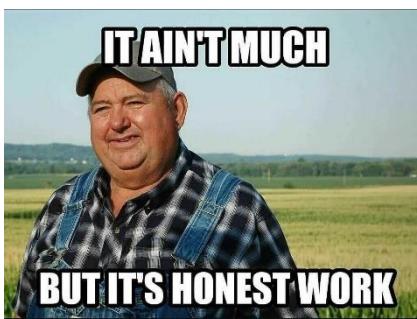


# Otázky ke stání závěrečné zkoušce

„Nebudeš slavnej, protože umíš hovno!“

- David „Cinzano Bianco“ Lister

Na:	Technické univerzitě v Liberci (TUL) Fakultě mechatroniky, informatiky a mezioborových studií (FM)
Pro:	Obor Informační technologie (IT) Specializace Aplikovaná informatika (AI)
Vypracoval:	OM
Léta Páně:	2023/24
Určeno jako:	Čistě studijní a relaxační čtení



# 1. Základy teorie dělitelnosti, základní pojmy (relace býti dělitelem a její vlastnosti, věta o dělení se zbytkem), eukleidův algoritmus, využití. Řetězové zlomky (konstrukce, vlastnosti), řešení kongruencí 1. stupně a jejich soustav.

## Pozn. – Základní pojmy

### Definice - kartézský součin

Nechť  $A, B$  jsou neprázdné množiny. Kartézský součin množin  $A, B$  budeme značit  $A \times B$  a definujeme ho jako množinu všech uspořádaných dvojic  $(a, b)$ , kde  $a \in A, b \in B$ , tj.

$$A \times B = \{(a, b) | a \in A, b \in B\}.$$

### Poznámky

- Je-li  $A \neq B$ , potom  $A \times B \neq B \times A$ , tj. kartézský součin není komutativní.
- Jsou-li  $A, B$  konečné množiny, potom platí  $|A \times B| = |A| \cdot |B|$ .
- V případě, kdy  $A = B$ , používáme místo zápisu  $A \times A$  obvykle zápis  $A^2$  a mluvíme o druhé kartézské mocnině množiny  $A$ .

### Definice – (binární) relace

Nechť je  $A \neq \emptyset$  množina. Binární relací na množině  $A$  rozumíme libovolnou podmnožinu  $A^2$ .

Značení:  $aRb$  „ $a$  je v relaci s  $b$ “

Znsl. vlastnosti relací:

Nejdříve  $R$  je relace na  $A$

i) Rěkeme, že  $R$  je reflexivní relace na  $A$ , jestliže

$$\forall a \in A \quad (a, a) \in R$$

ii) Rěkeme, že  $R$  je symetrická na  $A$ , jestliže

$$\forall a, b \in A \quad (a, b) \in R \rightarrow (b, a) \in R$$

iii) Rěkeme, že  $R$  je antisymetrická na  $A$ , jestliže

$$\forall a, b \in A \quad (a, b) \in R \wedge (b, a) \in R \rightarrow a = b$$

$$(a < b) \wedge (b > a) \rightarrow a = b$$

1	0	1	0
0	0	1	1
0	1		
1	0	0	
1	1	1	

- zpravidla platí jen pravidlo

- z nepřavidla platí celkově

iv)  $R$  je transitivní na  $A$ ,

$$\text{jestliže } \forall a, b, c \in A \quad (a, b) \in R \wedge (b, c) \in R \rightarrow (a, c) \in R$$

Celá čísla:  $\mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$

### Definice – zobrazení

Nechť  $\emptyset \neq f \subseteq A \times B$ . Jestliže pro každé  $a \in A$  existuje nejvýše jedno  $b \in B$  tak, že  $(a, b) \in f$ , potom relaci  $f$  nazýváme zobrazení z množiny  $A$  do množiny  $B$ .

# Základy teorie dělitelnosti

## Relace dělitelnosti

### Definice – dělitelnost

Řekneme, že nenulové celé číslo  $b$  dělí celé číslo  $a$ , píšeme  $b|a$ , jestliže

$$(\exists q \in \mathbb{Z}) (a = b \cdot q).$$

V opačném případě píšeme  $b \nmid a$  a říkáme, že  $b$  nedělí  $a$ .

V uvedeném vztahu je  $a$  ... násobek  $b$ ,

$b$  ... dělitel  $a$ ,

$q$  ... podíl

## Vlastnosti

- Plati: i)  $\forall a \in \mathbb{Z} \quad 1|a$   
ii)  $\forall a \in \mathbb{Z} - \{0\} \quad a|0$   
iii)  $\forall a \in \mathbb{Z} - \{0\} \quad a|a \quad (\text{reflex.})$   
iv)  $\forall a, b, c \in \mathbb{Z} - \{0\}$   
 $(c|b) \wedge (b|a) \rightarrow c|a \quad (\text{transit.})$   
v)  $\forall a, b \in \mathbb{Z} - \{0\}$   
 $(a|b) \wedge (a|c) \rightarrow b = c$   
vi)  $\forall a, b \in \mathbb{N}^+$   
 $(b|a) \wedge (a|b) \rightarrow a = b$

## (Ne)Vlastní dělitel

Popis: (ne)vlastní dělitel:

$$a \in \mathbb{Z} - \{0\}$$

$1|a \wedge a|a \dots$  nejmenší (minimální) dělitlel čísla  $a$

## Věta o dělení se zbytkem

Plati: věta o dělení se zbytkem

Nechť:  $a \in \mathbb{Z}$ ,  $b \in \mathbb{N}^+$  existují jedinečné  $q, r \in \mathbb{Z}$

~~zajímavé~~ fakt, že platí

$$a = b \cdot q + r, \text{ kde } 0 \leq r < b$$

$a$  ... dělence

$b$  ... dělitel

$q$  ... nejvyšší podíl

$r$  ... zbytek

## Eukleidův algoritmus a využití

### Eukleidův algoritmus

$$a \in \mathbb{Z} \quad b \in \mathbb{N}^+$$

$$a = b \cdot q_0 + r_1 \quad 0 < r_1 < b$$

$$b = r_1 q_1 + r_2 \quad 0 < r_2 < r_1$$

$$r_1 = r_2 q_2 + r_3 \quad 0 < r_3 < r_{m-1}$$

$$r_{m-2} = r_{m-1} q_{m-1} + r_m \quad 0 < r_m < r_{m-1}$$

$$r_{m-1} = r_m q_m$$

Eukleidův algor je končící! - nezáleží po do řešení  
dostání

platí: Lze

Euk. algor řešení největšího je řešení všechny ostatních algor  
menších a řešel  $a, b$ !

## Využití – pro hledání NSD

### Systém dělitelů , NSD(a, b)

Def: Řetězec, ře řadné posřední čísla  $d \in \mathbb{N}^+$  je systémem

velitelům ~~číslů~~ čísel  $a, b \in \mathbb{Z}_{(N)}$  jehož

$$d | a \quad \wedge \quad d | b$$

(NSD = největší společný dělitel.)

### Euklid. algor.

Platí (návratní NSD( $a, b$ ))

Nelze  $a, b \in \mathbb{N}^+$ . Bodem NSD( $a, b$ ) je největší posledním  
nezměněným zbytkem v Eukl. algor.  
na čísla  $a, b$

$$\text{tj. } \text{NSD}(a, b) = r_m$$

$$d | a \wedge d | b \iff d | \text{NSD}(a, b)$$

$$\text{NSD}(a, b) = \text{NSD}(b, r_m) = \text{NSD}(r_1, r_2) = \text{NSD}(r_{m-2}, r_m) = r_m$$

## Řetězové zlomky (konstrukce, vlastnosti)

Jedná se o nejlepší reálnou approximaci reálných čísel.

## Definice

Řetězovým zlomkem nazveme (konečný nebo nekonečný) výraz tvaru

$$q_0 + \cfrac{1}{q_1 + \cfrac{1}{q_2 + \cfrac{\ddots}{\cfrac{1}{q_n + \cfrac{1}{\ddots}}}}}$$

kde  $q_0 \in \mathbb{Z}, q_i \in \mathbb{N}^+ (i = 1, 2, \dots)$ .

Čísla  $q_i$  se nazývají členy rozvoje (v řetězový zlomek) a výrazy

$$\delta_0 = q_0, \delta_1 = q_0 + \cfrac{1}{q_1}, \dots, \delta_n = q_0 + \cfrac{1}{q_1 + \cfrac{1}{q_2 + \cfrac{\ddots}{\cfrac{1}{q_n}}}}, \dots$$

$$\vdots$$

$$+ \cfrac{1}{q_n}$$

se nazývají přibližné zlomky.

$$q_i \in \mathbb{Z}, q_i \in \mathbb{N} \quad (i = 1, 2, \dots)$$

$q_i$  ... členy rozvoje v řetězový zlomek

 $\delta_0 = q_0, \delta_1 = q_0 + \cfrac{1}{q_1}, \dots, \delta_n = q_0 + \cfrac{1}{q_1 + \cfrac{1}{q_2 + \cfrac{\ddots}{\cfrac{1}{q_n}}}}, \dots$ 

$\delta_n$  ... i-lého přibližného zlomku

 $\delta_i = [q_0, q_1, q_2, \dots, q_i]$ 
 $\delta_i = \frac{p_i}{q_i}$ 

$p_i$  ... čitatel i-lého přibližného zlomku

$q_i$  ... jmenovatel i-lého přibl. zlomku

Příklad: Reálné číslo  $a$  má řádný konečný (nekončející) rozvoj v řetězový zlomek právě tedyže  $a$  je理rationální číslo ( $a \in \mathbb{Q}$ )

## Řešení kongruencí 1. stupně a jejich soustav

Modulo:

### Definice

Řekneme, že celá čísla  $a, b$  jsou kongruentní modulo  $m$ , kde  $m \in \mathbb{N}^+$ , jestliže obě čísla mají při dělení modulem  $m$  stejný zbytek.

Skutečnost, že čísla  $a, b$  jsou kongruentní modulo  $m$  vyjadříme některým z následujících zápisů:

$$a \equiv b \pmod{m}, \text{ resp. } a \equiv b \text{ (mod } m).$$

V opačném případě ( $a, b$  nemají při dělení modulem  $m$  stejný zbytek) píšeme  $a \not\equiv b \pmod{m}$  a říkáme, že uvedená čísla nejsou kongruentní modulo  $m$ . Dále budeme používat zápis  $a = (b \text{ mod } m)$ , kterým vyjádříme skutečnost, že  $a$  je rovno zbytku při dělení čísla  $b$  modulem  $m$ . Například  $386 \equiv 777 \pmod{17}$ , ale  $12 = (386 \text{ mod } 17)$ .

## Kongruence

Kongruence je výraz

$$f(x) \equiv 0 \pmod{m},$$

kde  $m \in N - \{0,1\}$  je modul,

$f(x) = a_n x^n + \dots + a_1 x + a_0$  je nenulový polynom nad  $Z_m$  (tj.  $\forall i a_i \in Z_m$ ) a navíc  $m \nmid a_n$   
(číslo  $n$  nazýváme stupněm kongruence).

Hlavním cílem bude nalézt všechna  $x \in Z$ , pro která uvedená kongruence platí. V tomto kontextu je podstatné si uvědomit (dokažte!), že platí

$$f(x_0) \equiv 0 \pmod{m} \rightarrow \forall t \in Z \quad f(x_0 + mt) \equiv 0 \pmod{m}$$

a tedy pokud zkoumané kongruenci vyhovuje jisté celé číslo  $x_0$ , vyhovují jí také všechna celá čísla, která jsou s  $x_0$  kongruentní modulo  $m$ , tj.  $x \equiv x_0 \pmod{m}$ . Z těchto důvodů je rozumné, a dále tak budeme činit, považovat za jedno řešení celou zbytkovou třídu  $[x_0] = \{x_0 + mt | t \in Z\}$ . Zřejmým důsledkem pak je skutečnost, že uvedená kongruence má nejvýše  $m$  řešení (řádně zdůvodňte!), která lze nalézt metodou „hrubé síly“, tj. postupným dosazováním čísel  $0, 1, \dots, m-1$ . Na druhé straně je celá řada relevantních důvodů, pro které je vhodné znát efektivnější způsoby řešení. Vzhledem k rozsahu skript se omezíme na metody řešení kongruencí 1. stupně, tj. kongruencí tvaru

$$ax \equiv b \pmod{m}$$

### Poznámka – řešení obecných kongruencí 1. stupně

- Uvažujme obecnou kongruenci 1. stupně a označme  $d = NSD(m, a)$ . V situaci, kdy  $d = 1$  postupujeme dle předchozího tvrzení, proto předpokládejme, že  $d \geq 2$ . V tomto případě platí:
  - Jestliže  $d \nmid b$ , potom kongruence  $ax \equiv b \pmod{m}$  nemá řešení (řádně zdůvodňte!).
  - Jestliže  $d|b$ , potom kongruence  $ax \equiv b \pmod{m}$  má právě  $d$  následujících modulo  $m$  nekongruentních řešení

$$x \equiv x_0; x_0 + m_1; \dots; x_0 + (d-1)m_1 \pmod{m},$$

kde  $x_0$  je jediné řešení kongruence  $a_1 x \equiv b_1 \pmod{m_1}$ ,  $a_1 = a/d$ ,  $b_1 = b/d$ ,  $m_1 = m/d$ .

## Řešení soustav kongruencí 1. stupně – pomocí Čínské věty o zbytku

### Tvrzení - Čínská věta o zbytku

Uvažujme soustavu kongruencí tvaru

$$x \equiv b_1 \pmod{m_1}, \dots, x \equiv b_k \pmod{m_k},$$

kde  $\forall i \neq j NSD(m_i, m_j) = 1$ . Potom daná soustava má pro libovolné pravé strany  $b_1, \dots, b_k$  právě jedno řešení modulo  $M = m_1 \cdot \dots \cdot m_k$ . Toto řešení je tvaru

$$x \equiv x_0 \pmod{M},$$

kde  $x_0 = M_1 \cdot \bar{M}_1 \cdot b_1 + \dots + M_k \cdot \bar{M}_k \cdot b_k$ ,  $M_i = M/m_i$  a  $M_i \cdot \bar{M}_i \equiv 1 \pmod{m_i}$ .

### Poznámky

- Řešení soustavy  $k$  kongruencí 1. stupně převádíme na řešení  $k$  „nezávislých“ kongruencí 1. stupně, totiž  $M_i \cdot \bar{M}_i \equiv 1 \pmod{m_i}$ , kde  $i \in \{1, \dots, k\}$ .
- Při řešení soustavy kongruencí  $a_1 x \equiv b_1 \pmod{m_1}, \dots, a_k x \equiv b_k \pmod{m_k}$ , kde  $\forall i \neq j NSD(m_i, m_j) = 1$  a  $\forall i NSD(a_i, m_i) = 1$  postupujeme tak, že každou kongruenci vyřešíme, čímž původní soustavu převedeme na ekvivalentní tvar  $x \equiv c_1 \pmod{m_1}, \dots, x \equiv c_k \pmod{m_k}$ . Vzhledem k tomu, že  $\forall i \neq j$  platí  $NSD(m_i, m_j) = 1$ , postupujeme při jejím řešení dle Čínské věty o zbytku.

### Tvrzení - zobecněná Čínská věta o zbytku

Uvažujme soustavu kongruencí tvaru

$$x \equiv b_1 \pmod{m_1}, \dots, x \equiv b_k \pmod{m_k}.$$

Potom uvedená soustava má řešení právě tehdy, jestliže

$$\forall i \neq j \ NSD(m_i, m_j) | (b_i - b_j).$$

Označíme-li navíc  $M = NSN(m_1, \dots, m_k)$  a  $c_i, d_i, i = 1, \dots, k$  taková čísla, že

$$[M = d_1 \cdot \dots \cdot d_k] \wedge [\forall i \ d_i | m_i] \wedge [\forall i \neq j \ NSD(d_i, d_j)],$$

$$\forall i \ [c_i \equiv 0 \pmod{M/d_i}] \wedge [c_i \equiv 1 \pmod{d_i}],$$

potom jediné řešení soustavy je tvaru

$$x \equiv c_1 \cdot b_1 + \dots + c_k \cdot b_k \pmod{M}.$$

## 2. Elementární algebra – cyklická grupa, symetrická grupa. Polynomy nad tělesem (základní pojmy, operace s polynomy), irreducibilita nad R, C, Zp. Konečná tělesa.

Algebra -  $A \neq \emptyset$  (... neprázdná množina)

& operace na  $A$

mj.  $(\mathbb{Z}, +, \cdot)$

Binární operace (obecnější operace)

Def: (bin. operace na množině)

$A \neq \emptyset$  ... nosič operace

\* ... operace (bin.) na  $A$

\*:  $A^2 \rightarrow A$

$\underbrace{(a, b)}_{\substack{a, b \in A \\ \text{operandy}}} \rightarrow \underbrace{a * b \in A}_{\substack{\text{výsledek operace}}}$

(unární operace - předpis:  $\mathbb{F}: A \rightarrow A$  )

### Algebra

- a) S jednou binární operací (grupa)
- b) Se dvěma binárními operacemi (okruh  $\rightarrow$  obor integrity  $\rightarrow$  tělesa)

#### Definice - grupa

Nechť \* je binární operace na množině  $G \neq \emptyset$  (tzv. nosič grupy) s vlastnostmi:

- $\forall a, b, c \in G \quad (a * b) * c = a * (b * c)$  ... asociativita
- $\exists e \in G \quad \forall a \in G \quad a * e = e * a = a$  ... existence neutrálního prvku
- $\forall a \in G \quad \exists \bar{a} \in G \quad a * \bar{a} = \bar{a} * a = e$  ... symetrizovatelnost

Potom uspořádanou dvojici  $(G, *)$  nazýváme grupou.

zde nazíváme operace \* komutativní, nulovému & komutativní, nulovému & komutativnímu.

$(G, \cdot)$  multiplikativní grupa

$(G, +)$  aditivní grupa

#### Definice – cyklická grupa/podgrupa

Nechť  $(G, \cdot)$  je grupa,  $g \in G$ .

- a) Podgrupu  $(\langle g \rangle, \cdot)$  definovanou v předchozím tvrzení nazýváme cyklickou podgrupou grupy  $(G, \cdot)$ .
- b) Grupu  $(G, \cdot)$  nazýváme cyklickou, jestliže  $\exists g \in G$  takový, že  $G = \langle g \rangle$ .

Výše zmíněný prvek  $g \in G$  nazýváme generátorem grupy, resp. podgrupy.

**Generátor** cyklické grupy je prvek, jehož opakováním násobením (v aditivní notaci sčítáním) získáme všechny prvky této grupy.

#### Definice - permutace

Nechť  $A$  je  $n$ -prvková množina, tj.  $|A| = n$ . Potom permutací na množině  $A$  rozumíme libovolné vzájemně jednoznačné zobrazení  $A$  na  $A$ .

- Permutace budeme značit symboly  $\pi, \rho, \sigma, \dots$  a pro jejich zápis budeme využívat tzv. dvouřádkový zápis (později také zápis ve tvaru součinu obvykle disjunktních) cyklů.
- Dvouřádkový zápis permutace na množině  $A = \{1, 2, \dots, n\}$  má tvar

$$\pi = \begin{pmatrix} 1 & 2 & \cdots & n \\ \pi(1) & \pi(2) & \cdots & \pi(n) \end{pmatrix},$$

kde horní řádek obsahuje vzory a dolní řádek obsahuje jim odpovídající obrazy. Je zřejmě, že při zápisu permutace není podstatné pořadí sloupců, podstatné je pouze přiřazení obrazů vzorům. Z tohoto pohledu např. zápis  $\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 4 & 2 & 5 & 1 \end{pmatrix}, \begin{pmatrix} 3 & 1 & 5 & 2 & 4 \\ 2 & 3 & 1 & 4 & 5 \end{pmatrix}$  definují stejnou permutaci.

Označíme-li  $S_n$  množinu všech permutací na  $n$ -prvkové množině, lze na  $S_n$  definovat operaci násobení permutací  $\pi, \rho \in S_n$  jako skládání zobrazení, tj. následovně:

$$\pi\rho = \begin{pmatrix} 1 & 2 & \cdots & n \\ \rho(\pi(1)) & \rho(\pi(2)) & \cdots & \rho(\pi(n)) \end{pmatrix}.$$

**Cyklus** = uzavřená permutace

- Lze ho zapsat jako součin transpozic

**Definice – symetrická/permutační grupa**

Grupu  $(S_n, \cdot)$  nazýváme symetrickou grupou na  $n$ -prvkové množině. Každou podgrupu grupy  $(S_n, \cdot)$  nazýváme permutační grupou.

## Polynomy nad tělesem (základní pojmy, operace s polynomy)

**Definice – těleso**

Nechť  $+, \cdot$  jsou dvě binární operace na  $T \neq \emptyset$  (nosič tělesa) takové, že  $(T, +)$  tvoří abelovu grupu,  $(T - \{0\}, \cdot)$  tvoří abelovu grupu a platí distributivní zákony. Potom uspořádanou trojici  $(T, +, \cdot)$  nazýváme tělesem.

**Abelova grupa** je komutativní grupa, ve které operace skládání dvou prvků nezávisí na jejich pořadí, tj. pro všechny prvky  $(a)$  a  $(b)$  platí  $(a + b = b + a)$ .

**Definice - polynom**

Nechť  $(T, +, \cdot)$  je těleso,  $a_0, a_1, \dots, a_n \in T, a_n \neq 0, x \notin T$ . Potom výraz  $a_0 + a_1x + \cdots + a_nx^n$  nazýváme polynomem stupně  $n$  (v neurčité  $x$ ) nad tělesem  $T$ . Polynom, jehož vedoucí koeficient (koeficient u nejvyšší mocnině  $x$ , tj.  $a_n$ ) je roven 1 se nazývá monický polynom.

**Množina všech polynomů** nad tělesem  $T$  se značí  $T[x]$

**Stupeň polynomu** st  $f(x)$  je roven nejvyšší mocnině, u které se vyskytuje nenulový koeficient

**Kořen polynomu** je  $f(x) \in T[x]$  je definován jako libovolné  $b \in T$  taková, že  $f(b) = 0$

**Sčítání a násobení polynomů:**

Nyní na množině  $T[x]$  nařízíme (všeobecně známé) operace sčítání a násobení polynomů.

Nechť  $f(x), g(x) \in T[x]$ , kde  $f(x) = \sum_{i=0}^n a_i x^i, g(x) = \sum_{i=0}^m b_i x^i$ . Potom:

$$f(x) + g(x) = \sum_{i=0}^{\max\{st(f), st(g)\}} (a_i + b_i)x^i,$$

$$f(x) \cdot g(x) = \sum_{k=0}^{st(f)+st(g)} c_k x^k, \text{ kde } c_k = \sum_{i=0}^k a_i b_{k-i}.$$

Zřejmě platí (zdůvodněte):  $st(f+g) \leq \max\{st(f), st(g)\}$ ,  $st(f \cdot g) = st(f) + st(g)$ .

Násobení lze konvolucí

## Dělení polynomu se zbytkem:

Věta: (Dělení polynomu se zbytkem)

Nechť  $f(x), g(x) \in T[x]$ ,  $g(x)$  je nenulový polynom.

Potom existuje jedinečný polynom  $q(x), r(x) \in T[x]$

hodlající, že  $f(x) = q(x) \cdot g(x) + r(x)$ , kde je  $\deg r(x) < \deg g(x)$

$q(x)$  ... výplňový polynom

$r(x)$  ... zbytek

## Ireducibilita nad $R, C, Z_p$

**Ireducibilní polynom** nelze rozložit na součin nižších polynomů s koeficienty ve stejném tělese

$C$  ... komplexní čísla;  $R$  ... reálná čísla

### Tvrzení – Základní věta algebry

Nechť  $f(x) \in C[x], st(f) = n \geq 1$ . Potom  $f(x)$  má v tělese  $C$  alespoň jeden kořen.

### Tvrzení - ireducibilita nad $R, C$

- a) Jediné irreducibilní polynomy nad  $C$  jsou právě všechny polynomy 1. stupně. Každý polynom  $f(x) \in C[x]$  lze proto rozložit jediným způsobem na součin

$$f(x) = a(x - r_1)^{n_1} \cdot \dots \cdot (x - r_k)^{n_k},$$

kde  $a, r_1, \dots, r_k \in C$ ,  $\forall i \neq j r_i \neq r_j$  a  $\sum_{i=1}^k n_i = n$ .

- b) Jediné irreducibilní polynomy nad  $R$  jsou právě všechny polynomy 1. stupně a všechny polynomy 2. stupně (tj.  $ax^2 + bx + c$ ) se záporným diskriminantem (tj.  $b^2 - 4ac < 0$ ). Každý polynom  $f(x) \in R[x]$  lze proto rozložit jediným způsobem na součin

$$f(x) = a(x - r_1)^{n_1} \cdot \dots \cdot (x - r_k)^{n_k} \cdot (x^2 + p_1x + q_1)^{m_1} \cdot \dots \cdot (x^2 + p_lx + q_l)^{m_l},$$

kde  $a, r_1, \dots, r_k, p_1, \dots, p_l, q_1, \dots, q_l \in R$ ,  $\forall i \neq j (r_i \neq r_j) \wedge ((p_i, q_i) \neq (p_j, q_j))$ ,

$\forall i (p_i^2 - 4q_i < 0)$ ,  $\sum_{i=1}^k n_i + 2 \sum_{i=1}^l m_i = n$ .

### Tvrzení – existenční věta pro irreducibilní polynomy nad $Z_p$

Nechť  $p$  je prvočíslo,  $n \in N^+$ . Potom existuje polynom stupně  $n$  irreducibilní nad  $Z_p$ .

**Těleso  $Z_p$**  je množina celých čísel modulo prvočíslo  $p$  vybavená operacemi sčítání, odčítání, násobení a dělení (kromě dělení nulou), kde každé nenulové číslo má multiplikativní inverzní prvek

## Konečná tělesa

### Definice - Galoisovo těleso

Těleso  $(Z_p[x] / q(x), +, \cdot)$ , kde  $st(q) = n$  nazýváme Galoisovo těleso a značíme  $GF(p^n)$ .

### Poznámka

Zdůrazněme, že Galoisovo těleso je nezávislé na volbě irreducibilního polynomu, ale pouze na jeho stupni, tj. každý polynom  $q(x)$  stupně  $n$  irreducibilní nad  $Z_p$  vede ke stejnemu (izomorfnímu)  $GF(p^n)$ .

### Tvrzení

Každé **konečné** těleso je izomorfní s Galoisovým tělesem  $GF(p^n)$  pro vhodné prvočíslo  $p$  a kladné přirozené číslo  $n$ .

(izomorfni = stejné)

### 3. (Ne)homogenní lineární rekurentní vztahy a jejich řešení (existence a jednoznačnosti řešení). Využití vytvořujících funkcí k jejich řešení. Rekurentní vztahy vybraných elementárních číselných posloupností.

Pozn.:

Posloupnost = zobrazení  $\pi N$  do množiny  $\mathbb{R}$

$$A: n \rightarrow a_n$$

- zápis:  $\{a_m\}_{m=0}^{\infty}$ ,  $(a_m)_{m=0}^{\infty}$

- definice posl.  $a_m$ : a) explicitně

b) rekurentní vztahy (el. v. d. diferenciální  
metoda)

c) výběrové funkce

Diference posloupností:  $\Delta a_n = a_{n+1} - a_n$

#### Rekurentní vztah

Rekurentní vztahy = výraz  $\Phi(n, a_0, a_1, \dots, a_m) = 0$ , kde

$\Phi$  je fce proměnných  $n, a_0, a_1, \dots, a_m$ ,  $n$  může být i vícero hodnot  $a_m$

Rádovní rek. vztahy = posl.  $\{a_m\}_{m=0}^{\infty}$ , pro kterou platí vztah mezi hodnotami proměnných  $n, a_0, a_1, \dots, a_m$  platí  $\Phi(\dots) = 0$

- když rádovní rek. vztah je el. v. s nějakou dif. rádou

$$\Psi(n, \Delta a_m, \Delta^{(2)} a_m, \dots, \Delta^{(m)} a_m) = 0$$

Lini. rek. vztahy =  $\Phi$  je lini. fct. členů posl.

#### Homogenní a nehomogenní rekurentní vztahy

Homogenní lini. rek. vztahy:  $C_0 a_{n+k} + C_{k-1} a_{n+k-1} + \dots + C_0 a_n = 0$

- řešení je posl., kde každou rovnost plní

$$a_n = K_1 a_n^{(1)} + \dots + K_k a_n^{(k)}, K_1, \dots, K_k \in \mathbb{R}$$

Nehomogenní lini. rek. vztahy:  $C_0 a_{n+k} + \dots + C_0 a_n = f_n$

- řešení je posl. složené z homogenní a funkcionální části

$$a_n = a_n^{(h)} + a_n^{(f)}$$

Homogenní i nehomogenní se dělí na lineární a nelineární.

## Charakteristický polynom

### Definice

Charakteristickým polynomem příslušným HLR vztahu (\*) rozumíme polynom

$$C_k x^k + C_{k-1} x^{k-1} + \cdots + C_1 x + C_0.$$

(stupeň charakteristického polynomu je roven řádu příslušného rekurentního vztahu a má  $k$  kořenů, počítáno včetně jejich násobnosti)

## Řešení homogenní lineárního rekurentních (HLR) vztahů

### Tvrzení

Nechť  $r_1, \dots, r_l$  jsou kořeny charakteristického polynomu a  $m_1, \dots, m_l$  jsou jejich násobnosti, potom posloupnost  $\{a_n\}_{n=0}^{\infty}$ , kde

$$a_n = r_1^n \left( K_1^{(1)} + K_2^{(1)} n + \cdots + K_{m_1}^{(1)} n^{m_1-1} \right) +$$

$$r_2^n \left( K_1^{(2)} + K_2^{(2)} n + \cdots + K_{m_2}^{(2)} n^{m_2-1} \right) +$$

⋮

$$r_l^n \left( K_1^{(l)} + K_2^{(l)} n + \cdots + K_{m_l}^{(l)} n^{m_l-1} \right),$$

$K_1^{(1)}, \dots, K_{m_l}^{(l)}$  jsou libovolné konstanty,

je obecným řešením HLR vztahu (\*).

### Postup:

1. Sestavení charakteristického polynomu
2. Určení jeho kořenů
3. Sestavení obecného řešení
4. Pokud určeny počáteční podmínky, dopočtou se i konstanty  $K_i$

## Řešení nehomogenní lineárního rekurentního (NHLR) vztahu

- NHLR vztah řádu  $k \geq 1$  má nekonečně mnoho řešení.

- Obecné řešení  $\{a_n\}_{n=0}^{\infty}$  NHLR vztahu má tvar

$$a_n = a_n^{(p)} + a_n^{(h)},$$

kde  $\{a_n^{(p)}\}_{n=0}^{\infty}$  je tzv. partikulární řešení (jedna konkrétní posloupnost vyhovující danému NHLR vztahu),

$\{a_n^{(h)}\}_{n=0}^{\infty}$  je obecné řešení HLR vztahu příslušného NHLR vztahu.

- Je-li předepsáno  $k$  počátečních podmínek (tj.  $k$  po sobě jdoucích hodnot  $a_i, \dots, a_{i+k-1}$ ), potom existuje jediné řešení NHLR vztahu, které těmto podmínkám vyhovuje.

## Využití vytvářejících funkcí k řešení rekurentních vztahů

Cíl: Nalézt explicitní vyjádření  $n$ -tého členu posloupnosti tvořící řešení rekurentního vztahu jako funkce  $n$

### **Postup pro NHLR (analogicky pro HLR):**

Označme  $\{a_n\}_{n=0}^{\infty}$  posloupnost, která je řešením zadaného NHLR vztahu (\*\*) a  $f(x)$  její vytvořující funkci.

1. Vynásobme obě strany rekurentního vztahu nejvyšší mocninou  $x^{n+k}$ . Dostáváme rovnici, která je platná pro každé  $n \in N$ , tj. dostáváme „nekonečně“ mnoho rovnic.
2. Nyní uvedené rovnice sčítáme přes  $n = 0, 1, \dots$ , čímž na levé straně získáme výrazy tvaru  $\sum_{n=0}^{\infty} a_{n+i} x^{n+i}$ , které vyjádříme pomocí symbolu pro vytvořující funkci, tj.  $f(x)$ . Dostáváme tak rovnici, jejíž neznámou je právě vytvořující funkce  $f(x)$  hledané posloupnosti.
3. Vyřešíme výše zmíněnou rovnici, čímž získáme uzavřený tvar vytvořující funkce  $f(x)$ .
4. Nalezneme rozvinutý tvar  $f(x)$ , přičemž koeficient u  $x^n$  je hledané explicitní vyjádření  $a_n$ .

### **Rekurentní vztahy vybraných elementárních číselných posloupností**

1. Fibonacciho posloupnost

$$F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$$

2. Geometrická posloupnost

Posloupnost:  $\{a_1, a_1 r, a_1 r^2, \dots\}$

Rekurzivní vztah:  $a_n = r a_{n-1}, n \geq 2$

3. Catalanova posloupnost

$C_n = C_0 C_{n-1} + C_1 C_{n-2} + \dots + C_{n-1} C_0$ , resp.  $C_{n+1} = C_0 C_n + C_1 C_{n-1} + \dots + C_n C_0$ , kde  $C_0 = 1$ .

Rekurentní vztah Catalanovy posloupnosti  $\{C_n\}_{n=0}^{\infty}$  pro  $n$ -tý člen je:

$$C_0 = 1, C_n = \frac{1}{n+1} C_{2n}^n$$

4. Bellova čísla

= posloupnost čísel, které počítají počet způsobů, jak rozdělit množinu  $n$  prvků do nezávislých podmnožin (particí)

$$\begin{aligned} B_0 &= 1 \\ B_{n+1} &= \sum_{k=0}^n \binom{n}{k} B_k \end{aligned}$$

## 4. Vytvořující funkce (obyčejné) – základní pojmy, operace s vytvořujícími funkcemi. Příklady vytvořujících funkcí elementárních číselných posloupností. Věžové polynomy.

### Vytvořující funkce

**Definice** – obyčejná vytvořující funkce

Nechť  $\{a_n\}_{n=0}^{\infty}$  je reálná, resp. komplexní posloupnost. Algebraický výraz (formální mocninnou řadu)

$$f(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n + \cdots = \sum_{n=0}^{\infty} a_n x^n,$$

resp. jakýkoliv s ním ekvivalentní tvar, nazveme (obyčejnou) vytvořující funkci posloupnosti  $\{a_n\}_{n=0}^{\infty}$ .

Pokud mám  $f(x) = a_0 + K_1 a_1 + \dots + K_n a_n$ , tak přes jednotlivá  $a_i$  se odstávám pomocí derivací (původní člen, např.  $a_0$  mi po derivaci zmizí a dostanu se k  $a_1$ )

Tyto funkce slouží k definování posloupnosti

Každá posloupnost má jednoznačně danou vytvořující funkci

### Tvary vytvořujících funkcí:

### Operace s vytvořujícími funkcemi

Vytvořující funkce	Posloupnost
$\alpha f(x) + \beta g(x)$	$\{\alpha a_n + \beta b_n\}_{n=0}^{\infty}$
$x^k f(x)$	$0, \underbrace{0, \dots, 0}_{k \times 0}, a_0, a_1, a_2, \dots$
$f(x) - (\sum_{i=k}^n a_i x^i)$	$a_0, \dots, a_{k-1}, \underbrace{0, \dots, 0}_{(n-k+1) \times 0}, a_{n+1}, a_{n+2}, \dots$
$f(x^k)$	$a_0, \underbrace{0, \dots, 0}_{(k-1) \times 0}, a_1, \underbrace{0, \dots, 0}_{(k-1) \times 0}, a_2, \underbrace{0, \dots, 0}_{(k-1) \times 0}, a_3, 0, \dots$
$\frac{f(x) - \sum_{i=0}^{k-1} a_i x^i}{x^k}$	$a_k, a_{k+1}, a_{k+2}, \dots, \text{tj. } \{a_{k+n}\}_{n=0}^{\infty}$
$\frac{f(x)}{1-x}$	$a_0, a_0 + a_1, a_0 + a_1 + a_2, \dots, \text{tj. } \{\sum_{i=0}^n a_i\}_{n=0}^{\infty}$
$f'(x)$	$a_1, 2a_2, 3a_3, \dots, n a_n, \dots, \text{tj. } \{(n+1)a_{n+1}\}_{n=0}^{\infty}$
$\int_0^x f(t) dt$	$0, a_0, \frac{a_1}{2}, \frac{a_2}{3}, \dots, \frac{a_n}{n+1}, \dots$
$f(x)g(x)$	$a_0 b_0, a_0 b_1 + a_1 b_0, a_0 b_2 + a_1 b_1 + a_2 b_0, \dots, \text{tj. } \{\sum_{i=0}^n a_i b_{n-i}\}_{n=0}^{\infty}$

## Věžové polynomy

- Věžové polynomy*
- řešenice =  $m \times m$  čtvercových polí kritických sloupců ( $m$ ) a řádků ( $m$ ) obdobných
  - normaci  $C_i$
  - Počet řešenice  $\nearrow$  připomínka  
zvýšení
  - Věž = ohnivnice se z jinou, jinou - li ve stejném řádku či sloupci
  - $n_k(C)$  normované věži na řádku neohnivnice se
  - Věžový polygon = vykreslující fce posl.  $\{n_k(C)\}_{k=0}^{\infty}$
  - $n(x, C) = n_0(C) + n_1(C)x + \dots + n_d(C)x^d$ ;  $d \leq \min\{m, n\}$
  - všechny řešenice lze rozložit na součinu diagonálních polinomů

## Příklady vytvořujících funkcí elementárních číselných posloupností

Posloupnost	Vytvořující funkce
$\{q^n\}_{n=0}^{\infty}$ (geometrická posloupnost)	$\frac{1}{1-qx}$
$\{\frac{1}{n!}\}_{n=0}^{\infty}$	$e^x$
$0, 1, \frac{1}{2}, \frac{1}{3}, \dots$	$-\ln(1-x)$
$0, 1, -\frac{1}{2}, \frac{1}{3}, -\frac{1}{4}, \frac{1}{5}, \dots$	$\ln(1+x)$
$\{n\}_{n=0}^{\infty}$	$\frac{x}{(1-x)^2}$
$\{n^2\}_{n=0}^{\infty}$	$\frac{x+x^2}{(1-x)^3}$
$\{n^3\}_{n=0}^{\infty}$	$\frac{x+4x^2+x^3}{(1-x)^4}$
$\{C_n^k\}_{k=0}^{\infty}, n \in \mathbb{Z}$ (binomické koeficienty)	$(1+x)^n$
$\{\bar{C}_n^k\}_{k=0}^{\infty}, n \in \mathbb{Z}$ (kombinace s opakováním)	$(1-x)^{-n}$
$\{F_n\}_{n=0}^{\infty}$ (Fibonacciho čísla)	$\frac{x}{1-x-x^2}$
$\{L_n\}_{n=0}^{\infty}$ (Lucasova čísla)	$\frac{2-x}{1-x-x^2}$
$\{C_n\}_{n=0}^{\infty}$ (Catalanova čísla)	$\frac{1-\sqrt{1-4x}}{2x}$
$a_{2n} = 0, a_{2n+1} = \frac{(-1)^n}{(2n+1)!}, n \in N$	$\sin x$
$a_{2n} = \frac{(-1)^n}{(2n)!}, a_{2n+1} = 0, n \in N$	$\cos x$

## 5. Problematika rozkladů, základní výsledky pro varianty – nerozlišitelné objekty do (ne)rozlišitelných tříd. Stirlingova čísla 1. a 2. druhu.

### Rozklady

Úloha: určování počtu rozkladů za určitých omezujících podmínek

Přehled:

Rozklady (je rozklad množin na třídy rozkladů) - Všechny řešení  
① Nerozlišitelné objekty Rozkladů

A) Do neřešitelných tříd

- a) třídy mohou být právě  
- počet odpovídající řešení diofantické rovnice
- $x_1 + \dots + x_k = m$ ,  $\Rightarrow P(k-1, m)$
- b) třídy mají jednoznačné minimum  $x_i \leq x_j$   
 $C_{m+k-1}^k - \sum_i x_i$
- c) třídy mají jednoznačný min. i max. počet obyvatel  
 $a_i \leq x_i \leq b_i \Rightarrow 0 \leq \underbrace{x_i - a_i}_{c_i} \leq \underbrace{b_i - a_i}_{c_i}$
- $x_1 + \dots + x_k = m - \sum_i a_i$ , výsledek je 1E
- $N(\sum_i x_i) = N - \sum_i N(x_i) + \sum_i N(x_i, x_j) + \dots$
- $N = P(k-1, m)$ ;  $N(x_i) = P(k-1, m - c_i)$

B) Do rozlišitelných tříd - počet rozkladů je konstanta

rozklaďte m na řídíce (z diophant. rov.)

② Rozlišitelné objekty

A) do k neprázdných tříd

- a) řídíce řídíce množiny  $\{\alpha_i\}$
- b) řídíce řídíce množiny  $[x_i]$  - řídíce řídíce  
diophantické rovnice

**Diofantická** rovnice je rovnice, typicky polynomální rovnice se dvěmi nebo více neznámých s celočíselnými koeficienty, pro kterého jediná celočíselná řešení jsou zajímavá. V rozkladech nás zajímá počet řešení.

**Princip inkluze-exkluze** je kombinatorický princip, který umožňuje spočítat počet prvků v průniku a sjednocení více množin pomocí vzájemného vyjmutí (exkluze) a přidání (inkluze).

### Stirlingova čísla 2. druhu

**Definice** – Stirlingova čísla 2. druhu

Počet, kolika způsoby lze rozložit množinu obsahující  $n$  různých prvků do  $k$  neprázdných tříd, kde na pořadí tříd ani prvků v třídách nezáleží, značíme  $S(n, k)$  a nazýváme Stirlingovo číslo 2. druhu řádu  $n, k$ .

### Poznámky

- V kontextu rozkladů se místo pojmu Stirlingovo číslo 2. druhu běžně používá termín Stirling subset number (řádu  $n, k$ ) a značí se  $\{n\}_k$ , tj.  $S(n, k) = \{n\}_k$ . Oba názvy i symboly budeme používat jako ekvivalenty. Nicméně označení pomocí složených závorek budeme preferovat, neboť je návodné a zdůrazňuje skutečnost, že jde o třídy rozkladu, které chápeme jako množiny, tedy nezáleží na pořadí prvků v třídách rozkladu (na rozdíl od níže zavedených Stirling cycle numbers).
- Pro některé hodnoty  $n, k \in N$  je snadné určit  $\{n\}_k$  explicitně. Platí:

$$\begin{aligned}\{0\}_0 &= 1, & \{n\}_0 &= 0, \text{ pro } n \in N^+, & \{n\}_1 &= 1, \text{ pro } n \in N^+, \\ \{n\}_2 &= 2^{n-1} - 1, \text{ pro } 2 \leq n, & \left\{ \begin{array}{c} n \\ n-1 \end{array} \right\} &= \frac{n(n-1)}{2}, \text{ pro } n \in N^+, & \{n\}_n &= 1, \text{ pro } n \in N.\end{aligned}$$

### Tvrzení

Pro Stirlingova čísla 2. druhu (Stirling subset numbers) platí pro  $1 \leq k \leq n$  rekurentní vztah

$$\{n\}_k = \left\{ \begin{array}{c} n-1 \\ k-1 \end{array} \right\} + k \left\{ \begin{array}{c} n-1 \\ k \end{array} \right\}.$$

(s počátečními podmínkami uvedenými v předchozí poznámce)

## Stirling cycle numbers

### Definice – Stirling cycle numbers

Počet, kolika způsoby lze rozložit množinu obsahující  $n$  různých prvků do  $k$  neprázdných cyklů značíme  $[n]_k$  a nazýváme Stirling cycle number<sup>1</sup> řádu  $n, k$ .

### Poznámky

- Jak Stirling subset numbers  $\{n\}_k$ , tak i Stirling cycle numbers  $[n]_k$  se vztahují k rozkladům  $n$ -prvkové množiny na  $k$  neprázdných nerozlíšitelných tříd, resp. do  $k$  cyklů. Rozdíl spočívá v tom, že v případě Stirling subset numbers nezáleží na pořadí prvků v jednotlivých třídách rozkladu, kdežto v případě Stirling cycle numbers jsou prvky každé třídy rozkladu uspořádány na kružnici, a tedy záleží na tom, jak jsou na kružnici uspořádány. Pro libovolná  $0 \leq k \leq n$  proto zřejmě platí  $0 \leq \{n\}_k \leq [n]_k$ .
- Stirling cycle number  $[n]_k$  lze definovat také jako počet permutací na  $n$ -prvkové množině, které lze zapsat pomocí  $k$  disjunktních cyklů, tudíž platí  $\sum_{k=0}^n [n]_k = n!$
- Pro některé hodnoty  $n, k \in N$  je snadné určit  $[n]_k$  explicitně. Platí:

$$\begin{aligned}[0]_0 &= 1, & [n]_0 &= 0, \text{ pro } n \in N^+, & [n]_1 &= (n-1)!, \text{ pro } n \in N^+, \\ \left[ \begin{array}{c} n \\ n-1 \end{array} \right] &= \frac{n(n-1)}{2}, \text{ pro } n \in N^+, & [n]_n &= 1, \text{ pro } n \in N. & & \text{(řádně zdůvodněte!)}\end{aligned}$$

### Tvrzení

Pro Stirling cycle numbers platí pro  $1 \leq k \leq n$  rekurentní vztah

$$[n]_k = \left[ \begin{array}{c} n-1 \\ k-1 \end{array} \right] + (n-1) \left[ \begin{array}{c} n-1 \\ k \end{array} \right].$$

## Stirlingova čísla 1. druhu

Jsou vlastně Stirling cycle numbers, ale dochází u nich ke změnám znaménka (podrobně dále).

### Poznámky – Stirlingova čísla 1. a 2. druhu

- Je zřejmé, že padající faktoriál  $x^n$  je polynom stupně  $n$  a lze ho proto psát ve tvaru

$$x^n = \sum_{k=0}^n s(n, k) \cdot x^k.$$

Koeficienty  $s(n, k)$  se nazývají Stirlingova čísla 1. druhu.

Mezi Stirlingovými čísly 1. druhu a Stirling cycle numbers platí vztah

$$s(n, k) = (-1)^{n-k} \begin{Bmatrix} n \\ k \end{Bmatrix},$$

tedy

$$x^n = \sum_{k=0}^n (-1)^{n-k} \begin{Bmatrix} n \\ k \end{Bmatrix} \cdot x^k.$$

- Snadno lze ukázat i obrácené tvrzení, že mocninu  $x^n, n \in N$  lze jednoznačně vyjádřit ve tvaru lineární kombinace padajících faktoriálů  $x^k, k = 0, 1, \dots, n$ , tj. ve tvaru

$$x^n = \sum_{k=0}^n S(n, k) \cdot x^k.$$

Koeficienty  $S(n, k)$  se jsou právě Stirlingova čísla 2. druhu, resp. Stirling subset numbers a platí již zmíněný vztah  $S(n, k) = \begin{Bmatrix} n \\ k \end{Bmatrix}$ , takže lze také psát  $x^n = \sum_{k=0}^n \begin{Bmatrix} n \\ k \end{Bmatrix} \cdot x^k$ .

## 6. Minimální kódy – základní pojmy (Kraftova nerovnost, nejkratší kód), Huffmanova konstrukce, aritmetické kódy. Adaptivní metody (Huffman).

**Minimální kódování** je způsob kódování, jehož účelem je zmenšit objem dat ve zprávě

Proces označován jako *datová komprese (komprese dat)* – inverzně k němu expanze

Dělí se na *bezzrátovou* a *ztrátovou*

**Prosté kódování:** Taková, kdy různým zdrojovým znakům odpovídají vždy různá kódová slova

**Prefix slova:** Posloupnost znaků, která se nachází na začátku slova. Např.: PREFIX slova  $b_1b_2\dots b_k$  je každé ze slov  $b_1, b_1b_2, \dots, b_1b_2\dots b_k$ .

**Prefixový kód:** Kódování se nazývá prefixové, jestliže je prosté a žádné kódové slovo není prefixem jiného kódového slova.

### Kraftova nerovnost:

Při kódování  $n$  znaky můžeme sestavit prefixový kód s délkami slov  $d_1, d_2, \dots, d_r$ , právě když platí tzv. Kraftova nerovnost.

$$n^{-d_1} + n^{-d_2} + \dots + n^{-d_r} \leq 1$$

- Kraftova nerovnost slouží pro volbu délky kódových slov

### Nejkratší kód:

Nejkratším  $n$  znakovým kódováním zdrojové abecedy  $a_1, a_2, \dots, a_r$  s pravděpodobnostmi výskytu  $p_1, p_2, \dots, p_r$  se rozumí prefixové kódování této abecedy pomocí  $n$  znaků, které má nejmenší možnou průměrnou délku slova  $d$ .

**Průměrná délka slova:**  $\bar{d} = d_1p_1 + d_2p_2 + \dots + d_rp_r$

### Huffmanův algoritmus kódování

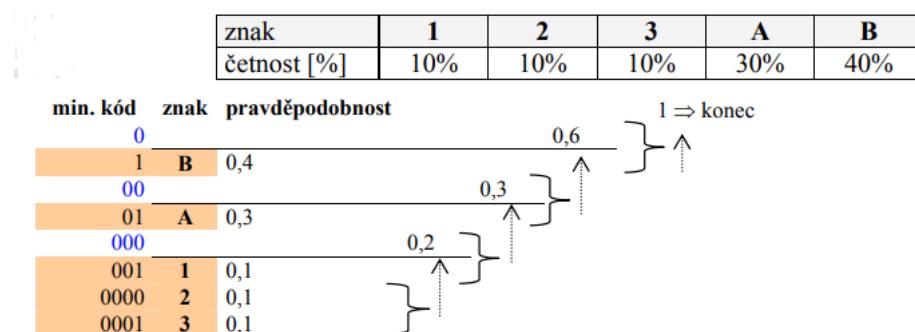
- Používá se pro bezzrátovou kompresi dat, např. JPEG, MP3
- Součást programů jako ZIP
- Základem je myšlenka zakódování znaku podle počtu výskytu ve vstupu

#### a) Huffmanova konstrukce binárního kódu:

- Je vždy optimální, ale není jednoznačná

**Algoritmus 1** (postup vhodný pro „ruční“ výpočet):

1. Zdrojové znaky uspořádáme tak, aby platilo:  $p_1 \geq p_2 \geq \dots \geq p_r$  (nerostoucí posloupnost).
2. Znaky  $a_1, a_2, \dots, a_r$  napíšeme pod sebe a vedle nich jejich pravděpodobnosti.
3. Sečteme poslední dvě pravděpodobnosti a výsledek zařadíme do posloupnosti dle 2.
4. Sečteme poslední dvě dosud nesečtené pravděpodobnosti a výsledek opět zařadíme.
5. Krok 4. opakujeme až do součtu pravděpodobností = 1.
6. Poté přiřazujeme kódová slova od prvního znaku  $a_1$  směrem k dalším dle prefixového principu. Nekódová slova využíváme jako použitelné prefixy mimo poslední součet.



## b) Huffmanova konstrukce obecného kódu

- Jako pro binární, ale místo 2 pravděpodobností, bereme obecně n
- Např. pro tříznakový (ternární kódování) postupně sčítáme (redukujeme) tři sčítance. Výjimku tvoří první součet, při kterém sčítáme s pravděpodobností. Určení čísla s můžeme provádět tak, že v seznamu zdrojových symbolů vytvoříme skupiny po  $n-1$  symbolech, až zbude poslední skupina počtu  $s \in [2, n]$ .

znak	pravděpodobnost	ternární min. kód, n=3
A	2/8	1
B	2/8	2
C	1/8	01
D	1/8	02
E	1/8	000
F	1/8	001

### Varianty Huffmmana:

#### 1) Statická varianta:

- Je dvoufázová
- Nejprve se provádí výpočet četnosti, resp. pravděpodobnosti výskytu daného znaku ve vstupním proudě. Následně se dle algoritmu vypočte prefixový kód, tj. provede se zakódování znaků zdrojové abecedy pomocí prefixového kódu z kódové abecedy
- Algoritmus využívá tzv. binární stromy, kde hodnoty uzlů od kořene k listům stromu tvoří jednotlivá kódová slova

#### 2) Adaptivní varianta:

- Vychází z předchozí statické varianty
- Zásadním rozdílem adaptivních kódů je pouze jeden průchod textem. Celý proces funguje tak, že algoritmus opět vytvoří binární strom, jehož listy jsou jednotlivé znaky. Poté začne kódovat, přičemž si u každého znaku vede statistiku kolikrát se v textu vyskytl. Takto vzniká po každém zakódovaném znaku nová statistika, a tedy i nový strom, který je přehodnocen po každém znaku
- Implementace je časově náročnější

## Aritmetické kódování

= algoritmus bezetrátové komprese dat, který provádí kódování celého vstupního řetězce do jednoho čísla. Obvykle na intervalu  $[0, 1]$

**Princip:** Fungování spočívá v přiřazení každému symbolu určitého intervalu reálných čísel, který je závislý na jeho pravděpodobnosti výskytu. Poté se tento interval postupně zužuje na základě vstupního textu. Kódující algoritmus pak kóduje celý text do jednoho čísla, které leží v konečném intervalu.

**Dekódování:** Číslo se rozdělí do částí, které odpovídají intervalům přiřazeným jednotlivým symbolům. Poté je text rekonstruován na základě těchto intervalů a postupně dekódován.

**Výhody:** Je schopné dosáhnout vyšší míry komprese než Huffmanovo kódování, pokud je správně navrženo a vstupní symboly mají různé pravděpodobnosti výskytu

**Nevýhody:** Jejich implementace je složitější a vyžaduje více výpočetních prostředků

## **7. Bezpečnostní kódy: Lineární kódy, jejich základní vlastnosti. Generující a kontrolní matice. Hammingovská vzdálenost, schopnost detekce a opravy chyb. Hammingovy kódy, kód (7,4)**

### **Bezpečnostní kódování**

(protichybové kódování, detekční a korekční kódování) je užito pro přenos informace reálným přenosovým kanálem, tj. kanálem, kde může dojít k ovlivnění přenášené informace vlivem chyby (šum, ztráta informace)

- Účel bezpečnostního kódování je detektovat, případně i přímo opravit vzniklou chybu. Možné chybě čelíme zavedením přídavné informace k přenášeným datům (tj. příjemce obdrží jiný znak, než odesilatel původně vyslal)

**Bezpečnostní kódy** uměle zvyšují redundanci, za účelem zabezpečení informace před možnými chybami.

Dva typy:

- a) **detekční kódy** (error-detection codes) – které umožňují pouze detektovat, že přijatý znak je chybný
- b) **korekční kódy** (samoopravné kódy, self-correcting codes) – kromě detekce chyby umožňují i opravu chybně přeneseného znaku, takže jej není nutné přenášet znovu (což u detekčního kódu obecně nutné je), např. blokové kódy

**Lineární prostor** je tvořen všemi řešeními lineárních rovnic o  $n$  neznámých, tedy podprostoru prostoru  $T^n$ . Je definován operací sčítání vektorů, násobení vektoru skalárem, asociativita a komutativitou sčítání, existencí nulového prvku, existence opačného prvku

Pozn.: Pro binární kódy je  $T = \{0,1\}$

Kódované znaky se dělí na *informační* a *kontrolní*.

**Dimenze kódu** je rovna počtu informačních znaků.

### **Lineární kód:**

Binární kód  $K$  se nazývá lineární, jestliže je podprostorem lineárního prostoru  $\{0,1\}^n$ , tj. jestliže součet dvou kódových slov je kódové slovo. Je-li  $K$  podprostorem dimenze  $k$ , mluvíme o lineárním  $(n,k)$ -kódu. Každý lineární  $(n, k)$ -kód má  $k$  informačních a  $n-k$  kontrolních znaků.

### **Kontrolní matice:**

Je-li binární kód popsán soustavou homogenních lineárních rovnic, potom matice  $H$  této soustavy se nazývá kontrolní maticí kódu.

Kontrolní matice lineárního kódu  $K$  je taková matice  $H$  z prvků abecedy  $T$ , pro kterou platí: slovo  $v = v_1v_2\dots v_n$  ( $v \in T^n$ ) je kódové, právě když splňuje následující podmínu.

$$Hv^T = 0^T \quad v \in K, \quad v = v_1v_2\dots v_n$$

### **Generující matice:**

Lineární  $(n, k)$ -kód s  $k \neq 0$  je určen svoují (libovolnou) bází  $b_1, b_2, \dots, b_k$ . Pokud napíšeme těchto  $k$  slov (bází) pod sebe, vznikne tzv. generující matice daného kódu  $G$ .

$$G = [b_1 \ b_2 \ \dots \ b_k]^T$$

Matice  $G$  typu  $(n, k)$  je generující maticí lineárního kódu, jestliže

- každý její řádek je kódovým slovem,
- každé kódové slovo je lineární kombinací řádků,
- řádky jsou lineárně nezávislé, takže hodnota matice  $G$  je rovna  $k$ .

## Hammingova vzdálenost

Hammingova vzdálenost  $\delta_H$  dvou slov  $v_1v_2\dots v_n$  a  $w_1w_2\dots w_n$  je definována jako počet odlišných znaků těchto slov, tj. velikost množiny  $\{i \mid i = 1, 2, \dots, n; v_i \neq w_i\}$ .

Hammingovu vzdálenost dvou kódových složek můžeme geometricky interpretovat jako počet hran krychle kterými je třeba projít na cestě vedoucí od jedné složky ke druhé.

**Minimální kódová vzdálenost (kódová vzdálenost)**  $D$ , blokového kódu  $K$ , je spočtena

jako minimální Hammingova vzdálenost mezi dvěma různými kódovými složkami.

**Detekční kódy** jsou pouze schopny chybu najít

**Korekční kódy** umí kromě detekce chyby i chybu opravit

**Pro detekční a korekční schopnosti kódu platí:**

- Detekční schopnost:  $D \geq td+1$
- Korekční schopnost:
  - A) pro  $D$  lichá:  $D \geq 2tc + 1$
  - B) pro  $D$  sudá:  $D \geq 2tc + 2$

kde  $tc$  je počet chybných prvků a  $D$  je kódová vzdálenost.

## Hammingovy kódy:

Třída kódů, které opravují jednoduché chyby

Snadno se dekódují a jsou tzv. **perfektní**, tj. mají nejmenší myslitelnou redundanci.

Hammingův binární kód s  $m$  kontrolními znaky ( $m=2, 3, 4, \dots$ ) má délku  $2m-1$ , takže dostáváme binární  $(2^m - 1, 2^m - m - 1)$ -kód, tj.  $(3,2)$ -kód,  $(7,4)$ -kód,  $(15,11)$ -kód, atd.

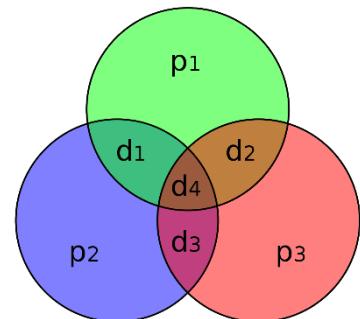
**Hammingův kód:** Binární kód se nazývá Hammingův, jestliže má kontrolní matici, jejíž sloupce jsou všechna nenulová slova dané délky a žádné z nich se neopakuje.

## (7,4)-kód

Také jako Hamming(7,4) je lineární kód pro opravu chyb, který zakóduje čtyři bity dat do sedmi bitů přidáním tří paritních bitů.

Pro  $m=3$  dostáváme Hammingův (7,4)-kód.

$$\mathbf{H} = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix} \xrightarrow{\text{permutace sloupců}} \begin{bmatrix} 1 & 7 & 2 & 6 & 4 & 5 \end{bmatrix} \rightarrow \mathbf{H}' \Rightarrow \mathbf{G}' \Rightarrow \mathbf{G}$$



## 8. Symetrické a asymetrické kryptosystémy – základní principy, vlastnosti.

### Hybridní kryptosystémy. Elektronický podpis, certifikáty. Hashovací funkce pro kryptografii.

Pozn. k šifrování:

- Kryptologie = kryptografie + kryptoanalýza
- Kryptografie – jak šifrovat
- Kryptoanalýza – jak lámat šifry

Kryptografické metody děleny dle práce s klíčem (heslem) na metody s tajným klíčem a veřejným klíčem

Dva typy šifrování:

- a) **Symetrické šifrování** – šifrovací algoritmus, který používá k šifrování i dešifrování jedený klíč
- b) **Asymetrické šifrování** (kryptografie s veřejným klíčem) – skupina kryptografických metod, ve kterých se pro šifrování a dešifrování používají odlišné klíče a to tzv. soukromý a veřejný klíč

Pojmy:

- i) **S-Box** = booleovská funkce převádějící binární vektor na jiný binární vektor
- ii) **Jednosměrná funkce** = matematická funkce, kterou v jednom směru (přímém) lze snadno spočítat, zatímco v opačném směru (inverzní zobrazení) probíhají výpočty velmi obtížně.
- iii) **Hashovací funkce** = vstupem (jednosměrné) hashovací funkce je blok proměnné délky (zpráva) a výstupem je blok pevné délky (obvykle 128 či 160 bitů) – hash

#### Kerckhoffův princip:

Utajení a bezpečnost zašifrovaných dat nesmí záležet na utajení postupu, kterým se šifrují.

Naopak, vždy se musí předpokládat, že váš nepřítel zná šifru (algoritmus) do nejmenších detailů. Utajení musí spočívat pouze v klíči (hesle), které nezná nikdo jiný.

**Konfúze:** vyžaduje substituci, aby vztah mezi klíčem a šifrovým textem byl co nejsložitější.

**Difúze:** vyžaduje transformaci, která rozptyluje statistické vlastnosti otevřeného textu napříč šifrovým textem.

#### Symetrické algoritmy

Založeno na principu jednoho klíče, např. DES

Výhody: rychlosť

Nevýhody: distribuce klíče

Dělí se na:

- a) Proudové šifry – zpracovávají otevřený text po jednotlivých bitech (např. RC4, FISH)
- b) Blokové šifry – rozdělí otevřený text na bloky stejné velikosti a doplní vhodným způsobem, poslední blok na stejnou velikost; většinou 64 bit; např. AES, Blowfish, RC2

#### Asymetrické algoritmy

**Veřejný klíč** je všeobecně komukoliv dostupný, tímto klíčem lze pouze zašifrovat zprávu pro určitého uživatele.

**Soukromý klíč** má každý u sebe schovaný a určitým způsobem chráněný proti ukradení, (heslem, na čipové kartě, na magnetické kartě) tímto klíčem lze provádět odkódování přijatých zpráv -> Tedy, je-li zpráva pouze pro mě, tak pouze já svým tajným klíčem ji mohu odšifrovat

**Princip:** Založeno na jednocestných funkcích, což jsou operace, které lze snadno provést pouze v jednom směru: ze vstupu lze snadno spočítat výstup, z výstupu však je velmi obtížné nalézt vstup. Nejběžnějším příkladem je např. násobení: je velmi snadné vynásobit dvě i velmi velká čísla, avšak rozklad součinu na činitele (tzv. faktorizace) je velmi obtížný

**Příklad: RSA** – založen na úvaze „Je snadné vynásobit dvě dlouhá (100-místná) prvočísla, ale bez jejich znalosti je prakticky nemožné zpětně provést rozklad výsledku na původní prvočísla.“

## Hashovací funkce

Jsou to funkce, které umí udělat vzorek jakéhokoli souboru tak, aby byl závislý na všech bitech původního souboru.

$$h: D \rightarrow R, \text{ kde } |D| > |R|$$

Vstupem (jednosměrné) hashovací funkce je blok proměnné délky (zpráva) a výstupem je blok pevné délky (obvykle 128 či 160 bitů) – hash

**Kolize** = dvojic vstupních dat  $(x,y)$  takových, že  $h(x)=h(y)$

**Požadavky:** Odolnost vůči získání předlohy, získání jiné předlohy a nalezení kolize

Příklady hash funkcí: MD5 a SHA1

## Elektronický podpis

= otisk zprávy (z hashovací funkce) + soukromý klíč + systémový čas

**Certifikační autorita (CA)** – vydává certifikáty; nejvyšší je kořenová CA

## Certifikát

= veřejný klíč + údaje (jméno, kdo vydal, doba platnosti, ...) + elektronický podpis CA

## Hybridní kryptosystémy

kombinují výhody symetrické a asymetrické kryptografie, kde asymetrické šifrování se používá pro bezpečnou výměnu klíčů (zahájení a ověření bezpečné komunikace) a symetrické šifrování pro samotnou komunikaci

## 9. Definice konečných strojů, jejich alternativy a vzájemná ekvivalence.

KA jsou matematické modely výpočetní teorie používané k reprezentaci a analýze toku informací,  $A = (Q, \Sigma, \delta, q_0, F)$  obsahují:

- **konečnou** neprázdnou množinu stavů – stavový prostor  $Q$ ,
- konečná neprázdná množina vstupních symbolů – vstupní abeceda  $\Sigma$ ;
- přechodová funkce  $\delta$ , počáteční (iniciální) stav  $q_0$  a cílová (finální) množina – množina koncových stavů  $F$

KA se v každém svém okamžiku nachází v některém ze stavů konečné množiny stavů (v daném okamžiku se nachází právě v jednom stavu). Stav se může měnit na základě vnějšího podnětu z okolí (zpracováním konkrétního vstupního symbolu), které nejčastěji přicházejí v diskrétních časových okamžicích.

KA najdou využití např.:

- při návrhu sekvenčních logických obvodů
- při návrhu, specifikaci a implementaci protokolů pro komunikaci
- v překladačích programovacích jazyků
- při modelování architektury softwarových komponent
- v řídicích systémech logického typu
- vyhodnocování regulárních výrazů. Obvykle jsou tedy součástí lexikálního analyzátoru v překladačích.

Reprezentace konečného automatu:

1. Reprezentace tabulkou
2. Reprezentace stavovým diagramem
3. Reprezentace stavovým stromem (jen pro automaty, u nichž je každý stav dosažitelný z počátečního).

### Alternativy FA

- a) DFA – deterministický konečný
  - DFA má pevně definovanou množinu stavů a přechodovou funkci. Pro každý vstupní symbol a aktuální stav existuje právě jeden následující stav.
  - Použití: Běžně se používá v lexikální analýze a v situacích, kde je požadována přesná a jednoznačná reakce na vstup.
- b) NFA – nedeterministický konečný
  - NFA je podobný DFA, ale umožňuje více než jeden následující stav pro daný vstupní symbol a stav. To znamená, že existuje více potenciálních cest pro stejný vstup.
  - Nedeterminismus umožňuje, že pro daný vstup může existovat více možných konfigurací stavů.
  - Užitečný pro návrh a pochopení teoretických aspektů výpočetních procesů, ale méně praktický pro reálné implementace než DFA.
- c) NFA $\epsilon$  – nedeterministický konečný s prázdným přechodem
  - (= Zobecněný nedeterministický konečný) zde připouští tzv.  $\epsilon$ -přechody, tj. je zde povoleno přejít z jednoho stavu do jiného, aniž by se přečetl vstupní symbol.

### Ekvivalence

DFA-NFA-NFA $\epsilon$  (lze je na sebe převádět, lze ukázat převody)

### Převod NKA na DKA

K nedeterministickému konečnému automatu  $A = (Q, \Sigma, \delta, I, F)$  lze sestrojit ekvivalentní deterministický konečný automat  $A' = (Q', \Sigma, \delta', I, F')$ , tedy  $L(A) = L(A')$ . Pokud NKA obsahuje n stavů, pak ekvivalentní DKA bude obsahovat nejvýše  $2^n$  stavů.

## 10. Univerzální Turingův stroj a problém nezastavení pro Turingovy stroje, souvislost Turingových strojů a jazyků typu 0.

Turingův stroj = teoretický model pro univerzální rozhodovací stroj;

Turingův stroj je teoretický model počítače, který se používá pro modelování algoritmů.

TS je podobný konečnému automatu (ten je součástí hlavy), obsahuje kromě vlastního automatu také vstupní pásku se čtecí/zapisovací hlavou. Charakteristickými vlastnostmi jsou:

- páška je jednorozměrná (lineární) a nekonečně dlouhá,
- hlava se může po vstupní pásce pohybovat oběma směry,
- na aktuální pozici hlavy je možné zapisovat.

**Definice:** Turingův stroj je každá uspořádaná šestice  $T = (Q, \Sigma, P, \delta, q_0, F)$  kde:

$Q$  je konečná neprázdná množina (stavy)

$\Sigma$  je konečná neprázdná množina (vstupní abeceda)

$P$  je konečná neprázdná množina (pásková abeceda),  $\Sigma \subset P$ ,  $P$  obsahuje symbol prázdné pásky

$\delta$  je zobrazení (přechodová funkce)  $(Q-F) \times P \rightarrow Q \times P \times \{-1, 0, 1\}$

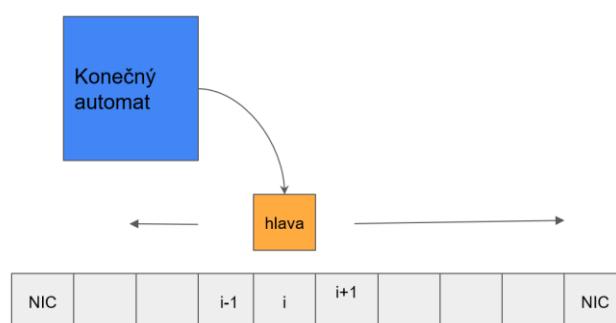
$q_0 \in Q$  (počáteční stav)

$F \subseteq Q$  je konečná množina (koncové stavy).

Univerzální rozhodovací stroj – na vstupu: slovo, na výstupu: patří/nepatří, užívá algoritmus (popis jazyka)

- Má dvě součásti:

- a) Pásová paměť rozdělená na buňky, zleva i zprava nekonečná
- b) Konečný automat řídí pohyb čtecí/zapisovací hlavy v paměti



- *Páska:* představuje vstup, kterým je konečný řetězec symbolů ze vstupní abecedy; řetězec z obou stran uzavřen prázdnou páskou (= symbol NIC, jež není prvkem  $\Sigma$ ); při výpočtu může být přepisována
- *Hlava:* pohybuje se po páscce o 1 krok vlevo i vpravo, a zapisuje i čte symboly na páscce; v každém kroku známe její pozici

Konečný automat – vstupem je symbol z pásky, má vnitřní stav a přechodovou funkci (jejíž součástí jsou navíc pokyny zápis/čtení a posun vpravo/vlevo)

## TS – Výpočet

- Před zahájením známe: Polohu hlavy, aktuální vstupní symbol a stav automatu
- Kroky:
  1. Změna stavu řídícího automatu dle přechodové funkce
  2. Přepis hodnoty na pásce (není povinné)
  3. Pohyb hlavy vlevo nebo vpravo
- Začátek výpočtu: Čtecí hlava je na nejlevějším vstupním symbolu, automat je v počátečním stavu
- Konec výpočtu:
  - a) Normální zastavení TS: Automat je v rozhodovacím (*konečném*) stavu
  - b) Abnormální zastavení TS: Automat nemá instrukci; pro danou konfiguraci není definován žádný přechod.
  - c) Nekonečný výpočet: Automat se cyklicky pohybuje mezi stavů, z nichž žádný není konečný.  
Výpočet se nikdy nezastaví.

Příklady programů: sčítání, odčítání, další algebraické operace...

Nedeterministický TS je stroj, který připouští, že dvě hrany mohou přepisovat stejný znak.

## Akceptování a zamítnutí slova TS

Slovo  $w \in \Sigma^*$  je umístěno vlevo na začátku pásky, hlava je na začátku slova, výpočet probíhá podle programu, začíná ve stavu START. Slovo  $w$  je akceptováno, když se TS dostane do stavu STOP. Slovo  $w$  je zamítnuto, když hlava je nad nejlevějším polem a je dán pokyn posunu vlevo nebo výpočet dosáhne stavu  $i$ , z něhož vycházejí hrany, po kterých se nemůže vydat.

## Problém nezastavení pro Turingovy stroje

Problém nezastavení se týká otázky, zda existuje algoritmus, který rozhodne, zda libovolný daný Turingův stroj na libovolném vstupu zastaví nebo bude pokračovat v nekonečném cyklu.

Význam: Alan Turing dokázal, že obecný algoritmus pro řešení problému nezastavení neexistuje.

## Jazyky typu 0 a TS

Jazyky typu 0: Jsou nejvíce obecnou třídou v Chomského hierarchii jazyků. Tyto jazyky jsou definovány pomocí gramatik bez omezení a mohou zahrnovat jakýkoliv možný jazyk, včetně těch, které jsou příliš složité pro analýzu pomocí méně mocných strojů, jako jsou konečné automaty nebo zásobníkové automaty.

Souvislost: Turingovy stroje jsou schopny rozpoznat nebo generovat jazyky typu 0. Každý jazyk, který je generován gramatikou typu 0, je Turingovsky rozhodnutelný nebo rozpoznatelný.

**Jazyky typu 0** = rekurzivně spočetné množiny = množiny akceptované vhodnými konečnými stroji se 2 zásobníky = množiny akceptované TS

## Programování TS pro jazyk $0^n1^n$

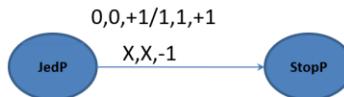
- Navrhněte TS, který nad abecedou  $\Sigma = \{0,1\}$  rozpozná řetězce ve tvaru  $0^n1^n$ , kde  $n$  je libovolné přirozené číslo větší než nula.
- Princip:
  - 1) Stroj nalezne levou nulu, nahradí ji značkou X.
  - 2) Posune se na konec řetězce, je-li zde jednička, nahradí ji značkou X.
  - 3) Posune se vpravo vedle poslední značky a situaci opakuje.

4) Je-li na pásce slovo z jazyka, bude po skončení páiska plná značek.

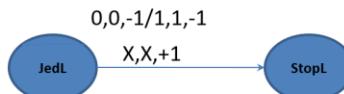
Čtené a zapisované symboly jsou  $P = \{B, X, 0, 1\}$

„Programování“ jazyka  $0^n1^n$ :

„Pohyb vpravo, dokud nenarazím na X a pak o krok zpět“:



„Pohyb vlevo, dokud nenarazím na X a pak o krok zpět“:



Pozn: Zápis ve tvaru A,B,+C: A čtu z pásky, B píšu na pásku, C posun

- Stroj (automat) prochází stavy až do požadovaného (OK) stavu

## 11. Totální a parciální rozhodnutelnost, problém zastavení pro Turingovy stroje, Postův problém přiřazení (korespondence) a problém zániku matic typu 3x3.

**Totální rozhodnutelnost:** Jazyk nebo problém je "totálně rozhodnutelný" (nebo rekurzivní), pokud existuje Turingův stroj, který pro každý vstup vždy zastaví a správně určí, zda vstup patří do jazyka/problému (tj. striktně (ne)akceptuje) = rozhoduje (nedochází k cyklu)

**Parciální rozhodnutelnost:** Jazyk nebo problém je "parciálně rozhodnutelný" (nebo rekurzivně vyčíslitelný), pokud existuje Turingův stroj, který pro vstupy patřící do jazyka/problému vždy zastaví a řekne "ano" (tj. akceptuje), ale pro vstupy mimo jazyk/problém buď zastaví a řekne "ne" (tj. neakceptuje), nebo bude pokračovat v nekonečném výpočtu = pro správné rozhoduje, pro nesprávné může skončit v cyklu

### TS jako algoritmy

Řešení tříd problémů typu ANO/NE – úlohy, na které lze jednoznačně odpovědět ANO/NE

**def.:** Řekneme, že daná třída problémů typu ANO/NE je **totálně rozhodnutelná** (řešitelná), jestliže existuje nějaký algoritmus (tj. TS), který ke každému problému dané třídy stanoví řešení, vždy zastaví a vydá odpověď ANO nebo NE, přičemž ANO znamená, že zastavení je typu akceptováno, NE – zamítnuto.

Říkáme, že daná třída typu ANO/NE je nerozhodnutelná (neřešitelná), jestliže neexistuje algoritmus (TS) se shora definujícími vlastnostmi.

**def.:** Třída problémů typu ANO/NE se nazývá **parciálně rozhodnutelná** (částečně řešitelná), jestliže existuje algoritmus (TS) takový, že pro každý problém z uvedené třídy platí:

1. je-li odpověď na daný problém typu ANO, pak algoritmus zastaví přechodem do příkazu akceptováno
2. je-li odpověď NE, pak algoritmus buď přejde do příkazu zamítnuto, nebo nikdy neskončí svou činnost

Třída, která je totálně rozhodnutelná, je i parciálně rozhodnutelná

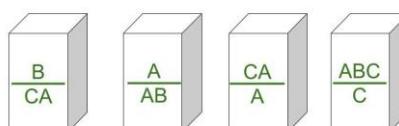
### Problém zastavení pro TS

**def.:** Problém zastavení je otázka, zda existuje algoritmus, který rozhodne, zda libovolný Turingův stroj na libovolném vstupu zastaví či nikoliv.

**věta: (Turing)** problém zastavení je nerozhodnutelný; neexistuje žádný algoritmus, který by mohl pro všechny možné páry strojů a vstupů určit, zda stroj na daném vstupu zastaví.

### Postův problém přiřazení (korespondence)

Post Correspondence Problem (PCP) je rozhodovací problém v teorii formálních jazyků a automatů. V problému jde o to, zda lze vybrat karty s řetězci na horní a dolní straně (jako díly domina) tak, aby jejich konkatenace (zřetězení) byla shodná. Tento problém je známý pro svou nerozhodnutelnost, což znamená, že neexistuje obecný algoritmus pro rozhodnutí, zda existuje řešení.



## Problém zániku matic typu 3x3

Problém zániku matic typu 3x3 je rozhodovací problém, který se zabývá otázkou, zda je možné dosáhnout nulové matice pomocí konečného počtu operací na dané matici 3x3 podle určitých pravidel.

Mějme čtvercovou matici  $A$  o rozměrech 3x3 s reálnými nebo komplexními prvky. Ptáme se, zda existuje **matici  $B$ , která nemá inverzi**, taková, že součin  $AB$  je nulová matice.

Matice  $B$  se nazývá "matice zániku" pro matici  $A$  v případě, že  $AB$  je nulová matice, ale  $B$  není nulová matice.

Tento problém má zajímavou spojitost s pojmy jako jádro a obraz lineárních transformací, ačkoliv je výzvou najít explicitní konstrukci matice  $B$ , která vyhovuje daným podmínkám.

Problém zániku matic typu 3x3 se také může spojovat s některými aspekty teorie grup, algebraických struktur a teorie reprezentací, což z něj činí matematicky zajímavý a multidisciplinární problém. Jedná se o otevřený problém, ačkoliv bylo dosaženo některých pokroků v jeho řešení.

## 12. Deterministické a nedeterministické konečné automaty, význam, ekvivalence a ukázky konkrétních návrhů.

**Deterministický** = pro daný vstupní symbol a aktuální stav existuje právě jeden přechod do nového stavu

**Nedeterministický** = pro daný vstupní symbol a aktuální stav existuje více možných přechodů do nového stavu, nebo i žádný přechod

**Def.:** Konečným automatem nad abecedou  $\Sigma$  rozumíme uspořádanou pětici

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

$Q$  je konečná neprázdná množina stavů – stavový prostor;

$\Sigma$  (sigma) je konečná neprázdná množina vstupních symbolů – vstupní abeceda;

$\delta$  (delta) je zobrazení nazývané přechodová funkce:

- a)  $\delta: Q \times \Sigma \rightarrow Q$       **Deterministický automat**
- b)  $\delta: Q \times \Sigma \rightarrow P(Q)$       **Nedeterministický automat**

$P(Q)$  ... (potenční množina) je taková množina, která obsahuje všechny podmnožiny množiny  $Q$ ; př.:

$$Q = \{1, 2\}, \text{ potom } P(Q) = \{\emptyset, \{1\}, \{2\}, \{1, 2\}\}$$

$q_0 \in Q$  je počáteční (iniciální) stav;

$F \subset Q$  je cílová (finální) množina – množina koncových stavů.

### Ekvivalence k DFA

Pro každý DFA existuje ekvivalentní NFA, a naopak. Díky ekvivalenci je možné automaty mezi sebou převádět.

Na DFA lze nahlížet jako na speciální druh NFA, ve kterém má přechodová funkce pro každý stav a symbol právě jeden stav. Je tedy jasné, že každý formální jazyk, který může být rozpoznán DFA, může být rozpoznán NFA.

DFA lze zkonstruovat pomocí konstrukce potence množiny.

### Význam

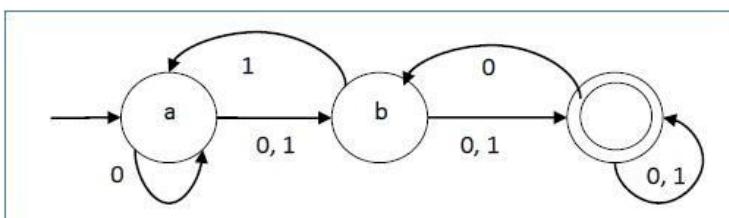
V praxi je ekvivalence důležitá pro přeměnu snáze sestavitelných NFA na efektivněji spustitelné DFA. Pokud má však NFA  $n$  stavů, výsledná DFA může mít až  $2^n$  stavů, což někdy činí konstrukci pro velké NFA nepraktickou.

Oba se užívají v překladačích a regulárních výrazech.

## Ukázky konkrétních návrhů

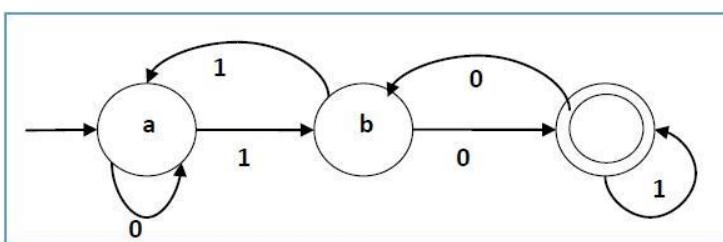
NKA:

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



DKA:

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



## 13. Regulární výrazy a rovnice, jejich význam a způsoby řešení.

### Regulární výrazy (RV)

Představují obvyklou notaci (způsob zápisu) regulárních množin.

Regulární výraz nad abecedou  $\Sigma$  definujeme takto:

$\emptyset$  je regulární výraz označující regulární množinu  $\emptyset$ .

$\epsilon$  je regulární výraz označující regulární množinu  $\{\epsilon\}$

a je regulární výraz označující regulární množinu  $\{a\}$  po všechna  $a \in \Sigma$

Jsou-li  $p$  a  $q$  regulární výrazy označující regulární množiny  $P$  a  $Q$  pak:

$(p + q)$  je regulární výraz označující regulární množinu  $P \cup Q$

$(pq)$  je regulární výraz označující regulární množinu  $P \cdot Q$

$(p^*)$  je regulární výraz označující regulární množinu  $P^*$

Regulárními výrazy jsou právě ty výrazy, které lze získat aplikací předchozího

**Užití:** Reg. výrazy využívají při hledání a filtrace textu, jako validátory vstupu a při lexikálních analýzách (součást překladačů).

### Rovnice nad regulárními výrazy

Rovnice, jejichž složky jsou koeficienty a neznámé reprezentující dané a hledané regulární výrazy. Obecně řešíme, jaké jazyky splňují regulární výrazy.

**Př. Řešením rovnice:**  $X = aX + b$  je regulární výraz  $X = a^*b$ . Důkaz:

1.  $a^*b = a(a^*b) + b$
2.  $a^*b = a^+b + b$
3.  $a^*b = (a^+ + \epsilon)b$
4.  $a^*b = a^*b$

Soustava rovnic nad RV je ve standardním tvaru vzhledem k neznámým  $\Delta = \{X_1, X_2, \dots, X_n\}$  má-li tvar  $\Lambda X_i = \alpha_{i0} + \alpha_{i1}X_1 + \alpha_{i2}X_2 + \dots + \alpha_{in}X_n$  pro  $1 \leq i \leq n$ . Je-li soustava rovnic ve standardním tvaru pak existuje její minimální pevný bod (řešení) a algoritmus jeho nalezení.

Rovnice slouží k popisu vztahů RV a manipulaci s nimi (zjednodušení, transformaci).

### Význam

Regulární přechodový graf je zobecněný KA, který obsahuje množinu počátečních stavů a regulární výrazy na hranách. Každý reg. přechodový graf je možné převést na reg. přechodový graf s jediným přechodem na kterém je hledaný RV.

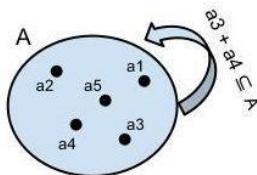
**Užití:** Regulární rovnice jsou převeditelné na deterministické konečné automaty a naopak. To znamená, že mohou být použity pro popis chování konečných automatů a naopak, což usnadňuje analýzu a návrh v oblasti teorie automatů.

## 14. Standardní uzávěrové vlastnosti na třídě regulárních jazyků, jejich využití.

Třída regulárních jazyků = množina všech regulárních jazyků

Def: Uzavřenost na operaci

(Lidsky) Výsledek operace mezi prvky z množiny je také součást množiny.



Regulární jazyky jsou uzavřeny na různé operace – tedy pokud jsou jazyky K a L regulární, je regulární i výsledek následujících operací:

- Konkatenace (Spojení, zřetězení):
  - Popis: Konkatenace dvou regulárních jazyků A a B, označovaná jako A.B, vytváří jazyk, který obsahuje všechny možné spojení řetězců z A následované řetězci z B.
  - Využití: Používá se v regulárních výrazech, a při analýze a generování jazyků.
- Unie (Alternativa, sjednocení):
  - Popis: Unie regulárních jazyků A a B, označovaná jako A U B, vytváří jazyk, který obsahuje všechny řetězce, které patří buď do A nebo do B.
  - Využití: Důležitá pro vytváření flexibilních vzorů v regulárních výrazech.
- Iterace (Kleeneho hvězda):
  - Popis: Iterace jazyka A, označovaná jako A\*, vytváří jazyk obsahující řetězce vytvořené opakovaným spojením nula nebo více kopii řetězců z A.
  - Využití: Klíčová pro reprezentaci opakujících se vzorů v regulárních výrazech.
- Průnik:
  - Popis: Průnik dvou regulárních jazyků A a B, označovaný jako A ∩ B, obsahuje ty řetězce, které jsou součástí obou jazyků A a B.
  - Využití: Umožňuje vytvářet složitější podmínky a filtry ve vyhledávání a analýze textu.
- Komplement (Doplňek):
  - Popis: Komplement regulárního jazyka A, označovaný jako ~A, obsahuje všechny řetězce, které nejsou v jazyku A.
  - Využití: Užitečný ve vyhledávání a při definování podmínek, které nesmí být splněny.
- Reverze:
  - Popis: Reverze jazyka A vytváří jazyk, někdy značený A<sup>R</sup>, který obsahuje všechny řetězce z A obrácené.
  - Využití: Může být použito v teoretických studiích jazyků a v některých aplikacích zpracování textu.

Pozn.: Každý regulární jazyk můžeme popsat konečným automatem a definován gramatikou typu 0.

## Využití

Znalost toho, že určité operace s regulárními jazyky vedou k jiným regulárním jazykům, umožňuje vývoj účinných algoritmů pro úlohy, jako je porovnávání vzorů, zpracování textu a lexikální analýza.

Uzavřenost vlastností je zásadní pro určení, zda je jazyk regulární. Pokud lze jazyk získat aplikací uzavíracích operací (zřetězení, sjednocení, doplněk atd.) na známé regulární jazyky, pak je také regulární.

V praxi hraje roli při návrhu překladačů (lexikální analýze, ...), optimalizaci (reg. jazyk -> optimalizace algoritmů) a dále třeba kombinace jazyků (např. zřetězením).

## 15. Chomského hierarchie gramatik a jazyků, význam, návrh gramatiky pro jednoduchý konečný automat

### Úvod – opakování

Abeceda  $\Sigma$  je množina znaků, z nich skládáme řetězce – slova nad touto abecedou (slovo může být i prázdné).

Jazyk nad abecedou  $\Sigma$  je libovolná množina slov nad  $\Sigma$ . Množinu všech slov nad abecedou  $\Sigma$  značíme  $\Sigma^*$ , množinu všech neprázdných slov  $\Sigma^+$ .

Gramatika je popis jazyka pomocí pravidel, podle kterých se vytvářejí všechna slova daného jazyka.

Gramatiky slouží pro charakteristiku formálních jazyků. Jinými prostředky pro charakteristiku jazyků jsou regulérní výrazy nebo konečné automaty (deterministické i nedeterministické).

**Def:** Gramatikou nazýváme každou čtveřici  $G = (N, \Sigma, P, S)$  kde:

$N$  – neprázdná konečná množina neterminálních symbolů

$\Sigma$  – konečná množina terminálních symbolů takových, že  $N \cap \Sigma = \emptyset$

$P$  – množina pravidel  $P \subseteq V * NV * \dots V *$ , kde  $V$  je  $N \cup \Sigma$

$S$  – počáteční neterminál,  $S \in N$

Pokud je pro každé slovo nejvýše jeden postup generování, gramatika je jednoznačná.

Gramatiky G1 a G2 nazveme jazykově ekvivalentní, právě když generují tentýž jazyk, tj.  $L(G1) = L(G2)$ .

**Generativní gramatikou** rozumíme uspořádanou čtveřici  $G = (\Pi, \Sigma, S, P)$ , kde  $\Pi$  a  $\Sigma$  jsou konečné abecedy a  $\Pi \cap \Sigma = \emptyset$ ,  $S \in \Pi$ ,  $P$  je konečná množina přepisovacích pravidel tvaru  $\alpha \rightarrow \beta$ , kde  $\alpha, \beta \in (\Pi \cup \Sigma)^*$  a  $\alpha$  obsahuje alespoň jeden symbol z  $\Pi$ . Abeceda  $\Pi$  se nazývá množinou neterminálů (proměnných). Abeceda  $\Sigma$  se nazývá množinou terminálů a  $S$  je počáteční symbol.

### Chomského hierarchie gramatik a jazyků

#### Typ 0 (Rekurzivně vyčíslitelné jazyky):

Gramatika: Neomezená gramatika

Jazyky: Všechny jazyky, které může rozpoznat Turingův stroj

Význam: Tyto jazyky jsou nejobecnější a nejméně omezené.

#### Typ 1 (Kontextově závislé jazyky – kontextové):

Jazyky: Jazyky, které vyžadují kontext pro definování produkčních pravidel

Význam: Tento typ jazyka umožňuje popsát složitější struktury, jako jsou některé přirozené jazyky.

Gramatika  $G = (\Pi, \Sigma, S, P)$  se nazývá gramatika typu 1 nebo kontextová gramatika, jestliže přepisovací pravidla z  $P$  jsou tvaru  $\alpha X \beta \rightarrow \alpha \gamma \beta$ , kde  $\alpha, \beta \in (\Pi \cup \Sigma)^*$ ,  $X \in \Pi$  a  $\gamma \in (\Pi \cup \Sigma)^+$

(tj.  $|\gamma| \geq 1$ ) (tj.  $X$  se přepíše na  $\gamma$ , kde  $\gamma \neq \epsilon$  právě tehdy, je-li obklopeno  $\alpha, \beta$ , jinak ne.)

Výjimkou může být pravidlo  $S \rightarrow e$ , jehož výskyt je ale spojen s požadavkem, že  $S$  se nesmí objevit na pravé straně žádného přepisovacího pravidla z  $P$ .

Jazyk typu 1 resp. kontextový jazyk je jazyk generovaný nějakou kontextovou gramatikou.

## Typ 2 (Kontextově nezávislé jazyky – bezkontextové):

Jazyky: Jazyky, kde produkční pravidla nezávisí na kontextu

Význam: Tyto jazyky jsou klíčové pro syntaxi programovacích jazyků a jsou analyzovatelné pomocí zásobníkových automatů.

Gramatika typu 2 neboli bezkontextová gramatika je každá gramatika  $G = (\Pi, \Sigma, S, P)$ , kde  $P$  obsahuje pouze pravidla typu  $X \rightarrow \gamma$ , kde  $X \in \Pi$  a  $\gamma \in (\Pi \cup \Sigma)^*$  (neterminál se přepíše na libovolné slovo, třeba i prázdné).

Jazyk se nazývá bezkontextový nebo typu 2, lze-li jej generovat bezkontextovou gramatikou.

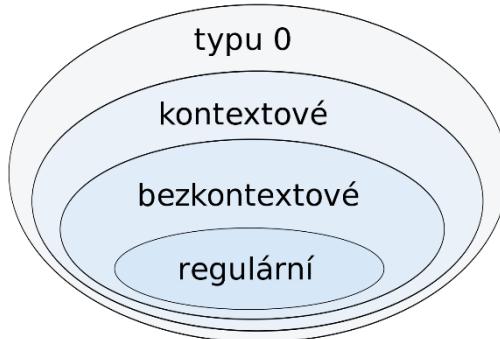
## Typ 3 (Regulární jazyky):

Jazyky: Nejsnadněji analyzovatelné jazyky, které odpovídají regulárním výrazům

Význam: Jsou ideální pro návrh vyhledávačů, lexikální analýzu a pro jednoduché vzory v textových řetězcích. Rozpoznávané konečným automatem.

Gramatika  $G = (\Pi, \Sigma, S, P)$  je typu 3 neboli regulární (nebo pravá lineární), je-li každé pravidlo z  $P$  buď tvaru  $X \rightarrow wY$ , nebo tvaru  $X \rightarrow w$ , kde  $X, Y \in \Pi$ ,  $w \in \Sigma^*$ .

Jazyk je regulární, neboli typu 3, lze-li jej generovat nějakou regulární gramatikou.



## Význam

Grammar	Languages	Recognizing Automaton	Production rules (constraints)*	Examples <sup>[5][6]</sup>
Type-3	Regular	Finite state automaton	$A \rightarrow a$ and $A \rightarrow aB$	$L = \{a^n   n \geq 0\}$
Type-2	Context-free	Non-deterministic pushdown automaton	$A \rightarrow \alpha$	$L = \{a^n b^n   n > 0\}$
Type-1	Context-sensitive	Linear-bounded non-deterministic Turing machine	$\alpha A \beta \rightarrow \alpha \gamma \beta$	$L = \{a^n b^n c^n   n > 0\}$
Type-0	Recursively enumerable	Turing machine	$\gamma \rightarrow \alpha$ ( $\gamma$ non-empty)	$L = \{w   w \text{ describes a terminating Turing machine}\}$

\* Meaning of symbols:

- $a$  = terminal
- $A, B$  = non-terminal
- $\alpha, \beta, \gamma$  = string of terminals and/or non-terminals

## Návrh gramatiky pro jednoduchý konečný automat

Př.: Pro následující automat vytvořte

Řešení:

Abecedu tvoří 0, 1

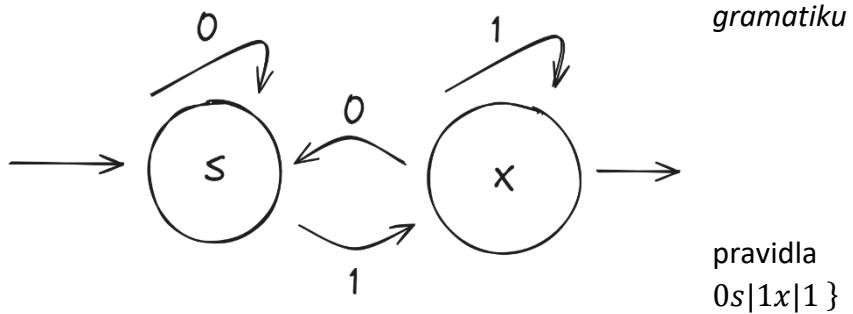
Počáteční neterminál je s

Konečný neterminál x

Pro sestavení gramatiky je třeba určit

přepisu  $P = \{s \rightarrow 0s|1x; x \rightarrow\}$

Výsledek:  $G = (\Pi, \Sigma, S, P) = (\{s, x\}, \{0, 1\}, \{s\}, \{s \rightarrow 0s|1x; x \rightarrow 0s|1x|1\})$



Při návrhu gramatiky pro jednoduchý konečný automat se typicky zaměřujeme na regulární gramatiku (Typ 3).

Příklad: Předpokládejme, že chceme navrhnout gramatiku pro jednoduchý automat, který akceptuje řetězce obsahující sudý počet 'a'.

Stavy: Můžeme mít stavy Q0 (počáteční stav, sudý počet 'a') a Q1 (lichý počet 'a').

Přechody:

$Q0 \rightarrow 'a' Q1$

$Q1 \rightarrow 'a' Q0$

Koncový stav: Q0 (sudý počet 'a' znamená přijetí řetězce).

Gramatika: Může být definována pomocí pravidel, např.  $A \rightarrow aB$ ,  $B \rightarrow aA$ , kde A odpovídá stavu Q0 a B stavu Q1.

## **16. Konstrukce překladače. Frontend a backend. Lexikální analýza.**

### **Syntaktická analýza. Sémantická kontrola. Intermediální jazyky. Generování kódu. Optimalizace kódu. Křížový překlad.**

**Překladač (compiler)** je program, který převádí kód psaný v programovacím jazyce vyšší úrovně, který je srozumitelný pro programátory, na druhý kód nižší úrovně, který vyžaduje počítač (machine code).

Kód může být také napsán jednou a následně přeložen různými počítači.

Důležité je, že překladač zachová sémantický význam (smysl) a také se pokusí odhalit a nahlásit chyby programátora.

Typy překladačů:

- a) Přímý překlad – překlad probíhá přímo pro užívaný počítač
- b) Křížový překlad – vytváří překlad pro nějaký cílový počítač, využívá se, pokud je efektivnější dělat překlad mimo cílový stroj a na něm už jen implementovat přeložený kód

### **Základní konstrukce překladače (frontend, backend a jejich součásti):**

Konstrukce překladače se dělí do následujících fází, první tři jsou **Frontend**:

**1) Lexikální analýza** – čtení a analyzování textu. Text je přečten a rozdělen do tokenů, které nalezneme symbolům v programovacím jazyce, např. jméno proměnné, klíčové slovo nebo čísla **2) Syntaktická analýza** – (= parsing) vezme tokeny z přechozí fáze a poskládá je do syntax tree, který odráží strukturu programu.

**3) Typová kontrola (sémantická)** – zde proběhne analýza syntax tree, kontroluje, zda nedochází k porušení pravidel, např. užitím nedeklerované proměnné nebo chybné užití datového typu

Následující části patří do **Backendu**:

**4) Generování intermediate kódu (mezikódu)** – program je přeložen do jednoduchého strojově nezávislého intermediate jazyka.

**5) Alokace registrů** – názvy symbolických proměnných používané v mezikódu jsou přeloženy na čísla, z nichž každé odpovídá registru v cílovém strojovém kódu.

**6) Generování strojového jazyka** – Intermediate jazyk je přeložen do jazyka assembleru pro konkrétní architekturu stroje.

**7) Assembly a linking** – Kód assembleru je přeložen do binární reprezentace a jsou určeny adresy proměnných, funkcí atd.

Pojmy:

**Linker** – vytváří spustitelný kód z produktů překladu a spojuje dohromady jednotlivé části

**Loader** (zavaděč) – přiděluje paměť (součást OS)

### **Lexikální analýza**

Dochází během ní k práci s *tokeny* (lexem), což jsou terminální symboly gramatiky. Za tokeny považujeme identifikátory, hodnoty (num, string), klíčová slova, funkce, operátory atd. Tokeny, které odpovídají nějakým vzorcům jsou nahrazeny identifikátory užitím regulárních výrazů. Regulární výrazy v gramatice definují vzorce, celé jazyky a výsledně konečné automaty. Právě vzorce z tokenů je třeba ve zdrojovém kódu najít a nahradit.

**Výstupem lexikální analýzy je:**

- a) Proud tokenů
- b) První verze tabulky symbolů
- c) Chybová hlášení

**Tabulka** symbolů spojuje identifikátory s tokeny, konkrétně: Názvy proměnných a konstant, názvy procedur a funkcí, překladačem generované dočasné symboly.

Důležité jsou datové typy, adresy funkcí, argumenty funkcí a vše ostatní deklarované v gramatice.

**Nástroje pro lexikální analýzu:** Lex/Flex (C++, Unix) a ANTLR (Java)

## Syntaktická analýza

V této části dochází k parsingu tokenů z lexikální analýzy a jejich rekombinaci do syntax tree.

Listy této analýzy jsou tokeny a pokud jsou přečteny získáme původní vstup. Díky listům a uzlům stromu dostaneme strukturu.

Zároveň syntaktická analýza odmítne nevalidní vstup a vrací syntax error.

K syntaktické analýze se využívá zásobníkový automat, ten slouží k záznamu odvození – přepis symbolů dle typu analýzy.

**Rozkladová tabulka** – obsahuje pravidla gramatiky přepsaná do řeči zásobníkového automatu. Liší se pro analýzu *shora* (slovo -> symbol) a *zdola* (symbol -> slovo).

**Při analýze shora:**

Je nejpoužívanější LL(1) analýza, kdy je třeba znát vždy jeden následující symbol. Obecně LL znamená, že jdeme přes levé derivace. LL(1) má dvě pravidla:

- Vlastnost First-First: Pokud  $Y \rightarrow X_1 | X_2 | \dots | X_n$ , potom musí platit:  $\text{First}(X_i) \cap \text{First}(X_j) = \emptyset$  pro všechna  $i \neq j$
- Vlastnost First-Follow (někdy též FFL): Lze-li z  $X_i$  odvordinat prázdný symbol, potom:  $\text{First}(X_i) \cap \text{Follow}(Y) = \emptyset$

**Automat** má vždy jeden stav, jeho abeceda se shoduje s abecedou gramatiky a tvoří ji terminály i neterminály.

Rozkladová tabulka určuje, podle kterého pravidla se provádí expanze jednotlivých neterminálů, když chceme získat konkrétní symboly. Řádky tabulky tvoří vše, co může obsahovat zásobník: neterminály, terminály, symbol konce zásobníku (#). Sloupce tvoří to, co může být vstup: terminály a symbol konce řetězce (\$).

Rozkladová tabulka je zobrazení se čtyřmi funkčními hodnotami:

- Expand i: Na vrcholu zásobníku je neterminál A, který je přepsán na pravou stranu pravidla, jež má A na levé straně.
- Pop: Na vrcholu zásobníku a na vstupu je stejný terminál, ten je z vrcholu zásobníku odstraněn a čte se další znak.
- Accept: Konec rozpoznávání s přijetím; prázdný zásobník.
- Error: Chyba při rozpoznávání, vstupní řetězec do jazyka nepatří.

**Analýza zdola** – LR(k), je Left-to-right: vstup čteme zleva. Je to zobrazení s jinými funkčními hodnotami (push, reduce, accept, error).

## Sémantická analýza

V této fázi je cílem typová kontrola. Určení rozsahu operací v rámci typů, analýza, zda jsou proměnné dostupné v určitou chvíli a nalezení neshod

**Vstup** je syntax tree a tabulka symbolů (pomocí tabulky se určí binding, tedy propojení symbolů a významu)

**Binding** je dvojího typu: Statical (lexical) – analýza při překladu a Dynamical – za běhu (interpretované jazyky, JIT).

S tabulkou symbolů je pracováno pomocí **operací**:

- empty table – vytvoření prázdné tabulky
- bind – nová dvojice jméno/informace, je-li již symbol v tabulce, nové propojení překrývá staré
- look up – vyhledání symbolu v tabulce, případně vrátí „not found“

- d) enter – vložení nové reference, tvorba kopie
- e) exit – ukončení daného rozsahu platnosti (maže kopii); návrat ke stavu původní tabulky

**Tabulky symbolů** jsou dvojího typu: Persistent – změny ji nemohou zničit, při modifikaci tvorba kopie (původní se zachová) a Imperative – jen jedna verze, změny obsahují záznamy o akcích pro návrat (zásobník).

Při užití zásobníku se pracuje vždy s jeho vrcholem

## Intermediální jazyky

Užívají **intermediální kód**

**Intermediální kód** je přechodem mezi frontendem a backendem (middle end). Má standardizovanou syntaxi a dokumentaci. Můžeme v něm sjednotit více high-level jazyků do jednoho intermediate.

Jeho cílem je zachovat strukturu a sémantický význam + tvorba univerzálnějšího kódu (přenosného mezi více stroji) vhodného pro generování strojového kódu.

Tvorbou mezikódu dochází ke výraznému zpomalení celého procesu.

Máme dva přístupy:

- a) Stromovou strukturu – často pro optimalizaci kódu
  - např. AST (Abstract Syntax Tree) – popsáný standard pro GCC (kolekce compilerů)
  - optimalizace odkazy ve stromu na již provedené výpočty (větve)
- b) Tříadresní kód (Lin. reprezentace) – podobný assambleru, možno nahlížet na něj jako na kód VM
  - Operace přepsány pro registry

**Příklad:** MSIL (Microsoft Intermediate Language)

Některá omezení rychlosti lze eliminovat převodem mezikódu do strojového kódu bezprostředně před nebo během provádění programu. Tato hybridní forma se nazývá komplikace **just-in-time** a často se používá pro spouštění mezikódu pro Javu

## Generování kódu

Fáze procesu komplikace, kde se překládá zdrojový kód do strojového kódu (případně intermediálního kódu), tedy proces, který převádí abstraktní reprezentaci programu na podobu, kterou může počítač spustit a vykonávat

Během této fáze jsou aplikována různá pravidla a optimalizace, které mají za cíl vytvořit efektivní a spustitelný kód.

Realizováno různými způsoby v závislosti na cílovém prostředí, cílové platformě a použitém komplikátoru či interpretru

## Optimalizace kódu

= úprava kódu, aby bylo splněno hned několik cílů:

- ✓ Rychlejší výkon kódu za běhu
- ✓ Menší nároky na paměť za běhu
- ✓ Menší stopa v paměti pro celý kód
- ✓ Menší spotřeba energie za běhu

**Priorita cílů** je zvolena pro konkrétní případ – navzájem si protiřečí

Optimalizace lokální (sekce kódu) a celkovou (kódu jako celku)

**Zrychlení kódu za běhu** – toho docílíme:

- odstraněním zpomalujících elementů kódu (skoky, volání funkcí, cykly, nepotřebné instrukce a *duplicity, invarianty cyklu*), omezení skoků = *inlining*
- kontrolou datového toku minimalizací přesunů mezi pamětí a registry, a úpravou pořadí operací

- vhodným užíváním konstant, vektorizace kódu (nutná podpora HW i překladače)

### **Menší nároky na paměť za běhu**

- závisí i na HW: souvisí s cache, predikcí kódu, využitím registrů
- vyvarovat se opakovanému výpočtu hodnot (neukládám mezi výsledky apod.)
- optimalizace uložení polí

**Malá paměťová stopa** – odstraněním všeho nepotřebného a duplicit, dále pak výpočet všech dílčích výrazů. Často vede k delšímu trvání kódu – nutí nás vyněchat inlining, ponechat funkce a skoky

**Spotřeba energie** – vázáno na architekturu, upravuje se pipeline, každá instrukce “něco stojí” lze dosáhnout:

- minimalizací přenosu dat mezi pamětí a procesorem
- vhodným využitím cache
- laděním pořadí instrukcí
- prací s využitím (napájením) sběrnic, např. sběrnic k paměťovým čipům apod.

Je třeba brát v úvahu matematickou náročnost – v praxi užívání speciálních frameworků a knihoven s předpřipravenými funkcemi.

**Optimalizace práce s úložištěm** – analýza toku, minimalizace přenosu dat mezi pamětí a úložištěm

**Vkládání funkcí** – zrychlení minimalizací skoků a ukládání kontextu

**Optimalizace smyček** – minimalizujeme “zbytečné”, zrychlujeme průchod

### **Křížový překlad**

Vytváří překlad pro nějaký cílový počítač

Využívá se, pokud je efektivnější dělat překlad mimo cílový stroj a na něm už jen implementovat přeložený kód

## 17. Funkcionální programování, čisté funkce, funkce jako hodnota 1. třídy.

### Náhrada cyklu rekurzí.

#### Funkcionální programování

= výpočet je vyhodnocením matematické funkce (definuje vztah mezi vstupními a výstupními hodnotami)

Vše je výraz, vyhodnocením vznikne hodnota

Vyhýbá se stavovým datům (proměnným) – průběžné hodnoty se předávají v parametrech a výsledcích funkcí; Časté použití rekurze a funkcí vyššího řádu

Např.: Lisp, Scheme, Haskell

#### Čisté funkce (výrazy)

Čisté funkce (nebo výrazy) nemají žádné vedlejší účinky (na paměť nebo vstup/výstup).

**Vedlejší účinky** funkce = pokud má nějaký pozorovatelný účinek jiný než primární účinek čtení hodnoty jejich argumentů a vrácení hodnoty vyvolávajícímu operaci (např. vyhození erroru, exceptions nebo čtení nelokální proměnná)

**Výhody** čistých funkcí (hl. optimalizační):

- Pokud není výsledek čistého výrazu použit, lze ho odstranit, aniž by to ovlivnilo ostatní výrazy
- Volání čisté funkce znova se stejnými argumenty vrátí stejný výsledek (optimalizace cachování, jako je memoizace)
- Pokud neexistuje závislost dat mezi dvěma čistými výrazy, jejich pořadí lze obrátit nebo je lze provádět paralelně a nemohou se vzájemně ovlivnit
- Pokud celý jazyk nepovoluje vedlejší účinky, pak má kompilátor svobodu k přeupořádání nebo kombinaci vyhodnocování výrazů v programu

#### Funkce jako hodnota 1. třídy

Jazyk má **first-class funkce**, pokud jazyk podporuje:

- předávání funkcí jako argumentů jiným funkcím
- umožňuje je vracet jako hodnoty z jiných funkcí
- umožňuje je přiřazovat proměnným nebo je ukládat do datových struktur

Nutnost pro funkcionálního programování, ve kterém je používání funkcí vyššího řádu (např. map) běžnou praxí

### Náhrada cyklu rekurzí

(Iterace (looping) ve funkcionálních jazycích se obvykle provádí pomocí rekurze)

**Rekurzivní funkce** se samy vyvolávají a nechají operaci opakovat, dokud nedosáhne základního případu (base case)

Rekurze vyžaduje udržování zásobníku – může způsobit, že použití rekurze namísto imperativních smyček bude neúměrně náročné

Tomu se lze vyhnout užitím **tail** rekurze, ta je pak adekvátně optimalizována kompilátorem

## **18. Využití analytického modelování v objektově orientovaném programování, objektové paradigma, úrovně abstrakce, diagramy UML (uveďte příklad diagramu tříd, sekvencí, stavů, ...).**

**Pozn.:**

Analytické modelování = Technika, která používá matematické modely k předpovídání chování služeb IT nebo jiných konfiguračních položek

*UML (Unified modeling language)* = univerzální vizuální modelovací jazyk, který má poskytovat standardní způsob vizualizace návrhu systému; poskytuje standardní notaci pro tři skupiny diagramů: diagramy chování, diagramy interakcí a diagramy struktury

### **Object-Oriented Analysis and Design (OOAD)**

= metodologie softwarového inženýrství, která využívá objektově orientované principy k modelování a navrhování složitých systémů

#### **Aspekty OOAD:**

- **Objektově orientované programování:** zahrnuje modelování objektů reálného světa jako softwarových objektů s vlastnostmi a metodami, které reprezentují chování těchto objektů. OOAD používá tento přístup k návrhu a implementaci softwarových systémů.
- **Návrhové vzory:** Návrhové vzory jsou opakováně použitelná řešení běžných problémů v návrhu softwaru. OOAD používá návrhové vzory, které pomáhají vývojářům vytvářet lépe udržovatelné a efektivnější softwarové systémy.
- **Diagramy UML:** Unified Modeling Language (UML) je standardizovaný zápis pro vytváření diagramů, které představují různé aspekty softwarového systému. OOAD používá diagramy UML k reprezentaci různých součástí a interakcí softwarového systému.
- **Případy užití:** představují způsob, jak popsat různé způsoby interakce uživatelů se softwarovým systémem. OOAD používá případy použití, které pomáhají vývojářům porozumět požadavkům systému a navrhovat softwarové systémy, které tyto požadavky splňují

### **Objektové paradigma**

**Objektově orientované programování (OOP)** je programovací paradigma založené na konceptu objektů, které mohou obsahovat data a kód:

- data ve formě polí (často známých jako atributy nebo vlastnosti)
- kód ve formě procedur. (často známé jako metody)

V OOP jsou počítačové programy navrženy tak, že jsou vytvořeny z objektů, které spolu vzájemně interagují

### **Úrovně abstrakce**

Abstrakce v objektově orientovaném programování (OOP) poskytuje způsob, jak skrýt složité implementační detaily za jednoduchá rozhraní. Existují dvě hlavní typy abstrakce v OOP: datová abstrakce a procesní abstrakce.

3. Datová abstrakce: Datová abstrakce skrývá vnitřní strukturu datových entit, což umožňuje programátorem pracovat s komplexními objekty, aniž by bylo nutné porozumět jejich podrobnostem.
4. Procesní abstrakce: Procesní abstrakce skrývá implementační detaily procesů, umožňuje uživatelům interakci s abstrahovanými akcemi, zatímco podkladové procesy se provádějí v pozadí

V souhrnu abstrakce v OOP zjednodušuje programování tím, že umožňuje vývojářům pracovat se zjednodušenými reprezentacemi složitých dat a procesů, aniž by bylo nutné porozumět jejich vnitřnímu fungování.

## OOP pojmy:

**Zapouzdření** (enkapsulace) = princip OOP, který umožňuje skrýt interní stav objektu a omezuje přístup k jeho datům a metodám pouze na definovaná rozhraní

**Dědičnost** = koncept v OOP, který umožňuje novým třídám získávat vlastnosti a chování svých nadřazených tříd (hierarchie tříd – potomci dědí po rodičích)

**Polymorfizmus** = schopnost objektu nebo funkce vykazovat různé formy chování v závislosti na kontextu, čímž umožňuje flexibilní a obecné použití kódu

**Třída** = základní stavební prvek OOP, který definuje vlastnosti a chování objektů daného typu; obecná definice položek a metod (typ)

**Objekt** = konkrétní instance třídy v OOP naplněný daty (proměnná), která má konkrétní stavy a může provádět akce definované v této třídě

## Diagramy UML a příklady

= grafické nástroje používané k vizualizaci, návrhu a dokumentaci softwarových systémů, které poskytují různé pohledy na strukturu, chování a interakce komponent systému

**Příklad třídy** reprezentována tabulkou se třemi segmenty: název<sup>1</sup>, atributy<sup>2</sup> a metody<sup>3</sup>

BankAccount	
-owner:String	1
-balance: Double = 0.0	2
+deposit: (amount: Double)	3
-withdraw: (amount: Double)	

Pro třídy lze nastavit specifickou **viditelnost**

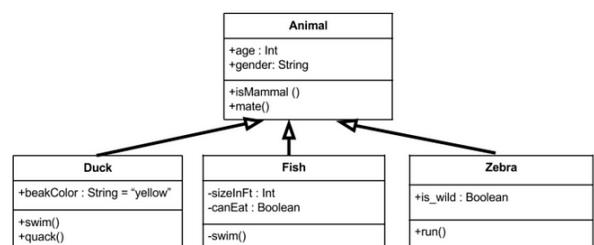
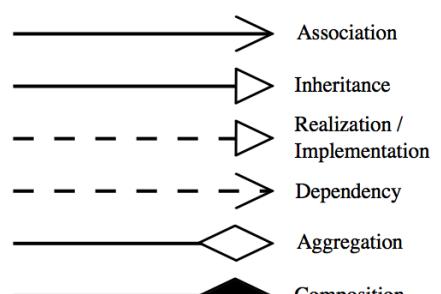
Typ	Notace	Jak definuje viditelnost
Public	+	Kdekoliv v programu a lze volat objektem v systému
Private	-	Definováno třídou
Protected	#	Definováno třídou nebo podtřídou této třídy
Package	~	Instancemi jiných tříd v stejném package

## Vztahy (relationships)

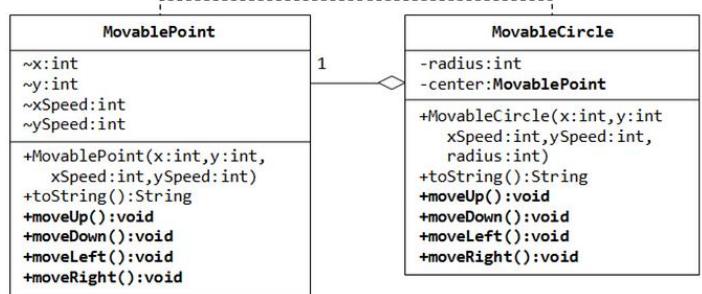
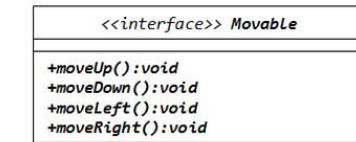
- a) **Sdružení (association)** = vztah mezi dvěma samostatnými třídami. Spojuje dvě zcela samostatné entity. Existují čtyři různé typy asociace: obousměrná, jednosměrná, agregační a reflexní. Nejběžnější jsou obousměrné a jednosměrné asociace.

To lze specifikovat pomocí multiplicity (jedna k jedné, jedna k mnoha, mnoho k mnoha atd.)

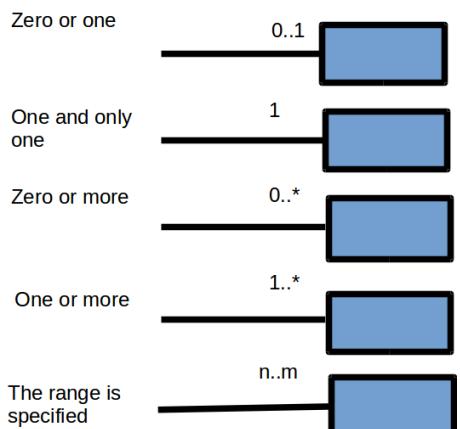
- b) **Dědičnost (inheritance)** = označuje, že podtřída (potomek) je považována za specializovanou formu nadřídy (rodiče)



- c) **Realizace/Implementace** = vztah mezi dvěma prvky modelu, ve kterém jeden prvek modelu implementuje/realizuje chování, které určuje druhý prvek modelu

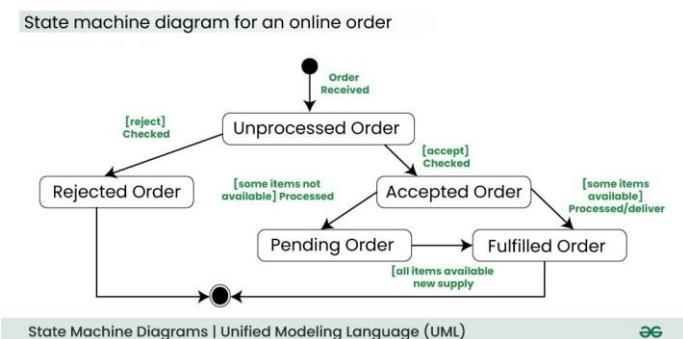


- d) **Závislost (dependency)** = řízený vztah, který se používá k prokázání, že nějaký prvek UML nebo sada prvků vyžaduje, potřebuje nebo závisí na jiných prvcích modelu pro specifikaci nebo implementaci (vztah *supplier – klient*)
- e) **Agregace** = forma asociace, která je jednosměrným (one way) vztahem mezi třídami. Vysvětlení: nazvat ho vztahem „má“ nebo „je součástí“ – např. dvě třídy: Peněženka a Peníze. Peněženka „má“ peníze. Ale peníze nepotřebují nutně mít peněženku, takže jde o jednosměrný vztah
- f) **Složení (composition)** = omezená forma agregace, ve které jsou dvě entity (třídy) na sobě vysoce závislé
- g) **Multiplicita** = popisuje, kolik instancí jedné třídy může být připojeno k instanci jiné třídy prostřednictvím daného přidružení  
(pozn. Kardinalita (cardinality) v UML určuje počet prvků vztahu mezi dvěma objekty nebo třídami, např. jeden k jednomu, jeden k mnoha nebo mnoho k mnoha)

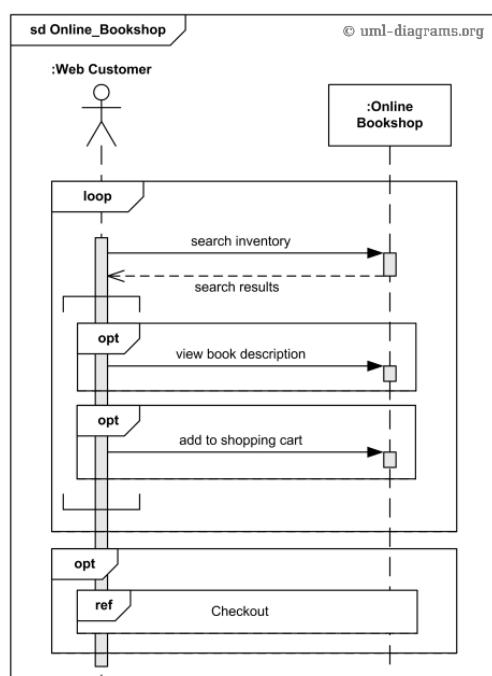


**Příklad sekvenčního diagramu** – Sekvenční diagram jednoduše znázorňuje interakci mezi objekty v sekvenčním pořadí, tj. v pořadí, ve kterém k těmto interakcím dochází; např: Online bookshop →

**Příklad stavů** – Vzor stavu je behaviorální vzor návrhu softwaru, který umožňuje objektu změnit své chování, když se změní jeho vnitřní stav (běh blízký konceptu konečných strojů) ↓



State Machine Diagrams | Unified Modeling Language (UML)



## 19. Návrhové vzory (design patterns), proč se používají, popis funkčnosti a zápis pomocí diagramů UML (uveďte příklady vzorů State, Strategy, Observer, Composite, ...).

### Návrhové vzory (design patterns)

Návrhové vzory jsou osvědčené a opakovaně použitelné řešení obvyklých problémů v softwarovém návrhu a vývoji, které poskytují strukturovaný a efektivní způsob řešení specifických situací či požadavků.

#### Proč se používají:

- Opětovná použitelnost: Vyvarování se pokaždé znovaobjevování kola
- Škálovatelnost a modularita: Navrhnutí flexibilní a adaptabilní software
- Údržba: Snazší úprava a ladění kódu
- Standardizace: Společný slovník a struktura napříč různými projekty
- Spolupráce: Snazší pro více vývojářů pracovat na stejně kódové základně

### Funkčnost

Liší se na základě konkrétních typů do tří skupin. Obecně mají funkčnost vzorů vytvářecích, chování a strukturní.

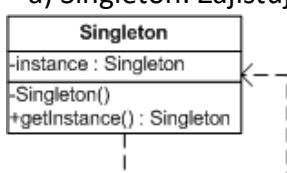
**Použití vzorce:** identifikace problému → najít vhodného vzorce → aplikace vzorce

### Příklady a zápis pomocí UML

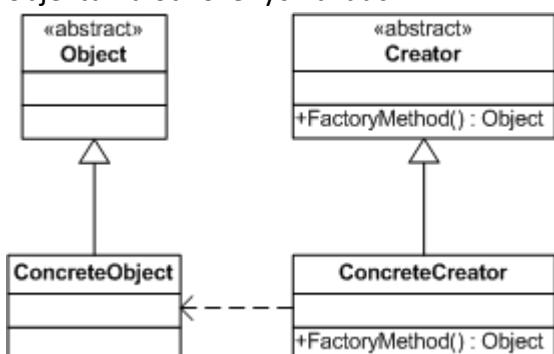
#### 1. Creational Patterns (Vzory vytváření):

= definují mechanismy pro vytváření instancí objektů. Implementace vzoru pro vytváření je zodpovědná za řízení životního cyklu vytvořeného objektu

a) Singleton: Zajišťuje, že třída má pouze jednu instanci a poskytuje globální přístup k ní.



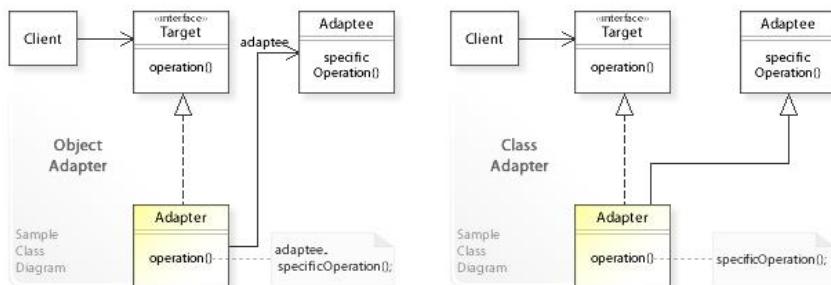
b) Factory Method: Definuje rozhraní pro vytváření objektů, ale ponechává konkrétní vytváření těchto objektů na odvozených třídách.



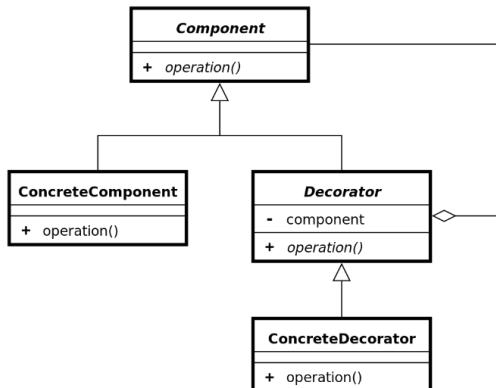
#### 2. Structural Patterns (Strukturní vzory):

Složení objektů a jejich organizace pro získání nové a rozmanité funkčnosti je základním základem strukturních vzorů.

a) Adapter: Objekt vzorce adaptéra obaluje různé nesourodé implementace rozhraní a představuje jednotné rozhraní pro přístup ostatních objektů



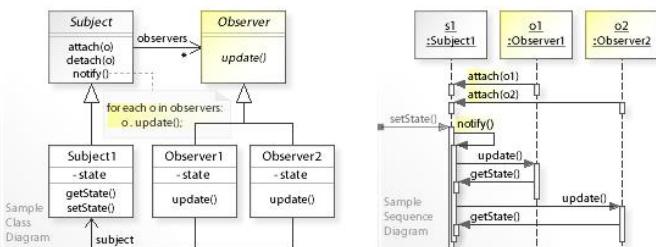
b) Decorator: Umožňuje dynamicky přidávat nové funkce nebo chování k existujícím objektům.



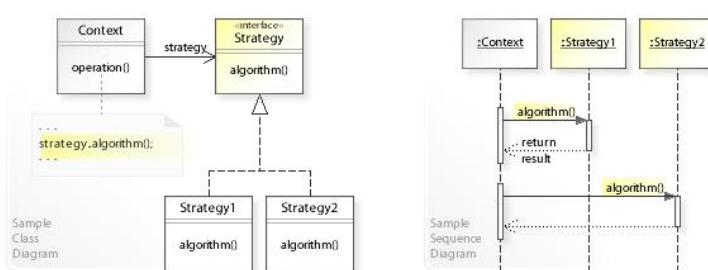
### 3. Behavioral Patterns (Vzory chování):

= zaštiťuje interakci mezi různými objekty

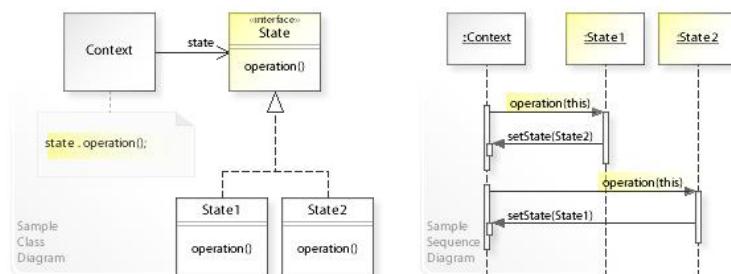
a) Observer: Definuje závislost mezi objekty tak, že když se stav jednoho objektu změní, všechny jeho závislé objekty jsou informovány a aktualizovány automaticky.



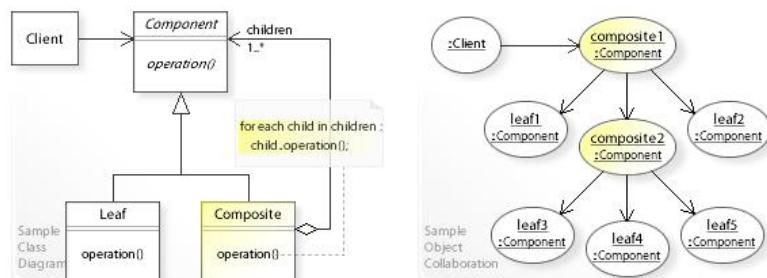
b) Strategy: Definuje sadu algoritmů, které mohou být vyměnitelně použity v rámci stejného kontextu.



c) State: Umožňuje objektu změnit své chování, když se změní jeho vnitřní stav.



d) Composite: Složený vzor popisuje skupinu objektů, se kterými se zachází stejným způsobem jako s jednou instancí stejného typu objektu. Záměrem kompozitu je „skládat“ objekty do stromových struktur, které reprezentují hierarchie část-celek. Implementace kompozitního vzoru umožňuje klientům zacházet s jednotlivými objekty a kompozicemi jednotně



## 20. WWW aplikace a služby. Architektura REST. Protokol HTTP a jeho verze, uchovávání stavové informace, cookie. Programování na straně klienta a serveru, jejich možnosti a omezení, nejběžnější používané prostředky a jazyky.

### Webové aplikace a služby (WWW)

= softwarové aplikace nebo funkce poskytované přes internetový prohlížeč, umožňující uživatelům přistupovat k obsahu, komunikovat, vykonávat operace a sdílet data online

**WWW aplikace:** aplikace naprogramovanou pomocí některého jazyka standardizovaného organizací W3C (World Wide Web Consortium), např. HTML

Aplikace bývají šířené pomocí HTTP protokolu nejčastěji nad TCP/IP protokolem využívaným celosvětovou sítí Internet.

### Webová služba:

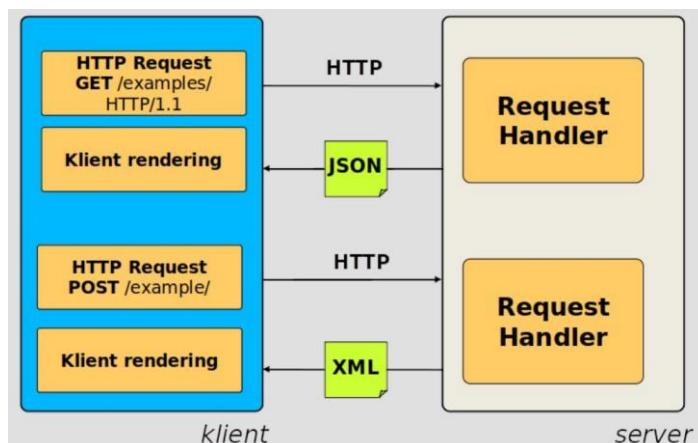
= softwarový systém navržený tak, aby podporoval interoperabilní interakci mezi stroji přes síť. Je to Webová aplikace pro jiné aplikace; Používání protokolu HTTP(S)

**Konzumenti webové služby:** další služby, web frontend JS SPA, nativní aplikace (mobilní, desktopové)

### Architektura webové služby:

- Komplexní služby (monolit)
- Mikroslužby / Microservices
- Cloud lambda funkce (AWS aj.)

Společné mají to, že vždy mají rozhraní (= API)



### REST (Representational State Transfer)

= softwarový architektonický styl, který byl vytvořen, aby řídil návrh a vývoj architektury pro World Wide Web (pro distribuované aplikace, hl. příklad je web); návrhový styl/vzor pro síťové protokoly

**Princip:** server odpoví reprezentací zdroje (=resource; nejčastěji HTML, XML nebo JSON dokument) a tento zdroj bude obsahovat hypermediální odkazy, po kterých lze stav systému změnit. Každá taková žádost zase obdrží reprezentaci zdroje a tak dále.

Důležitým důsledkem je, že jediný identifikátor, který je třeba znát, je identifikátor prvního požadovaného zdroje a všechny ostatní identifikátory budou objeveny. To znamená, že tyto identifikátory se mohou změnit, aniž by bylo nutné klienta předem informovat, a že mezi klientem a serverem může být pouze volné spojení.

## **REST – resource (zdroj)**

Základní abstrakce stavu aplikace – vždy musí mít:

- Uniform interface: URI, URL (uchování stavu aplikace)
- Methods: konečná množina operací s definovaným chováním
- Representation: konečná množina datových typů, podpora code on demand

## **HTTP (Hypertext Transfer Protocol)**

= protokol postavený podle REST arch. stylu a používaný pro přenos informací na internetu, který umožňuje komunikaci mezi klientem a serverem prostřednictvím požadavků a odpovědí

Navržen podle REST, proto je client-server, bezstavový, cacheable a vrstvový

### **Metody http:**

GET – bezpečná, opakovatelná, cacheable

POST DELETE – opakovatelná

PUT – opakovatelná

HEAD – bezpečná, opakovatelná

**TCP (Transmission Control Protocol):** spolehlivý, spojovaně orientovaný protokol používaný v síťové komunikaci pro zajištění doručení dat v pořadí, bez chyb a s potvrzením doručení mezi komunikujícími zařízeními

Verze	Status	Info
HTTP/0.9	Neaktuální	-
HTTP/1.0	Neaktuální	-
HTTP/1.1	Standard	Problémy s rychlostí
HTTP/2	Standard	Řeší problémy
HTTP/3	Standard	Zatím málo používaný

## **Uchovávání stavové informace**

HTTP je bezstavový – protokol neumí uchovávat stav komunikace, dotazy spolu nemají souvislost

Významnou překážkou při sledování stavu webové aplikace představuje pro programátora samotný tenký klient – webový prohlížeč

Z hlediska udržení stavu tři skupiny webových aplikací:

1. Webové aplikace bezestavové – Představitelem statické (x)HTML stránky nebo malé funkční bloky typu „aktuální přesný čas“, které nemají nutnou návaznost na předchozí kroky (stavy)
2. Stav dán jen klientovou pozicí na určité stránce/pohledu/stavu v aplikaci – např. webové aplikace bez nutnosti autentizace, kdy je zobrazená stránka závislá na předchozím kroku, ovšem informace o předchozím stavu je pro aplikaci ztracena. K dispozici ji má pouze samotný tenký klient (tlačítko "Zpět")
3. Stav je dán mnoha proměnnými – stav z bodu 2 doplněný o další přenášené informace. Typicky po autorizaci uživatele (cookies, sessions, certifikáty, ...), vícestránkové formuláře s hidden elementy atd.

## **Cookies**

= v protokolu HTTP označuje malé množství dat, která WWW server pošle prohlížeči, který je uloží na počítači uživatele. Při každé další návštěvě téhož serveru pak prohlížeč tato data posílá zpět serveru.

Cookies běžně slouží k rozlišování jednotlivých uživatelů, ukládá se do nich obsah „nákupního košíku“ v elektronických obchodech, uživatelské předvolby apod.

Programování na straně klienta a serveru, jejich možnosti a omezení, nejběžnější používané prostředky a jazyky.

### **Programování na straně klienta**

= slouží k změně chování rozhraní v rámci konkrétní webové stránky v reakci na akce vstupního zařízení nebo při určitých událostech načasování. V tomto případě dochází k dynamickému chování v rámci prezentace

**Obsah** na straně klienta je generován v místním počítačovém systému uživatele

Slouží hlavně k uspořádání médií (zvuk, animace, změna textu atd.) prezentace klientovi

**Jazyky:** nejčastěji JS (+ různé frameworky jako React, Vue, Angular), kotlin, Swift

**Nástroje:** webové prohlížeče (chrome, Firefox, ...), mobilní (android, iOS) a desktopové aplikace

**Výhody:** rychlá reakce aplikace na podněty uživatele

**Nevýhody:**

- skriptování nemusí být na straně klienta dostupné
- internetové prohledávače nemusí být schopné zaindexovat obsah
- lze provádět jen jednoduché operace, interpretace Javascriptu nebývá nejrychlejší
- kód klientského programu je dostupný uživateli

### **Programování na straně serveru, jejich možnosti a omezení, nejběžnější používané prostředky a jazyky**

= technika používaná při vývoji webu, která zahrnuje použití skriptů na webovém serveru, který vytváří přizpůsobenou odpověď na žádost každého uživatele (klienta) na webovou stránku.

Skriptování na straně serveru se liší od skriptování na straně klienta, kde se na straně klienta ve webovém prohlížeči spouštějí vložené skripty, jako je JS

**Jazyky:** JS/Node.js pro serverovou logiku (Node.js běžící na serveru)

Python (s frameworky jako Django, Flask) pro vývoj webových aplikací a API

Java (s frameworky jako Spring, Jakarta EE) pro robustní webové aplikace a velké systémy

C#/.NET pro vývoj webových aplikací a API na platformě Microsoft

**Prostředky:** Fyzické servery nebo virtuální servery (cloudové služby jako AWS, Azure, Google Cloud) a kontejnerové technologie (Docker, Kubernetes) pro správu a nasazování aplikací

**Možnosti:**

- Škálovatelnost: Server navržen pro zpracování velkého množství požadavků od klientů současně
- Zpracování logiky aplikace: Server může provádět složité výpočty, manipulaci s daty a udržování stavu aplikace
- Bezpečnost: Ochrana dat, provádění autentizaci a autorizaci klientů.

**Omezení:**

- Centralizovaná architektura: Server je jediným bodem, ke kterému přistupují všichni klienti, což může vést ke zpomalení nebo selhání systému při příliš velkém počtu požadavků
- Zpoždění sítě: Zpomalení komunikace mezi klientem a serverem kvůli latenci sítě
- Náklady na provoz a údržbu

## 21. Princip práce webových služeb, serializace, SOAP + WSDL + UDDI vs. RESTful (popis, srovnání, výhody a nevýhody), strojově čitelné formáty pro komunikaci.

### Princip práce webových služeb

Webové služby používají ke komunikaci mezi aplikacemi metodu požadavek-odpověď (request-response) Pro komunikaci potřebujeme médium (pro web services internet) a společný formát (běžně XML, nebo JSON)

**Klient:** dává požadavek (request) na nějakou službu od serveru

**Server:** poskytovatel služeb

Požadavek je odeslán prostřednictvím zprávy, která je v běžném formátu XML a v reakci na tuto žádost poskytovatel služby odpoví zprávou ve společném formátu (tj. XML).

### Serializace

= kritický proces při konverzi datových struktur nebo stavů objektů do formátu, který lze uložit nebo přenést a později rekonstruovat

Nejčastější formáty pro serializaci dat: JSON, XML, YAML, CSV, ProtoBuf a MessagePack

### SOAP (Simple Object Access Protocol)

Je to protokol udávající, jak bude probíhat komunikace mezi aplikací, založeno na XML (envelope+header+body)

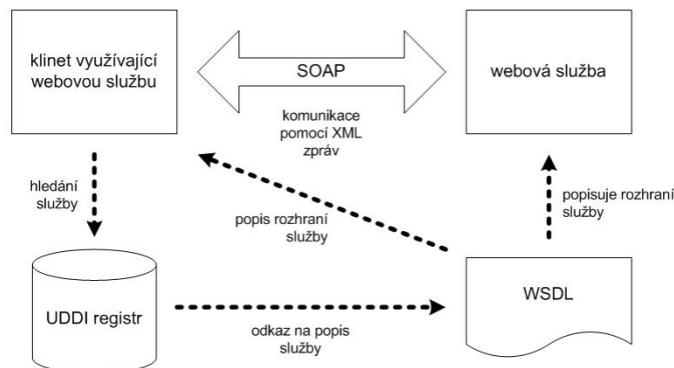
### WSDL (Web Services Description Language)

dokument XML obsahující pravidla pro komunikaci mezi různým softwarem. Definuje:

- Jak může k této službě přistupovat systém, který ji požaduje z jiných systémů
- Název služby
- Konkrétní parametry potřebné pro přístup k této službě a návratový typ
- Chybové zprávy zobrazené při problému s přístupem k datům

### UDDI (Universal Description, Discovery, and Integration)

adresář, který poskytuje podrobnosti o tom, který software je třeba kontaktovat pro konkrétní typ dat



### 1. Webové služby SOAP

- protokoly jsou založeny na XML
- jsou nezávislé na jazyce a lze je provozovat na jakékoli platformě
- podporují stavové i bezstavové operace

**stavový (stateful)** = server sleduje informace přijaté od klienta při každém požadavku

**bezstavový (stateless)** = každý požadavek obsahuje dostatek informací o stavu klienta a

server se tak nemusí trápit s ukládáním stavu klienta a tím se zvyšuje rychlosť komunikace

## **RESTful webové služby**

- také jazykově a platformově nezávislé a jsou rychlejší ve srovnání se SOAP
- dnes více využívané než SOAP
- s daty zacházejí jako se zdroji, vracejí data ve formátu JSON nebo XML.
- vytvářejí objekt a odesílají stav objektu v reakci na požadavky klienta
- operace (metody) CRUD = create retrieve update delete

(pozn. RPC (Remote Procedure Call) = mechanismus, který umožňuje volání funkcí nebo metod na vzdáleném počítači jako by byly volány lokálně)

### **Srovnání RESTful vs SOAP:**

- SOAP odpovídá principům RPC a je orientován na volání procedur/metod/operací + výměna dat
- REST je orientován na práci se zdroji
- Volám-li spíš procedury, používám middleware – SOAP
- Pracuji spíše s daty, klientem je browser – REST
- REST – sémantika a množina operací je konečná (CRUD)
- SOAP – větší režie volání, nekonečná sémantika, větší volnost ale těžší implementace
- Volba technologie v závislosti na aplikaci

## **Strojově čitelné formáty pro komunikaci**

= způsoby reprezentace dat, které jsou snadno čitelné a zpracovatelné počítači

JSON (JavaScript Object Notation)

- reprezentace struktur dat v podobě objektů a pole

XML (eXtensible Markup Language)

- formát s hierarchickou reprezentací a oddělením obsahu a struktury

CSV (Comma-Separated Values)

- formát pro tabulková data, který odděluje hodnoty čárkami/jinými oddělovači; v DB

Protobuf (Protocol Buffers)

YAML (YAML Ain't Markup Language) - hierarchická struktura; užívá se v konfiguračních souborech

## **22. Distribuované webové aplikace. Typy škálování, teorém CAP. Koncepce programování. Principy asynchronní komunikace pomocí front a událostí.**

### **Distribuované webové aplikace**

**Definice:** Distribuované webové aplikace jsou aplikace, které jsou nasazeny na více než jednom serveru a komunikují mezi sebou přes síť, aby poskytovaly služby uživatelům

= backend aplikace běží na více strojích

**Význam:** Umožňují zpracování velkého množství dat, vysoce dostupné služby a škálovatelnost aplikací na základě potřeb

### **Typy škálování**

**Vertikální škálování:** Zvětšení výkonu serveru (počítače) přidáním dalších zdrojů, jako jsou CPU, paměť nebo úložiště

- HW má konečné limity a neumí škálovat rovnoměrně, náklady rostou exponenciálně ne lineárně

**Horizontální škálování:** Přidání dalších serverů (strojů) do stávající infrastruktury, což umožňuje rovnoměrné rozdělení zátěže

- Load balancer rozděluje zátěž; lze použít pro aplikaci i data

### **Teorém CAP**

**Definice:** Teorém CAP popisuje možnosti distribuovaných systémů vzhledem k dostupnosti (Availability), konzistenci (Consistency) a toleranci na partition (Partition tolerance).

- a) Konzistence: všechny uzly obsahují stejnou verzi dat, klient vždy dostává stejný pohled na data
- b) Dostupnost: systém pracuje i při výpadku uzlu, všichni klienti mohou číst a zapisovat
- c) Partition tolerance: systém zůstává funkční i při rozpojení sítě

### **Koncepce programování:**

1. **Rozdelení na mikroslužby:** Distribuované aplikace mohou být navrženy jako sada nezávislých mikroslužeb, které komunikují přes API
  - o Aplikace složená ze služeb, které jsou propojené přes API
  - o Jde v podstatě o rozšíření principů OOP na úroveň aplikací
2. **Monolit:** Aplikace je vyvíjena jako jediný a integrovaný celek, kde všechny funkce a komponenty sdílí jeden kódový základ a běží v jediném prostředí
  - o Bezstavová aplikace a má opakovatelné operace bez vedlejších efektů
  - o Dělitelná data, bez zámků, asynchronní I/O

### **Principy asynchronní komunikace pomocí front a událostí**

Pozn. JS je single threaded, ale browser má neomezeně vláken -> asynchronní chování pomocí BrowserAPI (užívá TaskQueue, Callback, Promise)

**Asynchronní komunikace** = paralelní provádění požadavků klienta, zajištěné message brokerem (middleman, prostředník) mezi klientem a serverem

#### **Pomocí front (message queue):**

- *Klient* posílá požadavky do fronty, což je potvrzeno odezvou o přidání, nemusí tak čekat na server
- *Server* dostává požadavky z fronty a podle množství, které může zpracovávat najednou je odbaví díky asynchronní funkci

#### **Pomocí událostí (event-driven):**

- Služby na straně serveru čekají na nějakou událost (např. update statusu klienta), pokud dojde na straně klienta k provedení události, začne server jednat
- Mikroslužby odebírají události (subscribe to the events), aby je mohly přijímat asynchronně

## 23. Architektury paralelních systémů, granularita, Flynnova klasifikace (6 typů) + příklady existujících hardwarových struktur.

Pozn.:

Paralelismus = vytváření souběžnosti (ideál)

Pseudoparalelismus = procesy se dělí o časové kvantum jednoho CPU (realita)

### Architektury paralelních systémů

Multiprocesor = dva a více CPU v jednom počítače

#### a) Multiprocesory se sdílenou pamětí

- Procesory sdílí paměť RAM – „vše na jedné desce“
- U symetrických multiprocesorů běží kterýkoliv proces na libovolném procesoru – CPU jsou univerzální
- U asymetrických multiprocesorů jsou CPU specializované
- Komunikace přes sdílenou paměť, nutnost synchronizace

#### b) Multiprocesory s distribuovanou pamětí

- Jednotlivé počítače propojeny komunikační sítí
- Každý počítač je „plnohodnotný“ – vlastní sada CPU, RAM ...
- Komunikace pomocí zasílání zpráv (message passing)
- Masivně paralelní počítače + clustery

### Granularita

Dle velikosti úkolu, množství práce k vykonání (od nižší do vyšší úrovně granularity)

#### a) Nejnižší (fine grained)

- Paralelizace pomocí elementárních podprogramů
- Řešení na úrovni HW a strojových instrukcí
- Např. zřetězené zpracování (pipelined)

#### b) Střední (middle grained)

- Paralelizace na úrovni několika CPU (možný zásah vývojáře)
- Rozdělení úlohy na několik spolupracujících – nutno řešit sdílení dat, komunikaci, synchronizaci

#### c) Nejvyšší (coarse grained)

- Architektura s řadou CPU + simultánní běh několika úloh

### Flynnova klasifikace (6 typů)

Systémy klasifikovány dle počtu toků instrukcí a dat

#### 1. SISD (jeden tok instrukcí, jeden tok dat)

Počítač zpracovává data sériově podle jednoho programu

Př.: Klasický jednoprocesorový počítač von Neumannova typu

#### 2. SIMD (jeden tok instrukcí, vícenásobný tok dat)

Počítač s více stejnými procesory řízenými jedním stejným programem

Všechny CPU, ale provádí stejnou instrukci, pracují synchronně

Př.: Vektorové počítače, MMX, SSE rozšířené sady instrukcí, maticové počítače

#### 3. MISD (vícenásobný tok instrukcí, jeden tok dat)

Sada procesorů provádí různou činnost na stejných datech (nejedná se o pipeline!)

Př.: Speciální systémy – neuronové sítě, simulace mozku aj.

4. **MIMD** (vícenásobný tok instrukcí, vícenásobný tok dat)

Víceprocesorový systém, každý procesor řízen vlastním programem, pracuje nad vlastními daty

Dělí se dle práce s pamětí – sdílená nebo distribuovaná paměť

Př.: Symetrické multiprocesory (SMP), clustery a masivně paralelní počítače

Dnes 99% všech systémů

5. **MSIMD** (vícenásobné SIMD)

Systém, ve kterém pracuje několik SIMD podsystémů nezávisle na sobě

Jednotlivé podsystémy zpracovávají jiný program, proto řízeny stejně jako systém MIMD

6. **SPMD** (stejný program, vícenásobný tok dat)

Další modifikace systému SIMD

Všechny procesory vykonávají stejný program, ale jsou na sobě nezávislé (nejsou synchronizovány).

Jednotlivé procesory musí mít svůj řadič, který řídí rychlosť příslušného inštrukcí a jejich provádění

## **24. MPI (Message Passing Interface) – popis, oblast použití, komunikátory, synchronizace, základní sada operací (funkční), multicore vs. cluster mode, validní příjem zprávy, problém blokujících operací a jak je řešit, srovnání s PVM.**

### **MPI (Message Passing Interface)**

= standardizované API, které umožňuje procesům ve výpočetních klastrech komunikovat mezi sebou pomocí výměny zpráv

Jednotlivé procesy namapovány na množinu počítačů (uzlů, procesorů) v režii 1 proces = 1 procesor

K mapování dochází při běhu aplikace za pomoci démona (agenta) mpirun

### **Oblast použití**

především ve vysokovýkonných výpočetních prostředích (HPC) pro vědecké a inženýrské aplikace – simulace, analýzu dat, modelování a další numerické výpočty

### **Komunikátory**

**Komunikátor** = skupina procesů v rámci běžící MPI aplikace

Lze je dynamicky vytvářet při startu i za běhu

Vždy existuje globální komunikátor (MPI\_COMM\_WORLD) – zahrnuje všechny procesy dané aplikace

Každý proces má vlastní identifikátor v rámci komunikátoru – *rank*

Pro komunikaci jsou k dispozici 2 mechanismy: Message passing a Remote memory access

Každý komunikátor má vlastnost *size* = počet procesů (resp. procesorů) v daném komunikátoru

Rank a size komunikátoru jsou jeho jednoznačnými identifikátory – identifikace je nutná pro adresaci požadavků mezi procesy

### **Synchronizace**

Každý komunikátor realizuje tzv. bariéru, na které je možno všechny jeho procesy synchronizovat → běh procesů pokračuje tehdy, když se na bariéře „sejdou“ všechny procesy komunikátoru (jedná se tedy o synchronní (blokující) operaci)

### **Základní typy komunikace mezi jednotlivými procesy**

#### a) Komunikace point-to-point

- Slouží ke komunikaci mezi 2 procesy
- Identifikace příjemce – komunikátor + rank
- Typicky rozeslání požadavku od master uzlu svým „workerům“
- Operace jsou: Blokující (synchronní) a Neblokující (asynchronní)

Funkce point-to-point:

##### 1 Blokující funkce

- Analogie mechanismu request/response
- Jeden proces zašle jinému zprávu s žádostí o operaci, případně související data, následně proces čeká na odpověď, po obdržení odpovědi pokračuje v běhu

##### 2 Neblokující funkce

- Klasická asynchronní komunikace
- Jeden proces zašle zprávu, nicméně nečeká na odpověď a pokračuje v běhu (není blokován)
- Obsahuje mechanismus na zachycení odpovědi, která přijde se zpožděním

b) Kolektivní (globální) komunikace

Určena k zasílání zpráv (požadavků, dat) v rámci celého komunikátoru

Cílem je celá skupina procesů

Rozdělení do skupin: Synchronizace, přesuny dat a redukční operace

## Základní sada operací (funkční)

Point-to-Point Operace: MPI\_Send, MPI\_Recv – odesílání a příjem zpráv mezi dvěma procesy

Collective Operations: MPI\_Bcast (rozesílání dat), MPI\_Reduce (redukce dat), MPI\_Gather (shromáždění dat), MPI\_Scatter – operace, které zahrnují všechny procesy v komunikátoru

Synchronizace: MPI\_Barrier – synchronizace všech procesů.

Funkce pro inicializaci + ukončení (finalizaci) výpočtu

Funkce na práci s ranky

**Tag** – má ho každá zpráva a slouží ke správání původního request a respons

## Multicore vs. Cluster Mode

**Multicore Mode:** na vícejádrovém HW, procesy běží jako samostatné vlákna na různých jádrech

**Cluster Mode:** procesy běží na různých fyzických strojích a komunikují přes síť

## Validní příjem zprávy

Pro volání operace příjmu (MPI\_Recv) potřebujeme znát:

- 1) Info o délce bufferu
- 2) Datový typ
- 3) Rank odesílatele

## Problém blokujících operací a jak je řešit

Blokující operace mohou vést k deadlockům, kdy procesy čekají na příjem zpráv, které nikdy nepřijdou.

Tento problém lze řešit správnou implementací asynchronní (neblokující) operace v kritických místech

## Srovnání s PVM (Parallel Virtual Machine)

**PVM** = systém, který umožňuje programátorům pohlížet na heterogenní soubor strojů jako na jednolity paralelní počítač

- a) Flexibilita: PVM nabízí větší flexibilitu v dynamickém přidávání a odebrání uzel během běhu programu
- b) Portabilita: MPI je více standardizovaný
- c) Výkonnost: MPI je navrženo pro vysoký výkon s optimalizací na nízké úrovni pro specifické hardwarové architektury
- d) Rozsah použití: MPI je běžně používáno pro rozsáhlé paralelní aplikace, zatímco PVM je více zaměřeno na heterogenní výpočetní prostředí

## 25. Platforma Android, základní komponenty aplikace, životní cykly, intenty, persistentní ukládání dat, práce s vlákny, výměna dat, services, notifikace, broadcasting.

### Platforma Android

Poskytuje: OS, uživatelské prostředí pro koncové uživatele, specifikace ovladačů pro výrobce HW, nástroje pro vývoj (SDK)

Architektura – 5 vrstev:

- Linux Kernel – jádro OS
- Libraries – knihovny, napsány v C/C++
- Android Runtime – aplikační virtuální stroj DVM
- Application Framework – aplikační vrstva pro vývojáře
  - Sada View prvků – UI aplikací (button, checkbox, list ..)
  - Content providers – sdílení dat mezi aplikacemi
  - Resource manager – zdroje jako lokalizace, grafika, design
  - Notification manager – zasílání notifikací do stavového řádku
  - Activity manager – životní cyklus aplikací, zásobník aplikací
- Applications – samotné uživatelské aplikace

Verze Android SDK, vývoj v Android Studio IDE

### Základní komponenty aplikace

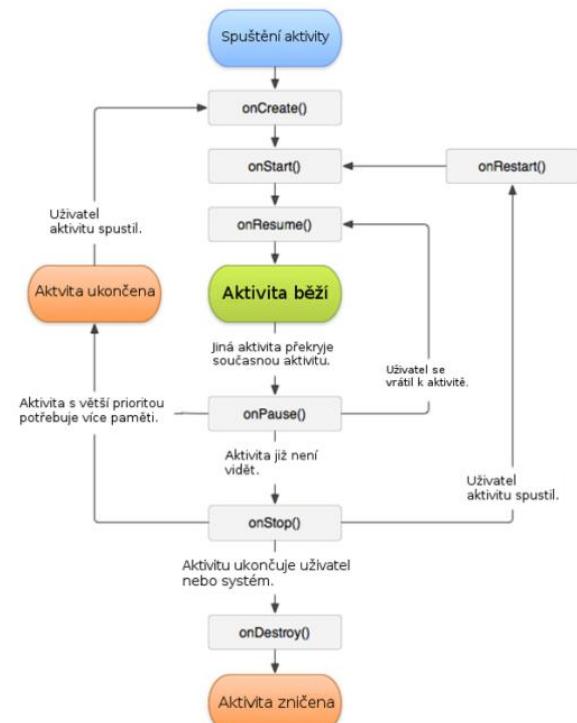
#### Activity

= Reprezentuje jednu obrazovku uživatelského rozhraní v aplikaci (zákl. komponenta)

Hlavní funkce: Inicializace uživatelského rozhraní, interakce s uživatelem.

Životní cyklus:

- Aktivita spuštěna – došlo ke spuštění aplikace
- Aktivita běží – aktivita spuštěna a je v popředí
- Aktivita v pozadí – je vidět, ale překryta jinou aplikací (příchozí SMS, hovor, jiná notifikace ..)
- Aktivita zastavená – není vidět, bez přístupu, ale není úplně zničena v Aktivita ukončená – úplné ukončení aktivity



#### Views

= stavební blok uživatelského rozhraní, reprezentuje komponenty jako tlačítka, textová pole

#### Intents

= základní asynchronní komunikační nástroj mezi prvky aplikací

Třída, která obsahuje popis a data nějakého záměru

Objekt s definicí cílového procesu s možností zaslat mu data

Typy:

- Explicitní – cílené na konkrétní komponentu, tzn. třídu

- b) Implicitní – pouze info o záměru (např. chci psát email) a případná data k předání
  - nechá na systému, kterou aplikaci (aktivitu) spustí nebo nabídne

## Resources

Layout: Definice rozložení uživatelského rozhraní (XML soubory).

Menu: Definice nabídky aplikace.

Styly: Definice vzhledu a formátování komponent.

Stringy: Textové řetězce používané v aplikaci.

Values: Ostatní hodnoty jako barvy, dimenze, konstanty.

## AndroidManifest.xml (Manifest)

= Konfigurační soubor aplikace.

Hlavní funkce: informace o balíčku, povolení, oprávnění komponent, deklarace komponent, metadata a verzi aplikace

## Persistentní ukládání dat

Android poskytuje dle účelu uložení a typu dat tyto způsoby:

- A) Shared Preferences – ukládání primitivních datových typů v podobě key / value; pro jakýkoliv účel
- B) Internal Storage – ukládání souborů do interní paměti zařízení (privátní soubory pro danou aplikaci)
- C) External Storage – ukládání souborů na SD kartu nebo veřejnou paměť
- D) SQLite DB – ukládání do privátní DB
- E) Sítová úložiště – cloudy, webové služby, ...

## Práce s vlákny

Každá aplikace má hl. vlákno – UI Thread, ve kterém běží veškeré Activity + Services s Broadcast Receivers

Třída Handler – Možnost asynchronního vykonání operace + promítnutí změn do UI

- Na principu zasílání a příjmu zpráv

AsyncTask – kombinace metod, z nichž jedna běží ve vedlejším vlákně a průběžné výsledky se spouští v hlavním

## Výměna dat

S daty se pracuje uvnitř aplikace, mezi aplikacemi a výměnou s externími systémy

Formáty dat: XML (standardizované + organizované, ale redundantní), JSON (malá redundance, ale méně strukturované)

## Services

Další mechanismus pro práci na pozadí, hl. pro dlouhodobou kontinuální činnost

Životní cyklus nezávisí na Activity

## Notifikace

O notifikace se stará služba Notofocation Manager – řeší se vzhled, informace a co udělá když se na ni klikne (pomocí Intent, např. otevře aplikaci, ...)

## Broadcasting

Slouží k zasílání daného Intentu zaregistrovaným BroadcastReceiverům

Broadcast je:

- a) Normal (non-ordered) – asynchronní, BroadcastReceivers obdrží zprávu bez definovaného pořadí
- b) Ordered – zpracovává jeden receiver po druhém, pořadí dle atributu priority

## **26. Relační databázový model (schéma relace, integritní omezení), normalizace v relačním modelu, bezzávislá dekompozice, optimalizace databázových struktur (typy indexů, případy jejich využití, výhody a nevýhody jednotlivých typů indexů).**

Pojmy:

Tabulka = relace

Entita = objekt

Schéma = výsledek modelování, zobrazeno formou diagramu

Atribut = vlastnost

Operace (seřazeny dle priority): projekce, selekce, kartézský součin, spojení, rozdíl, sjednocení, průnik

### **Relační databázový model**

= databázové schéma nezávislé na konkrétní implementaci relační DB

Relační databáze (RDB) = systém pro ukládání a správu dat, kde jsou data organizována do tabulek (relací) propojených pomocí klíčů

Konstrukty modelu

#### **1) Entita**

Má atributy – píšeme E(A) [entita E má množinu atributů A]

- Atribut
  - Je proměnlivý nebo stálý; má datový typ
  - Atomický (jednoduchý) nebo složený (např. adresa = psč, město, ulice)
  - Jednohodnotový nebo vícehodnotový (např. telefonní číslo)
  - Povinný nebo nepovinný (NULL) – značka NN
  - Jedinečnost (unique) – značka U
  - Identifikační klíč = atribut/kombinace atributů, která jednoznačně odliší instanci entity od jiných

#### **2) Vztah**

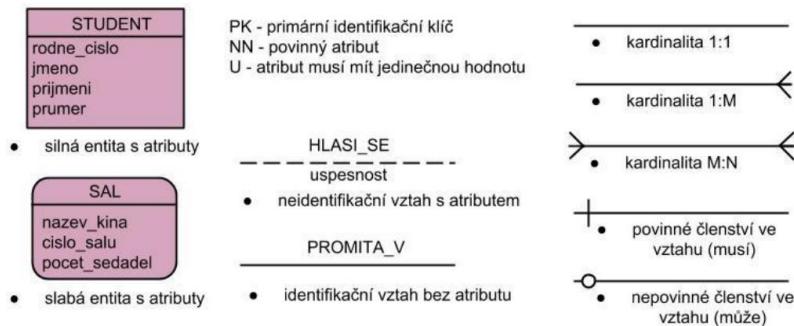
= vazba nebo asociace mezi entitami

- stupeň vztahu = počet entit ve vztahu

#### **3) Integritní omezení (IO)**

- entitní (na atribut, nebo kombinaci atributů)
  - doménové
  - null
  - jedinečnost
  - identifikační klíč
- vztahové
  - kardinalita – popisuje počet možných relací pro každou participující entitu  
1:1, 1:M, M:N
  - Členství ve vztahu (parcialita, existenční závislost) – popisu, jestli se všechny entity musí nacházet ve vztahu
    - totální (povinný) musí
    - parciální (nepovinný) může
  - Slabá (identifikačně a existenčně závislá na jiné entitě)
  - Silná (identifikačně a existenčně nezávislá) entita
- enterprise (na více entit, rozšíření entitní a vztahové)

## Přehled notace



## Normalizace v relačním modelu

= proces organizace dat v DB tak, aby se minimalizovala redundancy (duplicity data) a zajistila se integrita dat

**Primární klíč (PK)** = jedinečný identifikátor pro každý záznam v DB tabulce, který zajišťuje, že každý záznam je jednoznačně odlišitelný

**Kandidátní klíč (KK)** = atribut nebo kombinace atributů v DB tabulce, který může jednoznačně identifikovat každý záznam a ze kterého může být vybrán PK

**Funkční závislost (FZ)** = popisuje vztahy mezi atributy v relaci (vztah 1:1), vyjadřují integrální omezení

**Uzávěr atributu/ů X** = je množina všech atributů, které jsou funkčně závislé na X

### Normální formy:

#### 1NF (První normální forma)

Kritéria:

- Každá tabulka má jedinečný primární klíč
- Všechny atributy obsahují pouze atomické (nedělitelné) hodnoty
- Všechny hodnoty ve sloupci jsou stejného typu

#### 2NF (Druhá normální forma)

Kritéria:

- Tabulka je v 1NF
- Každý neklíčový atribut je plně funkčně závislý na PK

#### 3NF (Třetí normální forma)

Kritéria:

- Tabulka je v 2NF
- Žádný neklíčový atribut není tranzitivně závislý na PK

#### BCNF (Boyce-Coddova normální forma)

Kritéria:

- Tabulka je v 3NF
- Každý determinant (atribut na kterém závisí jiný atribut) je KK

**Výhody normalizace:** Minimalizace redundancy, zajištění integrity, lepší výkon

**Nevýhody normalizace:** Složitější dotazy

#### Anomálie v kontextu normalizace v relačním modelu

= problémy, které mohou nastat v databázi, pokud není správně normalizována

**INSERT anomálie** – při vložení nového záznamu (n-tice) musíme upravovat všechny výskytu a znát i všechny další atributy

- Př. u každého nového studenta musíme vložit i adresu univerzity, kde se hlásí. Když to popletu, v DB budou dvě různé adresy pro jednu univerzitu. => nekonzistence dat

**DELETE anomálie** – při vymazání může dojít ke ztrátě dat

**UPDATE anomálie** – při změně musíme upravovat všechny výskytu

- Př. když chceme změnit příjmení studenta, musíme to změnit ve všech řádcích, kde se daný student vyskytuje. Když to popletu, v DB budou dva různí studenti se stejným rodným číslem. => nekonzistence dat

## **Bezztrátová dekompozice**

**Dekompozice** = proces rozdělení tabulky na více menších tabulek tak, aby se minimalizovala redundance a zachovala integrita dat

Má za úkol zachovat sémantiku (význam) – Sémantika schématu je dána pomocí integritních omezení (IO), která jsou vyjádřeny (nejen) pomocí funkčních závislostí (FZ)

Mělo by platit:

- Výsledná schémata by měla mít stejnou sémantiku
- Výsledné relace by měly obsahovat stejná data, jaká obsahovala původní relace

## **Optimalizace databázových struktur (typy indexů, případy jejich využití, výhody a nevýhody jednotlivých typů indexů)**

Indexy = pomocná datová struktura, která slouží k urychlení základních operací nad záznamy

Typy indexů:

### **1. B-stromy a B+ stromy**

- Využití: Vhodné pro rozsahové dotazy.
- Výhody: Efektivní vkládání, mazání a vyhledávání.
- Nevýhody: Vyšší složitost implementace a údržby.

### **2. Hash indexy**

- Využití: Rychlé přesné vyhledávání (point queries).
- Výhody: Konstantní čas pro vyhledávání.
- Nevýhody: Nevhodné pro rozsahové dotazy, vyšší paměťová náročnost.

### **3. Bitmapové indexy**

- Využití: Databáze s nízkou kardinalitou atributů (např. sloupce s málo unikátními hodnotami).
- Výhody: Efektivní pro dotazy na více hodnotách, menší paměťová náročnost.
- Nevýhody: Nevhodné pro časté aktualizace.

### **4. Clustered indexy**

- Využití: Primární index, který určuje fyzické uspořádání záznamů.
- Výhody: Rychlejší rozsahové dotazy.
- Nevýhody: Může vést k fragmentaci při častých aktualizacích a vkládáních.

### **5. Non-clustered indexy**

- Využití: Sekundární indexy, které neovlivňují fyzické uspořádání dat.
- Výhody: Flexibilita v indexování více sloupců, může být více než jeden na tabulku.
- Nevýhody: Vyžaduje dodatečný prostor a může zpomalit vkládání a aktualizace.

## **Shrnutí volby**

B-stromy a B+ stromy jsou vhodné pro rozsahové dotazy

Hash indexy pro rychlé přesné vyhledávání

Bitmapové indexy pro nízkou kardinalitu atributů

Clustered indexy pro rychlejší přístup k rozsahovým dotazům

Non-clustered indexy poskytují flexibilitu v indexování více sloupců

## **27. Transakční zpracování dat (ACID, typy konfliktů, stupně izolace)**

**Transakce** = série příkazů čtení či zápisu na databázových objektech

**Čtení**: pro čtení se musí db objekt přenést do paměti z HDD a následně je přenesena do proměnné programu

**Zápis**: db objekt je modifikován v paměti a následně zapsán na disk

**DB objekty**: Jednotky, se kterými pracují programy – např. stránky, záznamy, ...

### **ACID**

#### **Atomicita (Atomicity)**

Definice: Transakce je nedělitelná jednotka práce; buď se provede celá, nebo vůbec.

Např.: Při převodu peněz mezi účty se buď odečte částka z jednoho účtu a přičte na druhý, nebo se nic nezmění.

#### **Konzistence (Consistency)**

Definice: Transakce přenáší databázi z jednoho konzistentního stavu do druhého.

Např.: Po provedení transakce musí být zachována všechna pravidla integrity (např. zůstatek na účtu nemůže být záporný)

#### **Izolace (Isolation)**

Definice: Současně probíhající transakce nesmí ovlivňovat jedna druhou.

Např.: Pokud dvě transakce provádějí změny na stejných datech, každá z nich uvidí databázi v takovém stavu, jako by byla jedinou aktivní transakcí

#### **Trvalost (Durability)**

Definice: Po potvrzení transakce (commit) musí změny přežít jakýkoli systémový výpadek.

Např.: Po potvrzení transakce záznamu o převodu peněz zůstane tento záznam zachován i po restartu databázového serveru.

### **Typy konfliktů**

#### **Ztracená aktualizace (Lost Update)**

Definice: Dvě transakce čtou stejný záznam a poté ho obě aktualizují, což způsobí, že jedna aktualizace je přepsána

Řešení: Použití zamykání (locking) nebo verze (versioning)

#### **Dirty Read**

Definice: Transakce čte data, která byla změněna jinou neukončenou transakcí

Řešení: Nastavení vyššího stupně izolace

#### **Non-repeatable Read**

Definice: Transakce čte stejné řádky vícekrát a pokaždé získává různé výsledky kvůli mezitímním aktualizacím jinou transakcí

Řešení: Použití stupně izolace REPEATABLE READ

#### **Phantom Read**

Definice: Transakce čte sadu řádků podle určité podmínky a při opakování čtení zjistí, že přibyly nové řádky odpovídající též podmínce

Řešení: Použití stupně izolace SERIALIZABLE

## **Stupně izolace**

### **1. Read Uncommitted**

Nejnižší úroveň, kde transakce může číst neukončené změny jiných transakcí.

Výhody: Nejvyšší výkon

Nevýhody: Dirty read, non-repeatable read, phantom reads

### **2. Read Committed**

Transakce mohou číst pouze potvrzené změny

Výhody: Zabraňuje dirty reads

Nevýhody: non-repeatable read, phantom reads

### **3. Repeatable Read**

Transakce vidí konzistentní stav dat od začátku do konce transakce, neumožňuje non-repeatable read

Výhody: Zabraňuje dirty reads a non-repeatable reads

Nevýhody: Phantom reads

### **4. Serializable**

Nejvyšší úroveň izolace, kde transakce jsou zcela izolovány a provádějí se sériově

Výhody: Zabraňuje všem typům anomalií

Nevýhody: Nejnižší výkon, vysoké nároky na zdroje

## **28. NoSQL databáze (typy škálování, teorém CAP), typy NoSQL databází, dokumentově orientované NoSQL databáze, HDFS**

### **NoSQL databáze**

Nerelační DB – Namísto typické tabulkové struktury relační databáze obsahují NoSQL data v rámci jedné datové struktury

Tento návrh nerelační databáze nevyžaduje schéma, nabízí rychlou škálovatelnost pro správu velkých a typicky nestrukturovaných souborů dat

### **Typy škálování**

#### **a) Vertikální škálování (Scale Up)**

- Zvyšování kapacity jednoho serveru přidáváním více CPU, paměti nebo úložiště
- Jednodušší správa a využití stávajících aplikací, ale fyzická omezení růstu serveru, vyšší náklady na hardware

#### **b) Horizontální škálování (Scale Out)**

- Přidávání více serverů do clusteru pro zpracování dat
- Teoreticky neomezené škálování, vyšší dostupnost a spolehlivost, ale složitější správa a koordinace dat, potřeba rozdělování dat (sharding)

### **Teorém CAP**

Říká, že v distribuovaném systému lze dosáhnout pouze dvou ze tří vlastností (Consistency, Availability, Partition Tolerance) současně.

#### **C (Consistency)**

- Definice: Každý čtení z DB po provedení zápisu získá nejnovější hodnotu
- Např.: Po aktualizaci hodnoty by každé následující čtení mělo vrátit aktualizovanou hodnotu

#### **A (Availability)**

- Definice: Každý požadavek na čtení nebo zápis obdrží odpověď, i když dojde k výpadku části systému
- Např.: Systém by měl vracet výsledky i v případě výpadku některých uzlů

#### **P (Partition Tolerance)**

- Definice: Systém pokračuje v práci i přes ztrátu komunikace mezi uzly (sítové rozdělení)
- Např.: I když jsou některé části sítě nedostupné, databáze by měla pokračovat v operacích

### **Typy NoSQL databází**

#### **1. Dokumentové databáze**

(např. MongoDB)

Ukládání a dotazování na dokumenty (JSON, XML)

Méně vhodné pro komplexní transakce

#### **2. Sloupcové databáze**

(Apache Cassandra, HBase)

Analýza velkých objemů dat, OLAP

Efektivní pro dotazy na velké objemy dat, vysoký výkon pro zápisy

#### **3. Key-Value databáze**

(Redis, DynamoDB)

Ukládání jednoduchých datových struktur (hashmapy)

Extrémně rychlé operace, jednoduchá škálovatelnost, ale omezené možnosti dotazování

## **4. Grafové databáze**

(Neo4j, Amazon Neptune)

Analýza a dotazování na vztahy mezi entitami

Optimalizované pro operace na grafech, výkonné dotazy na vztahy

## **Dokumentově orientované NoSQL databáze**

### **MongoDB**

Ukládání dat ve formátu JSON-like dokumentů (BSON).

Flexibilní schéma, vhodné pro rychle se měnící data, má horizontální škálování a replikační sady pro vysokou dostupnost, ale podpora složitých transakcí

### **CouchDB**

Ukládání dat ve formátu JSON, MapReduce pro dotazy.

*MapReduce* = Framework pro zpracování paralelizování úkolů na velkých datech

Využití pro offline první aplikace a synchronizace dat mezi klienty

Výhody: Vestavěná replikace a synchronizace, jednoduchý HTTP API

Nevýhody: Složitější dotazování než SQL-based systémy

### **HDFS (Hadoop Distributed File System)**

= Distribuovaný souborový systém (jako Linux filesystem) navržený pro běh na distribuovaných uzlech (počítačích)

Vlastnosti:

- Škálovatelnost: Efektivně škáluje na tisíce uzelů
- Spolehlivost: Automatická replikace dat pro zajištění odolnosti proti chybám
- Výkonnost: Optimalizován pro vysokou propustnost při zpracování velkých souborů
- Integrace: Úzce integrovaný s Hadoopovým ekosystémem (MapReduce, YARN)

Výhody: Velká kapacita, odolnost proti chybám, nákladová efektivita

Nevýhody: Nevhodnost pro malé soubory, zpoždění při čtení/zápisu

## **29. Proces dobývání znalostí, fáze metodologie CRISP-DM, názvosloví (datová matic, prediktor, cílová proměnná). Typy data miningových úloh a jejich příklady. Supervizované a nesupervizované učení**

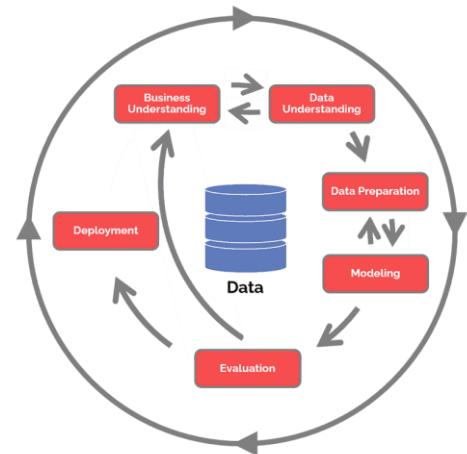
### **Metodologie CHRISP-DM**

= (Cross-Industry Standard Process for Data Mining) procesní model pro datovou analýzu, který zahrnuje šest hlavních fází: obchodní porozumění, porozumění datům, příprava dat, modelování, vyhodnocení a nasazení, sloužící jako strukturovaný přístup pro provádění dataminingových projektů

### **Fáze**

#### **Fáze 1: Business Understanding (Fáze analytická)**

- Porozumění problematice
- Nejdůležitější fáze (80 % významu, 20 % času)
- Zde definujeme, co vlastně budeme dělat a plánujeme celý projekt z manažerského hlediska
- Manažerská formulace je zde převedena do zadání úlohy pro dobývání znalostí
- Stanovíme si tzv. kritéria úspěšnosti, tzn. co očekáváme od daného projektu



#### **Fáze 2: Data Understanding (Fáze analytická)**

- Porozumění datům
- Prvotní sběr dat a vytvoříme si zde základní představu o datech
- Vytipování zajímavých podmnožin záznamů v databázi
- Zjišťují se zde deskriptivní charakteristiky dat pro různé atributy (četnosti, průměry...)

#### **Fáze 3: Data Preparation (Fáze výkonná)**

- Příprava dat
- Časově nejnáročnější fáze
- Vytváříme tzv. modelovací matici
  - o jedna tabulka, která je následně vhodná pro DM modely
- Modely z matice extrahují řešení problému
- Tato fáze determinuje úspěch v dalších fázích

#### **Fáze 4: Modeling (Fáze výkonná)**

- o Vytváření predikčních modelů – je vytvářen jeden nebo více modelů
- o Nástroje umožňují velké změny parametrů – je nutné vždy důkladně zaznamenat všechny nastavené hodnoty
- Nezbytnou součástí této fáze je ocenění modelů

#### **Fáze 5: Evaluation (Fáze výkonná)**

- Tento krok hodnotí úroveň, s jakou model dosahuje obchodních cílů
- Snaží se určit, zda je přítomen nějaký důvod (obchodní), proč je tento model nedostatečný
- Vytvořený model je možné ohodnotit tím způsobem, že jej užijeme na reálné situace a sledujeme jeho kvalitu. Je však nutné zvážit časové a rozpočtové podmínky, zda umožňují takovéto hodnocení

#### **Fáze 6: Deployment (Fáze výkonná)**

- Nasazení do praxe
- Časově velmi rozdílná projekt od projektu

**Datová matice** = je struktura, která organizuje data do formátu tabulky, kde řádky reprezentují jednotlivé záznamy (případy) a sloupce reprezentují různé atributy (vlastnosti) těchto záznamů

**Prediktor** = proměnná nebo model používaný k předpovídání hodnot závislé proměnné na základě hodnot jedné nebo více nezávislých proměnných

**Cílová proměnná** = (závislá proměnná) je proměnná, kterou se snažíme předpovědět nebo vysvětlit v rámci modelu prediktivní analýzy, na základě hodnot nezávislých proměnných (prediktorů)

## **Typy data miningových úloh a jejich příklady**

### **1. Klasifikace**

= cílem je přiřadit objekty do předem definovaných kategorií nebo tříd na základě atributů

Např.: klasifikace e-mailů jako "spam" nebo "ne-spam" dle atributů předmět, obsah e-mailu, odesílatel

### **2. Regrese**

= cílem je předpovědět kontinuální (kam se bude ubírat) hodnotu na základě vstupních atributů

Např.: předpověď ceny domu na základě atributů pozemku, počet pokojů, lokalita

### **3. Shlukování (Clustering)**

= cílem je rozdělit sadu objektů do skupin (shluků) tak, aby objekty ve stejném shluku byly podobné

Např.: segmentace zákazníků do různých skupin podle jejich nákupního chování

### **4. Asociační pravidla**

= cílem je najít zajímavé vztahy mezi položkami ve velkých datových souborech

Např.: zjištění, že zákazníci, kteří koupí chleba, často koupí také máslo (analýza košíku)

### **5. Anomální detekce (Anomaly Detection)**

= cílem je identifikovat neobvyklé vzory, které neodpovídají očekávanému chování

Např.: detekce podvodných transakcí na kreditních kartách

### **6. Redukce dimenzionality**

= cílem je snížit počet proměnných, které popisují data, při zachování podstatných informací

Např.: použití PCA (Principal Component Analysis) k redukci počtu atributů v datasetu pro zjednodušení modelování

## **Supervizované učení (Supervised Learning)**

= Typ učení, kde model trénuje na značených datech, tedy datech, která mají vstupní atributy a odpovídající cílové hodnoty (labely)

Úlohy: Klasifikace, regrese

Příklady algoritmů:

- Klasifikační algoritmy: Decision Trees, Random Forest, Support Vector Machines (SVM), Neural Networks

- Regresní algoritmy: Linear Regression, Ridge Regression, Lasso Regression

## **Nesupervizované učení (Unsupervised Learning)**

= Typ učení, kde model trénuje na neznačených datech, tedy datech, která mají pouze vstupní atributy a žádné cílové hodnoty

Úlohy: Shlukování, asociační pravidla, redukce dimenzionality

Příklady algoritmů:

- Shlukovací algoritmy: K-means, Hierarchical Clustering, DBSCAN

- Algoritmy pro asociační pravidla: Apriori, Eclat

- Algoritmy pro redukci dimenzionality: PCA, t-SNE, LDA (Linear Discriminant Analysis).

## **30. Asociační algoritmy – hledání asociačních pravidel, algoritmus Apriori (frekventovaná množina, statistiky implikací). Seskupovací algoritmy – dělení, využití, typy úloh, standardizace atributů, hodnocení podobnosti objektů**

### **Asociační algoritmy**

#### **Hledání asociačních pravidel**

Asociační algoritmy jsou používány k nalezení vztahů mezi proměnnými ve velkých datových souborech. Vztahy reprezentovány jako asociační pravidla.

- **Asociační pravidlo:** Vyjadřuje vztah mezi položkami v datovém souboru ve formě "Pokud X, pak Y", kde X a Y jsou položky nebo sady položek.
- **Podpora (Support):** Frekvence, s jakou se pravidlo vyskytuje v datovém souboru. Vyjadřuje se jako poměr počtu výskytů X U Y k celkovému počtu transakcí.
- **Důvěra (Confidence):** Míra pravděpodobnosti, že pokud nastane X, nastane i Y. Vyjadřuje se jako poměr počtu výskytů X U Y k počtu výskytů X.
- **Ziskovost (Lift):** Míra toho, jak je pravidlo zajímavé. Vyjadřuje se jako poměr důvěry pravidla k poměru výskytu Y v celém datovém souboru.

#### **Algoritmus Apriori**

= algoritmus pro hledání frekventovaných množin a generování asociačních pravidel.

1. Frekventovaná množina (Frequent Itemset): Množina položek, která se vyskytuje v datovém souboru s frekvencí vyšší než minimální prahová hodnota (support threshold).

2. Statistiky implikací Používají se k hodnocení kvality asociačních pravidel (např. podpora, důvěra, ziskovost).

Postup algoritmu Apriori:

1. Generování kandidátů
2. Pruning (Prořezávání): z hotového stromu se odstraní málo významné větve (podstromy)
3. Výpočet podpory
4. Výběr frekventovaných množin
5. Opakování: Opakování kroků 1-4, dokud neexistují žádné další frekventované množiny.

### **Seskupovací algoritmy**

(Clustering) používají se k seskupování objektů do skupin na základě jejich podobnosti

#### **1. Hierarchické shlukování:**

- Aglomerativní (bottom-up): Každý objekt začíná jako vlastní shluk a shluky se postupně spojují
- Divisivní (top-down): Začíná jedním shlukem a postupně se rozděluje

#### **2. Partition-based shlukování:**

- K-means: Rozděluje objekty do K shluků tak, aby byla minimalizována suma čtverců vzdáleností mezi objekty a centrem shluku

#### **3. Density-based shlukování:**

- DBSCAN: Identifikuje shluky na základě hustoty datových bodů.

#### **4. Model-based shlukování:**

- Expectation-Maximization (EM): Využívá pravděpodobnostní modely k určení shluků.

### **Využití a typy úloh**

Segmentace trhu, bioinformatika, obrazová analýza

## **Standardizace atributů**

Před použitím je často nutné standardizovat atributy, aby měly stejné váhy. Pomocí:

- a) Normalizace: Převedení hodnot na stejný rozsah (např. [0,1])
- b) Z-skóre: Převedení hodnot na standardní rozdělení s průměrem 0 a směrodatnou odchylkou 1

## **Hodnocení podobnosti objektů**

a) Euklidovská vzdálenost: Měří přímou vzdálenost mezi dvěma body v prostoru v  $n$ -rozměrném prostoru. Například u obrazových dat

b) Hammingova vzdálenost: Používá se pro porovnávání binárních řetězců, kde se počítá počet pozic, na kterých se dva řetězce liší.

c) Kosinusová podobnost: Měří kosinus úhlu mezi dvěma vektory, často používané u textových dat.

d) Míra korelace

### **31. Klasifikační algoritmy, predikce vycházející z historických dat, princip a typy rozhodovacích stromů, algoritmus CHAID, evaluace DM modelů (matice záměn, graf senzitivity a specifičnosti, ROC křivka), algoritmická rozšíření rozhodovacích stromů.**

Klasifikační algoritmy jsou nástroje používané k přiřazování objektů do jedné z předem definovaných kategorií nebo tříd na základě vstupních atributů.

#### **Predikce vycházející z historických dat**

Využití historických dat k vytvoření modelů, které dokáží předpovídat budoucí hodnoty nebo kategorie.

Např.: Predikce, zda zákazník koupí produkt na základě jeho minulého chování.

#### **Princip a typy rozhodovacích stromů**

Rozhodovací stromy jsou grafické reprezentace rozhodovacích procesů, kde každý uzel představuje test na atribut, každá větev odpovídá výsledku testu a každý listový uzel představuje třídu nebo hodnotu.

1. Binární stromy: Každý uzel má maximálně dvě děti.
2. Vícehodnotové stromy: Uzel může mít více než dvě děti.

#### **Algoritmus CHAID (Chi-squared Automatic Interaction Detector)**

používaný pro analýzu kategoriálních dat a výběr optimálních rozdělení atributů

Postup:

1. Sloučení kategorií: Kombinace kategorií atributů, které nejsou statisticky významně odlišné.
2. Výběr nejlepšího atributu: Atribut s nejmenší hodnotou chí-kvadrátu je vybrán pro rozdělení uzlu.
3. Rozdělení uzlu: Uzly se rozdělí podle vybraného atributu.
4. Opakování: Postup se opakuje, dokud nejsou splněny zastavovací podmínky.

#### **Evaluace DM modelů**

##### **Matice záměn (Confusion Matrix)**

Zobrazuje počet správných a nesprávných předpovědí rozdělených podle skutečných tříd

- True Positive (TP): Správně předpovězené pozitivní případy
- True Negative (TN): Správně předpovězené negativní případy
- False Positive (FP): Nesprávně předpovězené pozitivní případy
- False Negative (FN): Nesprávně předpovězené negativní případy

		Predikce	
		Pozitivní (PP)	Negativní (PN)
Realita	Pozitivní (P)	TP	FN
	Negativní (N)	FP	TN

## Graf senzitivity a specifičnosti

= závislost senzitivity na specifičnosti

*Senzitivita (Recall):* Podíl správně identifikovaných pozitivních případů ( $TP / (TP + FN)$ )

*Specifičnost:* Podíl správně identifikovaných negativních případů ( $TN / (TN + FP)$ )

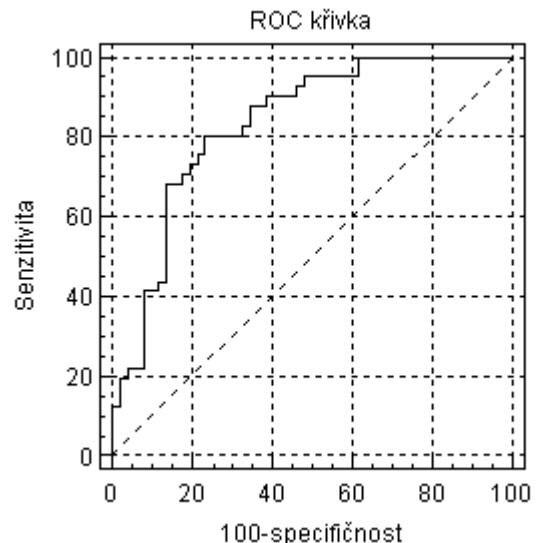
## ROC křivka (Receiver Operating Characteristic)

= grafické znázornění výkonnosti klasifikačního modelu při různých prahových hodnotách

- Osa y: True Positive Rate (Senzitivita)

- Osa x: False Positive Rate ( $1 - \text{Specifičnost}$ )

- *AUC (Area Under the Curve):* Hodnota vyjadřující celkový výkon modelu. Vyšší hodnota znamená lepší model



## Algoritmická rozšíření rozhodovacích stromů

1. Random Forest: Kombinuje mnoha rozhodovacích stromů, kde každý strom je trénován na náhodném podvzorku dat. Výsledná predikce je průměrem predikcí všech stromů

2. Gradient Boosting: Iterativně vytváří modely, kde každý nový strom opravuje chyby předchozích stromů

3. XGBoost: Efektivní a optimalizovaná implementace gradient boosting algoritmu

## 32. Jazyk XML – základní principy a pravidla. Definice jazyka a validace dokumentu.

### Jazyk XML (eXtensible Markup Language)

= metajazyk, definuje základní syntaktická pravidla dokumentů a nástroje pro popis prvků jazyka a jejich vztahů (univerzální, standardizován WWW)

Nic neříká o sémantice (významu), ten musí znát konkrétní aplikace

#### Struktura

- a) Prolog – verze, ... (např. <?xml version="1.0" encoding="UTF-8"?>)
- b) Kořenový prvek
- c) Běžné prvky

**Elementy (tagy, prvky)** = základní stavební bloky XML

**Atribut** = obsahuje doplňkové informace o prvku; zápis do zahajujících značek (<cd id="disk1">...</cd>)

Dále lze vkládat **entity** (např. &gt; je >), komentáře (<!-- text komentáře -->)

**Správnost XML dokumentu:** Dokument musí být "**well-formed document**", což znamená, že dodržuje syntaktická pravidla XML – má jeden kořenový prvek, správně ukončené prvky a správně vnořené prvky

**Validní dokument:** Kromě toho, že je správně strukturovaný, musí struktura prvků odpovídat definici jazyka, aby bylo zaručeno korektní zpracování v aplikacích

### Definice jazyka a validace dokumentu

#### 1. Definice jazyka pomocí DTD (Document Type Definition):

**Definice jazyka (DTD):** Document Type Definition určuje, jaké existují prvky, co mohou obsahovat (vzájemné vnořování) a jaké mají atributy, a zavádí obecnou strukturu dokumentu

Neumí datové typy

- a) Definice obsahu:  
Pomocí <!ELEMENT jméno obsah>, kde obsah je EMPTY, ANY nebo konkrétní text, nebo pomocí
- b) Definice atributu  
Pomocí <!ATTLIST prvek jméno typ implicit\_hodnota> (*pozn. typ je charakter, nikoliv datový typ*)  
Atributů jsou různé typy: ID, odkazy, konkrétní hodnoty, tokeny, ...

**Připojení DTD ke XML:** DTD může být odkazováno jako externí soubor nebo uvedeno přímo v XML souboru, což umožňuje validaci struktury dokumentu.

Pozn.:

Kompletní identifikace prvku pomocí: **jmenný prostor** + prvek

- **Jmenný prostor:** je identifikován lokátorem (URI) = jméno prostoru, zahrnuje jména prvků a atributů; jméno (URI) příliš dlouhé, bývá definována zkratka (prefix) pro zařazení identifikátoru do jmenného prostoru
- **Kvalifikované jméno** = jméno, které je interpretováno z hlediska příslušnosti ke jmennému prostoru
  - jména prvků v XML dokumentu
  - bez prefixu (např. h1) – náleží do implicitního jmenného prostoru, závisí na kontextu
  - s prefixem (např. html:h1) – jmenný prostor je určen prefixem

#### 2. Definice jazyka pomocí XML Schema Definition (XSD):

XSD je pokročilejší než DTD a umožňuje definovat datové typy elementů a atributů.

Definuje strukturu XML dokumentu s přesností na typy dat

## **Jmenný prostor** xmlns:xsd="http://www.w3.org/2001/XMLSchema"

Výrazově dost silné, ale kritizováno za složitost

Řada předdefinovaných typů, ale hlavní síla ve schopnosti definovat si vlastní typy

*Typy základní:* různá čísla (celá, desetinná), řetězce, ...

*Nové typy:* list, union (spojení několika typů), restrikce (např. délky řetězce) ...

*Složené datová typy* (= definují vzájemné vztahy jednotlivých prvků): dané pořadí, volné pořadí, možnost volby

Celé schéma lze připojit k XML

### **Tři základní druhy:**

#### **1) Matrjoška**

- definice prvků se do sebe ve schématu vkládají přímo, stejně jako prvky v dokumentu
- na nejvyšší úrovni vždy jen jeden prvek
- krátké a kompaktní schéma, ale pro složitější jazyky nepřehledné a nelze opakovaně využívat dílčí definice

#### **2) Plátkování**

- všechny prvky a atributy se definují na stejně (nejvyšší) úrovni
- odkazují se na sebe pomocí ref="jméno"
- definované prvky/atributy lze používat opakovaně
- koncepce blízká DTD
- obsah prvku se nemůže lišit podle kontextu (rodiče)

#### **3) Slepý Benátčka**

- předem se (na globální úrovni) definují typy
- prvky a atributy se definují lokálně (jako v matrjošce), ovšem triviálně s využitím připravených typů
- nejflexibilnější, vhodné pro složité jazyky
- typy lze používat opakovaně, ale schéma je delší

### **33. Strom dokumentu, XPath a jeho základní konstrukce (cesta, krok, osa, podmínka)**

#### **Strom Dokumentu XML**

XML dokument je strukturován jako strom, který se nazývá Document Object Model (DOM)

Tento strom reprezentuje hierarchickou strukturu dokumentu, kde každý prvek je uzel a vztahy mezi prvky jsou reprezentovány jako větve stromu.

#### **Typy uzelů:**

- **Korén** – uměle přidaný
- **Prvek** – odpovídají prvkům dokumentu
- **Text** – textový obsah, bezejmenné, vždy listové
- **Atribut** – připojen k prvku, který jej nese (ten je jeho rodičem, ale atribut není považován za dítě)
- **Jmenný prostor** – název=prefix, dědí se
- **Instrukce pro zpracování** – názvem je její cíl
- **Komentář** – bezejmenný

#### **Vztahy prvků:**

A je předkem B, B je potomkem A:

- B je obsažen v A

A je rodičem B, B je dítětem A:

- B je přímo obsažen v A (je ve struktuře právě o jednu úroveň níže)
- Rodič je vždy právě jeden

A je sourozencem B:

- mají stejněho rodiče

#### **XPath**

= nástroj pro vyhledávání informací v XML dokumentech – identifikaci uzelů a jejich skupin ve stromě (prvků, atributů, ...)

Řada (více než sto) vestavěných funkcí

Není samostatný jazyk – definuje syntax výrazů a je používán jako součást dalších mechanismů (zejména XSLT, XQuery)

Připomíná cestu k systémovému souboru

Výsledkem XPath výrazu je hodnota nebo skupina uzelů XML stromu vyhovujících podmínkám (interpretace začíná ve výchozím uzlu)

XPath **cesta** je sekvenčí kroků, oddělovány lomítky (*krok1/krok2/...*)

Každý **krok** může mít tři části:

- 1) **identifikátor osy** – směr procházení, výchozí množina
  - 2) test uzlu – kterých uzelů se týká, povinná část
  - 3) **podmínka** (predikát) – zužuje výsledky předchozího výběru
- plný tvar: *osa::uzel[podmínka]*

Oddělovače:

- a) Znak / prostý oddělovač
- b) Dvojice // mezi uzelů se může nacházet libovolný počet mezilehlých

Hledat lze dle: názvu, typu, vztahů, atributů

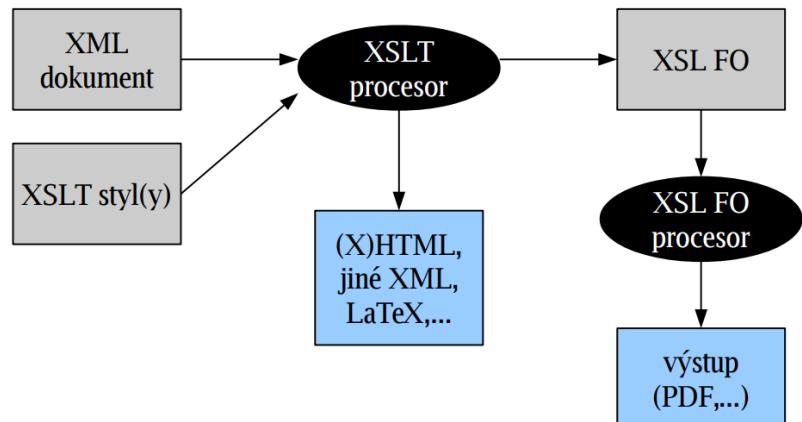
Také lze užívat funkce (např. last(), ...), podmínky (např. contains(kde, co), ...), či logické funkce

## 34. Principy transformace pomocí XSLT, šablony, vytváření prvků a atributů, využívání hodnot z dokumentu.

### XSL (eXtensible Stylesheet Language)

Dvě části:

- a) XSL FO (XSL Formatting Objects) – jazyk definující formátovanou podobu dokumentu; po zpracování XSL FO procesorem vede k finální podobě
- b) XSLT (XSL Transformations) – jazyk pro transformaci XML dokumentů, do XML FO, ale i jiných formátů



### XSLT

XML jazyk

Slouží k transformaci dokumentu:

- a) změna struktury a/nebo prvků
- b) přeuspořádání, přidávání, výběr informací

Výstupní formáty: XML (např. XHTML, XSL FO, ale i XML data pro jinou aplikaci), HTML, text

**XSLT procesor** = program implementující XSLT

Interně transformuje stromy – uplatňováním **šablon** na vstupní strom vytváří výstupní strom

**XSLT styl** = sada šablon definujících transformaci

Pro použití přidat: jmenný prostor XSLT je <http://www.w3.org/1999/XSL/Transform>  
celý styl je obalen prvkem stylesheet (nebo transform – synonymum)

### XSLT Šablona

Je třeba zadat na co vše bude v dokumentu použit (match="") a co vloží do výstupního stromu

Pomocí XPath lze udělat for cyklus, který bude aplikovat šablony na vše vyhovující

V šablonách lze nastavit řazení (sorting) a aplikaci šablon, jen je-li splněna podmínka (T/F)

Lze také definovat proměnné (pro znovu používání lokálně/globálně)

### Implicitní šablona

- provedena pro prvky, jež nevyhovují žádné šabloně
- opíše do výstupu textový obsah prvku
- rekurzivně prochází a transformuje jejich obsah

```
<xsl:template match="vzor">  
    obsah_šablony  
</xsl:template>
```

### Prvky a atributy

#### Generování hodnot prvků:

a) opisováním – obsahuje-li šablona prvky z jiného jmenného prostoru (než prostor XSLT), opíší se do výstupního stromu

b) pomocí xsl:element – jméno prvku je hodnotou atributu a tu lze vytvořit podle obsahu dokumentu

#### Generování atributů:

a) opisováním – atributy opisovaných prvků lze zapsat přímo (viz align)

- hodnotu lze vytvořit pomocí {}

b) pomocí xsl:attribute `<xsl:attribute name="jméno">hodnota</xsl:attribute>`

- umožňuje vytvořit i jméno atributu

**Kopírování částí** – pokud chceme převzít celou část původního dokumentu (`xsl:copy`)

**Komentáře a instrukce** – umožňuje generovat nové i kopírovat staré

Lze připravit i *sady atributů*, které budou přiřazovány více prvkům (*use-attributes-sets*)

Končený výstup a jeho náležitosti pomocí *xsl:output*

### **Využívání hodnot z dokumentu**

V attributech {výraz}

- vyhodnotí výraz a přiřadí výsledek jako hodnotu atributu

V těle prvků *<xsl:value-of select="výraz"/>*

- vyhodnotí výraz a výsledek vloží na místo svého použití

## **35. Základní architektury počítačů, architektury mikroprocesorů, architektury signálových a grafických procesorů, architektury mikrořadičů. Principy činnosti významných funkčních bloků v jednotlivých architekturách.**

Pojmy:

**Sběrnice** = komunikační systém, který přenáší data mezi různými komponenty počítače, jako jsou procesor, paměť a periferní zařízení

**UART (Universal Asynchronous Receiver/Transmitter)** je hardwarový modul, který umožňuje asynchronní sériovou komunikaci mezi počítači a periferními zařízeními tím, že převádí data mezi paralelním a sériovým formátem

**Řadič** = hardwarový nebo softwarový modul, který řídí přenos dat mezi různými částmi počítačového systému nebo mezi počítačem a jeho periferiemi

### **Základní Architektury Počítačů**

#### **1. Von Neumannova Architektura**

- Paměť: Společná paměť pro programy a data.
- CPU: Centrální procesorová jednotka (CU + ALU)
- Instrukční cyklus: Načtení instrukce, dekódování, provedení, ukládání výsledku
- Data Bus: Jednotná sběrnice pro přenos dat a instrukcí

#### **2. Harvardská Architektura**

- Paměť: Oddělená paměť pro programy a dat
- Sběrnice: Samostatné sběrnice pro instrukce a data
- Výhoda: Možnost paralelního načítání instrukcí a dat, což zvyšuje výkon

### **Architektury Mikroprocesorů**

#### **1. CISC (Complex Instruction Set Computing)**

- Instrukce: Velký počet složitých instrukcí
- Komplexita: Složitý dekodér instrukcí
- Příklad: Intel x86
- Výhody: Jednodušší programování, některé složité operace lze provést jednou instrukcí

#### **2. RISC (Reduced Instruction Set Computing)**

- Instrukce: Omezený počet jednoduchých instrukcí
- Komplexita: Jednodušší a rychlejší dekodér instrukcí
- Příklad: ARM (často v embedded systémech), MIPS
- Výhody: Vyšší výkon díky zjednodušenému dekódování a rychlejšímu provádění instrukcí

### **Architektury Signálových a Grafických Procesorů**

#### **1. DSP (Digital Signal Processors)**

- Účel: Specializované pro zpracování signálů v reálném čase
- Funkce: Rychlé násobení a akumulace, specializované paměťové architektury

#### **2. GPU (Graphics Processing Unit)**

- Účel: Specializované na paralelní zpracování grafických dat.
- Architektura: Masivně paralelní výpočetní jednotky (CUDA cores – od NVIDIA, Stream Processors)
- Příklad: NVIDIA, AMD

- Výhody: Vysoký výkon při grafických operacích, general-purpose computing (GPGPU), ale i učení neuronových sítí

## Architektury Mikrořadičů

Mikrořadiče (Microcontrollers, MCU, jednočipový počítač)

= integrovaný obvod obsahující kompletní mikropočítač, obsahující CPU, paměť a vstupně-výstupní periférie na jednom čipu

- Integrace: CPU, paměť (RAM, ROM/Flash), I/O porty na jednom čipu
- Účel: Řízení vestavěných systémů a jednoduchých aplikací
- Příklad: Atmel AVR, ARM Cortex-M, PIC
- Funkce: Nízká spotřeba energie, široká škála periferií (ADC, PWM, UART)

## Principy Činnosti Významných Funkčních Bloků

### 1. CPU (Central Processing Unit)

- CU (Control Unit): Řídí operace CPU, dekóduje instrukce
- ALU (Arithmetic Logic Unit): Provádí aritmetické (sčítání, násobení) a logické operace (AND, OR, XOR, NOT)
- Registry: Rychlá paměť pro dočasné ukládání dat a instrukcí

### 2. Paměťové Bloky

- RAM (Random Access Memory): Dočasná paměť pro data a instrukce
- ROM (Read-Only Memory): Trvalá paměť pro firmware
- Cache: Rychlá paměť mezi CPU a RAM pro zrychlení přístupu k často používaným datům

### 3. I/O (Input/Output)

- Periférie: Zařízení pro komunikaci s vnějším světem (klávesnice, displeje, senzory)
- Řadiče: Spravují vstupní a výstupní operace, jako jsou UART, SPI, I2C

### 4. DSP (Digital Signal Processor) Bloky

= zahrnují specializované jednotky jako MAC, které provádějí rychlé násobení a akumulaci, a také více paměťových bank pro paralelní přístup k datům, což umožňuje efektivní zpracování signálů v reálném čase

- Bloky optimalizované pro signálové operace (filtrování, Fourierova transformace)
- MAC (Multiply-Accumulate Unit): Specializovaná jednotka pro rychlé násobení a akumulaci.
- Specializované paměti: Více paměťových bank pro paralelní přístup.

### 5. GPU Bloky

- Shader Units: Paralelní jednotky pro výpočet grafických stínů a efektů
- Frame Buffer\*\*: Paměť pro ukládání grafických rámců před zobrazením

## 36. Hodnocení výkonnosti počítačů, Amdahlův zákon, výkonnostní rovnice procesoru. Srovnání systémů CISC a RISC.

### Hodnocení Výkonnosti Počítačů

**Výkonnost** = převrácená hodnota doby vykonání jednoho úkonu

**Propustnost** = počet úkonů za jednotku času (množství vykonané práce za jednotku času)

**Měření na zkušebních úlohách** – závislé na typu úlohy

**SPEC (Standard Performance Evaluation Corporation)** = všeobecný test, různé úlohy

**Instrukční mixy** = měří rychlosť různých druhů instrukcí

**Odezva systému** = měří se doba odezvy na uživatelské vstupy

**Spotřeba energie** = měří se energetická efektivnost systému

### Amdahlův Zákon

= analytický vztah sloužící k posuzování účinku paralelizace na výkon systému

Pozn.: V praxi časté střídání sériových a paralelních úseků, ne vždy se tedy vyplatí čistě jedna varianta

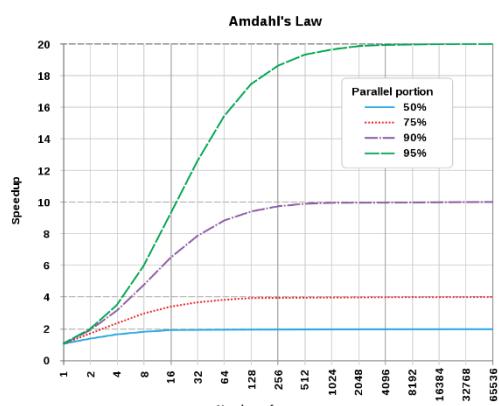
Cílem odvození je najít poměr mezi dobou zpracování na jednom procesoru a dobou zpracování při částečné paralelizaci

Tento poměr se nazývá součinitel zrychlení:  $Sz = \frac{T_{původní}}{T_{nový/optimální}}$

Po úpravě:  $Sz = \frac{1}{(1-P)+\frac{P}{S}}$

kde:  $P$  je poměr práce, která může být paralelizována

$S$  je rychlosť zlepšení paralelizované části



### Výkonnostní Rovnice Procesoru

Výkonnost CPU závisí na:

- počtu instrukcí ( $IC$  – Instruction Count)
- (průměrném) počtu taktů na instrukci ( $CPI$  – Cycles Per Instruction)
- periodě hodinového signálu ( $T_{clk}$ ) – doba cyklu (taktu)

$$T_{CPU} = IC \cdot CPI \cdot T_{clk}$$

(platí pro systémy bez cache)

### Srovnání Systémů CISC a RISC

	RISC	CISC
<b>Výhody</b>	Zjednodušená instrukční sada vede k rychlejšímu zpracování Pipe-lining může zvýšit výkon Nižší spotřeba energie Menší velikost čipu, což může vést k úspore nákladů	Schopnost provádět složité instrukce Programy vyžadují ke spuštění méně instrukcí Větší hardwarová podpora pro provádění složitých instrukcí
<b>Nevýhody</b>	Programy mohou vyžadovat více instrukcí k dokončení úkolu než CISC Omezená schopnost provádět složité instrukce	Zvýšená složitost může vést ke zpomalení doby zpracování Větší velikost čipu může vést ke zvýšení nákladů

<b>Srovnání</b>	RISC je ideální pro aplikace, které vyžadují rychlé a efektivní zpracování (např. mobilní zařízení a vestavěné systémy). CISC je vhodnější pro aplikace, které vyžadují složité operace (např. zpracování videa a obrazu)  Pozn.: Existují i hybridní architektury kombinující RISC a CISC
-----------------	---

## **37. Významné průmyslové komunikační sběrnice a protokoly. Sběrnice a protokoly v počítačových systémech – topologie, charakteristické vlastnosti**

**Sběrnice** = Informační cesta umožňující přenos informace mezi jednotlivými funkčními bloky systému nebo mezi systémem a okolím (složena z řadiče a jemu podřízených slaves, fyzicky je to soustava vodičů přenášející data nebo signál)

### **Významné Průmyslové Komunikační Protokoly**

#### **1. Modbus**

- Otevřený komunikační protokol pro průmyslová zařízení (I/O rozhraní, dotykové displeje)
- Vlastnosti: Jednoduché a přímočaré, podpora seriového a Ethernetového přenosu dat, podpora v různých průmyslových odvětvích

#### **2. CAN (Controller Area Network)**

- Sítový protokol pro komunikaci v automobilovém průmyslu
- Navržen pro komunikaci mezi řídícími jednotkami ve vozidlech díky odolnosti vůči elektromagnetickému rušení

#### **3. IP (Internet Protocol)**

- Většinou přenášeno pomocí Ethernetu

### **Významné Průmyslové Komunikační Sběrnice**

#### **1. RS-232**

- Sériová komunikační sběrnice, přenos dat mezi zařízeními pomocí sériového kabelu
- Využití: připojení počítače k periferiím jako tiskárny, modemy nebo sériová zařízení, pomocí konektorů typu D-Sub

#### **2. I<sup>2</sup>C (Inter-Integrated Circuit)**

- Sběrnice pro sériovou komunikaci mezi integrovanými obvody (ICs), dva typy zařízení master a slave
- Dva oboustranné vodiče: SDA (Serial Data Line) pro datový přenos a SCL (Serial Clock Line) pro synchronizaci komunikace
- Často u částí mikrokontrolérů, jako jsou senzory, displeje nebo paměťová zařízení

#### **3. SPI (Serial Peripheral Interface)**

- Sériová či paralelní komunikační sběrnice, zařízení master a slave
- Čtyři druhy vodičů: SCLK (Serial Clock), MOSI (Master Output Slave Input), MISO (Master Input Slave Output) a SS (Slave Select)

#### **4. USB (Universal Serial Bus)**

- Standard pro sériový přenos; má 4, resp. 8 vodičů
- Umožnuje přenos dat, napájení zařízení a propojení periferních zařízení s počítačem (myší, klávesnice, tiskáren, diskových jednotek, ...)
- Víceúrovňová hvězdicová struktura
- Verze: USB 1.0, USB 2.0, USB 3.0 a USB 3.1 (podle verze rychlosť přenosu)

Několik dalších:

- PCI (Peripheral Component Interconnect): Standardní sběrnice pro připojení periferních zařízení k počítači
- PCI Express (PCIe): Moderní sériová sběrnice pro připojení grafických karet, SSD disků a dalších periferií k počítači

- SATA (Serial ATA): Sběrnice pro připojení pevných disků a optických mechanik k počítači
- Ethernet: Sběrnice pro přenos dat v počítačových sítích pomocí kabelu
- UART (Universal Asynchronous Receiver/Transmitter): Sběrnice pro asynchronní sériovou komunikaci mezi zařízeními

## Sběrnice a Protokoly v Počítačových Systémech

- FSB (Front Side Bus):** Systémová sběrnice pro komunikaci mezi CPU a základní deskou. Má nižší rychlosť než interní frekvence jádra procesoru a přenáší data na obě hrany hodinového signálu. Propustnost může dosáhnout až 12,8 GB/s.
- QPI (QuickPath Interconnect):** Sběrnice od Intelu, která nahrazuje FSB. Skládá se ze dvou jednosměrných sběrnic a umožňuje rychlý přenos dat s propustností až 25,6 GB/s.
- IF (Infinity Fabric):** Proprietární sběrnice od AMD, která slouží pro přesuny dat na úrovni čipu nebo mezi čipy. Je škálovatelná s datovou propustností od 30 do 512 GB/s a kombinuje datovou a řídicí sběrnici.
- P-ATA (IDE, ATA):** Paralelní rozhraní pro připojení pevných disků a dalších zařízení s maximální přenosovou rychlosťí 133 MB/s a používá 40- nebo 80-žilový kabel.
- PCI (Peripheral Component Interconnect):** Paralelní sběrnice používaná pro připojení periferních zařízení k základní desce počítače. Má modulární strukturu a umožňuje připojení více zařízení.
- S-ATA (Serial ATA):** Sériové rozhraní pro připojení pevných disků, které nabízí vyšší přenosovou rychlosť (až 600 MB/s pro S-ATA III) a používá tenčí kably, což umožňuje lepší organizaci kabeláže uvnitř počítačových skříní.
- M.2:** Nové rozhraní pro připojení SSD, WiFi modulů a dalších komponent. Nabízí velkou propustnost a malou latenci, a to bez použití kabelů, což zvyšuje spolehlivost připojení. M.2 socket 3 má rychlosť PCIe x4 s propustností 4 GB/s.

## Topologie Sběrnic

Základní dělení na dvoubodové a vícebodové spojení

Dále pak na:

- Lineární (Line Bus): Zařízení jsou připojena na jednu linku
- Stromová (Tree Bus): Vytváří hierarchickou strukturu s hlavní a vedlejší linkami
- Hvězdicová (Star Bus): Zařízení jsou připojena k centrálnímu bodu (např. switch nebo hub)
- Kruhová (Ring Bus): Zařízení jsou propojena do kruhu, kde každé má dvě sousední

## Charakteristické Vlastnosti Sběrnic a Protokolů

Sběrnici lze definovat z hlediska:

- mechanického (tvar a rozměry mechanických dílů, konektorů) – maximální délka sběrnice, počet přípojných míst (zatížení), přenosová rychlosť (kapacita), impedance sběrnice (odpor), napěťové úrovně, konektory
- elektrického (úrovně U a I, přiřazení logických stavů)
- funkčního (význam jednotlivých signálů a jejich časové průběhy)
- operačního (informační kódy a formáty přenášených zpráv)

Přenos dat sběrnicí:

- Synchronní – společně s daty se vysílají synchronizační impulsy, příp. speciální synchronizační vodič
- Asynchronní – dávkový přenos dat, přenosový rámec obsahuje synchronizační informace
- Arytmický přenos – kombinace asynchronního a synchronního

## **38. Single board computer – definice, součásti, třídy výkonnosti. SoC. SoM.**

**Operační systémy pro vestavná zařízení. Linux pro vestavná zařízení. Úpravy OS pro cílové zařízení. Paměti Flash, systémy souborů, podpora v OS.**

### **Single Board Computer (SBC)**

= kompletní počítač postavený na jediné desce plošných spojů, která zahrnuje mikroprocesor, paměť (RAM), vstupně/výstupní porty a další nezbytné komponenty pro plnou funkčnost počítače.

#### **Součásti:**

1. Mikroprocesor (CPU): Centrální procesorová jednotka, která vykonává instrukce
2. Paměť (RAM): Dočasná paměť pro uchovávání dat a běžících programů
3. Úložný prostor: Obvykle ve formě pamětí Flash nebo slotů pro SD karty
4. Vstupně/výstupní porty (I/O): Porty pro připojení periferních zařízení, jako jsou USB, HDMI, Ethernet
5. Napájecí jednotka: Systém pro napájení celé desky
6. Síťová rozhraní: Ethernet, Wi-Fi nebo Bluetooth pro síťovou komunikaci
7. Grafické výstupy: Porty pro připojení monitorů nebo displejů

#### **Třídy výkonnosti:**

##### **1. Low-end SBC:**

- Nízký výkon, základní funkce
- Příklad: Raspberry Pi Zero
- Použití: Jednoduché úlohy, vzdělávání, prototypování

##### **2. Mid-range SBC:**

- Střední výkon, více I/O portů a funkcí
- Příklad: Raspberry Pi 4, BeagleBone Black
- Použití: Vývoj aplikací, domácí automatizace, IoT projekty

##### **3. High-end SBC:**

- Vysoký výkon, pokročilé funkce a větší paměť
- Příklad: NVIDIA Jetson Nano, Odroid XU4
- Použití: Počítačové vidění, umělá inteligence, složité výpočetní úlohy

### **System on Chip (SoC)**

= integrovaný obvod, který obsahuje veškeré komponenty počítače jediném čipu

Obsahuje CPU, paměť, vstupně/výstupní porty a další funkční moduly

### **System on Module (SoM)**

= kompaktní modul obsahující SoC a potřebné komponenty (paměť, napájení, I/O porty)

Navržen tak, aby byl integrován do větších systémů a poskytoval kompletní výpočetní platformu

### **Rozdíl SoC a SoM**

Zatímco čip poskytuje základní výpočetní schopnosti, modul poskytuje kompletní výpočetní platformu připravenou pro okamžité použití v různých aplikacích.

# **Operační systémy pro vestavná zařízení**

## **Běžné OS:**

### **1. Embedded Linux:**

- Použití: Široce používán pro svou flexibilitu a open-source charakter
- Příklad: Yocto, Buildroot, Ubuntu Core

### **2. RTOS (Real-Time Operating Systems):**

- Použití: Kritické aplikace vyžadující deterministickou odezvu
- Příklad: FreeRTOS, VxWorks

### **3. Windows IoT:**

- Použití: Microsoft řešení pro IoT a vestavné systémy
- Příklad: Windows 10 IoT Core

## **Linux pro vestavná zařízení**

### **Vlastnosti:**

1. Modularita: Možnost přizpůsobení a optimalizace jádra a součástí OS podle potřeb konkrétního zařízení
2. Flexibilita: Široká podpora hardwarových platform a periferií
3. Podpora komunitou: Velká komunita a množství dostupných zdrojů a balíčků

## **Úpravy OS pro cílové zařízení**

1. Kompilace jádra: Přizpůsobení jádra pro specifický hardware
2. Vlastní balíčky a aplikace: Integrace specifického softwaru potřebného pro zařízení
3. Optimalizace výkonu: Úpravy konfigurace systému pro maximální efektivitu a výkon

## **Paměti Flash a systémy souborů**

**Flash** = typ nevolatilní paměti, která uchovává data i po vypnutí napájení a používá se v různých zařízeních pro dlouhodobé ukládání dat

**Systém souborů (filesystem)** = způsob organizace a správy dat na úložném médiu, který určuje, jak jsou soubory ukládány, pojmenovávány, přístupně a chráněny

*Filesystem typy (obecně): Disk, Flash, Tape, Database, Transactional, Network, Shared Disk, ...*

### **Typy pamětí Flash:**

1. NOR Flash: Vhodná pro ukládání firmware, vysoká spolehlivost při čtení
2. NAND Flash: Vhodná pro ukládání velkých objemů dat, vysoká hustota zápisu

### **Systémy souborů pro Flash paměť:**

1. JFFS2 (Journaling Flash File System 2): Navržen pro NOR Flash
2. YAFFS (Yet Another Flash File System): Navržen pro NAND Flash
3. UBIFS (Unsorted Block Image File System): Moderní souborový systém pro NAND Flash

### **Podpora v OS:**

- Podpora různých souborových systémů je integrována v jádře Linuxu
- Umožňuje efektivní správu paměti Flash, wear leveling, a robustní správu dat

## 39. Pipeline pro 3D grafiku v reálném čase: struktura a datové toky, druhy a použití shaderů, komunikace programu na CPU s shadery.

### Struktura a Datové Toky

**3D grafická pipeline** = sekvenční zpracování datových toků, které přeměňuje 3D modely na 2D obraz (v reálném čase)

- Závislá na konkrétním renderu – Většinou formou 3D polygonal rendering

**3D Polygonal rendering** = proces převádění trojúhelníkových sítí do 2D obrazu pomocí grafické pipeline, která zahrnuje transformaci, osvětlení, rasterizaci a shading

**Obálka objektu** – popis pomocí 3D vektorového popisu

- **Polygonální popis:**
  - a) **vertexy** = tj. **body**
  - b) **hrany**
  - c) **plošky + normálové vektory**
- Křivky
- ...

### The World Coordinate System

= souřadnicový systém k definování polohy ve 3D scéně

WCS je globální referenční systém pro 3D scény (souřadnice na osách x, y, z)

Transformace mezi lokálními a světovými souřadnicemi je nezbytná pro správné umístění a zobrazení objektů

WCS umožňuje konzistentní správu a manipulaci s objekty ve 3D prostředí

**Origin** = referenční bod podle kterého jsou popsány pozice všech objektů v prostoru (space)

### Fáze pipeline:

Celá pipeline může být rozdělena na 3 hlavní části zakončené výstupem na obrazovce

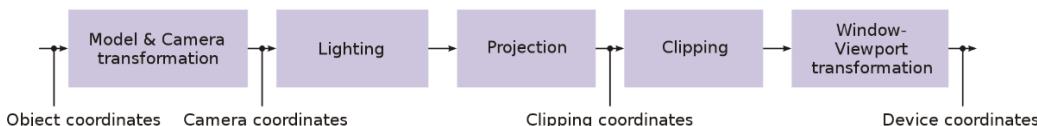
#### 1. Application Stage (Aplikační fáze)

- Během kroku aplikace se provádějí změny scény (např. interakcí uživatele pomocí vstupních zařízení nebo během animace)
- Následně je scéna se svými primitivy (obvykle trojúhelníky, čáry a body) je pak předána dále
- Běh na CPU

Příklady úloh fáze: detekce kolizí, animace, transformace, fyzikálních simulací, zpracování vstupů od uživatele

#### 2. Geometry Stage (Geometrická fáze)

- Obsahuje geometry pipeline během které je prováděná práce s polygony a vrcholy
- **Zahrnuje:** vertex, tessellation a geometry shader pro transformaci a manipulaci s geometrií
- Dělí se na další fáze:



- Fáze se mohou lišit dle konkrétní implementace

##### 1) Model and Camera transformation

3D objekty se transformují z jejich lokálních modelových souřadnic do světových souřadnic a aplikuje kamerová transformace

Tyto transformace jsou prováděny pomocí maticových operací a jsou klíčové pro správné umístění a perspektivu objektů ve 3D prostoru před jejich projekcí na 2D obrazovku

## 2) Lightning (nasvícení)

Určuje se, jak světlo interaguje s povrhy objektů ve scéně – výpočty ambientní, difuzní a spekulární složky osvětlení

- **Ambientní osvětlení** = světlo, které osvětuje všechny objekty rovnoměrně
- **Difuzní osvětlení** = závisí na úhlu dopadu světla na povrch a jeho odrazivosti, což vytváří měkké, rozptýlené světlo
- **Spekulární osvětlení** = simuluje lesklé odrazy světla na hladkých površích, vytvářející jasné odlesky

Výpočty osvětlování pomocí **shaderů** na GPU – berou v úvahu polohu a vlastnosti světelných zdrojů, materiálové vlastnosti objektů a pozici a orientaci kamery

## 3) Projection

Transformují souřadnice 3D scény z pohledových souřadnic do 2D projekčních souřadnic, které odpovídají zobrazení na obrazovce

Tato fáze využívá projekční matici, která může být ortogonální nebo perspektivní

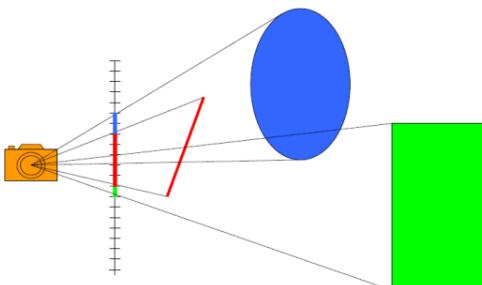
- Ortogonální projekce = zachovává paralelní linie bez perspektivní zkratky
- Perspektivní projekce = objekty vypadají menší než bližší objekty, čímž se vytváří dojem hloubky

Projekční matice transformuje 3D souřadnice do 2D prostoru obrazovky a zároveň uchovává informace o hloubce (z-souřadnici) pro účely z-bufferování (pro vykreslení překrývajících se objektů)

Výsledkem: příprava dat pro rasterizaci, kde se z transformovaných souřadnic stanoví pixely, které se vykreslí na obrazovce

## 4) Clipping (ořezávání)

Identifikace a odstraní části objektů, které leží mimo zorné pole kamery, aby se zefektivnil proces vykreslování – kontrolou, zda jsou trojúhelníky a další primitiva uvnitř pyramidového prostoru



## 5) Window-Viewport Render

Transformují normalizované souřadnice projekce na souřadnice obrazovky (viewport), které odpovídají rozměru okna nebo obrazovky, na které se má scéna vykreslit

Zahrnuje mapování souřadnic z rozsahu  $[-1, 1]$  do rozměru okna/obrazovky, provádění normalizace hloubkových hodnot (z-buffering), která zajistí správné vykreslení průhledných a překrývajících se objektů na obrazovce

## 3. Rasterization Stage (Rasterizační fáze)

- V kroku rasterizace se ze spojitých primitiv vytvoří diskrétní fragmenty
- **Fragmenty** = body mřížky
- Každý fragment odpovídá jednomu pixelu ve vyrovnanovací paměti snímků, a tedy jednomu pixelu obrazovky
- Určuje se viditelnost fragmentů blíže pozorovateli z překrývajících se polygonů – pomocí Z-buffer

- Barva fragmentu závisí na osvětlení, struktuře a dalších materiálových vlastnostech viditelného primitiva a je často interpolována pomocí vlastnosti vrcholu trojúhelníku
- Pokud je to možné, spustí se v kroku rastrování pro každý fragment objektu fragment shader (neboli Pixel Shader)
- Z fragmentů jsou tvořeny pixely
- **Zahrnuje** fragment (pixel) shadery, které určují barvu jednotlivých pixelů na základě textur a osvětlení

## Druhy a Použití Shaderů

**Shader** = program, který vypočítává vhodné úrovně světla, tmy a barvy během vykreslování 3D scény – proces známý jako shading (stínování)

- Shaderů je více druhů, každý má specifickou funkci
- Jejich nastavení možné v Shader-Controlled pipeline

### 1. Vertex Shader

- Popis: Provádí operace na jednotlivých vrcholech (vertexech)
- Použití: Účelem je transformovat 3D polohu každého vrcholu ve virtuálním prostoru na 2D souřadnici, na které se objeví na obrazovce (stejně jako hodnotu hloubky pro Z-buffer)
- Manipulace s vlastnostmi jako poloha, barva a souřadnice textury (nevytváří nové vrcholy)
- Výstup do dalších částí Geometry fáze nebo pro Rasterization

### 2. Tessellation Shaders

- Od OpenGL 4.0 a Direct3D 11
- Rozdělují plochy meshe na větší detaily – podrobnější mesh s více ploškami

### 3. Geometry Shader

- Operuje na celé primitivy
- Může generovat nová grafická primitiva (body, čáry a trojúhelníky) z těch primitiv, která byla předána na začátek grafické pipeline
- Spouštějí se po vertex shaderech
- Vstup: celý primitiv, případně s informací o sousedství (např. tři vrcholy trojúhelníku)
- Výstup: nula nebo více primitiv, které jsou rastrovány a jejich fragmenty nakonec předány pixel shaderu
- Využití: Generování nových geometrických prvků, efekty jako vybuchující objekty

### 4. Fragment (Pixel) Shader

- Slouží k výpočtu barvy a dalších atributů každého fragmentu
- Jednodušší verze: Vydávají jeden pixel obrazovky jako hodnotu barvy
- Složitější verze: Barvy + efekty jako stíny nebo zrcadlová světla, mapování nerovností, průhlednost a průsvitnost
- Mohou změnit hloubku fragmentu (pro Z-buffering)
- Funguje pouze na jediném fragmentu, bez znalosti geometrie scény (tj. vrcholových dat)

### 5. Compute Shader

- Neomezují se na grafické aplikace, ale využívají stejné prostředky k provádění obecných výpočtů mimo tradiční grafické operace
- Použití: Fyzikální simulace, výpočty nezávislé na vykreslování

## Komunikace programu na CPU s shadery

### 1. Shader Program Compilation

- Shadery jsou napsány ve specifických jazycích (např. GLSL, HLSL), komplikovány a linkovány do shader programů

### 2. Shader Program Binding

- Shader programy jsou načteny a aktivovány v grafické pipeline pomocí grafických API (OpenGL, Direct3D, Vulkan)

### 3. Uniforms and Attributes

- Uniforms: Konstantní data posílaná z CPU do shaderů (např. světelné pozice, textury)
- Attributes: Data specifická pro vrcholy (např. pozice, normály, texturové koordináty)

### 4. Buffer Objects

- Vertex Buffer Objects (VBO): Ukládají data vrcholů
- Index Buffer Objects (IBO): Ukládají indexy vrcholů pro efektivní vykreslování

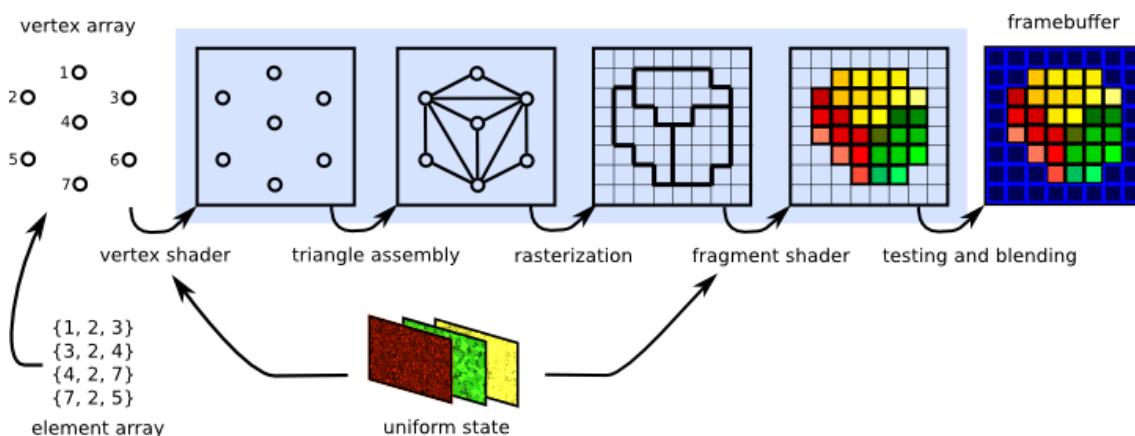
### 5. Textures

- Textury jsou načteny na CPU a nahrány do GPU paměti, kde je shadery využívají k zobrazení detailů povrchů

### Shrnutí

3D grafická pipeline v reálném čase je proces transformující 3D data do 2D obrazu, využívající různé typy shaderů pro specifické úkoly

Shadery provádějí operace na vrcholech a fragmentech, zatímco CPU komunikuje s GPU skrze shader programy, uniformy, atributy a textury, čímž umožňuje dynamické a interaktivní grafické aplikace



Příklad z OpenGL