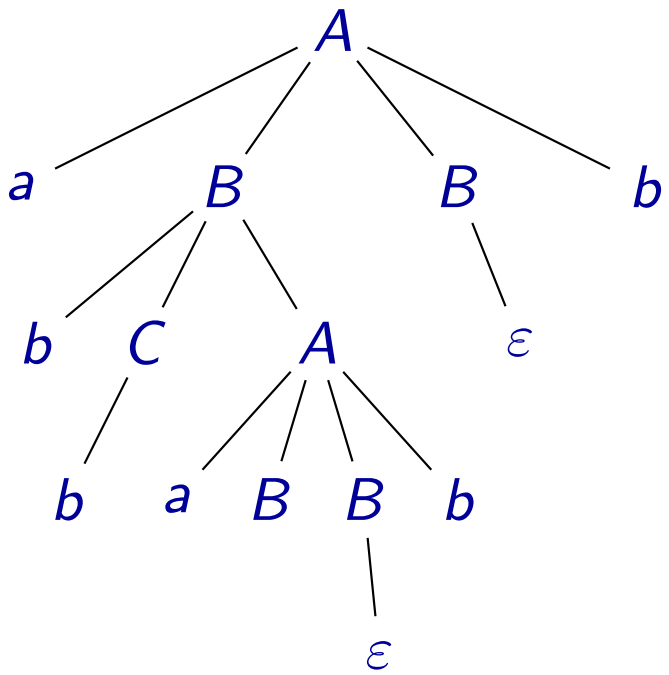
$A \longrightarrow aBBb \mid AaA$
$\underline{B} \longrightarrow \underline{\varepsilon} \mid bCA$
$C \longrightarrow AB \mid a \mid b$



$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBb\underline{B}b \Rightarrow abbaBbb$

Derivační strom

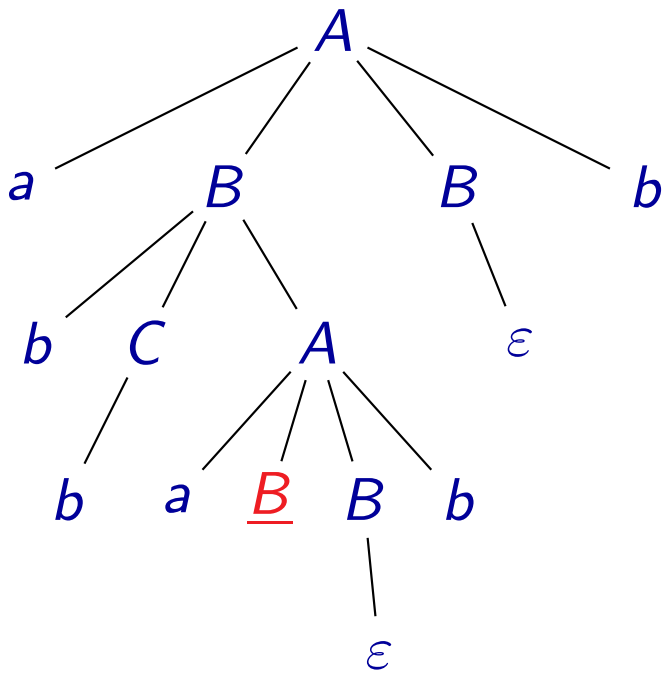
$A \longrightarrow aBBb \mid AaA$
$B \longrightarrow \varepsilon \mid bCA$
$C \longrightarrow AB \mid a \mid b$



$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBbBb \Rightarrow abbaBbb$

Derivační strom

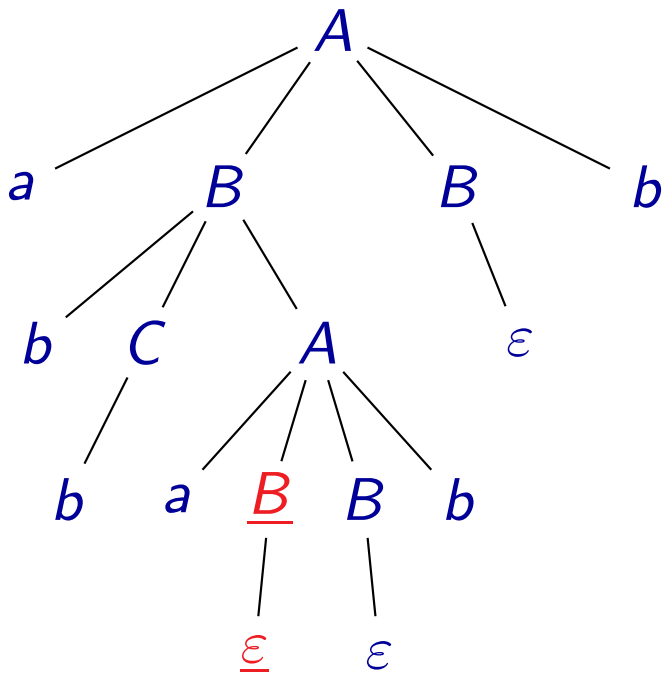
$A \longrightarrow aBBb \mid AaA$
$\underline{B} \longrightarrow \varepsilon \mid bCA$
$C \longrightarrow AB \mid a \mid b$



$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBbBb \Rightarrow abba\underline{B}bb$

Derivační strom

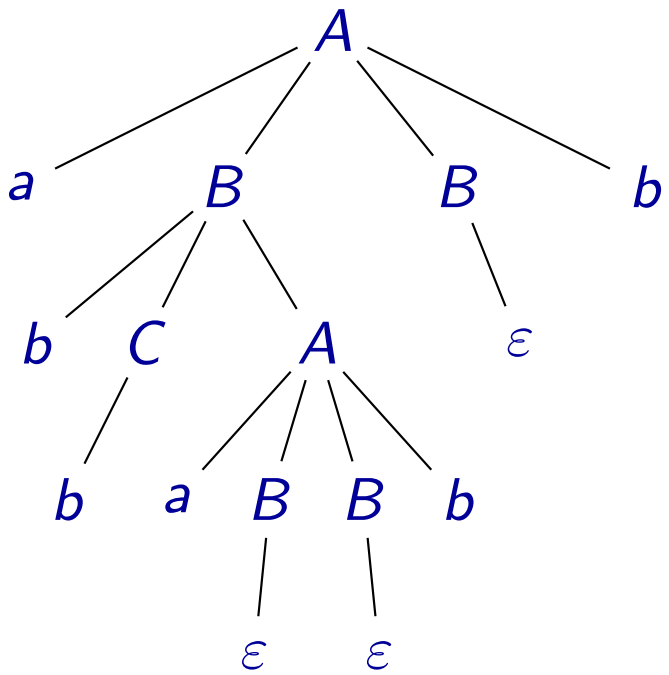
$A \longrightarrow aBBb \mid AaA$
$\underline{B} \longrightarrow \underline{\varepsilon} \mid bCA$
$C \longrightarrow AB \mid a \mid b$



$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBbBb \Rightarrow abba\underline{B}bb \Rightarrow abbabb$

Derivační strom

$A \to aBBb \mid AaA$
$B \to \varepsilon \mid bCA$
$C \to AB \mid a \mid b$



$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBbBb \Rightarrow abbaBbb \Rightarrow abbabb$

Derivační strom

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 ε.
- Ostatní vrcholy stromu jsou ohodnoceny neterminály.
- Pokud je vrchol ohodnocen neterminálem A, pak jeho potomci jsou ohodnoceni symboly pravé strany nějakého přepisovacího pravidla $A \to \alpha$.

Levá a pravá derivace

$$E \rightarrow E + E \mid E * E \mid (E) \mid a$$

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:

$$\underline{E} \Rightarrow \underline{E} + E \Rightarrow E * \underline{E} + E \Rightarrow E * a + \underline{E} \Rightarrow \underline{E} * a + a \Rightarrow a * a + a$$

Levá a pravá derivace

- Jednomu derivačnímu stromu může odpovídat více různých derivací.
- Každému derivačnímu stromu odpovídá právě jedna levá a právě jedna pravá derivace.

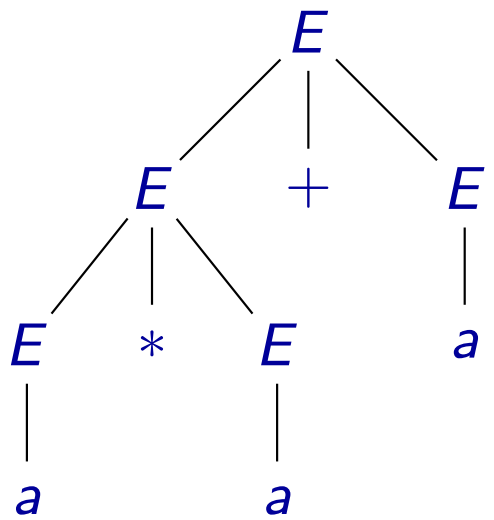
Nejednoznačné gramatiky

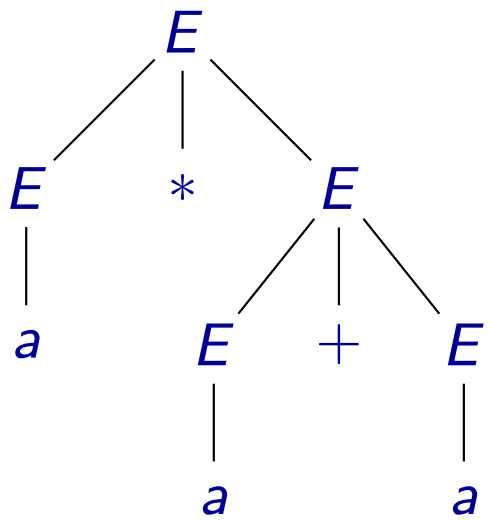
Gramatika G je **nejednoznačná**, jestliže existuje nějaké slovo $w \in L(G)$, kterému přísluší dva různé derivační stromy, resp. dvě různé levé či dvě různé pravé derivace.

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

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

můžeme nahradit ekvivalentní gramatikou

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

Poznámka: Pokud se nejednoznačná gramatika žádnou ekvivalentní jednoznačnou gramatikou nahradit nedá, říkáme, že je **podstatně nejednoznačná**.

Ekvivalence gramatik

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 Σ^*.

Bezkontextové jazyky

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

Bezkontextové jazyky

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 $L(G) = L(G_1)L(G_2)$:

$$G = (\Pi_1 \cup \Pi_2 \cup \{S\},\ \Sigma,\ S,\ P_1 \cup P_2 \cup \{S \to S_1 S_2\})$$

- Gramatika G taková, že $L(G) = L(G_1) \cup L(G_2)$:

$$G = (\Pi_1 \cup \Pi_2 \cup \{S\},\ \Sigma,\ S,\ P_1 \cup P_2 \cup \{S \to S_1, S \to S_2\})$$

- Gramatika G taková, že $L(G) = L(G_1)^*$:

$$G = (\Pi_1 \cup \{S\},\ \Sigma,\ S,\ P_1 \cup \{S \to \varepsilon, S \to S_1 S\})$$

Redukované gramatiky

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 sestrojit ekvivalentní redukovanou gramatiku.

Redukované gramatiky

Algoritmus, který zkonstruuje množinu

$$\mathcal{T} = \{X \in \Pi \mid \exists w \in \Sigma^* : X \Rightarrow^* w\}$$

```
1  T ← ∅
2  change ← TRUE
3  while change
4        do change ← FALSE
5           for each (X → α) ∈ P
6                do if X ∉ T and α ∈ (T ∪ Σ)*
7                   then T ← T ∪ {X}
8                        change ← TRUE
9  return T
```

Z gramatiky G pak vytvoříme gramatiku G' tak, že z G odstraníme všechny neterminály nepatřící do $\mathcal{T}$ a pravidla, kde se vyskytují.

Redukované gramatiky

Příklad: Gramatika $G = (\{S, A, B, C\}, \{a, b\}, S, P)$

$$
\begin{aligned}
S &\to A \mid B \\
A &\to aB \mid bS \mid b \\
B &\to AB \mid Ba \\
C &\to AS \mid b
\end{aligned}
$$

Redukované gramatiky

Příklad: Gramatika $G = (\{S, A, B, C\}, \{a, b\}, S, P)$

$$
\begin{aligned}
S &\to A \mid B \\
A &\to aB \mid bS \mid b \\
B &\to AB \mid Ba \\
C &\to AS \mid b
\end{aligned}
$$

$$\mathcal{T} = \{$$

Redukované gramatiky

Příklad: Gramatika $G = (\{S, A, B, C\}, \{a, b\}, S, P)$

$$
\begin{aligned}
S &\to A \mid B \\
A &\to aB \mid bS \mid b \\
B &\to AB \mid Ba \\
C &\to AS \mid b
\end{aligned}
$$

$$\mathcal{T} = \{A,$$

Redukované gramatiky

Příklad: Gramatika $G = (\{S, A, B, C\}, \{a, b\}, S, P)$

$$
\begin{aligned}
S &\to A \mid B \\
A &\to aB \mid bS \mid b \\
B &\to AB \mid Ba \\
C &\to AS \mid b
\end{aligned}
$$

$$\mathcal{T} = \{A, C,$$

Redukované gramatiky

Příklad: Gramatika $G = (\{S, A, B, C\}, \{a, b\}, S, P)$

$$
\begin{aligned}
S &\to A \mid B \\
A &\to aB \mid bS \mid b \\
B &\to AB \mid Ba \\
C &\to AS \mid b
\end{aligned}
$$

$$\mathcal{T} = \{A, C, S$$

Redukované gramatiky

Příklad: Gramatika $G = (\{S, A, B, C\}, \{a, b\}, S, P)$

$$
\begin{aligned}
S &\to A \mid B \\
A &\to aB \mid bS \mid b \\
B &\to AB \mid Ba \\
C &\to AS \mid b
\end{aligned}
$$

$$\mathcal{T} = \{A, C, S\}$$

Redukované gramatiky

Příklad: Gramatika $G = (\{S, A, B, C\}, \{a, b\}, S, P)$

$$
\begin{aligned}
S &\to A \mid B \\
A &\to aB \mid bS \mid b \\
B &\to AB \mid Ba \\
C &\to AS \mid b
\end{aligned}
$$

$$\mathcal{T} = \{A, C, S\}$$

Gramatika $G' = (\{S, A, C\}, \{a, b\}, S, P')$

$$
\begin{aligned}
S &\to A \\
A &\to bS \mid b \\
C &\to AS \mid b
\end{aligned}
$$

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  D ← {S}
2  change ← TRUE
3  while change
4        do change ← FALSE
5           for each (X → α) ∈ P' where X ∈ D
6                do for i ← 1 to |α|
7                         do if α[i] ∈ Π' and α[i] ∉ D
8                               then D ← D ∪ {α[i]}
9                                    change ← TRUE
10 return D
```

Z gramatiky G' pak vytvoříme gramatiku G'' tak, že z G' odstraníme všechny neterminály nepatřící do $\mathcal{D}$ a pravidla, kde se vyskytují.

Redukované gramatiky

Příklad: Gramatika $G' = (\{S, A, C\}, \{a, b\}, S, P')$

$$
\begin{aligned}
S &\to A \\
A &\to bS \mid b \\
C &\to AS \mid b
\end{aligned}
$$

Redukované gramatiky

Příklad: Gramatika $G' = (\{S, A, C\}, \{a, b\}, S, P')$

$$S \to A$$
$$A \to bS \mid b$$
$$C \to AS \mid b$$

$$\mathcal{D} = \{$$

Redukované gramatiky

Příklad: Gramatika $G' = (\{S, A, C\}, \{a, b\}, S, P')$

$$S \to A$$
$$A \to bS \mid b$$
$$C \to AS \mid b$$

$$\mathcal{D} = \{S,$$

Redukované gramatiky

Příklad: Gramatika $G' = (\{S, A, C\}, \{a, b\}, S, P')$

$$S \to A$$
$$A \to bS \mid b$$
$$C \to AS \mid b$$

$$\mathcal{D} = \{S, A$$

Redukované gramatiky

Příklad: Gramatika $G' = (\{S, A, C\}, \{a, b\}, S, P')$

$$S \to A$$
$$A \to bS \mid b$$
$$C \to AS \mid b$$

$$\mathcal{D} = \{S, A\}$$

Redukované gramatiky

Příklad: Gramatika $G' = (\{S, A, C\}, \{a, b\}, S, P')$

$$S \to A$$
$$A \to bS \mid b$$
$$C \to AS \mid b$$

$$\mathcal{D} = \{S, A\}$$

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

$$S \to A$$
$$A \to bS \mid b$$

Redukované gramatiky

Pořadí obou kroků je třeba dodržet. Pokud bychom je provedli v opačném pořadí, můžeme dostat gramatiku, která není redukovaná.

Příklad:

$$S \to a \mid A$$
$$A \to AB$$
$$B \to b$$

Pokud provedeme oba kroky algoritmu ve správném pořadí, dostaneme

$$S \to a$$

Pokud provedeme kroky algoritmu v opačném pořadí, dostaneme

$$S \to a$$
$$B \to b$$

Redukované gramatiky

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 $L(G) = \emptyset$.

Nevypouštějící gramatiky

Definice

Gramatika G je **nevypouštějící**, jestliže neobsahuje žádná ε-pravidla, tj. pravidla tvaru $X \rightarrow \varepsilon$, kde $X \in \Pi$.

Ke každé bezkontextové gramatice je možné sestrojit nevypouštějící gramatiku G' takovou, že $L(G') = L(G) - \{\varepsilon\}$.

Nevypouštějící gramatiky

Algoritmus, který zkonstruuje množinu

$$\mathcal{E} = \{X \in \Pi \mid X \Rightarrow^* \varepsilon\}$$

```
1  E ← ∅
2  change ← TRUE
3  while change
4        do change ← FALSE
5           for each (X → α) ∈ P
6                 do if X ∉ E and α ∈ E*
7                       then E ← E ∪ {X}
8                            change ← TRUE
9  return E
```

Nevypouštějící gramatiky

Nevypouštějící gramatiku G' pak z G vytvoříme tak, že pro každé pravidlo $X \to \alpha$ z G přidáme do G' všechna možná pravidla

$$X \to \alpha'$$

kde α' vznikne z α vypuštěním libovolného počtu symbolů z $\mathcal{E}$, přičemž ale $\alpha' \neq \varepsilon$.

Příklad: Pokud například $\mathcal{E} = \{A, C, D\}$, pak místo pravidla

$$A \to aASCbA$$

přidáme pravidla

$$A \to aASCbA \mid aSCbA \mid aASbA \mid aASCb \mid aSbA \mid aSCb \mid aASb \mid aSb$$

Nevypouštějící gramatiky

Příklad: Gramatika $G = (\{S, A, B, C\}, \{a, b\}, S, P)$

$$
\begin{aligned}
S &\to AB \mid \varepsilon \\
A &\to aAAb \mid BS \mid CA \\
B &\to BbA \mid CaC \mid \varepsilon \\
C &\to aBB \mid bS
\end{aligned}
$$

Nevypouštějící gramatiky

Příklad: Gramatika $G = (\{S, A, B, C\}, \{a, b\}, S, P)$

$$
\begin{aligned}
S &\to AB \mid \varepsilon \\
A &\to aAAb \mid BS \mid CA \\
B &\to BbA \mid CaC \mid \varepsilon \\
C &\to aBB \mid bS
\end{aligned}
$$

$$\mathcal{E} = \{$$

Nevypouštějící gramatiky

Příklad: Gramatika $G = (\{S, A, B, C\}, \{a, b\}, S, P)$

$$
\begin{aligned}
S &\to AB \mid \varepsilon \\
A &\to aAAb \mid BS \mid CA \\
B &\to BbA \mid CaC \mid \varepsilon \\
C &\to aBB \mid bS
\end{aligned}
$$

$$\mathcal{E} = \{S,$$

Nevypouštějící gramatiky

Příklad: Gramatika $G = (\{S, A, B, C\}, \{a, b\}, S, P)$

$$
\begin{aligned}
S &\to AB \mid \varepsilon \\
A &\to aAAb \mid BS \mid CA \\
B &\to BbA \mid CaC \mid \varepsilon \\
C &\to aBB \mid bS
\end{aligned}
$$

$$\mathcal{E} = \{S, B,$$

Nevypouštějící gramatiky

Příklad: Gramatika $G = (\{S, A, B, C\}, \{a, b\}, S, P)$

$$
\begin{aligned}
S &\to AB \mid \varepsilon \\
A &\to aAAb \mid BS \mid CA \\
B &\to BbA \mid CaC \mid \varepsilon \\
C &\to aBB \mid bS
\end{aligned}
$$

$$\mathcal{E} = \{S, B, A$$

Nevypouštějící gramatiky

Příklad: Gramatika $G = (\{S, A, B, C\}, \{a, b\}, S, P)$

$$
\begin{aligned}
S &\to AB \mid \varepsilon \\
A &\to aAAb \mid BS \mid CA \\
B &\to BbA \mid CaC \mid \varepsilon \\
C &\to aBB \mid bS
\end{aligned}
$$

$$\mathcal{E} = \{S, B, A\}$$

Nevypouštějící gramatiky

Příklad: Gramatika $G = (\{S, A, B, C\}, \{a, b\}, S, P)$

$$
\begin{aligned}
S &\to AB \mid \varepsilon \\
A &\to aAAb \mid BS \mid CA \\
B &\to BbA \mid CaC \mid \varepsilon \\
C &\to aBB \mid bS
\end{aligned}
$$

$$\mathcal{E} = \{S, B, A\}$$

Gramatika $G' = (\{S, A, B, C\}, \{a, b\}, S, P')$

$$
\begin{aligned}
S &\to AB \mid A \mid B \\
A &\to aAAb \mid aAb \mid ab \mid BS \mid B \mid S \mid CA \mid C \\
B &\to BbA \mid bA \mid Bb \mid b \mid CaC \\
C &\to aBB \mid aB \mid a \mid bS \mid b
\end{aligned}
$$

Chomského normální forma

Definice

Gramatika G je v **Chomského normální formě**, jestliže všechna její pravidla jsou pouze následujícího tvaru:

- $X \to YZ$, kde $X, Y, Z \in \Pi$, nebo
- $X \to a$, kde $X \in \Pi$ a $a \in \Sigma$.

Ke každé bezkontextové gramatice je možné sestrojit gramatiku G' v Chomského normální formě takovou, že $L(G') = L(G) - \{\varepsilon\}$.

Greibachové normální forma

Definice

Gramatika G je v **Greibachové normální formě**, jestliže všechna její pravidla jsou pouze následujícího tvaru:

- $X \to a\alpha$, kde $a \in \Sigma$, $X \in \Pi$ a $\alpha \in \Pi^*$.

Ke každé bezkontextové gramatice je možné sestrojit gramatiku G' v Greibachové normální formě takovou, že $L(G') = L(G) - \{\varepsilon\}$.

Regulární gramatiky

Definice

Gramatika G je **regulární**, jestliže všechna její pravidla jsou tvaru:

- $X \to wY$, kde $X, Y \in \Pi$, $w \in \Sigma^*$, nebo
- $X \to w$, 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.

Zásobníkové automaty

Zásobníkový automat

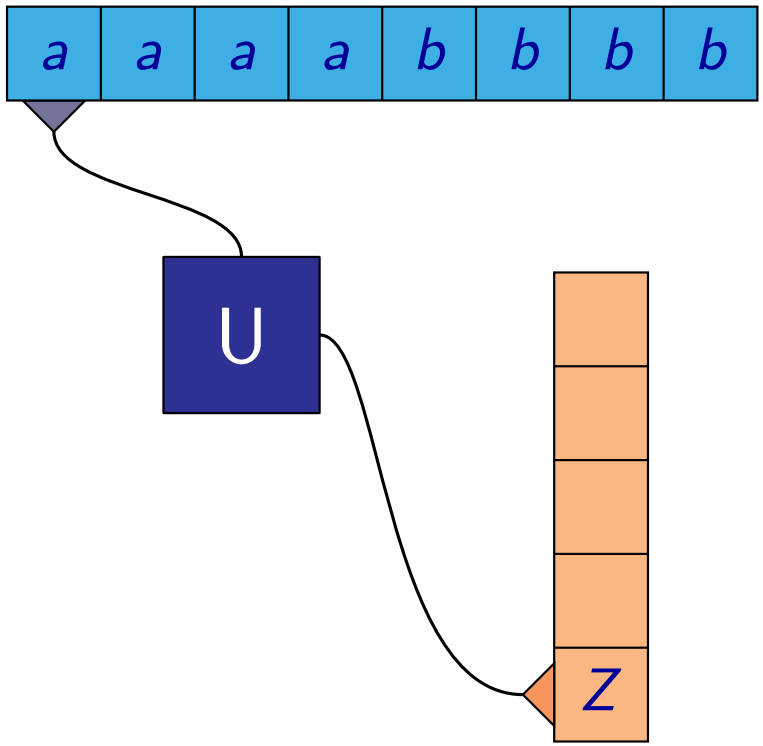
- Chtěli bychom rozpoznávat jazyk $L = \{a^i b^i \mid i \geq 1\}$
- Snažíme se navrhnout zařízení (podobné konečným automatům), které přečte slovo, a sdělí nám, zda toto slovo patří do jazyka L či ne.
- Při čtení a-ček si musíme pamatovat jejich počet, ať víme, kolik musí následovat b-ček

Zásobníkový automat

- 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

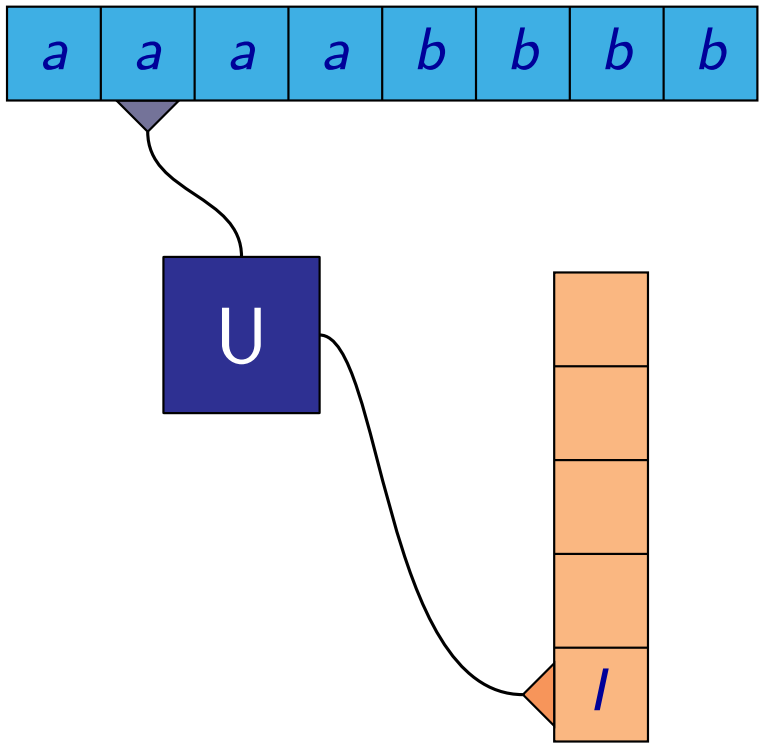
Zásobníkový automat

- Slovo *aaaabbbb* patří do jazyka $L = \{a^i b^i \mid i \geq 1\}$



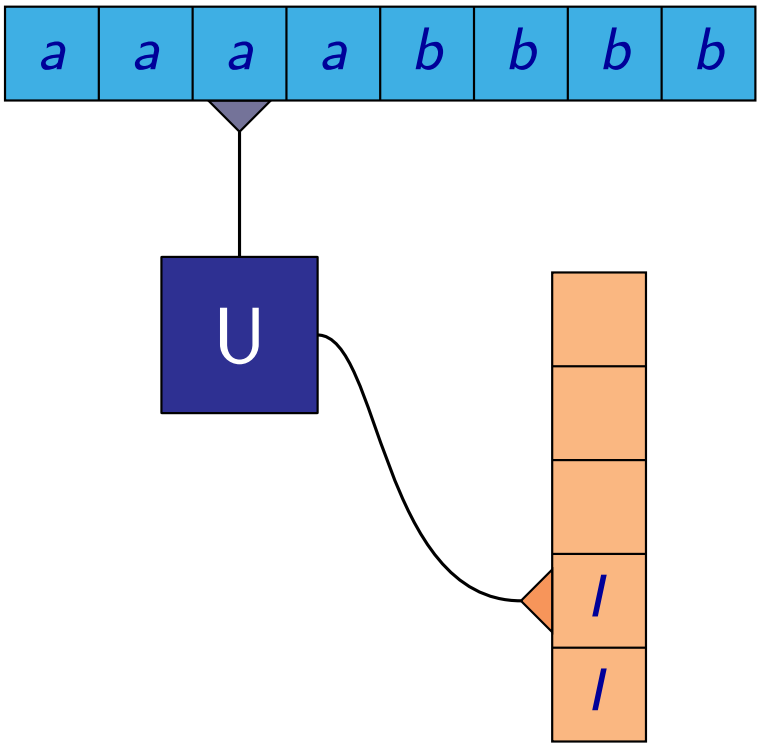
Zásobníkový automat

- Slovo *aaaabbbb* patří do jazyka $L = \{a^i b^i \mid i \geq 1\}$



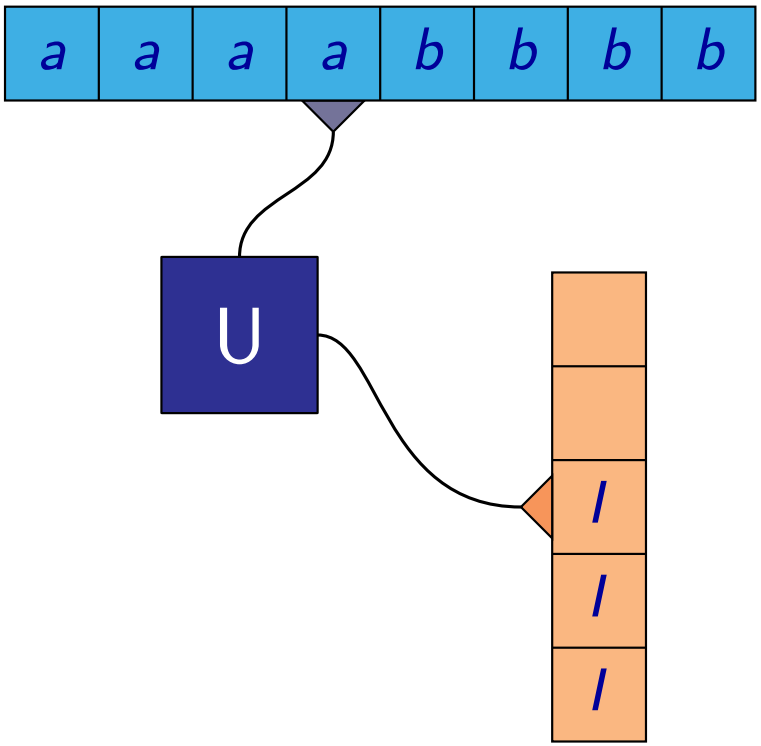
Zásobníkový automat

- Slovo *aaaabbbb* patří do jazyka $L = \{a^i b^i \mid i \geq 1\}$



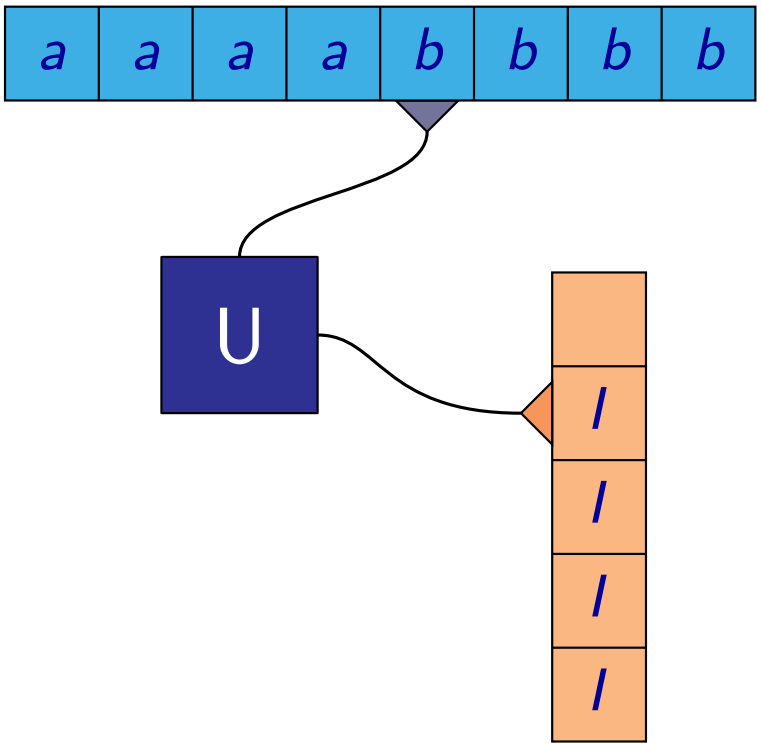
Zásobníkový automat

- Slovo *aaaabbbb* patří do jazyka $L = \{a^i b^i \mid i \geq 1\}$



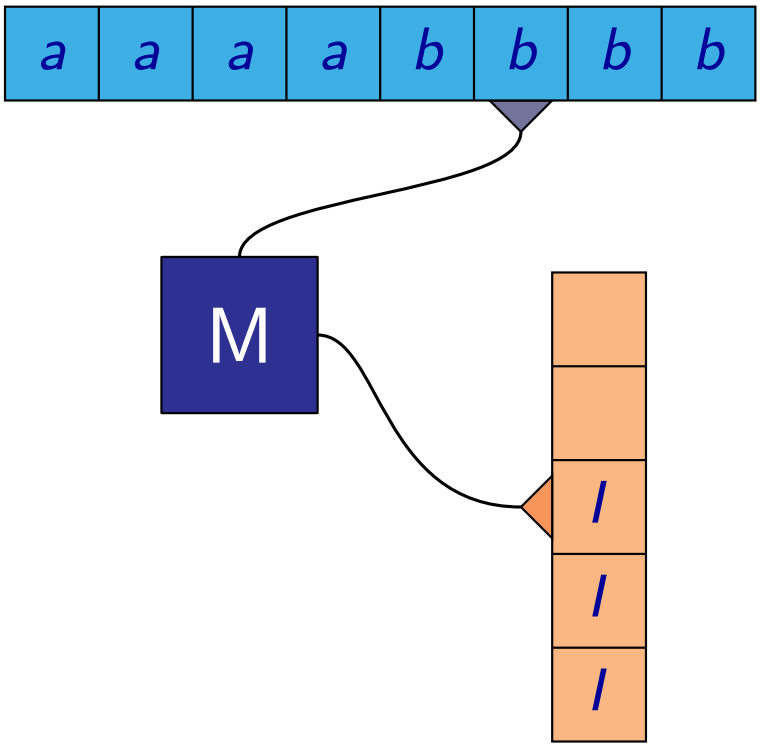
Zásobníkový automat

- Slovo *aaaabbbb* patří do jazyka $L = \{a^i b^i \mid i \geq 1\}$



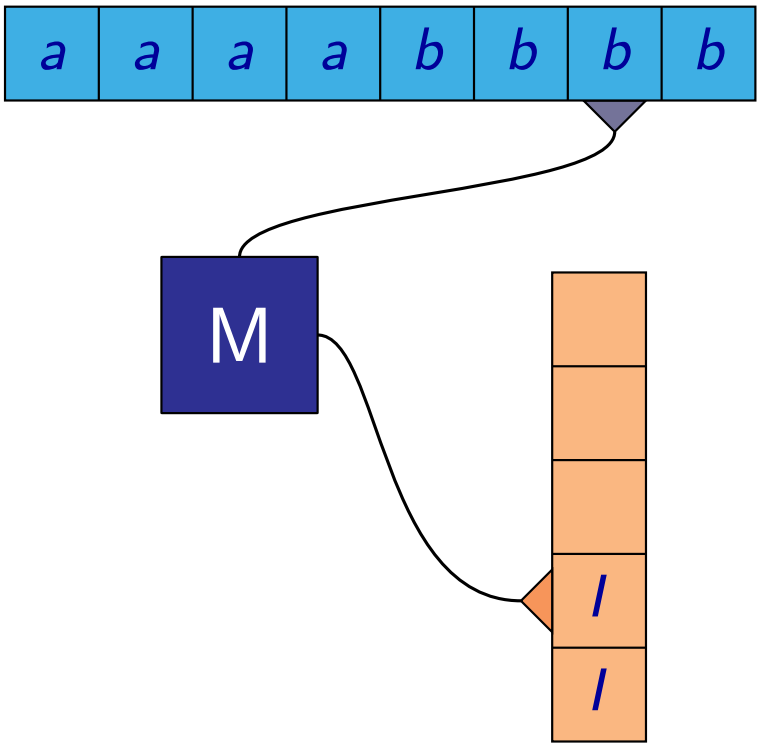
Zásobníkový automat

- Slovo *aaaabbbb* patří do jazyka $L = \{a^i b^i \mid i \geq 1\}$



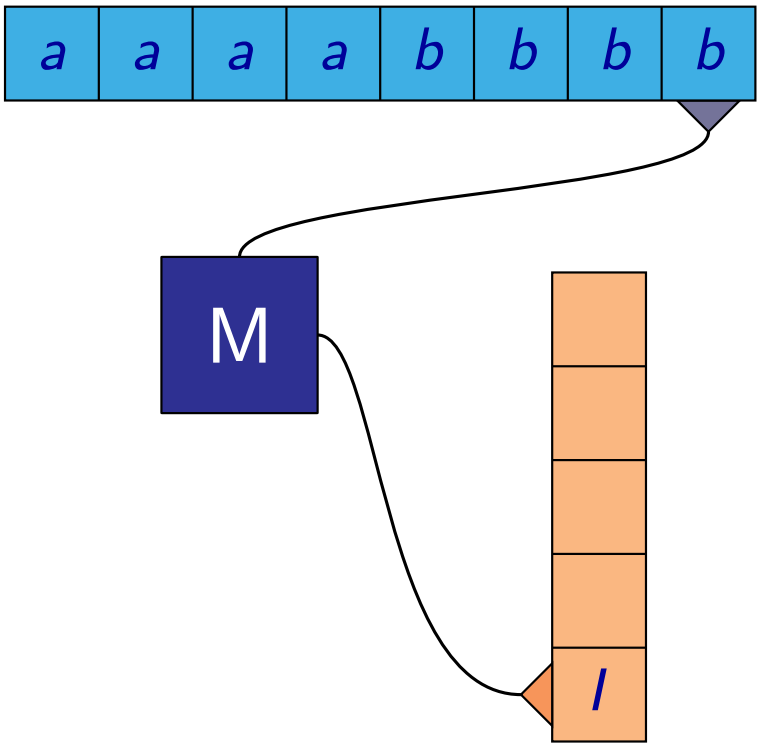
Zásobníkový automat

- Slovo *aaaabbbb* patří do jazyka $L = \{a^i b^i \mid i \geq 1\}$



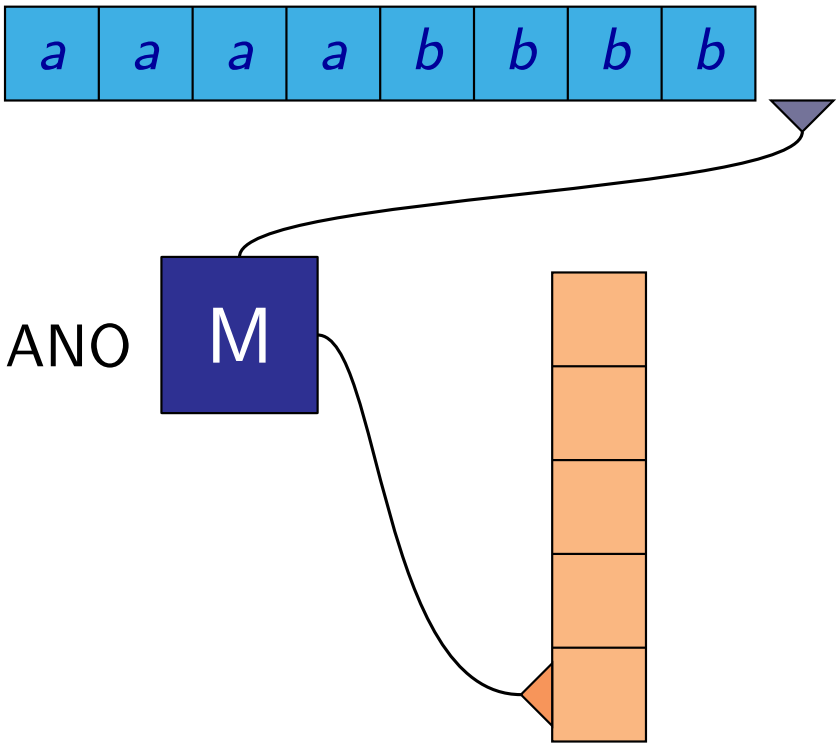
Zásobníkový automat

- Slovo *aaaabbbb* patří do jazyka $L = \{a^i b^i \mid i \geq 1\}$



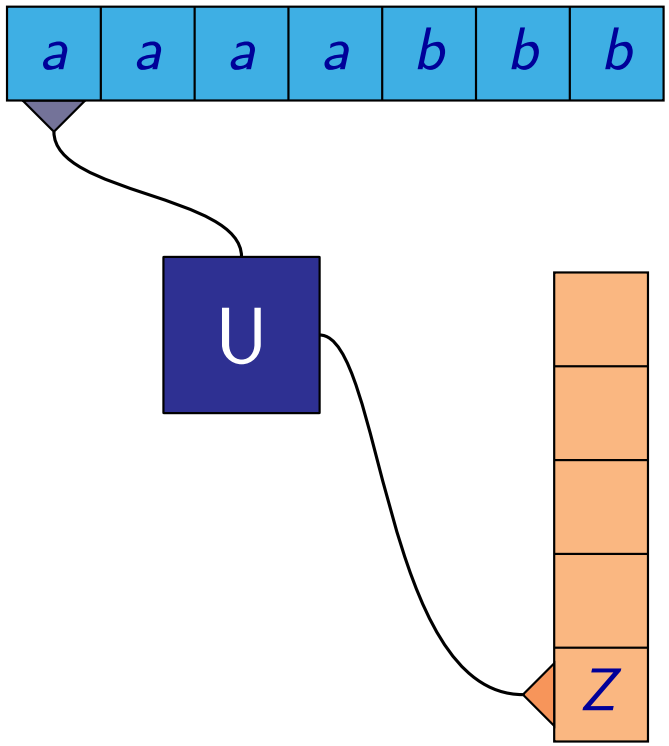
Zásobníkový automat

- 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



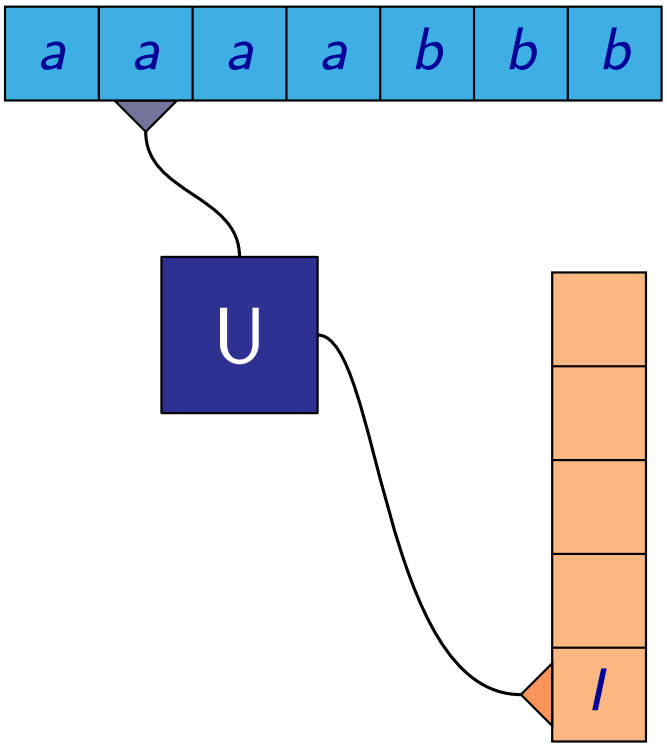
Zásobníkový automat

- Slovo *aaaabbb* nepatří do jazyka $L = \{a^i b^i \mid i \geq 1\}$



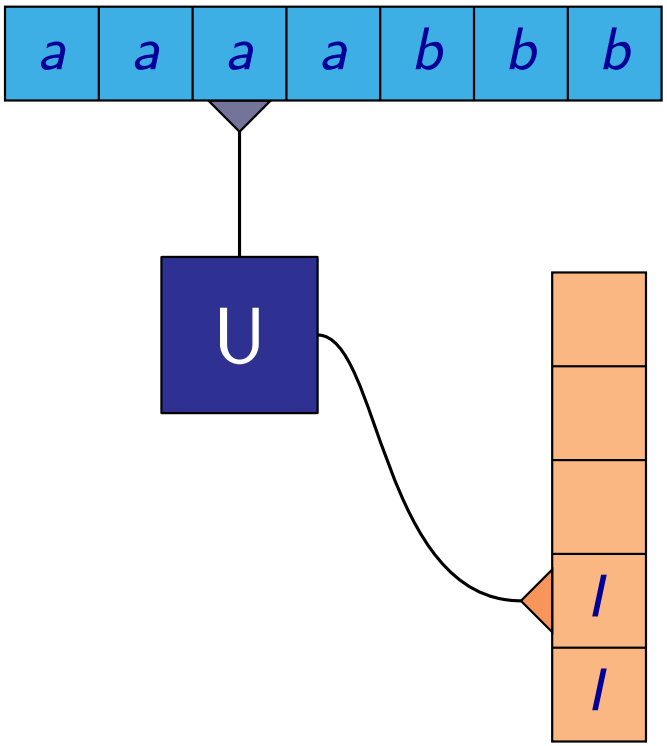
Zásobníkový automat

- Slovo *aaaabbb* nepatří do jazyka $L = \{a^i b^i \mid i \geq 1\}$



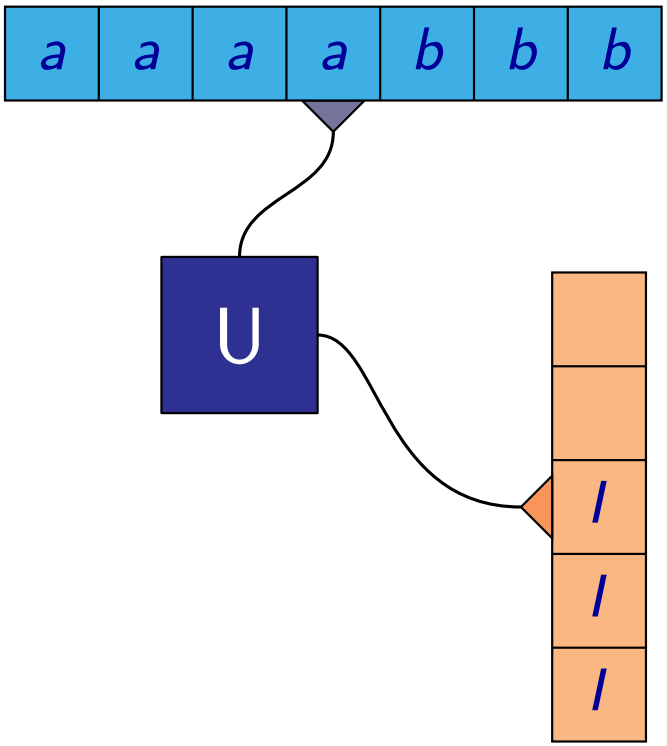
Zásobníkový automat

- Slovo *aaaabbb* nepatří do jazyka $L = \{a^i b^i \mid i \geq 1\}$



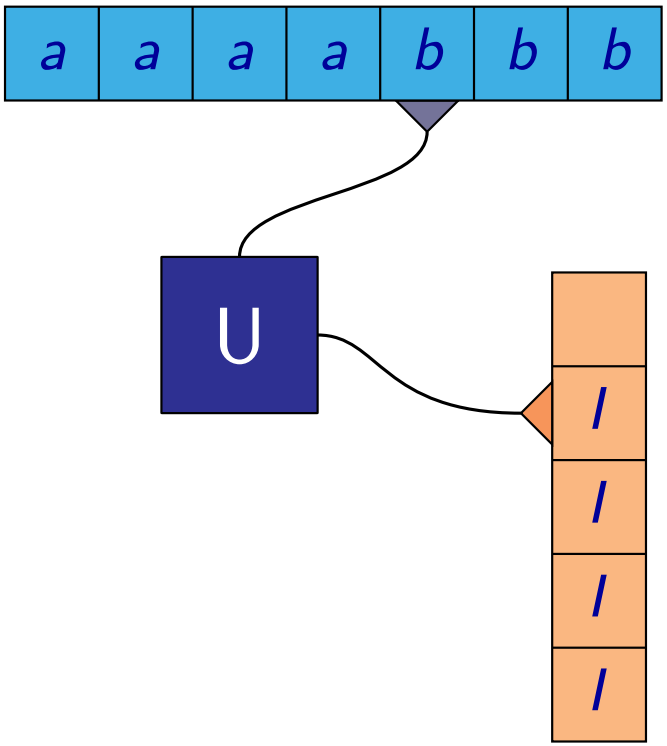
Zásobníkový automat

- Slovo *aaaabbb* nepatří do jazyka $L = \{a^i b^i \mid i \geq 1\}$



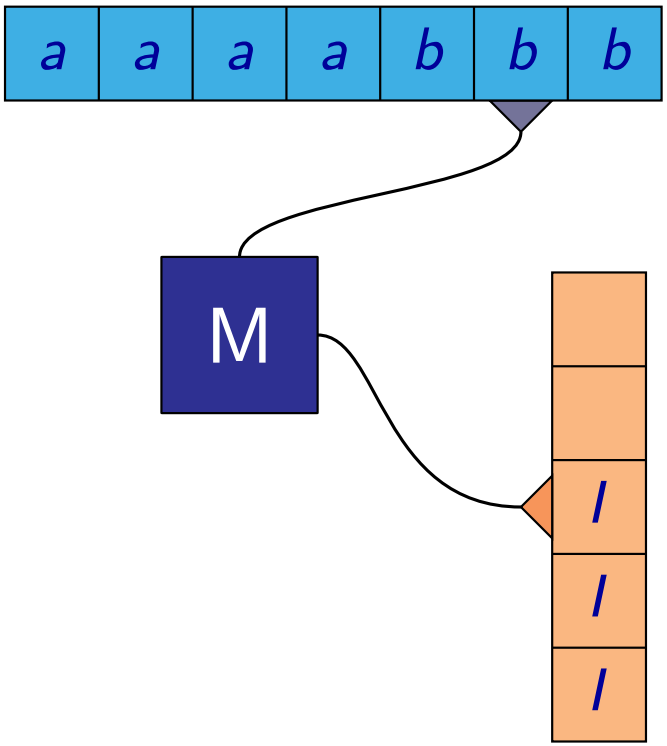
Zásobníkový automat

- Slovo *aaaabbb* nepatří do jazyka $L = \{a^i b^i \mid i \geq 1\}$



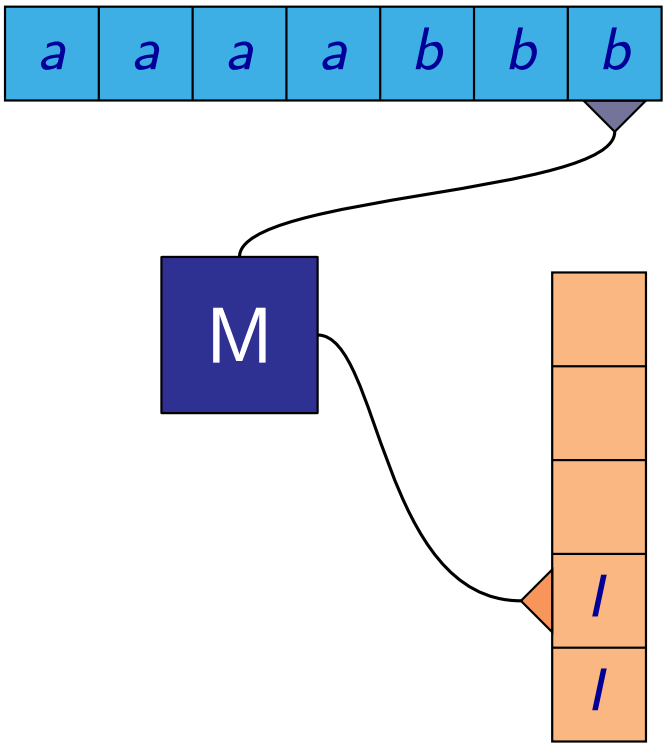
Zásobníkový automat

- Slovo *aaaabbb* nepatří do jazyka $L = \{a^i b^i \mid i \geq 1\}$



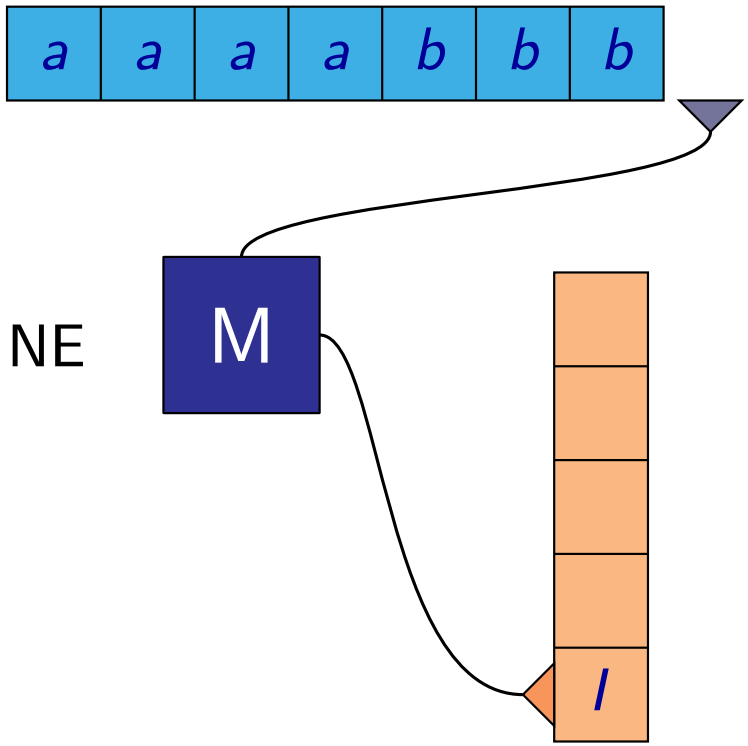
Zásobníkový automat

- Slovo *aaaabbb* nepatří do jazyka $L = \{a^i b^i \mid i \geq 1\}$



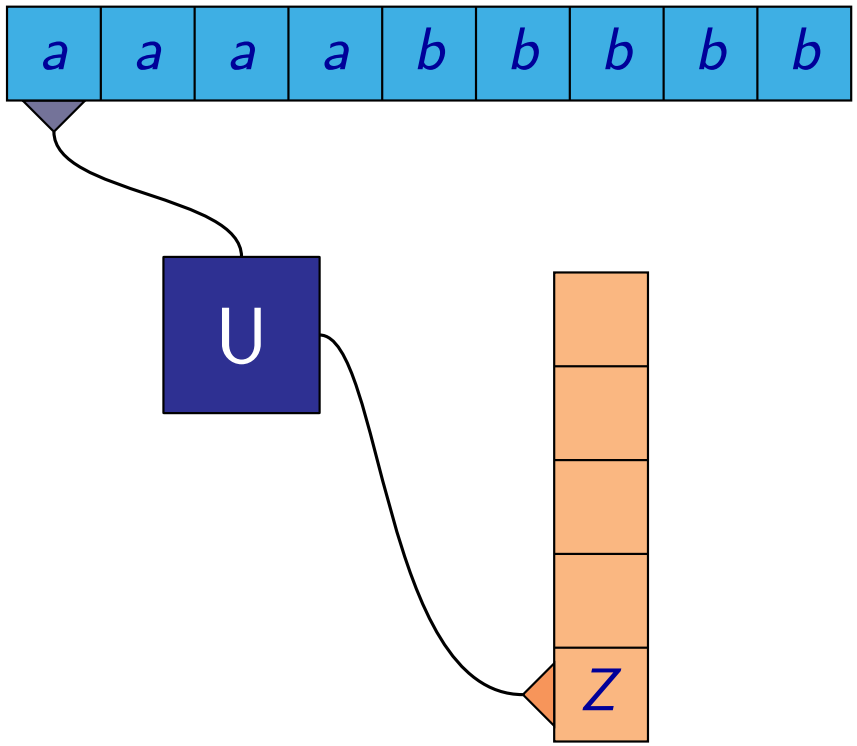
Zásobníkový automat

- Slovo *aaaabbb* nepatří do jazyka $L = \{a^i b^i \mid i \geq 1\}$
- Automat přečetl celé slovo, ale nevyprázdnil zásobník, takže slovo nepřijal



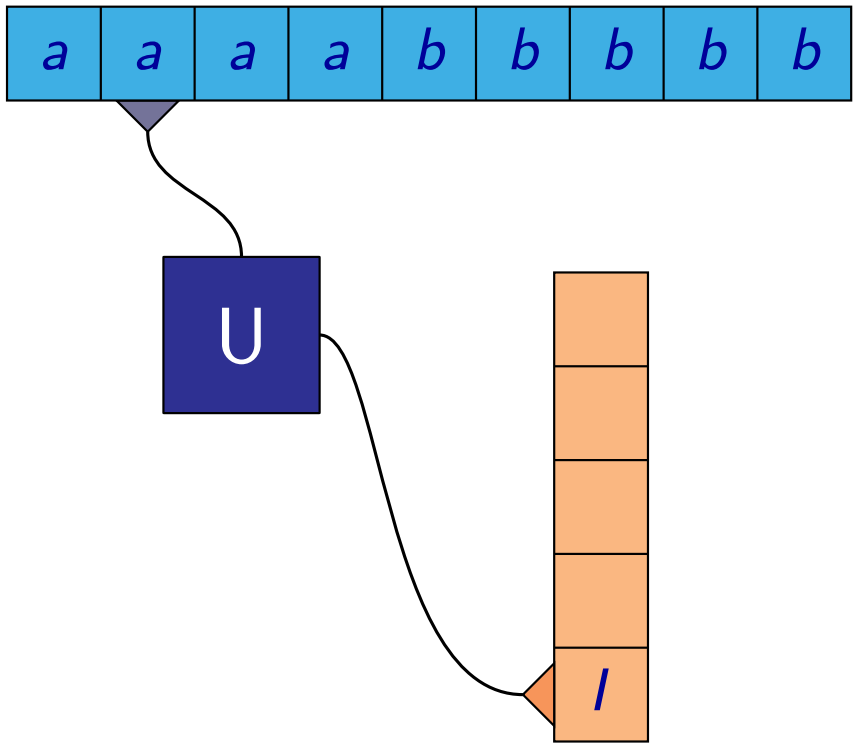
Zásobníkový automat

- Slovo *aaaabbbbb* nepatří do jazyka $L = \{a^i b^i \mid i \geq 1\}$



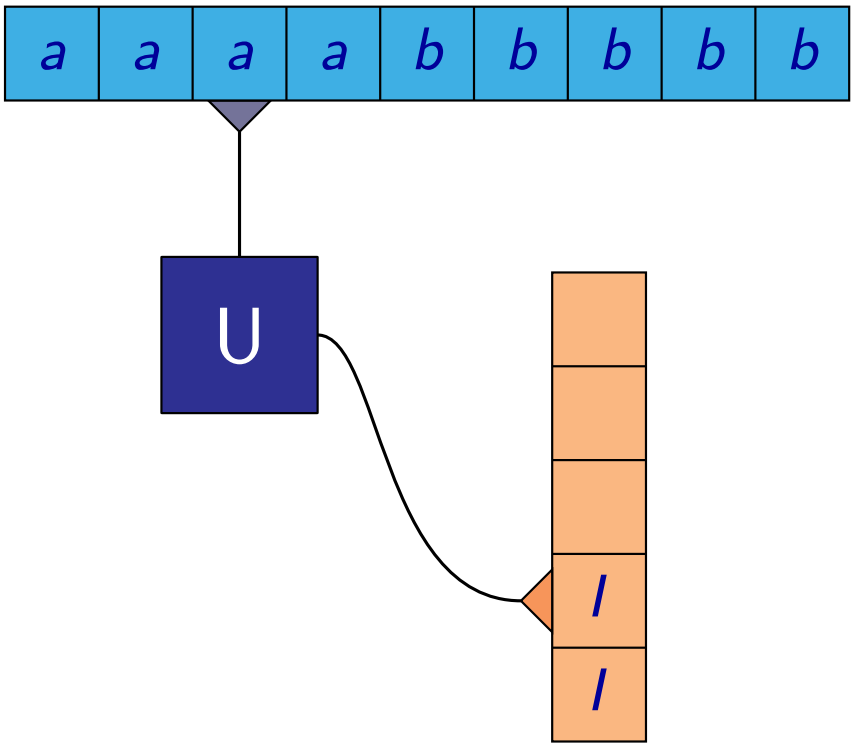
Zásobníkový automat

- Slovo *aaaabbbbb* nepatří do jazyka $L = \{a^i b^i \mid i \geq 1\}$



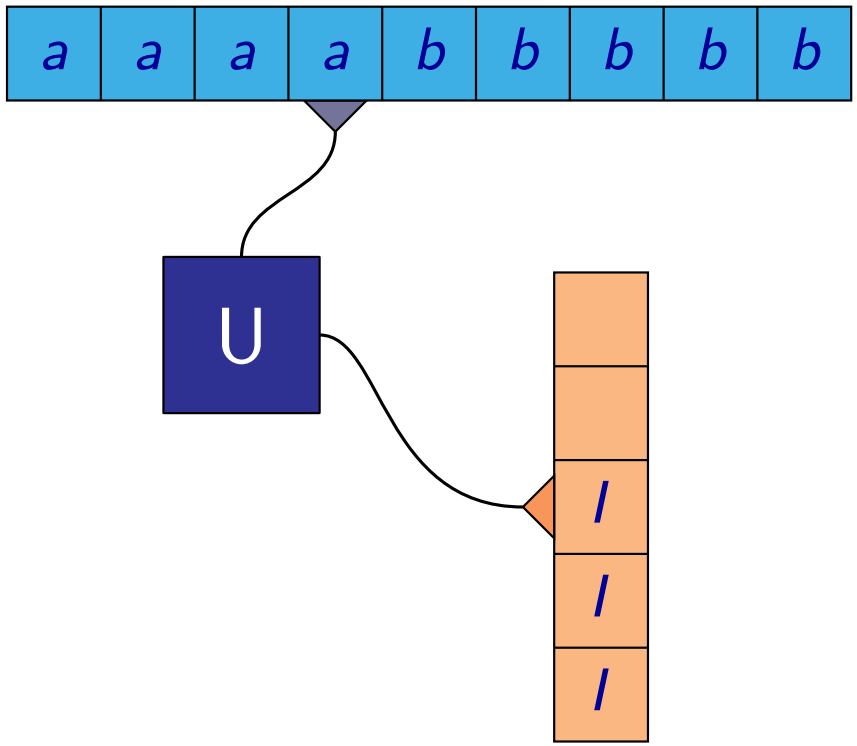
Zásobníkový automat

- Slovo *aaaabbbbb* nepatří do jazyka $L = \{a^i b^i \mid i \geq 1\}$



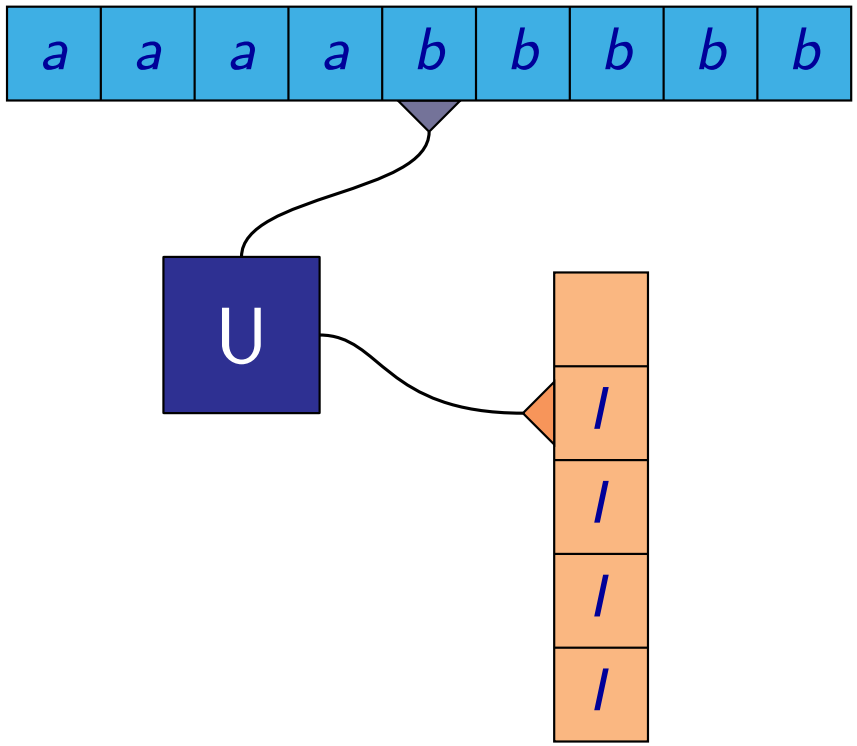
Zásobníkový automat

- Slovo *aaaabbbbb* nepatří do jazyka $L = \{a^i b^i \mid i \geq 1\}$



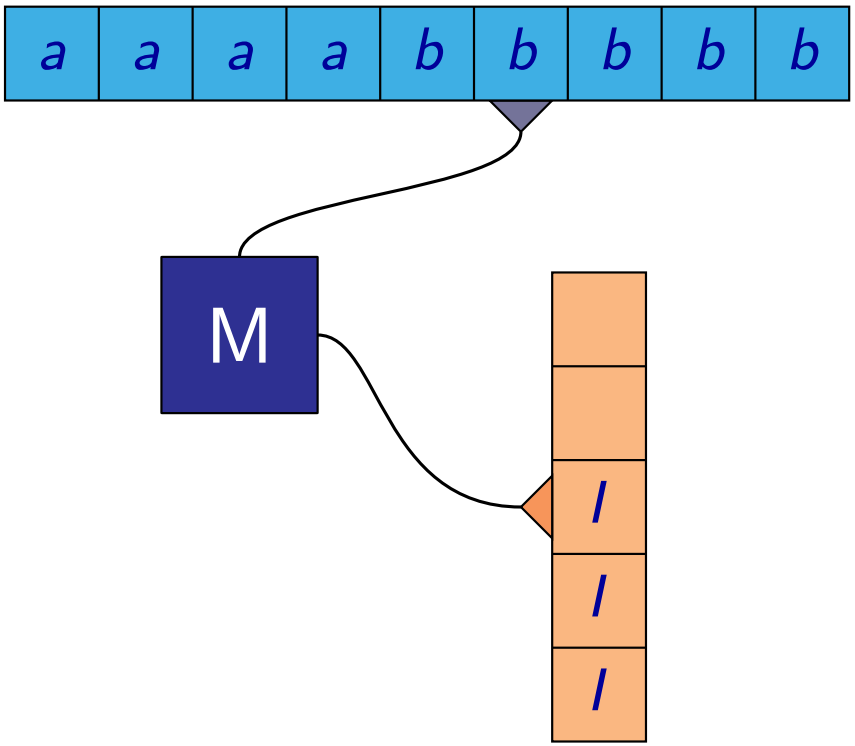
Zásobníkový automat

- Slovo *aaaabbbbb* nepatří do jazyka $L = \{a^i b^i \mid i \geq 1\}$



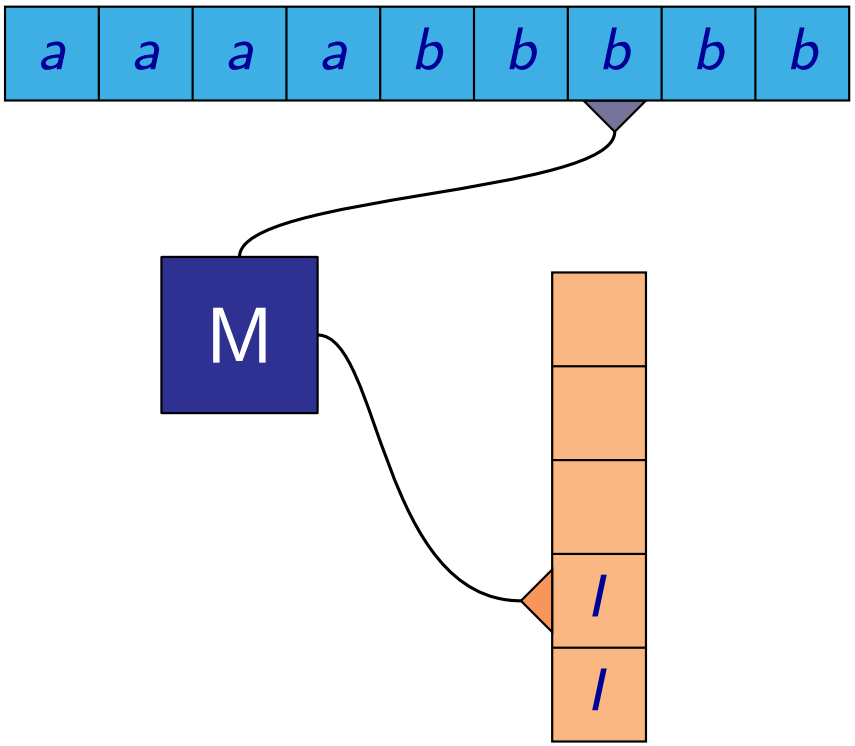
Zásobníkový automat

- Slovo *aaaabbbbb* nepatří do jazyka $L = \{a^i b^i \mid i \geq 1\}$



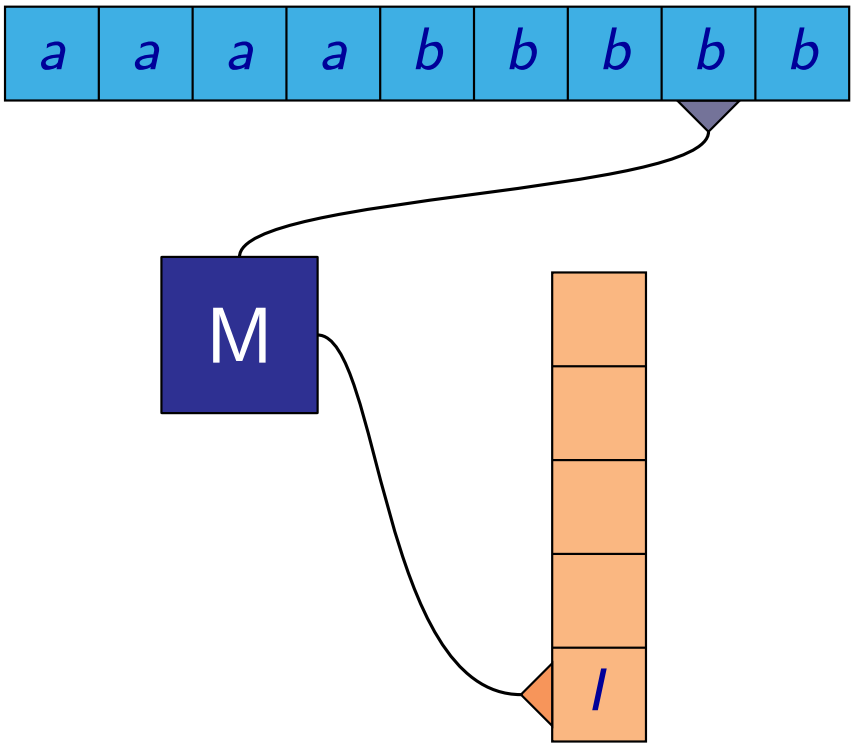
Zásobníkový automat

- Slovo *aaaabbbbb* nepatří do jazyka $L = \{a^i b^i \mid i \geq 1\}$



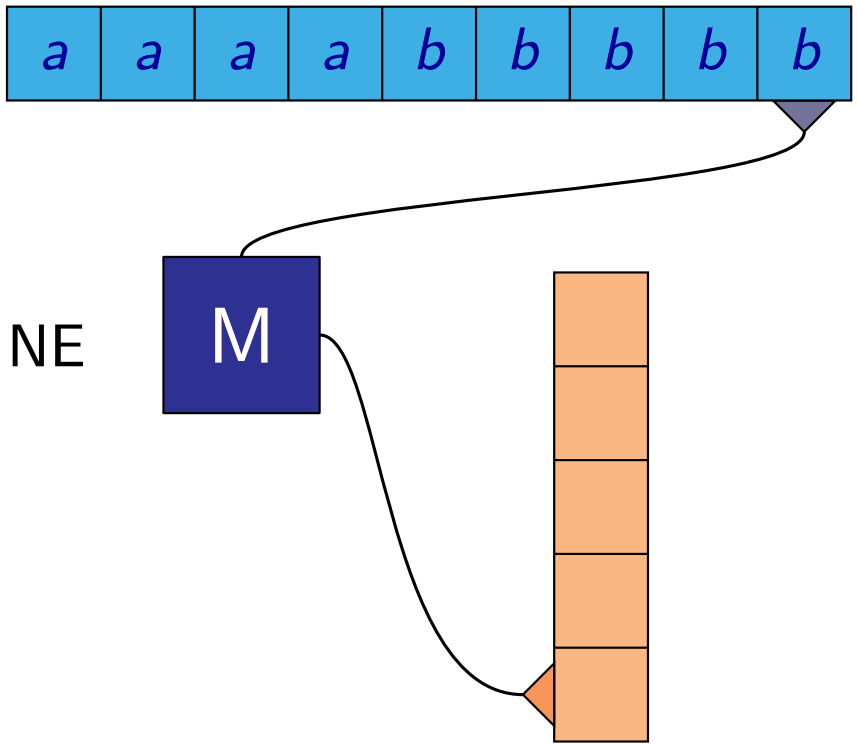
Zásobníkový automat

- Slovo *aaaabbbbb* nepatří do jazyka $L = \{a^i b^i \mid i \geq 1\}$



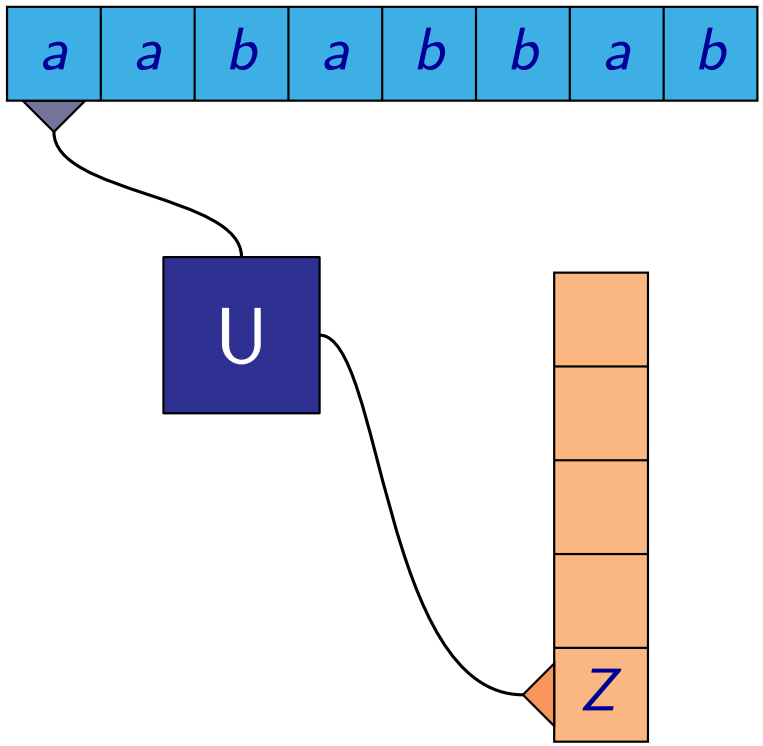
Zásobníkový automat

- Slovo *aaaabbbbb* 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



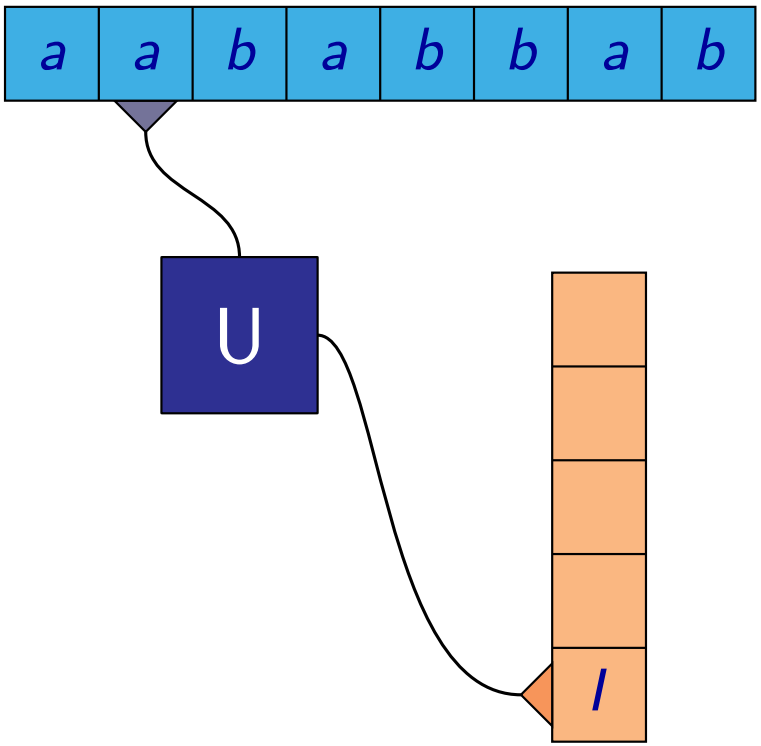
Zásobníkový automat

- Slovo *aababbab* nepatří do jazyka $L = \{a^i b^i \mid i \geq 1\}$



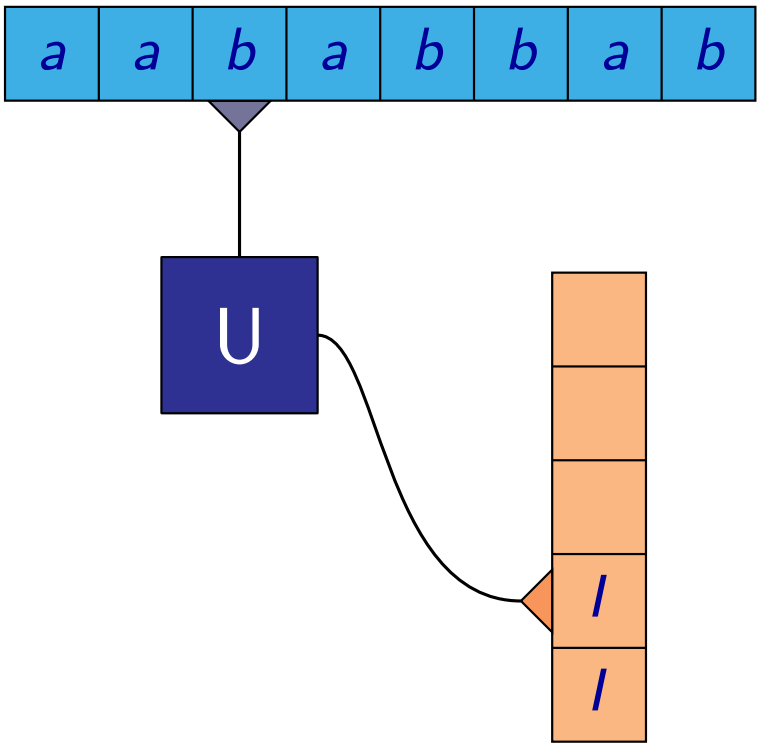
Zásobníkový automat

- Slovo *aababbab* nepatří do jazyka $L = \{a^i b^i \mid i \geq 1\}$



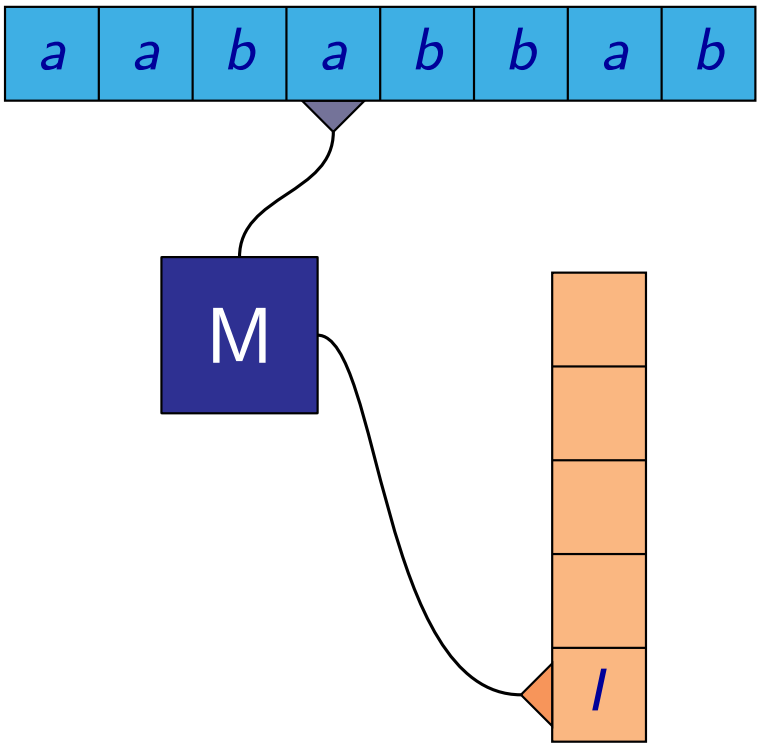
Zásobníkový automat

- Slovo *aababbbab* nepatří do jazyka $L = \{a^i b^i \mid i \geq 1\}$



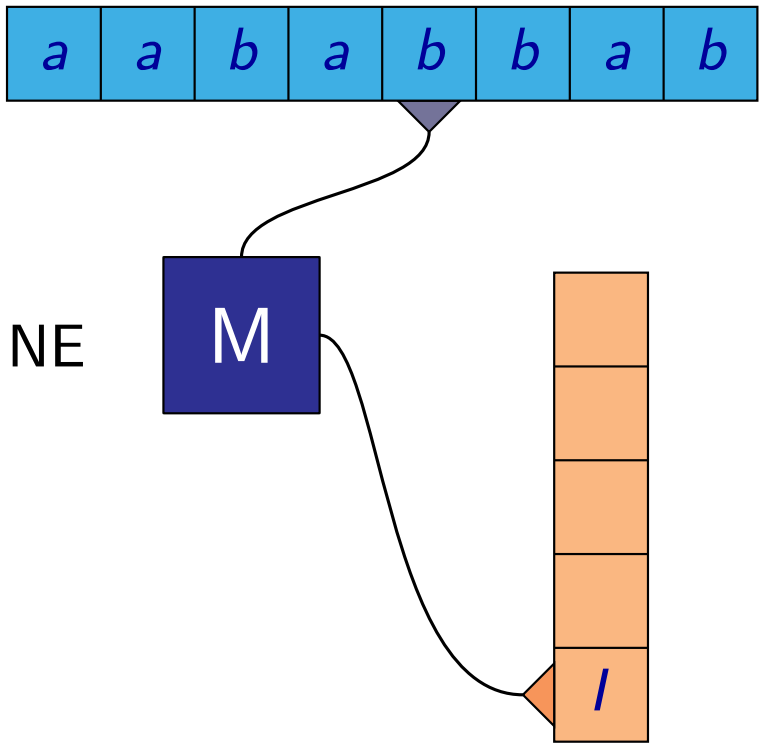
Zásobníkový automat

- Slovo *aababbab* nepatří do jazyka $L = \{a^i b^i \mid i \geq 1\}$



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

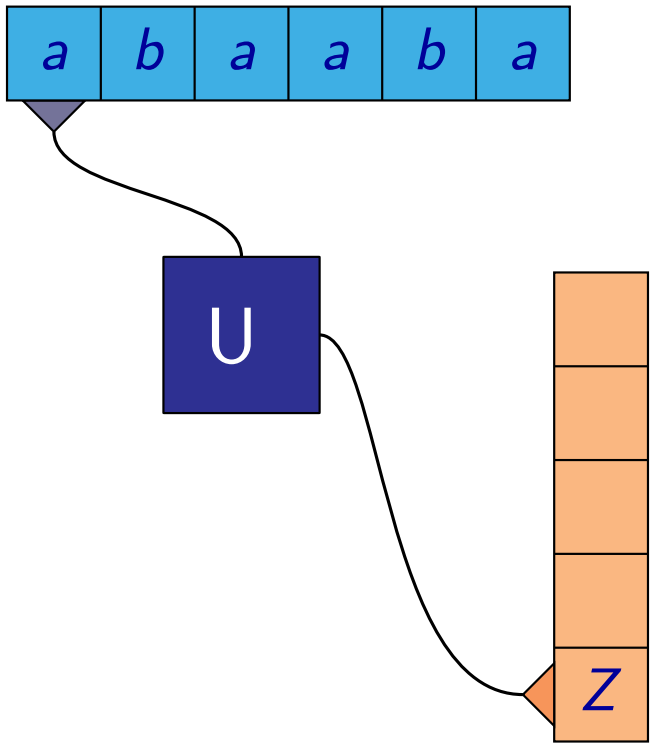
- Uvedený zásobníkový automat měl vždy jasně určeno pokračování - byl deterministický
- Je možné každý bezkontextový jazyk poznat deterministickým zásobníkovým automatem?

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

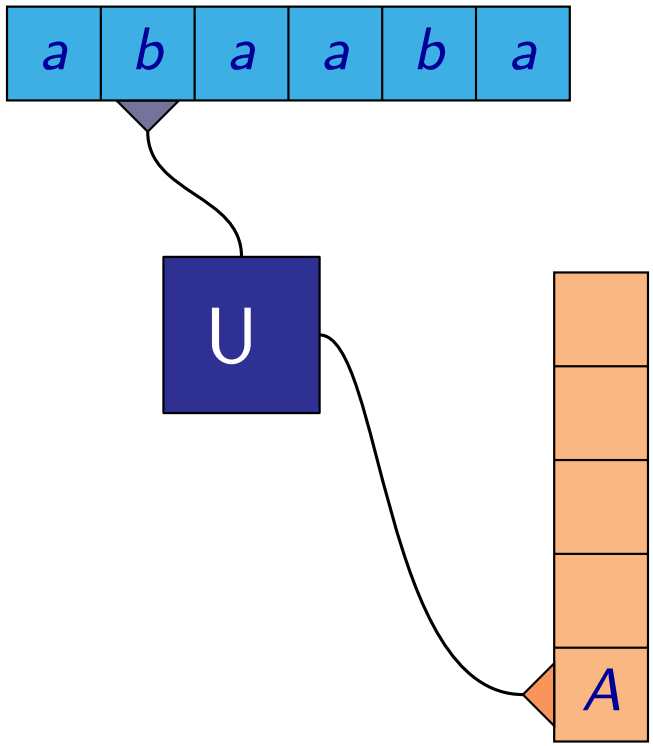
Zásobníkový automat

- Slovo *abaaba* patří do jazyka $L = \{w(w)^R \mid w \in \{a, b\}^*\}$



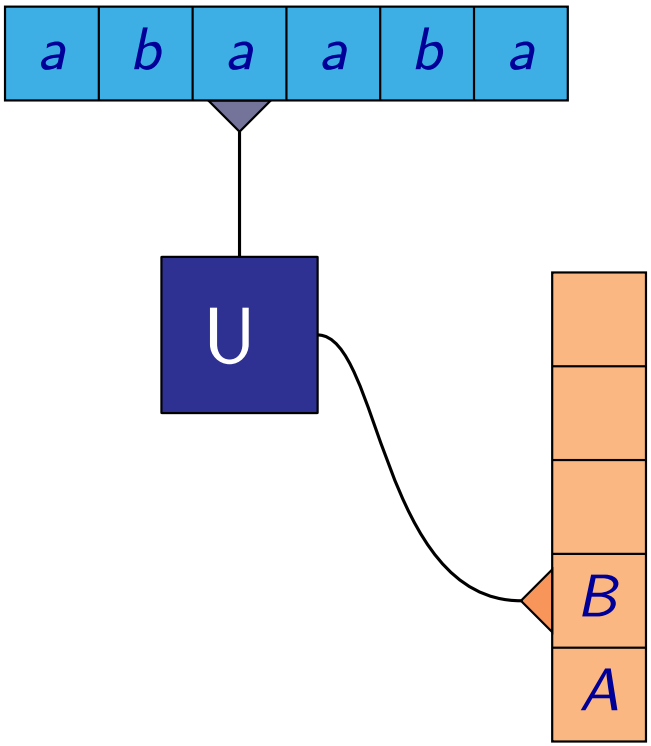
Zásobníkový automat

- Slovo *abaaba* patří do jazyka $L = \{w(w)^R \mid w \in \{a, b\}^*\}$



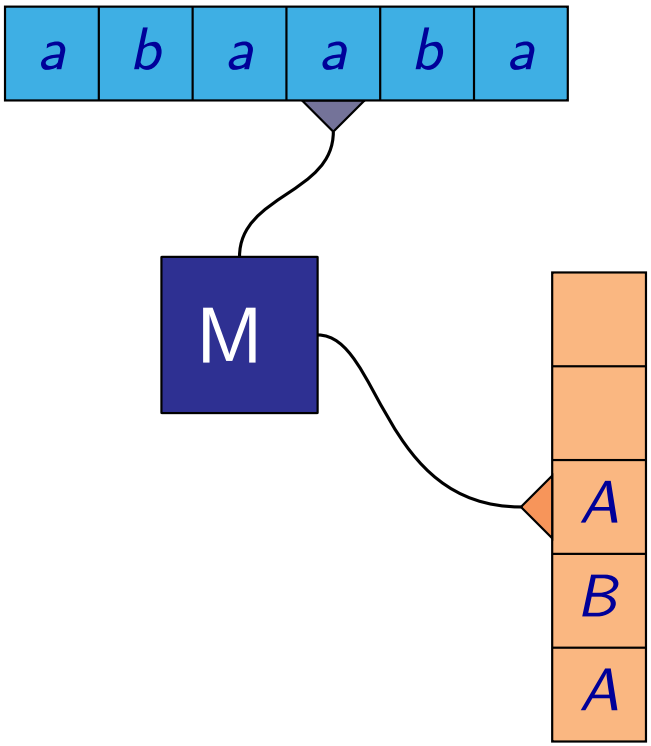
Zásobníkový automat

- 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



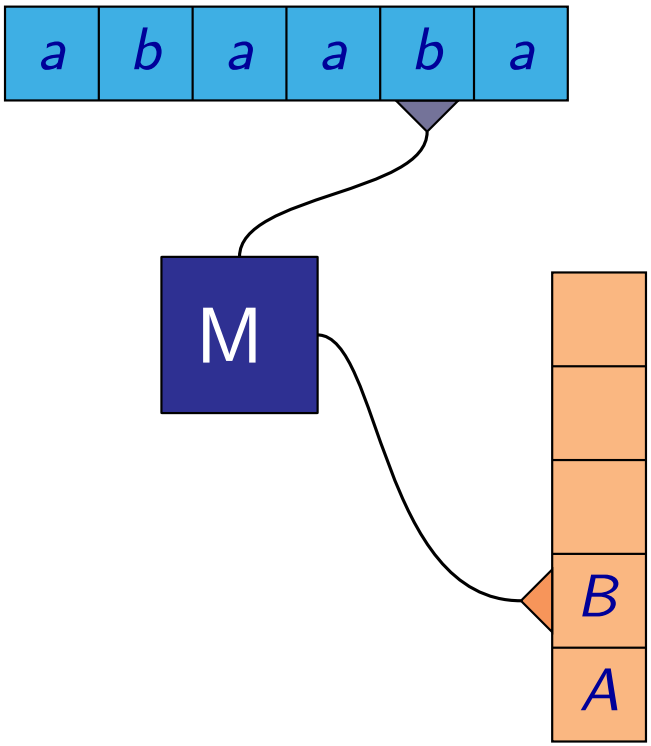
Zásobníkový automat

- 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



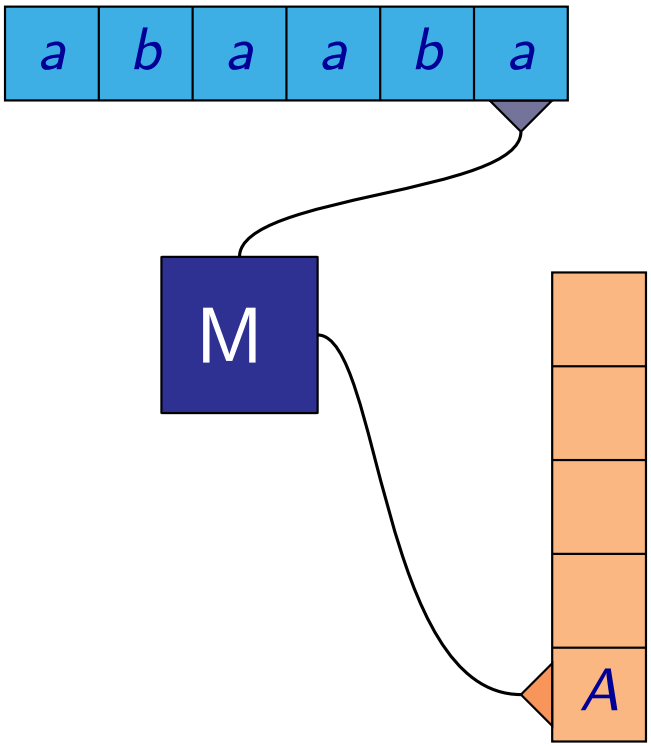
Zásobníkový automat

- 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



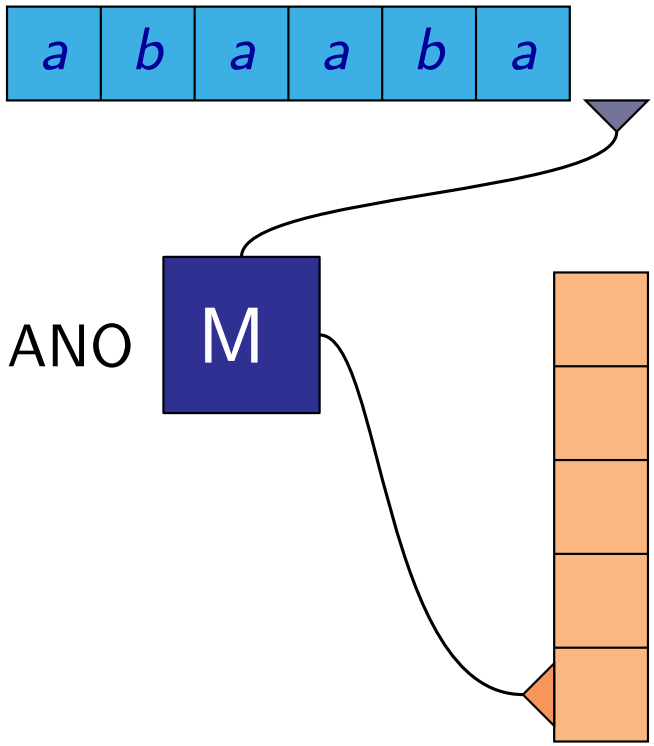
Zásobníkový automat

- 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



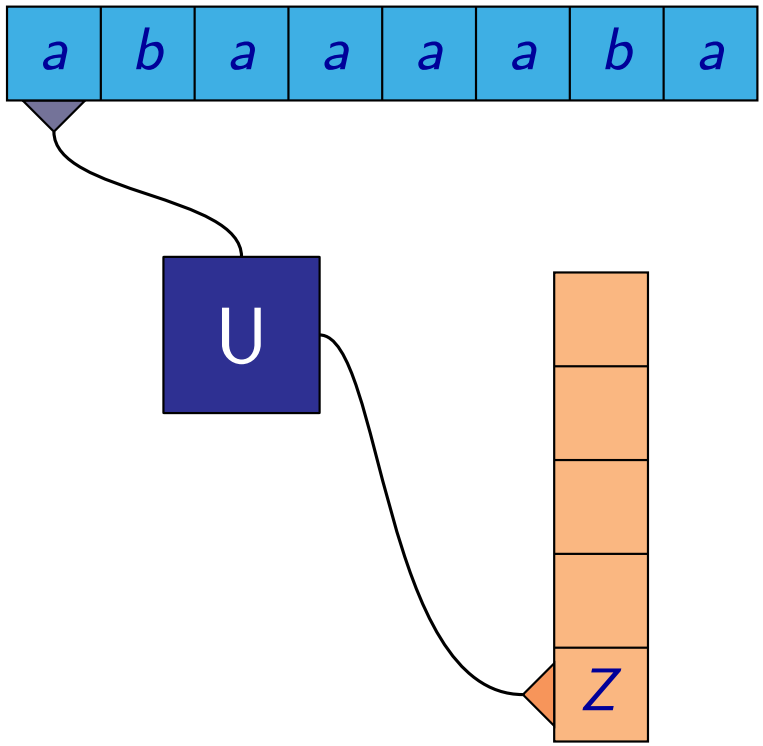
Zásobníkový automat

- 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



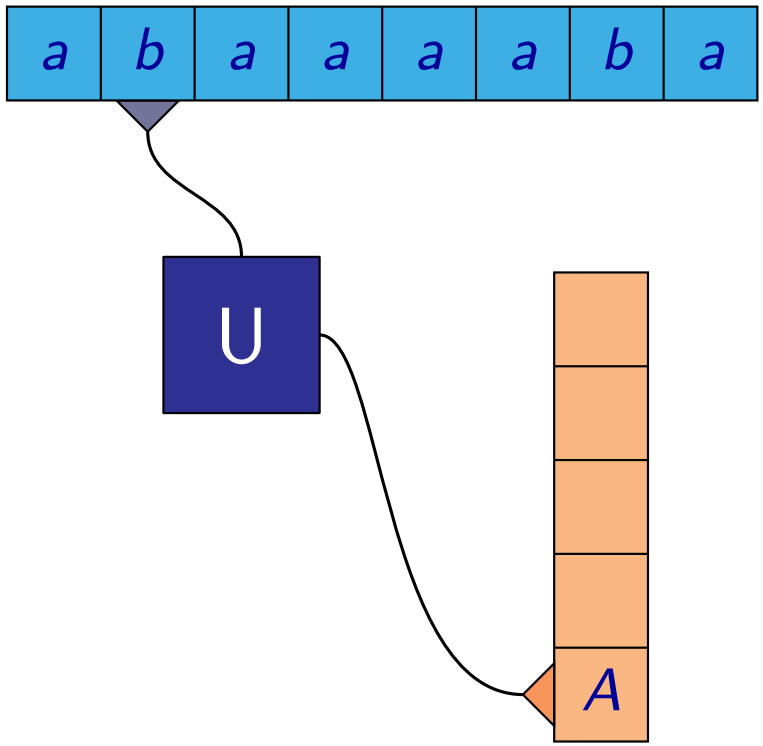
Zásobníkový automat

- Slovo *abaaaaba* patří do jazyka $L = \{w(w)^R \mid w \in \{a, b\}^*\}$



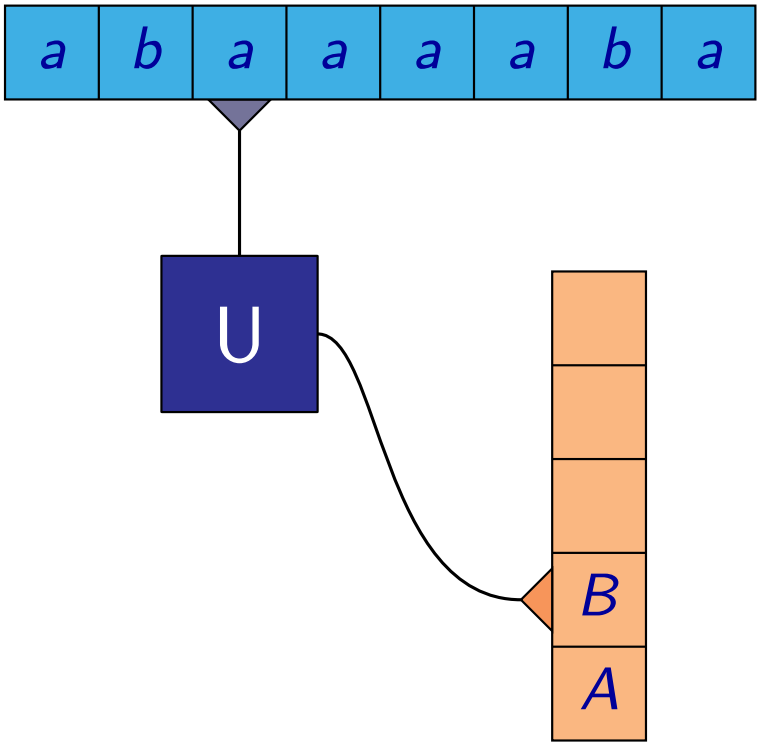
Zásobníkový automat

- Slovo *abaaaaba* patří do jazyka $L = \{w(w)^R \mid w \in \{a, b\}^*\}$



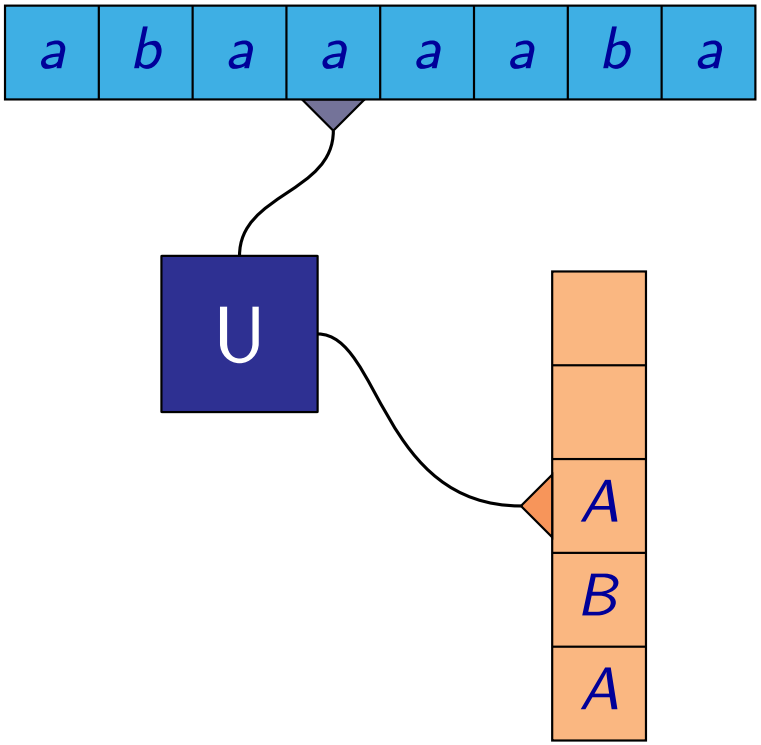
Zásobníkový automat

- 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



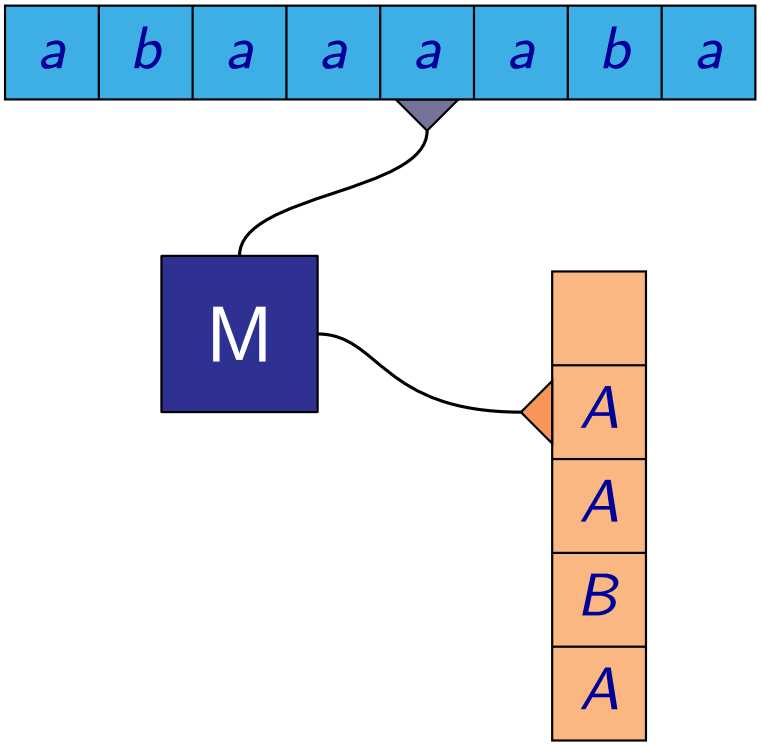
Zásobníkový automat

- 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



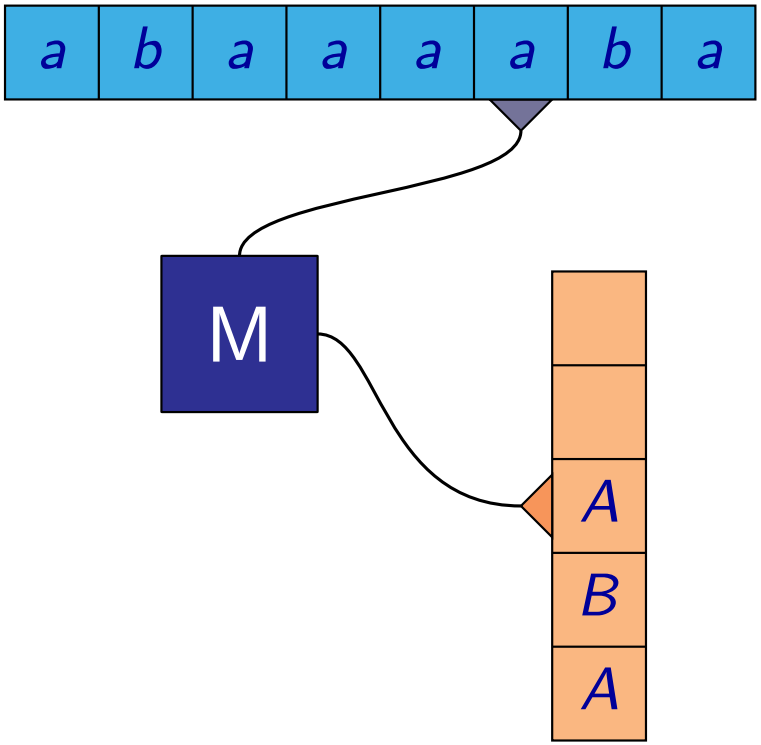
Zásobníkový automat

- 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



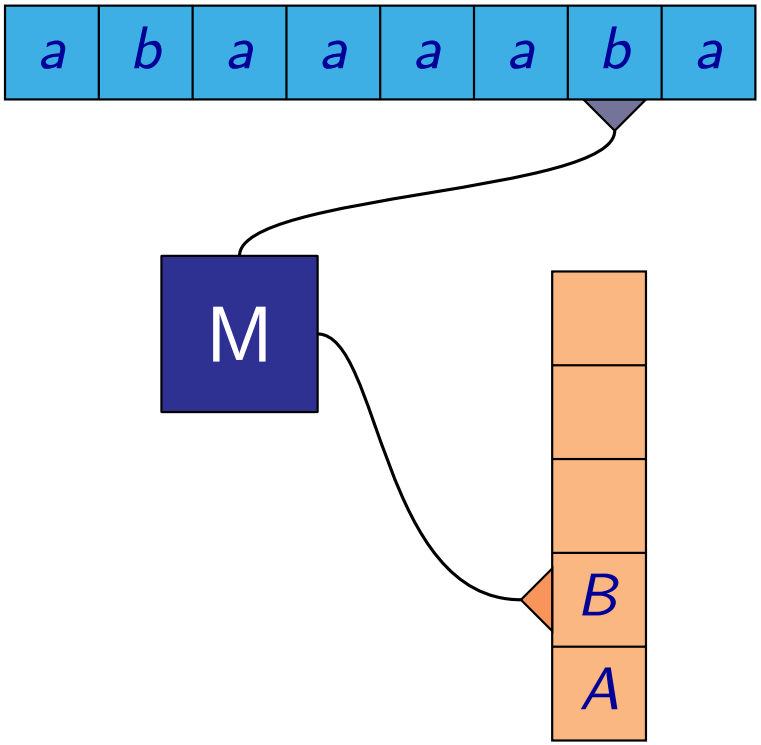
Zásobníkový automat

- Slovo *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



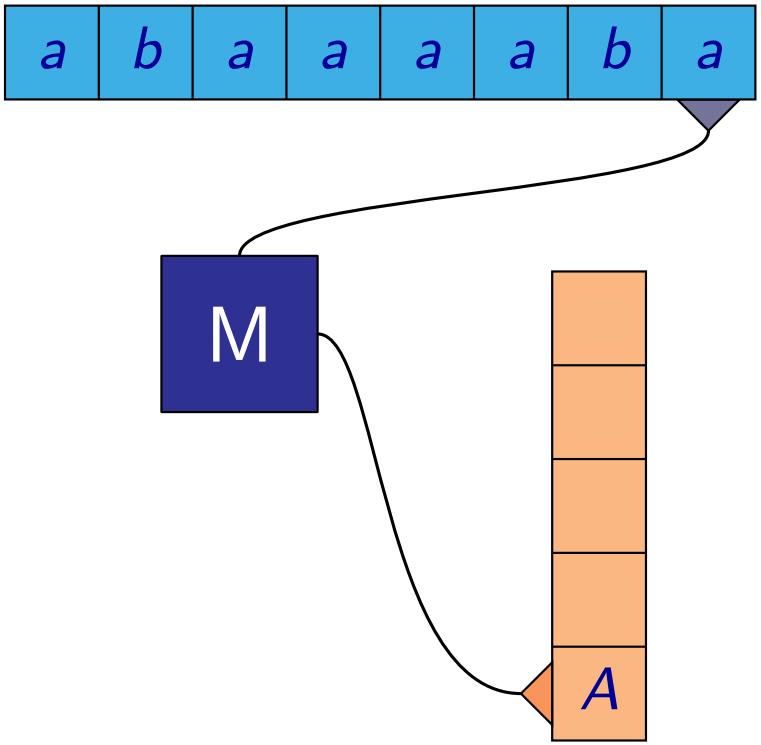
Zásobníkový automat

- 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



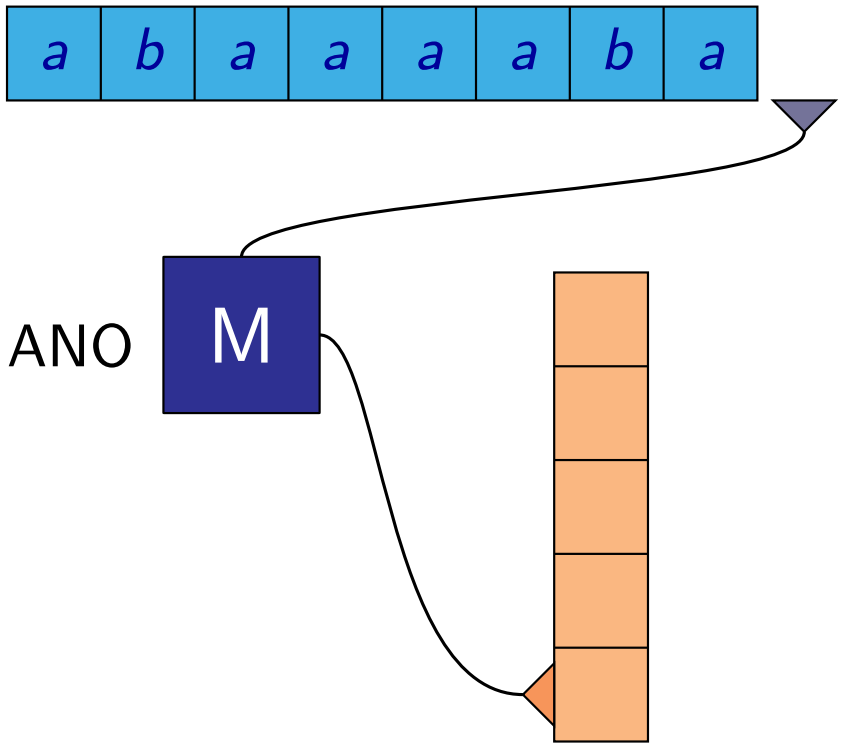
Zásobníkový automat

- 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 *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

- 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

Zásobníkový automat

Definice

Zásobníkový automat je uspořádaná šestice $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, Z_0)$, kde

- Q je konečná neprázdná množina stavů
- Σ je konečná neprázdná množina zvaná vstupní abeceda
- Γ je konečná neprázdná množina zvaná zásobníková abeceda
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to 2^{(Q \times \Gamma^*)}$ 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, α), 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**

Zásobníkový automat

- **Konfigurace ZA** je trojice (q, w, α), 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 $\vdash$

Zásobníkový automat

- **Konfigurace ZA** je trojice (q, w, α), 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 $L(\mathcal{M}) = \{w \in \Sigma^* \mid w \text{ je přijímáno ZA } \mathcal{M}\}$.

Zásobníkový automat

- 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 $L_{KS}(\mathcal{M}) = \{w \in \Sigma^* \mid (q_0, w, Z_0) \vdash^* (q, \varepsilon, \alpha)$ pro něj. $q \in F, \alpha \in \Gamma^*\}$
- Obě zmíněné definice přijímání ZA jsou ekvivalentní - jazyk je přijímán nějakým ZA $\mathcal{M}$ koncovým stavem právě tehdy, když je rozpoznáván nějakým ZA $\mathcal{M}'$ prázdným zásobníkem

Zásobníkový automat

Příklad: $L = \{a^i b^i \mid i \geq 1\}$
$\mathcal{M} = (Q, \Sigma, \Gamma, \delta, U, Z)$, kde

- $Q = \{U, M\}$
- $\Sigma = \{a, b\}$
- $\Gamma = \{Z, I\}$
- $$\begin{array}{ll} \delta(U, a, Z) = \{(U, I)\} & \delta(U, b, Z) = \emptyset \\ \delta(U, a, I) = \{(U, II)\} & \delta(U, b, I) = \{(M, \varepsilon)\} \\ \delta(M, a, I) = \emptyset & \delta(M, b, I) = \{(M, \varepsilon)\} \\ \delta(M, a, Z) = \emptyset & \delta(M, b, Z) = \emptyset \end{array}$$

Poznámka: Často se uvádí jen ty přechody přechodové funkce, které nejsou do prázdné množiny, tedy kdy je skutečně nějaký přechod definován

Zásobníkový automat

Příklad: $L = \{w(w)^R \mid w \in \{a,b\}^*\}$
$\mathcal{M} = (Q, \Sigma, \Gamma, \delta, U, Z)$, kde

- $Q = \{U, M\}$
- $\Sigma = \{a, b\}$
- $\Gamma = \{Z, A, B\}$
-

$$\begin{array}{ll}
\delta(U,a,Z) = \{(U,A)\} & \delta(U,b,Z) = \{(U,B)\} \\
\delta(U,a,A) = \{(U,AA),(M,AA)\} & \delta(U,b,A) = \{(U,BA),(M,BA)\} \\
\delta(U,a,B) = \{(U,AB),(M,AB)\} & \delta(U,b,B) = \{(U,BB),(M,BB)\} \\
\delta(M,a,A) = \{(M,\varepsilon)\} & \delta(M,b,A) = \emptyset \\
\delta(M,a,B) = \emptyset & \delta(M,b,B) = \{(M,\varepsilon)\} \\
\delta(M,a,Z) = \emptyset & \delta(M,b,Z) = \emptyset \\
\delta(U,\varepsilon,Z) = \{(U,\varepsilon)\} & \delta(M,\varepsilon,Z) = \emptyset \\
\delta(U,\varepsilon,A) = \emptyset & \delta(M,\varepsilon,A) = \emptyset \\
\delta(U,\varepsilon,B) = \emptyset & \delta(M,\varepsilon,B) = \emptyset
\end{array}$$

Ekvivalence bezkontextových gramatik a zás. automatů

Lemma

Ke každé bezkontextové gramatice G lze sestrojit 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 $X \in \Pi$: $\delta(q_0, \varepsilon, X) = \{(q_0, \alpha) \mid (X \to \alpha) \in P\}$,
- 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 $u \in \Sigma^*, \alpha \in \{\varepsilon\} \cup \Pi(\Pi \cup \Sigma)^*$

Ekvivalence bezkontextových gramatik a zás. automatů

Příklad: $L = \{a^i b^i \mid i \geq 1\}$
$G = (\{S\}, \{a, b\}, S, P)$, kde P je definována jako: $S \longrightarrow aSb|ab$

Ekvivalence bezkontextových gramatik a zás. automatů

Příklad: $L = \{a^i b^i \mid i \geq 1\}$
$G = (\{S\}, \{a, b\}, S, P)$, kde P je definována jako: $S \longrightarrow aSb|ab$

Sestrojíme $\mathcal{M} = (\{q_0\}, \{a, b\}, \{S, a, b\}, \delta, q_0, S)$, kde
$\delta(q_0, \varepsilon, S) = \{(q_0, aSb), (q_0, ab)\}$, $\delta(q_0, a, a) = \{(q_0, \varepsilon)\}$,
$\delta(q_0, b, b) = \{(q_0, \varepsilon)\}$

Ekvivalence bezkontextových gramatik a zás. automatů

Příklad: $L = \{a^i b^i \mid i \geq 1\}$
$G = (\{S\}, \{a, b\}, S, P)$, kde P je definována jako: $S \longrightarrow aSb|ab$

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 $aaaabbbb$
$S \Rightarrow aSb$
$(q_0, aaaabbbb, S) \vdash (q_0, aaaabbbb, aSb)$

Ekvivalence bezkontextových gramatik a zás. automatů

Příklad: $L = \{a^i b^i \mid i \geq 1\}$
$G = (\{S\}, \{a, b\}, S, P)$, kde P je definována jako: $S \longrightarrow aSb|ab$

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 $aaaabbbb$
$S \Rightarrow aSb$
$(q_0, aaaabbbb, S) \vdash (q_0, aaaabbbb, aSb) \vdash (q_0, aaabbbb, Sb)$

Ekvivalence bezkontextových gramatik a zás. automatů

Příklad: $L = \{a^i b^i \mid i \geq 1\}$
$G = (\{S\}, \{a, b\}, S, P)$, kde P je definována jako: $S \longrightarrow aSb|ab$

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 $aaaabbbb$
$S \Rightarrow aSb \Rightarrow aaSbb$
$(q_0, aaaabbbb, S) \vdash (q_0, aaaabbbb, aSb) \vdash (q_0, aaabbbb, Sb)$
$\vdash (q_0, aaabbbb, aSbb)$

Ekvivalence bezkontextových gramatik a zás. automatů

Příklad: $L = \{a^i b^i \mid i \geq 1\}$
$G = (\{S\}, \{a, b\}, S, P)$, kde P je definována jako: $S \longrightarrow aSb|ab$

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 $aaaabbbb$
$S \Rightarrow aSb \Rightarrow aaSbb$
$(q_0, aaaabbbb, S) \vdash (q_0, aaaabbbb, aSb) \vdash (q_0, aaabbbb, Sb)$
$\vdash (q_0, aaabbbb, aSbb) \vdash (q_0, aabbbb, Sbb)$

Ekvivalence bezkontextových gramatik a zás. automatů

Příklad: $L = \{a^i b^i \mid i \geq 1\}$
$G = (\{S\}, \{a, b\}, S, P)$, kde P je definována jako: $S \longrightarrow aSb|ab$

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 $aaaabbbb$
$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb$
$(q_0, aaaabbbb, S) \vdash (q_0, aaaabbbb, aSb) \vdash (q_0, aaabbbb, Sb)$
$\vdash (q_0, aaabbbb, aSbb) \vdash (q_0, aabbbb, Sbb) \vdash (q_0, aabbbb, aSbbb)$

Ekvivalence bezkontextových gramatik a zás. automatů

Příklad: $L = \{a^i b^i \mid i \geq 1\}$
$G = (\{S\}, \{a, b\}, S, P)$, kde P je definována jako: $S \longrightarrow aSb|ab$

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 $aaaabbbb$
$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb$
$(q_0, aaaabbbb, S) \vdash (q_0, aaaabbbb, aSb) \vdash (q_0, aaabbbb, Sb)$
$\vdash (q_0, aaabbbb, aSbb) \vdash (q_0, aabbbb, Sbb) \vdash (q_0, aabbbb, aSbbb)$
$\vdash (q_0, abbbb, Sbbb)$

Ekvivalence bezkontextových gramatik a zás. automatů

Příklad: $L = \{a^i b^i \mid i \geq 1\}$
$G = (\{S\}, \{a, b\}, S, P)$, kde P je definována jako: $S \longrightarrow aSb|ab$

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 $aaaabbbb$
$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaaabbbb$
$(q_0, aaaabbbb, S) \vdash (q_0, aaaabbbb, aSb) \vdash (q_0, aaabbbb, Sb)$
$\vdash (q_0, aaabbbb, aSbb) \vdash (q_0, aabbbb, Sbb) \vdash (q_0, aabbbb, aSbbb)$
$\vdash (q_0, abbbb, Sbbb) \vdash (q_0, abbbb, abbbb)$

Ekvivalence bezkontextových gramatik a zás. automatů

Příklad: $L = \{a^i b^i \mid i \geq 1\}$
$G = (\{S\}, \{a, b\}, S, P)$, kde P je definována jako: $S \longrightarrow aSb|ab$

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 $aaaabbbb$
$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaaabbbb$
$(q_0, aaaabbbb, S) \vdash (q_0, aaaabbbb, aSb) \vdash (q_0, aaabbbb, Sb)$
$\vdash (q_0, aaabbbb, aSbb) \vdash (q_0, aabbbb, Sbb) \vdash (q_0, aabbbb, aSbbb)$
$\vdash (q_0, abbbb, Sbbb) \vdash (q_0, abbbb, abbbb) \vdash (q_0, bbbb, bbbb)$

Ekvivalence bezkontextových gramatik a zás. automatů

Příklad: $L = \{a^i b^i \mid i \geq 1\}$
$G = (\{S\}, \{a, b\}, S, P)$, kde P je definována jako: $S \longrightarrow aSb|ab$

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 $aaaabbbb$
$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaaabbbb$
$(q_0, aaaabbbb, S) \vdash (q_0, aaaabbbb, aSb) \vdash (q_0, aaabbbb, Sb)$
$\vdash (q_0, aaabbbb, aSbb) \vdash (q_0, aabbbb, Sbb) \vdash (q_0, aabbbb, aSbbb)$
$\vdash (q_0, abbbb, Sbbb) \vdash (q_0, abbbb, abbbb) \vdash (q_0, bbbb, bbbb)$
$\vdash (q_0, bbb, bbb) \vdash (q_0, bb, bb) \vdash (q_0, b, b) \vdash (q_0, \varepsilon, \varepsilon)$

Ekvivalence bezkontextových gramatik a zás. automatů

Lemma

Ke každému zásobníkovému automatu $\mathcal{M}$ s jedním stavem lze sestrojit 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 $(A \to a\alpha) \in P \Leftrightarrow \delta(q_0, a, A) \ni (q_0, \alpha)$, 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 $u \in \Sigma^*, \alpha \in \Gamma^*$

Ekvivalence bezkontextových gramatik a zás. automatů

Lemma

Ke každému zásobníkovému automatu $\mathcal{M}$ lze sestrojit 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 nárust 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 rozpoznávaní takových jazyků
  - Turingův stroj
  - Lineárně omezený automat

Turingův stroj

- Rozšíříme deterministické konečné automaty o možnost zápisu na vstupní pásku a pohyb čtecí hlavy oběma směry

Turingův stroj

- 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**
- Γ je konečná množina **páskových symbolů**
- $\Sigma \subseteq \Gamma, \Sigma \neq \emptyset$ je konečná množina **vstupních symbolů**
- $\delta : (Q - F) \times \Gamma \to Q \times \Gamma \times \{-1, 0, +1\}$ 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 Σ
- 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

Turingův stroj

- 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

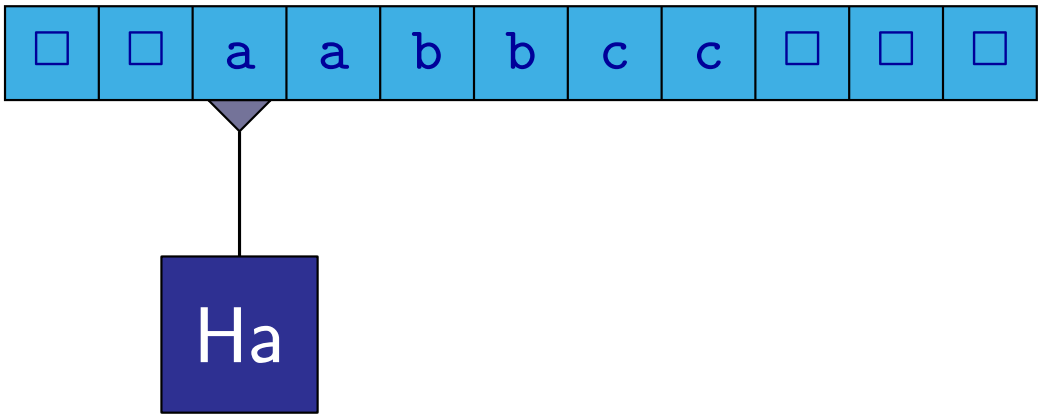
Turingův stroj

- 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

- 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.

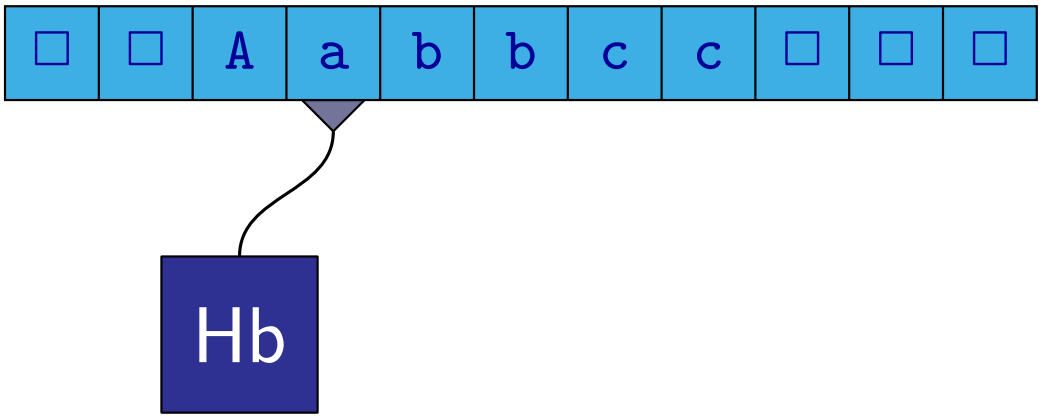
Turingův stroj



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.

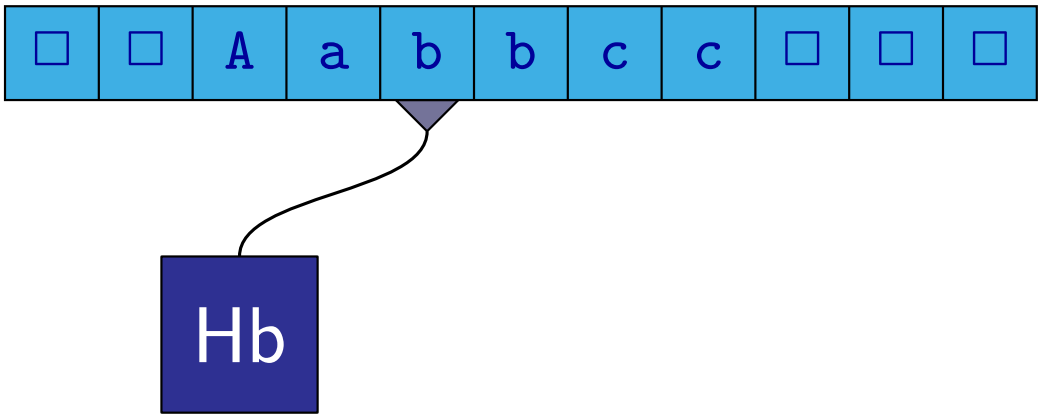
Turingův stroj



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 □ nebo c se zacyklí.

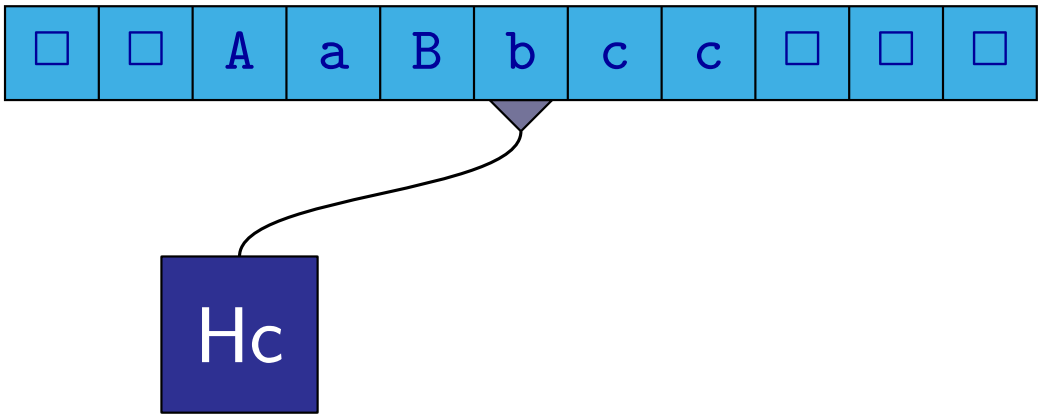
Turingův stroj



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 □ nebo c se zacyklí.

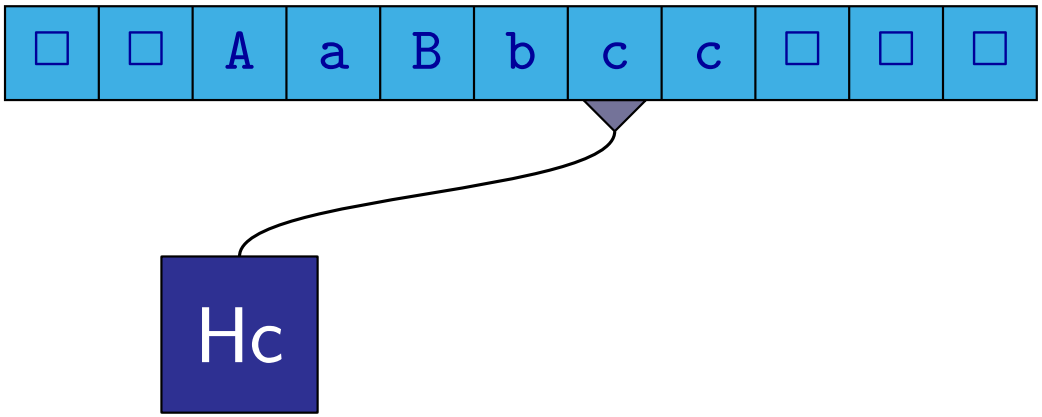
Turingův stroj



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 □ nebo c se zacyklí.
3. Čte do prvního c, nahradí jej C. Na □ se zacyklí.

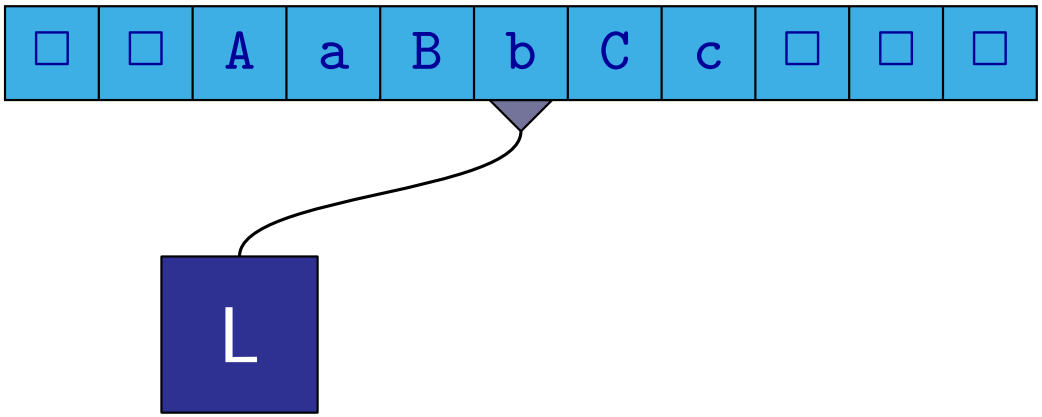
Turingův stroj



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 □ nebo c se zacyklí.
3. Čte do prvního c, nahradí jej C. Na □ se zacyklí.

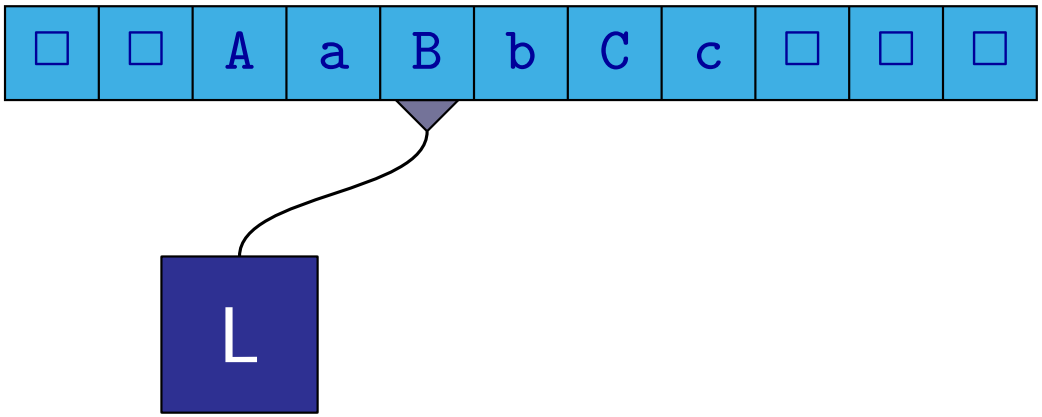
Turingův stroj



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 □ nebo c se zacyklí.
3. Čte do prvního c, nahradí jej C. Na □ se zacyklí.
4. Vrací se doleva na nejbližší A.

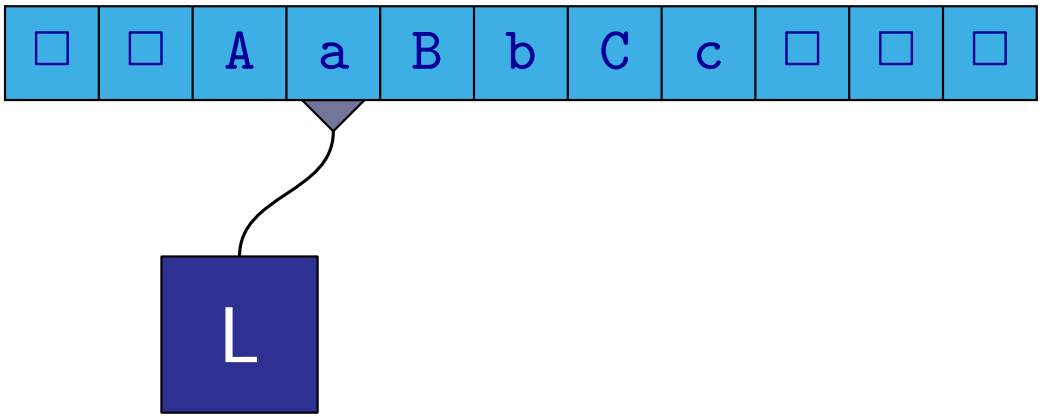
Turingův stroj



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 □ nebo c se zacyklí.
3. Čte do prvního c, nahradí jej C. Na □ se zacyklí.
4. Vrací se doleva na nejbližší A.

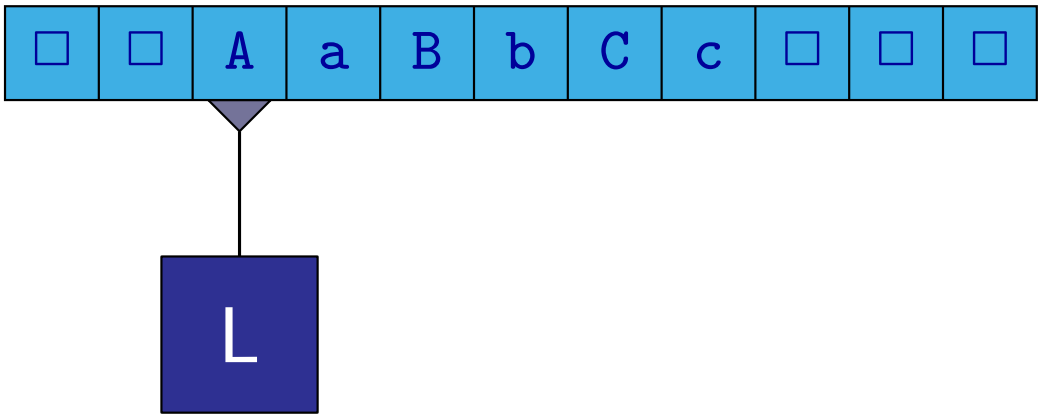
Turingův stroj



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 □ nebo c se zacyklí.
3. Čte do prvního c, nahradí jej C. Na □ se zacyklí.
4. Vrací se doleva na nejbližší A.

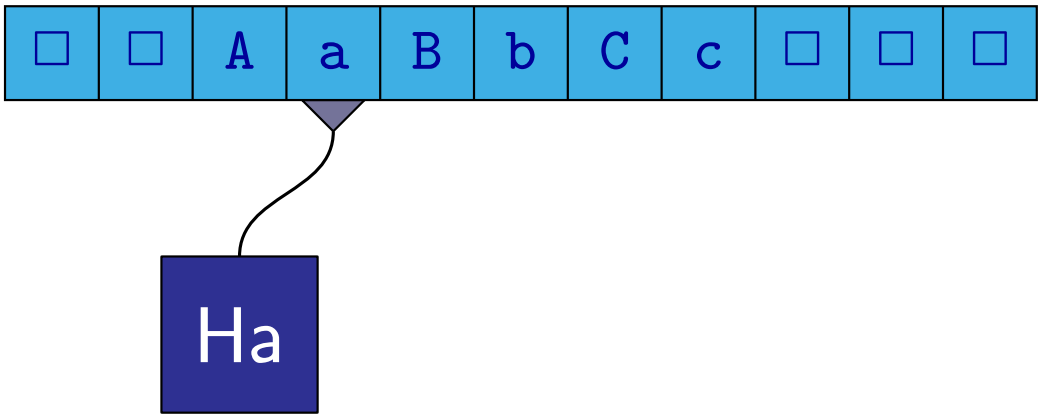
Turingův stroj



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 □ nebo c se zacyklí.
3. Čte do prvního c, nahradí jej C. Na □ se zacyklí.
4. Vrací se doleva na nejbližší A.

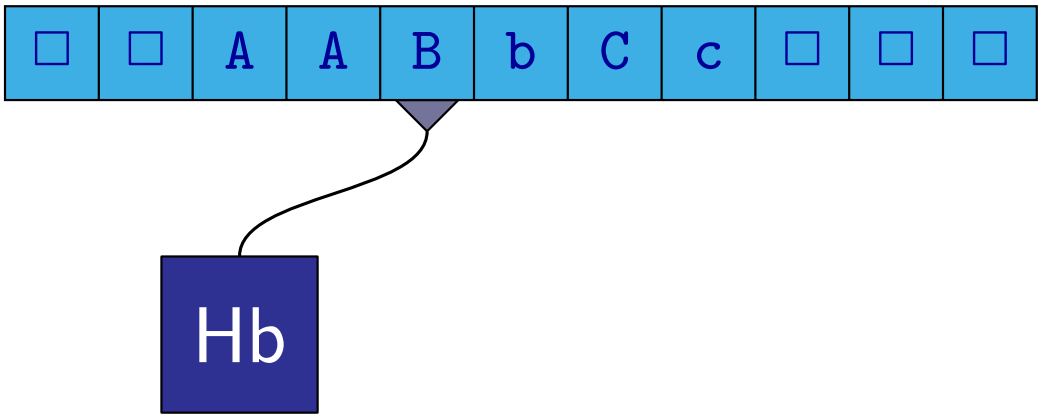
Turingův stroj



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 □ nebo c se zacyklí.
3. Čte do prvního c, nahradí jej C. Na □ se zacyklí.
4. Vrací se doleva na nejbližší A.
5. Pokračuje bodem 1

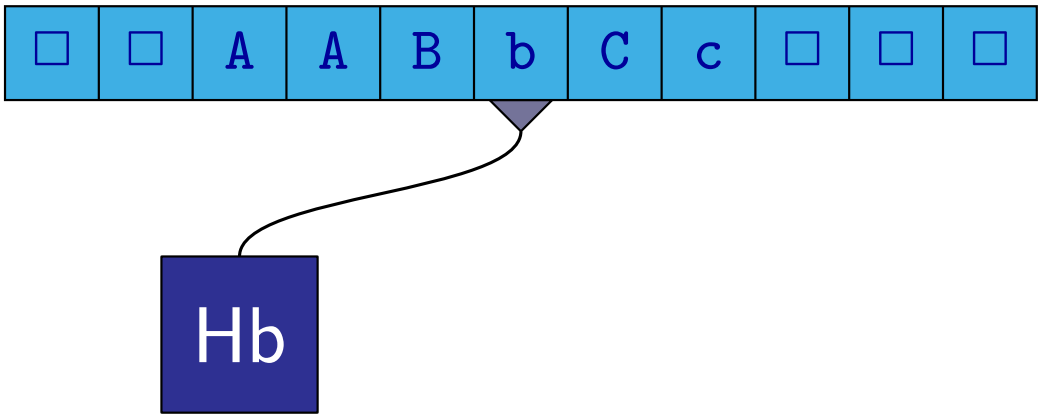
Turingův stroj



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 □ nebo c se zacyklí.
3. Čte do prvního c, nahradí jej C. Na □ se zacyklí.
4. Vrací se doleva na nejbližší A.
5. Pokračuje bodem 1

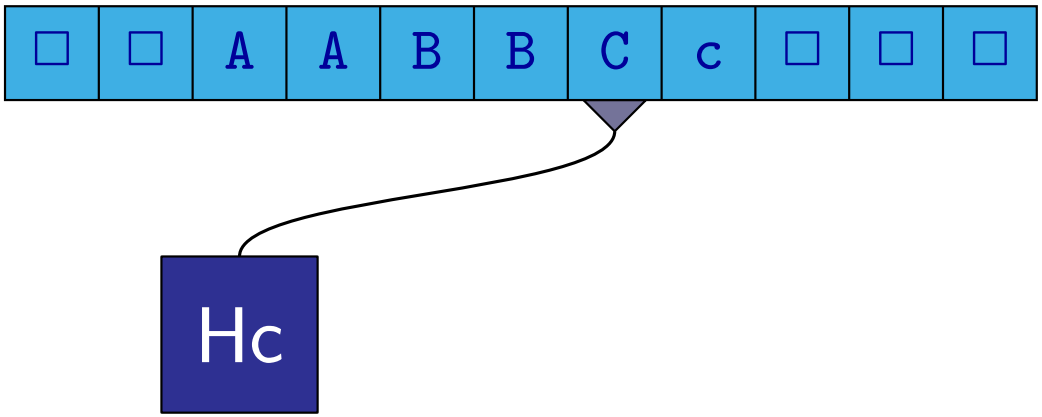
Turingův stroj



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 □ nebo c se zacyklí.
3. Čte do prvního c, nahradí jej C. Na □ se zacyklí.
4. Vrací se doleva na nejbližší A.
5. Pokračuje bodem 1

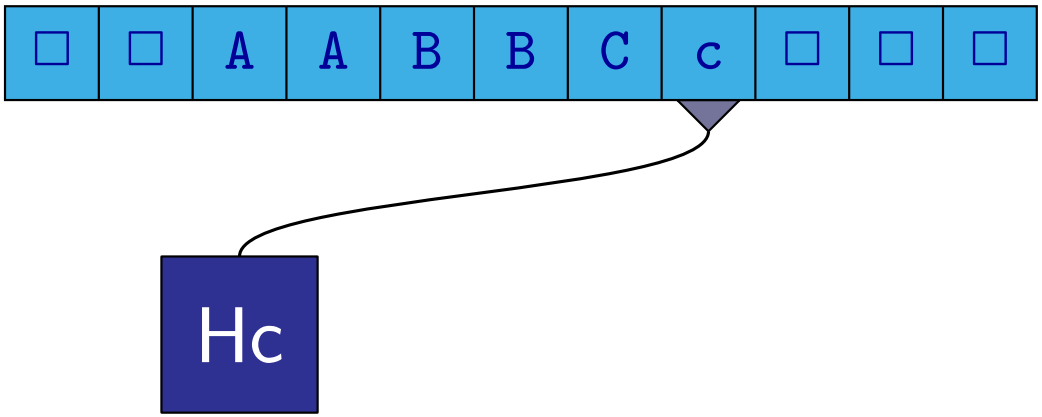
Turingův stroj



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 □ nebo c se zacyklí.
3. Čte do prvního c, nahradí jej C. Na □ se zacyklí.
4. Vrací se doleva na nejbližší A.
5. Pokračuje bodem 1

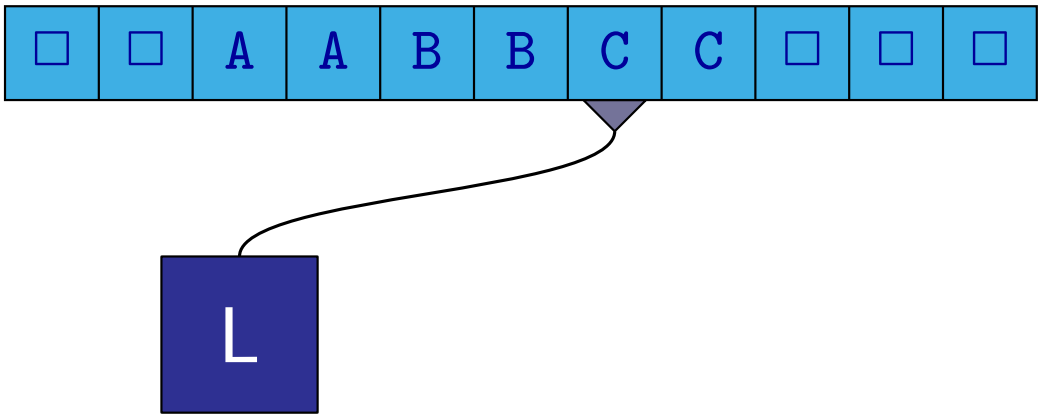
Turingův stroj



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 □ nebo c se zacyklí.
3. Čte do prvního c, nahradí jej C. Na □ se zacyklí.
4. Vrací se doleva na nejbližší A.
5. Pokračuje bodem 1

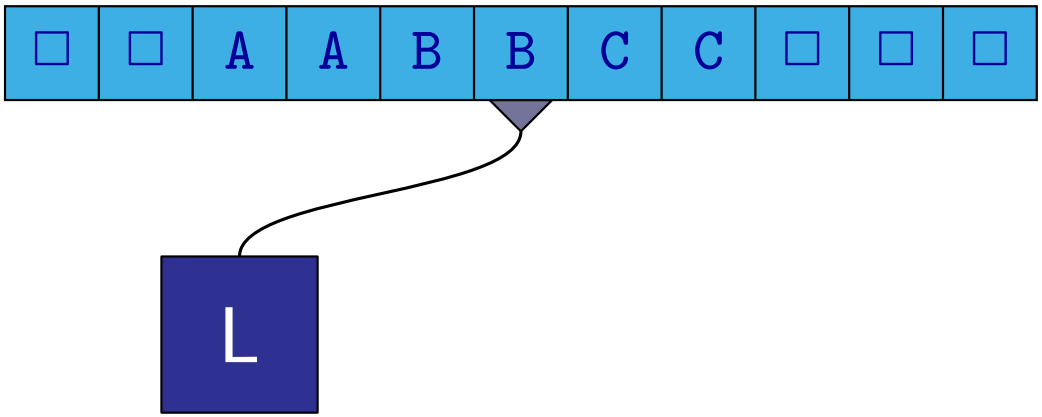
Turingův stroj



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 □ nebo c se zacyklí.
3. Čte do prvního c, nahradí jej C. Na □ se zacyklí.
4. Vrací se doleva na nejbližší A.
5. Pokračuje bodem 1

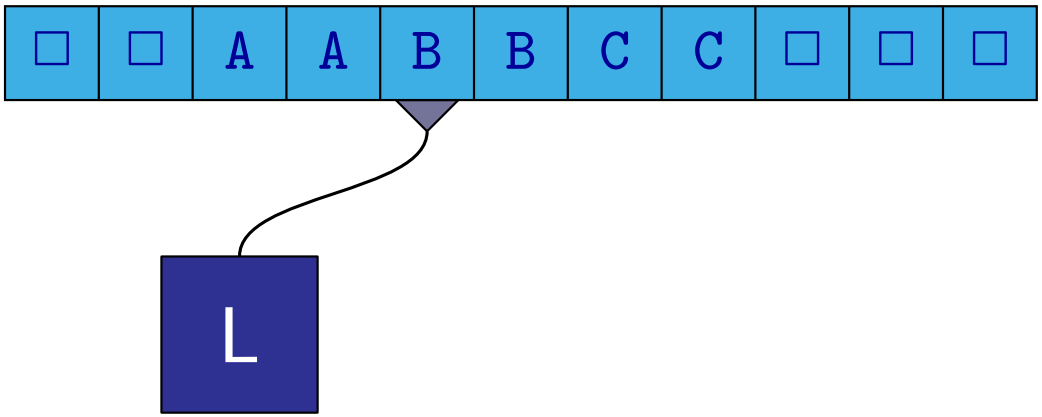
Turingův stroj



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 □ nebo c se zacyklí.
3. Čte do prvního c, nahradí jej C. Na □ se zacyklí.
4. Vrací se doleva na nejbližší A.
5. Pokračuje bodem 1

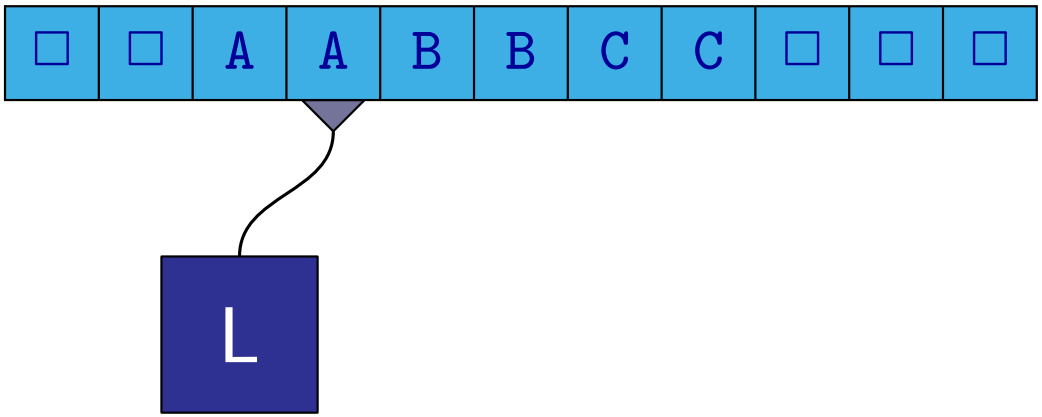
Turingův stroj



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 □ nebo c se zacyklí.
3. Čte do prvního c, nahradí jej C. Na □ se zacyklí.
4. Vrací se doleva na nejbližší A.
5. Pokračuje bodem 1

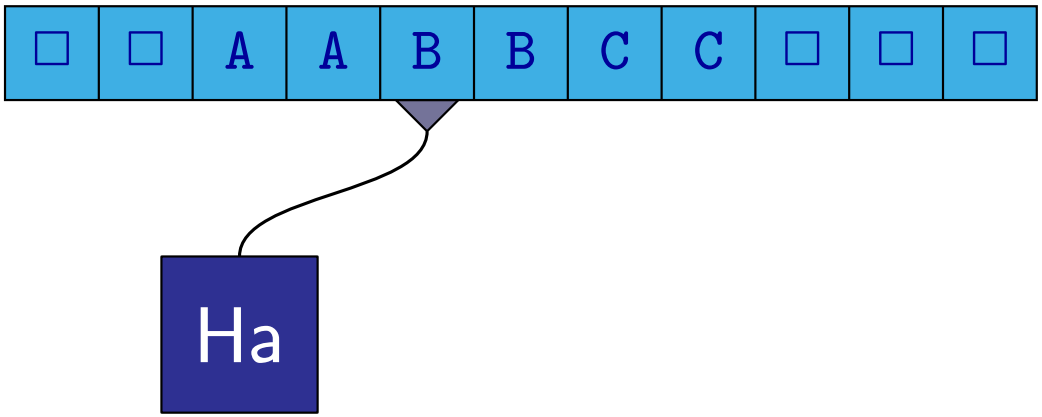
Turingův stroj



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 □ nebo c se zacyklí.
3. Čte do prvního c, nahradí jej C. Na □ se zacyklí.
4. Vrací se doleva na nejbližší A.
5. Pokračuje bodem 1

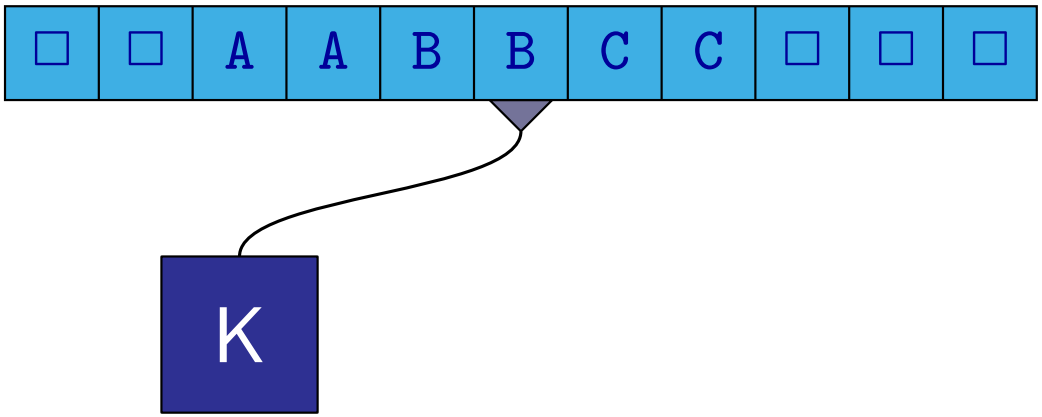
Turingův stroj



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 □ nebo c se zacyklí.
3. Čte do prvního c, nahradí jej C. Na □ se zacyklí.
4. Vrací se doleva na nejbližší A.
5. Pokračuje bodem 1

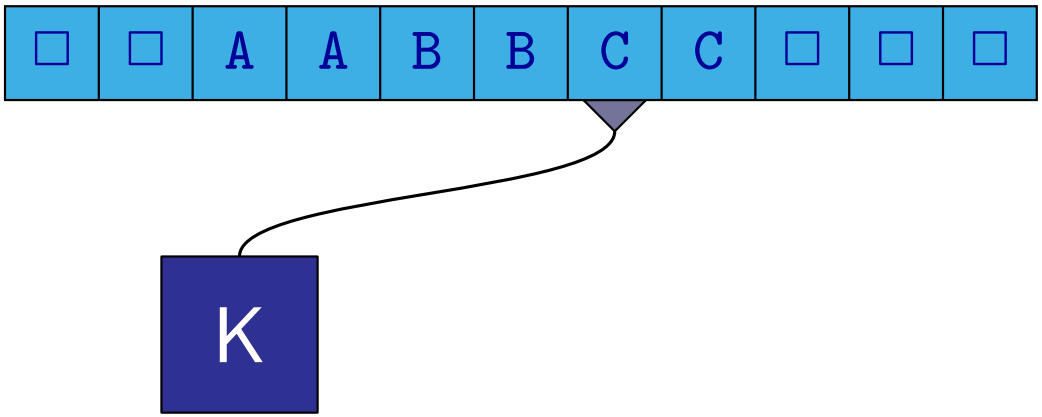
Turingův stroj



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 □ nebo c se zacyklí.
3. Čte do prvního c, nahradí jej C. Na □ se zacyklí.
4. Vrací se doleva na nejbližší A.
5. Pokračuje bodem 1

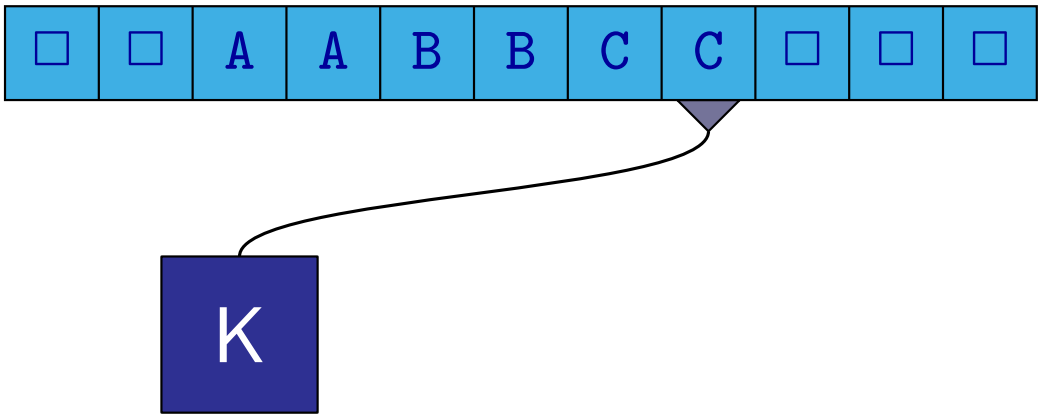
Turingův stroj



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 □ nebo c se zacyklí.
3. Čte do prvního c, nahradí jej C. Na □ se zacyklí.
4. Vrací se doleva na nejbližší A.
5. Pokračuje bodem 1

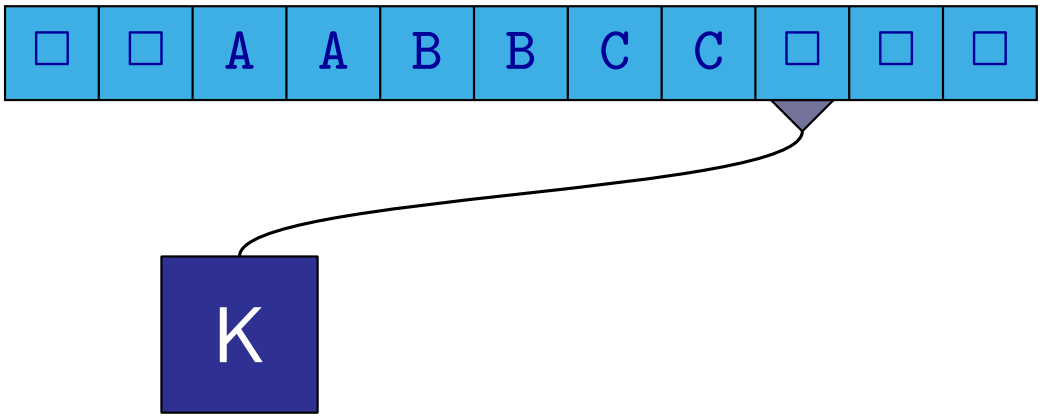
Turingův stroj



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 □ nebo c se zacyklí.
3. Čte do prvního c, nahradí jej C. Na □ se zacyklí.
4. Vrací se doleva na nejbližší A.
5. Pokračuje bodem 1

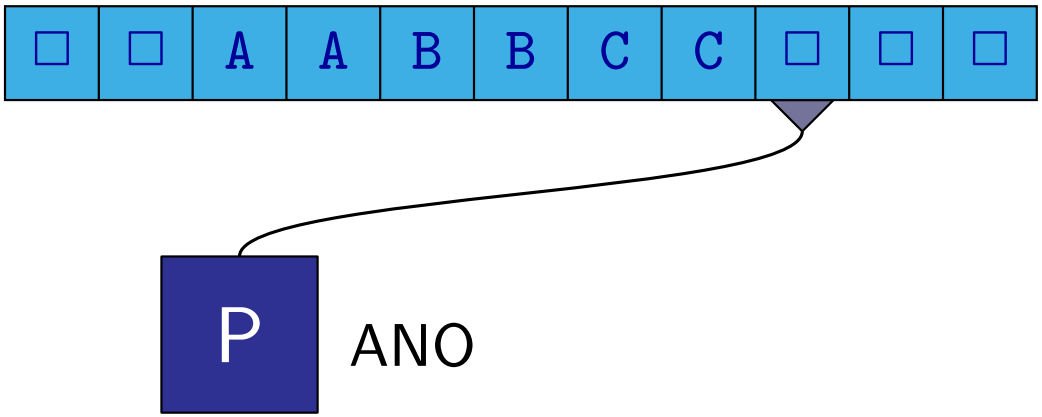
Turingův stroj



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 □ nebo c se zacyklí.
3. Čte do prvního c, nahradí jej C. Na □ se zacyklí.
4. Vrací se doleva na nejbližší A.
5. Pokračuje bodem 1

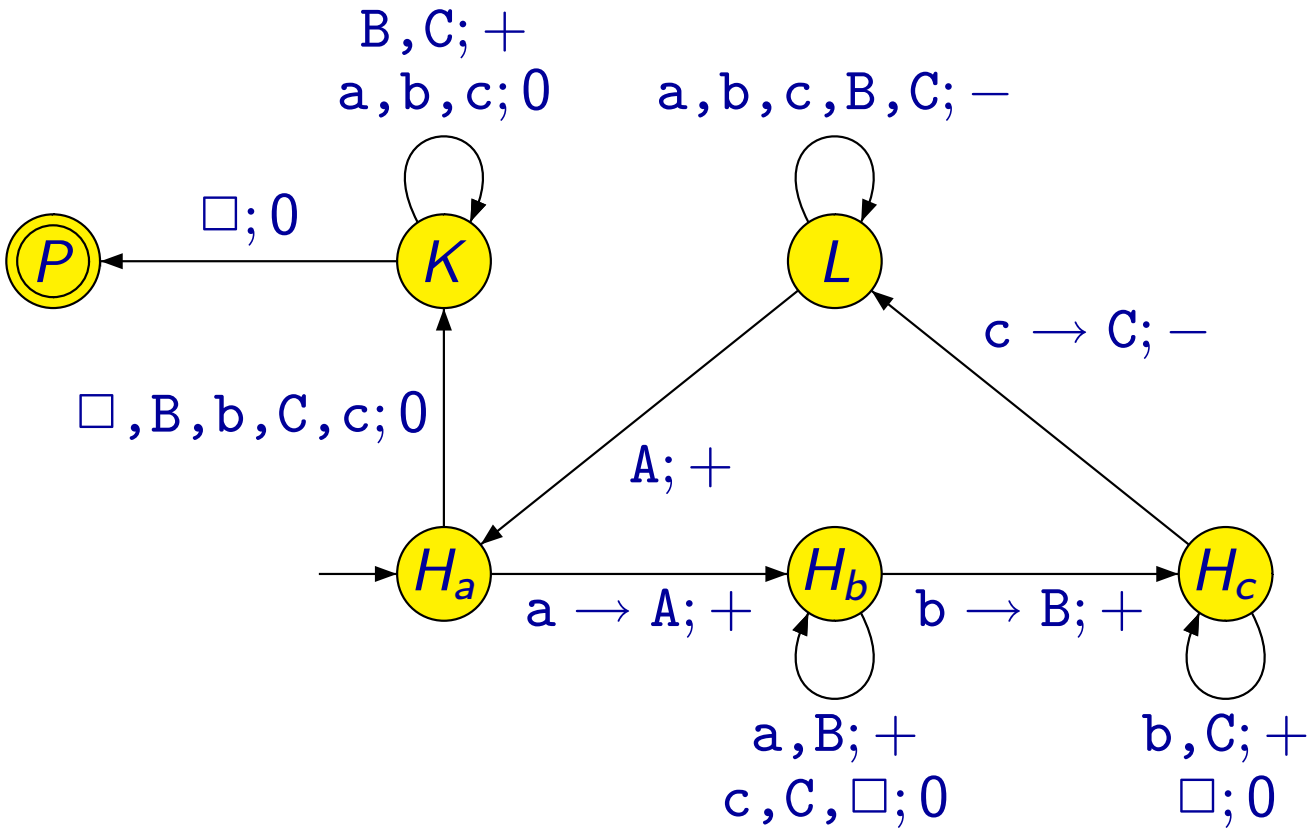
Turingův stroj



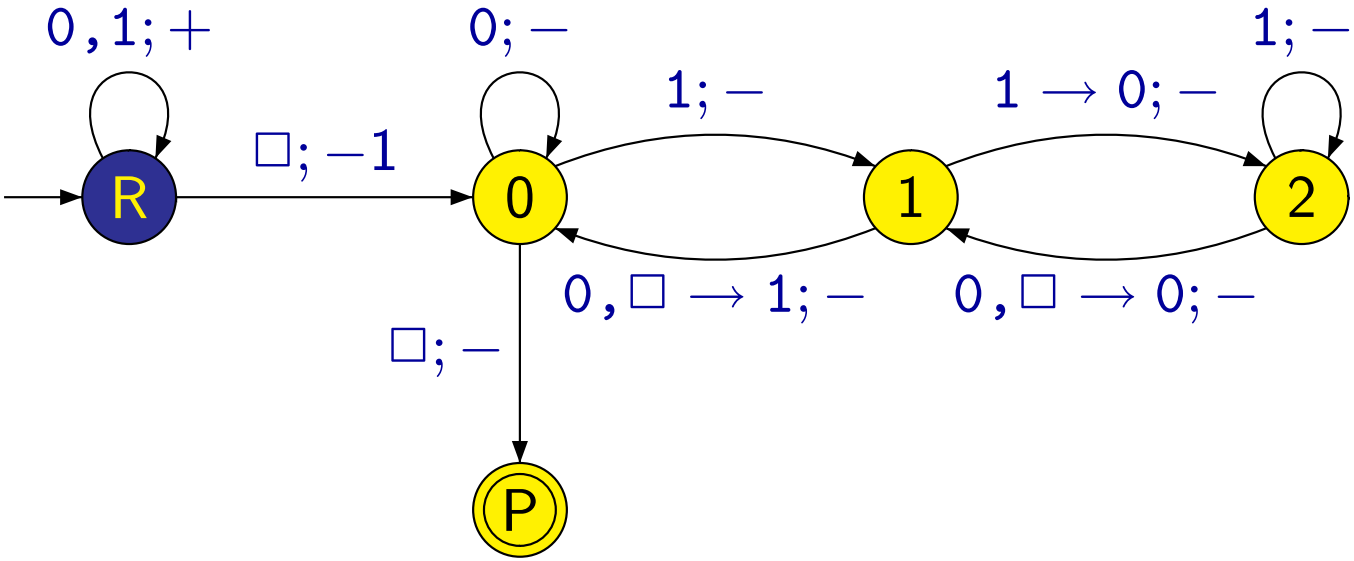
Jazyk $L = \{a^i b^i c^i \mid i \geq 0\}$

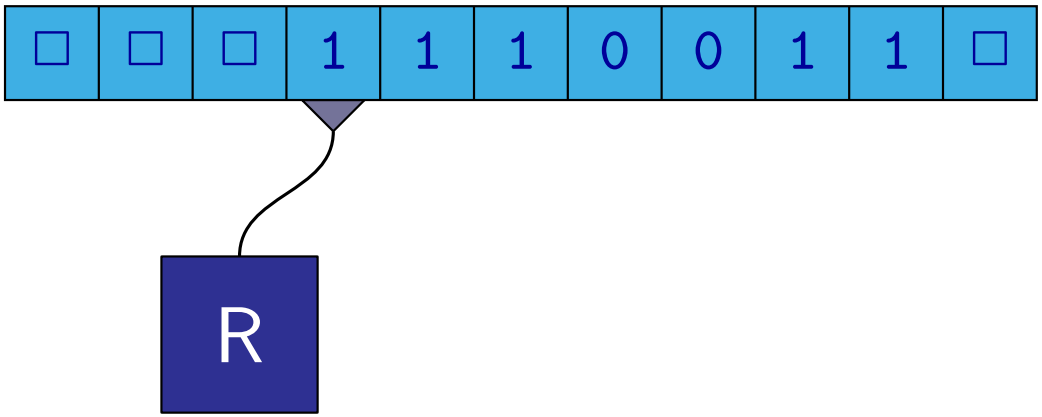
1. Čte do prvního a, nahradí jej A. Pokud najde dříve b nebo c, zacyklí se. Pokud už a není, zkontroluje, zda už jsou jen velká písmena.
2. Čte do prvního b, nahradí jej B. Na □ nebo c se zacyklí.
3. Čte do prvního c, nahradí jej C. Na □ se zacyklí.
4. Vrací se doleva na nejbližší A.
5. Pokračuje bodem 1

Turingův stroj

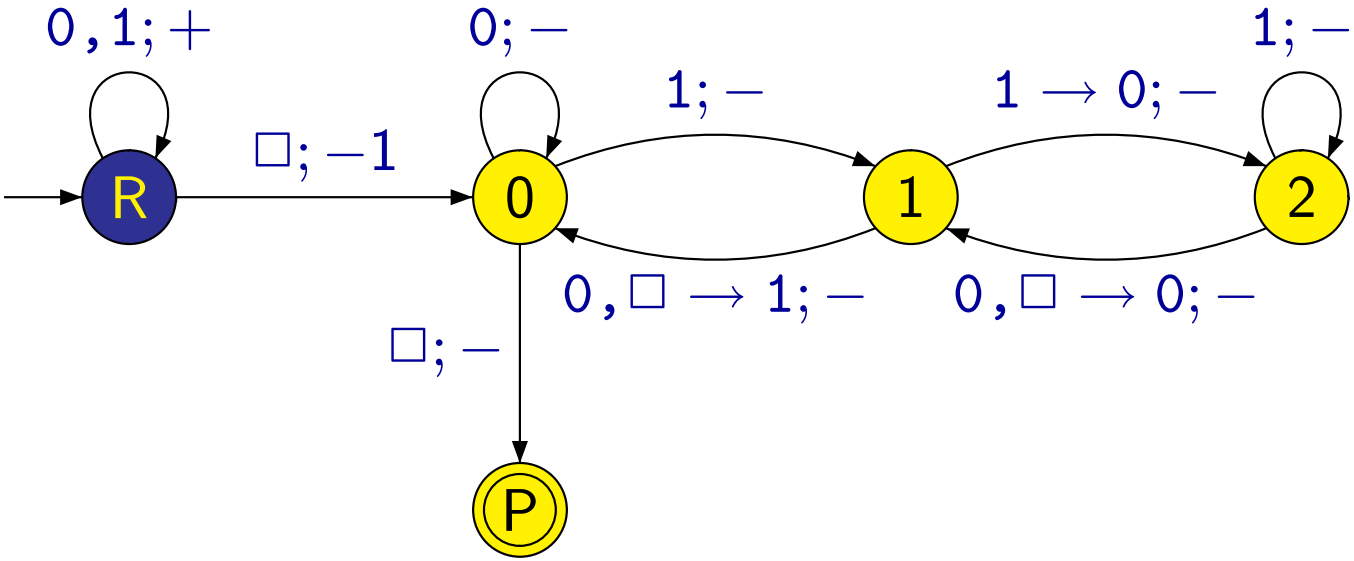


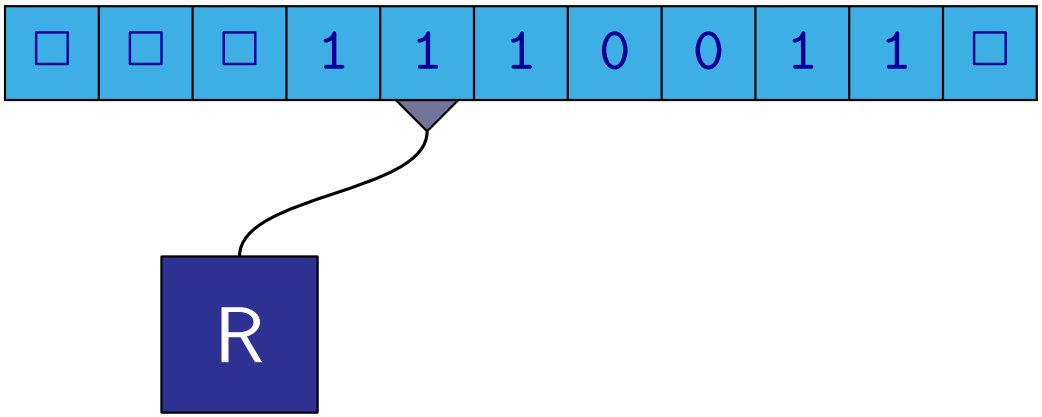
Turingův stroj - násobení třemi



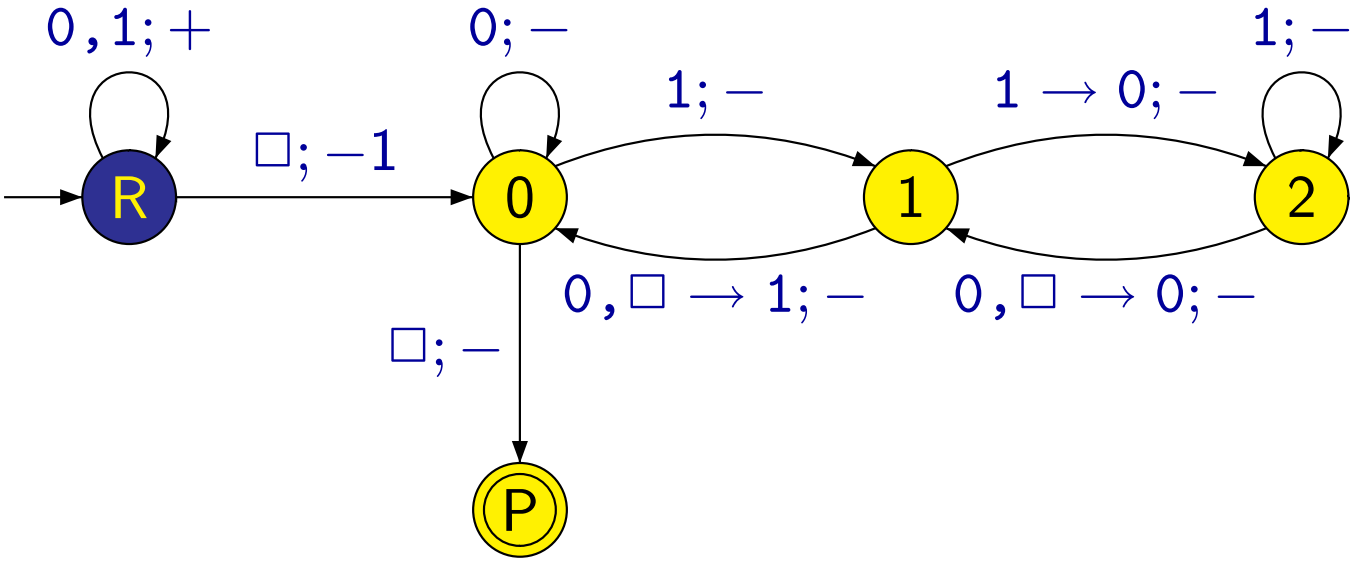


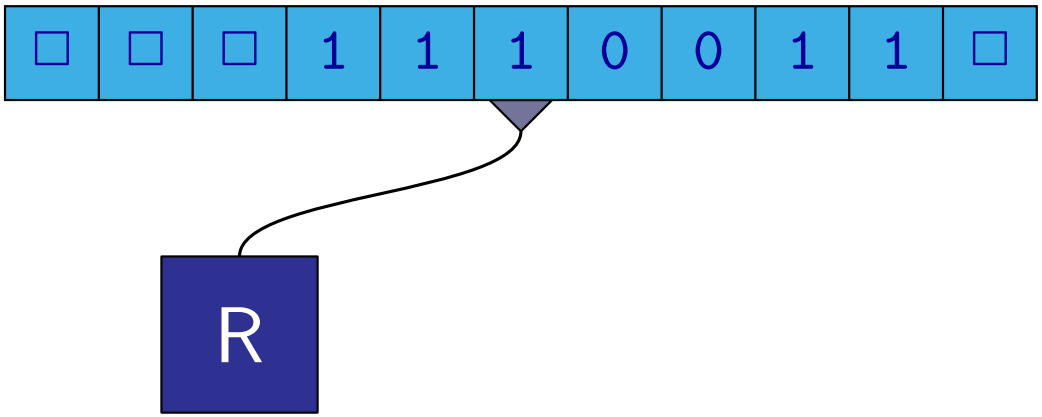
Turingův stroj - násobení třemi



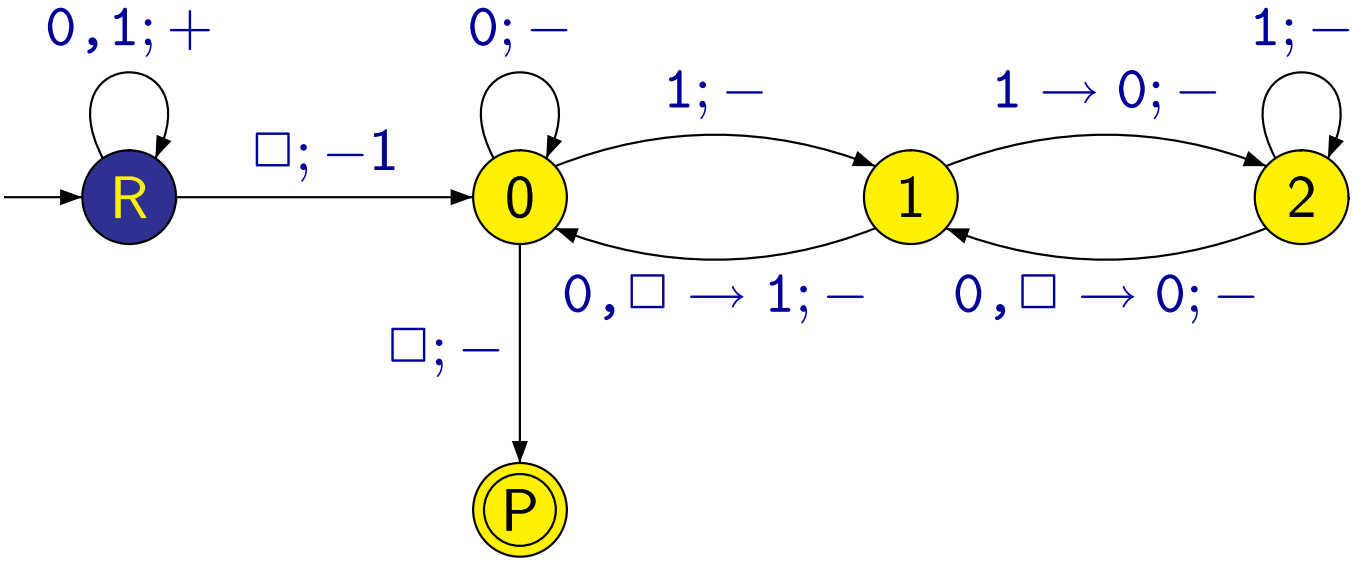


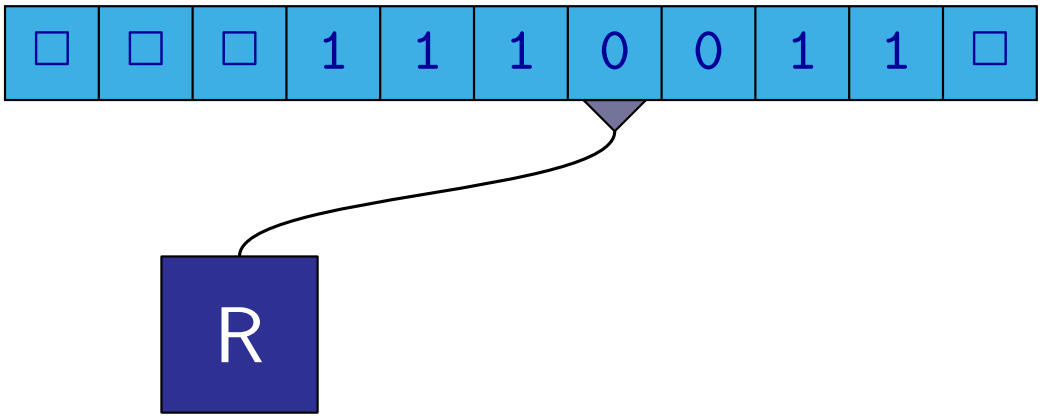
Turingův stroj - násobení třemi



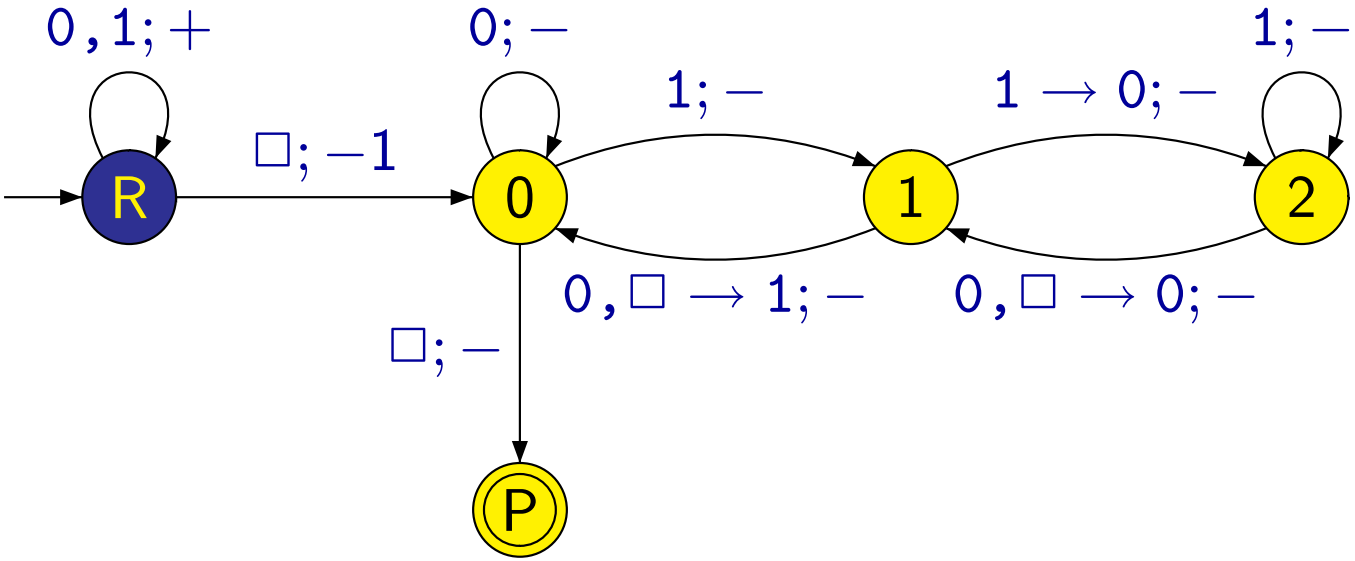


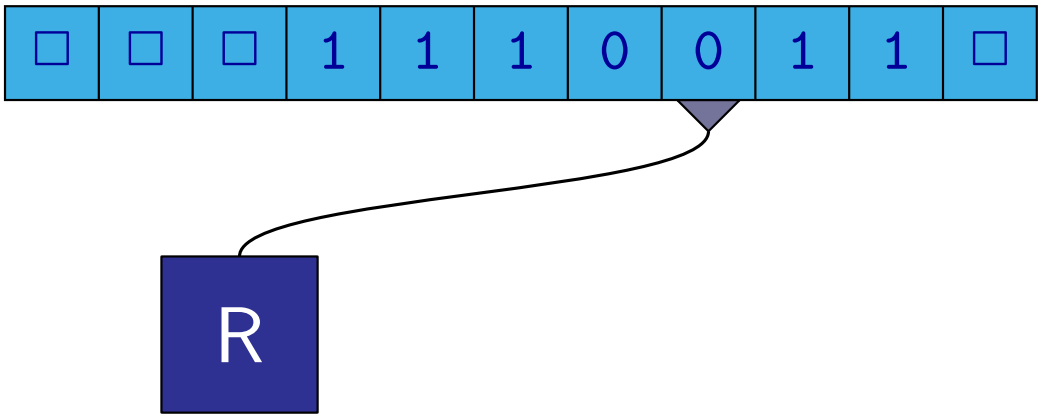
Turingův stroj - násobení třemi



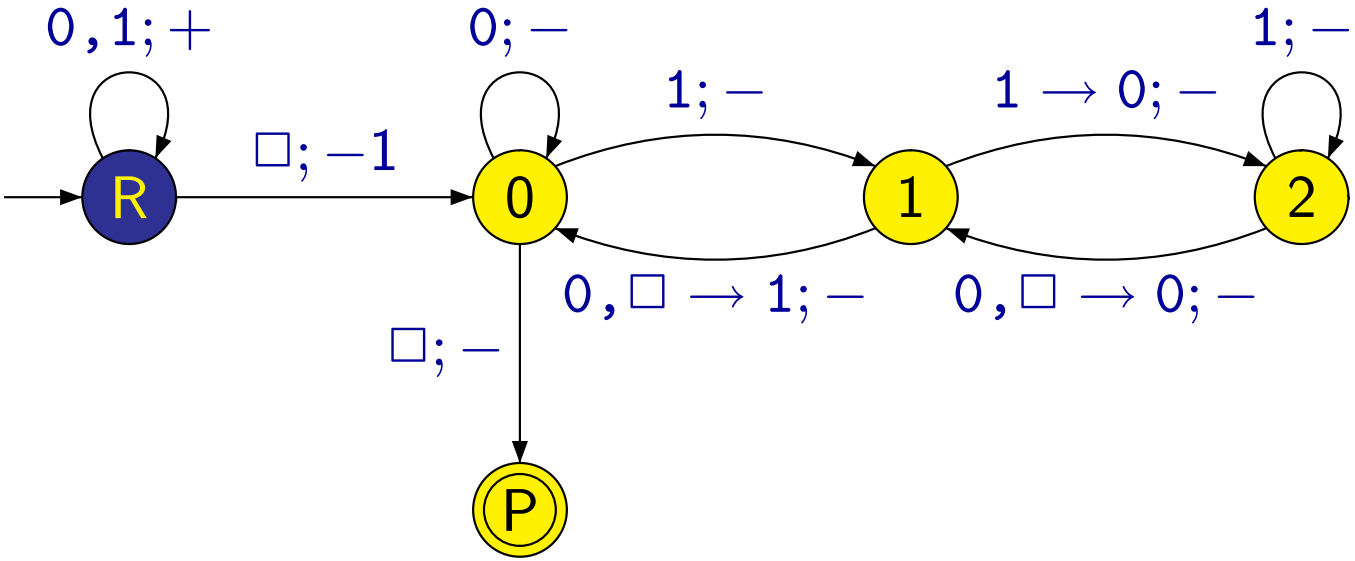


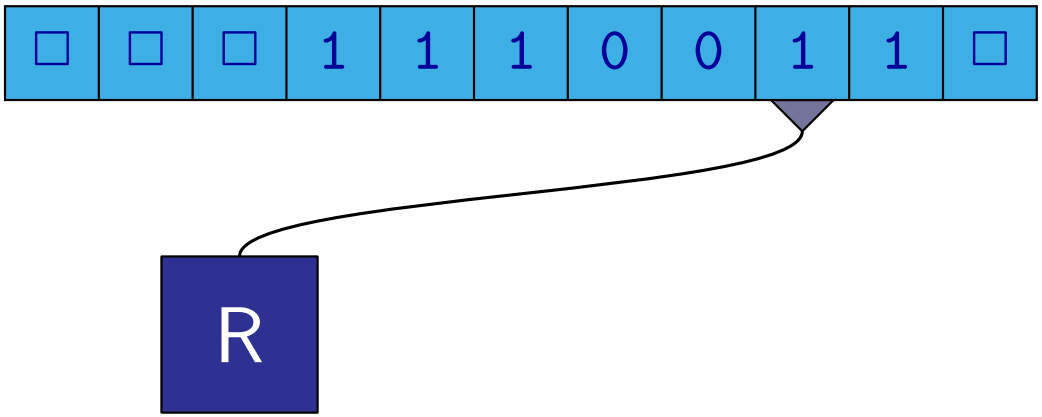
Turingův stroj - násobení třemi



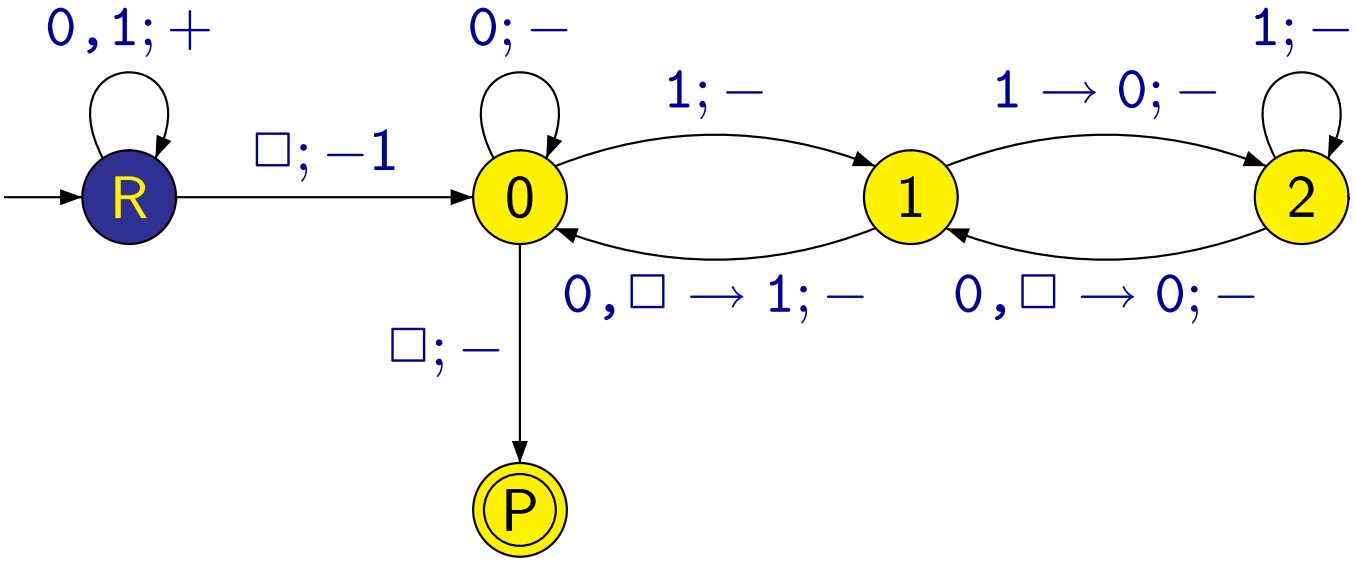


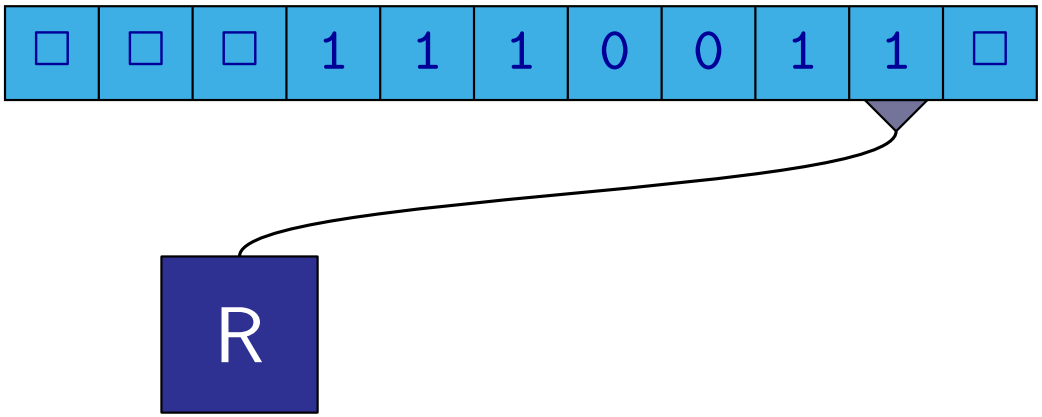
Turingův stroj - násobení třemi



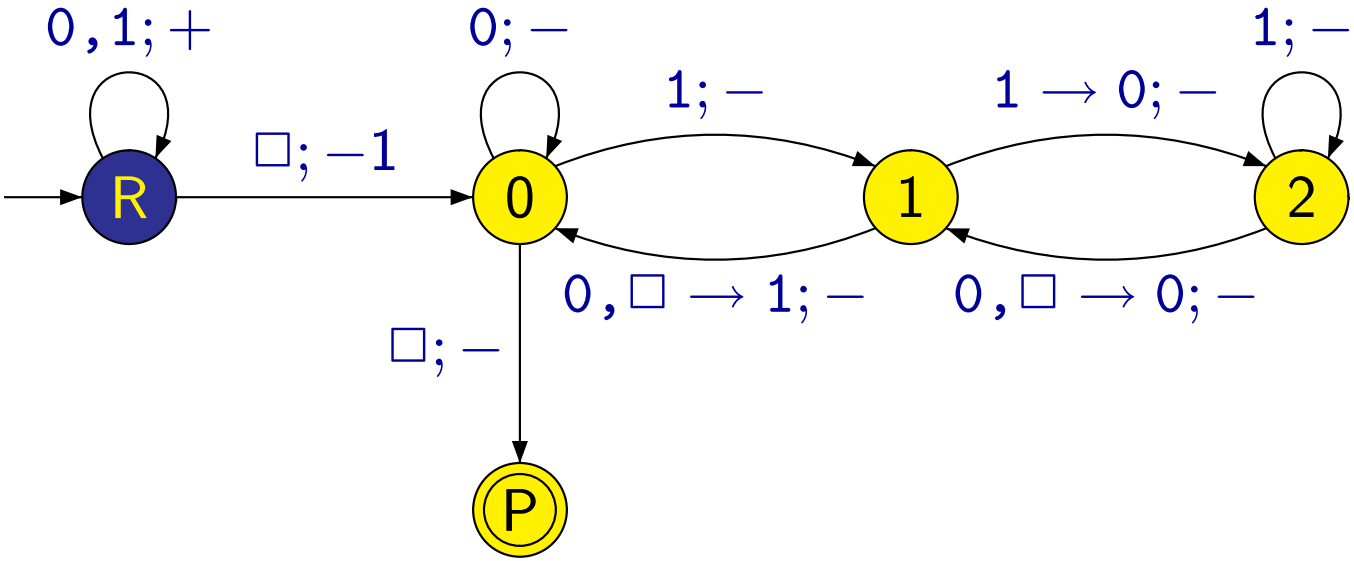


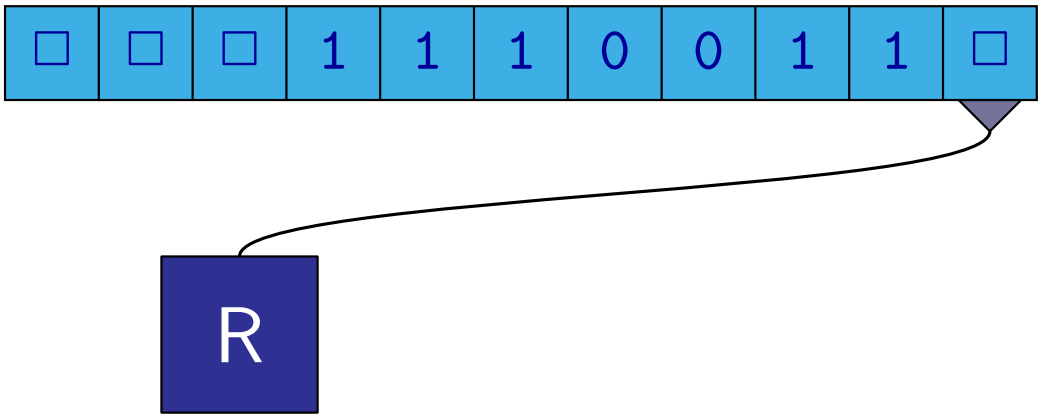
Turingův stroj - násobení třemi



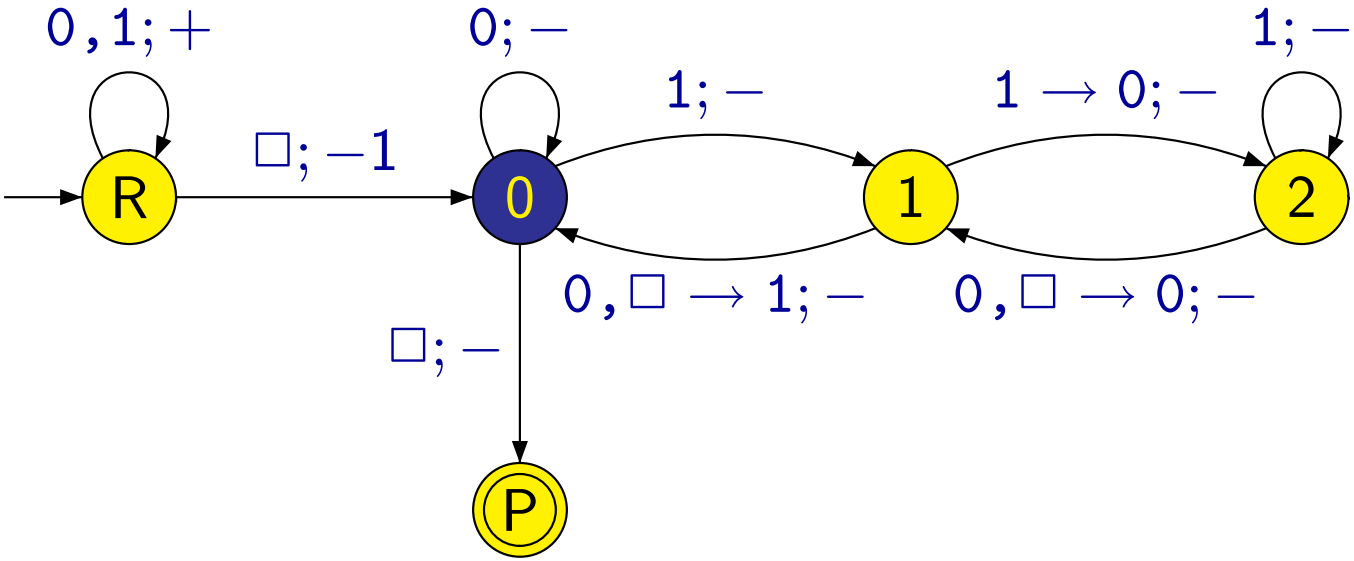


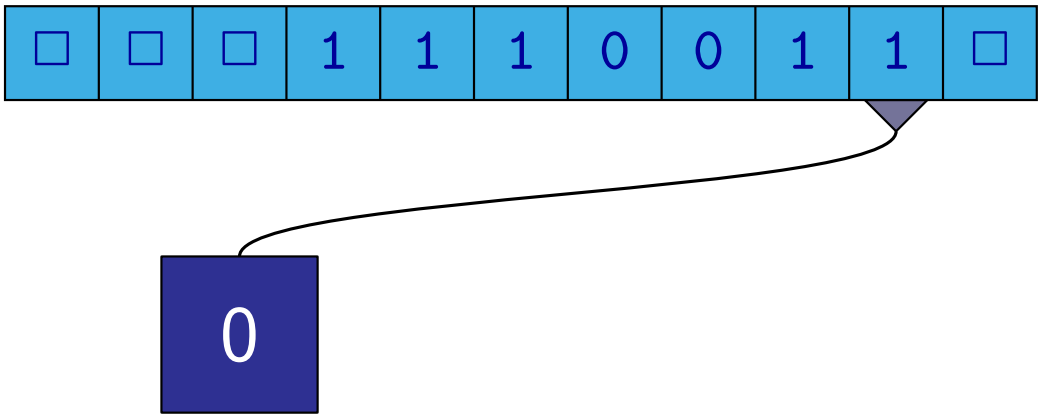
Turingův stroj - násobení třemi



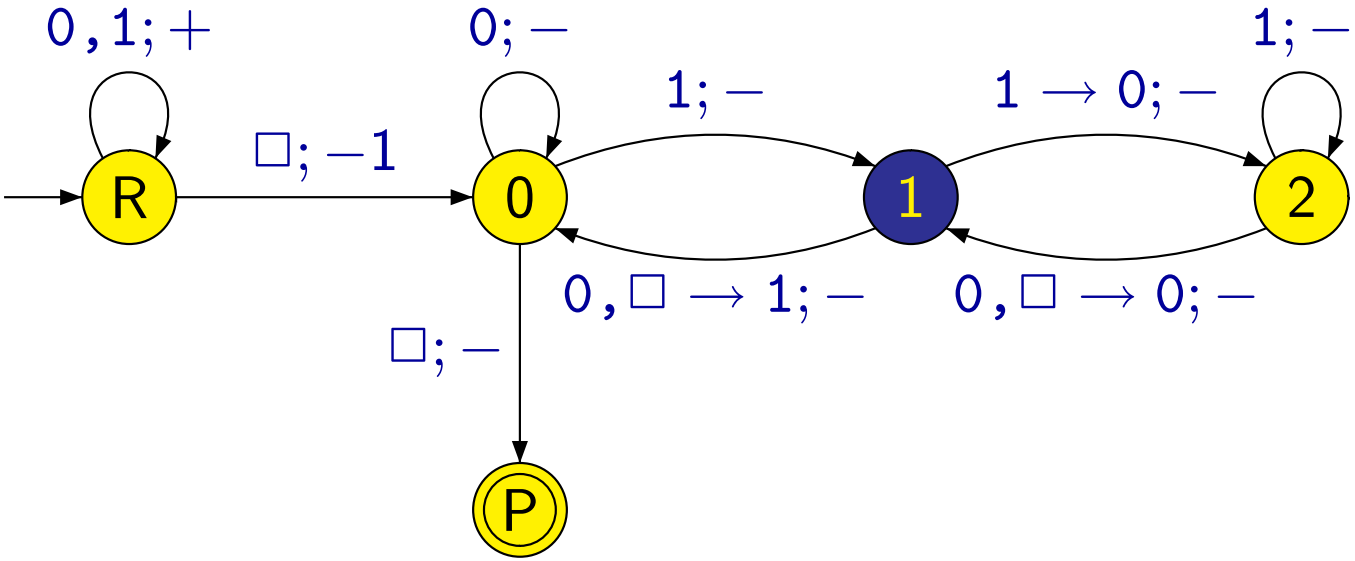


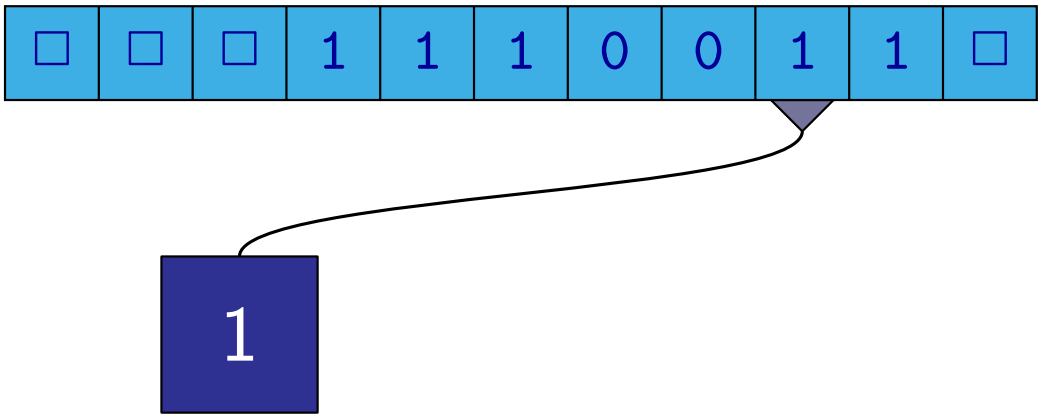
Turingův stroj - násobení třemi



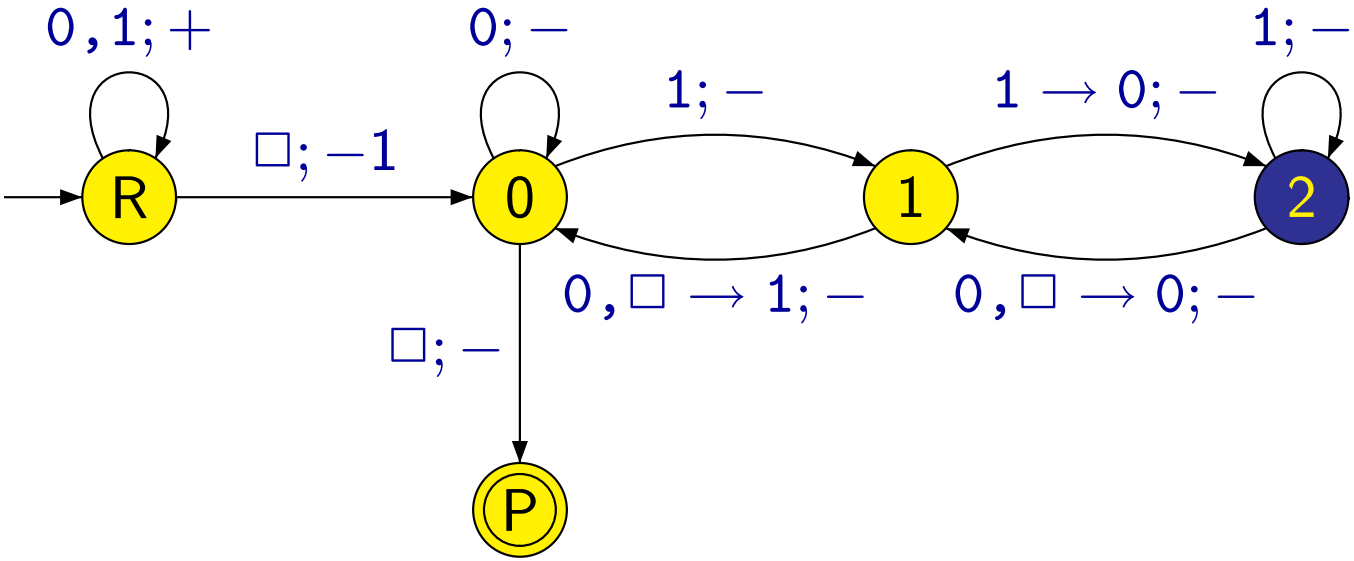


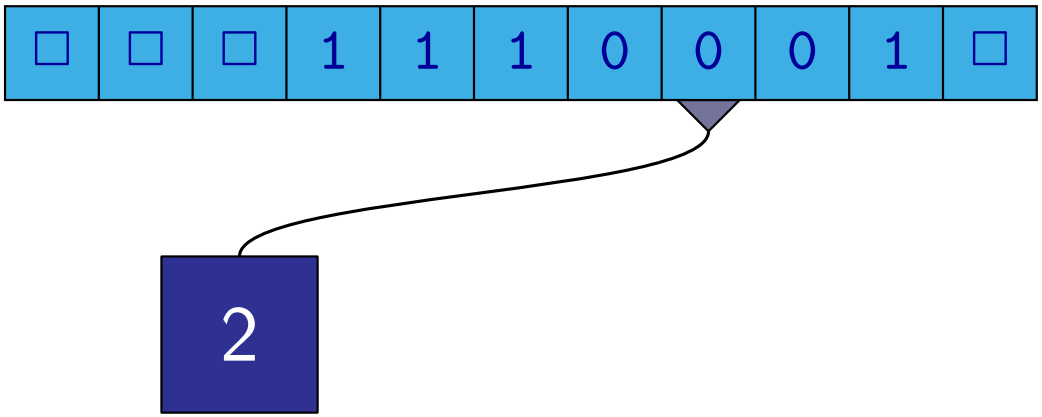
Turingův stroj - násobení třemi



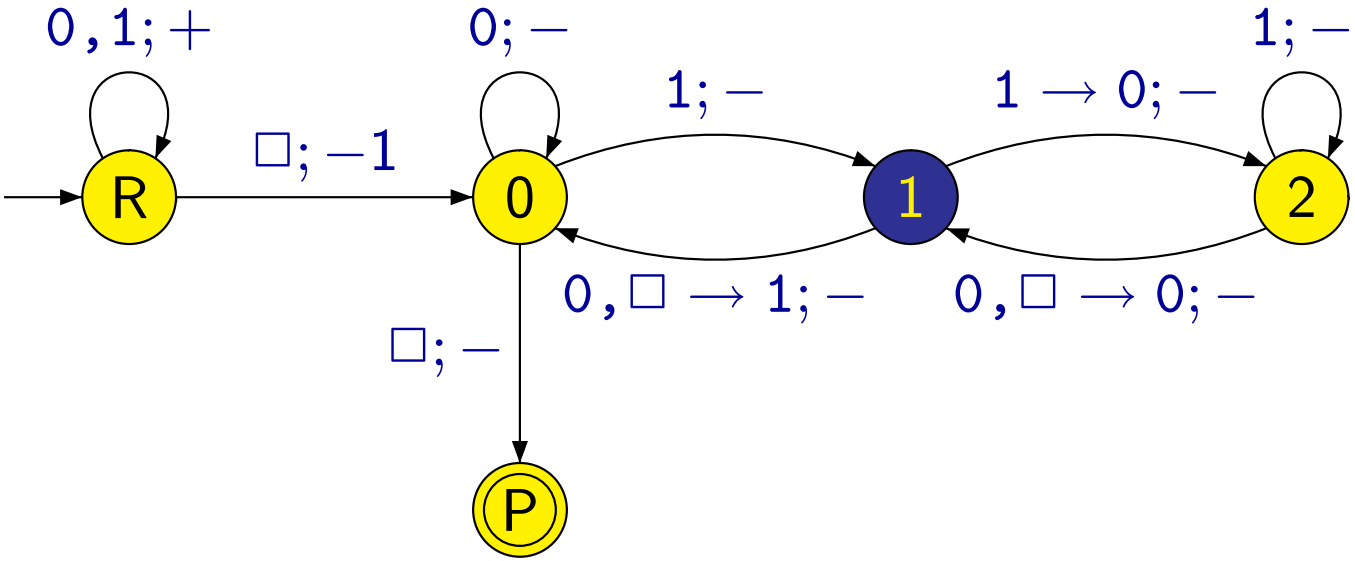


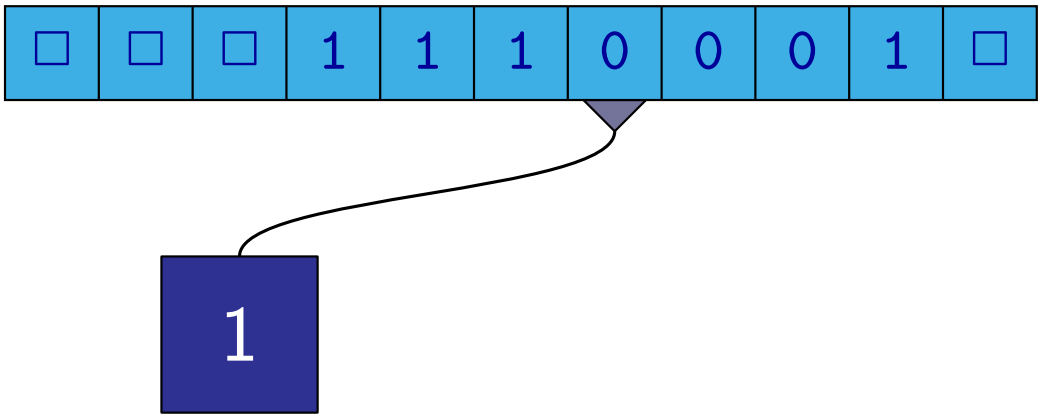
Turingův stroj - násobení třemi



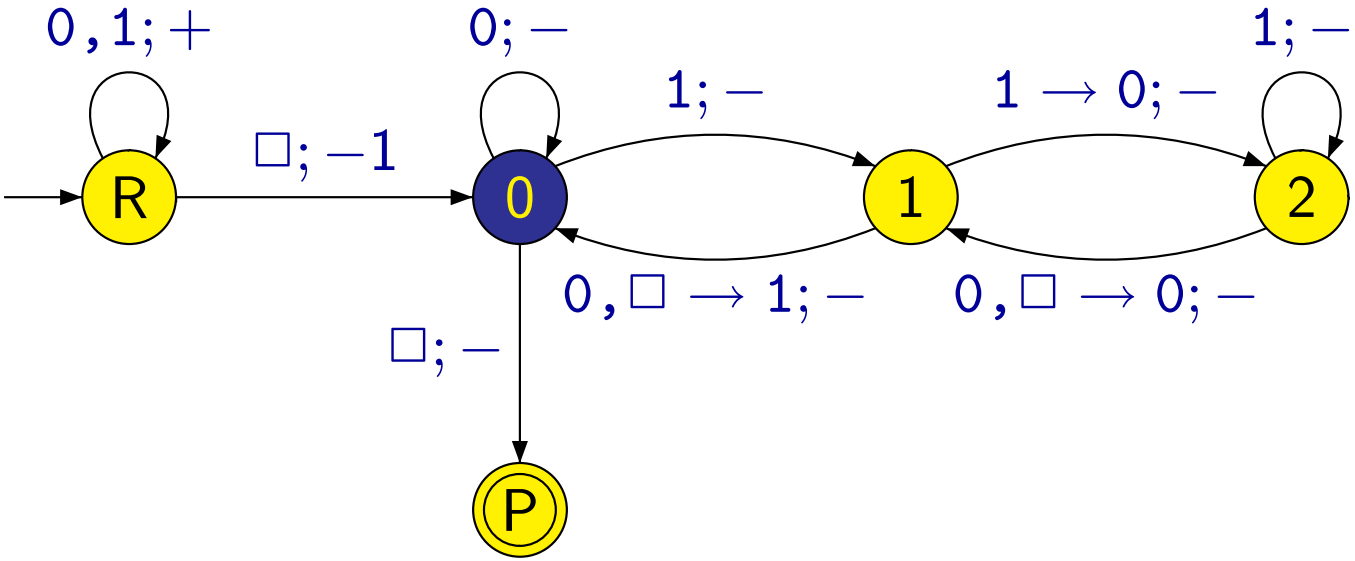


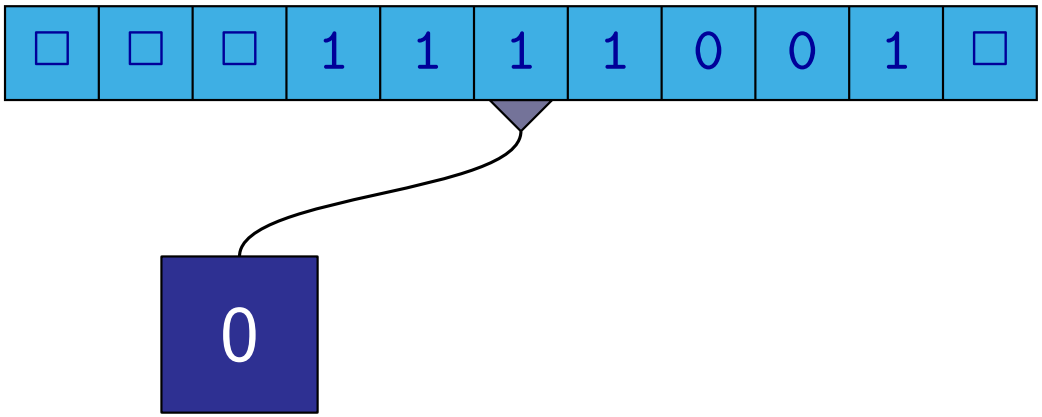
Turingův stroj - násobení třemi



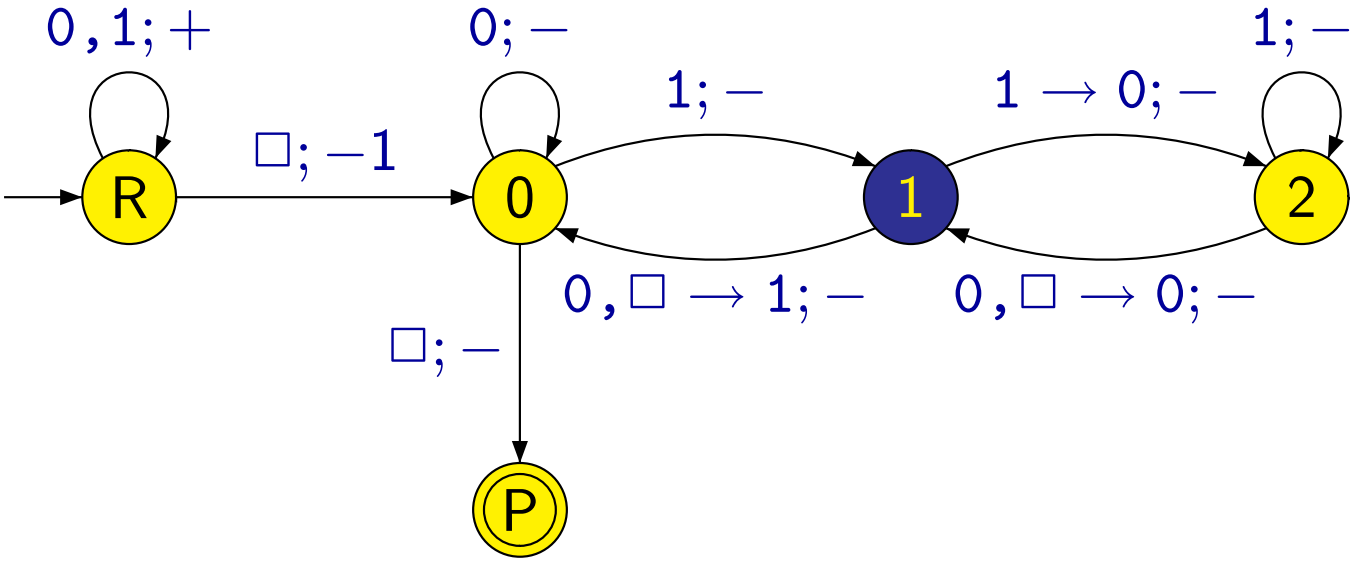


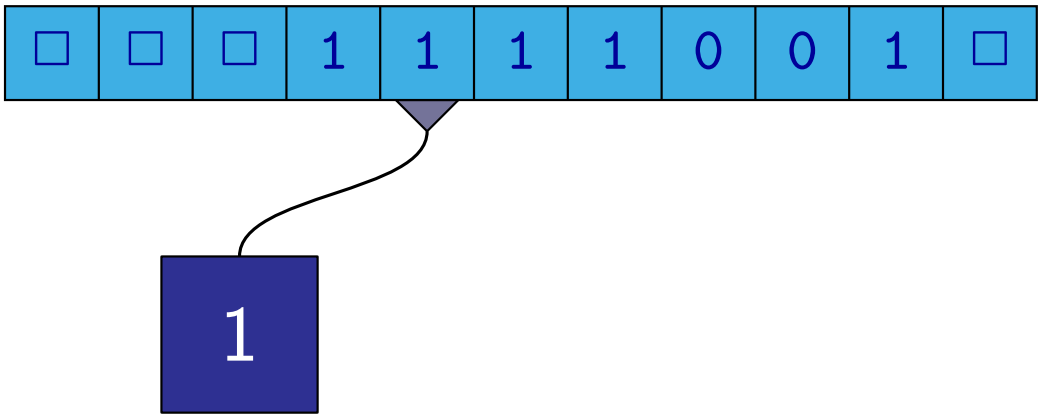
Turingův stroj - násobení třemi



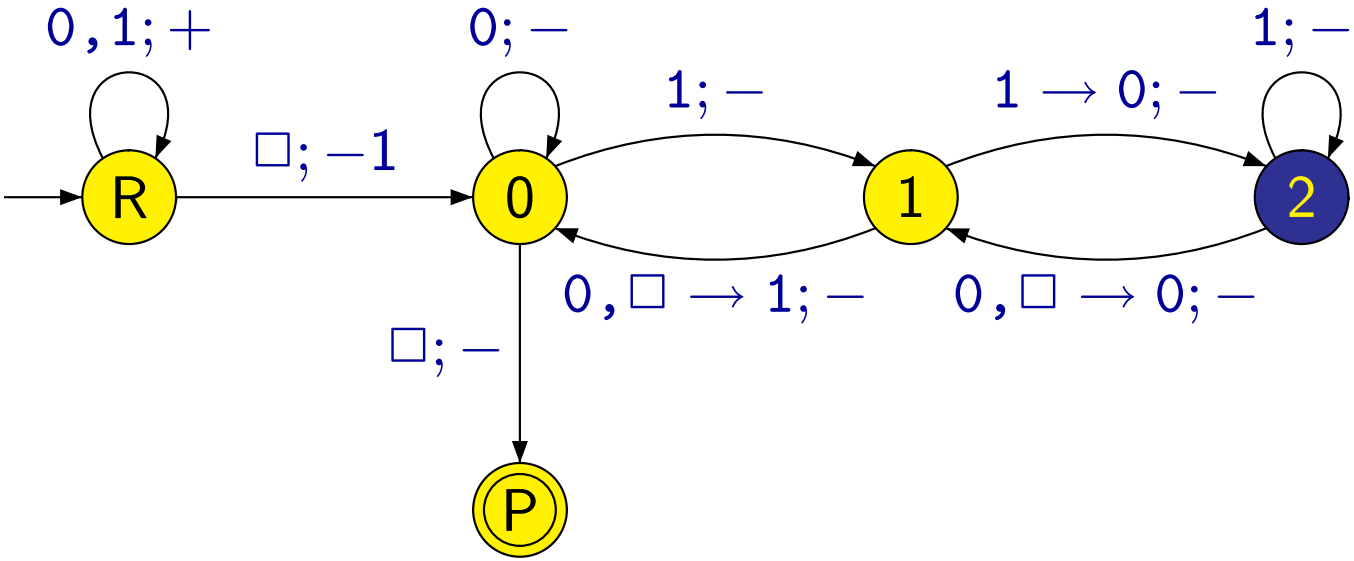


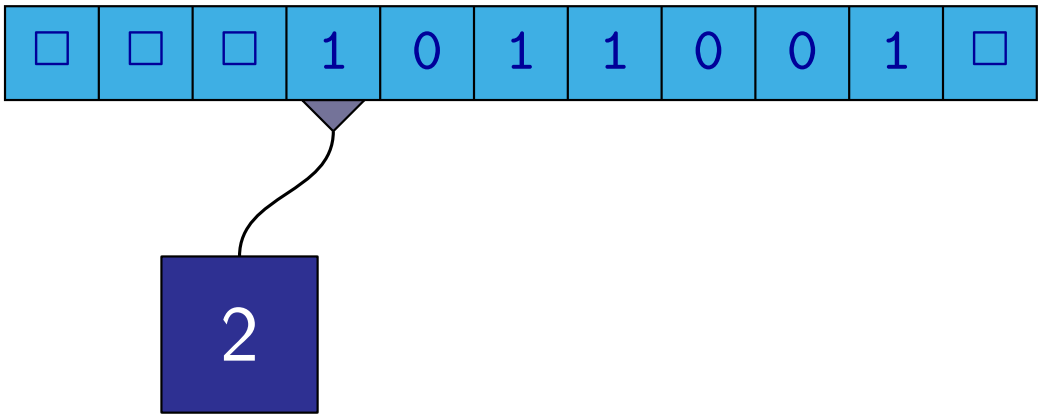
Turingův stroj - násobení třemi



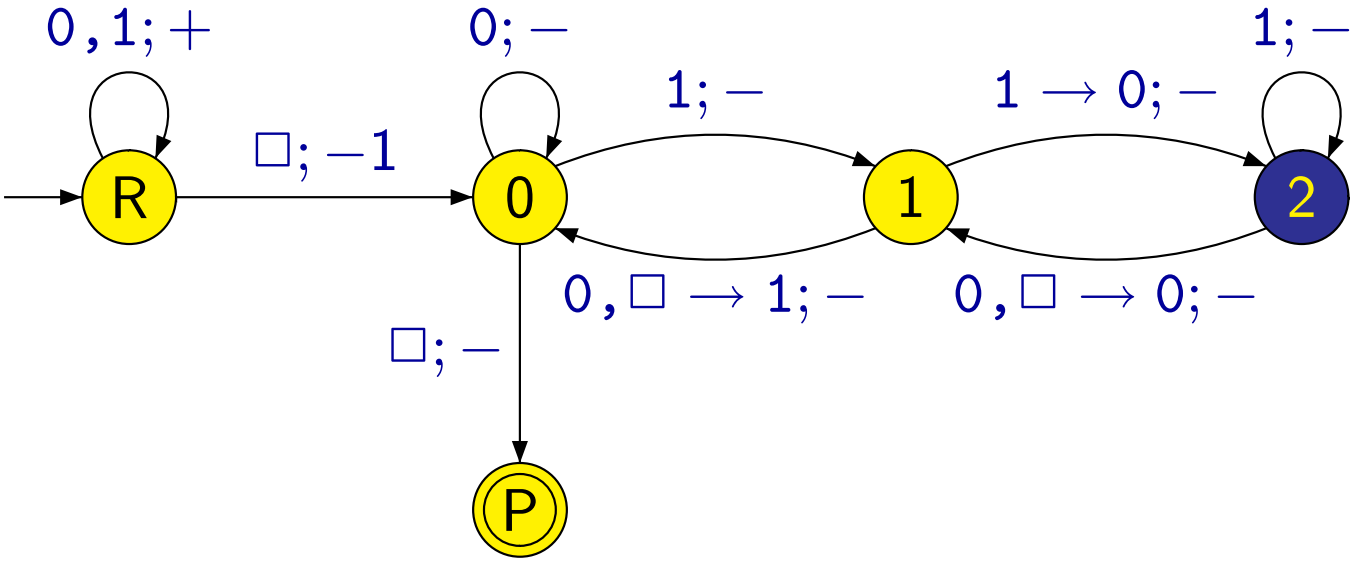


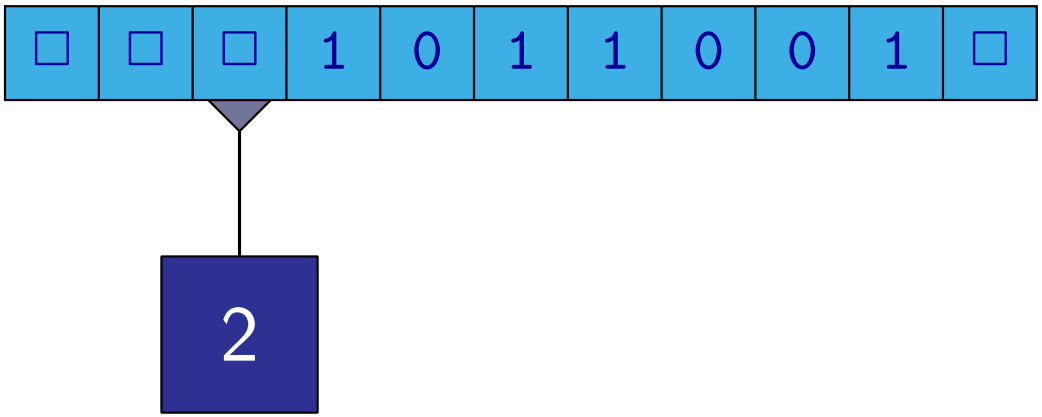
Turingův stroj - násobení třemi



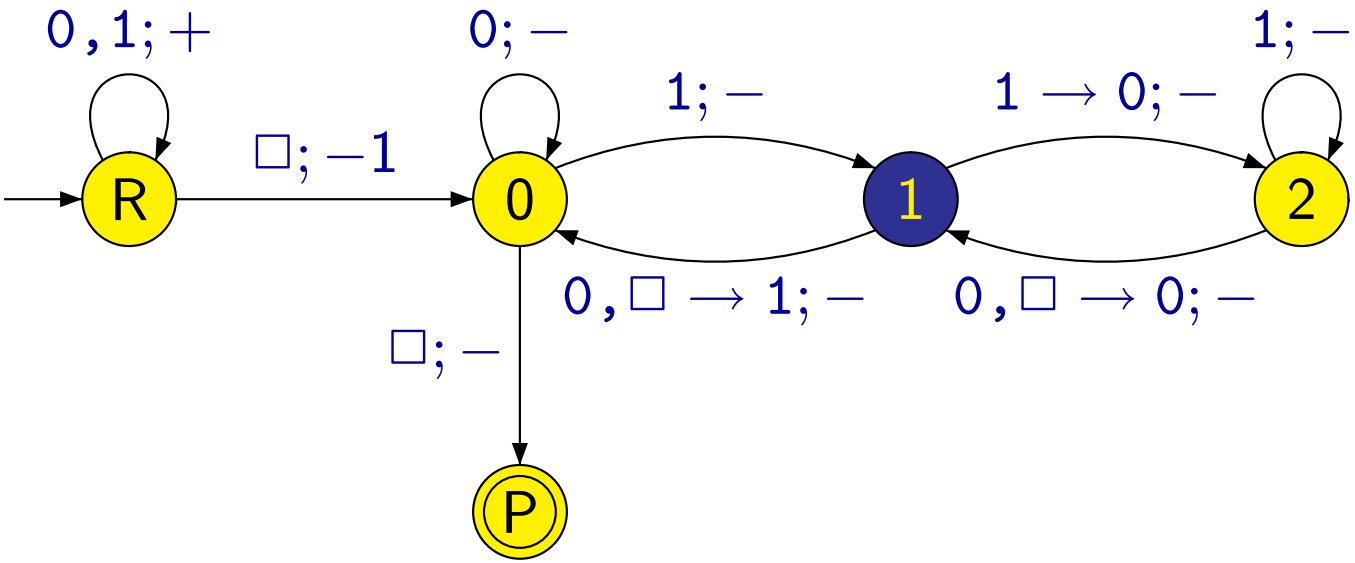


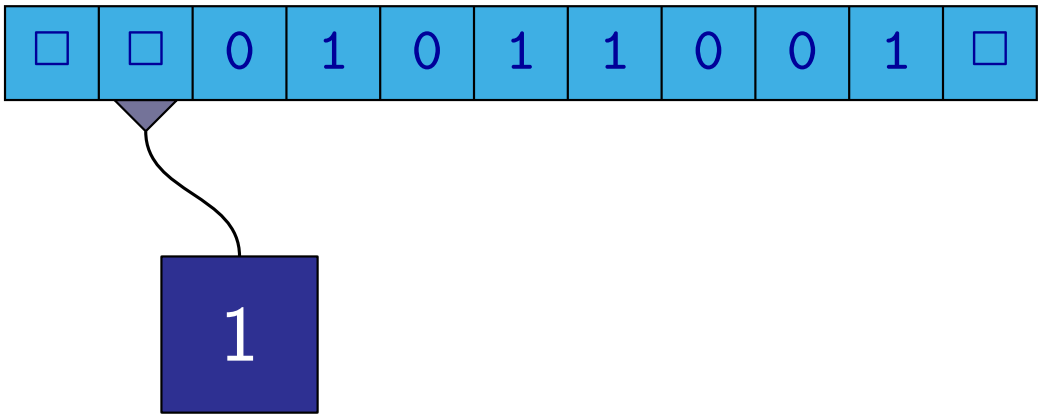
Turingův stroj - násobení třemi



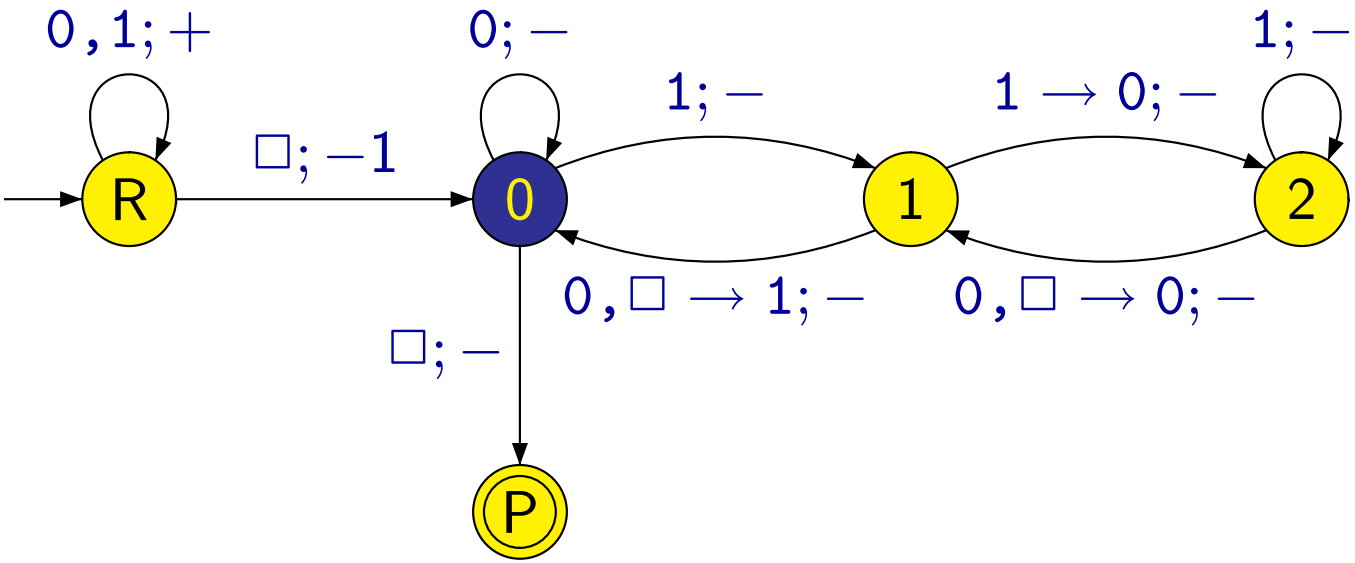


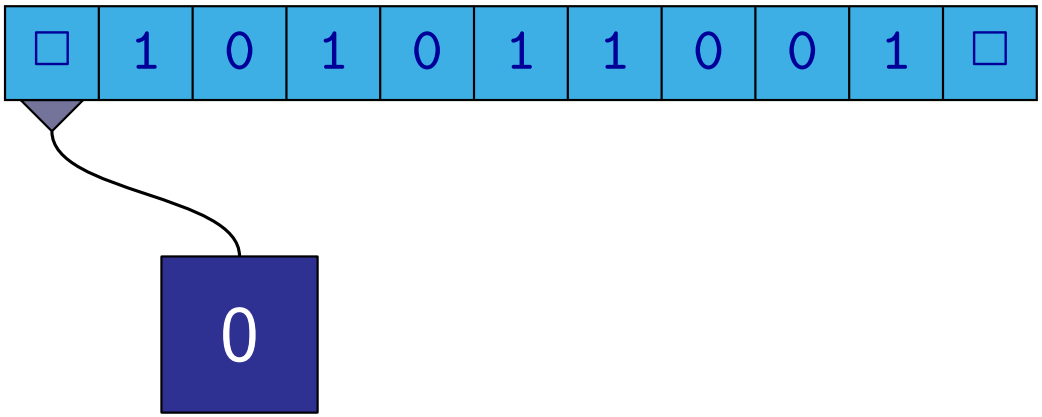
Turingův stroj - násobení třemi



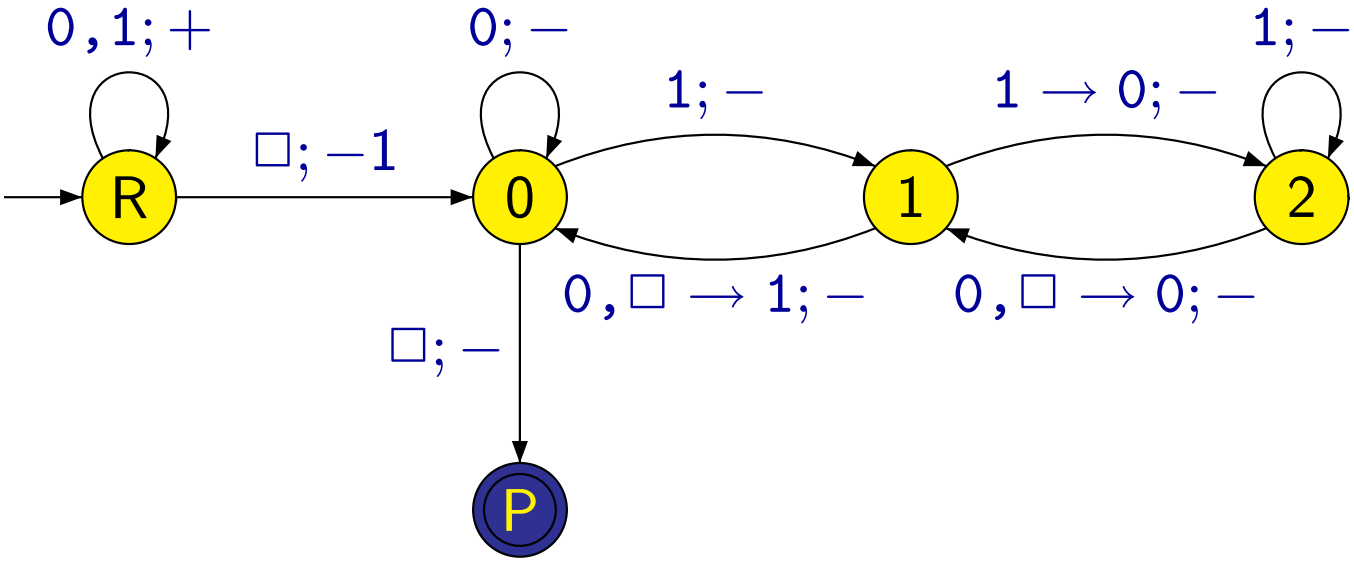


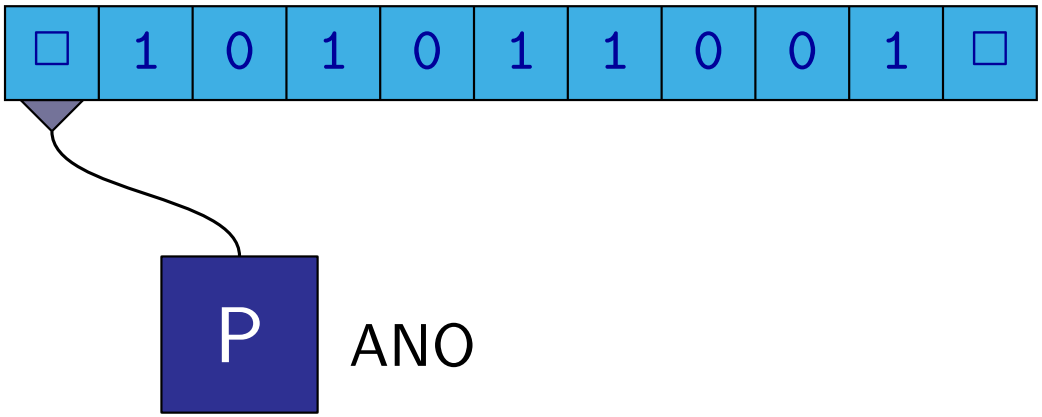
Turingův stroj - násobení třemi





Turingův stroj - násobení třemi





Lineárně omezený automat

- 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

Generativní gramatika

Definice

Generativní gramatika je dána čtveřicí parametrů $G = (\Pi, \Sigma, S, P)$, kde

- Π je konečná množina neterminálů
- Σ je konečná množina terminálů, $\Pi \cap \Sigma = \emptyset$
- $S \in \Pi$ je počáteční neterminál
- P je konečná množina pravidel typu $\alpha \to \beta$, 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 $\alpha \to \beta$ je pravidlo v P
- $L(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$

Chomského hierarchie

Podle tvaru pravidel, která v gramatice povolíme, dělíme gramatiky na gramatiky typu:

0 - Obecné **generativní gramatiky** - pravidla bez omezení

1 - **Kontextové gramatiky** - pravidla tvaru $\alpha X \beta \to \alpha \gamma \beta$, kde $|\gamma| \geq 1$ (Výjimka $S \to \varepsilon$, ale S pak není na pravé straně žádného pravidla)

2 - **Bezkontextové gramatiky** - pravidla tvaru $X \to \gamma$

3 - **Regulární gramatiky** - pravidla tvaru $X \to wY$ nebo $X \to w$

kde $\alpha, \beta, \gamma \in (\Pi \cup \Sigma)^*, X \in \Pi, w \in \Sigma^*$

Chomského hierarchie

Podle tvaru pravidel, která v gramatice povolíme, dělíme gramatiky na gramatiky typu:

0 - Obecné **generativní gramatiky** - pravidla bez omezení

1 - **Kontextové gramatiky** - pravidla tvaru $\alpha X \beta \to \alpha \gamma \beta$, kde $|\gamma| \geq 1$ (Výjimka $S \to \varepsilon$, ale S pak není na pravé straně žádného pravidla)

2 - **Bezkontextové gramatiky** - pravidla tvaru $X \to \gamma$

3 - **Regulární gramatiky** - pravidla tvaru $X \to wY$ nebo $X \to w$

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 $\mathcal{L}_3 \subset \mathcal{L}_2 \subset \mathcal{L}_1 \subset \mathcal{L}_0$

Chomského hierarchie

Pojmenování jazyků jednotlivých typů:

- 0 - Rekurzivně spočetné
- 1 - Kontextové
- 2 - Bezkontextové
- 3 - Regulární

Chomského hierarchie

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)

Rozhodnutelné a nerozhodnutelné problémy

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
- ...

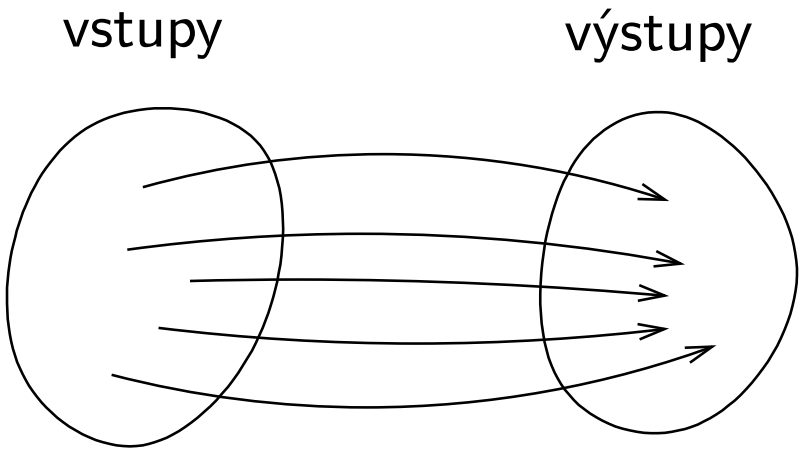
Algoritmy slouží k řešení různých **problémů**.

Problémy

Problém

V zadání **problému** musí být určeno:

- co je množinou možných vstupů
- co je množinou možných výstupů
- jaký je vztah mezi vstupy a výstupy



Příklady problémů

Problém „Třídění“

Vstup: Sekvence prvků $a_1, a_2, \ldots, a_n$.

Výstup: Prvky sekvence $a_1, a_2, \ldots, a_n$ 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.

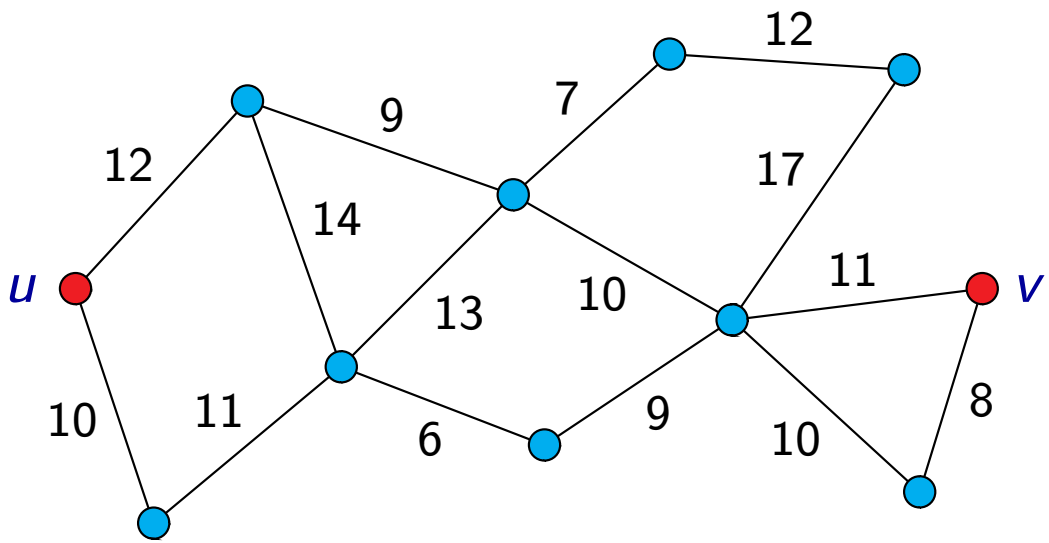
Příklady problému

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:



Další příklady problémů

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, ...

Rozhodovací problémy

Situace, kdy množina výstupů je $\{\text{ANO}, \text{NE}\}$ 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.

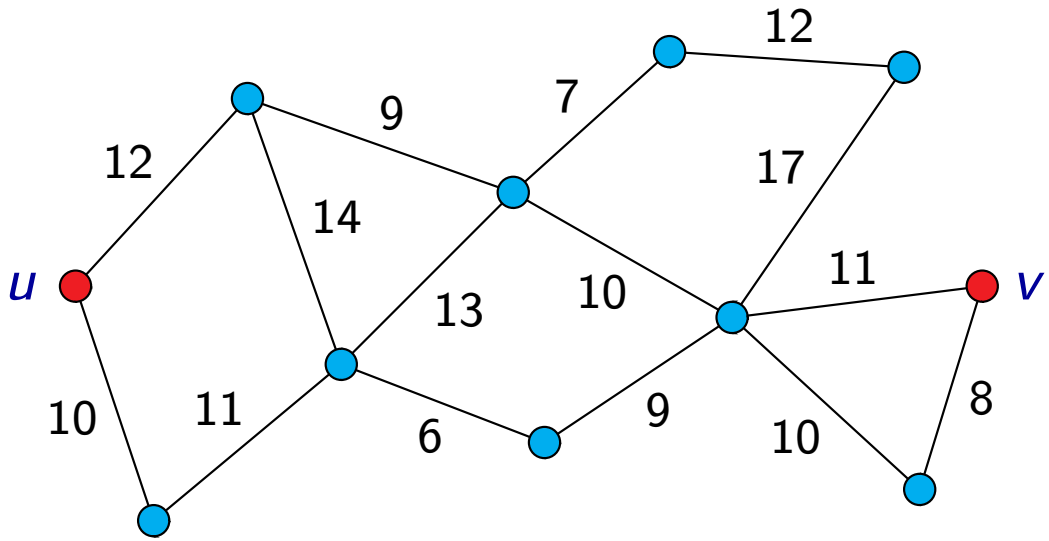
Optimalizační problém je problém, kde je úkolem vybrat z nějaké množiny přípustných řešení takové řešení, které je v nějakém ohledu optimální.

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 problémy „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 problémů

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 Σ.

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

$$13,101,128,562,826,3901$$

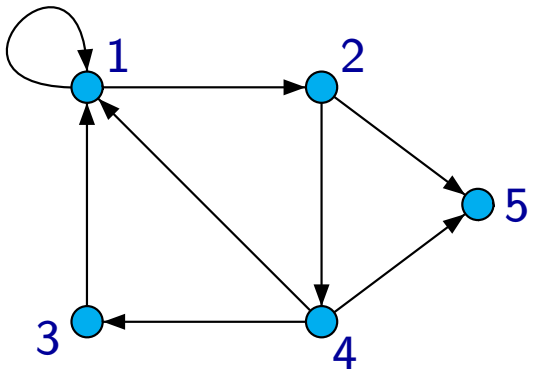
Kódování vstupu a výstupu

Poznámka: Ne každé slovo ze Σ^* 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).

Symboly jakékoli jiné abecedy lze reprezentovat jako sekvence bitů.

Příklad: Abeceda $\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}, \mathtt{f}, \mathtt{g}\}$

$$
\begin{aligned}
\mathtt{a} &\leftrightarrow 001 \\
\mathtt{b} &\leftrightarrow 010 \\
\mathtt{c} &\leftrightarrow 011 \\
\mathtt{d} &\leftrightarrow 100 \\
\mathtt{e} &\leftrightarrow 101 \\
\mathtt{f} &\leftrightarrow 110 \\
\mathtt{g} &\leftrightarrow 111
\end{aligned}
$$

Slovo '$\mathtt{defb}$' můžeme reprezentovat jako '100101110010'.

Funkce realizovaná algoritmem

Můžeme tedy říct, že každý algoritmus realizuje výpočet hodnot nějaké funkce

$$f : \Sigma^* \to \Sigma^*$$

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

Alternativně se také na algoritmy můžeme dívat tak, že realizují výpočet nějaké funkce

$$f : \mathbb{N} \to \mathbb{N}$$

kde $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$ je množina přirozených čísel, neboť na každou sekvenci bitů se můžeme dívat jako na zápis čísla v binární soustavě.

Funkce realizovaná algoritmem

Funkce

$$f : \Sigma^* \to \Sigma^* \qquad \text{resp.} \qquad f : \mathbb{N} \to \mathbb{N}$$

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 $f : A \to B$ 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.

Algoritmy a problémy

Stejně jako na algoritmy i na problémy se můžeme dívat jako na funkce typu:

$$f : \Sigma^* \to \Sigma^* \qquad \text{resp.} \qquad f : \mathbb{N} \to \mathbb{N}$$

Jestliže algoritmus realizuje nějakou funkci f', řekneme, že tento algoritmus **řeší** problém popsaný funkcí f, jestliže se pro libovolný vstup x zastaví s tím, že

$$f(x) = f'(x)$$

Rozhodovací problémy a jazyky

Na rozhodovací problémy můžeme pohlížet jako na jazyky.

Jazyk odpovídající danému rozhodovacímu problému je množina těch slov ze Σ^*, 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 $110 \notin L$.

Rozhodovací problémy

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.

Rozhodovací problémy

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?

Algoritmicky řešitelné problémy

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 „algoritmus“

Dosavadní definice pojmu algoritmus byla poněkud vágní.

Pokud bychom pro nějaký problém chtěli ukázat, že neexistuje algoritmus, který by daný problém řešil, tak by to s takovouto neurčitou definicí pojmu algoritmus asi nešlo.

Intuitivně chápeme, co by měl mít algoritmus za vlastnosti:

- Měl by se skládat z jednoduchých kroků, které je možno vykonávat „mechanicky“, bez porozumění problému.
- Objekty, se kterými algoritmus pracuje, i prováděné operace by měly být konečné.

Algoritmus realizovaný počítačem

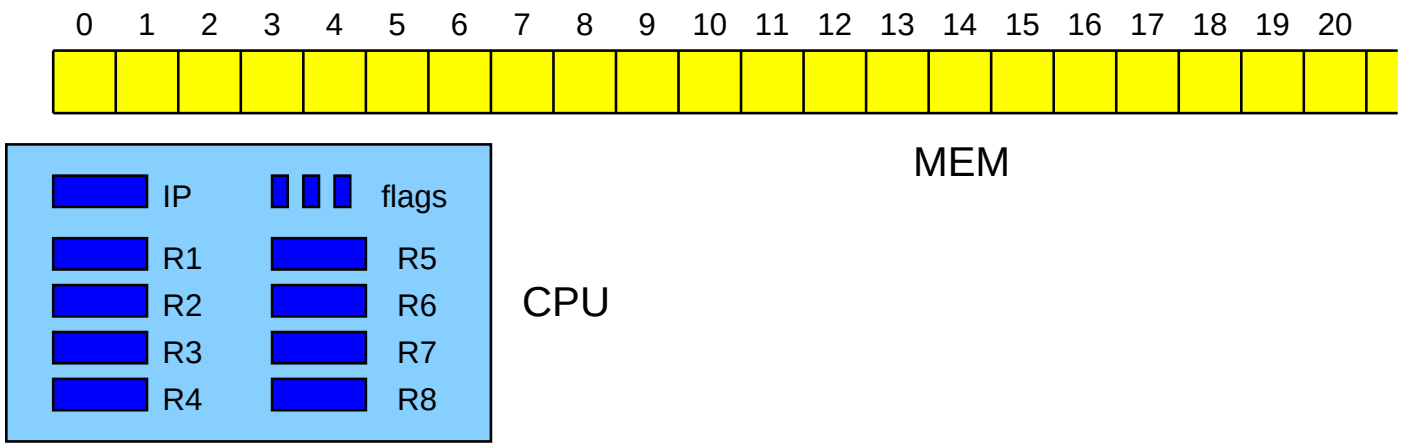
Například program v libovolném programovacím jazyce dané vlastnosti zcela jistě má.

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.

Činnost počítače

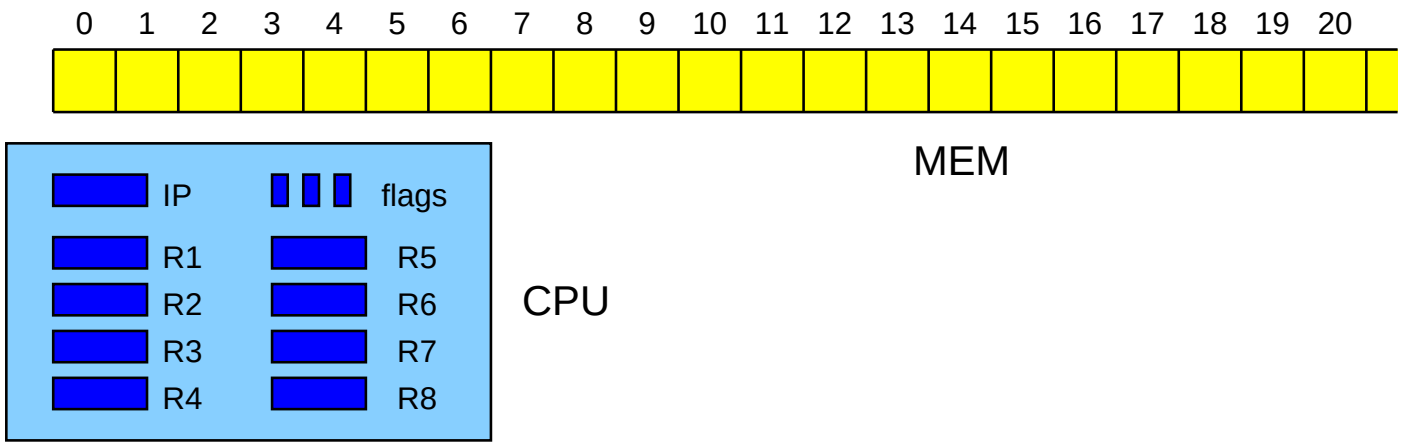
Typická architektura naprosté většiny počítačů vypadá následovně:



- 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 $0 \ldots 255$.
- Buňky jsou očíslovány. Číslo buňky se nazývá její **adresa**.

Činnost počítače

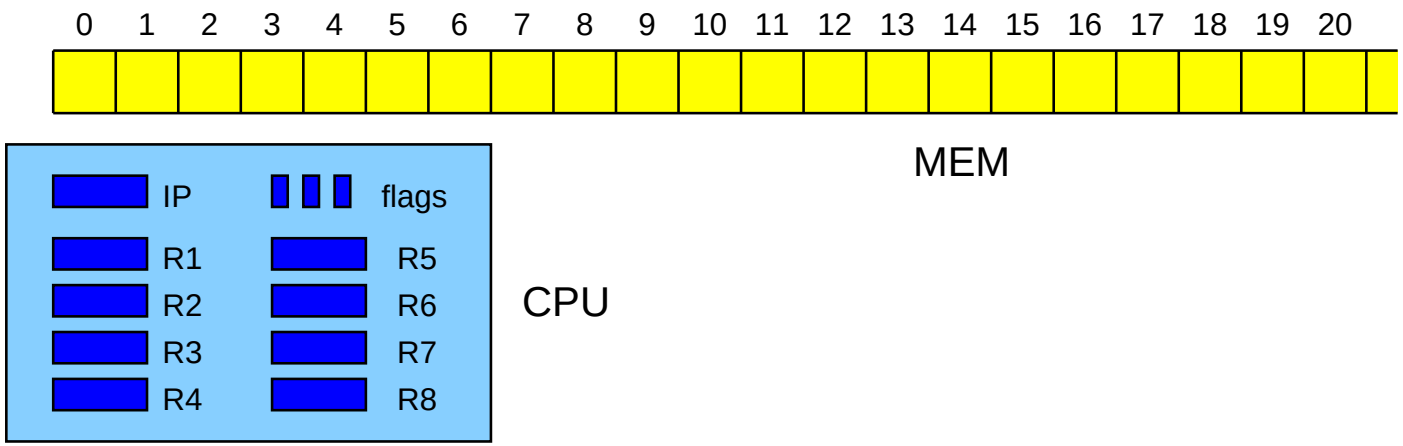
Typická architektura naprosté většiny počítačů vypadá následovně:



- 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.

Činnost počítače

Typická architektura naprosté většiny počítačů vypadá následovně:



- Procesor obsahuje několik **registrů** 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.)

Činnost počítače

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, ...)

Činnost počítače

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.

Činnost počítače

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

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.

Stroj RAM a Turingův stroj

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

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.

Churchova-Turingova teze

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.

Algoritmicky neřešitelné problémy

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?

Halting problem

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(String kod, 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.

Bonusový příklad

Idea řešení:

Vypiš následující text dvakrát za sebou, podruhé ho uzavři do úvozovek:
"Vypiš následující text dvakrát za sebou, podruhé ho uzavři do úvozovek:"

Bonusový příklad

Tuto konstrukci je možné použít k alternativnímu důkazu nerozhodnutelnosti Halting problému:

Předpokládáme, že $H(p, x)$ řešící HP existuje.

Vytvoříme program, který:

- Do proměnné p si nageneruje svůj vlastní kód.
- Zavolá $H(p, x)$, kde x je jeho vlastní vstup.
- Pokud $H(p, x)$ vrátí true, skočí do nekonečné smyčky, v opačném případě se zastaví.

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 Σ.

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. redukcí.

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

$$x \in A \qquad \Leftrightarrow \qquad P(x) \in B$$

tj. pro vstup x je v problému A odpověď ANO právě tehdy, když pro vstup $P(x)$ je v problému B odpověď ANO.

Redukce mezi problémy

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.

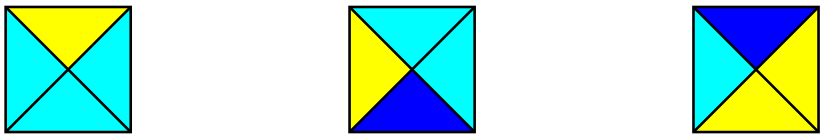
Další nerozhodnutelné problémy

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:

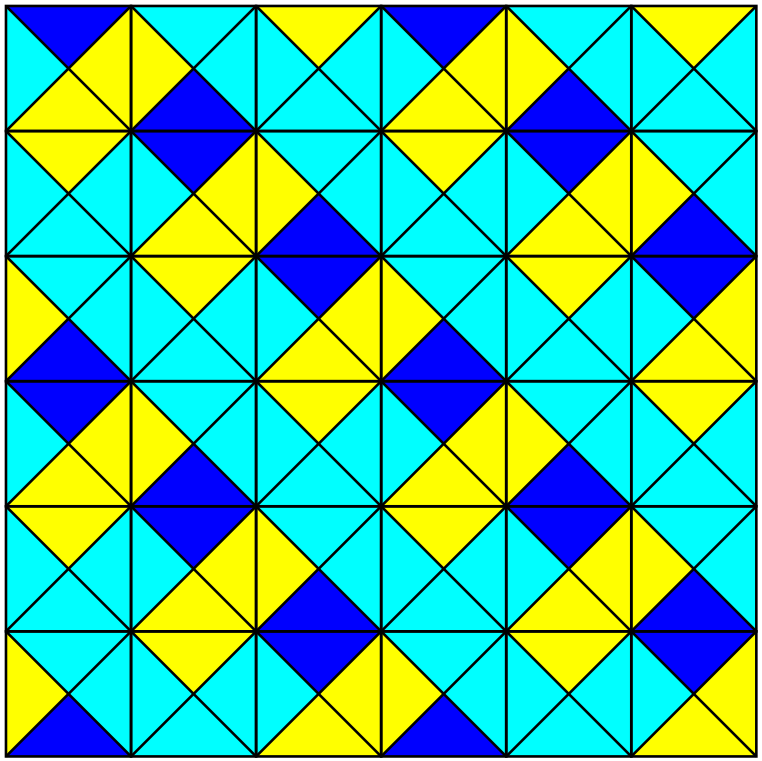


Otázka je, zda je možné použitím daných typů kachliček pokrýt každou libovolně velkou konečnou plochu tak, aby všechny kachličky spolu sousedily stejnými barvami.

Poznámka: Můžeme předpokládat, že máme v zásobě neomezené množství kachliček všech typů.

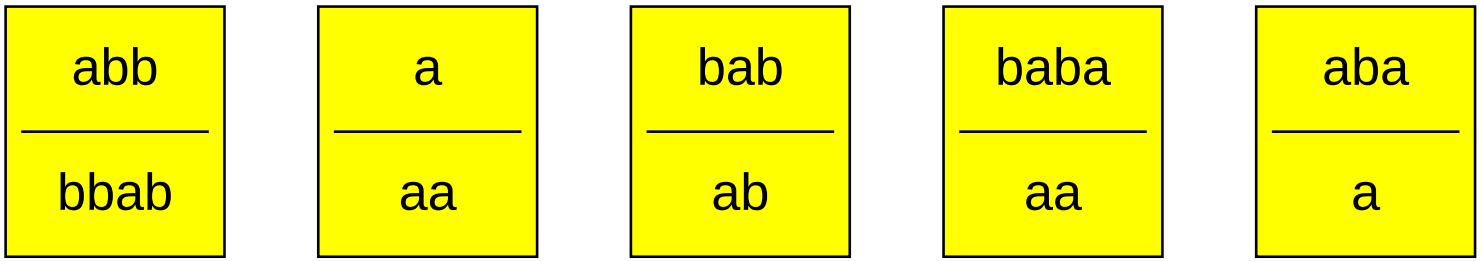
Kachličky není dovoleno otáčet.

Další nerozhodnutelné problémy

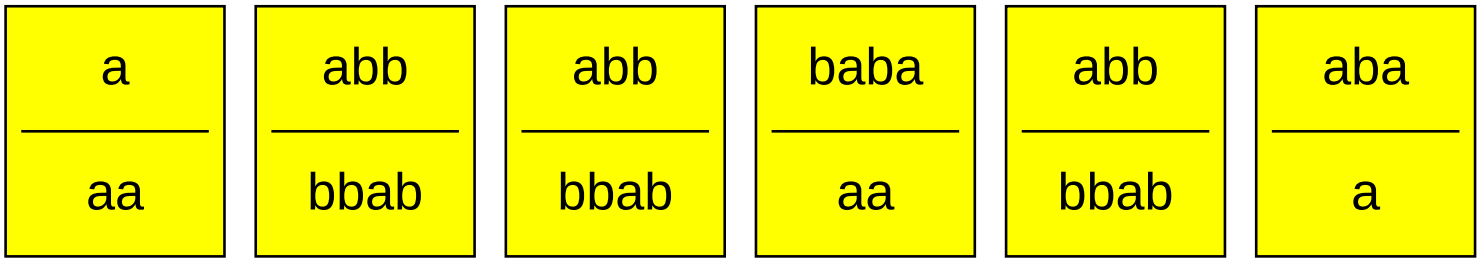


Další nerozhodnutelné problémy

Vstupem je množina typů kartiček, jako třeba:



Otázka je, zda je možné z těchto typů kartiček vytvořit neprázdnou konečnou posloupnost, kde zřetězením 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?

Další nerozhodnutelné problémy

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.

Rozhodnutelný problém

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:

$$\forall x \exists y \forall z((x + y = z) \wedge (y + 5 = x))$$

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 $s_{\mathcal{M}} : \mathbb{N} \to \mathbb{N}$, kde $s_{\mathcal{A}}(n)$ udává maximální velikost paměti ze všech výpočtů nad vstupy x délky n.

Prostorová (paměťová) složitost algoritmů

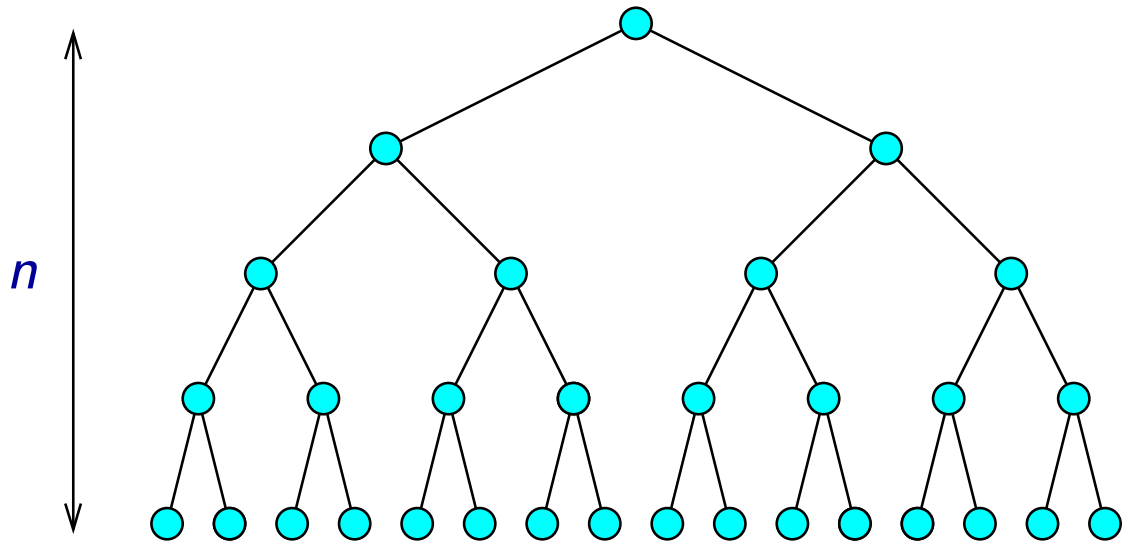
- Pro konkrétní problém můžeme mít dva algoritmy takové, že jeden má menší prostorovou složitost a druhý zase časovou složitost
- Prostorová složitost je někdy kritičtější než časová. Když dojde paměť, ve výpočtu se nedá pokračovat
- Je-li časová složitost algoritmu v $O(f(n))$ je i prostorová v $O(f(n))$ (počet buněk navšívených RAMem nemůže být větší než počet kroků, protože v každém kroku navšíví maximálně jednu buňku)

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.

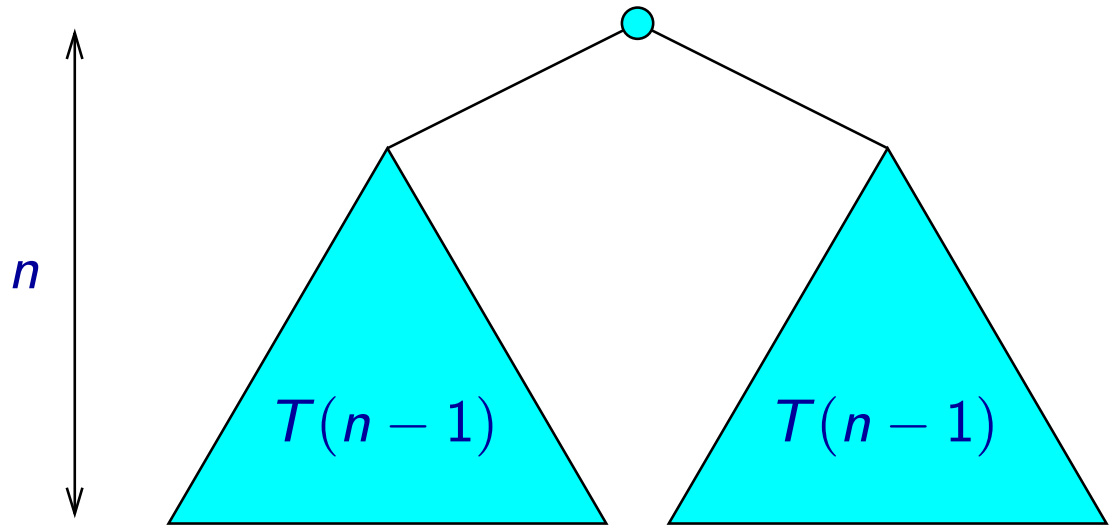
Příklad: Kolik vrcholů má úplný binární strom výšky n?



Rekurentní vztahy

$T(n)$ – počet vrcholů úplného binárního stromu výšky n

$$T(n) = \begin{cases} 1 & \text{pro } n = 0 \\ 2 \cdot T(n-1) + 1 & \text{pro } n > 0 \end{cases}$$



Rekurentní vztahy

$$T(n) = \begin{cases} 1 & \text{pro } n = 0 \\ 2 \cdot T(n-1) + 1 & \text{pro } n > 0 \end{cases}$$

$$T(0) = 1$$
$$T(1) = 2 \cdot T(0) + 1 = 2 \cdot 1 + 1 = 3$$
$$T(2) = 2 \cdot T(1) + 1 = 2 \cdot 3 + 1 = 7$$
$$T(3) = 2 \cdot T(2) + 1 = 2 \cdot 7 + 1 = 15$$
$$T(4) = 2 \cdot T(3) + 1 = 2 \cdot 15 + 1 = 31$$
$$T(5) = 2 \cdot T(4) + 1 = 2 \cdot 31 + 1 = 63$$
$$T(6) = 2 \cdot T(5) + 1 = 2 \cdot 63 + 1 = 127$$
$$T(7) = 2 \cdot T(6) + 1 = 2 \cdot 127 + 1 = 255$$
$$T(8) = 2 \cdot T(7) + 1 = 2 \cdot 255 + 1 = 511$$
$$T(9) = 2 \cdot T(8) + 1 = 2 \cdot 511 + 1 = 1023$$
$$T(10) = 2 \cdot T(9) + 1 = 2 \cdot 1023 + 1 = 2047$$

$\ldots$

Rekurentní vztahy

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.

Rekurentní vztahy

Příklad:

$$T(n) = \begin{cases} 1 & \text{pro } n = 0 \\ 2 \cdot T(n-1) + 1 & \text{pro } n > 0 \end{cases}$$

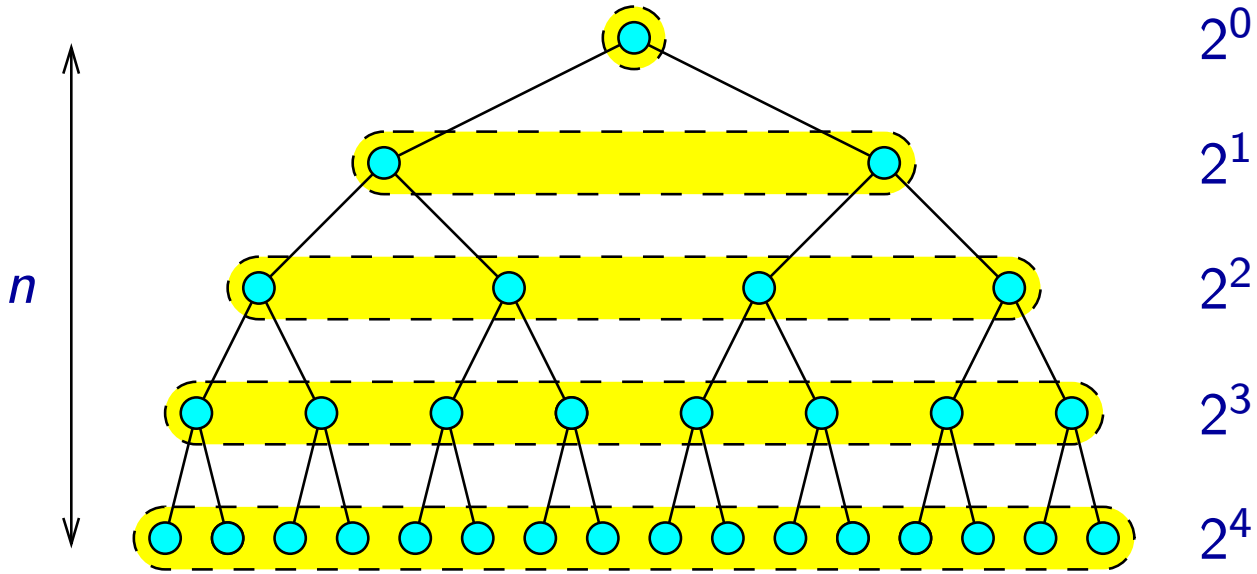
Uhodneme, že $T(n) = 2^{n+1} - 1$.

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.



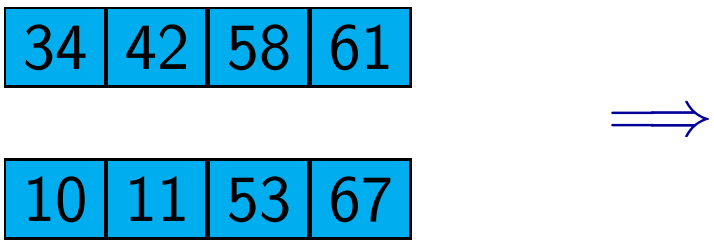
$$T(n) = \sum_{i=0}^{n} 2^i = \frac{2^{n+1} - 1}{2 - 1} = 2^{n+1} - 1$$

Rekurentní vztahy

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.

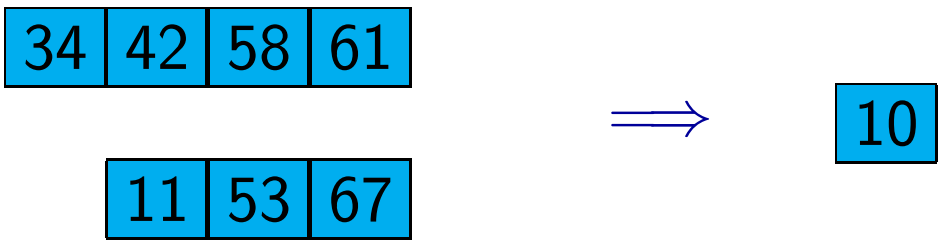


Rekurentní vztahy

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.

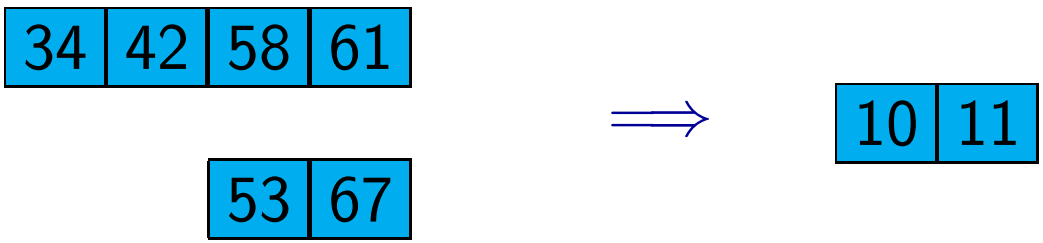


Rekurentní vztahy

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.

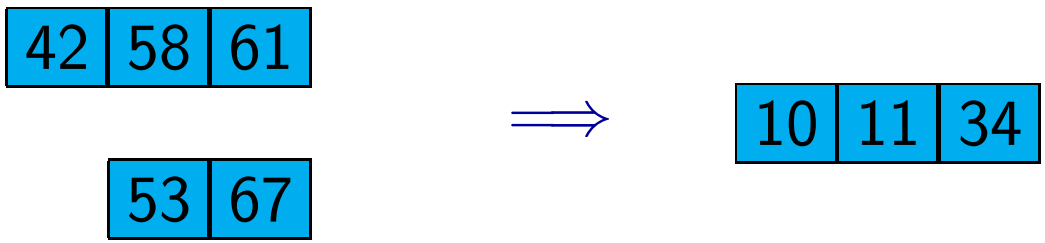


Rekurentní vztahy

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.

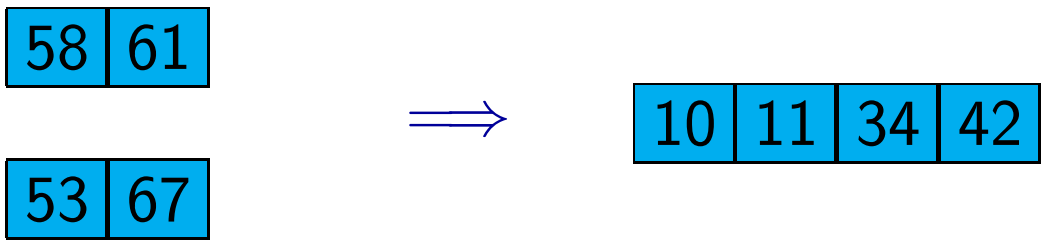


Rekurentní vztahy

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.

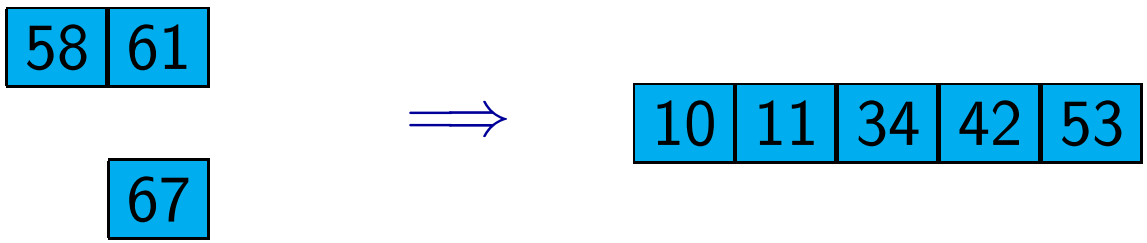


Rekurentní vztahy

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.

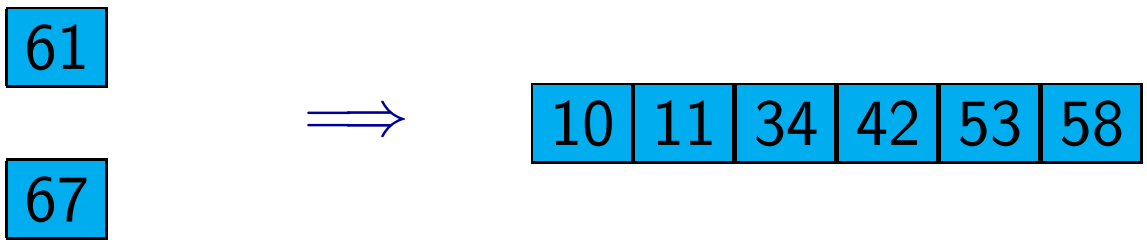


Rekurentní vztahy

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.

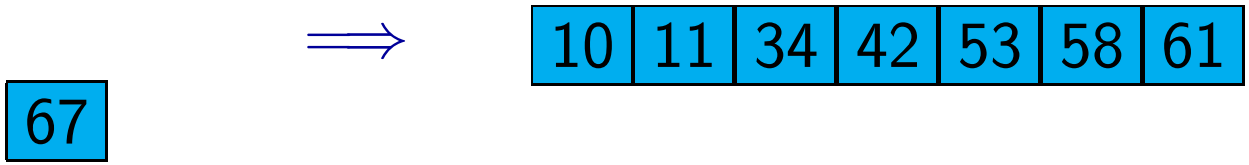


Rekurentní vztahy

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

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

MERGE-SORT(A, p, r)

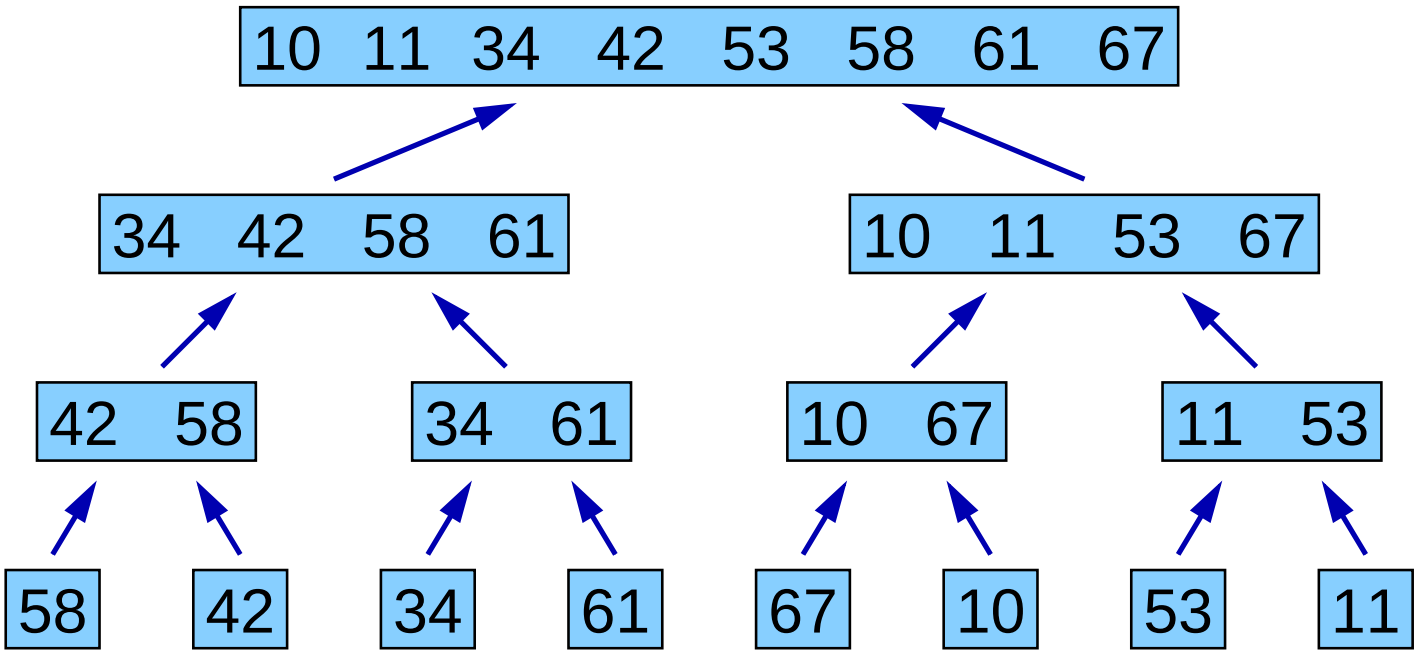
```
1  if r − p > 1
2      then q ← ⌊(p + r)/2⌋
3           MERGE-SORT(A, p, q)
4           MERGE-SORT(A, q, r)
5           MERGE(A, p, q, r)
```

Pro setřídění pole A, které obsahuje prvky $A[0], A[1], \cdots, A[n-1]$, 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



Rekurentní vztahy

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:

$$T(n) = \begin{cases} 4 & \text{pro } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + 5n + 11 & \text{pro } n > 1 \end{cases}$$

$$\begin{aligned}
T(1) &= 4 \\
T(2) &= T(\lceil 2/2 \rceil) + T(\lfloor 2/2 \rfloor) + 5 \cdot 2 + 11 = T(1) + T(1) + 21 = 29 \\
T(3) &= T(\lceil 3/2 \rceil) + T(\lfloor 3/2 \rfloor) + 5 \cdot 3 + 11 = T(2) + T(1) + 26 = 59 \\
T(4) &= T(\lceil 4/2 \rceil) + T(\lfloor 4/2 \rfloor) + 5 \cdot 4 + 11 = T(2) + T(2) + 31 = 89 \\
T(5) &= T(\lceil 5/2 \rceil) + T(\lfloor 5/2 \rfloor) + 5 \cdot 5 + 11 = T(3) + T(2) + 36 = 124 \\
T(6) &= T(\lceil 6/2 \rceil) + T(\lfloor 6/2 \rfloor) + 5 \cdot 6 + 11 = T(3) + T(3) + 41 = 159 \\
T(7) &= T(\lceil 7/2 \rceil) + T(\lfloor 7/2 \rfloor) + 5 \cdot 7 + 11 = T(4) + T(3) + 46 = 194 \\
&\dots
\end{aligned}$$

Rekurentní vztahy

- 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.

Rekurentní vztahy

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 $T(n) = O(n \log n)$, 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 $\log_2 n$.

Rekurentní vztahy

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 $T(1)$.

Rekurentní vztahy

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.

$$\begin{aligned} T(n) &\leq 2(c\lfloor n/2 \rfloor) + n \\ &\leq cn + n \\ &= O(n) \qquad \Longleftarrow \text{Chyba!} \end{aligned}$$

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.

Rekurentní vztahy

Lemma

Nechť $a_1, \ldots, a_k, c$ *jsou kladné konstanty takové, že* $a_1 + \ldots + a_k < 1$ *a pro funkci* $T : \mathbb{N} \to \mathbb{N}$ *platí*

$$T(n) \leq T(\lceil a_1 n \rceil) + T(\lceil a_2 n \rceil) + \ldots + T(\lceil a_k n \rceil) + cn\,.$$

Pak $T(n) = O(n)$.

Důkaz: Zvolme $\varepsilon > 0$ takové, že $a_1 + \ldots + a_k < 1 - 2\varepsilon$. Pak pro nějaké dostatečně velké n_0 platí pro všechna $n \geq n_0$

$$\lceil a_1 n \rceil + \ldots + \lceil a_k n \rceil \leq (1 - \varepsilon) n$$

Chceme ukázat, že pro nějaké vhodně zvolené d platí pro všechna $n \geq 1$

$$T(n) \leq dn$$

Rekurentní vztahy

- 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:

$$\begin{aligned} T(n) &\leq T(\lceil a_1 n \rceil) + \ldots + T(\lceil a_k n \rceil) + cn \\ &\leq d \cdot \lceil a_1 n \rceil + \ldots + d \cdot \lceil a_k n \rceil + cn \\ &= d \cdot (\lceil a_1 n \rceil + \ldots + \lceil a_k n \rceil) + cn \\ &\leq d \cdot (1 - \varepsilon) n + cn \\ &= dn - d\varepsilon n + cn \\ &= dn - (d\varepsilon - c) n \\ &\leq dn \end{aligned}$$

Poslední nerovnost platí, pokud d zvolíme tak, že $d\varepsilon > c$.

Lemma

Nechť $k \geq 2$ a $a_1, \ldots, a_k, c$ jsou kladné konstanty takové, že $a_1 + \ldots + a_k = 1$ a pro funkci $T : \mathbb{N} \to \mathbb{N}$ platí

$$T(n) \leq T(\lceil a_1 n \rceil) + T(\lceil a_2 n \rceil) + \ldots + T(\lceil a_k n \rceil) + cn\,.$$

Pak $T(n) = O(n \log n)$.

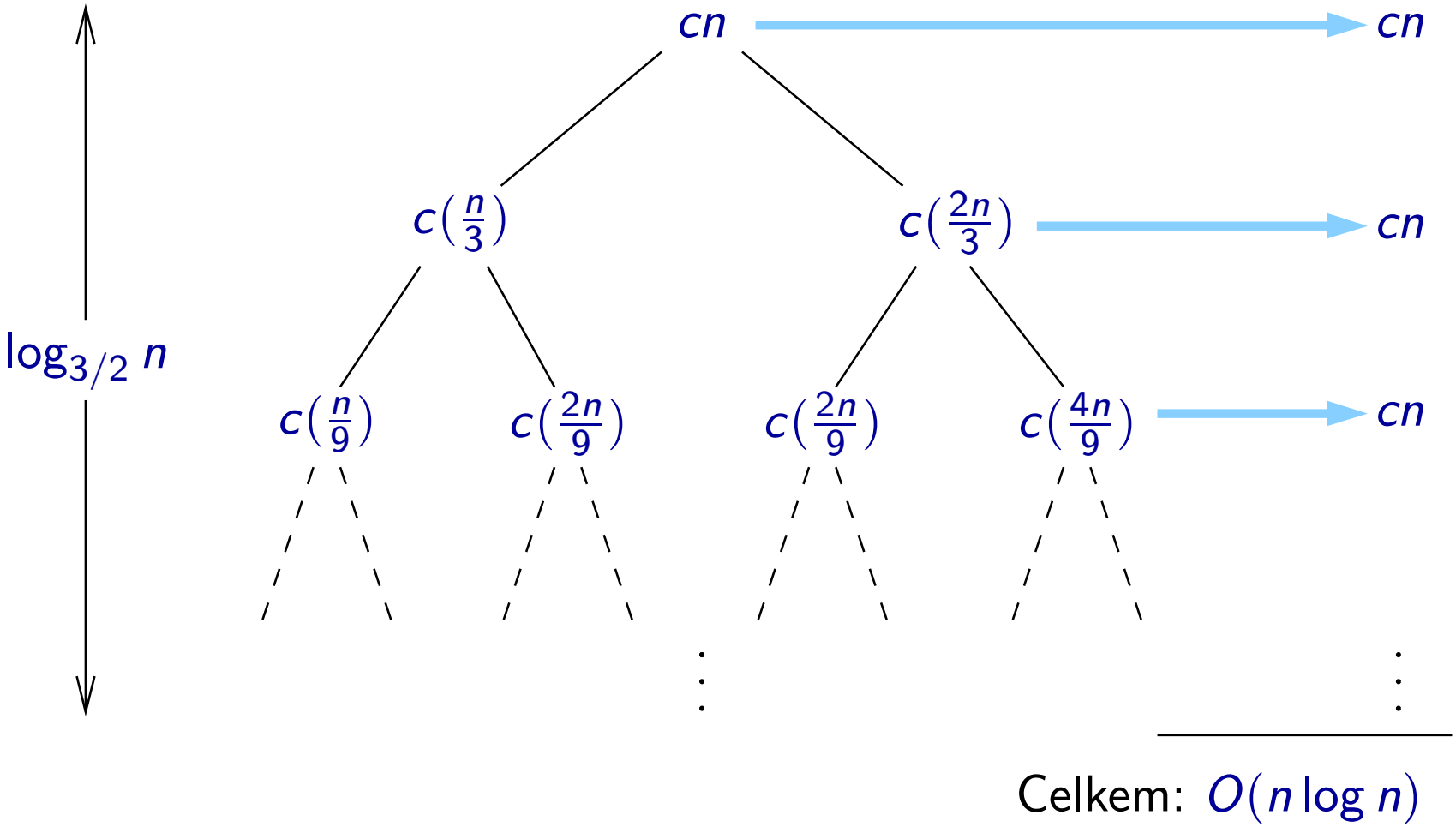
Důkaz by byl podobný jako v předchozím případě (o něco komplikovanější).

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, $f : \mathbb{N} \to \mathbb{N}$ je funkce a funkce $T : \mathbb{N} \to \mathbb{N}$ 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 $\lceil n/b \rceil$.

Složitost problémů

Složitost problémů

- 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?

Složitost problémů

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

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.

Třídy složitosti

Definice

Pro libovolnou funkci $f : \mathbb{N} \to \mathbb{N}$ 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 než existuje algoritmus s časovou složitostí $O(n^2)$
- $\mathcal{T}(n \log n)$ – třída všech problémů pro než existuje algoritmus s časovou složitostí $O(n \log n)$

Třídy složitosti

Definice

Pro libovolnou funkci $f : \mathbb{N} \to \mathbb{N}$ 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 než existuje algoritmus s prostorovou složitostí $O(n^2)$
- $\mathcal{S}(n \log n)$ – třída všech problémů pro než existuje algoritmus s prostorovou složitostí $O(n \log n)$

Třídy složitosti

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, vícepáskový Turingův stroj, ...).

Třídy složitosti

Pomocí tříd $\mathcal{T}(f(n))$ a $\mathcal{S}(f(n))$ můžeme definovat třídy PTIME a PSPACE jako

$$\text{PTIME} = \bigcup_{k \geq 0} \mathcal{T}(n^k) \qquad \text{PSPACE} = \bigcup_{k \geq 0} \mathcal{S}(n^k)$$

- 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.

Třídy složitosti

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.

Třídy složitosti

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.

Třídy složitosti

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.

Třídy složitosti

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

$$2^{2^{2^{\cdot^{\cdot^{\cdot^{2^{2^{O(n^k)}}}}}}}}}$$

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 $f : \mathbb{N} \to \mathbb{N}$ 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:

$$\text{PTIME} \subseteq \text{EXPTIME} \subseteq \text{2-EXPTIME} \subseteq \cdots$$

$$\text{LOGSPACE} \subseteq \text{PSPACE} \subseteq \text{EXPSPACE} \subseteq \text{2-EXPSPACE} \subseteq \cdots$$

Vztahy mezi třídami složitosti

Věta

Pro libovolnou prostorově zkonstruovatelnou funkci $f : \mathbb{N} \to \mathbb{N}$ 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 $f : \mathbb{N} \to \mathbb{N}$, 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.

Vztahy mezi třídami složitosti

Důsledek

Pro libovolné dvě funkce $f_1, f_2 : \mathbb{N} \to \mathbb{N}$ 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

$$\mathcal{S}(n^{\epsilon_1}) \subsetneq \mathcal{S}(n^{\epsilon_2})$$

- LOGSPACE $\subsetneq$ PSPACE
- PSPACE $\subsetneq$ EXPSPACE
- EXPSPACE $\subsetneq$ 2-EXPSPACE

Vztahy mezi třídami složitosti

Věta

Pro libovolnou časově zkonstruovatelnou funkci $t : \mathbb{N} \to \mathbb{N}$ 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 $t : \mathbb{N} \to \mathbb{N}$, 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 $t_1, t_2 : \mathbb{N} \to \mathbb{N}$ takové, že $t_1(n)$ je v $o(t_2(n)/\log t_2(n))$ a $t_2(n)$ je časově zkonstruovatelná, platí, že $\mathcal{T}(f_1(n)) \subsetneq \mathcal{T}(f_2(n))$.

Vztahy mezi třídami složitosti

Z předchozího plynou následující důsledky:

- Pro libovolná dvě reálná čísla $0 \leq \epsilon_1 < \epsilon_2$ platí

$$\mathcal{T}(n^{\epsilon_1}) \subsetneq \mathcal{T}(n^{\epsilon_2})$$

- PTIME $\subsetneq$ EXPTIME
- EXPTIME $\subsetneq$ 2-EXPTIME

Vztahy mezi třídami složitosti

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 řídící 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 stojem, počet bitů paměti použitých strojem RAM apod.).

Poznámka: Pokud máme abecedu Σ, kde $|\Sigma| = c$ tak:

- Počet slov délky n je c^n, tj. $2^{\Theta(n)}$.
- Počet slov délky nejvýše n je

$$\sum_{i=0}^{n} c^n = \frac{c^{n+1} - 1}{c - 1}$$

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 $f : \mathbb{N} \to \mathbb{N}$ platí, že $\mathcal{S}(f(n)) \subseteq \mathcal{T}(2^{f(n)})$.

Vztahy mezi třídami složitosti

Z předchozího plynou následující důsledky:

$$\begin{aligned} \text{LOGSPACE} &\subseteq \text{PTIME} \\ \text{PSPACE} &\subseteq \text{EXPTIME} \\ \text{EXPSPACE} &\subseteq \text{2-EXPTIME} \\ &\vdots \end{aligned}$$

Vztahy mezi třídami složitosti

Shrnutí:

$$\text{LOGSPACE} \subseteq \text{PTIME} \subseteq \text{PSPACE} \subseteq \text{EXPTIME} \subseteq \text{EXPSPACE} \subseteq$$
$$\subseteq \text{2-EXPTIME} \subseteq \text{2-EXPSPACE} \subseteq \cdots \subseteq \text{ELEMENTARY}$$

Navíc je známo, že:

- $\text{PTIME} \subsetneq \text{EXPTIME} \subsetneq \text{2-EXPTIME} \subsetneq \cdots$
- $\text{LOGSPACE} \subsetneq \text{PSPACE} \subsetneq \text{EXPSPACE} \subsetneq \text{2-EXPSPACE} \subsetneq \cdots$

Horní a dolní odhady složitosti problémů

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 PTIME.
- Problém ekvivalence dvou regulárních výrazů je v EXPSPACE.

Pokud chceme zjistit nějaký horní odhad složitosti problému, stačí ukázat, že existuje algoritmus s danou složitostí.

Horní a dolní odhady složitosti problémů

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.

Dolní odhad pro problém třídění

Problém „Třídění“

Vstup: Posloupnost prvků $a_1, a_2, \ldots, a_n$.

Výstup: Prvky $a_1, a_2, \ldots, a_n$ setříděné 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 $\{1, 2, \ldots, n\}$.

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

$$t \geq \log_c n!$$

Dolní odhad pro problém třídění

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

$$\log n! \leq \log n^n \leq \log(n!)^2$$

$$\log n! \leq n \log n \leq 2 \log n!$$

$$\frac{1}{2} n \log n \leq \log n! \leq n \log n$$

NP-úplnost

Rozhodovací problémy

Definice

Rozhodovací problém je takový, kde je množina možných výstupů dvouprvková $\{\textsc{Ano}, \textsc{Ne}\}$.

- 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**
- Γ je konečná množina **páskových symbolů**
- $\Sigma \subseteq \Gamma, \Sigma \neq \emptyset$ je konečná množina **vstupních symbolů**
- $\delta : (Q - F) \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{-1, 0, +1\})$ 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 vypočtu 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

Nedeterminismus

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.

Význam nedeterminismu

- 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.

Nedeterministické třídy složitosti

Definice

Pro funkci $f : \mathbb{N} \to \mathbb{N}$ 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 $f : \mathbb{N} \to \mathbb{N}$ rozumíme **třídou prostorové složitosti** $\mathcal{NS}(f)$ množinu těch problémů, které jsou řešeny nedeterminictickými RAMy s prostorovou složitostí v $O(f(n))$.

Třída NPTIME

Definice

$$\text{NPTIME} = \cup_{k=0}^{\infty} \mathcal{NT}(n^k)$$

- 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.

Třída NPTIME

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.

Třída NPTIME

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.

Třídy složitosti

- 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 nepředávájí, 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.

$$\text{PTIME} \subseteq \text{NPTIME} \subseteq \text{PSPACE} = \text{NPSPACE}$$

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 P na x je ANO právě tehdy, když existuje y takové, že $\mathcal{A}$ vrací ANO pro vstup (x, y).

- y je jakási "nápověda", která se často nazývá **svědek**.
- Není důležité, jak je těžké svědka najít. Jen když je znám, musí běžet algoritmus v polynomiálním čase.

Třída NPTIME

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 $\rho : V_1 \to V_2$ takové, že $(x, y) \in E_1 \Leftrightarrow (\rho(x), \rho(y)) \in E_2$. NE pokud takové zobrazení neexistuje.

Svědkem je v tomto případě zobrazení ρ. Algoritmus pouze ověří, jestli hrany mezi vrcholy v G_1 odpovídají příslušným hranám v G_2.

Třída NPTIME

Problém „loupežníka“

Vstup: Množina přirozených čísel $A = \{a_1, a_2, \ldots, a_n\}$

Výstup: ANO, jestliže je možné najít množinu X tak, že $X \subset A$, $\sum_{a_i \in X} a_i = \sum_{a_j \in A - X} a_j$.

Svědkem je v tomto případě množina X a algoritmus kontroluje, že je podmnožinou A a součty se rovnají.

Třída NPTIME

- Obě defnice 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 (víc 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 nagenerovat svědka (v polynomiálním čase)
  - Dále již nedeterministický algoritmus pracuje s vygenerovaným svědkem stejně, jako deterministický

Polynomiální převoditelnost

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 $P_1 \in \text{NPTIME}$

Polynomiální převoditelnost

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)

Polynomiální převoditelnost

- 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.

NP-úplnost

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'.

NP-úplnost

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-úplnost

- 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, neznamená nic pro složitost NP-těžkých problémů
- Otázka jestli $\mathrm{P} = \mathrm{NP}$ je jednou z nejznámějších dlouhodobě otevřených otázek teoretické informatiky
- Předpokládá se $\mathrm{P} \neq \mathrm{NP}$, 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 Φ v konjunktivní normální formě

Výstup: ANO, pokud existuje ohodnocení μ 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.

Cookova věta

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 algortimus jen kontroluje, jesli je ohodnocení správné).
- Přesný důkaz NP-obtížnosti není úplně snadný, musí se ošetřit mnoho drobností

Cookova věta

- 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				n^k
0	#	(q_0, w_1)	w_2	$\cdots$	w_n	# $\cdots$ #	#
i	#	α_1	α_2	$\cdots$	α_n	$\alpha_{n+1}\cdots$	#
n^k	#	β_1	β_2	$\cdots$	β_n	$\beta_{n+1}\cdots$	#

Cookova věta

- 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 $0 \leq j \leq n^k$, $0 \leq j \leq n^k$, $s \in \Gamma$ 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.
- Formuli $\phi_{\mathcal{M},w}$ vytvoříme jako

$$\phi_{\mathcal{M},w} = \phi_{cell} \wedge \phi_{start} \wedge \phi_{move} \wedge \phi_{accept}$$

Cookova věta

ϕ_{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.

Cookova věta

ϕ_{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.

$$\phi_{start} = x_{0,0,\#} \wedge x_{0,1,(q_0,w_1)} \wedge x_{0,2,w_2} \wedge \ldots \wedge x_{0,n,w_n} \wedge x_{0,n+1,\#} \wedge \ldots \wedge x_{0,n^k,\#}$$

Cookova věta

ϕ_{accept} musí zajistit, že na posledním řádku tabulky se vyskytuje stav q_{ANO}. Předpokládáme, že pokud Turingův stroj skončí výpočet dřív než po n^k krocích, všechny následující konfigurace (řádky tabulky) jsou stejné.

$$\phi_{accept} = \bigvee_{1 \leq j \leq n^k, w \in \Gamma} x_{n^k, j, (q_{\text{ANO}}, w)}$$

Cookova věta

ϕ_{move} musí zajistit, že každá následující konfigurace vznikne z předchozí spravným způsobem.

- Musíme prozkoumat všechna "okna" o dvou řádcích a třech sloupcích.
- Pokud horní řádek neobsahuje stav, muže 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

$$\phi_{move} = \bigwedge_{1 \leq i,j \leq n^k} \bigvee_{(a_1,\ldots,a_6) \in SO} (x_{i-1,j-1,a_1} \wedge \ldots x_{i,j+1,a_6})$$

Cookova věta

- 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 nesplnitelná, 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

NP-úplné problémy

NP-úplnost

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.

Polynomiální převoditelnost

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'.

- Algoritmus převodu musí být obecný, tedy ke každému možnému vstupu problému P_1 umět sestrojit 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 Φ v konjunktivní normální formě

Výstup: ANO, pokud existuje ohodnocení μ 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 Φ v konjunktivní normální formě taková, že každá klauzule obsahuje právě 3 literály

Výstup: ANO, pokud existuje ohodnocení μ 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.

3-SAT - Splnitelnost logických formulí

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

Převod SAT na 3-SAT

Algoritmus převodu musí k obecné formuli v konjunktivní normální formě sestrojit 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 $(x_1 \vee x_2 \vee \ldots \vee x_n)$ sestrojíme $n - 2$ klauzulí:
  $(x_1 \vee x_2 \vee y_1) \wedge (\neg y_1 \vee x_3 \vee y_2) \wedge (\neg y_2 \vee x_4 \vee y_3) \wedge \ldots \wedge (\neg y_{n-3} \vee x_{n-1} \vee x_n)$,
  kde $x_1, x_2, \ldots, x_n$ jsou literály (proměnné nebo negace proměnných) a $y_1, y_2, \ldots, y_{n-3}$ jsou nově přidané proměnné.

Převod SAT na 3-SAT

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:

Převod SAT na 3-SAT

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

Převod SAT na 3-SAT

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 $(\neg a_1 \vee \neg a_1 \vee a_3)$

Převod SAT na 3-SAT

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 $(a_5 \vee a_5 \vee a_5)$

Převod SAT na 3-SAT

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
  $(\neg a_1 \vee \neg a_3 \vee y_1) \wedge (\neg y_1 \vee \neg a_4 \vee y_2) \wedge (\neg y_2 \vee a_5 \vee \neg a_5)$

Převod SAT na 3-SAT

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 $(a_1 \vee \neg a_2 \vee a_4) \wedge (\neg a_1 \vee \neg a_1 \vee a_3) \wedge (\neg a_1 \vee \neg a_3 \vee y_1) \wedge (\neg y_1 \vee \neg a_4 \vee y_2) \wedge (\neg y_2 \vee a_5 \vee \neg a_5) \wedge (a_5 \vee a_5 \vee a_5)$

Převod SAT na 3-SAT

- 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.

Převod SAT na 3-SAT

- 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 $(x_1 \vee x_2 \vee \ldots \vee x_n)$ přiřadit 1.

Převod SAT na 3-SAT

- 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 $(x_1 \vee x_2 \vee \ldots \vee x_n)$ přiřadit 1.
  - Stejné ohodnocení znamená, že v $(x_1 \vee x_2 \vee y_1) \wedge (\neg y_1 \vee x_3 \vee y_2) \wedge$ $\wedge (\neg y_2 \vee x_4 \vee y_3) \wedge \ldots \wedge (\neg y_{n-3} \vee x_{n-1} \vee x_n)$ bude alespoň jedna klauzule mít hodnotu 1 bez ohledu na hodnoty $y_1, \ldots, y_{n-3}$

Převod SAT na 3-SAT

- 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 $(x_1 \vee x_2 \vee \ldots \vee x_n)$ přiřadit 1.
  - Stejné ohodnocení znamená, že v $(x_1 \vee x_2 \vee y_1) \wedge (\neg y_1 \vee x_3 \vee y_2) \wedge$ $\wedge (\neg y_2 \vee x_4 \vee y_3) \wedge \ldots \wedge (\neg y_{n-3} \vee x_{n-1} \vee x_n)$ bude alespoň jedna klauzule mít hodnotu 1 bez ohledu na hodnoty $y_1, \ldots, y_{n-3}$
  - Pokud to není první klauzule z nově vzniklých, je v ní je literál $\neg y_{i-1}$, který můžeme ohodnotit 0 a v předchozí klauzuli je y_{i-1} ohodnocena 1 a hodnota této klauzule je tedy také 1. Obdobně nejde-li o poslední klauzuli, je v ní je literál $\neg y_i$, který můžeme ohodnotit 0 a v následující klauzuli je y_i ohodnocena 1 a hodnota klauzule je 1.

Převod SAT na 3-SAT

- 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 $(x_1 \vee x_2 \vee \ldots \vee x_n)$ přiřadit 1.
  - Stejné ohodnocení znamená, že v $(x_1 \vee x_2 \vee y_1) \wedge (\neg y_1 \vee x_3 \vee y_2) \wedge$ $\wedge (\neg y_2 \vee x_4 \vee y_3) \wedge \ldots \wedge (\neg y_{n-3} \vee x_{n-1} \vee x_n)$ bude alespoň jedna klauzule mít hodnotu 1 bez ohledu na hodnoty $y_1, \ldots, y_{n-3}$
  - Pokud to není první klauzule z nově vzniklých, je v ní je literál $\neg y_{i-1}$, který můžeme ohodnotit 0 a v předchozí klauzuli je y_{i-1} ohodnocena 1 a hodnota této klauzule je tedy také 1. Obdobně nejde-li o poslední klauzuli, je v ní je literál $\neg y_i$, který můžeme ohodnotit 0 a v následující klauzuli je y_i 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.

Převod SAT na 3-SAT

- Pokud vznikne splnitelná formule byla i původní splnitelná:
  - Ohodnocení, které přiřadí formuli $(x_1 \vee x_2 \vee y_1) \wedge (\neg y_1 \vee x_3 \vee y_2) \wedge$ $\wedge (\neg y_2 \vee x_4 \vee y_3) \wedge \ldots \wedge (\neg y_{n-3} \vee x_3 \vee y_2)$ 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.

Převod SAT na 3-SAT

- Pokud vznikne splnitelná formule byla i původní splnitelná:
  - Ohodnocení, které přiřadí formuli $(x_1 \vee x_2 \vee y_1) \wedge (\neg y_1 \vee x_3 \vee y_2) \wedge$ $\wedge (\neg y_2 \vee x_4 \vee y_3) \wedge \ldots \wedge (\neg y_{n-3} \vee x_3 \vee y_2)$ 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 odpověd ANO pro 3-SAT a opačně
- Délka vytvořené formule nebude více než trojnásobně delší než původní
- Problém 3-SAT je tedy NP-úplný

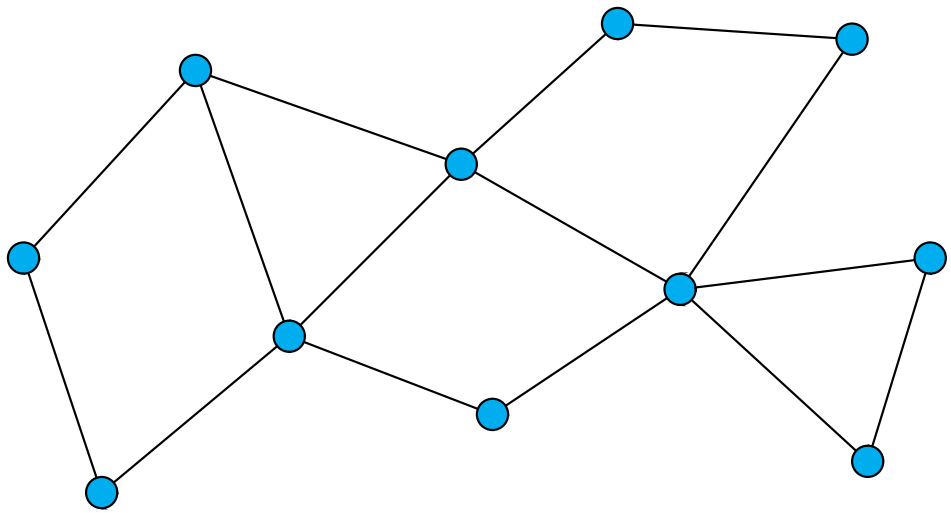
Barvení grafu 3 barvami

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:



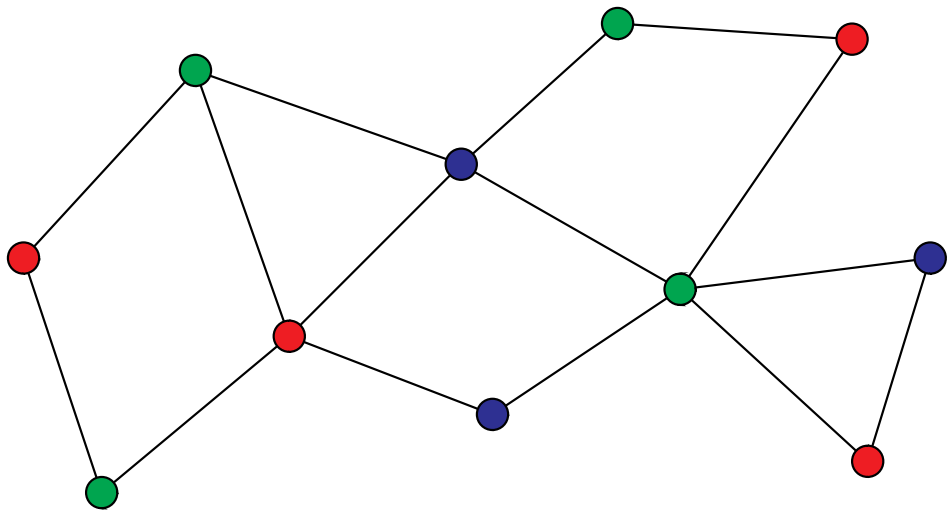
Barvení grafu 3 barvami

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: Ano

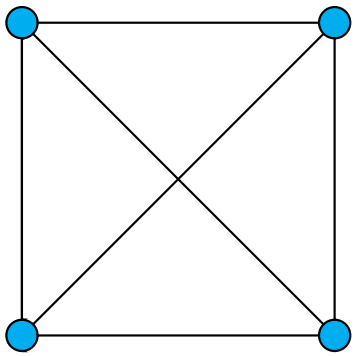
Barvení grafu 3 barvami

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:



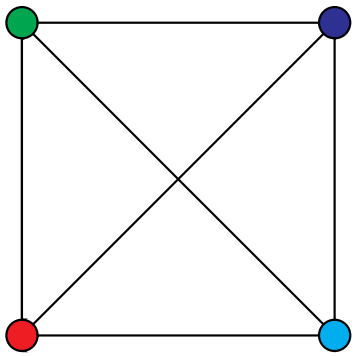
Barvení grafu 3 barvami

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

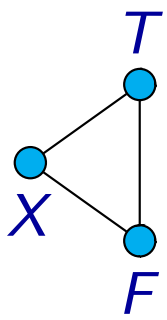
Převod 3-SAT na 3-COL

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

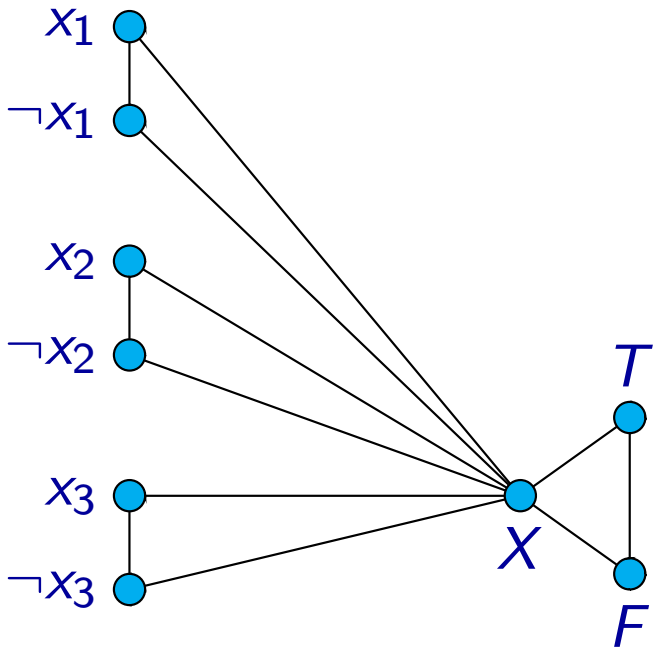
Převod 3-SAT na 3-COL

Příklad: Uvažujme formuli $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_3)$



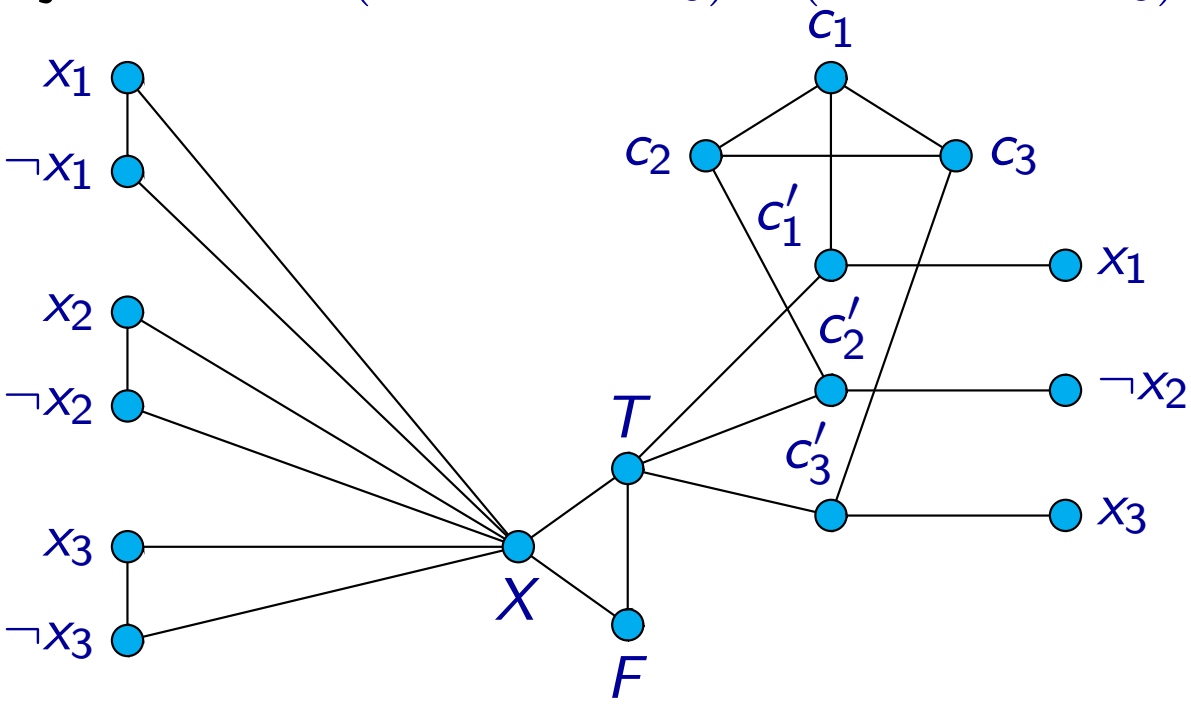
Převod 3-SAT na 3-COL

Příklad: Uvažujme formuli $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_3)$



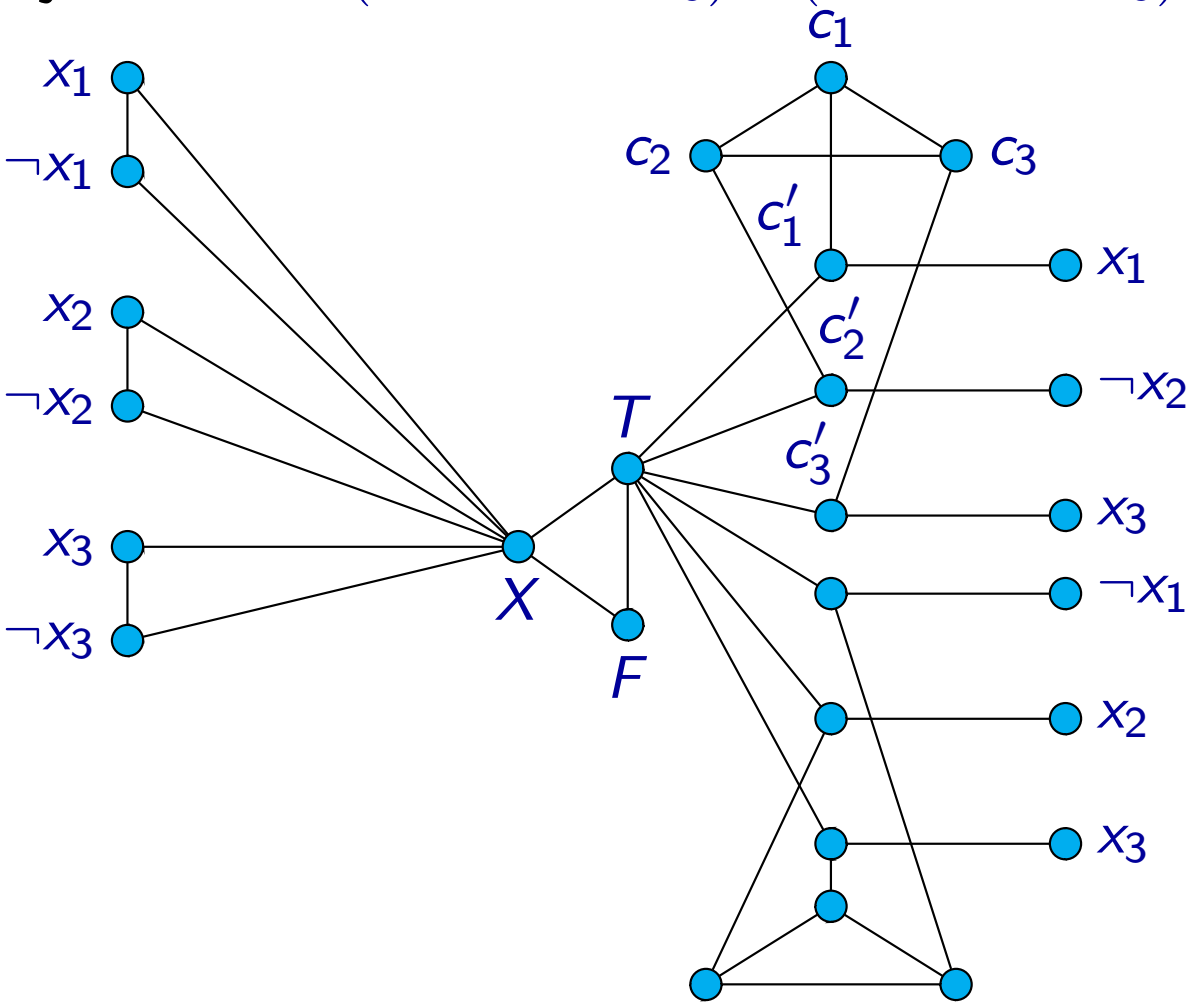
Převod 3-SAT na 3-COL

Příklad: Uvažujme formuli $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_3)$



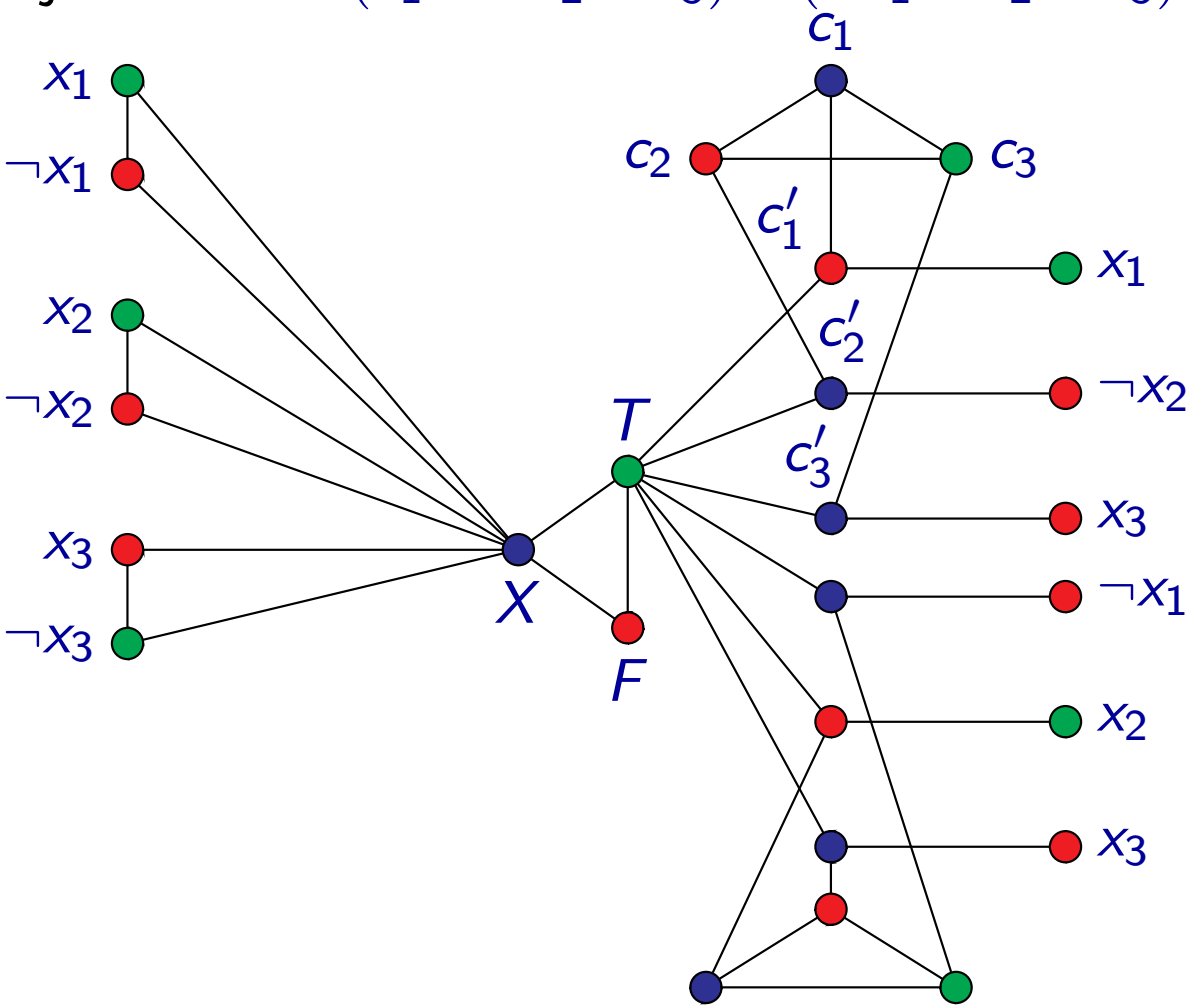
Převod 3-SAT na 3-COL

Příklad: Uvažujme formuli $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_3)$



Převod 3-SAT na 3-COL

Příklad: Uvažujme formuli $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_3)$



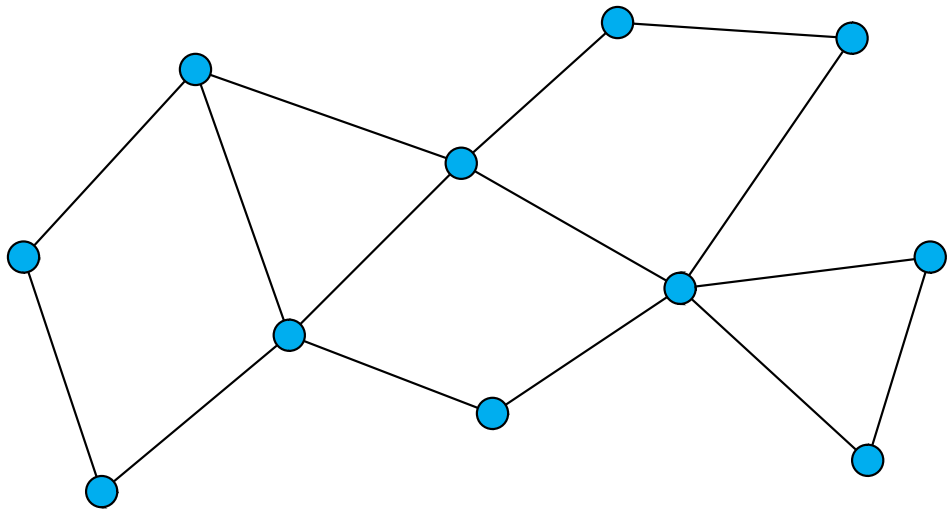
Nezávislá množina

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: $k = 5$



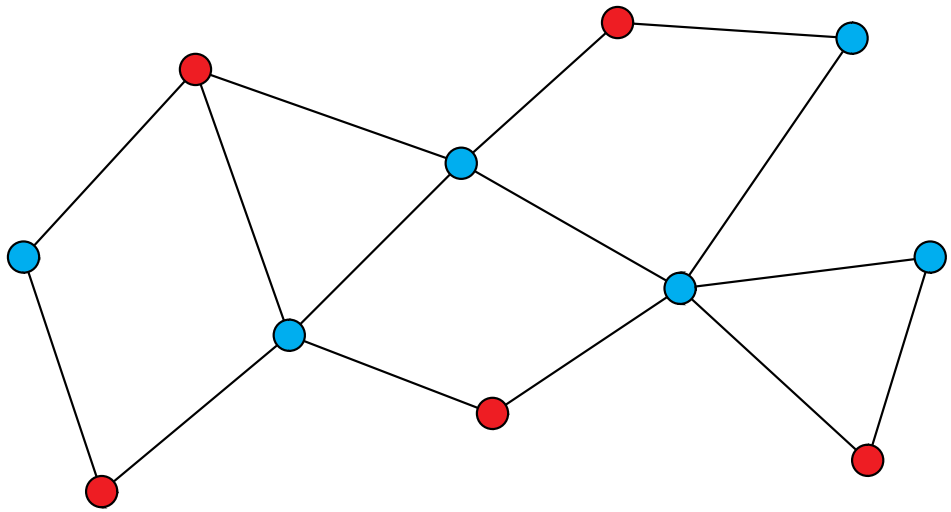
Nezávislá množina

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

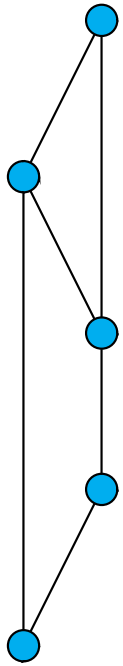
Převod 3-COL na IS

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 $|G|$

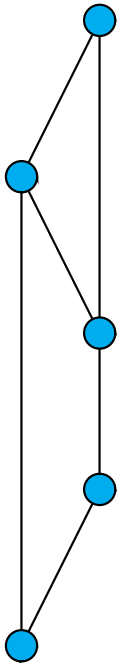
Převod 3-COL na IS

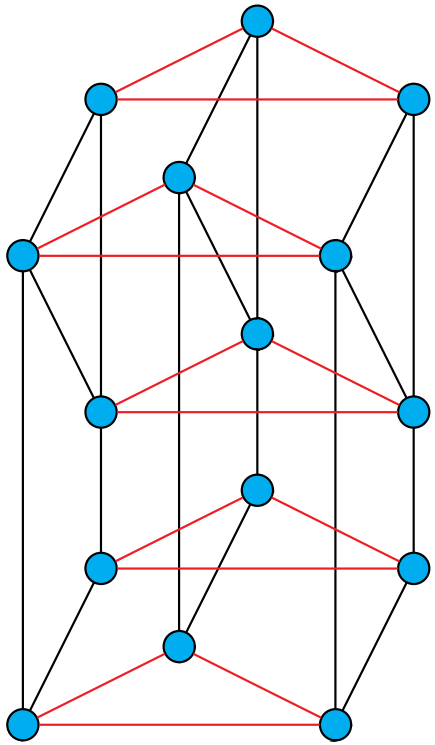
Příklad:



Převod 3-COL na IS

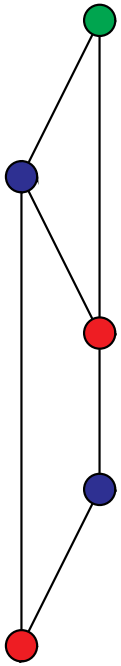
Příklad:

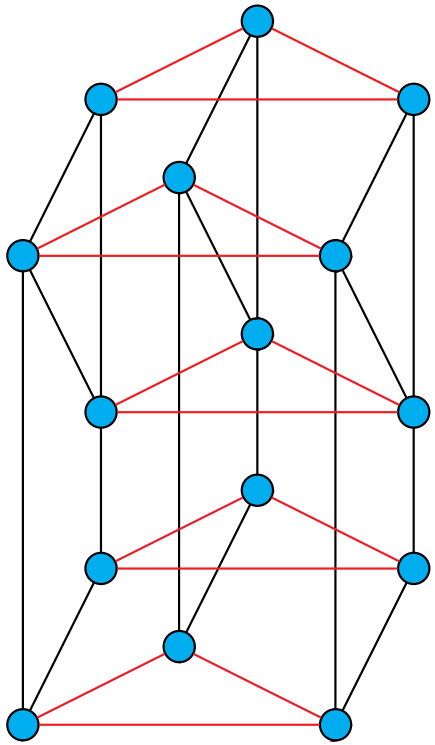




Převod 3-COL na IS

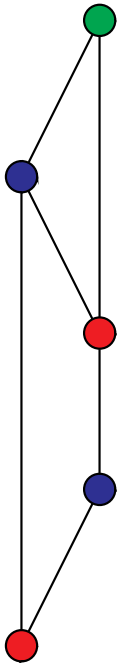
Příklad:

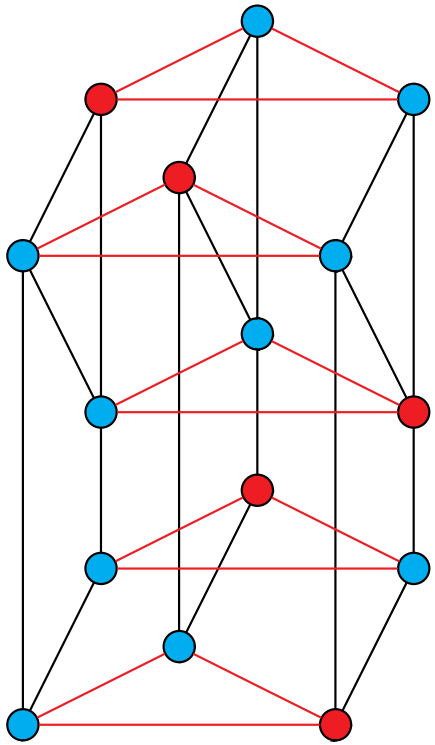




Převod 3-COL na IS

Příklad:





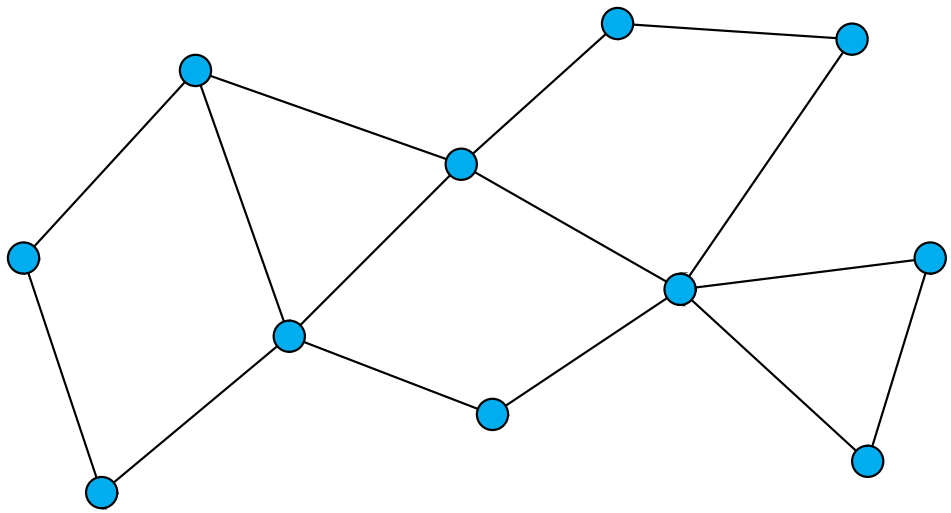
Vrcholové pokrytí

IS - 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: $k = 6$



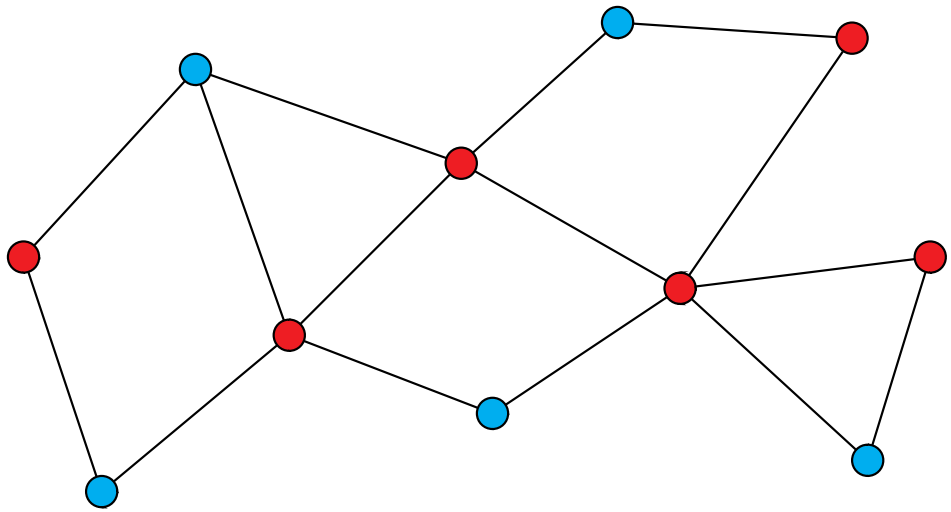
Vrcholové pokrytí

IS - 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 $n = |G| - k$

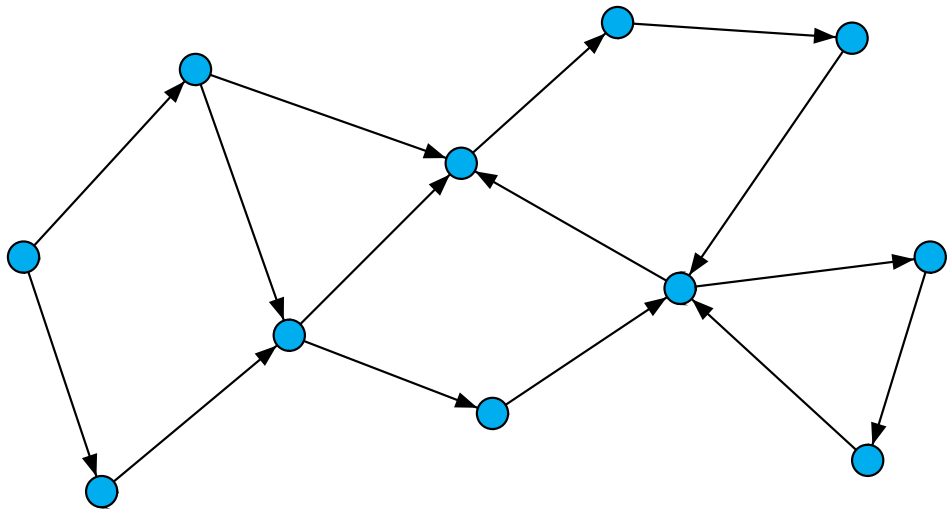
Hamiltonovský cyklus

HC - Problém „Hamiltonovský cyklus“

Vstup: Orientovaný graf G

Výstup: Existuje v grafu Hamiltonovský cyklus (orientovaná kružnice procházející každý vrchol právě jednou)?

Příklad:



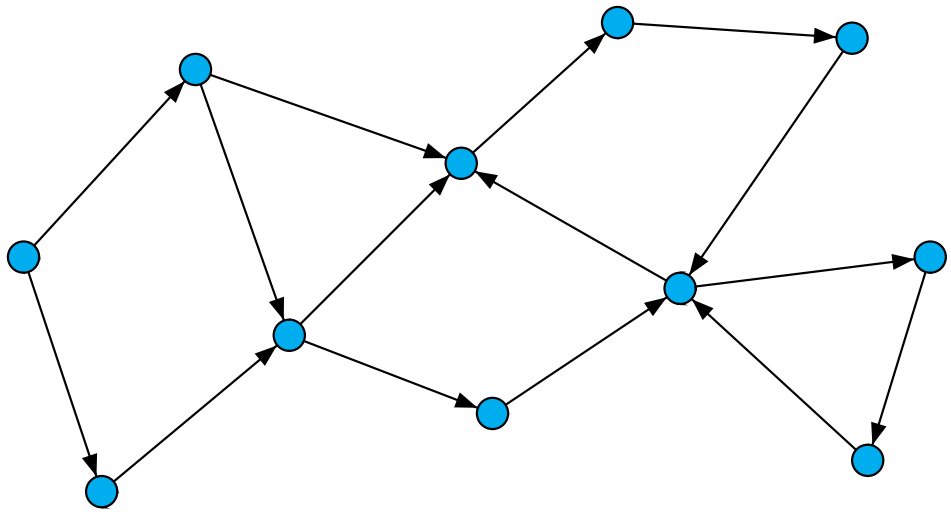
Hamiltonovský cyklus

HC - Problém „Hamiltonovský cyklus“

Vstup: Orientovaný graf G

Výstup: Existuje v grafu Hamiltonovský cyklus (orientovaná kružnice procházející každý vrchol právě jednou)?

Příklad:



Odpověď: NE

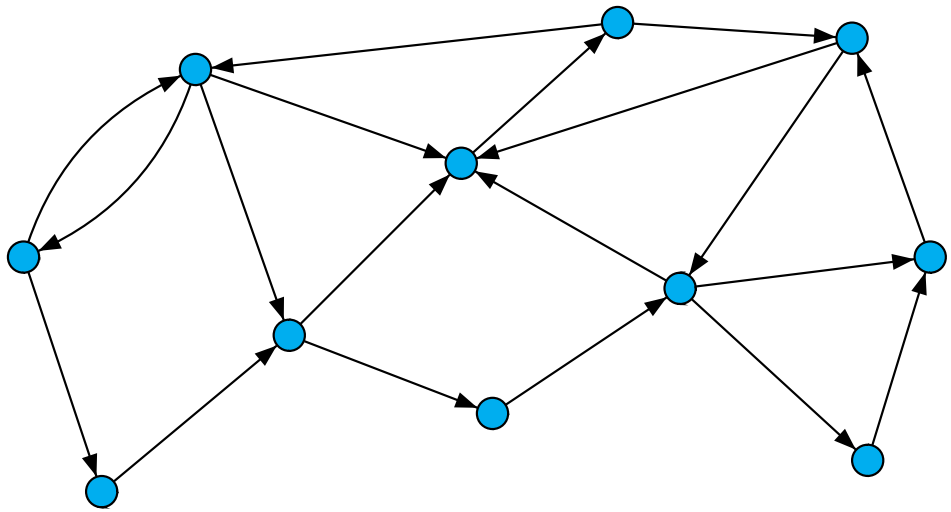
Hamiltonovský cyklus

HC - Problém „Hamiltonovský cyklus“

Vstup: Orientovaný graf G

Výstup: Existuje v grafu Hamiltonovský cyklus (orientovaná kružnice procházející každý vrchol právě jednou)?

Příklad:



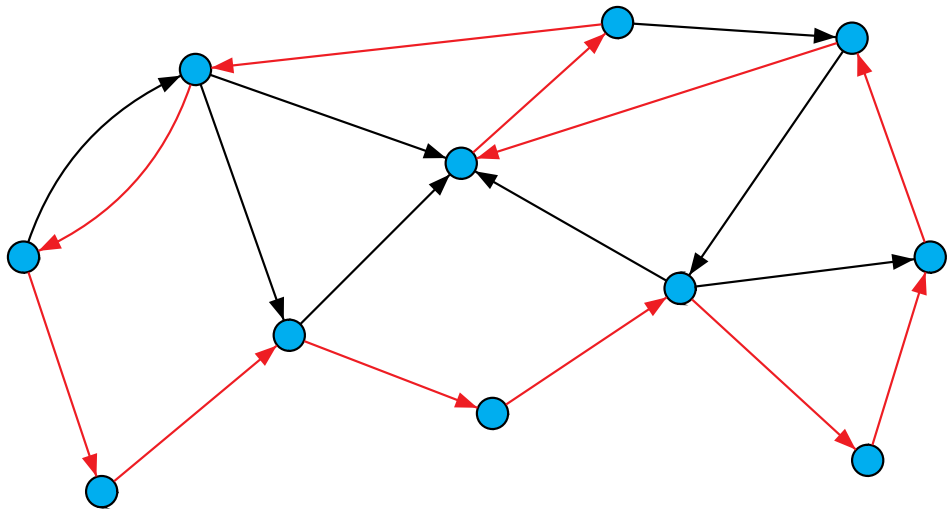
Hamiltonovský cyklus

HC - Problém „Hamiltonovský cyklus“

Vstup: Orientovaný graf G

Výstup: Existuje v grafu Hamiltonovský cyklus (orientovaná kružnice procházející každý vrchol právě jednou)?

Příklad:



Odpověď: ANO

Hamiltonovský cyklus

- 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

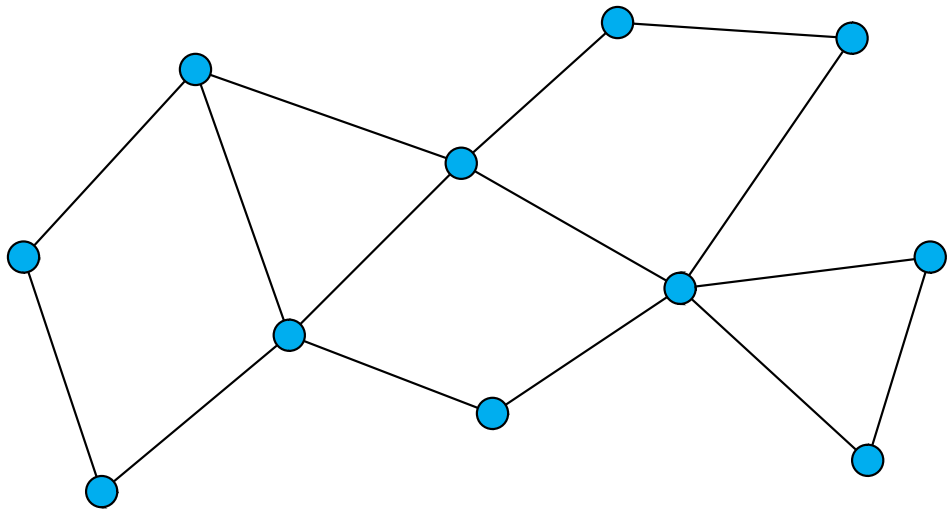
Hamiltonovská kružnice

HK - Problém „Hamiltonovská kružnice“

Vstup: Neorientovaný graf G

Výstup: Existuje v grafu Hamiltonovská kružnice (neorientovaná kružnice procházející každý vrchol právě jednou)?

Příklad:



Odpověď: NE

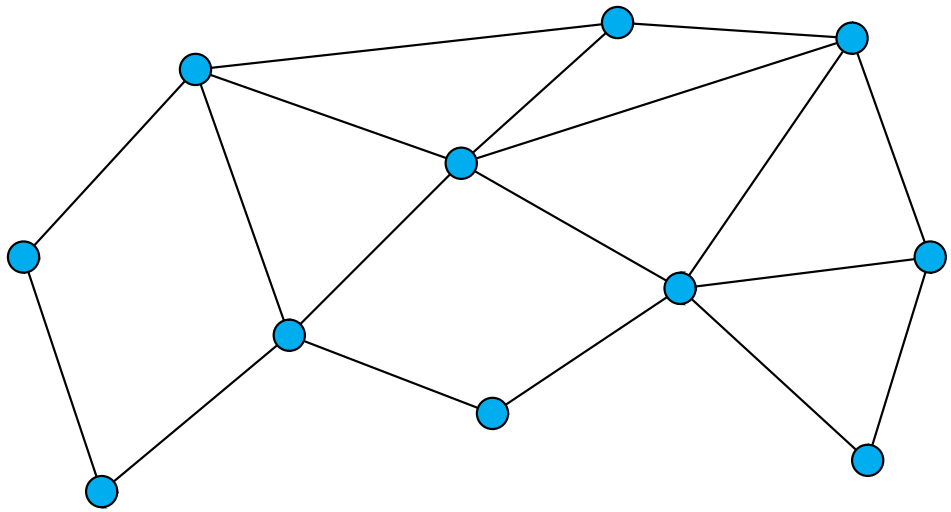
Hamiltonovská kružnice

HK - Problém „Hamiltonovská kružnice“

Vstup: Neorientovaný graf G

Výstup: Existuje v grafu Hamiltonovská kružnice (neorientovaná kružnice procházející každý vrchol právě jednou)?

Příklad:



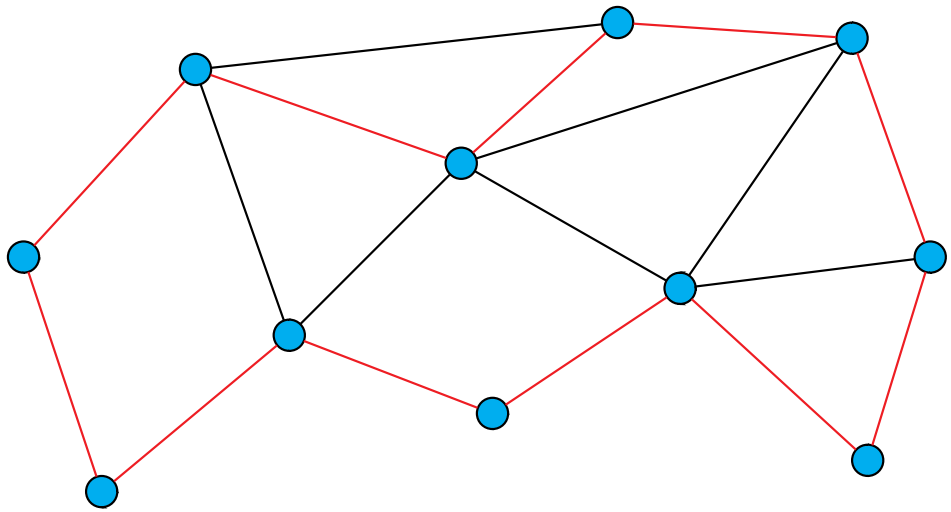
Hamiltonovská kružnice

HK - Problém „Hamiltonovská kružnice“

Vstup: Neorientovaný graf G

Výstup: Existuje v grafu Hamiltonovská kružnice (neorientovaná kružnice procházející každý vrchol právě jednou)?

Příklad:



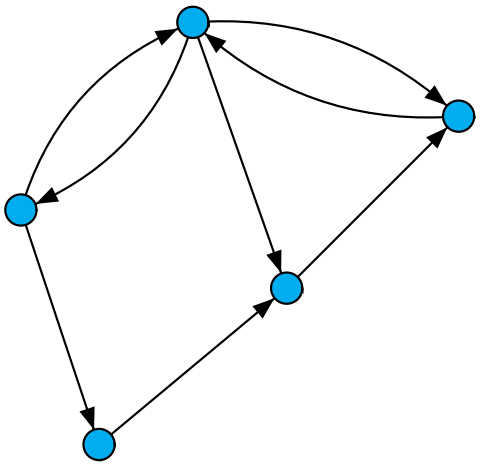
Odpověď: ANO

Hamiltonovská kružnice

- 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)

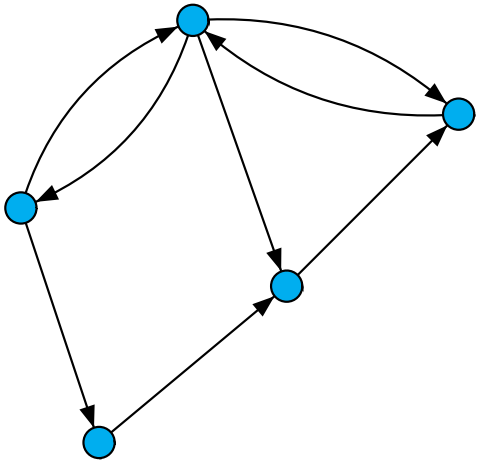
Převod HC na HK

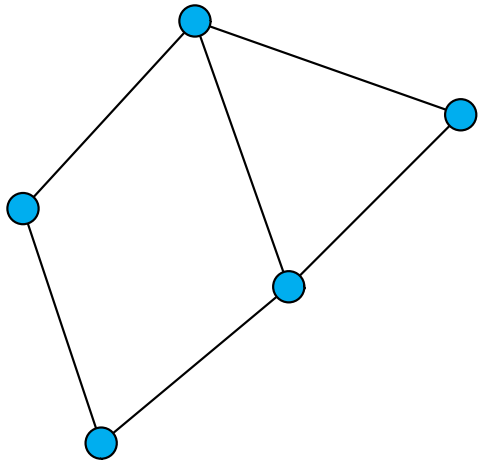
První nápad:



Převod HC na HK

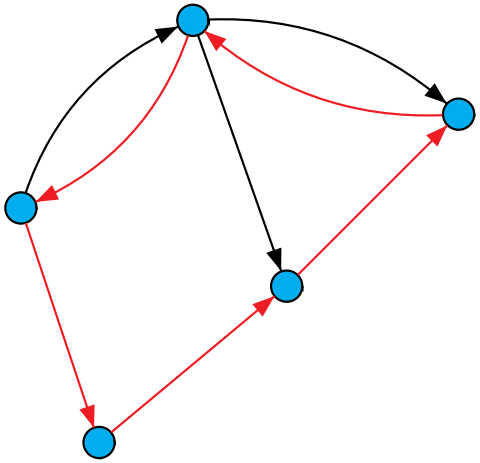
První nápad:

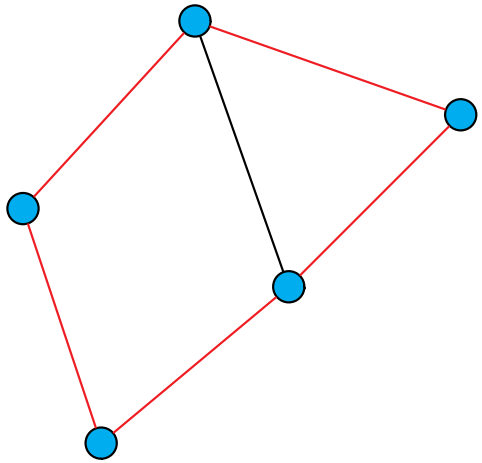




Převod HC na HK

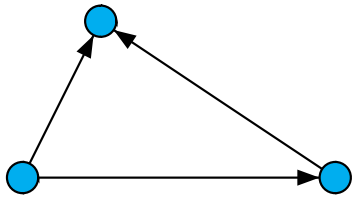
První nápad:





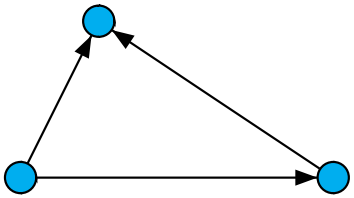
Převod HC na HK

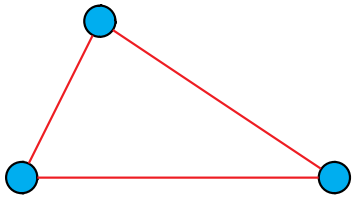
Problém:



Převod HC na HK

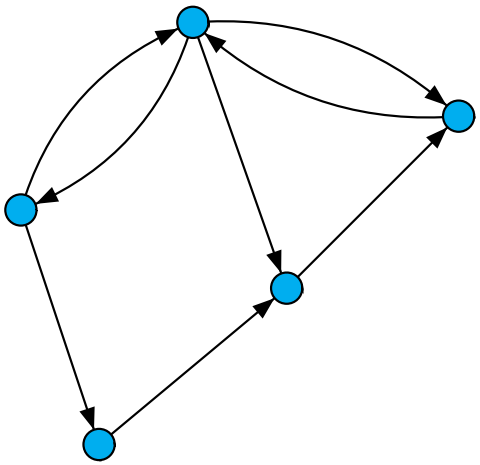
Problém:





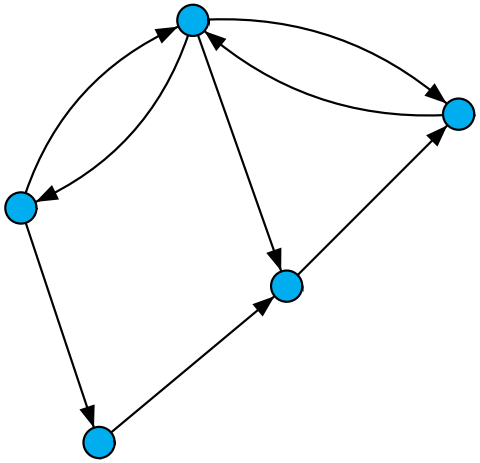
Převod HC na HK

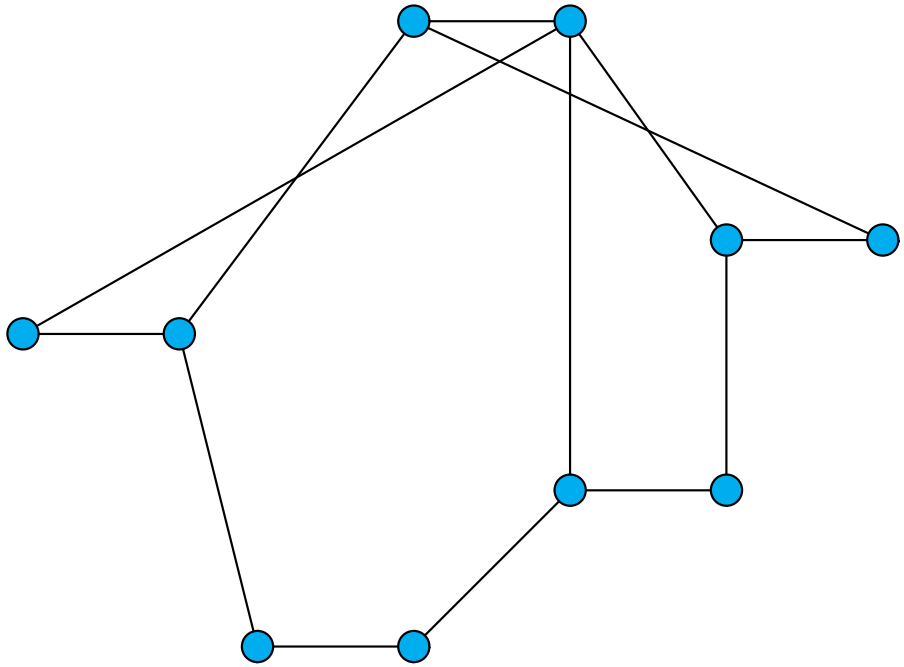
Druhý nápad:



Převod HC na HK

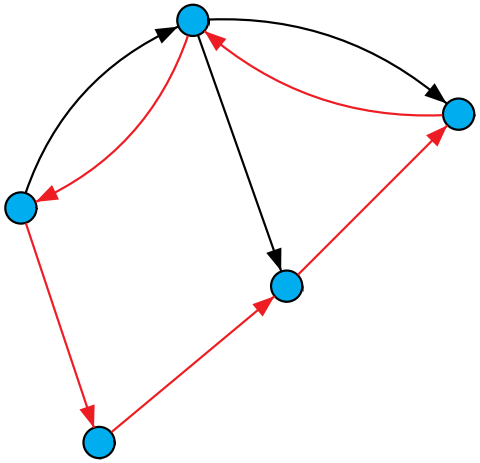
Druhý nápad:

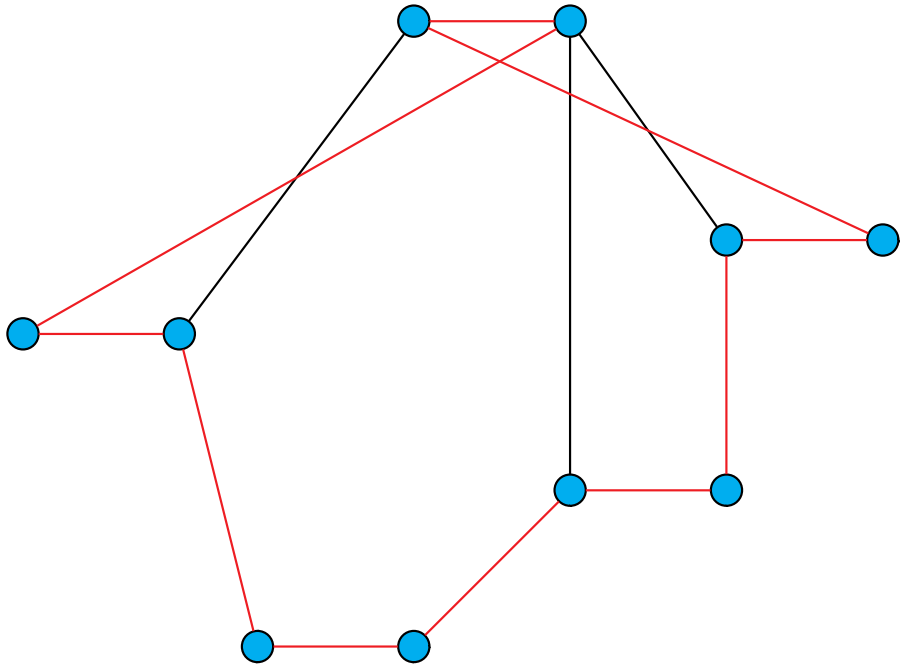




Převod HC na HK

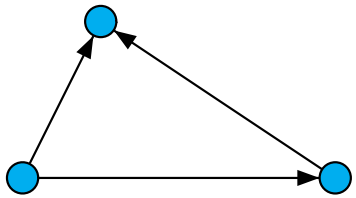
Druhý nápad:





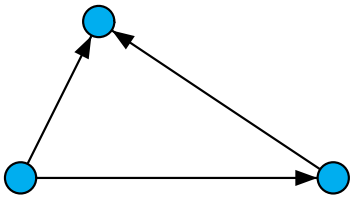
Převod HC na HK

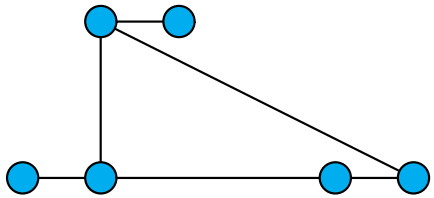
Již správný převod:



Převod HC na HK

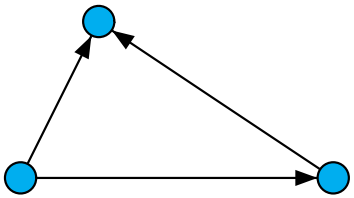
Již správný převod:

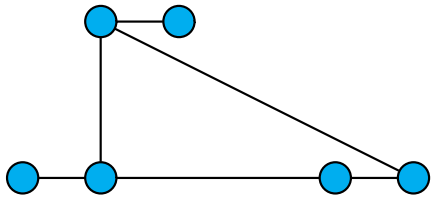




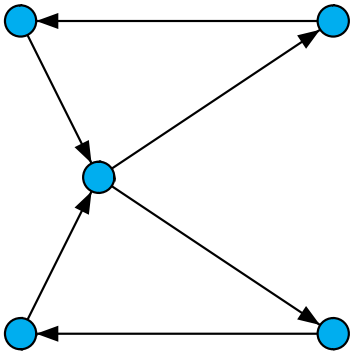
Převod HC na HK

Již správný převod:



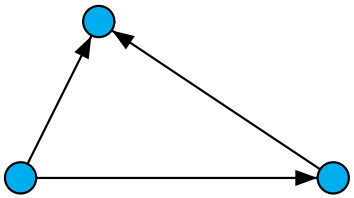


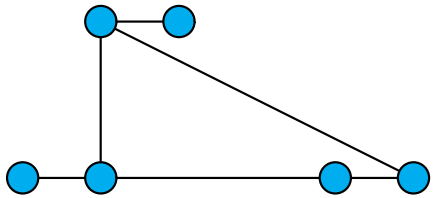
Problém:



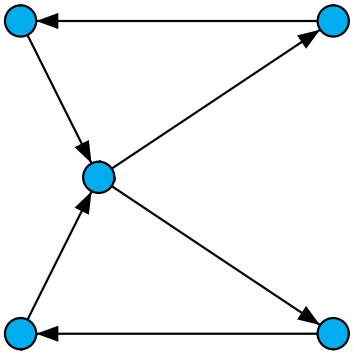
Převod HC na HK

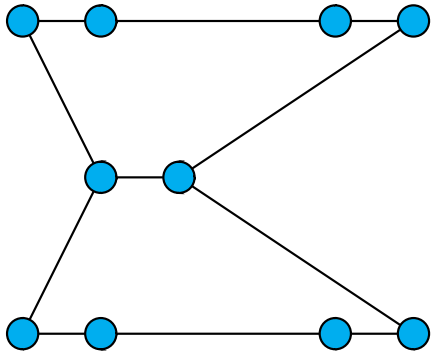
Již správný převod:





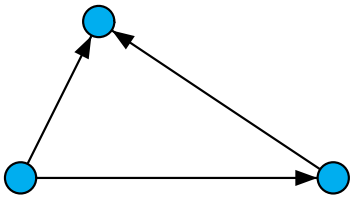
Problém:

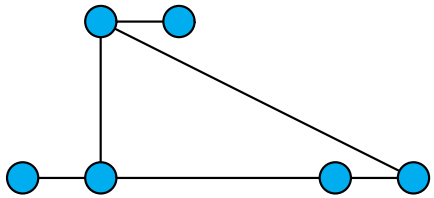




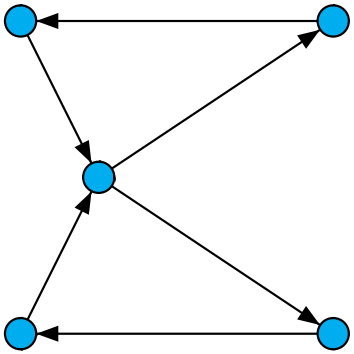
Převod HC na HK

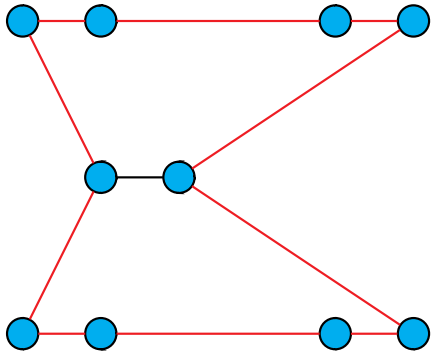
Již správný převod:



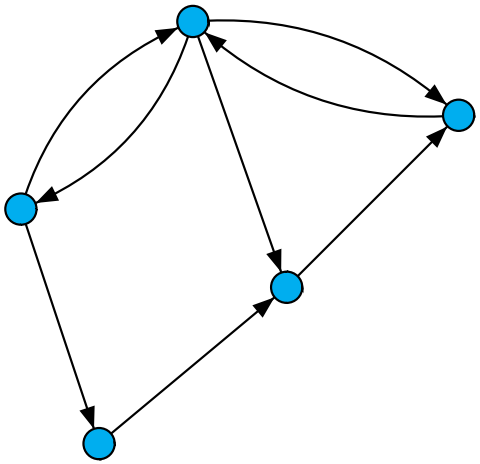


Problém:

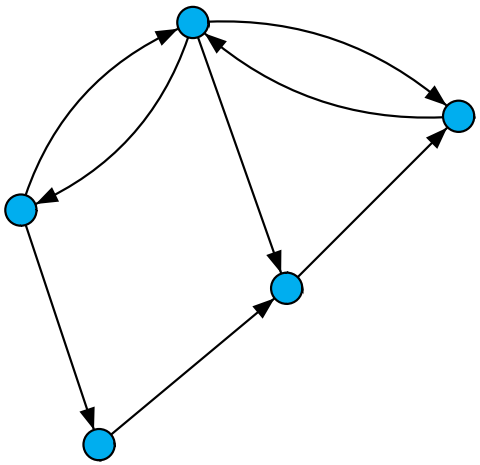


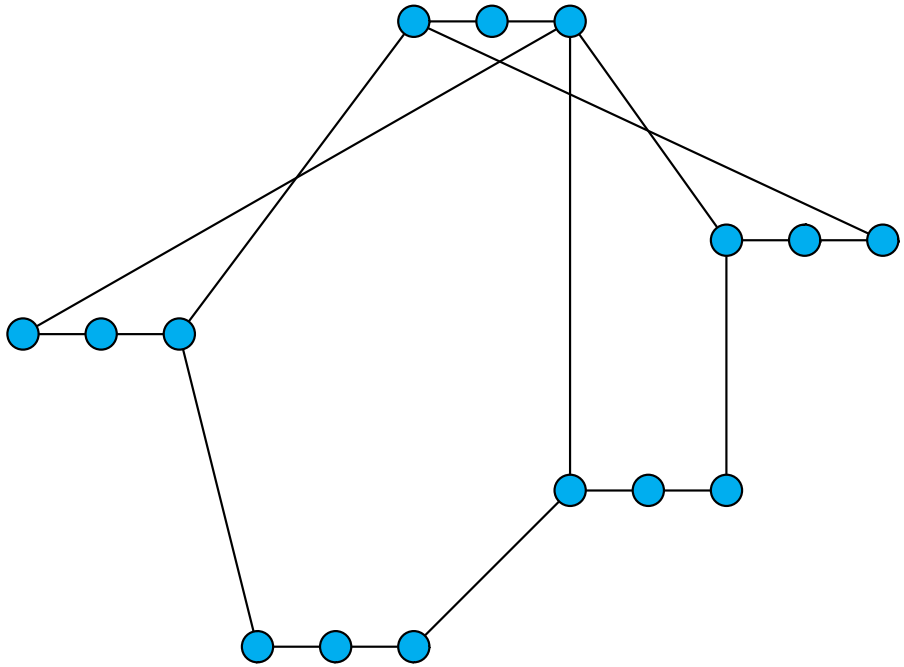


Převod HC na HK

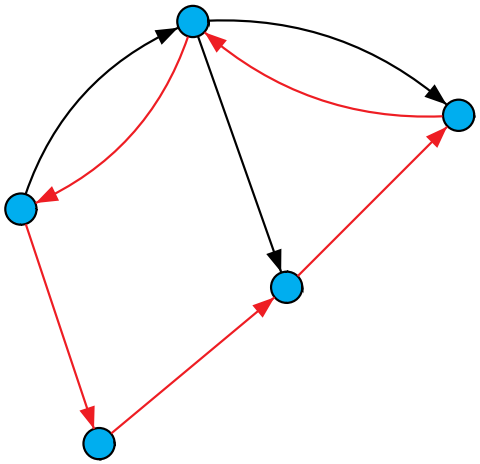


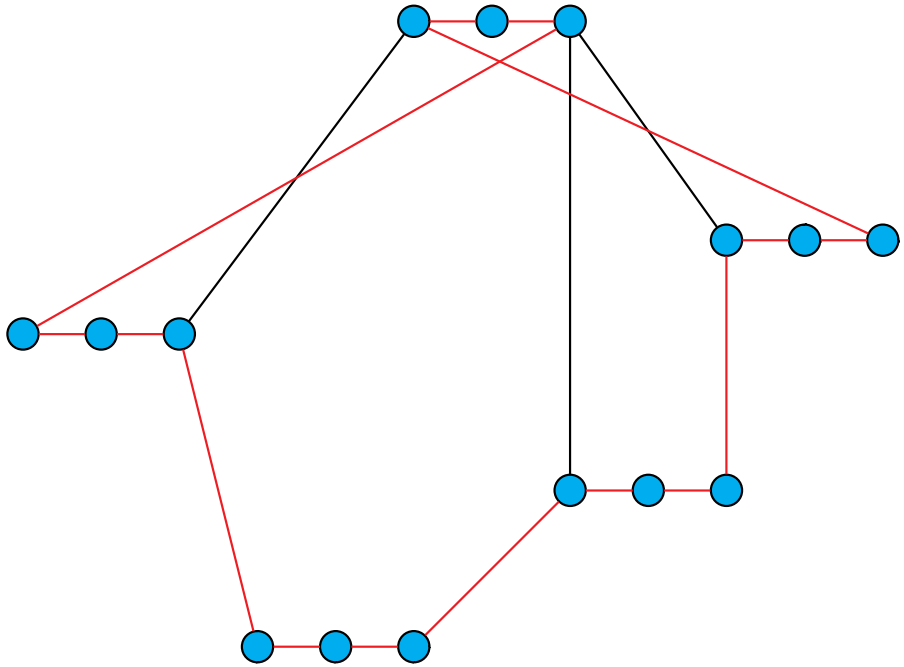
Převod HC na HK





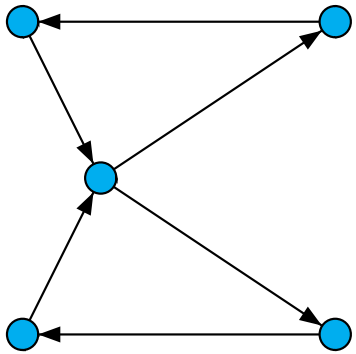
Převod HC na HK





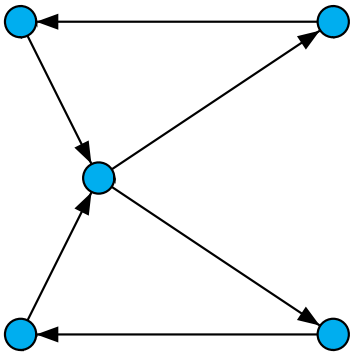
Převod HC na HK

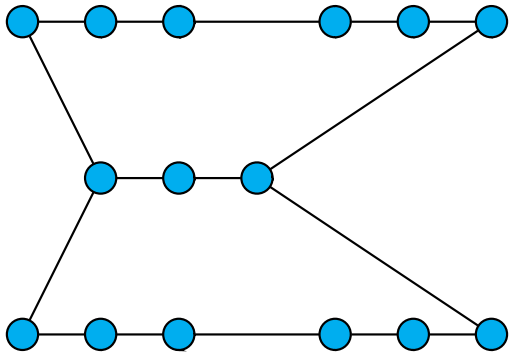
Již správný převod:



Převod HC na HK

Již správný převod:





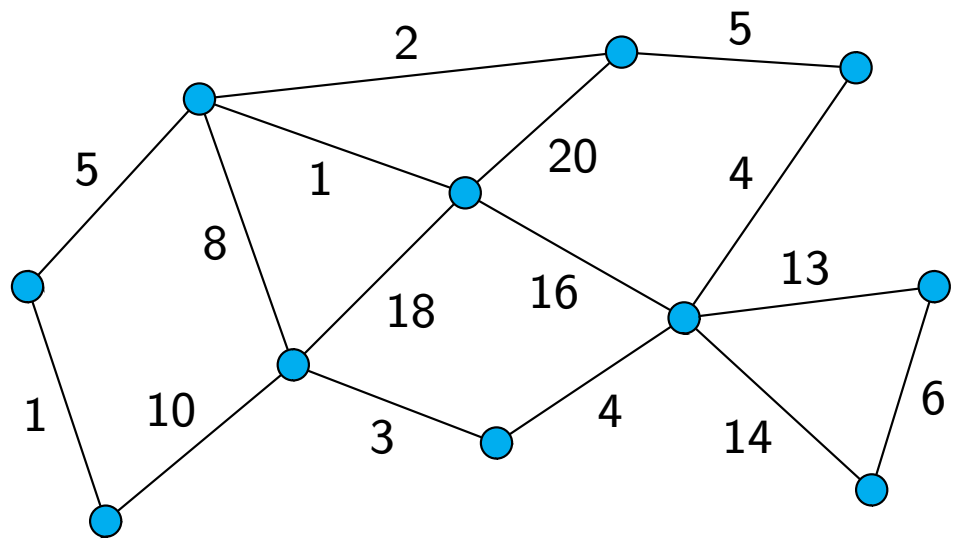
Problém obchodního cestujícího

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: $k = 70$



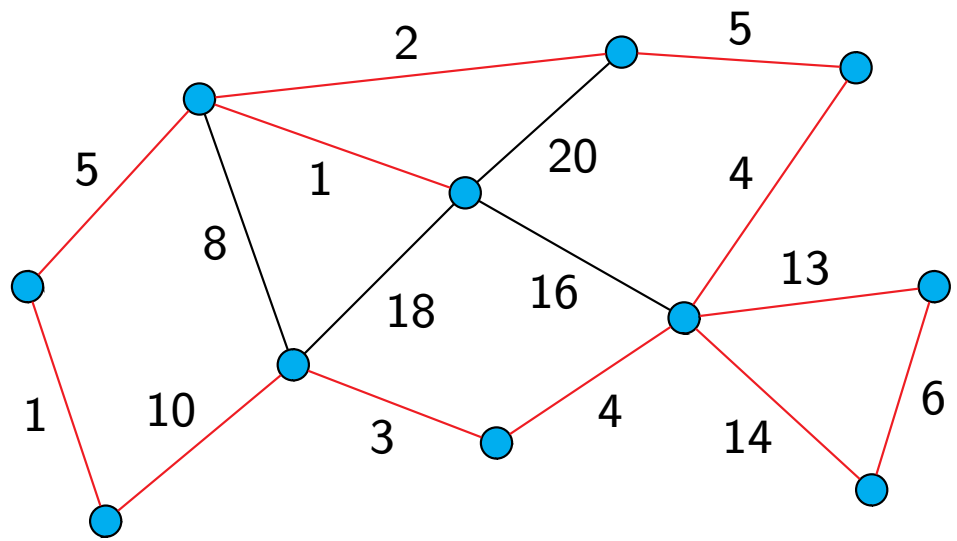
Problém obchodního cestujícího

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 obchodního cestujícího

- 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 $|G|$

Problém loupežníka

PART - Problém „loupežníka“

Vstup: Množina přirozených čísel $A = \{a_1, a_2, \ldots, a_n\}$

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
	111	011	011

$\neg x_1$	100	000	001
$\neg x_2$	010	001	000
x_3	001	001	001
F	000	001	001
	111	011	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 opověď je Ano.

Převod PART na ILP

- Uvažujme $A = \{a_1, a_2, \ldots, a_n\}$ množinu přirozených čísel, která je vstupem problému loupežníka
- Nechť $a = \sum_{a_i \in A} a_i$.
- Sestavíme sestavu dvou nerovnic:

$$a_1 z_1 + a_2 z_2 + \ldots + a_n z_n \leq a/2$$

$$a_1(1 - z_1) + a_2(1 - z_2) + \ldots + a_n(1 - z_n) \leq a/2$$

- 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

Další partie teorie složitosti

Řešení těžkých problémů

- 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.

Řešení těžkých problémů

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.

Řešení těžkých problémů

- **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 $a_1, a_2, \ldots, a_m$ a číslo s.

Otázka: Existuje podmnožina množiny čísel $a_1, a_2, \ldots, a_m$ taková, že součet čísel v této podmnožině je s?

Poznámka: Jako velikost instance bereme celkový počet bitů v zápisu čísel $a_1, a_2, \ldots, a_m$ 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. $m = 10000$).

Řešení těžkých problémů

Problém „HORN-SAT“

Vstup: Booleovská formule φ v KNF obsahující pouze Hornovy klauzule.

Otázka: Je φ splnitelná?

Problém HORN-SAT se dá (narozdíl od obecného problému SAT) řešit 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

$$(x_1 \wedge x_2 \wedge x_3 \wedge x_4) \Rightarrow x_5$$

Řešení těžkých problémů

Následující problém je rovněž možné řešit v polynomiálním čase:

Problém „2-SAT“

Vstup: Booleovská formule φ v KNF, kde každá klauzule obsahuje nejvýše 2 literály.

Otázka: Je φ splnitelná?

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 ε.

- **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).

Testování prvočíselnosti

Prvočíselnost

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 prvočíselnosti

To, zda je možné testovat prvočíselnost 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.

Testování prvočíselnosti

S problémem testování prvočíselnosti úzce souvisí následující problém.

Faktorizace

Vstup: Přirozené číslo p.

Výstup: Rozklad čísla p na prvočísla.

Na rozdíl od prvočíselnosti 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 $1, 2, \ldots, n$

Je dokázáno, že

$$\lim_{n \to \infty} \frac{\pi(n)}{n / \ln n} = 1$$

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.

Randomizované algoritmy

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* $\varepsilon > 0$ *musíme být schopni zaručit, že pravděpodobnost chyby nebude větší než* ε.

Randomizované algoritmy

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.

Randomizované algoritmy

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 $100\,\mathrm{m}^2$ zemského povrchu.

Randomizované algoritmy

Randomizované algoritmy jsou typicky založeny na hledání **svědků** (witness), kteří „dosvědčí“ 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 $\{2, 3, \ldots, n-1\}$:

- Pokud n je prvočíslo, žádní svědci složenosti neexistují.
- Pokud n není prvočíslo, je zaručeno, že alespoň polovina čísel v množině $\{2, 3, \ldots, n-1\}$ jsou svědci složenosti n.

Testování prvočíselnosti

Malá Fermatova věta:

Pokud n je prvočíslo, pak pro každé a z množiny $\{1, 2, \ldots, n-1\}$ platí

$$a^{n-1} \equiv 1 \pmod{n}$$

Malou Fermatovu větu lze použít k testování prvočíselnosti (tzv. Fermatův test):

- Pro dané n zvolíme nějaké $a \in \{2, 3, \ldots, n-1\}$.
- 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.

Testování prvočíselnosti

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 $n \to \infty$ 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 $a \in \{1, 2, \ldots, n-1\}$ 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

Umocňování v modulární aritmetice

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 ← 1
2  ⟨b_k, b_{k−1}, ..., b_0⟩ je binární reprezentace b
3  for i ← k downto 0
4        do d ← (d · d) mod n
5           if b_i = 1
6              then d ← (d · a) mod 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 ← 1 to s
2        do a ← 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 ← 1
 2  ⟨b_k, b_{k−1}, ..., b_0⟩ je binární reprezentace n − 1
 3  for i ← k downto 0
 4        do x ← d
 5           d ← (d · d) mod n
 6           if d = 1 and x ≠ 1 and x ≠ n − 1
 7              then return TRUE
 8           if b_i = 1
 9              then d ← (d · a) mod n
10  if d ≠ 1
11     then return TRUE
12  return FALSE
```

Randomizované algoritmy

- 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 nějací skuteční svědci existují, pak jich je dostatečně mnoho, čímž je zaručeno, že pravděpodobnost, že na nějakého svědka narazíme, je dostatečně vysoká.

Aproximační algoritmy

Mnoho důležitých problémů je NP-úplných. U optimalizačních problémů často nepotřebujeme nalézt nejoptimálnější 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 definovánu množinu S_w všech **přípustných řešení** a funkci $c : S_w \to \mathbb{R}^+$.

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.

Aproximační algoritmy

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 cílem hodnotu $c(x)$ maximalizovat, bychom uvažovali poměr $c(x^*)/c(x)$.

Aproximační algoritmy

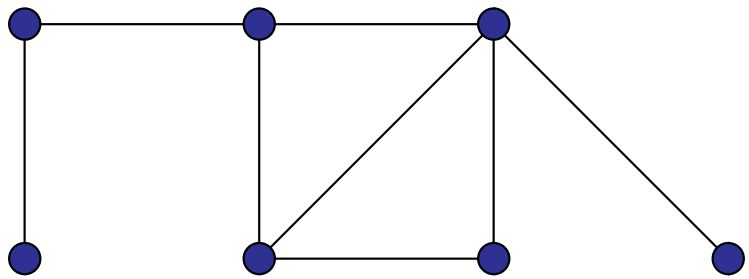
O algoritmu, který dosahuje aproximačního poměru $\rho(n)$, říkáme, že je to $\rho(n)$-**aproximační algoritmus**.

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

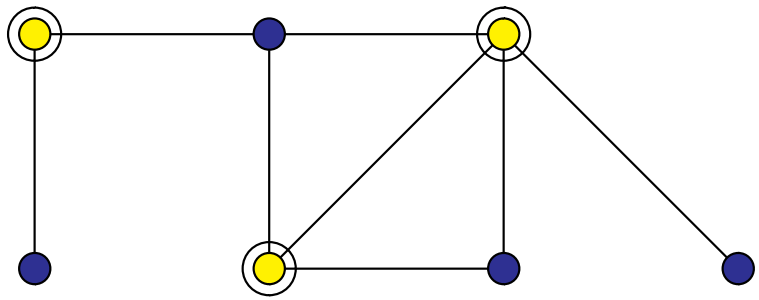
Mějme neorientovaný graf $G = (V, E)$. **Vrcholové pokrytí** (vertex-cover) grafu G je množina $C \subseteq V$ taková, že pro každou hranu $(u, v) \in E$ platí, že buď $u \in C$ nebo $v \in C$.



Úkolem je najít pro zadaný graf minimální vrcholové pokrytí.

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.

Vrcholové pokrytí grafu

Máme tedy vyřešit následující problém:

Minimální vrcholové pokrytí grafu (vertex cover)

Vstup: Neorientovaný graf $G = (V, E)$.

Výstup: Minimalní množina C ($C \subseteq V$) tvořící vrcholové pokrytí grafu G.

Vrcholové pokrytí grafu

Máme tedy vyřešit následující problém:

Minimální vrcholové pokrytí grafu (vertex cover)

Vstup: Neorientovaný graf $G = (V, E)$.

Výstup: Minimalní množina C ($C \subseteq V$) tvořící vrcholové pokrytí grafu G.

Následující (rozhodovací) varianta tohoto problému je NP-úplná:

Vrcholové pokrytí (vertex cover)

Vstup: Neorientovaný graf $G = (V, E)$ a přirozené číslo k.

Otázka: Existuje vrcholové pokrytí grafu G tvořené k vrcholy?

Tento problém tedy není možné řešit v polynomiálním čase (leda by platilo PTIME = NPTIME).

Vrcholové pokrytí grafu

Máme tedy vyřešit následující problém:

Minimální vrcholové pokrytí grafu (vertex cover)

Vstup: Neorientovaný graf $G = (V, E)$.

Výstup: Minimalní množina C ($C \subseteq V$) tvořící vrcholové pokrytí grafu G.

Existuje však 2-aproximační algoritmus řešící tento problém:

- Algoritmus najde pro zadaný graf G vrcholové pokrytí C takové, že

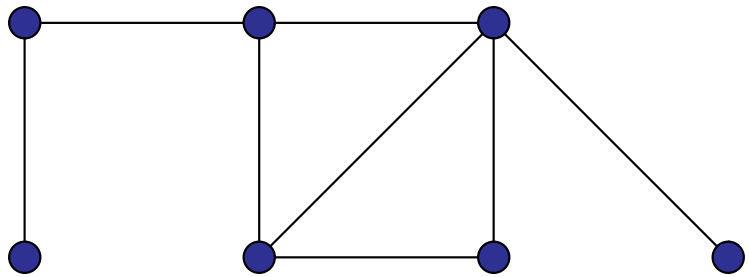
$$|C| \leq 2k$$

kde k je velikost minimálního vrcholového pokrytí grafu G.

Vrcholové pokrytí grafu

APPROX-VERTEX-COVER(G)

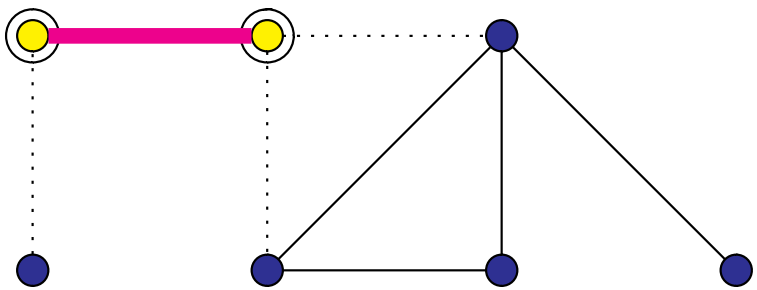
1 $C \leftarrow \emptyset$
2 $E' \leftarrow E$
3 **while** $E' \neq \emptyset$
4 **do** vyber libovolnou hranu (u, v) z E'
5 $C \leftarrow C \cup \{u, v\}$
6 odstraň z E' hrany incidentní s u nebo v
7 **return** C



Vrcholové pokrytí grafu

APPROX-VERTEX-COVER(G)

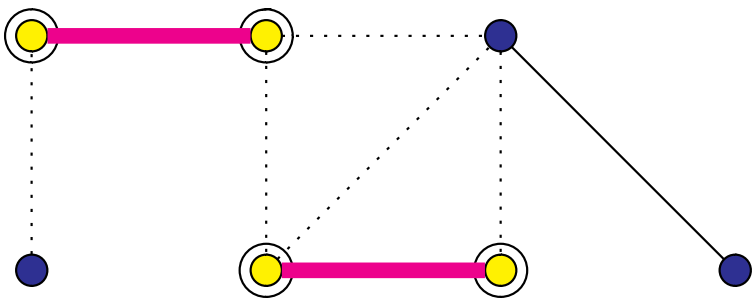
1 $C \leftarrow \emptyset$
2 $E' \leftarrow E$
3 **while** $E' \neq \emptyset$
4 **do** vyber libovolnou hranu (u, v) z E'
5 $C \leftarrow C \cup \{u, v\}$
6 odstraň z E' hrany incidentní s u nebo v
7 **return** C



Vrcholové pokrytí grafu

APPROX-VERTEX-COVER(G)

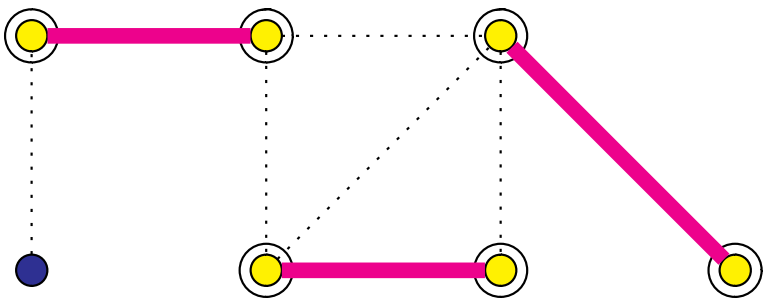
1 $C \leftarrow \emptyset$
2 $E' \leftarrow E$
3 **while** $E' \neq \emptyset$
4 **do** vyber libovolnou hranu (u, v) z E'
5 $C \leftarrow C \cup \{u, v\}$
6 odstraň z E' hrany incidentní s u nebo v
7 **return** C



Vrcholové pokrytí grafu

APPROX-VERTEX-COVER(G)

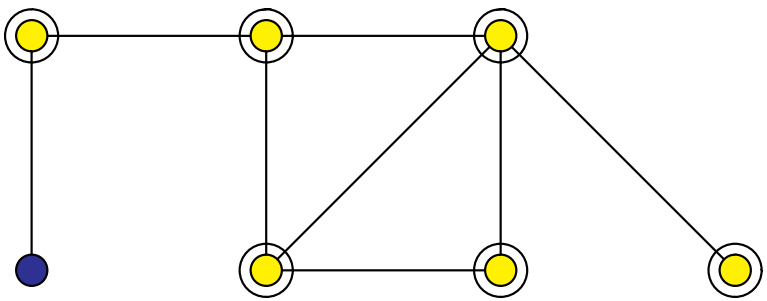
1 $C \leftarrow \emptyset$
2 $E' \leftarrow E$
3 **while** $E' \neq \emptyset$
4 **do** vyber libovolnou hranu (u, v) z E'
5 $C \leftarrow C \cup \{u, v\}$
6 odstraň z E' hrany incidentní s u nebo v
7 **return** C



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



Vrcholové pokrytí grafu

Tvrzení

APPROX-VERTEX-COVER je polynomiální 2-aproximační algoritmus.

Důkaz: Označme A množinu všech hran vybraných v kroku 4, C vrcholové pokrytí nalezené algoritmem a C^* minimální vrcholové pokrytí grafu G.

Jakékoliv vrcholové pokrytí musí obsahovat alespoň jeden z koncových vrcholů každé hrany z A.

Pro libovolné vrcholové pokrytí C' tedy platí $|A| \leq |C'|$.

Speciálně pro C^* tedy také musí platit $|A| \leq |C^*|$.

Na druhou stranu, zjevně platí $|C| = 2 \cdot |A|$.
Dohromady tedy dostáváme $|C| \leq 2 \cdot |C^*|$.

Problém obchodního cestujícího (TSP)

Problém obchodního cestujícího (TSP)

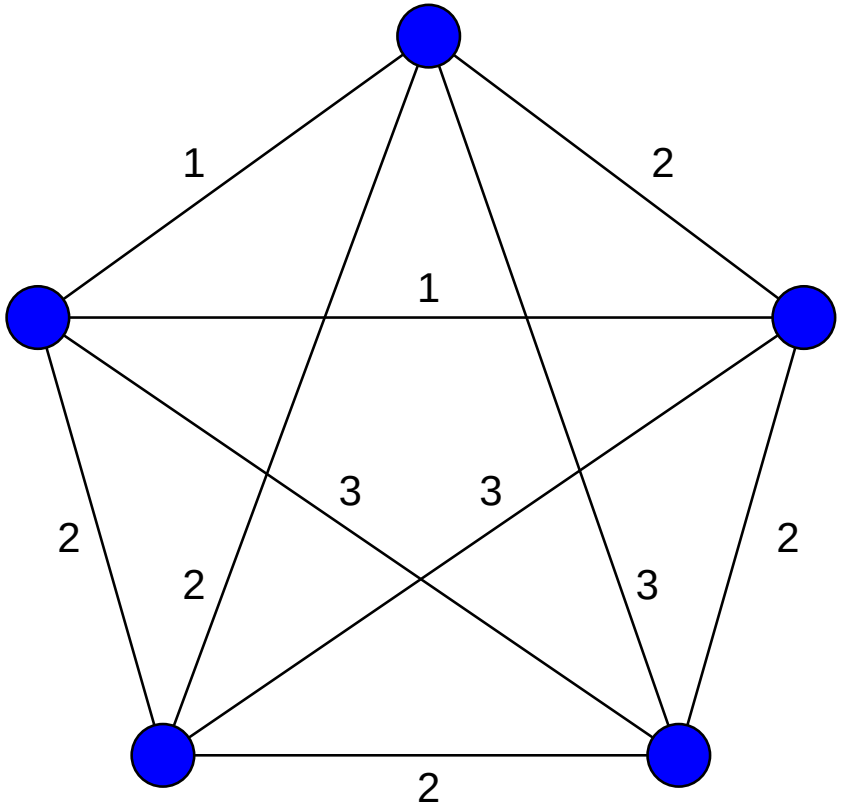
Vstup: Množina n měst a vzdálenosti mezi nimi.

Výstup: Nejkratší okružní cesta procházející všemi městy.

Poznámka: Slovem „okružní“ myslíme, že cesta končí ve stejném městě, kde začíná.

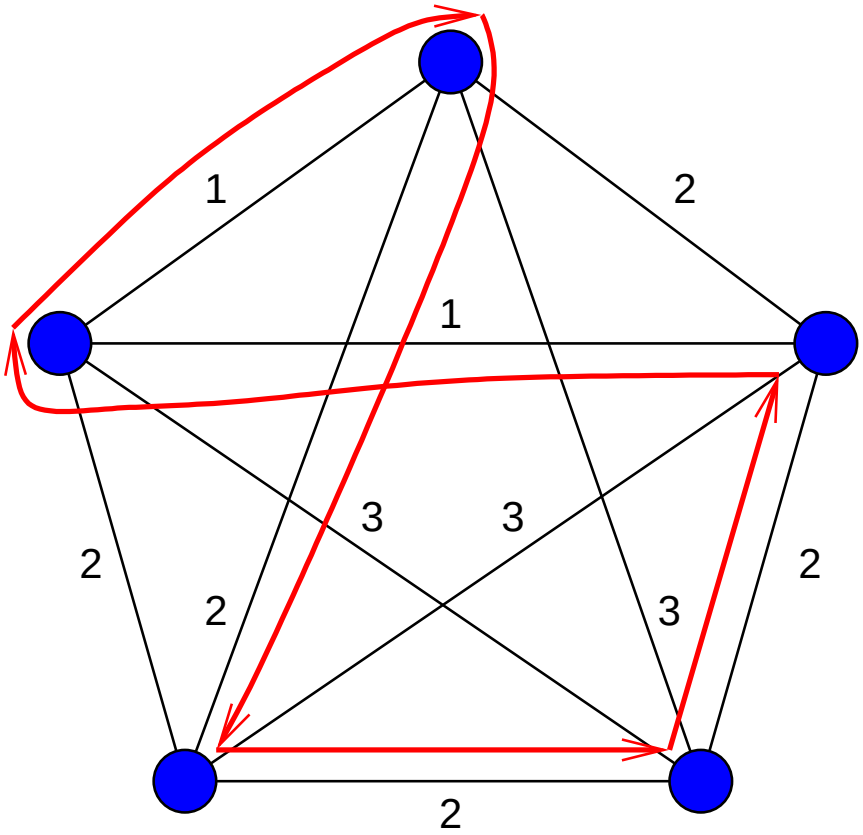
Problém obchodního cestujícího (TSP)

Příklad instance problému:



Problém obchodního cestujícího (TSP)

Příklad instance problému:



Nejkratší okružní cesta má délku 8.

Problém obchodního cestujícího (TSP)

Můžeme uvažovat dvě různé varianty tohoto problému:

- Každé město musí být navštíveno právě jednou.
- Města je možné navštěvovat opakovaně.

Problém obchodního cestujícího (TSP)

V následujícím výkladu se nám bude hodit poněkud formálnější definice problému:

Na instanci problému (tj. množinu měst a vzdálenosti mezi nimi) se můžeme dívat jako na úplný neorientovaný graf $G = (V, E)$ s ohodnocením hran d (kde $d : E \to \mathbb{N}$).

Pro libovolnou množinu hran $E' \subseteq E$ definujeme

$$d(E') = \sum_{e \in E'} d(e)$$

Pro libovolný cyklus H pak definujeme $d(H) = d(E')$, kde E' je množina hran ležících na cyklu H.

Problém obchodního cestujícího (TSP)

Problém TSP pak můžeme formulovat následovně:

Problém obchodního cestujícího (TSP)

Vstup: Úplný neorientovaný graf $G = (V, E)$ s ohodnocením hran d.

Výstup: Cyklus H procházející všemi vrcholy grafu G takový, že hodnota $d(H)$ je minimální možná.

Problém obchodního cestujícího (TSP)

Následující problém je NP-úplný (ať už je či není povoleno navštěvovat vrcholy opakovaně):

Problém obchodního cestujícího (TSP) – rozhodovací varianta

Vstup: Úplný neorientovaný graf $G = (V, E)$ s ohodnocením hran d a číslo L.

Otázka: Existuje v grafu G cyklus H procházející všemi vrcholy takový, že $d(H) \leq L$?

Poznámka: Nemůžeme tedy očekávat, že by problém TSP (ať už v té či oné variantě) byl řešitelný v polynomiálním čase, leda že by platilo P = NP.

Problém obchodního cestujícího

Pro problém TSP, kde vyžadujeme aby byl každý vrchol navštíven právě jednou, se dá dokázat následující:

Věta

Pokud $\mathrm{P} \neq \mathrm{NP}$, pak pro žádnou konstantu $\rho \geq 1$ neexistuje polynomiální ρ-aproximační algoritmus řešící problém obchodního cestujícího.

Důkaz: Předpokládejme, že by pro nějaké ρ 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 $\mathrm{P} = \mathrm{NP}$.

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

$$d(u, v) = \begin{cases} 1 & \text{pokud } (u, v) \in E \\ \rho \cdot |V| + 1 & \text{jinak} \end{cases}$$

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, ρ-aproximační algoritmus musí nějakou takovou kružnici vrátit jako výslednou okružní cestu.

Problém obchodního cestujícího (TSP)

Není těžké si rozmyslet, že varianta TSP kde je povoleno opakovaně navštěvovat vrcholy se dá snadno převést na variantu, kde musí být každý vrchol navštíven právě jednou:

- Pro daný graf G s ohodnocením d sestrojíme nové ohodnocení d', kde $d'(u, v)$ je délka nejkratší cesty z u do v

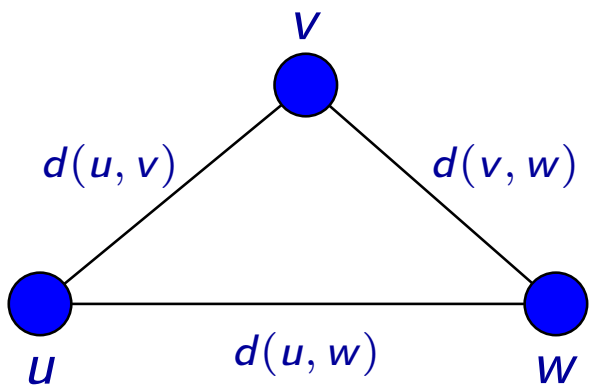
Poznámka: Pro nalezení nejkratších cest mezi dvojicemi vrcholů existují rychlé polynomiální algoritmy (např. Dijkstrův, Floydův-Warshallův apod.)

Graf s ohodnocením d' navíc splňuje tzv. **trojúhelníkovou nerovnost**.

Problém obchodního cestujícího (TSP)

V grafu G s ohodnocením d je splněna **trojúhelníková nerovnost**, jestliže pro libovolnou trojici jeho vrcholů u, v, w platí

$$d(u, w) \leq d(u, v) + d(v, w)$$



Tj. nejkratší cesta z u do w je vždy po hraně (u, w) a nemá cenu jít „oklikou“ přes nějaký jiný vrchol.

Problém obchodního cestujícího (TSP)

Variantu TSP, ve které se omezujeme pouze na instance, ve kterých je splněna trojúhelníková nerovnost (a kde musí být každý vrchol navštíven právě jednou), označujeme **Δ-TSP**.

Problém obchodního cestujícího (TSP)

Variantu TSP, ve které se omezujeme pouze na instance, ve kterých je splněna trojúhelníková nerovnost (a kde musí být každý vrchol navštíven právě jednou), označujeme Δ-**TSP**.

Pro problém Δ-TSP je znám 1.5-aproximační polynomiální algoritmus, tj. algoritmus, který pro daný graf G s ohodnocením d vrátí cyklus H procházející všemi vrcholy takový, že

$$d(H) \leq 1.5 \cdot d(H^*)$$

kde H^* optimální řešení (tj. cyklus s minimální hodnotou $d(H^*)$).

Problém obchodního cestujícího (TSP)

Variantu TSP, ve které se omezujeme pouze na instance, ve kterých je splněna trojúhelníková nerovnost (a kde musí být každý vrchol navštíven právě jednou), označujeme Δ-**TSP**.

Pro problém Δ-TSP je znám 1.5-aproximační polynomiální algoritmus, tj. algoritmus, který pro daný graf G s ohodnocením d vrátí cyklus H procházející všemi vrcholy takový, že

$$d(H) \leq 1.5 \cdot d(H^*)$$

kde H^* optimální řešení (tj. cyklus s minimální hodnotou $d(H^*)$).

My si ukážeme poněkud jednodušší 2-aproximační polynomiální algoritmus pro problém Δ-TSP.

Problém obchodního cestujícího (TSP)

Před vlastním popisem algoritmu si připomeňme některé pojmy:

Kostra grafu $G = (V, E)$ je libovolný souvislý acyklický graf $T = (V', E')$, kde $V = V'$ a $E' \subseteq E$ (tj. T je souvislý podgraf grafu G obsahující všechny vrcholy z G).

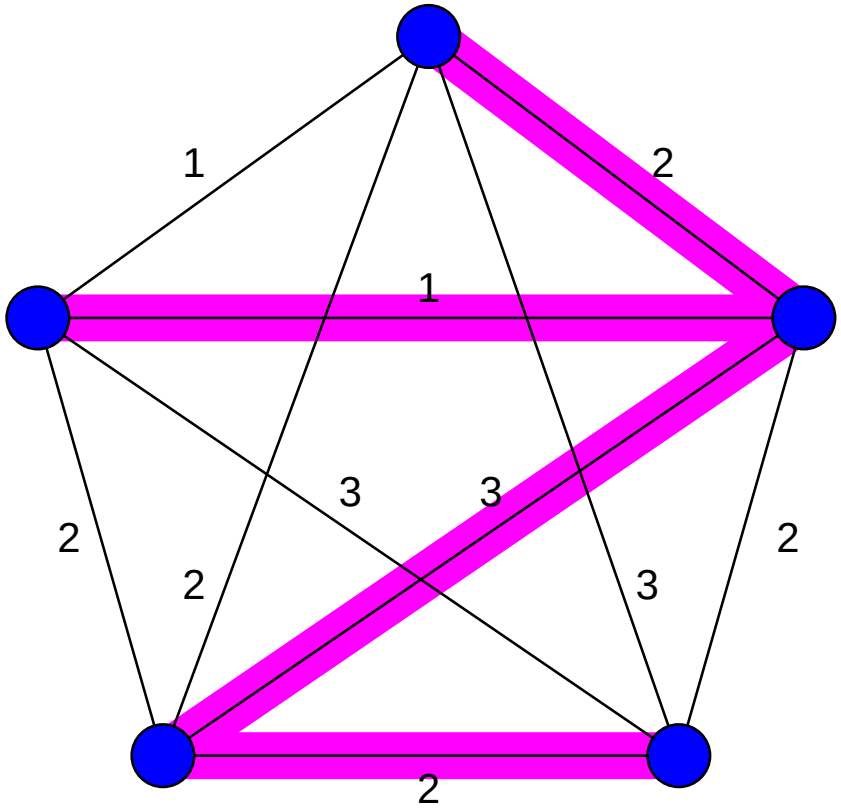
Hodnotu $d(T)$ definujeme jako součet hodnot hran v této kostře, tj. $d(T) = d(E')$.

Kostra T je **minimální**, jestliže pro libovolnou jinou kostru T' v grafu G platí $d(T) \leq d(T')$.

Pro problém nalezení minimální kostry v daném ohodnoceném grafu jsou známy rychlé polynomiální algoritmy (např. Kruskalův nebo Jarníkův (Primův)).

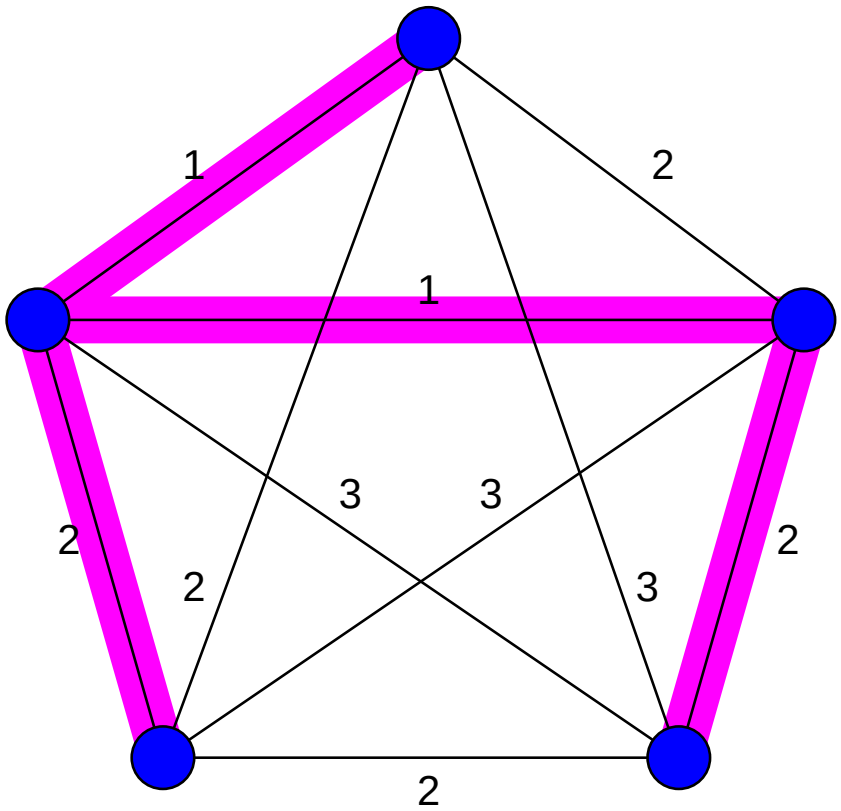
Problém obchodního cestujícího (TSP)

Příklad kostry T, kde $d(T) = 8$:



Problém obchodního cestujícího (TSP)

Příklad minimální kostry T, kde $d(T) = 6$:



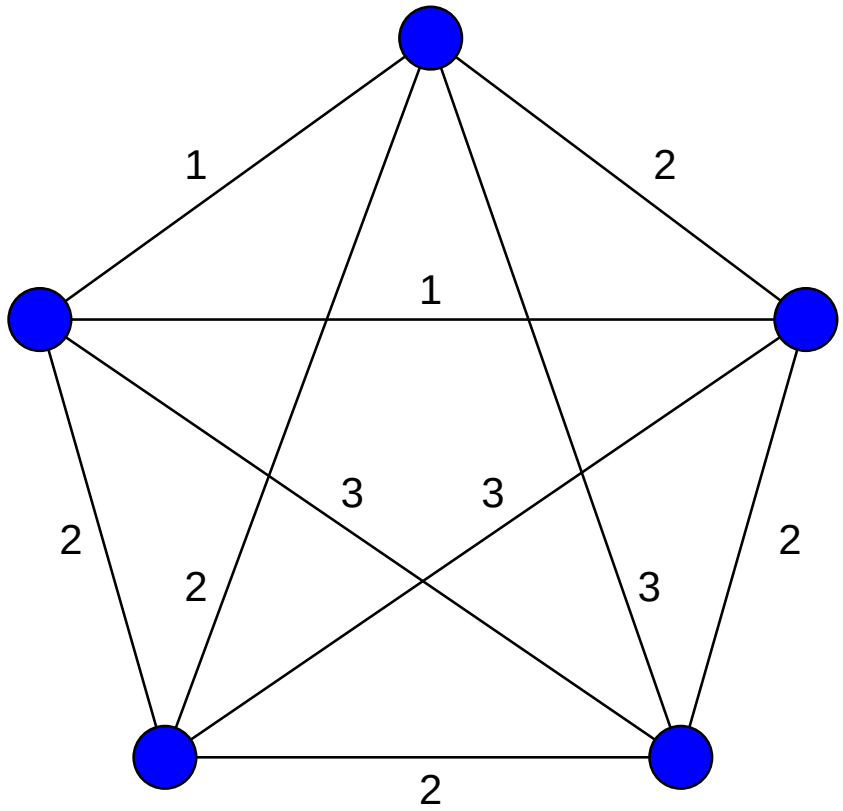
Problém obchodního cestujícího (TSP)

2-aproximační algoritmus pro Δ-TSP pracuje v následujících krocích:

- Najde minimální kostru grafu G.
- Vytvoří uzavřený tah podél této kostry.
- Z vytvořeného tahu odstraní opakující se vrcholy a výsledný cyklus vrátí jako výsledek.

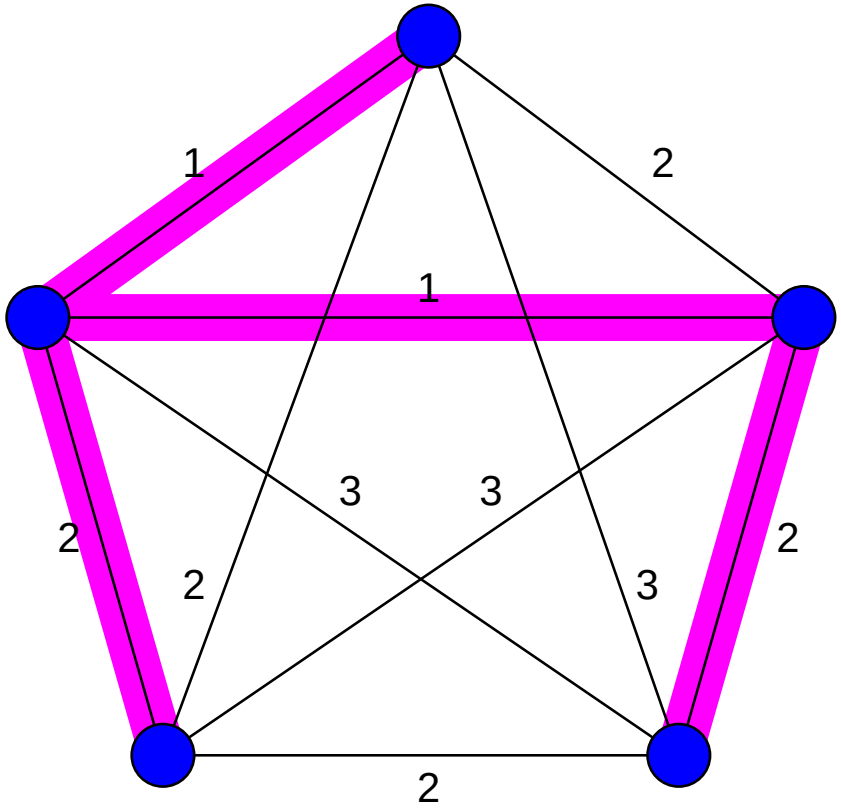
Problém obchodního cestujícího (TSP)

Vezměme si následující instanci Δ-TSP.



Problém obchodního cestujícího (TSP)

Krok 1: Nalezení minimální kostry T



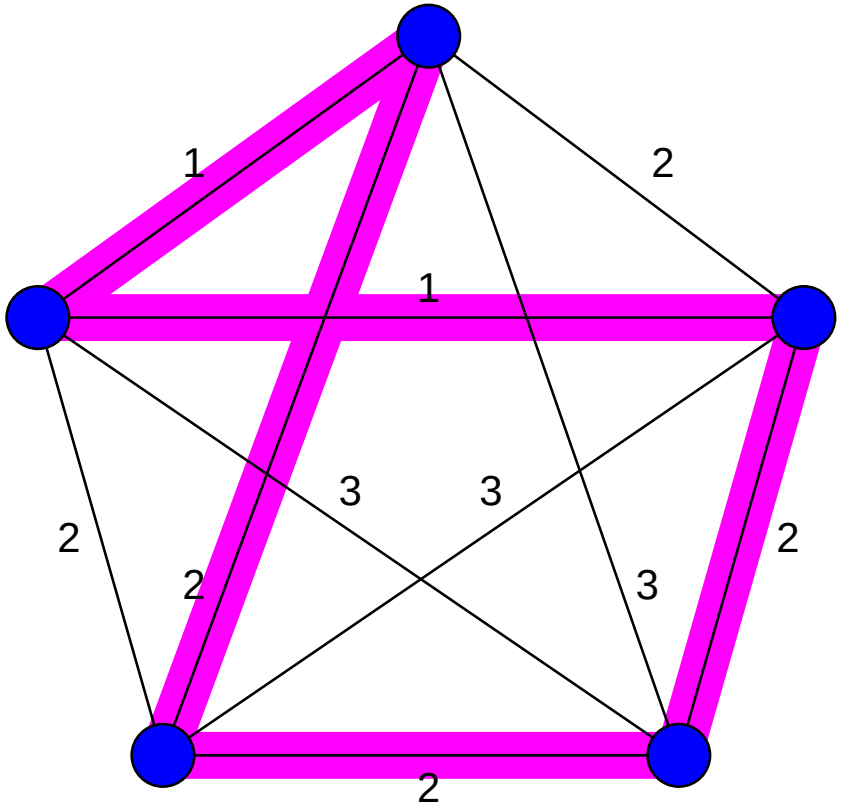
Problém obchodního cestujícího (TSP)

Všimněme si, že $d(T) < d(H^*)$:

- Pokud z H^* odstraníme libovolnou hranu, dostaneme kostru T'. Zjevně platí $d(T') < d(H^*)$.
- Pro libovolnou kostru T' platí $d(T) \leq d(T')$.

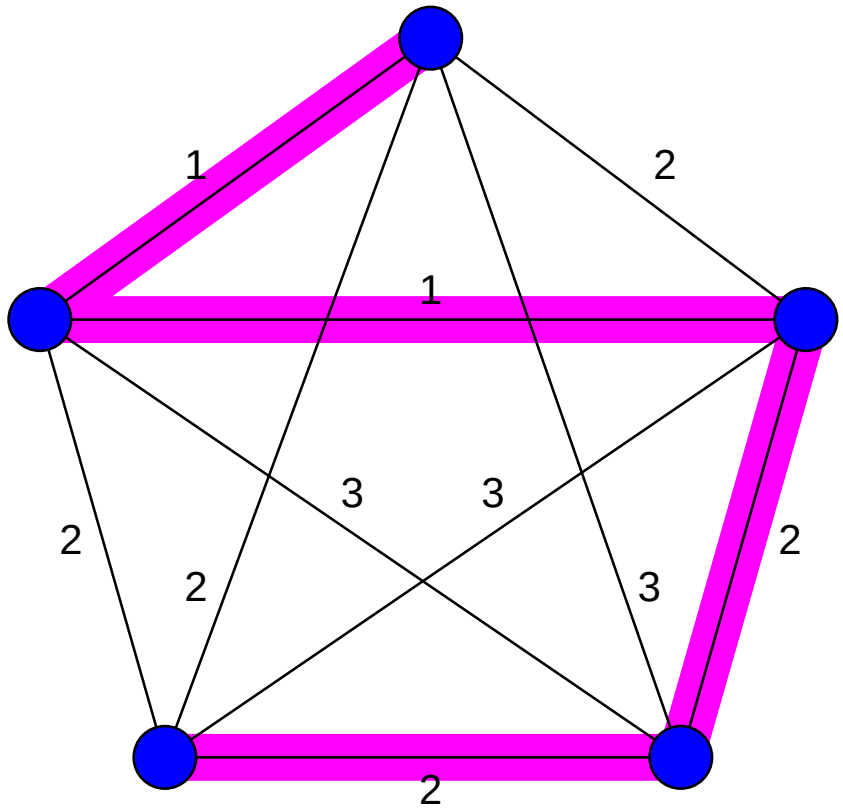
Problém obchodního cestujícího (TSP)

Příklad: Vezměme si optimální cyklus



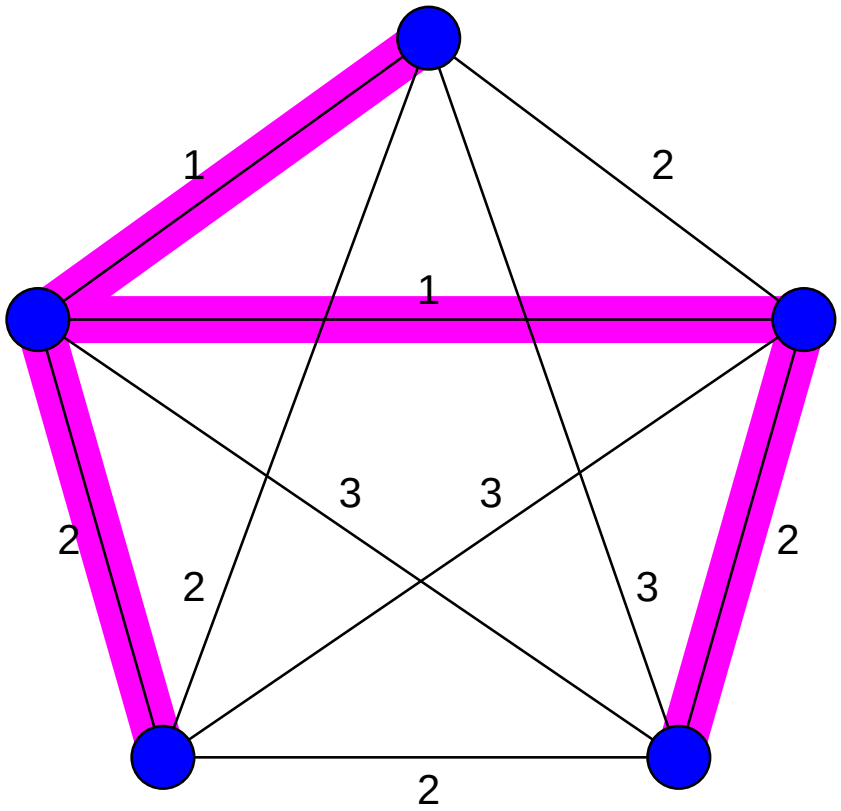
Problém obchodního cestujícího (TSP)

Příklad: Odstraněním jedné hrany vznikne kostra



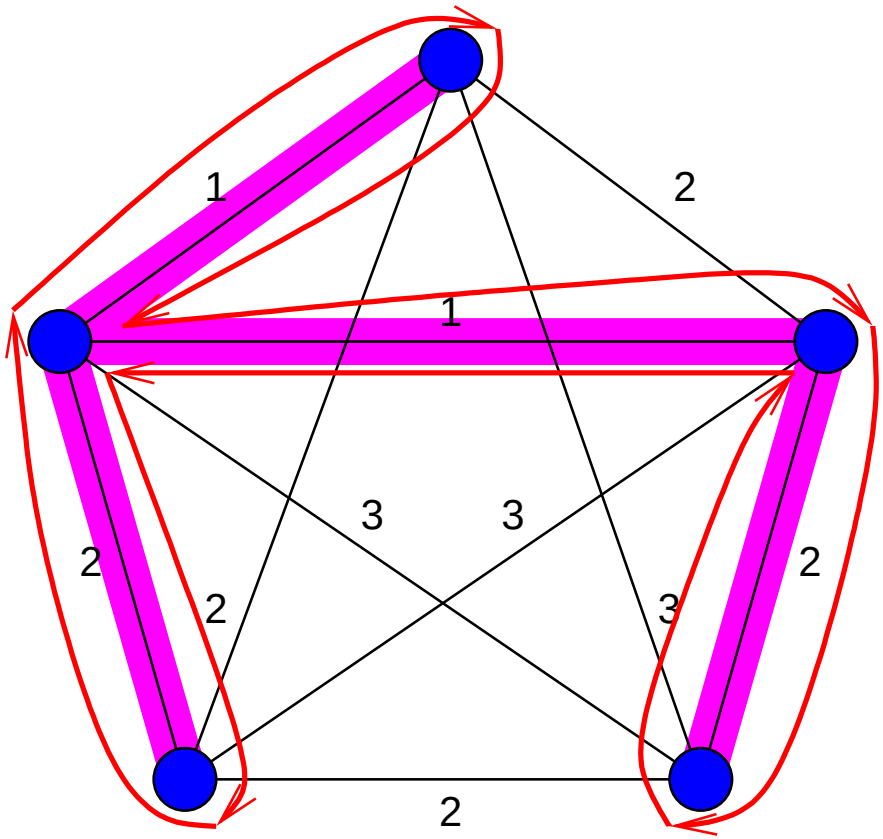
Problém obchodního cestujícího (TSP)

Krok 1: Nalezení minimální kostry



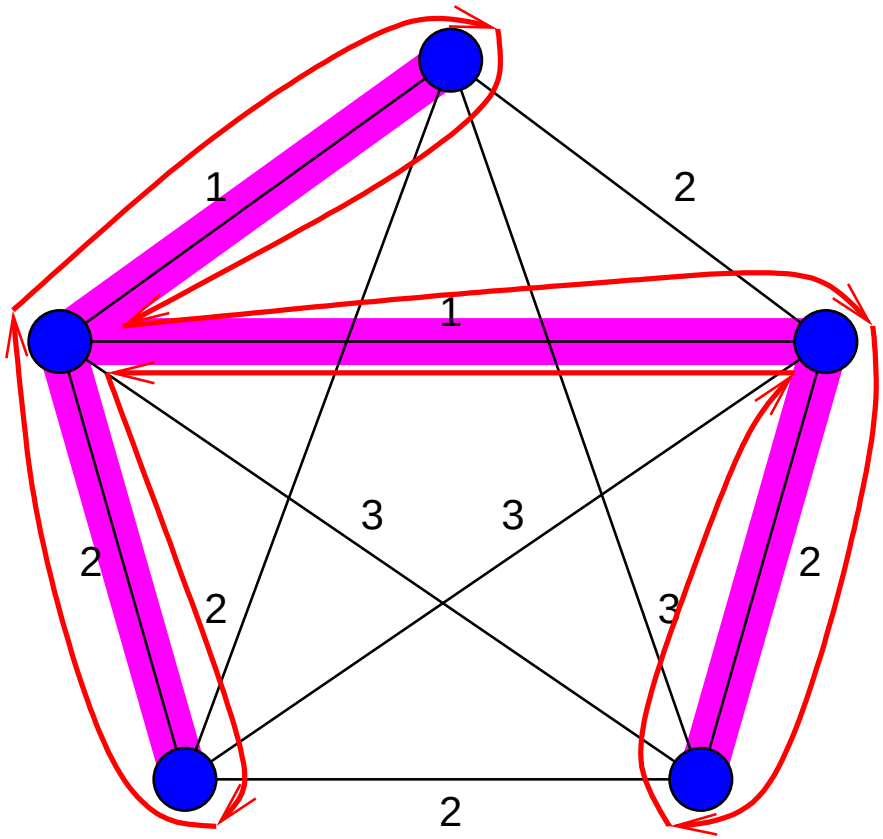
Problém obchodního cestujícího (TSP)

Krok 2: Vytvoření tahu H podél kostry



Problém obchodního cestujícího (TSP)

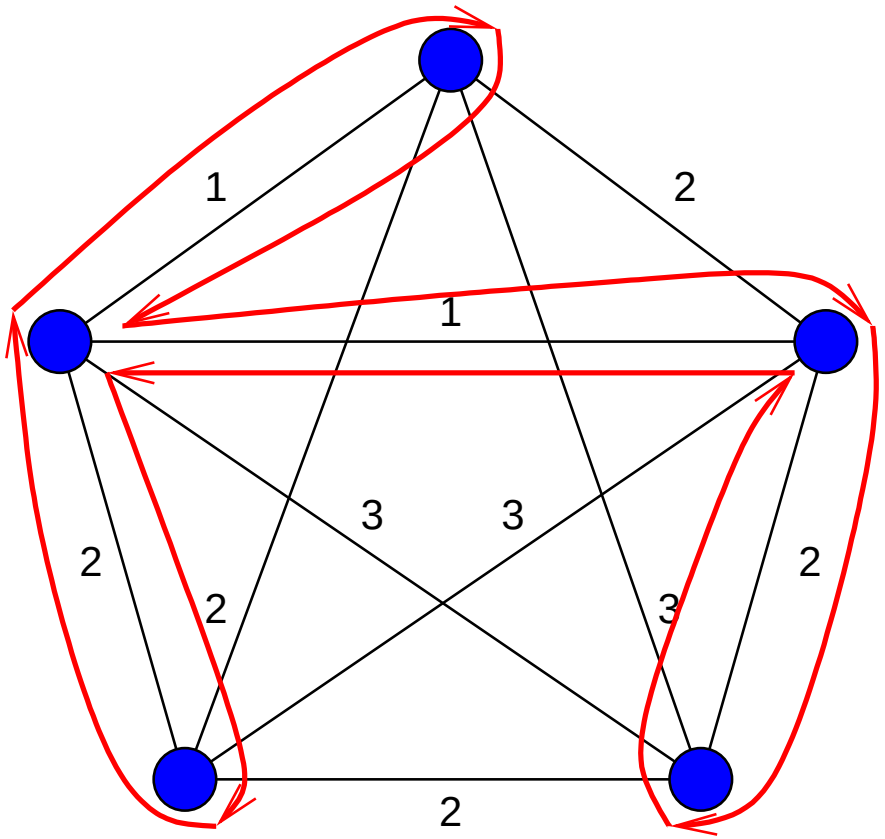
Krok 2: Vytvoření tahu H podél kostry



Každou hranu procházíme dvakrát, platí tedy $d(H) = 2d(T) < 2d(H^*)$.

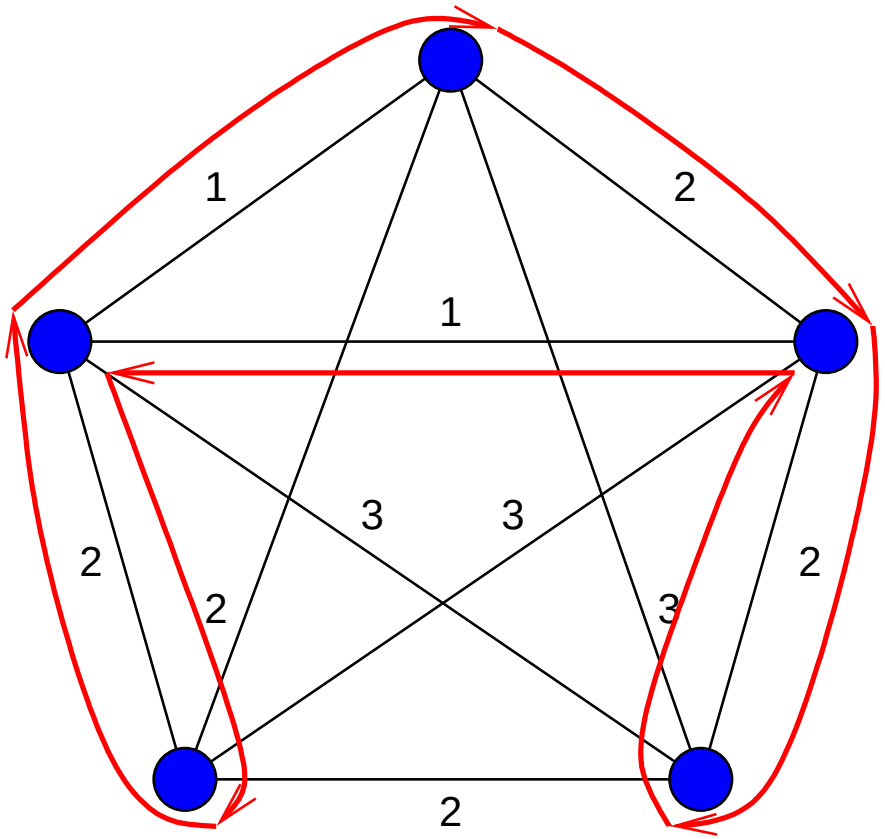
Problém obchodního cestujícího (TSP)

Krok 2: Vytvoření tahu H podél kostry



Problém obchodního cestujícího (TSP)

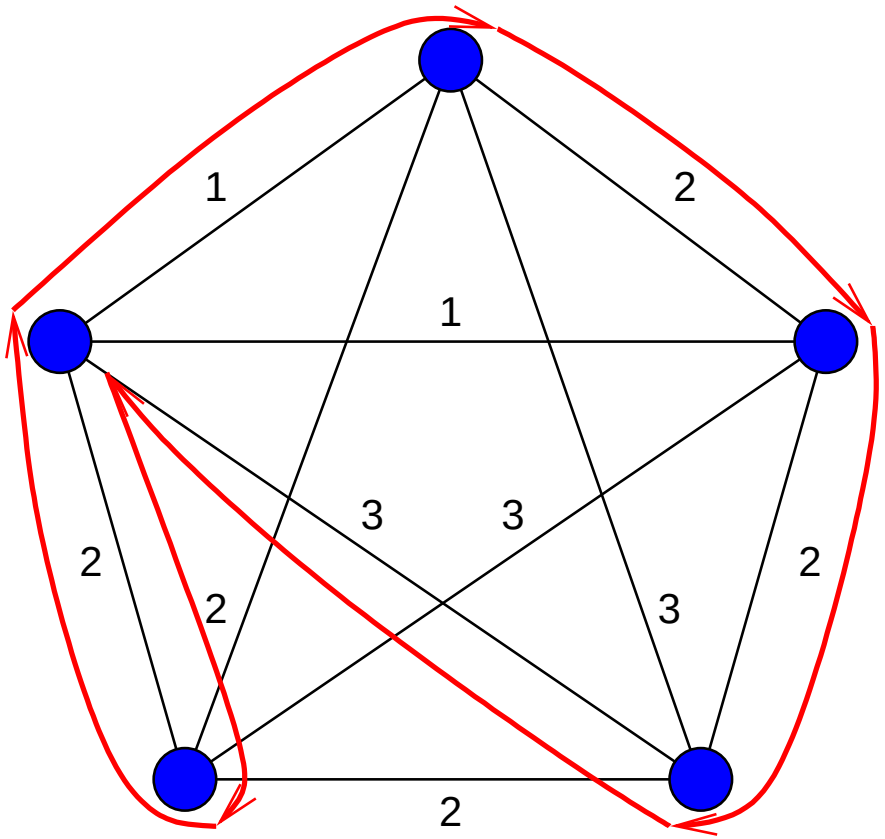
Krok 3: Postupné vypouštění opakujících se vrcholů z tahu H



Každé vypuštění vrcholu tah leda zkrátí.

Problém obchodního cestujícího (TSP)

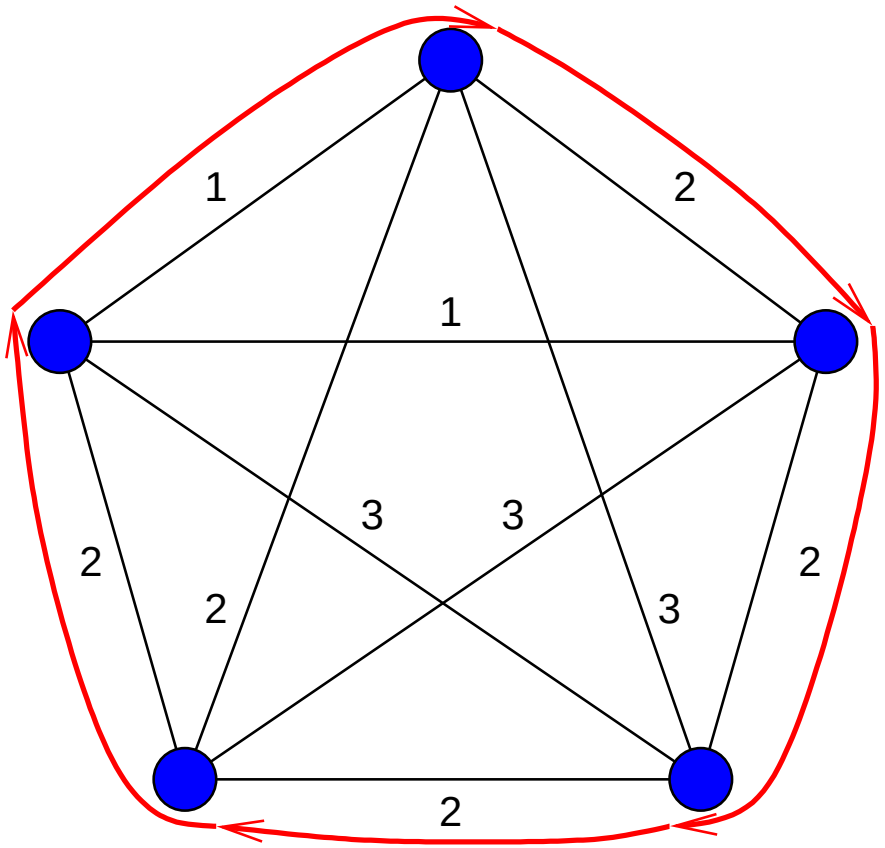
Krok 3: Postupné vypouštění opakujících se vrcholů z tahu H



Každé vypuštění vrcholu tah leda zkrátí.

Problém obchodního cestujícího (TSP)

Krok 3: Postupné vypouštění opakujících se vrcholů z tahu H



Každé vypuštění vrcholu tah leda zkrátí.

Problém obchodního cestujícího (TSP)

Nalezený cyklus: