Základy teoretické informatiky

1. Konečné automaty, regulární výrazy, uzávěrové vlastnosti třídy regulárních jazyků.

Konečné automaty

Konečný automat (KA) tvoří množina stavů, vstupní abceda, přechodová funkce, počáteční a koncové stavy. Můžeme jej znázornit jako tabulku, graf či strom.

Konečné automaty se dělí na determistické a nedetermistické. Deterministický konečný automat má pouze jeden počáteční stav a přechodová funkce vrací jeden stav. Zatímco nedeterministický KA může mít více počátečních stavů a přechodová funkce vrací množinu stavů.

Slovo přijaté automatem je taková sekvence symbolů (ze vstupní abecedy), pro kterou automat skončí v koncovém stavu (příklad).

Regulární jazyk je takový jazyk (množina slov) který lze popsat konečným automatem.

Definice konečného automatu

Konečný automat je uspořádaná pětice A (Q, Σ, δ, g_0, F), kde:

- Q je **stavový prostor** - konečná neprázdná **množina stavů**
- Σ je **vstupní abeceda** - konečná neprázdná množina vstupních symbolů
- δ je **přechodová funkce** - zobrazení $\delta: Q \times \Sigma \rightarrow Q$, které na základě stavu a symbolu abecedy vrátí další stav
- g_0 je **počáteční stavy** - $g_0 \in Q$
- F je **množina koncových stavů** - $F \subseteq Q$

Regulární výrazy

Regulární výraz je textový řetězec, který umožňuje popsat nějakou množinu slov. Regulární výrazy také můžeme chápat jako jednoduchý způsob, jak popsat konečný automat umožňující generovat všechna možná slova patřící do daného jazyka. Pomocí regulárních výrazů můžeme definovat regulární jazyky. Regulární jazyk je takový jazyk, který je přijímán nějakým konečným automatem.

V regulárních výrazech využíváme znaky abecedy a symboly pro sjednocení, zřetězení a iterace regulárních výrazů. Za regulární výraz se považuje i samotný znak abecedy (např. a) stejně jako prázdné slovo ε a prázdný jazyk ∅.

Definice bezkontextové gramatiky

Regulární výrazy popisující jazyky nad abecedou Σ:

- ∅, ε, a (kde $a \in \Sigma$) jsou regulární výrazy:
- ∅ . . . označuje prázdný jazyk
- ε . . . označuje jazyk {ε}
- a . . . označuje jazyk {a}

Jestliže α, β jsou regulární výrazy, pak i (α + β), (α · β), (α∗) jsou regulární výrazy:

- (α + β) . . . označuje sjednocení jazyků označených α a β
- (α · β) . . . označuje zřetězení jazyků označených α a β

- (α∗) . . . označuje iteraci jazyka označeného α

(a + b) = { a, b } ... **sjednocení** => vrací ***a*** nebo ***b***
(a • b) = ab = { ab } ... **zřetězení** (píše se taky bez puntíků) slepí jednotlivé výrazy k sobě
a* = { ε, a, aa, aaa, aaaa, aaaaa, ... } ... **iterace** - opakuje výraz nula až nekonečno krát

---

Příklady: ***ve všech případech Σ = {0, 1}***

(0 + 1) • 0* = { 0, 1, 00, 10, 000, 100, 0000, 1000, ...}
(0 + 1)*00 = {000, 100, 0000, 0100, 1000, 1100, ...} ***jazyk tvořený všemi slovy končícími 00***
(0 + 1)*1(0 + 1)* = {1,01,10,11, 001, ...} ***jazyk tvořený všemi slovy obsahujícími alespoň jeden symbol 1***

Uzávěrové vlastnosti třídy regulárních jazyků

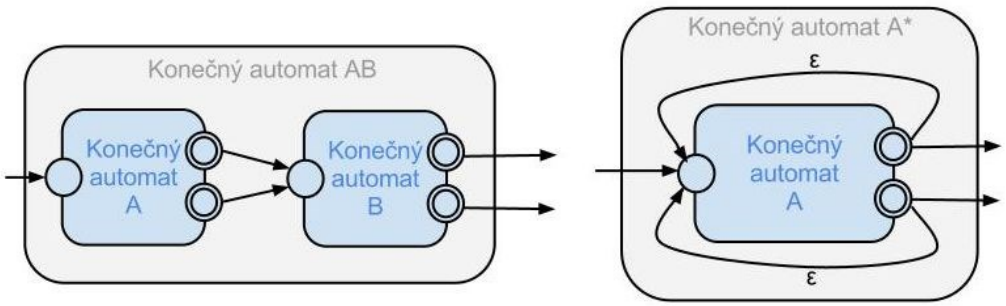
Třídu regulárních jazyků značíme REG a je to množina všech regulárních jazyků.

Uzavřenost množiny nad operací znamená že výsledek operace s libovolnými prvky z množiny bude opět spadat do dané množiny.

Třída REG je uzavřena vůči **sjednocení**, **průniku**, **doplňku**. (Je-li tedy $L_1, L_2 \in \mathrm{REG}$, pak také $L_1 \cup L_2$, $L_1 \cap L_2$, L_1 jsou v REG.) a **zřetězení** a **iteraci**. (Je-li tedy $L_1, L_2 \in \mathrm{REG}$, pak také $L_1 \cdot L_2$, a $(L_1)_*$ jsou v REG.) a **operaci zrcadlového obrazu**. (Je-li tedy $L \in \mathrm{REG}$, pak také $L_R \in \mathrm{REG}$.)

Každý regulární jazyk můžeme popsat konečným automatem.

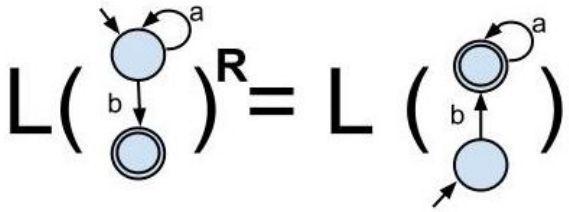
Konečný automat pak ziterujeme tak, že spojíme koncové stavy s počátečními ε přechodem. Na obrázku generuje automat A* jazyk L(A*) = L(A)*, který je iterací jazyku generovaného modrého automatu A.



Dva konečné automaty zřetězíme podobně. Spojíme koncové stavy jednoho s počátečními stavy druhého. Na obrázku dole generuje konečný automat AB jazyk L(AB) = L(A) • L(B).

Třída REG je uzavřena vůči zrdcadlení.
Jazyk (***L^R***) který je zrdcadlovým obrazem jiného jazyka (***L)*** má všechny slova 'osově souměrné'.
Příklad: Je-li L={ ab, aab, aaab, aaaab, ...}, pak zrdcadlový obraz k jazyku L vypadá následovně.
L^R={ ba, baa, baaa, baaaa, ...}

Z automatu dostaneme zrdcadlový obraz jazyka tak, že jej pustíme pozpátku. Takže **konečný automat zezrcadlíme** tak, že jej celý převrátíme. Přehodíme orientaci všech přechodů, z počátečních stavů uděláme koncové a naopak.



2. Bezkontextové gramatiky a jazyky. Zásobníkové automaty, jejich vztah k bezkontextovým gramatikám.

Bezkontextové gramatiky

Bezkontextová gramatika definuje bezkontextový jazyk - jazyk jehož slova se tvoří nezávisle na okolí (na předchozích krocích). Bezkontextovou gramatiku tvoří **neterminály** (proměnné), **terminály** (konstanty) a **pravidla**, které každému neterminálu definují zač je lze přepsat. Jeden neterminál označíme jako startovní, tam začínáme a podle pravidel je dál přepisujeme na výrazy složené z terminálu a neterminálu. Jakmile už není co přepisovat, výraz obsahuje už jen neterminály, získali jsme slovo.
Bezkotext gram. Je je uzavřena na sjednocení, zřetězení, iteraci a zrcadlový obraz.
Ke každé bezkontextové gramatice existuje ekvivalentní zásobníkový automat

---

Příklad: Bezkontextová gramatika generující početní příklady pro sčítání a násobení do pěti.

S* -> O | (O)
O -> O+O | O*O | S | 1 | 2 | 3 | 4 | 5

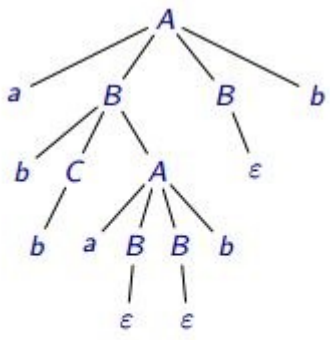
S a C jsou neterminály. S jsme označili * jako startovní neterminál. Číslice, znaménka a závorky jsou terminály (už je nemůžeme přepsat). V gramatice máme definované dvě pravidla (pro každý terminál jedno), které určují za co lze terminály přepsat. Jednotlive přepisy jsou oddělené svislou čárou. Touto gramatikou můžeme vygenerovat libovolný početní příklad dle zadání. Zkusme třeba ((1+3)*5)+(3+2)

S -> O+O -> S+S -> (O)+(O) -> (O*O)+(O+O) -> (S*5)+(3+2) -> ((O)*5)+(3+2) -> ((O+O)*5)+(3+2) -> ((1+3)*5)+(3+2)

Derivace slova je odvození slova pomocí gramatiky, tedy záznam postupných přepisů od startovního neterminálu po konečné slovo. Příklad derivace vidíme na konci příkladu. Derivace se podle postupu při přepisování dělí na levou a pravou. Pokud přepisujeme nejprve levé neterminály, jde o **levou derivaci**. Pokud jedeme zprava jedná se o **derivaci pravou**.

Derivační strom je grafické znázornění derivace slova stromem. Pro všechny možné derivace (levou, pravou, moji) by měl derivační strom být stejný. Není-li tomu tak jedná se o **nejednoznačnou gramatiku**, což je nežádoucí jev.

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



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

Definice bezkontextové gramatiky

Bezkontextová gramatika je čtveřice G = (Π,Σ, S, P), kde

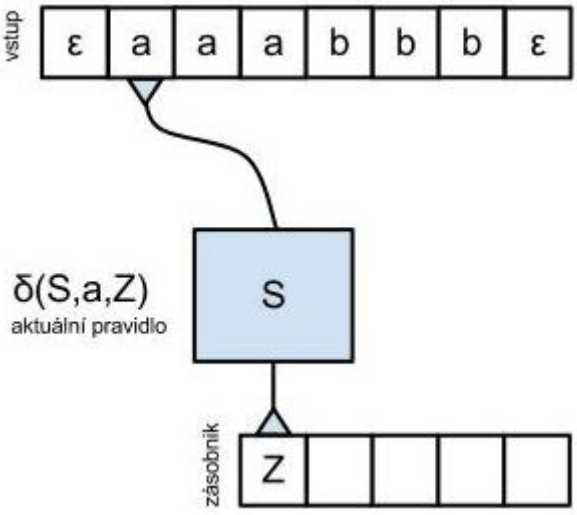
- Π je konečná množina neterminálů (proměnných)
- Σ je konečná množina terminálu (konstant)
- S je počáteční neterminál S ∈ Π
- P je konečná množina pravidel, přepisů neterminálů na řetězce terminálu a neterminálů

Chomského normální forma - pravidla gramatiky může obsahovat pravidla které se přepisují pouze na jeden terminál, nebo na dva neterminály (A -> B | ab). Nesmí tam být pravidlo typu A -> Aa.

Nevypouštějící gramatika - neobsahuje epsilon přechody.

Zásobníkové automaty

Podobně jako slouží konečný automat k rozpoznávání regulárních výrazu, slouží zásobníkový automat k rozpoznání bezkontextových jazyků. Konečný automat si pamatuje akorát aktuální stav. Neví kolik znaku přečetl a které to byly. A to právě potřebujeme k rozpoznání bezkontextového jazyku. Zásobníkový automat je v podstatě konečný automat rozšířený o zásobník.



Zásobníkový automat na základě aktuálního znaku na vstupu, prvního (posledně zapsaného) znaku v zásobníku a aktuálního stavu, změní svůj stav a přepíše znak v zásobníku, a to podle daného pravidla δ.

Pravidlo	Vysvětlení
$\delta(S,a,Z)=\{(S,AZ)\}$	**přidání** prvku do zásobníku Do zásobníku se přidá A.
$\delta(S,a,Z)=\{(S,A)\}$	**přepsání** prvku v zásobníku První prvek zásobníku se přepíše na A.
$\delta(S,a,Z)=\{(S,\varepsilon)\}$	**smazání** prvku v zásobníku První prvek zásobníku se smaže (nahradí prázdným slovem ε)
$\delta(S,a,Z)=\{(T,Z)\}$	**změna stavu** Stav S se změní na stav T. Pro stav T budou platit pravidla typu δ(T,...
$\delta(S,a,Z)=\emptyset$	**pád automatu** Ukončení výpočtu, slovo nebylo přijato.

Kdyby bylo pravidlo z obrázku definováno takto δ(S,a,Z)={(U,AZ)}, změnil by se stav S na stav U a zásobník by obsahoval znaky AZ. V dalším kroku by se na zásobníku četlo A a na vstupu další znak slova, což je 'a'. Aplikovalo by se tedy pravidlo δ(U,a,A).

Pravidla definují chování automatu. V následující tabulce jsou popsány různé typy pravide. Všechny se aplikují v situaci, která je zobrazena na obrázku a popsána v odstavci výše.

Po dokončení činnosti (po přečtení celého vstupu, pokud do té doby nedojde k chybě) je rozhodnuto, jestli automat vstupní řetězec přijal. K tomu mohou sloužit dvě kritéria:

- stav, ve kterém se na konci automat nachází, patří do množiny přijímajících stavů, nebo
- zásobník je na konci prázdný.

Definice zásobníkového automatu:

Zásobníkový automat M je definován jako šestice $M = (Q,\Sigma, \Gamma, \delta, q0, Z0)$, kde

- Q je konečná neprázdná množina stavů,
- Σ je konečná neprázdná množina vstupních symbolů (vstupní abeceda),
- Γ je konečná neprázdná množina zásobníkových symbolů (zásobníková abeceda),
- $q0 \in Q$ je počáteční stav,
- $Z0 \in \Gamma$ je počáteční zásobníkový symbol a
- δ je zobrazení množiny $Q\times(\Sigma\cup\{\varepsilon\})\times\Gamma$ do množiny všech konečných podmnožin množiny $Q \times \Gamma*$.

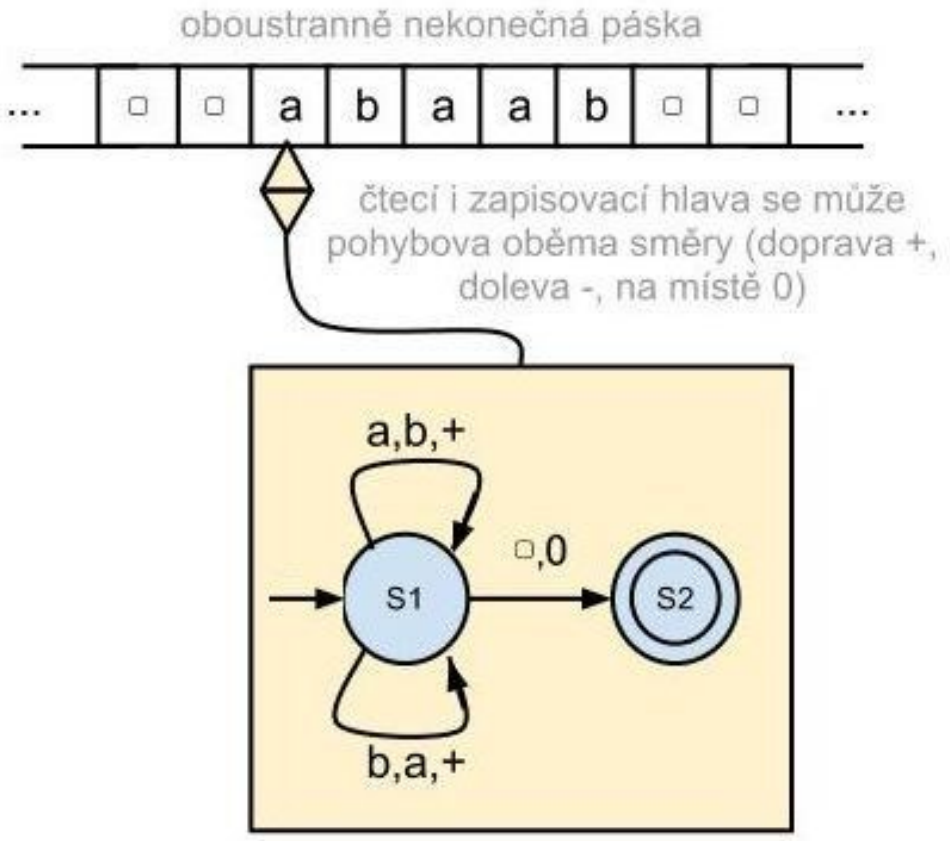
3. Matematické modely algoritmů - Turingovy stroje a stroje RAM. Složitost algoritmu, asymptotické odhady. Algoritmicky nerozhodnutelné problémy.

Turingovy a RAM stroje

Ve snaze definovat algoritmus si vymysleli matematici Turingovy a RAM stroje... Jde o dva různé přístupy (modely) univerzálních počítačů/programovacích jazyků/algoritmů. Jinými slovy těmito stroji lze definovat a provést libovolný algoritmus.

Model Turingova stroje byl popsán dříve, ještě před rozmachem počítačů, proto se od reálného počítače (programování) podstatně liší, na rozdíl od RAM stroje. Turing například pracuje s celou abecedou zatímco RAM (podobně jako počítač) s čísly.

Turingovy stroje



Turingův stroj je podobný konečnému automatu, ale má oboustranně nekonečnou pásku, místo symbolu ε pro prázdné znaky se používá ▢, hlava je čtecí i zapisovací a pohybuje se po pásce v obou směrech.

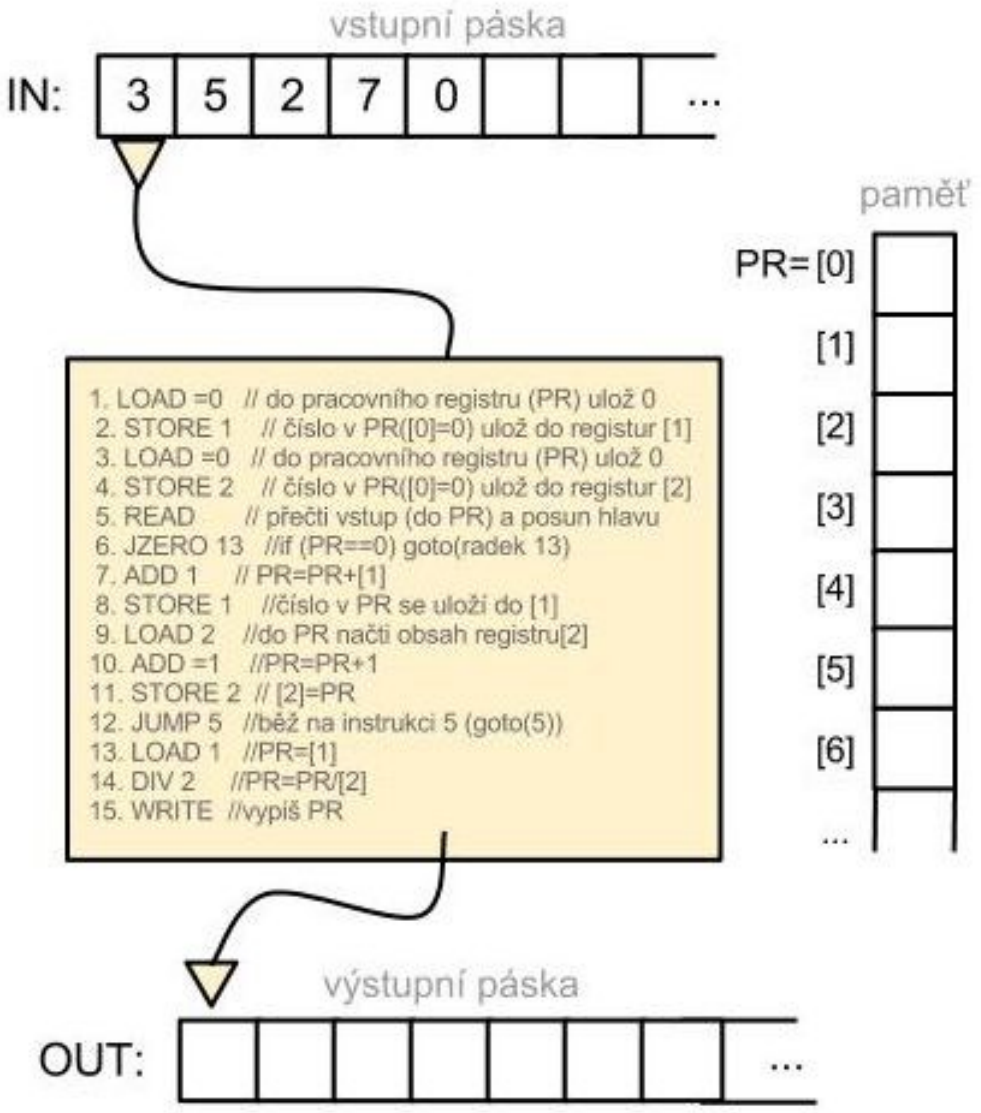
Turingův stoj definujeme seznamem přechodových funkcí δ, které tvoří instrukce algoritmu. Přechodová funkce má tvar:

δ(stav, znak na pásce) = (novy stav, novy znak, posun (-1;0;1))

Na obrázku vidíme Turingův stroj realizující algoritmus znázorněný modrým automatem (zapis u hran znamená - čtený_znak, psaný_znak, posunu). Přechodové funkce by vypadaly následovně: $\delta_1(S1,a)=(S1,b,+)$, $\delta_2(S1,b)=(S1,a,+)$, $\delta_3(S1,▢) = (S2,▢, 0)$.

Zobrazeny algoritmus invertuje slovo na pásce. Ve stavu S1, přepisuje 'a' na 'b' a naopak, a posunuje se doprava (+). Jakmile je slovo přečtené (narazí na prázdný znak ▢), přechází do koncového stavu S2 a dál se neposouvá.

RAM stroje



RAM stroje už vycházejí ze skutečných počítačů. RAM stroje mají paměť, pracují s přirozenými čísly (1,2,3,...) a instrukce se podobají klasickýcm příkazům. Buňky paměti jsou oadresovány čísly 0 - ∞, přičemž první registr [0] se bere jako pracovní registr a registr [1] je indexový registr (pouzívá se u příkazu s hvězdičkou).

Na obrázku je znázorněný RAM stroj s vnitřním kodem, pro výpočet průměru ze sekvence čísle ukončených nulou.

Instrukce turingova stroje:

Čísla (operátory) za příkazy mají následující významy:

op	vysvětlení
6	Hodnota 6
6	Hodnota v registru [6] Pro podmínky (JZERO, JGTZ) a JUMP je to číslo řádku
*6	Hodnota registru [IR], kde IR je hodnota indexivého registur [1].

	Toho se využívá při procházení poli.

V následující tabulce jsou vypsány instrukce RAM stroje. Je-li za instrukcí ***op*** viz předchozí tabulka, ***r*** znamená číslo řádku a není li uvedeno nic, používá se příkaz samostatně.

instrukce	**vysvětlení**
READ	do pracovního registru se načte vstup a hlava se posune [0]=IN
WRITE	na vystup se vypíše PR OUT=[0]
LOAD ***op***	do pracovního registru se načte hodnota dána operátorem [0]=***op***
STORE ***op****	pracovní registr se zapíše do registru daného operátorem => * op nemůže být ve tvaru =číslo ***op***=[0]
ADD ***op***	sčítání - k PR se přičte hodnota daná operátorem [0]=[0]+***op***
SUB ***op***	odčítání - od PR se odečte hodnota daná operatorem [0]=[0]-***op*** **(PR-=op)**
MUL ***op***	násobení - PR se vynásobí hodnotou danou operátorem [0]=[0]***op***
DIV ***op***	celočíselné dělení - PR se vydělí hodnotou danou operátorem [0]=[0]/***op***
JUMP ***r***	skok na řádek r
JZERO ***r***	rovno nule - je-li prac.registr. roven nule, pokračuje kod na řádku ***r***
JGTZ ***r***	kladné - je-li prac.registr. větší než nula, pokračuje kod na řádku ***r***
HALT ***r***	korektní ukončení programu

Složitost algoritmů

Abychom mohli porovnávat různé algoritmy řešící stejný problém, zavádí se pojem složitost algoritmu. Složitost je jinak řečeno náročnost algoritmu - čím menší složitost tím je algoritmus lepší.

Přičemž nás může zajímat složitost z pohledu času, či paměti. Jdo o tzv.:

- časová složitost - sleduje nároky algoritmu na čas. Jak dlouho trvá výpočet s tímto algoritmem? Dočkají se výsledku alespoň mé vnoučata?
- prostorová složitost - sleduje nároky algoritmu na paměť. Je schopen provést tento algoritmus můj počítač s omezenou pamětí?

Jelikož konkrétní čísla (čas, bity) se liší v závislosti vstupních datech, množství zpracovávaných dat a použitém programovacím jazyku, neudává se složitost čísly, nýbrž funkcí závislou na velikosti vstupních dat. Tato funkce se získá počítáním proběhlých instrukcí algoritmu sestaveném v univerzálním RAM stroji. A počítá se s nejhorším možným případem vstupu. To je důležité například u třídících algoritmů, kde hraje velkou roli to, jak moc už je vstupní pole setříděné (vstupuje-li do algoritmu už setříděná posloupnost čísel, algoritmus skončí okamžitě, zatímco s opačně seřazenými čísly se bude trápit dlouho. Proto při počítání složitosti cpeme do algoritmu ten nejhorší případ vstupu, při kterém se bude algoritmus trápit nejdéle.).

Příklad z obrázku RAM stroje pro výpočet průměru sekvence čísel ukončených nulou:

1. LOAD =0 //do pracovního registru (PR) ulož 0
2. STORE 1 // číslo v PR ([0]=0) ulož do registru [1]
3. LOAD =0 //do pracovního registru ulož nulu. PR = [0] = 0
4. STORE 2 // hodnotu PR ulož do registru 2. [2]=[0]
5. READ // načti vstup a posuň hlavu. [0]=IN
6. JZERO 13 // if([0]==0) jump(radek 13)
7. ADD 1 // K PR přičti registr [1]. [0]=[0]+[1]
8. STORE 1 // PR ulož do registru [1]. [1]=[0]
9. LOAD 2 // [0]=[2]
10. ADD =1 // [0]=[0]+1
11. STORE 2 // [2]=[0]
12. JUMP 5 // bez na radek 5
13. LOAD 1 // [0]=[1]
14. DIV 2 // [0]=[0] / [2]
15. WRITE // vypis pracovni registr [0]
16. HALT // konec programu

Výpočet složitosti pro algoritmus popsány instrukcemi RAM stroje vpravo bude následující:

Prvních 6 instrukcí se provede vždy, stejně jako poslední 4 instrukce. To je celkem 10 instrukcí které se provedou vždy i pro velikost vstupu ***n=1***. Instrukce 5 až 12 tvoří smyčku čítající 8 instrukcí, která je závislá na velikosti vstupu ***n*** a provede se ***n-krát***. Touto úvahou získáme následující funkci:

$f(n) = 6 + n*8 + 4 = 8n + 10$

Asymptotický odhad složitosti

Asymptonický odhad složitosti je další zobecnění složitosti algoritmu. Výpočet přesné funkce složitosti je pro komplikovanější algoritmy příliš náročný a navíc zbytečný. Pro představu o náročnosti algoritmu nám stačí vědět jak rychle funkce roste.
U předchozího příkladu nás tedy nezajímá kolik přesně instrukci se provedlo vně či vně cyklu, důležité je že funkce roste lineárně. Jak moc je lineární funkce nakloněná či zvednutá, není podstatné, protože každá kvadratická funkce ji nakonec předběhne. Tedy kvadratická složitost se při dostatečně velkém vstupu, stane vždy horší.

V souvislosti s asymptotickými odhady složitosti se používjí tyto zapisy:

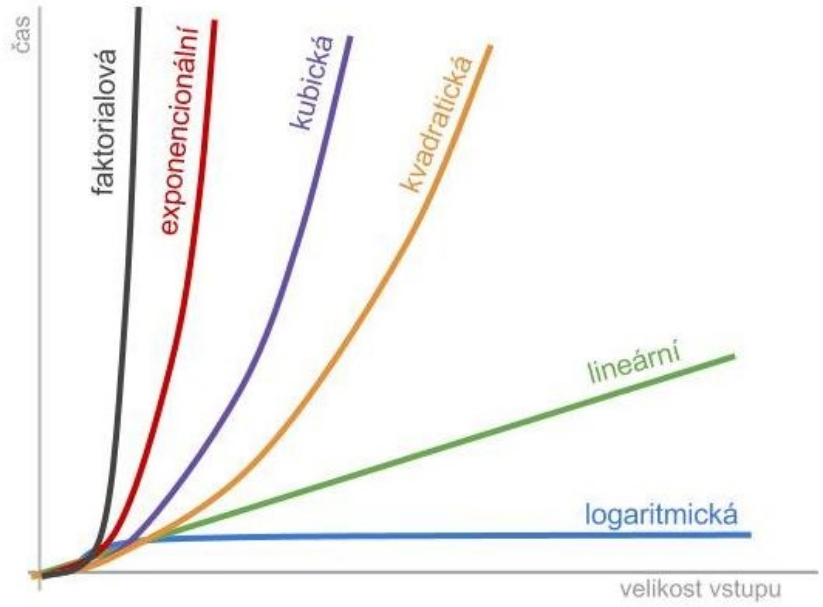
- $f \in O(g)$ . . . . f roste nejvýše tak rychle jako g = f je asymptoticky ohraničena funkcí g shora
- $f \in o(g)$ . . . . f roste (striktně) pomaleji než g = f je asymptoticky ohraničena funkcí g shora ostře
- $f \in \Omega(g)$ . . . . f roste rychleji než g = f je asymptoticky ohraničena funkcí g zdola
- $f \in \omega(g)$ . . . . f je asymptoticky ohraničena funkcí g zdola ostře
- $f \in \Theta(g)$ . . . . f roste stejně rychle jako g

O	≤
o	<
Ω	≥
ω	>
Θ	=

Konstanta c
Log n
n
n^k
$2^n(k^n)$
n!
n^n

Pokud takto "zaokrouhlíme" a zjednoduším funkci složitosti, mluvíme pak o složitosti podle typu funkce.

• f(n) ∈ Θ(log n) . . . . logaritmická funkce (složitost)
• f(n) ∈ Θ(n) . . . . lineární funkce (složitost)
• f(n) ∈ Θ(n^2) . . . . kvadratická funkce (složitost)
• f(n) ∈ O(n^c) pro nějaké c > 0 . . . . polynomiální
• f(n) ∈ Θ(c^n) pro nějaké c > 1 . . . . exponenciální



Algoritmická rozhodnutelnost (řešitelnost) problému

Problém je rozhodnutelný (řešitelný) pokud pro libovolný vstup z množiny vsupů, skončí algoritmus svůj výpočet a vydá správný výstup.
Pokud tedy najdu takový vstup problému, na kterém si všechny dosavadní algoritmy řešící tento problém vylámou zuby, můžu tento problém nazvat neřešitelný. Důkaz neřešitelnosti lze provést také skrze jiné - už dokázané - problémy. Speciální případ jsou doplňkové problémy, které vracejí přesně opačné výsledky než původní problém.
Viz Riceova věta: každá netriviální (triviální: má to každý stroj, Turingův stroj má počáteční stav…), vstupní výstupní vlastnost (jde zjistit z tabulky) je nerozhodnutelná

Definice problému:

Problém je určen trojicí (IN,OUT, p), kde IN je množina (přípustných) vstupů, OUT je množina výstupů a p : IN → OUT je funkce přiřazující každému vstupu odpovídající výstup.

Ano/Ne problémy

Jsou to problémy, jejichž výstupní množina obsahuje dva prvky OUT={ano, ne}.

Na ano/ne problémy se dají převést ostatní problémy nepotřebujeme-li znát přesný výsledek. Například:

- nepotřebuji najít v poli nejmenší číslo, stačí mi vědět zda pole obsahuje číslo menší než nula.
- nepotřebuji znát nejkratší cestu grafem, ale stačí mi najít cestu která je kratší než 8.

Optimalizační problémy

Optimalizační problémy hledají nejlepší řešení. V množině různých řešení hledáme to nejlepší. Příkladem je například hledání nejkratší cesty, nejmenší kostry, apod.

Název: **Nejkratší cesta v grafu**
Vstup: Orientovaný graf G = (V,E) a dvojice vrcholů u, v ∈ V .
Výstup: Nejkratší cesta z u do v.

Název: **Minimální kostra**
Vstup: Neorientovaný souvislý graf G = (VG,EG) s ohodnocenými hranami
Výstup: Souvislý graf H = (VH,EH), kde VH = VG a EH ⊆ EG, který má součet hodnot všech hran minimální.

Název: **Eq-CFG (Ekvivalence bezkontextových gramatik)**
Vstup: Dvě bezkontextové gramatiky G1,G2.
Otázka: Platí L(G1) = L(G2)? Generují obě gramatiky stejný jazyk?

Název: **HP (Problém zastavení [Halting Problem])**
Vstup: Turingův stroj M a jeho vstup w.
Otázka: Zastaví se M na w (tzn. je výpočet stroje M pro vstupní slovo w konečný)?

4. Třídy složitosti problémů. Třída PTIME a NPTIME, NP-úplné problémy.

Třída PTIME

Do této třídy složitosti spadají všechny problémy řešené s polynomiálními algoritmy.

$$PTIME = T(n^k),$$

kde **k** je $\text{konstanta}\ (0,\infty)$. Třídu těchto problému považujeme za zvládnutelnou. Tato třída je robustní**.** ***Robustnost třídy*** znamená nezávislost na zvoleném modelu počítačů. Je tedy jedno zda algoritmus budeme implementovat na Turingově či RAM stroji, složitost problému zůstane v PTIME třídě.

Do této třídy spadá mnoho praktických problému:

- Třídění
- Vyhledávání
- Aritmetické operace
- Ekvivalence determistických konečných automatů
- Nejkratší cesta v grafu
- Minimální kostra grafu
- Přijatelnost slova bezkontextovou gramatikou
- Výběr aktivit
  Vstup: Množina aktivit s časovými intervaly, kdy je lze vykonávat.
  Výstup: Největší možný počet kompatibilních aktivit (aktivit, které se nekryjí)
- Optimalizace násobení řetězce matic
  Vstup: Posloupnost matic
  Výstup: Plně uzávorkovaný součin
- LCS - problém nejdelší společné posloupnosti
  Vstup: Dvě posloupnosti v,w v nějaké abecedě Σ
  Výstup: Nejdelší společná podposloupnost posloupností v,w.

Třída NPTIME

Třída NPTIME zahrnuje všechny rozhodovací (ano/ne) problémy, které jsou rozhodovány nedeterministickými polynomiálními algoritmy - tedy nedeterministickými algoritmy s polynomiální časovou složitosti $O(n^c)$.

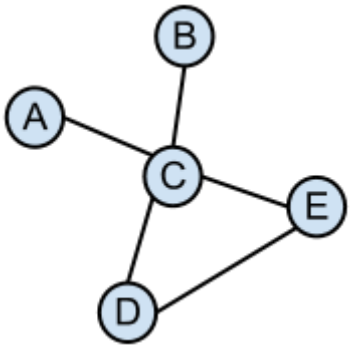
Nedeterministický algoritmus rozhoduje ano/ne problémy, ale **ne vždy správně**. Při odpovědi 'ano' je výsledek vždy správný (našel se alespoň jeden výpočet pro který je odpověď ano) zatímco odpověď 'ne' nemusí být vždy pravdivá a to z důvodu, že by na výstupu muselo být vždy ne (algoritmus by musel otestovat všechny možnosti).

Pointou nedeterministických algoritmů je to že náhodně nastřelují nějaké řešení a ověřují jejich správnost.

Například nedeterministický algoritmus pro problém IS (viz. obrázek a definice níže) vybere náhodně ***k*** vrcholů z grafu a ověří zdali nejsou některé spojeny hranou. Když mezi nimi hranu nenajde, vráti odpověď "ANO, v grafu existuje nezávislá množina o velikosti ***k*** ". Když hranu mezi zvolenou množinou najde, vrátí odpověď "NE". Přičemž odpověď ne může být chybná. Třeba pro graf ABCDE na obrázku níže a ***k=3***, vybere algoritmus vrcholy {A, B, C}, které závislé jsou a vrátí chybnou odpověď "NE". Když to ale algoritmus provede vícekrát, pro různé vrcholy, pravděpodobnost správnosti odpovědi se zvyšuje. A o tom to je. Nepotřebujeme znát 100% správnou odpověď, ale chceme se dočkat alespoň nějaké odpovědi.

Problémy třídy NPTIME

Ukázka IS problému nezávilých množin:



Výstup pro k>3: NE.
Výstup pro k=3: ANO. (ABD, ABE)
Výstup pro k=2: ANO. (AB, AD, AE, BD, BE)

Název: **IS (problém nezávislé množiny** [Independent Set] (také nazývaný problém anti-kliky [anti-clique]))

Vstup: Neorientovaný graf G (o n vrcholech); číslo k ($k \leq n$).
Otázka: Existuje v G nezávislá množina velikosti k (tj. množina k vrcholů, z nichž žádné dva nejsou spojeny hranou)?

Název: **Složenost čísla**

Vstup: Přirozené číslo ℓ.
Otázka: Je číslo ℓ složené (dělitelné beze zbytku)?

Název: **Isomorfismus grafů**

Vstup: Dva neorientované grafy G a H.
Otázka: Jsou grafy G a H izomorfní?

Název: **CG (Barvení grafu)**

Vstup: Neorientovaný graf G a číslo k.
Otázka: Je možné graf G obarvit k barvami (tj. existuje přiřazení barev vrcholům tak, aby žádné dva sousední vrcholy nebyly obarveny stejnou barvou)?

Název: **SAT (problém splnitelnosti booleovských formulí)**

Vstup: Booleovská formule v konjunktivní normální formě.
Otázka: Je daná formule splnitelná (tj. existuje pravdivostní ohodnocení proměnných, při kterém je formule pravdivá)?

Název: **HK (problém hamiltonovské kružnice) / HC (problém hamiltonovskho cyklu)**

Vstup: Neorientovaný graf G. / Orientovaný grav G.
Otázka: Existuje v G hamiltonovská kružnice (uzavřená cesta, procházející každým vrcholem právě jednou)?

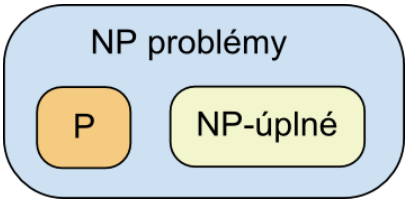
Název: **Subset-Sum**

Vstup: Množina přirozených čísel M = {x1, x2, . . . , xn} a přirozené číslo s.
Otázka: Existuje podmnožina množiny M, pro niž součet jejích prvků je roven s?

Třída NP-úplných problémů

NP-úplné problémy jsou takové nedeterministicky polynomiální problémy, na které jsou polynomiálně redukovatelné všechny ostatní problémy z NP. To znamená, že třídu NP-úplných úloh tvoří v jistém smyslu ty nejtěžší úlohy z NP. Pokud by byl nalezen polynomiální deterministický algoritmus pro nějakou NP-úplnou úlohu, znamenalo by to, že všechny nedeterministicky polynomiální problémy jsou řešitelné v polynomiálním čase (tedy že NP = P). Otázka, zda nějaký takový algoritmus existuje, zatím nebyla rozhodnuta, předpokládá se však, že NP ≠ P (je však zřejmé, že P ⊆ NP). Tento vztah se řadí mezi jeden z problému tisíciletí a za jeho vyřešení se nabízí velká odměna ;)

Definice NP-úplných problémů
Skupina nejtěžších NPTIME problémů. Problém je NP-úplný, když je NP-těžky a náleží do třídy NP. Problém Q nazveme NP-těžkým, pokud každý problém ve třídě NP lze na problém Q polynomiálně převést, tedy pokud platí:
∀P ∈ NPTIME : P ⊳ Q.
NP těžký: existuje nedeterministický algoritmus, který řeší problém v polynomiálním čase.
Problém Q nazveme NP-úplným, pokud je NP-těžký a náleží do třídy NP.



NP-úplné problémy:

- SAT - problém splnitelnosti booleovských formulí

- 3-SAT (problém SAT s omezením na 3 literály) - je splnitelná boolovská formule v konjuktivní normální formě, kde každá klauzule obsahuje právě tři literály?
- CG - barvení grafu ***k*** barvami
- 3-CG - barvení grafu třemi barvami
- HK, HC - hamiltonovské kružnice, cyklu
- Subset-Sum - množina součtů
- TSP - problém obchodního cestujícího (Ano/Ne verze) - lze objet ***n*** měst a neujet víc než danou vzdálenost?
- ILP (problém celočíselného lineárního programování)
  Vstup: Matice ***A*** typu ***m × n*** a sloupcový vektor ***b*** velikosti ***m***, jejichž prvky jsou celá čísla.
  Otázka: Existuje celočíselný sloupcový vektor ***x*** (velikosti ***n***) tž. ***Ax ≥ b***?

5) Jazyk predikátové logiky prvního řádu. Práce s kvantifikátory a ekvivalentní transformace formulí.

Formalizace v predikátové logice 1.řádu (PL1):

Predikátová logika umožňuje uvažování nad vlastnostmi, jež jsou sdíleny mnoha objekty, díky použití proměnných a kvantifikátorů. V predikátové logice 1.řádu by byl uvedený úsudek formalizován takto: ∀x [B(x) ⊃ Z(x)], Z(K) |= B(K), resp. následující formulí

∀x [B(x) ⊃ Z(x)] ∧ Z(K) ⊃ B(K)

kde,

- x ...je předmětová (individuová) ***proměnná*** pocházející ze specifické univerzální množiny (zvané **univerzum**)
- K ...je individuová ***konstanta*** z daného univerza (***v uvedeném příkladě "tento kůň"***),
- B, Z ...jsou určité vlastnosti předmětů z universa diskursu, označovaných také jako predikáty. Predikát B v našem příkladě říká, že objekt, reprezentovaný proměnnou x ***je žába***. Predikát Z zase, že proměnná v závorkách ***je zelená***. Výraz Z(K) tudíž znamená, že proměnná ***tento kůň*** má vlastnost ***být zelený.***
- zápis ∀x [ ] ...značí, že pro všechna individua z univerza platí to, co je uvedeno v hranatých závorkách (znak ∀ je jedním z kvantifikátorů)

Pokud bychom chtěli formalizovat úsudky, které navíc vypovídají i o vlastnostech vlastností a vztahů a o vztazích mezi vlastnostmi a vztahy, museli bychom použít predikátovou logiku druhého řádu a vyššího. Tou se ale nebudeme zabývat.

Definice formálního jazyka predikátové logiky PL1

1. ***Abeceda*** predikátové logiky je tvořena následujícími skupinami symbolů:

   a. Logické symboly

      - Individuové proměnné: x, y, z,***... (příp. s indexy)***
      - Symboly pro spojky: ¬, ∨, ∧, ⊃, ≡
      - Symboly pro kvantifikátory ∀, ∃

   b. Speciální symboly (určují specifiku jazyka)

- Predikátové symboly: P^n, Q^n, ... n - arita = počet argumentů
- Funkční symboly: f_n, g_n, ...

c. Pomocné symboly /závorky/: (,) /případně i [,],{,}/

2. ***Gramatika***, která udává, jak tvořit:

   a. Termy: každá individuová proměnná i konstanta je term
   b. Atomické formule: jsou-li $t_1,\ldots,t_n$ termy, pak výraz $p(t_1,\ldots,t_n)$ je atomická formule
   c. Formule:

      - Každá atomická formule je formule
      - Je-li výraz A, B formule a x proměnná pak i výrazy: ¬A, ∀xA a ∃xA, (A ∨ B), (A ∧ B), (A ⊃ B), (A ≡ B) jsou formulemi

Převod z přirozeného jazyka do jazyka PL1

Tvorba kvantifikátorů:

- ∀ „všichni“, „žádný“, „nikdo“, ... "
- ∃ „někdo“, „něco“, „někteří“, „existuje“, ...

Větu musíme často ekvivalentně přeformulovat, pozor: v češtině dvojí zápor !

Jako pomůcka k řešení může sloužit tato zásada:

- Po všeobecném kvantifikátoru následuje formule ve tvaru implikace **∀ + ⊃**
- Po existenčním kvantifikátoru formule ve tvaru konjunkce **∃ + ∧**

Při ekvivalentních úpravách používáme ***de Morganovy zákony PL1*** :

- ¬∀xA ⇔ ∃x¬A
- ¬∃xA ⇔ ∀x¬A

Příklady:

Není pravda, že všichni vodníci jsou zelení. ⇔ Někteří vodníci nejsou zelení.

¬∀x [V(x) ⊃ Z(x)] ⇔ ∃x [V(x) ∧ ¬Z(x)]

Není pravda, že někteří vodníci jsou zelení. ⇔ Žádný vodník není zelený.

¬∃x [V(x) ∧ Z(x)] ⇔ ∀x [V(x) ⊃ ¬Z(x)]

Everybody loves somebody sometimes.

∀x ∀y ∀z L(x,y,z)

Marie má ráda pouze vítěze.

∀x [R(m,x) ⊃ V(x)]

Substituce termů za proměnné

A(t/x) vznikne z A korektní substitucí termu t za proměnnou x

Pravidla substituce termů:

- Substituovat lze pouze za volné výskyty proměnné x ve formuli A a při substituci nahrazujeme všechny volné výskyty proměnné x ve formuli A termem t.
- Žádná individuová proměnná vystupující v termu t se po provedení substituce t/xnesmí stát ve formuli A vázanou (v takovém případě není term t za proměnnou x ve formuli A substituovatelný).

Příklad:

A(x): $P(x) \supset \forall y\, Q(x, y)$ ...provedeme-li substituci termu f(y) ⇒A(f(y)/x):

A(f(y)): $P(f(y)) \supset \forall y\, Q(f(y), y)$.

⇒***term f(y) není substituovatelný za x v dané formuli A, protože y je vázaná proměnná a substitucí bychom změnili smysl celé formule***

Ekvivalentní transformace

Aplikace negace: $\neg\forall x\, [V(x) \supset Z(x)] \Leftrightarrow \exists x\, [V(x) \wedge \neg Z(x)]$
De morganovy zákony: $\forall x\, [((P(x) \wedge Q(x)) \vee D(x)] \Leftrightarrow \forall x\, [(P(x) \vee D(x)) \wedge (Q(x) \vee D(x))]$
Převod implikace: $\forall x\, (P(x) \supset G(x)) \Leftrightarrow \forall x\, (\neg P(x) \vee G(x))$
Patří zde i část z převodu do Skolemovy Klauzární formy

1. Eliminace nadbytečných kvantifikátorů
2. Eliminace spojek ⊃, ≡
3. Přesun negace dovnitř
4. Přejmenování proměnných
5. Přesun kvantifikátorů doprava
6. Přesun všeobecných kvantifikátorů doleva
7. Použití distributivních zákonů

6. Pojem relace, operace s relacemi, vlastnosti relací. Typy binárních relací. Relace ekvivalence a relace uspořádání.

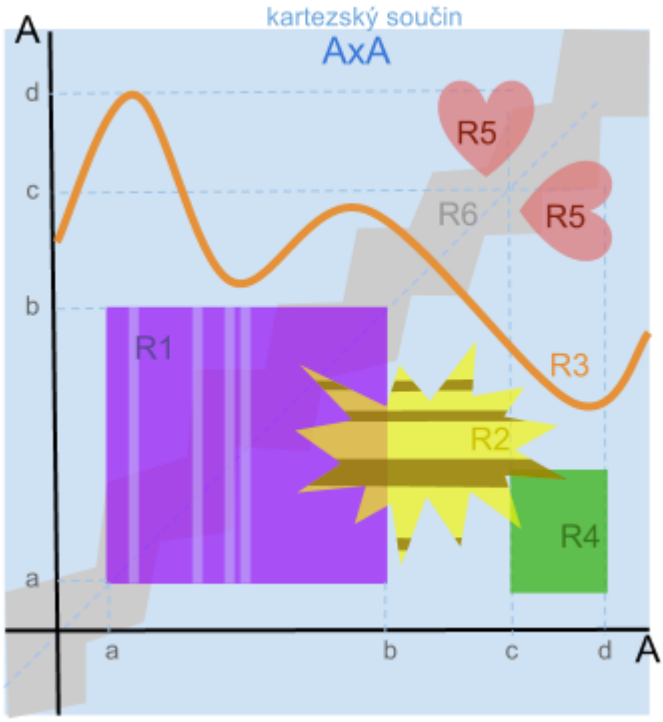
Relace

Relace na množinách $A_1, A_2, \ldots, A_n$ je libovolná podmnožina kartézského součinu $A_1 \times A_2 \times \ldots \times A_n$

Kartézský součin množin A a B, označovaný A × B, je množina všech uspořádaných dvojic, kde první prvek z dvojice patří do množiny A a druhý do množiny B. ***Příklad:*** {a, b} × {a, b, c} = {(a, a), (a, b), (a, c), (b, a), (b, b), (b, c)}

Je-li kartézský součin ze dvou množin (n=2), jde o **binární** relaci, v případě tří množin jde o relaci ternální, atd. Pokud jsou množiny kartézského součinu shodné jde o relaci **homogení** v opačném případě, jde o **heterogenní** relaci.

Na obrázku vidíme příklady binárních homogenních relací, protože kartezská součin AxA je dán dvěma shodnými množinami.



Vlastnosti relací

Binární relace $R \subseteq A \times A$ je:

- **reflexivní**, jestliže pro všechna $a \in A$ platí $(a, a) \in R$.
  Relací musí obahovat celou osu kartézského součinu (obr. šedá relace R6).
- **ireflexivní**, jestliže pro všechna $a \in A$ platí $(a, a) \notin R$.
  Relací nesmí obsahovat ani kousek osy kartézského součinu (obr. R4, R5).
- **symetrická**, jestliže pro všechna $a, b \in A$ platí, že pokud $(a, b) \in R$, pak $(b, a) \in R$.
  Relace je symetrická podle osy kartézského součinu (obr. R1, R5).
- **asymetrická**, jestliže pro všechna $a, b \in A$ platí, že pokud $(a, b) \in R$, pak $(b, a) \notin R$.
  Relace se nachází pouze v jedné polovině kart. součinu rozděleného osou, přičemž se nesmí dotýkat osy (obr. R4).
- **antisymetrická**, jestliže pro všechna $a, b \in A$ platí, že pokud $(a, b) \in R$ a $(b, a) \in R$, pak $a = b$.
  Relace se nachází pouze v jedné polovině kart. součinu rozděleného osou a může obsahovat osu.
- **tranzitivní**, jestliže pro všechna $a, b, c \in A$ platí, že pokud $(a, b) \in R$ a $(b, c) \in R$, pak $(a, c) \in R$.

Příklad:

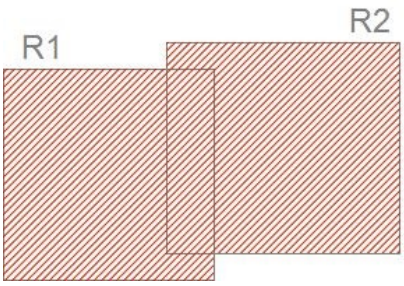
- Relace “=” na N je reflexivní, symetrická, antisymetrická a tranzitivní, ale není ireflexivní ani asymetrická.
- Relace “≤” na N je reflexivní, antisymetrická a tranzitivní, ale není ireflexivní, symetrická ani asymetrická.
- Relace “<” na N je ireflexivní, asymetrická, antisymetrická a tranzitivní, ale není reflexivní ani symetrická.

Operace s relacemi

Množinové operace (sjednocení, průnik, doplněk), inverze a skládání relací.

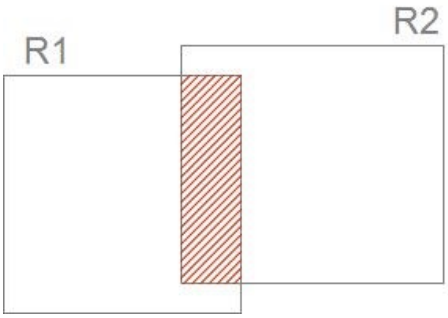
Sjednocení

Prvek ***a*** náleží do sjednocení relací ***R1*** ∪ ***R2***,
pokud patří do množiny **R1** (***a*** ∈ ***R1***) **nebo R2** (***a*** ∈ ***R1***).



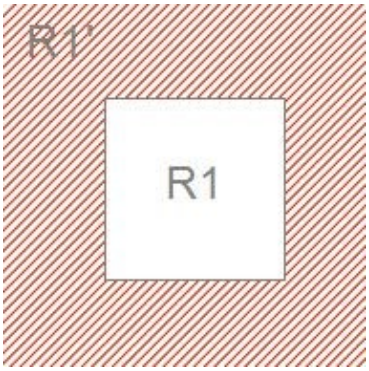
Průnik

Prvek ***a*** náleží do průniku relací **R1** ∩ **R2**,
pokud patří do množiny **R1** (***a*** ∈ ***R1***) **a zároveň** do **R2** (***a*** ∈ ***R1***).



Doplněk

Doplňkem **R1'** k relaci R1 rozumíme všechny prvky které **nepatří do R1.**



Inverze

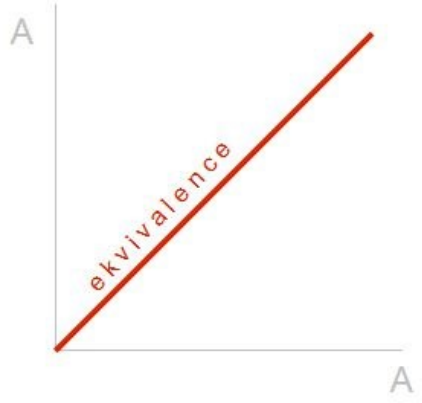
Relace $\mathbf{R^{-1} \subseteq B \times A}$ je inverzní k relaci $\mathbf{R \subseteq A \times B}$, pokud $\mathbf{xR^{-1}y}$ právě tehdy když **yRx.**

Typy binárních relací

Mezi nejznámější binární relace patří ekvivalenece (=) a uspořádání (<,>, ≤, ≥).

Ekvivalence

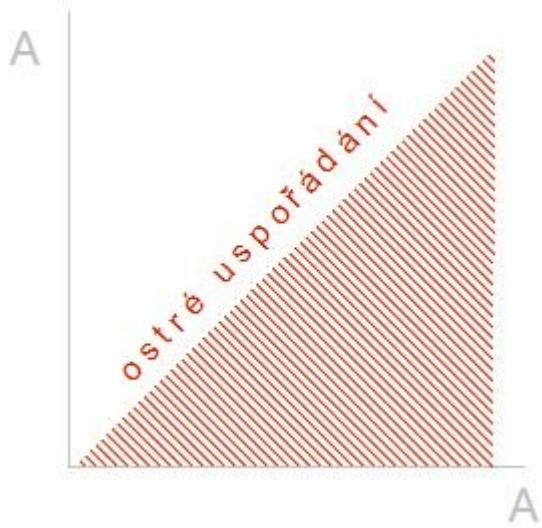
Relace ekvivalence představuje jakési zjemnění relace rovnosti. Vždy můžeme rozhodnout, že jsou dva prvky množiny stejné, tj. že a = a. Ale někdy se nám hodí zjistit, zda jsou si dva prvky pouze podobné, ne nutně stejné. Neboli – zda mají stejnou nějakou zásadní vlastnost. Například dvě knihy můžeme považovat za podobné, pokud mají stejný žánr – nebo pomocí ekvivalence: dvě knihy jsou ekvivalentní pokud mají stejný žánr.
Např: Bydlet ve stejném městě. Jirka určitě bydlí ve stejném městě jako Jirka (reflexivita). Pokud Jirka bydlí ve stejném městě jako Ondra, pak i Ondra bydlí ve stejném městě jako Jirka (symetrie). A pokud Ondra bydlí ve stejném městě jako Martin, pak i Jirka bydlí ve stejném městě jako Martin (transitivita).
Ekvivalence je binární relace, která je

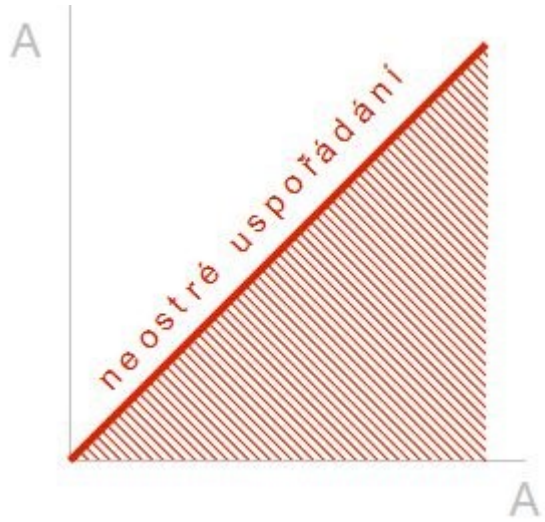


- **reflexivní** (všechna a ∈ A platí (a, a) ∈R)
  a=a
- **symetrická** (pokud (a, b) ∈ R, pak (b, a) ∈ R)
  když ***a=b***, pak ***b=a***
- **tranzitivní** (pokud (a, b) ∈ R a (b, c) ∈ R, pak (a, c) ∈ R)
  když ***a=b*** a ***b=c***, pak se musí ***a=c***

Třída ekvivalence prvku ***a*** - je množina všech prvků ekvivalentních s daným prvkem ***a.***

Uspořádání





Uspořádání je běžný pojem, se kterým pracujeme i v běžném životě. Spoustu věcí můžeme nějakým způsobem uspořádat – slova podle abecedy, žáky ve škole podle výšky, zboží podle ceny. Toto všechno popisuje relace uspořádání, kterou obvykle značíme pomocí symbolu ≤. Související symboly jsou: <, >, ≥. Jejich význam je zřejmý.
Co bychom od takové relace očekávali? Už ze symbolu je jasné, že v relaci uspořádání budou prvky, které jsou stejné – jde nám tedy o to nadefinovat vlastnosti relace menší nebo rovno. Taková relace by tak měla být **reflexivní**. Musí platit, že a ≤ a.
Co dále? Určitě nebude platit symetrie – pokud máme uspořádání podle ceny, tak pokud je auto levnější než chleba, tak určitě nebude zároveň chleba levnější než auto. Co bychom obecně očekávali, pokud by se stalo, že a ≤ b a zároveň b ≤ a? Dávalo by to někdy smysl? Ano, pokud by platilo a = b. Jediný případ, kdy nezáleží na pořadí prvků vložených do relace je, když jsou ty prvky stejné – uspořádání je tak **antisymetrické**.

Nakonec pokud víme, že auto je dražší než chleba a letadlo je ještě dražší než auto, tak očekáváme, že letadlo bude určitě dražší než chleba. To popisuje **transitivita**.

Binární relaci nazveme **neostrým uspořádáním** pokud je:

- **reflexivní** (všechna a ∈ A platí (a, a) ∈R)
  a* ≤ *a*, *a* ≥ *a
- **antisymetrická**, jestliže pro všechna a, b ∈ A platí, že pokud (a, b) ∈ R a (b, a) ∈ R, pak a = b.
  když ***a* ≤ *b*** a zároveň ***b* ≥ *a***, pak se musí ***a*=*b***
- **tranzitivní** (pokud (a, b) ∈ R a (b, c) ∈ R, pak (a, c) ∈ R)
  když ***a* ≤ *b*** a ***b* ≤ *c***, pak se musí ***a* ≤ *c***

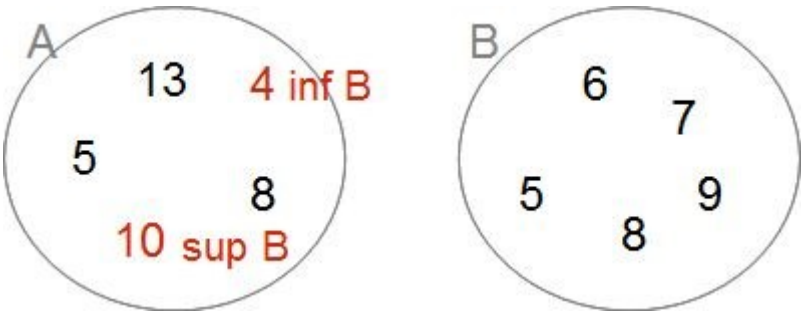
Binární relaci nazveme **ostrým uspořádáním** pokud je:

- **asymetrická**, jestliže pro všechna a, b ∈ A platí, že pokud (a, b) ∈ R, pak (b, a) ∉ R.
  když ***a* > *b*** pak nemůže platit ***b* > *a***
- **ireflexivní, jestliže pro všechna a ∈ A platí (a, a) ∉R.**
  nemůže platit ***a* < *b***
- **tranzitivní** (pokud (a, b) ∈ R a (b, c) ∈ R, pak (a, c) ∈ R)
  když ***a* > *b*** a ***b* > *c***, pak se musí ***a* > *c***

Uspořádání je **úplné** (lineární) pokud neexistují neporovnatelné prvky.

Minimum - je prvek menší než všechny ostatní.
Maximum - je prvek větší než všechny ostatní.

Infimum množiny B (inf B) **-** je největší prvek, který je však pořád **menší než** prvky množíny **B**.
Suprebum množiny B (sup B) - je nejmenší prvek **větší než** prvky množiny **B**.



7. Pojem operace a obecný pojem algebra. Algebry s jednou a dvěma binárními operacemi.

Algebra

Algebra je odvětví matematiky zabývající se objekty (abstraktní pojmy, struktury) a vztahy mezi nimi (operace).

Příklady algeber:

($\mathbb{N}$, {+^{2}}) - sčítání nad množinou přirozených čísel
(2^M , {∩,∪}) - mnozina vsech podmnozin M s operacemi prunik a sjednoce

Definice: Univerzální algebra je dvojice (A, F^A) kde:

- A je nosič algebry - množina objektů (čísilek, proměnných, a čertví čeho ještě)
- F^A množina operací nad A

Nejznámější algebrou je elementární algebra zabývající se řešením rovnic a jejich soustav.

Operace je zobrazení z kartezského součinu jednich množin do kart.součinu jiných množin $A_1 \times A_2 \times A_3 \ldots \rightarrow A_i \times A_{i+1}$. V algebře se pracuje zejména s **n-ární operaci**, která zobrazuje kartezský součin množin $A_1 \times A_2 \times \ldots \times A_n \rightarrow A$. Podle ***n*** se pak dělí na:

- nulární - spíše teoretická operace bez vstupu, ale dá se pod ní představit třeba konstanta.
- unární - na vstupu má jeden prvek, který převede na druhý. Např. operace absolutní hodnoty, převod znamének.
- binární - nejčastější operace pracující s dvěmi prvky. Např. sčítání, odčítání, mocnění,...
- ternární - na vstupu ma tři prvky. Například operace IF(podmínka, true-vysledek, false-vysledek)
- n>3 ární operace se pak používají v programování a chápeme metody, funkce a procedury s více parametry.

Vlastnosti:

- komutativita - nezáleží na pořadí prvku v operaci. $a \circ^{op} b = b \circ^{op} a$
- asociativita - nezáleží na závorkách. $a \circ^{op} (b \circ^{op} c) = (b \circ^{op} a) \circ^{op} c$
- s neutrálním prvkem - který nezmění výsledek (0 u sčítání, 1 u násobení)
- s agresivním prvkem - který změní výsledek na sebe (0 u násobení)
- s inverzními prvky - každý prvek lze převést na inverzní
- distributivita - schopnost převést jednu operaci na druhou

Typy (N,∘) N-nosič, ∘-binární operace

- NxN->N uzavření na N => **Grupoid**
- Nezáleží na závorkách [asociativita]: (a ∘ b) ∘ c = y ∘ (b ∘ c) => **Pologrupa**
- Existence jednotkového prvku e **:** ∃e ∀a: a ∘ e = e ∘ a = a => **Monoid** (u **+** je to 0, u ***** je to 1)
- Inverzní prvek $\forall e \exists a^{-1}$: $a \circ a^{-1} = e = a^{-1} \circ a$ => **Grupa** (u +:1-1)
- Komutativnost- nezáleží na pořadí operací **Abelova Grupa**

Konečný grupoid - je grupoid na konečné množině A. Ten se dá vyjádřit pomocí Cayleyovy tabulky (sloupce i řádky tvoří prvky množiny A a prvky tabulky značí výsledek operace $a_i \circ^{op} a_j$)

Struktury s jednou binární operací

	Asociativita	Neutrální prvek	Inverzní prvek
Grupa	✓	✓	✓
Monoid	✓	✓	✗
Pologrupa	✓	✗	✗
Lupa	✗	✓	✓
Kvazigrupa	✗	✗	✓
Grupoid	✗	✗	✗

Okruh je **algebra s dvěmi operacemi** pro které musí platit:

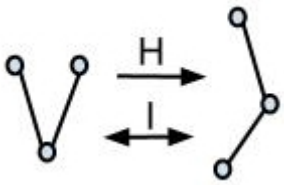
- komutativita
- asociativita
- neutrální prvek

- inverzními prvky
- distributivita

Příklad: Je (A,{+,*}) okruh?

- Komutativní + ? ANO: a+b = b+a. Komutitativní *? ANO: a*b = b*a.
- Asociativní +? ANO: a+(b+c) = (b+a)+c. Asociativní *? ANO: a*(b*c) = (b*a)*c
- Neutrálním prvky? ANO: pro + 0, pro *1.
- Inverzními prvky? ANO u sčítání je inverzní prvek k 1-a. U násobeni je 1/a
- Distributivita? ANO: a*(b+c) = (a*b)+(a*c)

Homomorfismus - zobrazení, které převádí jednu algebraickou strukturu na jinou stejného typu.



Pokud existuje zobrazení které je dokáže převést i zpátky, pak se jedná o izomorfismus.

Boolová algebra - je algebra, kde nosič A = {0,1} a operace F = {∧,∨,¬}. Jsou pro ni definovány **axiomy** pro:

- komutativitu: a ∧ b = b ∧ a. a ∨ b = b ∨ a.
- distributivitu: a ∧ (b ∨ c) = (a ∧ b) ∨ (a ∧ c). a ∨ (b ∧ c) = (a ∨ b) ∧ (a ∨ c).
- neutrální prvky: 0 ∨ a = a. 1 ∧ a = a.
- agresivní prvky: a ∧ ¬a = 0. a ∨ ¬a = 1.
- nedegerovatelnost: 0≠1

8. Obecná rezoluční metoda a logické programování: rezoluční pravidlo a algoritmus unifikace v predikátové logice 1. řádu.

Obecná rezoluční metoda a logické programování

Rezoluční metoda je metodou pro ověřování tautologií a logického vyplývání. Můžeme ji provádět buď ve výrokové logice nebo v predikátové logice. Pro osvěžení základních znalostí z logiky si můžeš pročíst kapitolu o matematické logice.

Rezoluční metoda je univerzální dokazovací metodou, kterou lze využít pro automatické strojové zpracování formulí a je základem pro logické programování (prolog). Pomocí rezoluční metody můžeme dokazovat, že formule je nesplnitelná (důkaz sporem). U výrokové logiky můžeme použít i přímý důkaz, a určit co vyplývá z daných předpokladů (odvození rezolventa).

Rezoluční metoda ve výrokové logice

Touto metodou dokazujeme ***nesplnitelnost*** dané formule (resp. množiny formulí) a je uplatnitelná na formuli v ***konjunktivní normální formě*** (KNF). Využívá dvou jednoduchých tvrzení:

1) Je-li formule A tautologie, pak formule ¬A je kontradikce a naopak.

   |= A právě když ¬A |=

2) Rezoluční pravidlo odvozování: Nechť p je literál. Z formule (A ∨ p) ∧ (B ∨ ¬p) odvodíme (A ∨ B). Zapisujeme:

$$\frac{(A \vee p) \wedge (B \vee \neg p)}{(A \vee B)}$$

Toto pravidlo není přechodem k ekvivalentní formuli, ale zachovává splnitelnost.

Příklad:

Je doma nebo odešel do kavárny.	$d \vee k$
Je-li doma, pak nás očekává.	$d \supset o \Leftrightarrow \neg d \vee o$
----------	————————
Z: Jestliže nás neočekává, pak odešel do kavárny.	$\neg o \supset k \Leftrightarrow o \vee k$

Splnitelnost formule můžeme ověřit buď přímo nebo nepřímo - sporem.

(1) ***Přímý důkaz:***

- Jednotlivé klausule napíšeme pod sebe a odvozujeme **rezolventy** (které vyplývají).

1. $d \vee k$
2. $\neg d \vee o$

   ————————
3. $o \vee k$ (1. a 2.)

⇒ z 1. a 2. formule jsme odvodili 3. řádek a zjistili jsme že se shoduje se závěrem příkladu. ***Úsudek je tudíž platný***.

(2) ***Nepřímý důkaz sporem:***

Při nepřímém se snažíme dojít k prazdné klauzuli. Při ověřování splnitelnosti negujeme závěr formule a snažíme se dojít ke sporu (prázdné klauzuli).

1. $d \vee k$
2. $\neg d \vee o$
3. $\neg o$ ***... negovaný závěr***
4. $\neg k$ ***... negovaný závěr***

   ————————
5. d (1. a 4.)
6. o (2. a 5.)
7. $\square$ (3. a 6.)

⇒ v 7. kroku jsme dostali prázdnou klauzuli. Negovaný závěr je tudíž ve sporu s předpoklady, a proto ***je úsudek platný***.

Rezoluční pravidlo a algoritmus unifikace v predikátové logice 1. řádu

Rezoluční metoda v predikátové logice je obdobou stejnojmenné metody ve výrokové logice, s tím rozdílem že je složitější. Může za to bohatější struktura predikátové logiky. Tato metoda se používá v logickém programování a je základem pro Prolog (programovací jazyk).

Při uplatňování rezoluční metody je nutné nejprve formuli převést do **Skolemovy klauzulární formy**.

Zavedení základních pojmů:

Literál - je atomická formule, nebo negace atomické formule. např.: P(x, g(y)), ¬Q(z)

Klauzule - je disjunkce literálů, např.: P(x, g(y)) ∨ ¬Q(z)

Skolemova klauzulární forma - $\forall x_1 \ldots x_n [C_1 \wedge \ldots \wedge C_m]$, uzavřená formule, kde C_i jsou klausule. Všechny kvantifikátory stojí na začátku formule a neobsahuje žádný existenční kvantifikátor.

Skolemizace - proces odstranění existenčních kvantifikátorů, je jedním z kroků převodu do skolemovy klauzulární formy

$\exists \mathbf{x}\ \forall y_1 \ldots y_n\ A(x, y_1, \ldots, y_n) \Rightarrow \forall y_1 \ldots y_n\ A(\mathbf{c}, y_1, \ldots, y_n)$,kde c je nová dosud nepoužitá konstanta

$\forall y_1 \ldots y_n\ \exists \mathbf{x}\ A(x, y_1, \ldots, y_n) \Rightarrow \forall y_1 \ldots y_n\ A(\mathbf{f(y1,\ldots,yn)}, y1, \ldots, yn)$,kde f je nový dosud nepoužitý funkční symbol

⇒ ***pokud je existenční kvantifikátor první - přepíšeme proměnnou (tu za kvantifikátorem) uvnitř formule za novou konstantu, pokud je existenční kvantifikátor druhý - přepíšeme proměnnou za funkci s atributem druhé proměnné, u obou případů smažeme při přepisu existenční kvantifikátor***

Příklad skolemizace:

$\exists \mathbf{y} \forall z\ P(z, \mathbf{y}) \wedge \forall x \exists \mathbf{u}\ Q(x, \mathbf{u}) \Leftrightarrow \forall z\ P(z, \mathbf{a}) \wedge \forall x\ Q(x, \mathbf{f(x)}) \Leftrightarrow \forall z \forall x\ [P(z,a) \wedge Q(x,f(x))]$

Převod do Skolemovy klauzulární formy

Krok 1. a 7. nejsou ekvivalentní, pouze zachovávají splnitelnost formule.

8. Utvoření existenčního uzávěru (zachovává splnitelnost)
9. Eliminace nadbytečných kvantifikátorů
10. Eliminace spojek ⊃, ≡
11. Přesun negace dovnitř
12. Přejmenování proměnných
13. Přesun kvantifikátorů doprava
14. Eliminace existenčních kvantifikátorů (Skolemizace – zachovává splnitelnost)
15. Přesun všeobecných kvantifikátorů doleva
16. Použití distributivních zákonů

Příklad převodu do Skolemovy klauzulární formy:

$\forall x \{P(x) \supset \exists z \{\neg \forall y [Q(x,y) \supset P(f(x_1))] \wedge \forall y [Q(x,y) \supset P(x)]\}\}$

1.,2. Existenční uzávěr a eliminace ∃z:

$\exists x_1 \forall x \{P(x) \supset \{\neg\forall y [Q(x,y) \supset P(f(x_1))] \wedge \forall y [Q(x,y) \supset P(x)]\}\}$

3.,4. Přejmenování y, eliminace ⊃:

$\exists x_1 \forall x \{\neg P(x) \vee \{\neg\forall y [\neg Q(x,y) \vee P(f(x_1))] \wedge \forall z [\neg Q(x,z) \vee P(x)]\}\}$

5.,6. Negace dovnitř a kvantifikátory doprava:

$\exists x_1 \forall x \{\neg P(x) \vee \{[\exists y Q(x,y) \wedge \neg P(f(x_1))] \wedge [\forall z \neg Q(x,z) \vee P(x)]\}\}$

7. Eliminace existenčních kvantifikátorů:

$\forall x \{\neg P(x) \vee \{[Q(x,g(x)) \wedge \neg P(f(a))] \wedge [\forall z \neg Q(x,z) \vee P(x)]\}\}$

8. Kvantifikátor doleva:

$\forall x \forall z \{\neg P(x) \vee \{[Q(x,g(x)) \wedge \neg P(f(a))] \wedge [\neg Q(x,z) \vee P(x)]\}\}$

9. Distributivní zákon:

$\forall x \forall z \{[\neg P(x) \vee Q(x,g(x))] \wedge [\neg P(x) \vee \neg P(f(a))] \wedge [\neg P(x) \vee \neg Q(x,z) \vee P(x)]\}$

10. zjednodušení:

$\forall x \{[\neg P(x) \vee Q(x,g(x))] \wedge [\neg P(x) \vee \neg P(f(a))]\}$

Unifikace literálů

Chceme-li rezolvovat klauzule, brání nám to, že termy / argumenty nejsou stejné. Všechny proměnné jsou ale kvantifikovány všeobecným kvantifikátorem, a tudíž můžeme použít zákon konkretizace a pokusit se najít vhodnou substituci termů za proměnné tak, abychom dostali shodné ”unifikované” literály.

Zákon konkretizace: Je-li term t substituovatelný za x ve formuli A(x), pak $\forall x A(x) \models A(x/t)$

Příklad:

1. P(x, f(x))
2. ¬P(a, z) ∨ ¬Q(z, v)
3. Q(y, h(y))

⇒ provedeme substituci

x/a, z / f(a)

1. P(a,f(a))
2. ¬P(a, f(a)) ∨ ¬Q(f(a), v)
3. Q(a, h(a))

Pro automatickou unifikaci se používají 2 algoritmy:

- **Herbrandova procedura**
- **Robinsonův unifikační algoritmus**

Postup důkazů rezoluční metodou

- Důkaz, že formule A je logicky pravdivá
    1. Formuli A znegujeme
    2. ¬A převedeme do Skolemovy klausulární formy

3. Uplatňujeme pravidla rezoluce a dokazujeme splnitelnost formule

- Důkaz platnosti úsudku $P_1,...Pn \vdash Z$
  1. Závěr znegujeme
  2. Formule $P_1,...Pn$ a $\neg Z$ převedeme do klausulární formy
  3. Uplatňujeme pravidla rezoluce a dokazujeme **ne**splnitelnost formule

Příklad rezoluční metody v PL1:

$\forall x \forall y\ [\{P(x,y) \supset Q(x, f(g(x)))\} \wedge \{R(x) \vee \exists x\ \neg Q(x, f(g(x)))\} \wedge \exists x\ \neg R(x)] \supset \exists x\ \neg P(x, g(x))$

1) Formuli znegujeme:

$\forall x \forall y\ [\{P(x,y) \supset Q(x, f(g(x)))\} \wedge \{R(x) \vee \exists x\ \neg Q(x, f(g(x)))\} \wedge \exists x\ \neg R(x)] \wedge \forall x\ P(x, g(x))$

2) Odstraníme existenční kvantifikátory a přejmenujeme:

$\forall x \forall y\ [\{P(x,y) \supset Q(x, f(g(x)))\} \wedge \{R(x) \vee \neg Q(a, f(g(a)))\} \wedge \neg R(b)] \wedge \forall z\ P(z, g(z))$

3) Odstraníme implikace:

$\forall x \forall y\ [\{\neg P(x,y) \vee Q(x, f(g(x)))\} \wedge \{R(x) \vee \neg Q(a, f(g(a)))\} \wedge \neg R(b)] \wedge \forall z\ P(z, g(z))$

4) Sepíšeme pod sebe a uplatňujeme pravidla rezoluce a unifikace literálů:

1. $\neg P(x,y) \vee Q(x, f(g(x)))$
2. $R(x) \vee \neg Q(a, f(g(a)))$
3. $\neg R(b)$
4. $P(z, g(z)$
5. **$Q(x, f(g(x)))$ 1,4: z/x, y/g(x)**
6. **$\neg Q(a, f(g(a)))$ 2,3: x/b**
7. $\square$

$\Rightarrow$ vyšla prázdná klauzule, z čehož plyne že negovaná formule je nesplnitelná (kontradikce), proto je původní formule ***logicky pravdivá***