IB013 Logické programování I

Hana Rudová

jaro 2007

Hodnocení předmětu

- **Zápočtový projekt:** celkem až 40 bodů
- **Průběžná písemná práce:** až 30 bodů (základy programování v Prologu)
  - pro každého jediný termín: **19. března**
  - alternativní termín pouze v případech závažných důvodů pro neúčast
- **Závěrečná písemná práce:** až 150 bodů
  - cca tři řádné termíny písemky, vzor písemky na webu předmětu
  - žádná opravná písemka
  - opravný termín: ústní zkouška
  - alternativní termín po dohodě jen v případech závažných důvodů pro neúčast spíše formou delší ústní zkoušky
- Hodnocení: součet bodů za projekt a za obě písemky
  - známka A za cca 175 bodů, známka F za cca 110 bodů
  - známka bude zapsána pouze těm, kteří dostanou zápočet za projekt

Základní informace

- **Přednáška:** účast není povinná
- **Cvičení:** zápočet udělen za zápočtový projekt
- **Web předmětu na IS:** Studijní materiály -> Titulní strana

  https://is.muni.cz/auth/elearning/warp.pl?fakulta=1433;obdobi=3524;kod=IB013;qurl=/el/1433/jaro2007/IB013/index.qwarp
  - průsvitky dostupné postupně v průběhu semestru
  - harmonogram výuky
  - předběžný obsah výuky pro jednotlivé přednášky během semestru
  - elektronicky dostupné materiály
- **Obsah přednášky**
  - základy programování v jazyce Prolog
  - teorie logického programováni
  - logické programování s omezujícími podmínkami
  - implementace logického programováni

Literatura

- Bratko, I. **Prolog Programming for Artificial Intelligence.** Addison-Wesley, 2001.
  - prezenčně v knihovně
- Clocksin, W. F. – Mellish, Ch. S. **Programming in Prolog.** Springer, 1994.
- Sterling, L. – Shapiro, E. Y. **The art of Prolog : advanced programming techniques.** MIT Press, 1987.
- Nerode, A. – Shore, R. A. **Logic for applications.** Springer-Verlag, 1993.
  - prezenčně v knihovně
- Dechter, R. **Constraint Processing.** Morgan Kaufmann Publishers, 2003.
  - prezenčně v knihovně

+ Elektronicky dostupné materiály (viz web předmětu)

Software: SICStus Prolog

- Doporučovaná implementace Prologu
- Dokumentace: `http://www.fi.muni.cz/~hanka/sicstus/doc/html`
- Komerční produkt

  Zakoupena licence pro instalace na domácí počítače studentů
- Podrobné informace na webu předmětu

Cvičení

- Zaměřeno na praktické aspekty, u počítačů
- Skupiny:
  - skupina 01, sudé pondělí, první cvičení **19.února**
  - skupina 02, liché pondělí, první cvičení **26.února**
- Zápočtové projekty: **Adriana Strejčková <ada@fi.muni.cz>**
  - zápočtové projekty dostupné přes web předmětu
  - podrobné pokyny k zápočtovým projektům na webu předmětu
  - vystavení projektů na webu předmětu: **do 28.února**
  - zahájení registrace řešitelů projektu: **14. března**
  - předběžná analýza řešeného problému: **4. dubna**
  - termín pro odevzdání projektů: **18. května**
  - předvádění projektů (po registraci): **28.května - 15.června**

Průběžná písemná práce

- Pro každého jediný termín **19. března**
- Alternativní termín pouze v závažných důvodech pro neúčast
- Celkem až 30 bodů (150 závěrečná písemka, 40 projekt)
- 3 příklady, 40 minut
- Napsat zadaný predikát, porovnat chování programů
- Oblasti, kterých se budou příklady zejména týkat
  - unifikace
  - backtracking
  - řez
  - seznamy
  - optimalizace posledního volání
  - aritmetika
- Ukázka průběžné písemné práce na webu

Úvod do Prologu

Prolog

- PROgramming in LOGic
  - část predikátové logiky prvního řádu
- Deklarativní programování
  - specifikační jazyk, jasná sémantika, nevhodné pro procedurální postupy
  - **Co dělat** namísto **Jak dělat**
- Základní mechanismy
  - unifikace, stromové datové struktury, automatický backtracking

Prolog: historie a současnost

- Rozvoj začíná po roce 1970
  - Robert Kowalski – teoretické základy
  - Alain Colmerauer, David Warren *(Warren Abstract Machine)* – implementace
  - pozdější rozšíření Prologu o logické programování s omezujícími podmínkami
- Prolog v současnosti
  - zavedené aplikační oblasti, nutnost přidání inteligence
    - hypotéky; pediatrický sw; konfigurace a pravidla pro stanovení ceny objednávky; testovací nástroje, modelové testování; ...
  - náhrada procedurálního kódu Prologem vede k
    - desetinásobnému zmenšení kódu, řádově menšímu času na vývoj, jednodušší údržbě
  - efektivita Prologu?
    - zrychlení počítačů + výrazné zvětšení nároků sw

    ⇒ ve prospěch kompaktnosti i rychlosti Prologu

Program = fakta + pravidla

- **(Prologovský) program** je **seznam programových klauzulí**
  - programové klauzule: fakt, pravidlo
- **Fakt:** deklaruje vždy pravdivé věci
  - `clovek( novak, 18, student ).`
- **Pravidlo:** deklaruje věci, jejichž pravdivost závisí na daných podmínkách
  - `studuje( X ) :- clovek( X, _Vek, student ).`
  - **alternativní (obousměrný) význam pravidel**

pro každé X,	pro každé X,
X studuje, jestliže	X je student, potom
X je student	X studuje

  - `pracuje( X ) :- clovek( X, _Vek, CoDela ), prace( CoDela ).`
- **Predikát:** množina pravidel a faktů se stejným **funktorem** a **aritou**
  - značíme: `clovek/3`, `student/1`; analogie **procedury** v procedurálních jazycích,

Komentáře k syntaxi

- Klauzule ukončeny tečkou
- Základní příklady argumentů
  - **konstanty**: (`tomas, anna`) ... začínají malým písmenem
  - **proměnné**
    - `X, Y` ... začínají velkým písmenem
    - `_, _A, _B` ... začínají podtržítkem (nezajímá nás vracená hodnota)
- Psaní komentářů

```
clovek( novak, 18, student ).        % komentář na konci řádku
clovek( novotny, 30, ucitel ).         /* komentář */
```

Dotaz

- **Dotaz**: uživatel se ptá programu, zda jsou věci pravdivé

```
?- studuje( novak).        % yes     splnitelný dotaz
?- studuje( novotny).      % no      nesplnitelný dotaz
```

- **Odpověď** na dotaz
  - positivní – **dotaz je splnitelný a uspěl**
  - negativní – **dotaz je nesplnitelný a neuspěl**
- Proměnné jsou během výpočtu **instanciovány** (= nahrazeny objekty)
  - `?- clovek( novak, 18, Prace ).`
  - výsledkem dotazu je **instanciace proměnných** v dotazu
  - dosud nenainstanciovaná proměnná: **volná proměnná**
- Prolog umí generovat více odpovědí pokud existují

```
?- clovek( novak, Vek, Prace ).      % všechna řešení přes ";"
```


Klauzule = fakt, pravidlo, dotaz

- **Klauzule** se skláda z **hlavy** a **těla**
- Tělo je **seznam cílů** oddělených čárkami, čárka = konjunkce
- **Fakt**: pouze hlava, prázdné tělo
  - `rodic( pavla, robert ).`
- **Pravidlo**: hlava i tělo
  - `upracovany_clovek( X ) :- clovek( X, _Vek, Prace ), prace( Prace, tezka ).`
- **Dotaz**: prázdná hlava, pouze tělo
  - `?- clovek( novak, Vek, Prace ).`
    `?- rodic( pavla, Dite ), rodic( Dite, Vnuk ).`

Rekurzivní pravidla

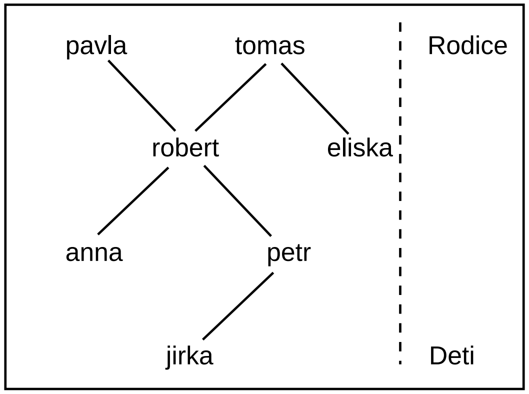
```
predek( X, Z ) :- rodic( X, Z ).          % (1)

predek( X, Z ) :- rodic( X, Y ),          % (2)
                  rodic( Y, Z ).

predek( X, Z ) :- rodic( X, Y ),          % (2')
                  predek( Y, Z ).
```


Příklad: rodokmen

```
rodic( pavla, robert ).
rodic( tomas, robert ).
rodic( tomas, eliska ).
rodic( robert, anna ).
rodic( robert, petr ).
rodic( petr, jirka ).
```

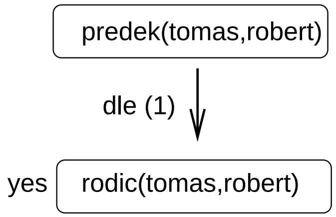



```
predek( X, Z ) :- rodic( X, Z ).          % (1)

predek( X, Z ) :- rodic( X, Y ),          % (2')
                  predek( Y, Z ).
```

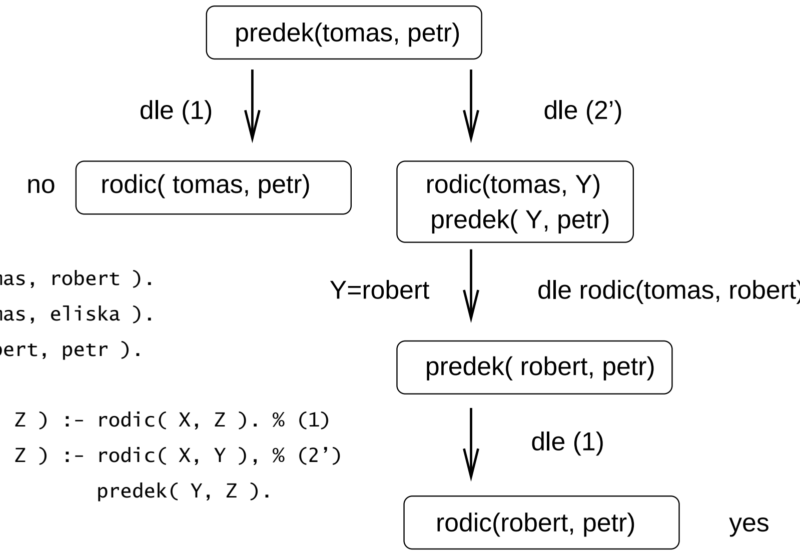
Výpočet odpovědi na dotaz ?– predek(tomas,robert)

```
rodic( pavla, robert ).
rodic( tomas, robert ).
rodic( tomas, eliska ).
rodic( robert, anna ).
rodic( robert, petr ).
rodic( petr, jirka ).
```




```
predek( X, Z ) :- rodic( X, Z ).      % (1)
predek( X, Z ) :- rodic( X, Y ),      % (2’)
                  predek( Y, Z ).
```


Výpočet odpovědi na dotaz ?- predek(tomas, petr)

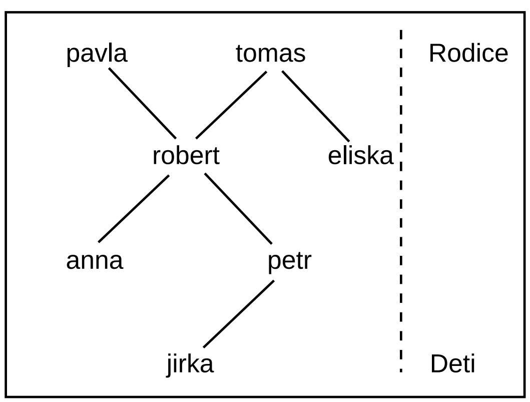



```
rodic( tomas, robert ).
rodic( tomas, eliska ).
rodic( robert, petr ).

predek( X, Z ) :- rodic( X, Z ). % (1)
predek( X, Z ) :- rodic( X, Y ), % (2’)
                  predek( Y, Z ).
```


Odpověď na dotaz ?- predek(robert, Potomek)

```
rodic( pavla, robert ).
rodic( tomas, robert ).
rodic( tomas, eliska ).
rodic( robert, anna ).
rodic( robert, petr ).
rodic( petr, jirka ).
```




```
predek( X, Z ) :- rodic( X, Z ).     % (1)
predek( X, Z ) :- rodic( X, Y ),     % (2’)
                  predek( Y, Z ).

predek(robert,Potomek) --> ???
```


Syntaxe a význam Prologovských programů

Syntaxe Prologovských programů

- **Typy objektů jsou rozpoznávány podle syntaxe**
- **Atom**
  - řetězce písmen, čísel, „_” začínající malým písmenem: `pavel, pavel_novak, x25`
  - řetězce speciálních znaků: `<-->, ====>`
  - řetězce v apostrofech: `'Pavel', 'Pavel Novák'`
- Celá a reálná **čísla**: `0, -1056, 0.35`
- **Proměnná**
  - řetězce písmen, čísel, „_” začínající velkým písmenem nebo „_”
  - **anonymní proměnná**: `ma_dite(X) :- rodic( X, _ ).`
    - hodnotu anonymní proměnné Prolog na dotaz nevrací: `?- rodic( X, _ )`
  - lexikální rozsah proměnné je pouze jedna klauzule:

```
prvni(X,X,X).
prvni(X,X,_).
```


Termy

- **Term** – datové objekty v Prologu: `datum( 1, kveten, 2003 )`
  - **funktor**: `datum`
  - **argumenty**: `1, kveten, 2003`
  - **arita** – počet argumentů: 3
- Všechny strukturované objekty v Prologu jsou **stromy**
  - `trojuhelnik( bod(4,2), bod(6,4), bod(7,1) )`
- **Hlavní funktor** termu – funktor v kořenu stromu odpovídající termu
  - `trojuhelnik` je hlavní funktor v `trojuhelnik( bod(4,2), bod(6,4), bod(7,1) )`

Unifikace

- Termy jsou **unifikovatelné**, jestliže
  - jsou identické nebo
  - proměnné v obou termech mohou být instanciovány tak,
    že termy jsou po substituci identické
  - `datum( D1, M1, 2003 ) = datum( 1, M2, Y2)` **operátor =**
    `D1 = 1, M1 = M2, Y2 = 2003`
- **Nejobecnější unifikátor** *(most general unifier (MGU)*
  - jiné instanciace? `...D1 = 1, M1 = 5, Y2 = 2003`
  - `?- datum( D1, M1, 2003 ) = datum( 1, M2, Y2), D1 = M1.`
- **Test výskytu** *(occurs check)*

```
?- X=f(X).
X = f(f(f(f(f(f(f(f(f(f(...))))))))))
```


Unifikace

Termy S a T jsou unifikovatelné, jestliže

1. S a T jsou konstanty a tyto konstanty jsou identické;
2. S je proměnná a T cokoliv jiného – S je instanciována na T;
   T je proměnná a S cokoliv jiného – T je instanciována na S
3. S a T jsou termy
   - S a T mají stejný funktor a aritu a
   - všechny jejich odpovídající argumenty jsou unifikovatelné
   - výsledná substituce je určena unifikací argumentů

Příklady:

```
k = k ... yes, k1 = k2 ... no, A = k(2,3) ... yes, k(s,a,l(1)) = A ... yes
s(sss(2),B,ss(2)) = s(sss(2),4,ss(2),s(1))... no
s(sss(A),4,ss(3)) = s(sss(2),4,ss(A))... no
s(sss(A),4,ss(C)) = s(sss(t(B)),4,ss(A))... A=t(B),C=t(B)... yes
```

Deklarativní a procedurální význam programů

- `p :- q, r.`
- Deklarativní: **Co** je výstupem programu?
  - p je pravdivé, jestliže q a r jsou pravdivé
  - Z q a r plyne p

  ⇒ význam mají logické relace
- Procedurální: **Jak** vypočítáme výstup programu?
  - p vyřešíme tak, že **nejprve** vyřešíme q a **pak** r

  ⇒ kromě logických relací je významné i pořadí cílů
  - výstup
    - indikátor yes/no určující, zda byly cíle splněny
    - instanciace proměnných v případě splnění cílů

Deklarativní význam programu

Máme-li program a cíl G, pak **deklarativní význam** říká:

cíl G je splnitelný právě tehdy, když cíl `?- ma_dite(petr).`

existuje klauzule C v programu taková, že

existuje instance I klauzule C taková, že

hlava I je identická s G a

všechny cíle v těle I jsou pravdivé.

Instance klauzule: proměnné v klauzuli jsou substituovány termem

- `ma_dite(X) :- rodic( X, Y ).` % klauzule

  `ma_dite(petr) :- rodic( petr, Z ).` % instance klauzule

Konjunce "," vs. disjunkce ";" cílů

- **Konjunce** = nutné splnění **všech cílů**
  - `p :- q, r.`
- **Disjunkce** = stačí splnění **libovolného cíle**
  - `p :- q; r.` `p :- q.` `p :- r.`
  - priorita středníku je vyšší:

```
p :- q, r; s, t, u.

p :- (q, r) ; (s, t, u).

p :- q, r.
p :- s, t, u.
```


Pořadí klauzulí a cílů

```
(a) a(1).                                   ?- a(1).
    a(X) :- b(X,Y), a(Y).
    b(1,1).

(b) a(X) :- b(X,Y), a(Y).     % změněné pořadí klauzulí v programu vzhledem k (a)
    a(1).
    b(1,1).                   % nenalezení odpovědi: nekonečný cyklus
```

```
(c) a(X) :- b(X,Y), c(Y).                   ?- a(X).
    b(1,1).
    c(2).
    c(1).

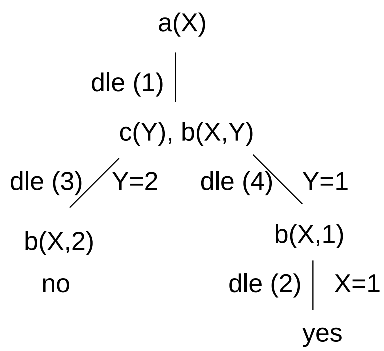
(d) a(X) :- c(Y), b(X,Y).     % změněné pořadí cílů v těle klauzule vzhledem k (c)
    b(1,1).
    c(2).
    c(1).                     % náročnější nalezení první odpovědi než u (c)
```

V obou případech **stejný deklarativní ale odlišný procedurální význam**

Pořadí klauzulí a cílů II.

```
(1) a(X) :- c(Y), b(X,Y).
(2) b(1,1).
(3) c(2).
(4) c(1).
```

`?- a(X).`



Vyzkoušejte si:

```
a(X) :- b(X,X), c(X).
a(X) :- b(X,Y), c(X).
b(2,2).
b(2,1).
c(1).
```

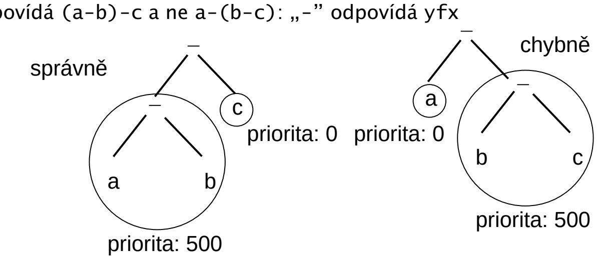

Operátory, aritmetika

Operátory

- Infixová notace: `2*a + b*c`
- Prefixová notace: `+( *(2,a), *(b,c) )` priorita +: 500, priorita *: 400
- **Priorita operátorů**: operátor s **nejvyšší** prioritou je hlavní funktor
- Uživatelsky definované operátory: zna
  `petr zna alese.` `zna( petr, alese).`
- Definice operátoru: `:- op( 600, xfx, zna ).` priorita: 1..1200
  - `:- op( 1100, xfy, ; ).` nestrukturované objekty: 0
    `:- op( 1000, xfy, , ).`
    `p :- q,r; s,t.` `p :- (q,r) ; (s,t).` ; má vyšší prioritu než ,
  - `:- op( 1200, xfx, :- ).` :- má nejvyšší prioritu
- Definice operátoru není spojena s datovými manipulacemi
  (kromě speciálních případů)

Typy operátorů

- Typy operátorů
  - infixové operátory: `xfx, xfy, yfx` př. xfx = yfx -
  - prefixové operátory: `fx, fy` př. fx ?- fy -
  - postfixové operátory: `xf, yf`
- x a y určují **prioritu argumentu**
  - x reprezentuje argument, jehož priorita musí být **striktně menší** než u operátoru
  - y reprezentuje argument, jehož priorita je **menší nebo rovna** operátoru
  - `a-b-c` odpovídá `(a-b)-c` a ne `a-(b-c)`: „-” odpovídá `yfx`



Aritmetika

- Předdefinované operátory

  `+, -, *, /,` `**` mocnina, `//` celočíselné dělení, mod zbytek po dělení

- `?- X = 1 + 2.` `X = 1 + 2` = odpovídá unifikaci
- `?- X is 1 + 2.`
  `X = 3` „`is`" **je speciální předdefinovaný operátor**, který vynutí evaluaci
  - porovnej: `N = (1+1+1+1+1)` `N is (1+1+1+1+1)`
  - pravá strana musí být vyhodnotitelný výraz (bez proměnné)

    volání `?- X is Y + 1.` způsobí chybu
- Další speciální předdefinované operátory

  `>, <, >=, =<,` `=:=` **aritmetická rovnost,** `=\=` **aritmetická nerovnost**
  - porovnej: `1+2 =:= 2+1` `1+2 = 2+1`
  - obě strany musí být vyhodnotitelný výraz: volání `?- 1 < A + 2.` způsobí chybu

Různé typy rovností a porovnání

`X = Y`	X a Y jsou unifikovatelné
`X \= Y`	X a Y nejsou unifikovatelné, (také `\+ X = Y`)
`X == Y`	X a Y jsou identické
	porovnej: ?- A == B. . . . no ?- A=B, A==B. . . . B = A yes
`X \== Y`	X a Y nejsou identické
	porovnej: ?- A \== B. . . . yes ?- A=B, A \== B. . . . A no
`X is Y`	Y je aritmeticky vyhodnoceno a výsledek je přiřazen X
`X =:= Y`	X a Y jsou si aritmeticky rovny
`X =\= Y`	X a Y si aritmeticky nejsou rovny
`X < Y`	aritmetická hodnota X je menší než Y (`=<, >, >=`)
`X @< Y`	term X předchází term Y (`@=<, @>, @>=`)
	1. porovnání termů: podle alfabetického n. aritmetického uspořádání
	2. porovnání struktur: podle arity, pak hlavního funktoru a pak zleva podle argumentů
	?- f(pavel, g(b)) @< f(pavel, h(a)). . . . yes

Prolog: příklady

Příklad: průběh výpočtu

```
a :- b,c,d.
b :- e,c,f,g.
b :- g,h.
c.
d.
e :- i.
e :- h.
g.
h.
i.
```

Jak vypadá průběh výpočtu pro dotaz `?- a.`

Příklad: věž z kostek

Příklad: postavte věž zadané velikosti ze tří různě velkých kostek tak, že kostka smí ležet pouze na větší kostce.

```
kostka(mala).  kostka(stredni).  kostka(velka).

vetsi(zeme,velka).  vetsi(zeme,stredni).  vetsi(zeme,mala).
vetsi(velka,stredni).  vetsi(velka,mala).
vetsi(stredni,mala).

% ?- postav_vez(vez(zeme,0), vez(Kostka,0)).
% ?- postav_vez(vez(zeme,0), vez(Kostka,3)).
postav_vez( Vez, Vez ).
postav_vez( Vstup, Vystup ) :- pridej_kostku( Vstup, Pridani ),
                               postav_vez( Pridani, Vystup ).

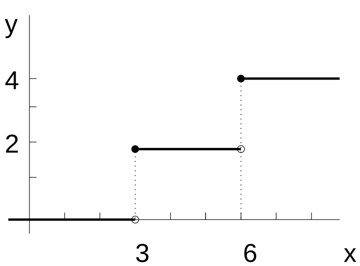
pridej_kostku( Vstup, Pridani ) :- Vstup = vez( Vrchol, Vyska ),
                                   kostka( Kostka ),
                                   vetsi( Vrchol, Kostka ),
                                   NovaVyska is Vyska + 1,
                                   Pridani = vez( Kostka, NovaVyska ).
```


Řez, negace

Řez a upnutí

```
f(X,0) :- X < 3, !.
f(X,2) :- 3 =< X, X < 6, !.
f(X,4) :- 6 =< X.
```

přidání **operátoru řezu** ,,!''




```
?- f(1,Y), Y>2.

f(X,0) :- X < 3, !.  %(1)
f(X,2) :- X < 6, !.  %(2)
f(X,4).

?- f(1,Y).
```

- Smazání řezu v (1) a (2) změní chování programu
- **Upnutí:** po splnění podcílů před řezem se už další klauzule neuvažují

Řez a ořezání

```
f(X,Y) :- s(X,Y).
s(X,Y) :- Y is X + 1.
s(X,Y) :- Y is X + 2.

?- f(1,Z).
Z = 2 ? ;
Z = 3 ? ;
no
```

```
f(X,Y) :- s(X,Y), !.
s(X,Y) :- Y is X + 1.
s(X,Y) :- Y is X + 2.

?-  f(1,Z).
Z = 2 ? ;
no
```

- **Ořezání:** po splnění podcílů před řezem se už neuvažuje další možné splnění těchto podcílů
- Smazání řezu změní chování programu

Chování operátoru řezu

- Předpokládejme, že klauzule `H :- T1, T2, ..., Tm, !, ...Tn.` je aktivována voláním cíle G, který je unifikovatelný s H. G=h(X,Y)
- V momentě, kdy je nalezen řez, existuje řešení cílů T1, ..., Tm X=1,Y=1
- **Ořezání:** při provádění řezu se už další možné splnění cílů T1, ..., Tm nehledá a všechny ostatní alternativy jsou odstraněny Y=2
- **Upnutí:** dále už nevyvolávám další klauzule, jejichž hlava je také unifikovatelná s G X=2

```
?- h(X,Y).

h(1,Y) :- t1(Y), !.
h(2,Y) :- a.

t1(1) :- b.
t1(2) :- c.
```

```
          h(X,Y)
    X=1  /    \  X=2
      t1(Y)     a   (vynechej: upnutí)
  Y=1 /    \ Y=2
    b        c   (vynechej: ořezání)
  /
```


Řez: příklad

```
c(X) :- p(X).                 c1(X) :- p(X), !.
c(X) :- v(X).                 c1(X) :- v(X).

            p(1).  p(2).        v(2).

?- c(2).                      ?- c1(2).
true ? ;  %p(2)               true ?  ; %p(2)
true ? ;  %v(2)               no
no

?- c(X).                      ?- c1(X).
X = 1 ? ; %p(1)               X = 1 ?  ; %p(1)
X = 2 ? ; %p(2)               no
X = 2 ? ; %v(2)
no
```


Řez: cvičení

1. Porovnejte chování uvedených programů pro zadané dotazy.

```
a(X,X) :- b(X).        a(X,X) :- b(X),!.      a(X,X) :- b(X),c.
a(X,Y) :- Y is X+1.    a(X,Y) :- Y is X+1.    a(X,Y) :- Y is X+1.
b(X) :- X > 10.        b(X) :- X > 10.        b(X) :- X > 10.
                                              c :- !.

?- a(X,Y).
?- a(1,Y).
?- a(11,Y).
```

2. Napište predikát pro výpočet maxima `max( X, Y, Max )`

Typy řezu

- Zlepšení efektivity programu: určíme, které alternativy nemá smysl zkoušet
- **Zelený řez:** odstraní pouze neúspěšná odvození
  - `f(X,1) :- X >= 0, !.  f(X,-1) :- X < 0.` bez řezu zkouším pro nezáporná čísla 2. klauzuli
- **Modrý řez:** odstraní redundantní řešení
  - `f(X,1) :- X >= 0, !.  f(0,1). f(X,-1) :- X < 0.` bez řezu vrací f(0,1) 2x
- **Červený řez:** odstraní úspěšná řešení
  - `f(X,1) :- X >= 0, !.  f(_X,-1).` bez řezu uspěje 2. klauzule pro nezáporná čísla

Negace jako neúspěch

- **Speciální cíl pro nepravdu (neúspěch)** `fail` **a pravdu** `true`
- X a Y nejsou unifikovatelné: `different(X, Y)`
- `different( X, Y ) :- X = Y, !, fail.`
  `different( _X, _Y ).`
- X je muž: `muz(X)`

  `muz( X ) :- zena( X ), !, fail.`
  `muz( _X ).`

Negace jako neúspěch: operátor \+

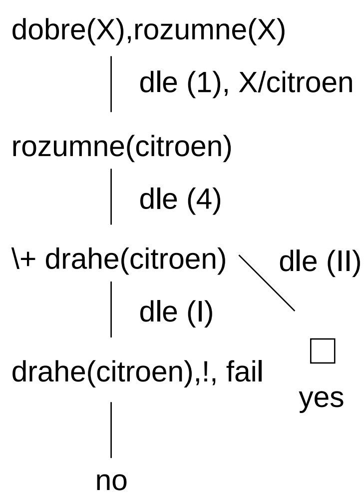
- `different(X,Y) :- X = Y, !, fail.`
  `different(_X,_Y).`
  `muz(X) :- zena(X), !, fail.`
  `muz(_X).`
- Unární operátor \+ P
  - jestliže P uspěje, potom \+ P neuspěje
    `\+(P) :- P, !, fail.`
  - v opačném případě \+ P uspěje
    `\+(_).`
- `different( X, Y ) :- \+ X=Y.`
- `muz( X ) :- \+ zena( X ).`
- Pozor: takto definovaná negace \+P vyžaduje **konečné odvození** P

Negace a proměnné

```
\+(P) :- P, !, fail. % (I)
\+(_).               % (II)

dobre( citroen ).         % (1)
dobre( bmw ).             % (2)
drahe( bmw ).             % (3)
rozumne( Auto ) :-        % (4)
   \+ drahe( Auto ).

?- dobre( X ), rozumne( X ).
```

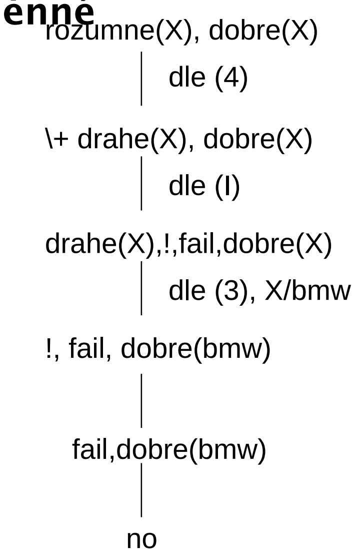


Negace a proměnné

```
\+(P) :- P, !, fail. % (I)
\+(_).               % (II)

dobre( citroen ).         % (1)
dobre( bmw ).             % (2)
drahe( bmw ).             % (3)
rozumne( Auto ) :-        % (4)
   \+ drahe( Auto ).

?- rozumne( X ), dobre( X ).
```



Bezpečný cíl

- `?- rozumne( citroen ).` yes

  `?- rozumne( X ).` no
- `?- \+ drahe( citroen ).` yes

  `?- \+ drahe( X ).` no
- `\+ P` **je bezpečný: proměnné** P **jsou v okamžiku volání** P **instanciovány**
  - negaci používáme pouze pro bezpečný cíl P

Chování negace

- `?- \+ drahe( citroen ).` yes

  `?- \+ drahe( X ).` no
- Negace jako neúspěch používá **předpoklad uzavřeného světa**

  pravdivé je pouze to, co je dokazatelné
- `?- \+ drahe( X ).` \+ drahe(X) :- drahe(X),!,fail. \+ drahe(X).

  z definice \+ plyne: není dokazatelné, že existuje X takové, že drahe(X) platí

  tj. **pro všechna** X platí `\+ drahe( X )`
- `?- drahe( X ).`

  VÍME: existuje X takové, že `drahe( X )` platí
- ALE: pro cíle s negací neplatí **existuje** X takové, že `\+ drahe( X )`

⇒ **negace jako neúspěch není ekvivalentní negaci v matematické logice**

Predikáty na řízení běhu programu I.

- **řez „!"**
- `fail`: cíl, který vždy neuspěje `true`: cíl, který vždy uspěje
- \+ P: negace jako neúspěch

  `\+ P :- P, !, fail; true.`
- `once(P)`: vrátí pouze jedno řešení cíle P

  `once(P) :- P, !.`
- Vyjádření **podmínky:** `P -> Q ; R`
  - jestliže platí P tak Q `(P -> Q ; R) :- P, !, Q.`
  - v opačném případě R `(P -> Q ; R) :- R.`
  - příklad: `min(X,Y,Z) :- X =< Y -> Z = X ; Z = Y.`
- `P -> Q`
  - odpovídá: `(P -> Q; fail)`
  - příklad: `zaporne(X) :- number(X) -> X < 0.`

Predikáty na řízení běhu programu II.

- `call(P)`: zavolá cíl P a uspěje, pokud uspěje P
- nekonečná posloupnost backtrackovacích voleb: `repeat`

```
repeat.
repeat :- repeat.
```

klasické použití: **generuj akci** X**, proveď ji a otestuj, zda neskončit**

```
Hlava :- ...
         uloz_stav( StaryStav ),
         repeat,
           generuj( X ),        % deterministické: generuj, provadej, testuj
           provadej( X ),
           testuj( X ),
         !,
         obnov_stav( StaryStav ),
         ...
```

Seznamy

Reprezentace seznamu

- **Seznam**: `[a, b, c]`, prázdný seznam `[]`
- **Hlava (libovolný objekt)**, **tělo (seznam)**: `.(Hlava, Telo)`
  - všechny strukturované objekty stromy – i seznamy
  - funktor ".", dva argumenty
  - `.(a, .(b, .(c, []))) = [a, b, c]`
  - notace: `[ Hlava | Telo ] = [a|Telo]`

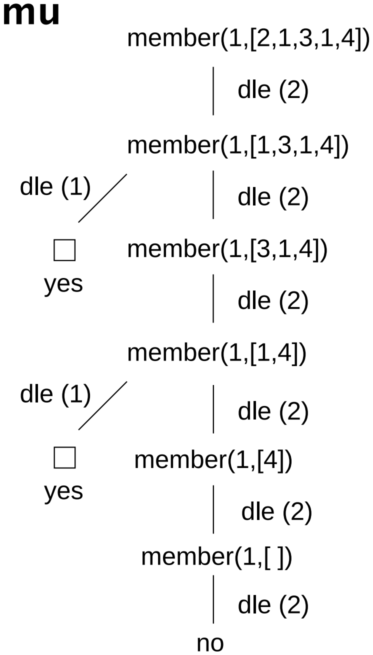
    `Telo` je v `[a|Telo]` seznam, tedy píšeme [a, b, c] = [a | [b, c]]
- Lze psát i: `[a,b|Telo]`
  - před "|" je libovolný počet prvků seznamu , za "|" je seznam zbývajících prvků
  - `[a,b,c] = [a|[b,c]] = [a,b|[c]] = [a,b,c|[]]`
  - pozor: `[ [a,b] | [c] ]` ≠ `[ a,b | [c] ]`
- Seznam jako **neúplná datová struktura:** `[a,b,c|T]`
  - `Seznam = [a,b,c|T], T = [d,e|S], Seznam = [a,b,c,d,e|S]`

Prvek seznamu

- `member( X, S )`
- platí: `member( b, [a,b,c] ).`
- neplatí: `member( b, [[a,b]|[c]] ).`
- X je prvek seznamu S, když
  - X je hlava seznamu S nebo

    `member( X, [ X | _ ] ).  %(1)`
  - X je prvek těla seznamu S

    ```
    member( X, [ _ | Telo ] ) :-
            member( X, Telo ).   %(2)
    ```
- Další příklady použití:
  - `member(X,[1,2,3]).`
  - `member(1,[2,1,3,1]).`



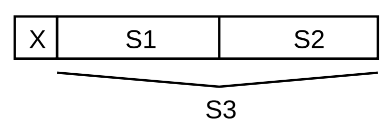
Spojení seznamů

- `append( L1, L2, L3 )`
- Platí: `append( [a,b], [c,d], [a,b,c,d] )`
- Neplatí: `append( [b,a], [c,d], [a,b,c,d] ),`

  `append( [a,[b]], [c,d], [a,b,c,d] )`
- Definice:
  - pokud je 1. argument prázdný seznam, pak 2. a 3. argument jsou stejné seznamy:

    `append( [], S, S ).`
  - pokud je 1. argument neprázdný seznam, pak má 3. argument stejnou hlavu jako 1.:

    `append( [X|S1], S2, [X|S3] ) :- append( S1, S2, S3).`



Příklady použití append

- append([], S, S).
  append([X|S1], S2, [X|S3]) :- append(S1, S2, S3).
- **Spojení seznamů:** append([a,b,c], [1,2,3], S).
  S = [a,b,c,1,2,3]
  append([a, [b,c], d], [a, [], b], S).
  S = [a, [b,c], d, a, [], b]]
- **Dekompozice seznamu na dva seznamy:** append(S1, S2, [a, b]).
  S1 = [], S2 = [a,b] ;
  S1 = [a], S2 = [b] ? ;
  S1 = [a,b], S2 = []
- **Vyhledávání v seznamu:** append(Pred, [c | Za], [a,b,c,d,e]).
  Pred = [a,b], Za = [d,e]
- **Předchůdce a následník:** append(_, [Pred,c,Za|_], [a,b,c,d,e]).
  Pred = b, Za = d

Smazání prvku seznamu delete(X, S, S1)

- Seznam S1 odpovídá seznamu S, ve kterém je smazán prvek X
  - jestliže X je hlava seznamu S, pak výsledkem je tělo S
    delete(X, [X|Telo], Telo).
  - jestliže X je v těle seznamu, pak X je smazán až v těle
    delete(X, [Y|Telo], [Y|Telo1]) :- delete(X, Telo, Telo1).
- delete smaže libovolný výskyt prvku pomocí backtrackingu
  ?- delete(a, [a,b,a,a], S).
  S = [b,a,a];
  S = [a,b,a];
  S = [a,b,a]
- delete, který smaže pouze první výskyt prvku X
  - delete(X, [X|Telo], Telo) :- !.
    delete(X, [Y|Telo], [Y|Telo1]) :- delete(X, Telo, Telo1).

Optimalizace posledního volání

- **Last Call Optimization (LCO)**
- Implementační technika snižující nároky na paměť
- Mnoho vnořených rekurzivních volání je náročné na paměť
- Použití LCO umožňuje vnořenou rekurzi s konstantními paměťovými nároky
- Typický příklad, kdy je možné použití LCO:
  - procedura musí mít pouze jedno rekurzivní volání: **v posledním cíli poslední klauzule**
  - cíle předcházející tomuto rekurzivnímu volání musí být **deterministické**
  - ```
    p( ... ) :- ...          % žádné rekurzivní volání v těle klauzule
    p( ... ) :- ...          % žádné rekurzivní volání v těle klauzule
    ...
    p(...) :- ..., !, p( ... ). % řez zajišt'uje determinismus
    ```
- Tento typ **rekurze lze převést na iteraci**

LCO a akumulátor

- Reformulace rekurzivní procedury, aby umožnila LCO
- Výpočet délky seznamu length(Seznam, Delka)
  ```
  length( [], 0 ).
  length( [ H | T ], Delka ) :- length( T, Delka0 ), Delka is 1 + Delka0.
  ```
- Upravená procedura, tak aby umožnila LCO:
  ```
  % length( Seznam, ZapocitanaDelka, CelkovaDelka ):
  %          CelkovaDelka = ZapocitanaDelka + ,,počet prvků v Seznam''

  length( Seznam, Delka ) :- length( Seznam, 0, Delka ). % pomocný predikát

  length( [], Delka, Delka ).              % celková délka = započítaná délka
  length( [ H | T ], A, Delka ) :- A0 is A + 1, length( T, A0, Delka ).
  ```
- Přídavný argument se nazývá **akumulátor**
  ```

`max_list` s akumulátorem

Výpočet největšího prvku v seznamu `max_list(Seznam, Max)`

```
max_list([X], X).

max_list([X|T], Max) :-
   max_list(T,MaxT),
   ( MaxT >= X, !, Max = MaxT
   ;
     Max = X ).
```

```
max_list([H|T],Max) :- max_list(T,H,Max).

max_list([], Max, Max).
max_list([H|T], CastecnyMax, Max) :-
   ( H > CastecnyMax, !,
     max_list(T, H, Max )
   ;
     max_list(T, CastecnyMax, Max) ).
```


Akumulátor jako seznam

- Nalezení seznamu, ve kterém jsou prvky v opačném pořadí
  `reverse( Seznam, OpacnySeznam )`
  - ```
    reverse( [], [] ).
    reverse( [ H | T ], Opacny ) :-
             reverse( T, OpacnyT ),
             append( OpacnyT, [ H ], Opacny ).
    ```
  - naivní reverse s kvadratickou složitostí
- reverse pomocí akumulátoru s lineární složitostí
  - ```
    % reverse( Seznam, Akumulator, Opacny ):
    %   Opacny  obdržíme přidáním prvků ze  Seznam  do  Akumulator  v opacnem poradi
    reverse( Seznam, OpacnySeznam ) :- reverse( Seznam, [], OpacnySeznam).

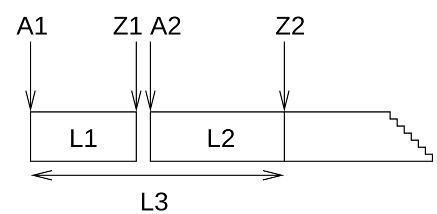
    reverse( [], S, S ).
    reverse( [ H | T ], A, Opacny ) :-
         reverse( T, [ H | A ], Opacny ).                 % přidání H do akumulátoru
    ```
  - zpětná konstrukce seznamu (srovnej s předchozí dopřednou konstrukcí, např. append)

Neefektivita při spojování seznamů

- Sjednocení dvou seznamů
- `append( [], S, S ).`
  `append( [X|S1], S2, [X|S3] ) :- append( S1, S2, S3 ).`
- `?- append( [2,3], [1], S ).`
  postupné volání cílů:
  append([2,3], [1], S) → append([3], [1], S’) → append([], [1], S’’)
- Vždy je nutné projít celý první seznam

Rozdílové seznamy

- Zapamatování konce a připojení na konec: **rozdílové seznamy**
- `[a,b] = L1-L2 = [a,b|T]-T = [a,b,c|S]-[c|S] = [a,b,c]-[c]`
- Reprezentace prázdného seznamu: `L-L`




```
append( A1-Z1, Z1-Z2, A1-Z2 ).
        L1     L2     L3
```

- `?- append( [2,3|Z1]-Z1, [1|Z2]-Z2, S ).`
  `S = A1 - Z2 = [2,3|Z1] - Z2 = [2,3| [1|Z2] ] - Z2`
  `Z1 = [1|Z2]     S = [2,3,1|Z2]-Z2`
- Jednotková složitost, oblíbená technika ale není tak flexibilní

Akumulátor vs. rozdílové seznamy: `reverse`

```
reverse( [], [] ).
reverse( [ H | T ], Opacny ) :-
        reverse( T, OpacnyT ),
        append( OpacnyT, [ H ], Opacny ).              kvadratická složitost
```

```
reverse( Seznam, Opacny ) :- reverse0( Seznam, [], Opacny ).
reverse0( [], S, S ).
reverse0( [ H | T ], A, Opacny ) :-
         reverse0( T, [ H | A ], Opacny ).             akumulátor (lineární)
```

```
reverse( Seznam, Opacny ) :- reverse0( Seznam, Opacny-[]).
reverse0( [], S-S ).
reverse0( [ H | T ], Opacny-OpacnyKonec  ) :-                rozdílové seznamy
         reverse0( T, Opacny-[ H | OpacnyKonec] ).                   (lineární)
```

Příklad: operace pro manipulaci s frontou

- test na prázdnost, přidání na konec, odebrání ze začátku

Vestavěné predikáty

Vestavěné predikáty

- Predikáty pro řízení běhu programu
  - `fail`, `true`, ...
- Různé typy rovností
  - unifikace, aritmetická rovnost, ...
- Databázové operace
  - změna programu (programové databáze) za jeho běhu
- Vstup a výstup
- Všechna řešení programu
- Testování typu termu
  - proměnná?, konstanta?, struktura?, ...
- Konstrukce a dekompozice termu
  - argumenty?, funktor?, ...

Databázové operace

- Databáze: specifikace množiny relací
- Prologovský program: **programová databáze**, kde jsou relace specifikovány explicitně (fakty) i implicitně (pravidly)
- Vestavěné predikáty pro změnu databáze během provádění programu:

`assert( Klauzule )`	přidání `Klauzule` do programu
`asserta( Klauzule )`	přidání na začátek
`assertz( Klauzule )`	přidání na konec
`retract( Klauzule )`	smazání klauzule unifikovatelné s `Klauzule`

- Pozor: nadměrné použití těchto operací snižuje srozumitelnost programu

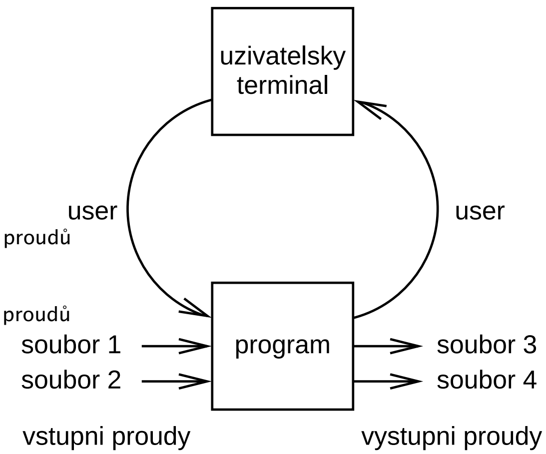
Příklad: databázové operace

- ***Caching***: odpovědi na dotazy jsou přidány do programové databáze
  - `?- solve( problem, Solution),`
    `asserta( solve( problem, Solution) ).`
  - `:- dynamic solve/2.` % nezbytné při použití v SICStus Prologu
- Příklad:

```
uloz_trojice( Seznam1, Seznam2 ) :-
        member( X1, Seznam1 ),
        member( X2, Seznam2 ),
        spocitej_treti( X1, X2, X3 ),
        assertz( trojice( X1, X2, X3 ) ),
        fail.
uloz_trojice( _, _ ) :- !.
```


Vstup a výstup

- program může číst data ze **vstupního proudu** (*input stream*)
- program může zapisovat data do **výstupního proudu** (*output stream*)
- dva **aktivní proudy**
  - aktivní vstupní proud
  - aktivní výstupní proud
- **uživatelský terminál** – `user`
  - datový vstup z terminálu chápán jako jeden ze vstupních proudů
  - datový výstup na terminál chápán jako jeden z výstupních proudů



Vstupní a výstupní proudy: vestavěné predikáty

- změna (**otevření**) aktivního vstupního/výstupního proudu: `see(S)/tell(S)`

```
cteni( Soubor ) :- see( Soubor ),
                   cteni_ze_souboru( Informace ),
                   see( user ).
```

- **uzavření** aktivního vstupního/výstupního proudu: `seen/told`
- **zjištění** aktivního vstupního/výstupního proudu: `seeing(S)/telling(S)`

```
cteni( Soubor ) :- seeing( StarySoubor ),
                   see( Soubor ),
                   cteni_ze_souboru( Informace ),
                   seen,
                   see( StarySoubor ).
```


Sekvenční přístup k textovým souborům

- **čtení** dalšího **termu**: `read(Term)`
  - při čtení jsou termy odděleny tečkou

```
| ?- read(A), read( ahoj(B) ), read( [C,D] ).
|: ahoj. ahoj( petre ). [ ahoj( 'Petre!' ), jdeme ].
A = ahoj, B = petre, C = ahoj('Petre!'), D = jdeme
```

  - po dosažení konce souboru je vrácen atom `end_of_file`
- **zápis** dalšího **termu**: `write(Term)`

```
?- write( ahoj ).    ?- write( 'Ahoj Petre!' ).
```

  nový řádek na výstup: `nl`

  N mezer na výstup: `tab(N)`
- **čtení/zápis** dalšího **znaku**: `get0(Znak), get(NeprazdnyZnak)/put(Znak)`
  - po dosažení konce souboru je vrácena `-1`

Příklad čtení ze souboru

```
process_file( Soubor ) :-
        seeing( StarySoubor ),        % zjištění aktivního proudu
        see( Soubor ),                % otevření souboru  Soubor
        repeat,
           read( Term ),              % čtení termu  Term
           process_term( Term ),      % manipulace s termem
           Term == end_of_file,       % je konec souboru?
        !,
        seen,                         % uzavření souboru
        see( StarySoubor ).           % aktivace původního proudu

repeat.                               % opakování
repeat :- repeat.
```


Čtení programu ze souboru

- **Interpretování** kódu programu
  - `?- consult(program).`
  - `?- consult('program.pl').`
  - `?- consult( [program1, 'program2.pl'] ).`
  - `?- [program].`
  - `?- [user].` **zadávání kódu ze vstupu** ukončené CTRL+D
- **Kompilace** kódu programu
  - ?- compile([program1, 'program2.pl']).
  - další varianty podobně jako u interpretování
  - typické zrychlení: 5 až 10 krát

Všechna řešení

- Backtracking vrací pouze jedno řešení po druhém
- Všechna řešení dostupná najednou: `bagof/3, setof/3, findall/3`
- `bagof( X, P, S )`: vrátí seznam S, všech objektů X takových, že P je splněno

```
vek( petr, 7 ).
vek( anna, 5 ).
vek( tomas, 5 ).

?- bagof( Dite, vek( Dite, 5 ), Seznam ).
   Seznam = [ anna, tomas ]
```

- Volné proměnné v cíli P jsou **všeobecně kvantifikovány**

```
?- bagof( Dite, vek( Dite, Vek ), Seznam ).
   Vek = 7, Seznam = [ petr ];
   Vek = 5, Seznam = [ anna, tomas ]
```


Všechna řešení II.

- Pokud neexistuje řešení `bagof(X,P,S)` neuspěje
- bagof: pokud nějaké řešení existuje několikrát, pak S obsahuje duplicity
- `bagof, setof, findall`:
  P je libovolný cíl

```
vek( petr, 7 ).
vek( anna, 5 ).
vek( tomas, 5 ).

?- bagof( Dite, ( vek( Dite, 5 ), Dite \= anna  ), Seznam ).
   Seznam = [ tomas ]
```

- `bagof, setof, findall`:
  na objekty shromažďované v X nejsou žádná omezení

```
?- bagof( Dite-Vek, vek( Dite, Vek ), Seznam ).
   Seznam = [petr-7,anna-5,tomas-5]
```

Existenční kvantifikátor „ˆ ”

- Přidání **existenčního kvantifikátoru** „ˆ ” ⇒ hodnota proměnné nemá význam

```
?- bagof( Dite, Vekˆvek( Dite, Vek ), Seznam ).
   Seznam = [petr,anna,tomas]
```

- Anonymní proměnné jsou všeobecně kvantifikovány,
  i když jejich hodnota není (jako vždy) vracena na výstup

```
?- bagof( Dite, vek( Dite, _Vek ), Seznam ).
   Seznam = [petr] ;
   Seznam = [anna,tomas]
```

- Před operátorem „ˆ ” může být i seznam

```
?- bagof( Vek ,[Jmeno,Prijmeni]ˆvek( Jmeno, Prijmeni, Vek ), Seznam ).
   Seznam = [7,5,5]
```


Všechna řešení III.

- `setof( X, P, S )`: rozdíly od bagof
  - S je uspořádaný podle @<
  - případné duplicity v S jsou eliminovány
- `findall( X, P, S )`: rozdíly od bagof
  - všechny proměnné jsou existenčně kvantifikovány
    ?- findall(Dite, vek(Dite, Vek), Seznam).
    ⇒ v S jsou shromažďovány všechny možnosti i pro různá řešení
    ⇒ `findall` uspěje přesně jednou
  - výsledný seznam může být prázdný ⇒ pokud neexistuje řešení, uspěje a vrátí S = []
  - ```
    ?- bagof( Dite, vek( Dite, Vek ), Seznam ).
       Vek = 7, Seznam = [ petr ];
       Vek = 5, Seznam = [ anna, tomas ]
    ?- findall( Dite, vek( Dite, Vek ), Seznam ).
       Seznam = [petr,anna,tomas]
    ```


Testování typu termu

`var(X)`	X je volná proměnná
`nonvar(X)`	X není proměnná
`atom(X)`	X je atom (`pavel, 'Pavel Novák', <-->`)
`integer(X)`	X je integer
`float(X)`	X je float
`atomic(X)`	X je atom nebo číslo
`compound(X)`	X je struktura

Určení počtu výskytů prvku v seznamu

```
count( X, S, N ) :- count( X, S, 0, N ).

count( _, [], N, N ).
count( X, [X|S], N0, N) :- !, N1 is N0 + 1, count( X, S, N1, N).
count( X, [_|S], N0, N) :- count( X, S, N0, N).

:-? count( a, [a,b,a,a], N )      :-? count( a, [a,b,X,Y], N).
N=3                               N=3

count( _, [], N, N ).
count( X, [Y|S], N0, N ) :- nonvar(Y), X = Y, !,
                            N1 is N0 + 1, count( X, S, N1, N ).
count( X, [_|S], N0, N ) :- count( X, S, N0, N ).
```
```

Konstrukce a dekompozice atomu

- **Atom** (opakování)
  - řetězce písmen, čísel, „_" začínající malým písmenem: `pavel, pavel_novak, x2, x4_34`
  - řetězce speciálních znaků: `+, <->, ===>`
  - řetězce **v apostrofech**: `'Pavel', Pavel Novák', 'prší', 'ano'`

  `?- 'ano'=A.        A = ano`
- **Řetězec znaků v uvozovkách**
  - př. "ano", "Pavel"

  ```
  ?- A="Pavel".                  ?- A="ano".
  A = [80,97,118,101,108]        A=[97,110,111]
  ```
  - př. použití: konstrukce a dekompozice atomu na znaky, vstup a výstup do souboru
- **Konstrukce atomu ze znaků, rozložení atomu na znaky**

```
name( Atom, SeznamASCIIKodu )          name( ano, [97,110,111] )
                                       name( ano, "ano" )
```


Konstrukce a dekompozice termu

- Konstrukce a dekompozice termu

  `Term =.. [ Funktor | SeznamArgumentu ]`

  ```
  a(9,e) =.. [a,9,e]
  Cil =.. [ Funktor | SeznamArgumentu ], call( Cil )
  atom =.. X ⇒ X = [atom]
  ```
- Pokud chci znát pouze funktor nebo některé argumenty, pak je efektivnější:

```
functor( Term, Funktor, Arita )        functor( a(9,e), a, 2 )
                                       functor(atom,atom,0)   functor(1,1,0)
arg( N, Term, Argument )               arg( 2, a(9,e), e)
```


Rekurzivní rozklad termu

- Term je proměnná (`var/1`), atom nebo číslo (`atomic/1`) ⇒ konec rozkladu
- Term je seznam (`[_|_]`) ⇒
  procházení seznamu a rozklad každého prvku seznamu
- Term je složený (`=../2, functor/3`) ⇒
  procházení seznamu argumentů a rozklad každého argumentu
- Příklad: `ground/1` uspěje, pokud v termu nejsou proměnné; jinak neuspěje

```
ground(Term) :- atomic(Term), !.
ground(Term) :- var(Term), !, fail.
ground([H|T]) :- !, ground(H), ground(T).
ground(Term) :- Term =.. [ _Funktor | Argumenty ],
                ground( Argumenty ).
```

```
?- ground(s(2,[a(1,3),b,c],X)).      ?- ground(s(2,[a(1,3),b,c])).
no                                   yes
```


Příklad: dekompozice termu I.

- `count_term( Integer, Term, N )` určí počet výskytů celého čísla v termu
  - `?- count_term( 1, a(1,2,b(x,z(a,b,1)),Y), N ).        N=2`
  - `count_term( X, T, N ) :- count_term( X, T, 0, N).`

  ```
  count_term( X, T, N0, N ) :- integer(T), X = T, !, N is N0 + 1.
  count_term( _, T, N, N ) :- atomic(T), !.
  count_term( _, T, N, N ) :- var(T), !.
  count_term( X, T, N0, N ) :- T =.. [ _ | Argumenty ],
                               count_arg( X, Argumenty, N0, N ).
  count_arg( _, [], N, N ).
  count_arg( X, [ H | T ], N0, N ) :- count_term( X, H, 0, N1),
                                      N2 is N0 + N1,
                                      count_arg( X, T, N2, N ).
  ```
  - `?- count_term( 1, [a,2,[b,c],[d,[e,f],Y]], N ).`

  `count_term( X, T, N0, N ) :- T = [_|_], !, count_arg( X, T, N0, N ).`

  klauzuli přidáme **před** poslední klauzuli count_term/4

Cvičení: dekompozice termu

- Napište predikát `substitute( Podterm, Term, Podterm1, Term1)`, který nahradí všechny výskyty `Podterm` v `Term` termem `Podterm1` a výsledek vrátí v `Term1`
- Předpokládejte, že `Term` a `Podterm` bez proměnných)
- `?- substitute( sin(x), 2*sin(x)*f(sin(x)), t, F ).`   `F=2*t*f(t)`

Technika a styl programování v Prologu

Technika a styl programování v Prologu

- Styl programování v Prologu
  - některá pravidla správného stylu
  - správný vs. špatný styl
  - komentáře
- Ladění
- Efektivita

Styl programování v Prologu I.

- Cílem stylistických konvencí je
  - redukce nebezpečí programovacích chyb
  - psaní čitelných a srozumitelných programů, které se dobře ladí a modifikují
- Některá pravidla správného stylu
  - krátké klauzule
  - krátké procedury; dlouhé procedury pouze s uniformní strukturou (tabulka)
  - klauzule se základními (hraničními) případy psát před rekurzivními klauzulemi
  - vhodná jmena procedur a proměnných
    - nepoužívat seznamy (`[...]`) nebo závorky (`{...}, (...)`) pro termy pevné arity
  - vstupní argumenty psát před výstupními
  - **struktura programu** – **jednotné konvence** v rámci celého programu, např.
    - mezery, prázdné řádky, odsazení
    - klauzule stejné procedury na jednom místě; prázdné řádky mezi klauzulemi; každý cíl na zvláštním řádku

Správný styl programování

- konstrukce setříděného seznamu `Seznam3` ze `setříděných seznamů Seznam1, Seznam2`: `merge( Seznam1, Seznam2, Seznam3 )`
- `merge( [2,4,7], [1,3,4,8], [1,2,3,4,4,7,8] )`
-

```
merge( [], Seznam, Seznam ) :-
       !.                          % prevence redundantních řešení

merge( Seznam, [], Seznam ).

merge( [X|Telo1], [Y|Telo2], [X|Telo3] ) :-
       X < Y, !,
       merge( Telo1, [Y|Telo2], Telo3 ).

merge( Seznam1, [Y|Telo2], [Y|Telo3] ) :-
       merge( Seznam1, Telo2, Telo3 ).
```


Špatný styl programování

```
merge( S1, S2, S3 ) :-
 S1 = [], !, S3 = S2;          % první seznam je prázdný
 S2 = [], !, S3 = S1;          % druhý seznam je prázdný
 S1 = [X|T1],
 S2 = [Y|T2],
 ( X < Y, !,
   Z = X,                      % Z je hlava seznamu S3
   merge( T1, S2, T3 );
   Z = Y,
   merge( S1, T2, T3) ),
 S3 = [ Z | T3 ].
```


Styl programování v Prologu II.

- **Středník** „;" může způsobit nesrozumitelnost klauzule
  - nedávat středník na konec řádku, používat závorky
  - v některých případech: rozdělení klauzle se středníkem do více klauzulí
- Opatrné používání **operátoru řezu**
  - preferovat použití zeleného řezu (nemění deklarativní sémantiku)
  - červený řez používat v jasně definovaných konstruktech

    negace: `P, !, fail; true` `\+ P`

    alternativy: `Podminka, !, Cil1 ; Cil2` `Podminka -> Cil1 ; Cil2`
- Opatrné používání **negace** „\+"
  - negace jako neúspěch: negace není ekvivalentní negaci v matematické logice
- Pozor na **`assert` a `retract`**: snižuji transparentnost chování programu

Dokumentace a komentáře

- co program dělá, jak ho používat (jaký cíl spustit a jaké jsou očekávané výsledky), příklad použití
- které predikáty jsou hlavní (*top-level*)
- jak jsou hlavní koncepty (objekty) reprezentovány
- doba výpočtu a paměťové nároky
- jaké jsou limitace programu
- zda jsou použity nějaké speciální rysy závislé na systému
- jaký je význam predikátů v programu, jaké jsou jejich argumenty, které jsou vstupní a které výstupní (pokud víme)
  - vstupní argumenty „+", výstupní „-" `merge( +Seznam1, +Seznam2, -Seznam3 )`
  - `JmenoPredikatu/Arita` `merge/3`
- algoritmické a implementační podrobnosti

Ladění

- Přepínače na trasování: `trace/0`, `notrace/0`
- Trasování specifického predikátu: `spy/1, nospy/1`
  - `spy( merge/3 )`
- debug/0, nodebug/0: pro trasování pouze predikátů zadaných `spy/1`
- Libovolná část programu může být spuštěna

  zadáním vhodného dotazu: **trasování cíle**
  - vstupní informace: jméno predikátu, hodnoty argumentů při volání
  - výstupní informace
    - při úspěchu hodnoty argumentů splňující cíl
    - při neúspěchu indikace chyby
  - nové vyvolání přes ";": stejný cíl je volán při backtrackingu

Krabičkový (4-branový) model

- Vizualizace řídícího toku (backtrackingu) na úrovni predikátu
  - `Call`: volání cíle
  - `Exit`: úspěšné ukončení volání cíle
  - `Fail`: volání cíle neuspělo
  - Redo: jeden z následujících cílů neuspěl a systém backtrackuje, aby nalezl alternativy k předchozímu řešení

```
                   *------------------------------------------*
  Call             |                                          |    Exit
-----------------> +  predek( X, Z ) :- rodic( X, Z ).        + --------->
                   |                                          |
                   |  predek( X, Z ) :- rodic( X, Y ),        |
<----------------- +                    predek( Y, Z ).       + <---------
  Fail             |                                          |     Redo
                   *------------------------------------------*
```


Příklad: trasování

```
a(X) :- nonvar(X).
a(X) :- c(X).
a(X) :- d(X).
c(1).
d(2).

         *-------------------*
 Call   |                   |  Exit
------> + a(X) :- nonvar(X).| ------>
        | a(X) :- c(X).     |
<------ + a(X) :- d(X).     + <------
  Fail  |                   |   Redo
        *-------------------*
```

```
| ?-  a(X).
        1      1 Call: a(_463) ?
        2      2 Call: nonvar(_463) ?
        2      2 Fail: nonvar(_463) ?
        3      2 Call: c(_463) ?
        3      2 Exit: c(1) ?
?       1      1 Exit: a(1) ?
X = 1 ? ;
        1      1 Redo: a(1) ?
        4      2 Call: d(_463) ?
        4      2 Exit: d(2) ?
        1      1 Exit: a(2) ?
X = 2 ? ;
no
% trace
| ?-
```


Efektivita

- Čas výpočtu, paměťové nároky, a také časové nároky na vývoj programu
  - u Prologu můžeme častěji narazit na problémy s časem výpočtu a pamětí
  - Prologovské aplikace redukují čas na vývoj
  - vhodnost pro symbolické, nenumerické výpočty se strukturovanými objekty a relacemi mezi nimi
- Pro zvýšení efektivity je nutno se zabývat **procedurálními aspekty**
  - **zlepšení efektivity při prohledávání**
    - odstranění zbytečného backtrackingu
    - zrušení provádění zbytečných alternativ co nejdříve
  - návrh **vhodnějších datových struktur**, které umožní efektivnější operace s objekty

Zlepšení efektivity: základní techniky

- **Optimalizace posledního volání (LCO) a akumulátory**
- **Rozdílové seznamy** při spojování seznamů
- **Caching**: uložení vypočítaných výsledků do programové databáze
- **Indexace** podle prvního argumentu
  - např. v SICStus Prologu
  - při volání predikátu s prvním nainstaniovaným argumentem se používá hašovací tabulka zpřístupňující pouze odpovídající klauzule
  - `zamestnanec( Prijmeni, KrestniJmeno, Oddeleni, ...)`
- **Determinismus**:
  - rozhodnout, které klauzule mají uspět vícekrát, ověřit požadovaný determinismus

Predikátová logika 1.řádu

Teorie logického programování

- PROLOG: PROgramming in LOGic, část predikátové logiky 1.řádu
  - fakta: `rodic(petr,petrik)`, $\forall X$ `a(X)`
  - klauzule: $\forall X \forall Y$ `rodic(X,Y)` $\Rightarrow$ `predek(X,Y)`
- Predikátová logika I. řádu (PL1)
  - soubory objektů: lidé, čísla, body prostoru, ...
  - syntaxe PL1, sémantika PL1, pravdivost a dokazatelnost
- Rezoluce ve výrokové logice, v PL1
  - dokazovací metoda
- Rezoluce v logickém programování
- Backtracking, řez, negace vs. rezoluce

Predikátová logika I. řádu (PL1)

Abeceda $\mathcal{A}$ **jazyka** $\mathcal{L}$ PL1 se skládá ze symbolů:

- **proměnné** X, Y, ... označují libovolný objekt z daného oboru
- **funkční symboly** f, g, ... označují operace (příklad: +, $\times$)
  - **arita** = počet argumentů, n-**árni** symbol, značíme f/n
  - nulární funkční symboly – **konstanty**: označují význačné objekty (příklad: 0, 1, ...)
- **predikátové symboly** p,q, ... pro vyjádření vlastností a vztahů mezi objekty
  - **arita** = počet argumentů, n-**árni** symbol, značíme p/n příklad: $<$, $\in$
- **logické spojky** $\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\equiv$
- **kvantifikátory** $\forall$, $\exists$
  - logika I. řádu používá **kvantifikaci pouze pro individua** (odlišnost od logik vyššího řádu)
  - v logice 1. řádu nelze: v $\mathbb{R}$: $\forall A \subseteq \mathbb{R}, \forall f : \mathbb{R} \to \mathbb{R}$
- **závorky**:),(

Jazyky PL1

- Specifikace jazyka $\mathcal{L}$ je definována funkčními a predikátovými symboly
  symboly tedy určují oblast, kterou jazyk popisuje
- **Jazyky s rovností**: obsahují predikátový symbol pro rovnost „="

Příklady

- jazyk teorie uspořádání
  - jazyk s =, binární prediátový symbol $<$
- jazyk teorie množin
  - jazyk s =, binární predikátový symbol $\in$
- jazyk elementární aritmetiky
  - jazyk s =, nulární funkční symbol 0 pro nulu,
    unární funkční symbol s pro operaci následníka,
    binární funkční symboly pro sčítání + a násobení $\times$

Term, atomická formule, formule

- **Term** nad abecedou $\mathcal{A}$
  - každá proměnná z $\mathcal{A}$ je term
  - je-li f/n z $\mathcal{A}$ a $t_1, \ldots, t_n$ jsou termy, pak $f(t_1, \ldots, t_n)$ je také term
  - každý term vznikne konečným počtem užití přechozích kroků f(X, g(X,0))
- **Atomická formule (atom)** nad abecedou $\mathcal{A}$
  - je-li p/n z $\mathcal{A}$ a $t_1, \ldots, t_n$ jsou termy, pak $p(t_1, \ldots, t_n)$ je atomická formule f(X) < g(X,0)
- **Formule** nad abecedou $\mathcal{A}$
  - každá atomická formule je formule
  - jsou-li F a G formule, pak také $(\neg F), (F \wedge G), (F \vee G), (F \Rightarrow G), (F \equiv G)$ jsou formule
  - je-li X proměnná a F formule, pak také $(\forall \mathrm{X}\ F)$ a $(\exists \mathrm{X}\ F)$ jsou formule
  - každá formule vznikne konečným počtem užití přechozích kroků (∃X ((f(X) = 0) ∧ p(0)))

Interpretace

- **Interpretace** $\mathcal{I}$ jazyka $\mathcal{L}$ nad abecedou $\mathcal{A}$ je dána
  - neprázdnou množinou $\mathcal{D}$ (také značíme $|\mathcal{I}|$, nazývá se **univerzum**) a
  - zobrazením, které
    - každé konstantě $c \in \mathcal{A}$ přiřadí nějaký **prvek** $\mathcal{D}$
    - každému funkčnímu symbolu $f/n \in \mathcal{A}$ přiřadí n-ární **operaci** nad $\mathcal{D}$
    - každému predikátovému symbolu $p/n \in \mathcal{A}$ přiřadí n-ární **relaci** nad $\mathcal{D}$
- Příklad: uspořádání na $\mathbb{R}$
  - jazyk: predikátový symbol $mensi/2$
  - interpretace: univerzum $\mathbb{R}$; zobrazení: $mensi(x,y) := x < y$
- Příklad: elementární aritmetika nad množinou $\mathbb{N}$ (včetně 0)
  - jazyk: konstanta $zero$, funční symboly $s/1$, $plus/2$
  - interpretace:
    - univerzum $\mathbb{N}$; zobrazení: $zero := 0$, $s(x) := 1 + x$, $plus(x,y) := x + y$

Sémantika formulí

- **Ohodnocení proměnné** $\varphi(X)$: každé proměnné X je přiřazen prvek $|\mathcal{I}|$
- **Hodnota termu** $\varphi(t)$: každému termu je přiřazen prvek univerza
  - příklad: nechť $\varphi(X) := 0$
    $\varphi(plus(s(zero),X)) = \varphi(s(zero)) + \varphi(X) = (1 + \varphi(zero)) + 0 = (1+0) + 0 = 1$
- Každá **dobře utvořená formule** označuje **pravdivostní hodnotu** (PRAVDA, NEPRAVDA) v závislosti na své struktuře a interpretaci

  Pravdivá formule $\mathcal{I} \models_\varphi Q$: formule Q označena PRAVDA
  Neravdivá formule $\mathcal{I} \not\models_\varphi Q$: formule Q označena NEPRAVDA
  - příklad: $p/1$ predikátový symbol, tj. $p \subseteq |\mathcal{I}|$ $p := \{\langle 1\rangle, \langle 3\rangle, \langle 5\rangle, \ldots\}$
    $\mathcal{I} \models p(zero) \wedge p(s(zero))$ iff $\mathcal{I} \models p(zero)$ a $\mathcal{I} \models p(s(zero))$
    iff $\langle\varphi(zero)\rangle \in p$ a $\langle\varphi(s(zero))\rangle \in p$
    iff $\langle\varphi(zero)\rangle \in p$ a $\langle(1 + \varphi(zero)\rangle \in p$
    iff $\langle 0\rangle \in p$ a $\langle 1\rangle \in p$
    $\langle 1\rangle \in p$ ale $\langle 0\rangle \notin p$, tedy formule je nepravdivá v $\mathcal{I}$

Model

- Interpretace se nazývá **modelem** formule, je-li v ní tato formule pravdivá
  - interpretace množiny $\mathbb{N}$ s obvyklými operacemi je modelem formule (1 + s(0) = s(s(0)))
  - interpretace, která se liší přiřazením s/1: s(x):=x není modelem této formule
- **Teorie** $\mathcal{T}$ jazyka $\mathcal{L}$ je množina formulí jazyka $\mathcal{L}$, tzv. **axiomů**
  - ¬ s(X) = 0 je jeden z axiomů teorie elementární aritmetiky
- **Model teorie**: libovolná interpretace, která je modelem všech jejích axiomů
  - všechny axiomy teorie musí být v této interpretaci pravdivé
- **Pravdivá formule v teorii** $\mathcal{T} \models F$: pravdivá v každém z modelů teorie $\mathcal{T}$
  - říkáme také formule **platí v teorii** nebo je **splněna v teorii**
  - formule 1 + s(0) = s(s(0)) je pravdivá v teorii elementárních čísel
- **Logicky pravdivá formule** $\models F$**:** libovolná interpretace je jejím modelem
  - nebo-li F je pravdivá v každém modelu libovolné teorie
  - formule G ∨ ¬ G je logicky pravdivá, formule 1 + s(0) = s(s(0)) není logicky pravdivá

Zkoumání pravdivosti formulí

- Zjištění pravdivosti provádíme důkazem

  Důkaz: libovolná posloupnost $F_1, \ldots, F_n$ formulí jazyka $\mathcal{L}$, v níž každé F_i je buď' axiom teorie jazyka $\mathcal{L}$ nebo lze F_i odvodit z předchozích F_j $(j < i)$ použitím určitých **odvozovacích pravidel**
- Odvozovací pravidla – příklady
  - **pravidlo modus ponens**: z formulí F a $F \Rightarrow G$ lze odvodit G
  - **rezoluční princip**: z formulí $F \vee A$, $G \vee \neg A$ odvodit $F \vee G$
- F je **dokazatelná z formulí** $A_1, \cdots, A_n$ $\qquad A_1, \cdots, A_n \vdash F$

  existuje-li důkaz F z $A_1, \cdots, A_n$
- Dokazatelné formule v teorii $\mathcal{T}$ nazýváme **teorémy** teorie $\mathcal{T}$

Korektnost a úplnost

- **Uzavřená formule**: neobsahuje volnou proměnnou (bez kvantifikace)
  - ∀Y ((0 < Y) ∧ (∃X (X < Y))) je uzavřená formule
  - (∃X (X < Y)) není uzavřená formule
- Množina odvozovacích pravidel se nazývá **korektní**, jestliže pro každou množinu uzavřených formulí $\mathcal{P}$ a každou uzavřenou formuli F platí:

  jestliže $\mathcal{P} \vdash F$ pak $\mathcal{P} \models F$ (jestliže je něco dokazatelné, pak to platí)

  Odvozovací pravidla jsou **úplná**, jestliže

  jestliže $\mathcal{P} \models F$ pak $\mathcal{P} \vdash F$ (jestliže něco platí, pak je to dokazatelné)
- PL1: úplná a korektní dokazatelnost, tj.
  pro teorii $\mathcal{T}$ s množinou axiomů $\mathcal{A}$ platí: $\mathcal{T} \models F$ **právě když** $\mathcal{A} \vdash F$

Rezoluce v predikátové logice I. řádu

Rezoluce

- rezoluční princip: z $F \vee A$, $G \vee \neg A$ odvodit $F \vee G$
- dokazovací metoda používaná
  - v Prologu
  - ve většině systémů pro automatické dokazování
- procedura pro **vyvrácení**
  - hledáme důkaz pro negaci formule
  - snažíme se dokázat, že negace formule je nesplnitelná
    $\Longrightarrow$ formule je vždy pravdivá

Formule

- **literál** l
  - **pozitivní literál** = atomická formule $p(t_1, \cdots, t_n)$
  - **negativní literál** = negace atomické formule $\neg p(t_1, \cdots, t_n)$
- **klauzule** C = konečná množina literálů reprezentující jejich disjunkci
  - příklad: $p(X) \vee q(a, f) \vee \neg p(Y)$ notace: $\{p(X), q(a, f), \neg p(Y)\}$
  - **klauzule je pravdivá** $\Longleftrightarrow$ je pravdivý alespoň jeden z jejích literálů
  - **prázdná klauzule** se značí $\square$ a je vždy nepravdivá (neexistuje v ní pravdivý literál)
- **formule** F = množina klauzulí reprezentující jejich konjunkci
  - formule je v tzv. konjuktivní normální formě (konjunkce disjunkcí)
  - příklad: $(p \vee q) \wedge (\neg p) \wedge (p \vee \neg q \vee r)$ notace: $\{\{p, q\}, \{\neg p\}, \{p, \neg q, r\}\}$
  - **formule je pravdivá** $\Longleftrightarrow$ všechny klauzule jsou pravdivé
  - prázdná formule je vždy pravdivá (neexistuje klauzule, která by byla nepravdivá)
- **množinová notace:** literál je prvek klauzule, klauzule je prvek formule, ...

Splnitelnost

- **[Opakování:]** Interpretace $\mathcal{I}$ jazyka $\mathcal{L}$ je dána univerzem $\mathcal{D}$ a zobrazením, které přiřadí konstantě c prvek $\mathcal{D}$, funkčnímu symbolu f/n n-ární operaci v $\mathcal{D}$ a predikátovému symbolu p/n n-ární relaci.
  - příklad: $F = \{\{f(a, b) = f(b, a)\}, \{f(f(a, a), b) = a\}\}$
    interpretace $\mathcal{I}_1$: $\mathcal{D} = \mathbb{Z}, a := 1, b := -1, f := "+"$
- Formule je **splnitelná**, existuje-li interpretace, pro kterou je pravdivá
  - formule je konjunkce klauzulí, tj. všechny klauzule musí být v dané interpretaci pravdivé
  - příklad (pokrač.): F je splnitelná (je pravdivá v $\mathcal{I}_1$)
- Formule je **nesplnitelná**, neexistuje-li interpretace, pro kterou je pravdivá
  - tj. formule je ve všech iterpretacích nepravdivá
  - tj. neexistuje interpretace, ve které by byly všechny klauzule pravdivé
  - příklad: $G = \{\{p(b)\}, \{p(a)\}, \{\neg p(a)\}\}$ je nesplnitelná
    ($\{p(a)\}$ a $\{\neg p(a)\}$ nemohou být zároveň pravdivé)

Rezoluční princip ve výrokové logice

- **Rezoluční princip** = pravidlo, které umožňuje odvodit z klauzulí $C_1 \cup \{l\}$ a $\{\neg l\} \cup C_2$ klauzuli $C_1 \cup C_2$

$$\frac{C_1 \cup \{l\} \qquad \{\neg l\} \cup C_2}{C_1 \cup C_2}$$

- $C_1 \cup C_2$ se nazývá **rezolventou** původních klauzulí
- příklad:

$$\frac{\{p, r\} \qquad \{\neg r, s\}}{\{p, s\}} \qquad \frac{(p \vee r) \wedge (\neg r \vee s)}{p \vee s}$$

  obě klauzule $(p \vee r)$ a $(\neg r \vee s)$ musí být pravdivé
  protože r nestačí k pravdivosti obou klauzulí,
  musí být pravdivé p (pokud je pravdivé $\neg r$) nebo s (pokud je pravdivé r),
  tedy platí klauzule $p \vee s$

Rezoluční důkaz

- **rezoluční důkaz klauzule C z formule F** je konečná posloupnost $C_1, \ldots, C_n = C$ klauzulí taková, že C_i je buď klauzule z F nebo rezolventa C_j, C_k pro $k, j < i$.
- příklad: rezoluční důkaz $\{p\}$ z formule $F = \{\{p, r\}, \{q, \neg r\}, \{\neg q\}\}$

  $C_1 = \{p, r\}$ klauzule z F

  $C_2 = \{q, \neg r\}$ klauzule z F

  $C_3 = \{p, q\}$ rezolventa C_1 a C_2

  $C_4 = \{\neg q\}$ klauzule z F

  $C_5 = \{p\} = C$ rezolventa C_3 a C_4

Rezoluční vyvrácení

- rezoluční důkaz formule F spočívá v **demonstraci nesplnitelnosti** $\neg F$
  - $\neg F$ nesplnitelná $\Rightarrow$ $\neg F$ je nepravdivá ve všech interpretacích $\Rightarrow$ F je vždy pravdivá
- začneme-li z klauzulí reprezentujících $\neg F$, musíme postupným uplatňováním rezolučního principu **dospět k prázdné klauzuli** $\Box$
- Příklad:

  $F \ldots \neg a \vee a$

  $\neg F \ldots a \wedge \neg a$

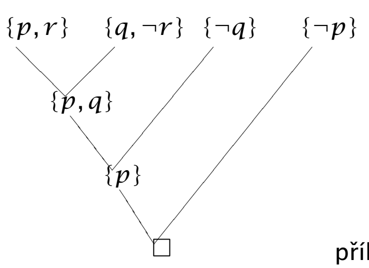
  $\neg F \ldots \{\{a\}, \{\neg a\}\}$

  $C_1 = \{a\}, C_2 = \{\neg a\}$

  rezolventa C_1 a C_2 je $\Box$, tj. F je vždy pravdivá
- rezoluční důkaz $\Box$ z formule F se nazývá **rezoluční vyvrácení formule** F

Strom rezolučního důkazu

- **strom rezolučního důkazu** klauzule C z formule F je binární strom:
  - kořen je označen klauzulí C,
  - listy jsou označeny klauzulemi z F a
  - každý uzel, který není listem,
    - má bezprostředními potomky označené klauzulemi C_1 a C_2
    - je označen rezolventou klauzulí C_1 a C_2
- příklad: $F = \{\{p, r\}, \{q, \neg r\}, \{\neg q\}, \{\neg p\}\}$ $C = \Box$



strom rezolučního vyvrácení

(rezoluční důkaz $\Box$ z F)

příklad: $\{\{p, r\}, \{q, \neg r\}, \{\neg q\}, \{\neg p, t\}, \{\neg s\}, \{s, \neg t\}\}$

Substituce

- **co s proměnnými? vhodná substituce a unifikace**
  - $f(X, a, g(Y)) < 1, f(h(c), a, Z) < 1, \quad X = h(c), Z = g(Y) \Longrightarrow f(h(c), a, g(Y)) < 1$
- **substituce** je libovolná funkce θ zobrazující výrazy do výrazů tak, že platí
  - $\theta(E) = E$ pro libovolnou konstantu E
  - $\theta(f(E_1, \cdots, E_n)) = f(\theta(E_1), \cdots, \theta(E_n))$ pro libovolný funkční symbol f
  - $\theta(p(E_1, \cdots, E_n)) = p(\theta(E_1), \cdots, \theta(E_n))$ pro libovolný predik. symbol p
- **substituce** je tedy homomorfismus výrazů, který **zachová vše kromě proměnných** – ty lze nahradit čímkoliv
- substituce zapisujeme zpravidla ve tvaru seznamu $[X_1/\xi_1, \cdots, X_n/\xi_n]$ kde X_i jsou proměnné a ξ_i substituované termy
  - příklad: $p(X)[X/f(a)] \equiv p(f(a))$
- **přejmenování proměnných**: speciální náhrada proměnných proměnnými
  - příklad: $p(X)[X/Y] \equiv p(Y)$

Unifikace

- Ztotožnění dvou literálů p, q pomocí vhodné substituce σ takové, že $p\sigma = q\sigma$ nazýváme **unifikací** a příslušnou substituci **unifikátorem**.
- **Unifikátorem** množiny S literálů nazýváme substituce θ takovou, že množina

$$S\theta = \{t\theta | t \in S\}$$

má jediný prvek.

  - příklad: S = { datum(D1, M1, 2003), datum(1, M2, Y2) }
    unifikátor θ = [D1/1, M1/2, M2/2, Y2/2003] $S\theta$ = { datum(1, 2, 2003) }
- Unifikátor σ množiny S nazýváme **nejobecnějším unifikátorem** (**mgu** – ***most general unifier***), jestliže pro libovolný unifikátor θ existuje substituce λ taková, že $\theta = \sigma\lambda$.
  - příklad (pokrač.): nejobecnější unifikátor σ = [D1/1, Y2/2003, M1/M2], λ=[M2/2]

Rezoluční princip v PL1

- základ:
  - rezoluční princip ve výrokové logice $\dfrac{C_1 \cup \{l\} \qquad \{\neg l\} \cup C_2}{C_1 \cup C_2}$
  - substituce, unifikátor, nejobecnější unifikátor
- **rezoluční princip v PL1** je pravidlo, které
  - připraví příležitost pro uplatnění vlastního rezolučního pravidla nalezením vhodného unifikátoru
  - provede rezoluci a získá rezolventu

$$\frac{C_1 \cup \{A\} \qquad \{\neg B\} \cup C_2}{C_1\rho\sigma \cup C_2\sigma}$$

  - kde ρ **je přejmenováním proměnných** takové, že klauzule $(C_1 \cup A)\rho$ a $\{B\} \cup C_2$ nemají společné proměnné
  - σ **je nejobecnější unifikátor** klauzulí $A\rho$ a B

Příklad: rezoluce v PL1

- příklad: $C_1 = \{p(X,Y), \quad q(Y)\} \qquad C_2 = \{\neg q(a), \quad s(X,W)\}$
- přejmenování proměnných: $\rho = [X/Z]$
  $C_1 = \{p(Z,Y), \quad q(Y)\} \qquad C_2 = \{\neg q(a), \quad s(X,W)\}$
- nejobecnější unifikátor: $\sigma = [Y/a]$
  $C_1 = \{p(Z,a), \quad q(a)\} \qquad C_2 = \{\neg q(a), \quad s(X,W)\}$
- rezoluční princip: $C = \{p(Z,a), \quad s(X,W)\}$
- vyzkoušejte si:
  $C_1 = \{q(X), \quad \neg r(Y), \quad p(X,Y), \quad p(f(Z),f(Z))\}$
  $C_2 = \{n(Y), \quad \neg r(W), \quad \neg p(f(a),f(a)), \quad \neg p(f(W),f(W))\}$

Rezoluce v PL1

- **Obecný rezoluční princip v PL1**

$$\frac{C_1 \cup \{A_1, \cdots, A_m\} \qquad \{\neg B_1, \cdots, \neg B_n\} \cup C_2}{C_1\rho\sigma \cup C_2\sigma}$$

  - kde ρ je přejmenováním proměnných takové, že množiny klauzulí $\{A_1\rho, \cdots, A_m\rho\}$ a $\{B_1, \cdots, B_n\}$ nemají společné proměnné
  - σ je nejobecnější unifikátor množiny $\{A_1\rho, \cdots, A_m\rho, B_1, \cdots, B_n\}$
  - příklad: $A_1 = a(X)$ vs. $\{\neg B_1, \neg B_2\} = \{\neg a(b), \neg a(Z)\}$
    v jednom kroku potřebuji vyrezolvovat zároveň B_1 i B_2
- Rezoluce v PL1
  - **korektní**: jestliže existuje rezoluční vyvrácení F, pak F je nesplnitelná
  - **úplná**: jestliže F je nesplnitelná, pak existuje rezoluční vyvrácení F

Zefektivnění rezoluce

- rezoluce je intuitivně efektivnější než axiomatické systémy
  - axiomatické systémy: který z axiomů a pravidel použít?
  - rezoluce: pouze jedno pravidlo
- stále ale příliš mnoho možností, jak hledat důkaz v prohledávacím prostoru
- problém SAT= $\{S|S$ je splnitelná $\}$ NP úplný,
  nicméně: menší prohledávací prostor vede k rychlejšímu nalezení řešení
- strategie pro zefektivnění prohledávání ⇒ varianty rezoluční metody
- vylepšení prohledávání
  - zastavit prohledávání cest, které nejsou slibné
  - specifikace pořadí, jak procházíme alternativními cestami

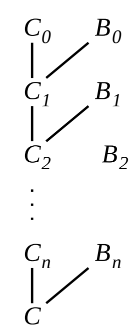
Varianty rezoluční metody

- **Věta:** Každé omezení rezoluce je korektní.
  - stále víme, že to, co jsme dokázali, platí
- T-**rezoluce**: klauzule účastnící se rezoluce nejsou tautologie — úplná
  - tautologie nepomůže ukázat, že formule je nesplnitelná
- **sémantická rezoluce**: — úplná
  zvolíme libovolnou interpretaci a pro rezoluci používáme jen takové klauzule, z nichž alespoň jedna je v této interpretaci nepravdivá
  - pokud jsou obě klauzule pravdivé, těžko odvodíme nesplnitelnost formule
- **vstupní (*input*) rezoluce:** — neúplná
  alespoň jedna z klauzulí, použitá při rezoluci, je z výchozí **vstupní množiny** S
  - $\{\{p,q\}, \quad \{\neg p,q\}, \quad \{p,\neg q\}, \quad \{\neg p,\neg q\}\}$
    existuje rezoluční vyvrácení
    neexistuje rezoluční vyvrácení pomocí vstupní rezoluce

Rezoluce a logické programování

Lineární rezoluce

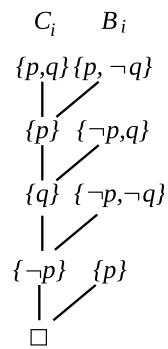
- varianta rezoluční metody
  - snaha o generování lineární posloupnosti místo stromu
  - v každém kroku kromě prvního můžeme použít bezprostředně předcházející rezolventu a k tomu buď některou z klauzulí vstupní množiny S nebo některou z předcházejících rezolvent
- **lineární rezoluční důkaz** C **z** S je posloupnost dvojic $\langle C_0,B_0\rangle,\ldots\langle C_n,B_n\rangle$ taková, že $C = C_{n+1}$ a
  - C_0 a každá B_i jsou prvky S nebo některé $C_j, j < i$
  - každá $C_{i+1}, i \leq n$ je rezolventa C_i a B_i
- **lineární vyvrácení** S = lineární rezoluční důkaz $\square$ z S



Lineární rezoluce II.

- příklad: $S = \{A_1, A_2, A_3, A_4\}$

  $A_1 = \{p, q\}$
  $A_2 = \{p, \neg q\}$
  $A_3 = \{\neg p, q\}$
  $A_4 = \{\neg p, \neg q\}$



- S: **vstupní množina** klauzulí
- C_i: **střední klauzule**
- B_i: **boční klauzule**

Prologovská notace

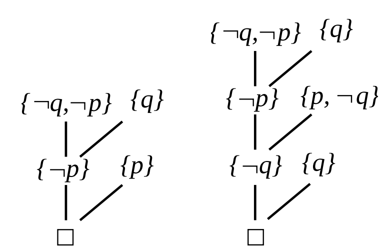
- Klauzule v matematické logice
  - $\{H_1, \cdots, H_m, \neg T_1, \cdots, \neg T_n\}$ $\quad H_1 \vee \cdots \vee H_m \vee \neg T_1 \vee \cdots \vee \neg T_n$
- **Hornova klauzule**: nejvýše jeden pozitivní literál
  - $\{H, \neg T_1, \ldots, \neg T_n\}$ $\quad \{H\}$ $\quad \{\neg T_1, \ldots, \neg T_n\}$
  - $H \vee \neg T_1 \vee \cdots \vee \neg T_n$ $\quad H$ $\quad \neg T_1 \vee \cdots \vee \neg T_n$
- **Pravidlo**: jeden pozitivní a alespoň jeden negativní literál
  - Prolog: $H : - \ T_1, \cdots, T_n.$ Matematická logika: $H \Leftarrow T_1 \wedge \cdots \wedge T_n$
  - $H \Leftarrow T$ $\quad H \vee \neg T$ $\quad H \vee \neg T_1 \vee \cdots \vee \neg T_n$ Klauzule: $\{H, \neg T_1, \ldots, \neg T_n\}$
- **Fakt**: pouze jeden pozitivní literál
  - Prolog: $H.$ Matematická logika: H Klauzule: $\{H\}$
- **Cílová klauzule**: žádný pozitivní literál
  - Prolog: $: - \ T_1, \ldots T_n.$ Matematická logika: $\neg T_1 \vee \cdots \vee \neg T_n$ Klauzule: $\{\neg T_1, \cdots \neg T_n\}$

Logický program

- **Programová klauzule**: právě jeden pozitivní literál (fakt nebo pravidlo)
- **Logický program**: konečná množina programových klauzulí
- Příklad:
  - logický program jako množina klauzulí:
    $P = \{P_1, P_2, P_3\}$
    $P_1 = \{p\}, \quad P_2 = \{p, \neg q\}, \quad P_3 = \{q\}$
  - logický program v prologovské notaci:
    $p.$
    $p : -q.$
    $q.$
  - cílová klauzule: $G = \{\neg q, \neg p\}$ $\quad : -q, p.$

Lineární rezoluce pro Hornovy klauzule

- Začneme s cílovou klauzulí: $C_0 = G$
- Boční klauzule vybíráme z programových klauzulí P
- $G = \{\neg q, \neg p\}$ $\quad P = \{P_1, P_2, P_3\}: \quad P_1 = \{p\}, \quad P_2 = \{p, \neg q\}, \quad P_3 = \{q\}$
- $: -q, p.$ $\quad p.$ $\quad p : -q,$ $\quad q.$



- **Střední klauzule jsou cílové klauzule**

Lineární vstupní rezoluce

- **Vstupní rezoluce na** $P \cup \{G\}$
  - (opakování:) alespoň jedna z klauzulí použitá při rezoluci je z výchozí vstupní množiny
  - začneme s cílovou klauzulí: $C_0 = G$
  - boční klauzule jsou vždy z P (tj. jsou to programové klauzule)
- (Opakování:) **Lineární rezoluční důkaz** C **z** S je posloupnost dvojic $\langle C_0, B_0\rangle, \dots \langle C_n, B_n\rangle$ taková, že $C = C_{n+1}$ a
  - C_0 a každá B_i jsou prvky S **nebo některé** $C_j, j < i$
  - každá $C_{i+1}, i \leq n$ je rezolventa C_i a B_i
- **Lineární vstupní (*Linear Input*) rezoluce (LI-rezoluce)** C z $P \cup \{G\}$ posloupnost dvojic $\langle C_0, B_0\rangle, \dots \langle C_n, B_n\rangle$ taková, že $C = C_{n+1}$ a
  - $C_0 = G$ **a každá** B_i **jsou prvky** P lineární rezoluce + vstupní rezoluce
  - každá $C_{i+1}, i \leq n$ je rezolventa C_i a B_i

Cíle a fakta při lineární rezoluci

- **Věta:** Je-li S nesplnitelná množina Hornových klauzulí, pak S obsahuje alespoň **jeden cíl a jeden fakt**.
  - pokud nepoužiji cíl, mám pouze fakta (1 pozit.literál) a pravidla (1 pozit.literál a alespoň jeden negat. literál), při rezoluci mi stále zůstává alespoň jeden pozit. literál
  - pokud nepoužiji fakt, mám pouze cíle (negat.literály) a pravidla (1 pozit.literál a alespoň jeden negat. literál), v rezolventě mi stále zůstávají negativní literály
- **Věta:** Existuje-li rezoluční důkaz prázdné množiny z množiny S Hornových klauzulí, pak tento rezoluční strom má v listech **jedinou cílovou klauzuli**.
  - pokud začnu důkaz pravidlem a faktem, pak dostanu zase pravidlo
  - pokud začnu důkaz dvěma pravidly, pak dostanu zase pravidlo
  - na dvou faktech rezolvovat nelze

  $\Rightarrow$ dokud nepoužiji cíl pracuji stále s množinou faktů a pravidel
  - pokud použiji v důkazu cílovou klauzuli,
    fakta mi ubírají negat.literály, pravidla mi je přidávají,
    v rezolventě mám stále samé negativní literály, tj. nelze rezolvovat s dalším cílem

Korektnost a úplnost

- **Věta:** Množina S Hornových klauzulí je nesplnitelná, právě když existuje rezoluční vyvrácení S pomocí **vstupní rezoluce**.
- **Korektnost** platí stejně jako pro ostatní omezení rezoluce
- **Úplnost LI-rezoluce pro Hornovy klauzule**:
  Nechť P je množina programových klauzulí a G cílová klauzule.
  Je-li množina $P \cup \{G\}$ Hornových klauzulí nesplnitelná,
  pak existuje rezoluční vyvrácení $P \cup \{G\}$ pomocí LI-rezoluce.
  - vstupní rezoluce pro (obecnou) formuli sama o sobě není úplná

  $\Longrightarrow$ LI-rezoluce aplikovaná na (obecnou) formuli nezaručuje,
  že nalezeneme důkaz, i když formule platí!
- **Význam LI-rezoluce pro Hornovy klauzule**:
  - $P = \{P_1, \dots, P_n\}$, $G = \{G_1, \dots, G_m\}$
  - LI-rezolucí ukážeme nesplnitelnost $P_1 \wedge \dots \wedge P_n \wedge (\neg G_1 \vee \dots \vee \neg G_m)$
  - pokud tedy předpokládáme, že program $\{P_1, \dots, P_n\}$ platí,
    tak musí být nepravdivá $(\neg G_1 \vee \dots \vee \neg G_m)$, tj. musí platit $G_1 \wedge \dots \wedge G_m$

Uspořádané klauzule (*definite clauses*)

- Klauzule = množina literálů
- **Uspořádáná klauzule (*definite clause*) = posloupnost literálů**
  - nelze volně měnit pořadí literálů
- **Rezoluční princip pro uspořádané klauzule**:

$$\frac{\{\neg A_0, \dots, \neg A_n\} \qquad \{B, \neg B_0, \dots, \neg B_m\}}{\{\neg A_0, \dots, \neg A_{i-1}, \neg B_0\rho, \dots, \neg B_m\rho, \neg A_{i+1}, \dots, \neg A_n\}\sigma}$$

  - **uspořádaná rezolventa**: $\{\neg A_0, \dots, \neg A_{i-1}, \neg B_0\rho, \dots, \neg B_m\rho, \neg A_{i+1}, \dots, \neg A_n\}\sigma$
  - ρ je přejmenování proměnných takové, že klauzule $\{A_0, \dots, A_n\}$ a $\{B, B_0, \dots, B_m\}\rho$ nemají společné proměnné
  - σ je nejobecnější unifikátor pro A_i a $B\rho$
  - **rezoluce je realizována na literálech** $\neg A_i\sigma$ **a** $B\rho\sigma$
  - je dodržováno pořadí literálů, tj.
    $\{\neg B_0\rho, \dots, \neg B_m\rho\}\sigma$ **jde do uspořádané rezolventy přesně na pozici** $\neg A_i\sigma$

Uspořádané klauzule II.

- Uspořádáné klauzule

$$\frac{\{\neg A_0, \dots, \neg A_n\} \qquad \{B, \neg B_0, \dots, \neg B_m\}}{\{\neg A_0, \dots, \neg A_{i-1}, \neg B_0\rho, \dots, \neg B_m\rho, \neg A_{i+1}, \dots, \neg A_n\}\sigma}$$

Hornovy klauzule

$$\frac{: -A_0, \dots, A_n. \qquad B : -B_0, \dots, B_m.}{: -(A_0, \dots, A_{i-1}, B_0\rho, \dots, B_m\rho, A_{i+1}, \dots, A_n)\sigma.}$$

- Příklad:

$$\frac{\{\neg s(X), \neg t(1), \neg u(X)\} \qquad \{t(Z), \neg q(Z,X), \neg r(3)\}}{\{\neg s(X), \neg q(1,A), \neg r(3), \neg u(X)\}}$$

$$\frac{: -s(X), t(1), u(X). \qquad t(Z) : -q(Z,X), r(3).}{: -s(X), q(1,A), r(3), u(X).}$$

$$\rho = [X/A] \qquad \sigma = [Z/1]$$

LD-rezoluce

- **LD-rezoluční vyvrácení** množiny uspořádaných klauzulí $P \cup \{G\}$ je posloupnost $\langle G_0, C_0\rangle, \dots, \langle G_n, C_n\rangle$ taková, že
  - G_i, C_i jsou uspořádané klauzule
  - $G = G_0$
  - $G_{n+1} = \square$
  - G_i je uspořádaná cílová klauzule
  - C_i je přejmenování klauzule z P
    - C_i neobsahuje proměnné, které jsou v $G_j, j \leq i$ nebo v $C_k, k \leq i$
  - $G_{i+1}, 0 \leq i \leq n$ je uspořádaná rezolventa G_i a C_i
- LD-rezoluce: korektní a úplná

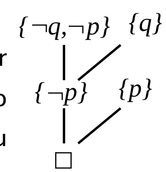
SLD-rezoluce

- **Lineární rezoluce se selekčním pravidlem = SLD-rezoluce** (***Selected Linear resolution for Definite clauses***)
  - rezoluce
  - **Selekční** pravidlo
  - **Lineární** rezoluce
  - **Definite** (uspořádané) klauzule
  - vstupní rezoluce
- **Selekční pravidlo** R je funkce, která každé neprázdné klauzuli C přiřazuje nějaký z jejích literálů $R(C) \in C$
  - při rezoluci vybírám s klauzule literál určený selekčním pravidlem
- Pokud se R neuvádí, pak se předpokládá výběr **nejlevějšího literálu**
  - nejlevější literál vybírá i Prolog

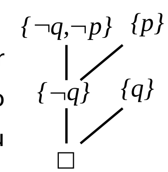
Lineární rezoluce se selekčním pravidlem

- $P = \{\{p\}, \{p, \neg q\}, \{q\}\}, \quad G = \{\neg q, \neg p\}$

výběr nejlevějšího literálu



výběr nejpravějšího literálu

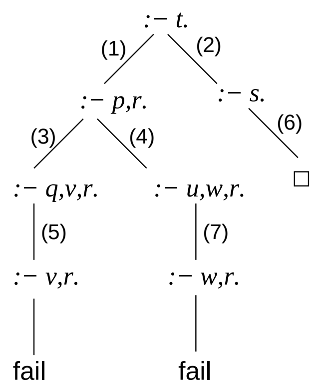


- **SLD-rezoluční vyvrácení** $P \cup \{G\}$ pomocí selekčního pravidla R je LD-rezoluční vyvrácení $\langle G_0, C_0\rangle, \dots, \langle G_n, C_n\rangle$ takové, že $G = G_0, G_{n+1} = \square$ a $R(G_i)$ je literál rezolvovaný v kroku i
- SLD-rezoluce – korektní, úplná
- Efektivita SLD-rezoluce je závislá na
  - selekčním pravidle R
  - způsobu výběru příslušné programové klauzule pro tvorbu rezolventy
    - v Prologu se vybírá vždy klauzule, která je v programu první

Příklad: SLD-strom

$t :- p, r.$ (1)
$t :- s.$ (2)
$p :- q, v.$ (3)
$p :- u, w.$ (4)
$q.$ (5)
$s.$ (6)
$u.$ (7)

$:- t.$



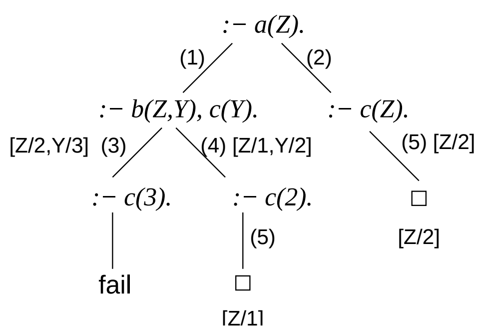
Strom výpočtu (SLD-strom)

- **SLD-strom** je strom tvořený všemi možnými výpočetními posloupnostmi logického programu P vzhledem k cíli G
- kořeny stromy jsou programové klauzule a cílová klauzule G
- v uzlech jsou rezolventy
- výchozím kořenem rezoluce je cílová klauzule G
- listy jsou dvojího druhu:
  - označené prázdnou klauzulí – jedná se o **úspěšné uzly** (*succes nodes*)
  - označené neprázdnou klauzulí – jedná se o **neúspěšné uzly** (*failure nodes*)
- úplnost SLD-rezoluce zaručuje **existenci** cesty od kořene k úspěšnému uzlu pro každý možný výsledek příslušející cíli G

Příklad: SLD–strom a výsledná substituce

$:- a(Z).$

$a(X) :- b(X, Y), c(Y).$ (1)
$a(X) :- c(X).$ (2)
$b(2, 3).$ (3)
$b(1, 2).$ (4)
$c(2).$ (5)



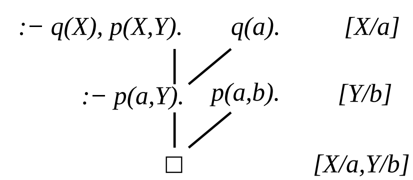
Cvičení:

$p(B) :- q(A, B), r(B).$ ve výsledné substituci jsou pouze proměnné z dotazu
$p(A) :- q(A, A).$ výsledné substituce jsou $[Z/1]$ a $[Z/2]$
$q(a, a).$ nezajímá mě substituce $[Y/2]$
$q(a, b).$
$r(b).$

Výsledná substituce (*answer substitution*)

q(a).
p(a,b).

:−q(X), p(X,Y).
X=a, Y=b



- Každý krok SLD-rezoluce vytváří novou unifikační substituci θ_i
  ⇒ potenciální instanciace proměnné ve vstupní cílové klauzuli
- **Výsledná substituce** (*answer substitution*)

$$\theta = \theta_0\theta_1 \cdots \theta_n$$ složení unifikací

Význam SLD-rezolučního vyvrácení $P \cup \{G\}$

- Množina P programových klauzulí, cílová klauzule G
- **Dokazujeme nesplnitelnost**

(1) $P \wedge (\forall \vec{X})(\neg G_1(\vec{X}) \vee \neg G_2(\vec{X}) \vee \cdots \vee \neg G_n(\vec{X}))$

kde $G = \{\neg G_1, \neg G_2, \cdots, \neg G_n\}$ a $\vec{X}$ je vektor proměnných v G

nesplnitelnost (1) je ekvivalentní tvrzení (2) a (3)

(2) $P \vdash \neg G$

(3) $P \vdash (\exists \vec{X})(G_1(\vec{X}) \wedge \cdots \wedge G_n(\vec{X}))$

a jedná se tak o **důkaz existence vhodných objektů**, které na základě vlastností množiny P splňují konjunkci literálů v cílové klauzuli

- Důkaz nesplnitelnosti $P \cup \{G\}$ znamená **nalezení protipříkladu** ten pomocí SLD-stromu **konstruuje termy (odpověď)** splňující konjunkci v (3)

Výpočetní strategie

- **Korektní výpočetní strategie** prohledávání stromu výpočtu musí zaručit, že se každý (konečný) výsledek nalézt v konečném čase
- Korektní výpočetní strategie = **prohledávání stromu do šířky**
  - exponenciální paměťová náročnost
  - složité řídící struktury
- Použitelná výpočetní strategie = **prohledávání stromu do hloubky**
  - jednoduché řídící struktury (zásobník)
  - lineární paměťová náročnost
  - **není ale úplná**: nenalezne vyvrácení i když existuje
    - procházení nekonečné větve stromu výpočtu
      $\Rightarrow$ na nekonečných stromech dojde k zacyklení
    - nedostaneme se tak na jiné existující úspěšné uzly

SLD-rezoluce v Prologu: úplnost

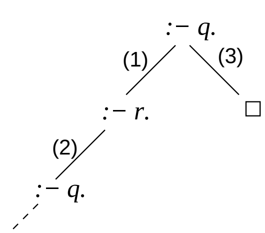
- **Prolog**: prohledávání stromu do hloubky
  $\Rightarrow$ **neúplnost** použité výpočetní strategie
- Implementace SLD-rezoluce v Prologu
  - **není úplná**

logický program: $q : -r.$ (1)

$r : -q.$ (2)

$q.$ (3)

dotaz: $: -q.$



Test výskytu

- Kontrola, zda se proměnná vyskytuje v termu, kterým ji substituujeme
  - dotaz $: -a(B, B).$
  - logický program: $a(X, f(X)).$
  - vede k: $[B/X]$, $[X/f(X)]$
- Unifikátor pro $g(X_1, \ldots, X_n)$ a $g(f(X_0, X_0), f(X_1, X_1), \ldots, f(X_{n-1}, X_{n-1}))$

$$X_1 = f(X_0, X_0), \quad X_2 = f(X_1, X_1), \ldots, \quad X_n = f(X_{n-1}, X_{n-1})$$
$$X_2 = f(f(X_0, X_0), f(X_0, X_0)), \ldots$$

délka termu pro X_k exponenciálně narůstá

$\Longrightarrow$ **exponenciální složitost** na ověření kontroly výskytu

- Test výskytu se **při unifikaci v Prologu neprovádí**
- Důsledek: $? - X = f(X)$ uspěje s $X = f(f(f(f(f(f(f(f(f(f(\ldots))))))))))$

SLD-rezoluce v Prologu: korektnost

- Implementace SLD-rezoluce v Prologu nepoužívá při unifikaci test výskytu

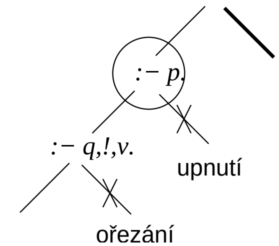
  $\Longrightarrow$ **není korektní**

(1) $t(X) :- p(X, X).$ $:- t(X).$
$p(X, f(X)).$ $X = f(f(f(f(...)))))))))$ problém se projeví

(2) $t :- p(X, X).$ $:- t.$
$p(X, f(X)).$ yes dokazovací systém nehledá unifikátor pro X a $f(X)$

- Řešení: problém typu (2) převést na problém typu (1) ?
  - každá proměnná v hlavě klauzule se objeví i v těle, aby se vynutilo hledání unifikátoru (přidáme $X = X$ pro každou X, která se vyskytuje pouze v hlavě)
    $t :- p(X, X).$
    $p(X, f(X)) :- X = X.$
  - optimalizace v kompilátoru mohou způsobit opět odpověď „yes"

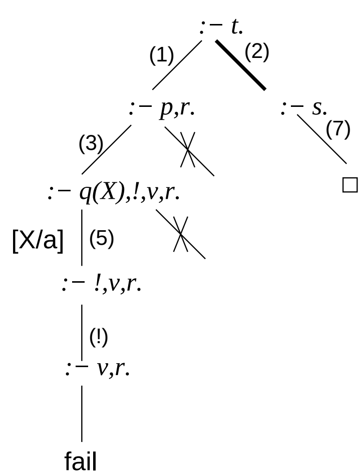
Řízení implementace: řez



- nemá ale žádnou deklarativní sémantiku
- místo toho **mění implementaci programu**
- $p :- q, !, v.$
- snažíme se splnit q
- řez se syntakticky chová jako kterýkoliv jiný literál
- pokud uspěji
  - ⇒ přeskočím řez a pokračuji jako by tam řez nebyl
- pokud ale **neuspěji (a tedy i při backtrackingu) a vracím se přes řez**
  - ⇒ **vracím se až na rodiče** $:- p.$ a zkouším další větev
  - ⇒ nezkouším tedy další možnosti, jak splnit p upnutí
  - ⇒ a nezkouším ani další možnosti, jak splnit q v SLD-stromu ořezání

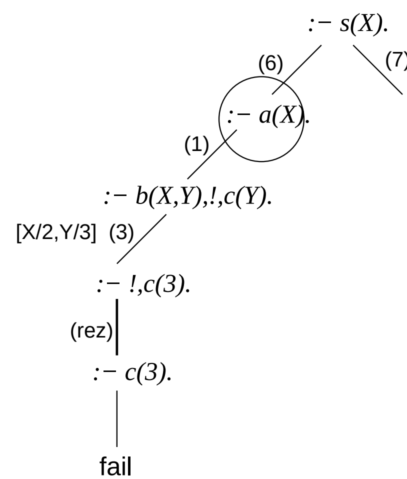
Příklad: řez

$t :- p, r.$	(1)
$t :- s.$	(2)
$p :- q(X), !, v.$	(3)
$p :- u, w.$	(4)
$q(a).$	(5)
$q(b).$	(6)
$s.$	(7)
$u.$	(8)



Příklad: řez II

$a(X) :- b(X, Y), !, c(Y).$	(1)
$a(X) :- c(X).$	(2)
$b(2, 3).$	(3)
$b(1, 2).$	(4)
$c(2).$	(5)
$s(X) :- a(X).$	(6)
$s(X) :- p(X).$	(7)
$p(B) :- q(A, B), r(B).$	(8)
$p(A) :- q(A, A).$	(9)
$q(a, a).$	(10)
$q(a, b).$	(11)
$r(b).$	(12)

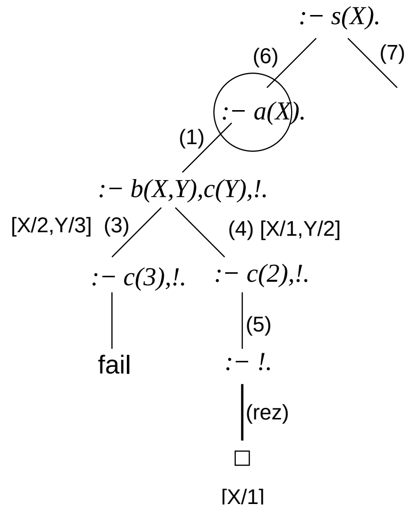


Příklad: řez III

$a(X) :- b(X,Y), c(Y), !.$ (1)
$a(X) :- c(X).$ (2)
$b(2,3).$ (3)
$b(1,2).$ (4)
$c(2).$ (5)

$s(X) :- a(X).$ (6)
$s(X) :- p(X).$ (7)

$p(B) :- q(A,B), r(B).$ (8)
$p(A) :- q(A,A).$ (9)
$q(a,a).$ (10)
$q(a,b).$ (11)
$r(b).$ (12)



Operační a deklarativní semantika

Operační sémantika

- **Operační sémantikou** logického programu P rozumíme množinu $O(P)$ všech atomických formulí bez proměnných, které lze pro nějaký cíl $\mathcal{G}^1$ odvodit nějakým rezolučním důkazem ze vstupní množiny $P \cup \{\mathcal{G}\}$.

  [1]tímto výrazem jsou míněny všechny cíle, pro něž zmíněný rezoluční důkaz existuje.
- **Deklarativní sémantika** logického programu P ???

Opakování: interpretace

- **Interpretace** $\mathcal{I}$ jazyka $\mathcal{L}$ je dána univerzem $\mathcal{D}$ a zobrazením, které přiřadí konstantě c prvek $\mathcal{D}$, funkčnímu symbolu f/n n-ární operaci v $\mathcal{D}$ a predikátovému symbolu p/n n-ární relaci.
  - příklad: $F = \{\{f(a,b) = f(b,a)\}, \{f(f(a,a),b) = a\}\}$
    interpretace $\mathcal{I}_1$: $\mathcal{D} = \mathbb{Z}, a := 1, b := -1, f := "+"$
- Interpretace se nazývá **modelem** formule, je-li v ní tato formule pravdivá
  - interpretace množiny $\mathbb{N}$ s obvyklými operacemi je modelem formule (0 + s(0) = s(0))

Herbrandovy interpretace

- Omezení na obor skládající se ze **symbolických výrazů tvořených z predikátových a funkčních symbolů daného jazyka**
  - při zkoumání pravdivosti není nutné uvažovat modely nad všemi interpretacemi
- **Herbrandovo univerzum**: množina všech termů bez proměnných, které mohou být tvořeny funkčními symboly a konstantami daného jazyka
- **Herbrandova interpretace**: libovolná interpretace, která přiřazuje
  - proměnným prvky Herbrandova univerza
  - konstantám sebe samé
  - funkčním symbolům funkce, které symbolu f pro argumenty $t_1, \cdots, t_n$ přiřadí term $\mathrm{f}(t_1, \cdots, t_n)$
  - predikátovým symbolům libovolnou funkci z Herbrand. univerza do pravdivostních hodnot
- **Herbrandův model** množiny uzavřených formulí $\mathcal{P}$:
  Herbrandova interpretace taková, že každá formule z $\mathcal{P}$ je v ní pravdivá.

Specifikace Herbrandova modelu

- Herbrandovy interpretace mají předdefinovaný význam funktorů a konstant
- Pro specifikaci Herbrandovy interpretace tedy stačí zadat relace pro každý predikátový symbol
- Příklad: Herbrandova interpretace a Herbrandův model množiny formulí

```
lichy(s(0)).                 % (1)
lichy(s(s(X))) :- lichy(X). % (2)
```

  - $\mathcal{I}_1 = \emptyset$ není model (1)
  - $\mathcal{I}_2 = \{lichy(s(0))\}$ není model (2)
  - $\mathcal{I}_3 = \{lichy(s(0)), lichy(s(s(s(0))))\}$ není model (2)
  - $\mathcal{I}_4 = \{lichy(s^n(0)) | n \in \{1, 3, 5, 7, \ldots\}\}$ Herbrandův model (1) i (2)
  - $\mathcal{I}_5 = \{lichy(s^n(0)) | n \in \mathbb{N}\}\}$ Herbrandův model (1) i (2)

Příklad: Herbrandovy interpretace

```
rodic(a,b).
rodic(b,c).
predek(X,Y) :- rodic(X,Y).
predek(X,Z) :- rodic(X,Y), predek(Y,Z).
```

$\mathcal{I}_1 = \{rodic(a,b), rodic(b,c), predek(a,b), predek(b,c), predek(a,c)\}$
$\mathcal{I}_2 = \{rodic(a,b), rodic(b,c),$
$predek(a,b), predek(b,c), predek(a,c), predek(a,a)\}$

$\mathcal{I}_1$ i $\mathcal{I}_2$ jsou Herbrandovy modely klauzulí

Deklarativní a operační sémantika

- Je-li S množina programových klauzulí a M libovolná množina Herbrandových modelů S, pak **průnik těchto modelů** je opět Herbrandův model množiny S.
- **Důsledek:**
  Existuje **nejmenší Herbrandův model** množiny S, který značíme $M(S)$.
- **Deklarativní sémantikou** logického programu P rozumíme jeho minimální Herbrandův model $M(P)$.
- **Operační sémantikou** logického programu P rozumíme množinu $O(P)$ všech atomických formulí bez proměnných, které lze pro nějaký cíl $\mathcal{G}$[1] odvodit nějakým rezolučním důkazem ze vstupní množiny $P \cup \{\mathcal{G}\}$.
  [1]tímto výrazem jsou míněny všechny cíle, pro něž zmíněný rezoluční důkaz existuje.
- Pro libovolný logický program P platí $M(P) = O(P)$

Negace v logickém programování

Negativní znalost

- logické programy vyjadřují **pozitivní znalost**
- **negativní literály:** pozice určena definicí Hornových klauzulí
  ⇒ nelze vyvodit **negativní** informaci z logického programu
  - každý predikát definuje úplnou relaci
  - negativní literál **není** logickým důsledkem programu
- relace vyjádřeny explicitně v nejmenším Herbrandově modelu
  - $nad(X,Y) :- na(X,Y).$ $\qquad na(c,b).$
    $nad(X,Y) :- na(X,Z), nad(Z,Y).$ $\qquad na(b,a).$
  - nejmenší Herbrandův model: $\{na(b,a), na(c,b), nad(b,a), nad(c,b), nad(c,a)\}$
- ani program ani model nezahrnují negativní informaci
  - a není nad c, a není na c
  - i v realitě je negativní informace vyjadřena explicitně zřídka, např. jízdní řád

Předpoklad uzavřeného světa

- neexistence informace chápána jako opak:
  předpoklad uzavřeného světa (*closed world assumption*, CWA)
- převzato z databází
- určitý vztah platí **pouze** když je vyvoditelný z programu.
- „odvozovací pravidlo" (A je (uzavřený) term): $\dfrac{P \not\models A}{\neg A}$ (CWA)
- pro SLD-rezoluci: $P \not\models nad(a,c)$, tedy lze podle CWA odvodit $\neg nad(a,c)$
- problém: není rozhodnutelné, zda daná atomická formule je logickým důsledkem daného logického programu.
  - nelze tedy určit, zda pravidlo CWA je aplikovatelné nebo ne
- CWA v logickém programování obecně nepoužitelná.

Negace jako neúspěch (*negation as failure*)

- slabší verze CWA: **definitivně neúspěšný (*finitely failed*) SLD-strom** cíle $:-A$
  $\dfrac{:-A \text{ má definitivně (konečně) neúspěšný SLD-strom}}{\neg A}$ **(*negation as failure*, NF)**
- **normální cíl**: cíl obsahující i negativní literály
  - $:-nad(c,a), \neg nad(b,c).$
- rozdíl mezi CWA a NF
  - program $nad(X,Y) :- nad(X,Y)$, cíl $:- \neg nad(b,c)$
  - neexistuje odvození cíle podle NF, protože SLD-strom $:-nad(b,c)$ je nekonečný
  - existuje odvození cíle podle CWA, protože neexistuje vyvrácení $:-nad(b,c)$
- CWA i NF jsou nekorektní: A není logickým důsledkem programu P
- řešení: definovat programy tak, aby jejich důsledkem byly i negativní literály
  zúplnění logického programu

Podstata zúplnění logického programu

- převod všech **if** příkazů v logickém programu na **iff**
  - $nad(X,Y) :- na(X,Y).$
    $nad(X,Y) :- na(X,Z), nad(Z,Y).$
  - lze psát jako: $nad(X,Y) :- (na(X,Y)) \vee (na(X,Z), nad(Z,Y)).$
  - zúplnění: $nad(X,Y) \leftrightarrow (na(X,Y)) \vee (na(X,Z), nad(Z,Y)).$
  - X je nad Y **právě tehdy, když alespoň jedna z podmínek platí**
  - tedy **pokud žádná z podmínek neplatí**, X **není** nad Y
- kombinace klauzulí je možná pouze pokud mají identické hlavy
  - $na(c,b).$
    $na(b,a).$
  - lze psát jako: $na(X_1,X_2) :- X_1 = c, X_2 = b.$
    $na(X_1,X_2) :- X_1 = b, X_2 = a.$
  - zúplnění: $na(X_1,X_2) :- (X_1 = c, X_2 = b) \vee (X_1 = b, X_2 = a).$

Zúplnění programu

- **Zúplnění programu** P je: $\mathrm{comp}(P) := \mathrm{IFF}(P) \cup \mathrm{CET}$
- Základní vlastnosti:
  - $\mathrm{comp(P)} \models \mathrm{P}$
  - do programu je přidána pouze negativní informace
- $\mathbf{IFF}(\mathbf{P})$: spojka $:-$ v $\mathrm{IF(P)}$ je nahrazena spojkou $\leftrightarrow$
- $\mathbf{IF}(P)$: množina všech formulí $\mathbf{IF}(q,P)$
  pro všechny predikátové symboly q v programu P
- $def(p/n)$ predikátu p/n množina všech klauzulí predikátu p/n

$\mathbf{IF}(q,P)$

$na(X_1,X_2) :- \underline{\exists Y(X_1 = c, X_2 = b, f(Y))} \vee (X_1 = b, X_2 = a, g).$

$\underline{na(c,b) :- f(Y).}$ $na(b,a) :- g.$

- q/n predikátový symbol programu P
- $X_1, \ldots, X_n$ jsou „nové" proměnné, které se nevyskytují nikde v P
- Nechť C je klauzule ve tvaru
  $q(t_1, \ldots, t_n) :- L_1, \ldots, L_m$
  kde $m \geq 0$, $t_1, \ldots, t_n$ jsou termy a $L_1, \ldots, L_m$ jsou literály.

  Pak označme $\mathbf{E}(C)$ výraz $\exists Y_1, \ldots, Y_k(X_1 = t_1, \ldots, X_n = t_n, L_1, \ldots, L_m)$
  kde $Y_1, \ldots, Y_k$ jsou všechny proměnné v C.
- Nechť $def(q/n) = \{C_1, \ldots, C_n\}$.

  Pak formuli $\mathbf{IF}(q,P)$ získáme následujícím postupem:

  $q(X_1, \ldots, X_n) :- \mathbf{E}(C_1) \vee \mathbf{E}(C_2) \vee \cdots \vee \mathbf{E}(C_j)$ pro $j > 0$ a
  $q(X_1, \ldots, X_n) :- \Box$ pro $j = 0$ [q/n není v programu P]

Clarkova Teorie Rovnosti (CET)

všechny formule jsou univerzálně kvantifikovány:

1. $X = X$
2. $X = Y \rightarrow Y = X$
3. $X = Y \wedge Y = Z \rightarrow X = Z$
4. pro každý f/m: $X_1 = Y_1 \wedge \cdots \wedge X_m = Y_m \rightarrow f(X_1, \ldots, X_m) = f(Y_1, \ldots, Y_m)$
5. pro každý p/m: $X_1 = Y_1 \wedge \cdots \wedge X_m = Y_m \rightarrow (p(X_1, \ldots, X_m) \rightarrow p(Y_1, \ldots, Y_m))$
6. pro všechny různé f/m a g/n, $(m, n \geq 0)$: $f(X_1, \ldots, X_m) \neq g(Y_1, \ldots, Y_n)$
7. pro každý f/m: $f(X_1, \ldots, X_m) = f(Y_1, \ldots, Y_m) \rightarrow X_1 = Y_1 \wedge \cdots \wedge X_m = Y_m$
8. pro každý term t obsahující X jako vlastní podterm: $t \neq X$

$X \neq Y$ je zkrácený zápis $\neg(X = Y)$

Korektnost a úplnost NF pravidla

- **Korektnost NF pravidla**: Nechť P logický program a $:-A$ cíl.
  Jestliže $:-A$ má definitivně neúspěšný SLD-strom,
  pak $\forall(\neg A)$ je logickým důsledkem $\mathrm{comp}(P)$ (nebo-li $\mathrm{comp}(P) \models \forall(\neg A)$)
- **Úplnost NF pravidla**: Nechť P je logický program. Jestliže $\mathrm{comp}(P) \models \forall(\neg A)$,
  pak existuje definitivně neúspěšný SLD-strom $:-A$.
  - zůstává problém: není rozhodnutelné, zda daná atomická formule je logickým důsledkem daného logického programu.
  - teorém mluví pouze o **existenci** definitivně neúspěšného SLD-stromu
  - definitivně (konečně) neúspěšný SLD-strom sice existuje, ale nemusíme ho nalézt
    - např. v Prologu: může existovat konečné odvození, ale program přesto cyklí (Prolog nenajde definitivně neúspěšný strom)
- Odvození pomocí NF pouze **test**, nelze **konstruovat** výslednou substituci
  - v $(\mathrm{comp}(P) \models \forall(\neg A))$ je A všeob. kvantifikováno, v $\forall(\neg A)$ nejsou volné proměnné

Normální a stratifikované programy

- **normální program:** obsahuje negativní literály v pravidlech
- problém: existence zúplnění, která nemají žádný model
  - $p :- \neg p.$ zúplnění: $p \leftrightarrow \neg p$
- rozdělení programu na vrstvy
  - vynucují použití negace relace pouze tehdy pokud je relace úplně definovaná
$a.$	$a.$
$a :- \neg b, a.$	$a :- \neg b, a.$
$b.$	$b :- \neg a.$
stratifikovaný	není stratifikovaný
- normální program P je **stratifikovaný**: množina predikátových symbolů programu lze rozdělit do disjunktních množin $S_0, \ldots, S_m$ ($S_i \equiv$ **stratum**)
  - $p(\ldots) :- \ldots, q(\ldots), \ldots \in P,\ p \in S_k \Longrightarrow q \in S_0 \cup \ldots \cup S_k$
  - $p(\ldots) :- \ldots, \neg q(\ldots), \ldots \in P,\ p \in S_k \Longrightarrow q \in S_0 \cup \ldots \cup S_{k-1}$

Stratifikované programy II

- program je m-**stratifikovaný** $\Longleftrightarrow$ m je nejmenší index takový,
  že $S_0 \cup \ldots \cup S_m$ je množina všech predikátových symbolů z P
- **Věta:** Zúplnění každého stratifikovaného programu má Herbrandův model.
  - $p :- \neg p.$ nemá Herbrandův model
  - $p :- \neg p.$ ale není stratifikovaný
- stratifikované programy nemusí mít **jedinečný** minimální Herbrandův model
  - $cykli :- \neg zastavi.$
  - dva minimální Herbrandovy modely: $\{cykli\}, \{zastavi\}$
  - důsledek toho, že $cykli :- \neg zastavi.$ je ekvivalentní $cykli \vee zastavi$

SLDNF rezoluce: úspěšné odvození

- NF pravidlo: $\dfrac{:-C.\ \text{má konečně neúspěšný SLD-strom}}{\neg C}$
- Pokud máme negativní podcíl $\neg C$ v dotazu G, pak hledáme důkaz pro C
- **Pokud odvození C selže (strom pro C je konečně neúspěšný), pak je odvození G (i $\neg C$) celkově úspěšné**

$nahore(X) :- \neg blokovany(X).$
$blokovany(X) :- na(Y, X).$
$na(a, b).$

$:- nahore(c).$
yes

```
:- nahore(c).            :- blokovany(c).
      |                         |
:-¬ blokovany(c).          :- na(Y,c).
                                |
                              FAIL
```

$\Rightarrow$ **uspěšné odvození**

SLDNF rezoluce: neúspěšné odvození

- NF pravidlo: $\dfrac{:-C. \text{ má konečně neúspěšný SLD-strom}}{\neg C}$
- Pokud máme negativní podcíl $\neg C$ v dotazu G, pak hledáme důkaz pro C
- **Pokud existuje vyvrácení C s prázdnou substitucí (strom pro C je konečně úspěšný), pak je odvození G (i $\neg C$) celkově neúspěšné**

$nahore(X) :- \neg blokovany(X).$
$blokovany(X) :- na(Y,X).$
$na(\_,\_).$

$:- nahore(X).$
no

$:- nahore(X).$
|
$:- \neg blokovany(X).$

$:- blokovany(X).$
|
$:- na(Y,X).$
|
$\Box$

$\Rightarrow$ **neúspěšné odvození**

SLDNF rezoluce: uvázlé odvození

- NF pravidlo: $\dfrac{:-C. \text{ má konečně neúspěšný SLD-strom}}{\neg C}$
- Pokud máme negativní podcíl $\neg C$ v dotazu G, pak hledáme důkaz pro C
- **Pokud existuje vyvrácení C s neprázdnou substitucí (strom pro C je konečně úspěšný), pak je odvození G (i $\neg C$) uvázlé**

$nahore(X) :- \neg blokovany(X).$
$blokovany(X) :- na(Y,X).$
$na(a,b).$

$:- nahore(X).$

$:- nahore(X).$
|
$:- \neg blokovany(X).$

$:- blokovany(X).$
|
$:- na(Y,X).$
| $[Y/a,X/b]$
$\Box$
$[X/b]$

$\Rightarrow$ **uvázlé odvození**

SLD$^+$ odvození

- P je normální program, G_0 normální cíl, R selekční pravidlo:
SLD$^+$-odvození G_0 je buď konečná posloupnost

$$\langle G_0; C_0\rangle, \ldots, \langle G_{i-1}; C_{i-1}\rangle, G_i$$

nebo nekonečná posloupnost

$$\langle G_0; C_0\rangle, \langle G_1; C_1\rangle, \langle G_2; C_2\rangle, \ldots$$

kde v každém kroku $m+1 (m \geq 0)$, R vybírá **pozitivní literál** v G_m a dospívá k G_{m+1} obvyklým způsobem.

- konečné SLD$^+$-odvození může být:
  1. **úspěšné:** $G_i = \Box$
  2. **neúspěšné**
  3. **blokované:** G_i je negativní (např. $\neg A$)

SLDNF rezoluce: pojmy

- **Úroveň cíle**
  - P normální program, G_0 normální cíl, R selekční pravidlo:
  úroveň cíle G_0 se rovná
    - $\mathbf{0} \Longleftrightarrow$ žádné SLD$^+$-odvození s pravidlem R není blokováno
    - $k+\mathbf{1} \Longleftrightarrow$ maximální úroveň cílů $:-A$,
    které ve tvaru $\neg A$ blokují SLD$^+$-odvození G_0, je k
  - nekonečná úroveň cíle: **blokované SLDNF odvození**
- **Množina SLDNF odvození** = {(SLDNF odvození G_0) $\cup$ (SLDNF odvození $:-A$)}
  - při odvozování G_0 jsme se dostali k cíli $\neg A$
- SLDNF odvození cíle G ?

SLDNF rezoluce

P normální program, G_0 normální cíl, R selekční pravidlo:
množina SLDNF odvození a podmnožina neúspěšných SLDNF odvození cíle G_0 jsou takové nejmenší množiny, že:

- každé **SLD⁺-odvození** G_0 je SLDNF odvození G_0
- je-li SLD⁺-odvození $\langle G_0; C_0\rangle, \ldots, G_i$ **blokováno na** $\neg A$
  - tj. G_i je tvaru $:- L_1, \ldots, L_{m-1}, \neg A, L_{m+1}, \ldots, L_n$

  pak
  - **existuje-li SLDNF odvození** $:- A$ (pod R) s prázdnou cílovou substitucí, pak $\langle G_0; C_0\rangle, \ldots, G_i$ je **neúspěšné SLDNF odvození**
  - je-li **každé úplné SLDNF odvození** $:- A$ **(pod** R**) neúspěšné** pak $\langle G_0; C_0\rangle, \ldots, \langle G_i, \epsilon\rangle, (:- L_1, \ldots, L_{m-1}, L_{m+1}, \ldots, L_n)$ je **(úspěšné) SLDNF odvození cíle** G_0
    - ϵ označuje prázdnou cílovou substituci

Typy SLDNF odvození

Konečné SLDNF-odvození může být:

1. **úspěšné:** $G_i = \square$
2. **neúspěšné**
3. **uvázlé (*flounder*):** G_i je negativní ($\neg A$) a $:- A$ je úspěšné **s neprázdnou cílovou substitucí**
4. **blokované:** G_i je negativní ($\neg A$) a $:- A$ nemá konečnou úroveň.

Korektnost a úplnost SLDNF odvození

- **korektnost SLDNF-odvození:**
  P normální program, $:- G$ normální cíl a R je selekční pravidlo:
  je-li θ cílová substituce SLDNF-odvození cíle $:- G$, pak
  $G\theta$ je logickým důsledkem comp(P)
  - implementace SLDNF v Prologu není korektní
  - Prolog neřeší uvázlé SLDNF-odvození (neprázdná substituce)
  - použití bezpečných cílů (negace neobsahuje proměnné)
- **úplnost SLDNF-odvození:** SLDNF-odvození **není** úplné
  - pokud existuje konečný neúspěšný strom $:- A$, pak $\neg A$ platí
    ale místo toho se odvozování $:- A$ může zacyklit, tj. SLDNF rezoluce $\neg A$ neodvodí

  ⇒ $\neg A$ tedy sice platí, ale SLDNF rezoluce ho nedokáže odvodit

Logické programování s omezujícími podmínkami

Constraint Logic Programming: CLP

CP: elektronické materiály

- Dechter, R. **Constraint Processing.** Morgan Kaufmann Publishers, 2003.
  - `http://www.ics.uci.edu/~dechter/ics-275a/fall-2001/readings.html`
- Barták R. **Přednáška Omezující podmínky na MFF UK, Praha**.
  - `http://kti.ms.mff.cuni.cz/~bartak/podminky/prednaska.html`
- **SICStus Prolog User's Manual**, 2004. Kapitola o CLP(FD).
  - `http://www.fi.muni.cz/~hanka/sicstus/doc/html/`
- **Příklady v distribuci SICStus Prologu**: cca 25 příkladů, zdrojový kód
  - `aisa:/software/sicstus-3.10.1/lib/sicstus-3.10.1/library/clpfd/examples/`
- **Constraint Programming Online**
  - `http://slash.math.unipd.it/cp/index.php`

Probírané oblasti

- Obsah
  - úvod: od LP k CLP
  - základy programování
  - základní algoritmy pro řešení problémů s omezujícími podmínkami
- Příbuzné přednášky na FI
  - PA163 Programování s omezujícími podmínkami
    - `http://www.fi.muni.cz/~hanka/cp`
  - PA167 Rozvrhování
    - `http://www.fi.muni.cz/~hanka/rozvrhovani`
    - zahrnuty CP techniky pro řešení rozvrhovacích problémů

Historie a současnost

- **1963** interaktivní grafika (Sutherland: Sketchpad)
- **Polovina 80. let:** logické programování omezujícími podmínkami
- **Od 1990:** komerční využití
- Už v roce **1996:** výnos řádově stovky milionů dolarů
- Aplikace – příklady
  - **Lufthansa:** krátkodobé personální plánování
    - reakce na změny při dopravě (zpoždění letadla, ...)
    - minimalizace změny v rozvrhu, minimalizace ceny
  - **Nokia:** automatická konfigurace sw pro mobilní telefony
  - **Renault:** krátkodobé plánování výroby, funkční od roku 1995

Omezení (*constraint*)

- Dána
  - množina **(doménových) proměnných** $Y = \{y_1, \ldots, y_k\}$
  - **konečná** množina hodnot (**doména**) $D = \{D_1, \ldots, D_k\}$

  Omezení c na Y je podmnožina $D_1 \times \ldots \times D_k$
  - omezuje hodnoty, kterých mohou proměnné nabývat současně
- Příklad:
  - proměnné: A,B
  - domény: {0,1} pro A {1,2} pro B
  - omezení: A≠B nebo (A,B) ∈ {(0,1),(0,2),(1,2)}
- Omezení c definováno na $y_1, \ldots y_k$ je **splněno**, pokud pro $d_1 \in D_1, \ldots d_k \in D_k$ platí $(d_1, \ldots d_k) \in c$
  - příklad (pokračování): omezení splněno pro $(0,1), (0,2), (1,2)$, není splněno pro $(1,1)$

Problém splňování podmínek (CSP)

- Dána
  - konečná množina **proměnných** $X = \{x_1, \ldots, x_n\}$
  - konečná množina hodnot (**doména**) $D = \{D_1, \ldots, D_n\}$
  - konečná množina **omezení** $C = \{c_1, \ldots, c_m\}$
    - omezení je definováno na podmnožině X

  Problém splňování podmínek je trojice (X, D, C)
  (constraint satisfaction problem)
- Příklad:
  - proměnné: A,B,C
  - domény: {0,1} pro A {1,2} pro B {0,2} pro C
  - omezení: A≠B, B≠C

Řešení CSP

- **Částečné ohodnocení proměnných** $(d_1, \ldots, d_k), k < n$
  - některé proměnné mají přiřazenu hodnotu
- **Úplné ohodnocení proměnných** $(d_1, \ldots, d_n)$
  - všechny proměnné mají přiřazenu hodnotu
- **Řešení CSP**
  - úplné ohodnocení proměnných, které splňuje všechna omezení
  - $(d_1, \ldots, d_n) \in D_1 \times \ldots \times D_n$ je **řešení** (X, D, C)
    - pro každé $c_i \in C$ na $x_{i_1}, \ldots x_{i_k}$ platí $(d_{i_1}, \ldots d_{i_k}) \in c_i$
- Hledáme: jedno nebo
  všechna řešení nebo
  optimální řešení (vzhledem k objektivní funkci)

Příklad: jednoduchý školní rozvrh

- **proměnné:** `Jan, Petr, ...`
- **domény:** $\{3,4,5,6\}, \{3,4\}, \ldots$
- **omezení:** `all_distinct([Jan,Petr,...])`
- **částečné ohodnocení:** `Jan=6, Anna=5, Marie=1`
- **úplné ohodnocení:**
  `Jan=6, Petr=3, Anna=5, Ota=2, Eva=4,` **Marie=6**
- **řešení CSP:**
  `Jan=6, Petr=3, Anna=5, Ota=2, Eva=4, Marie=1`
- **všechna řešení:** ještě `Jan=6,` **Petr=4**`, Anna=5, Ota=2,` **Eva=3,** `Marie=1`
- optimálizace: ženy učí co nejdříve
  `Anna+Eva+Marie #= Cena` minimalizace hodnoty proměnné Cena
  optimální řešení: `Jan=6,` **Petr=4,** `Anna=5, Ota=2,` **Eva=3,** `Marie=1`

učitel	min	max
Jan	3	6
Petr	3	4
Anna	2	5
Ota	2	4
Eva	3	4
Marie	1	6

CLP(FD) program

- Základní struktura **CLP programu**
  1. definice proměnných a jejich domén
  2. definice omezení
  3. hledání řešení
- (1) a (2) deklarativní část
  - **modelování** problému
  - vyjádření problému splňování podmínek
- (3) řídící část
  - **prohledávání** stavového prostoru řešení
  - procedura pro hledání řešení (enumeraci) se nazývá **labeling**
  - umožní nalézt jedno, všechna nebo optimální řešení

Kód CLP(FD) programu

```
% základní struktura CLP programu
solve( Variables ) :-
     declare_variables( Variables ),        domain([Jan],3,6], ...
     post_constraints( Variables ),         all_distinct([Jan,Petr,...])
     labeling( Variables ).

% triviální labeling
labeling( [] ).
labeling( [Var|Rest]  ) :-
     fd_min(Var,Min),                % výběr nejmenší hodnoty z domény
     ( Var#=Min, labeling( Rest )
     ;
       Var#>Min , labeling( [Var|Rest] )
     ).
```


Příklad: algebrogram

- Přiřaďte cifry 0, . . . 9 písmenům S, E, N, D, M, O, R, Y tak, aby platilo:
  - SEND + MORE = MONEY
  - různá písmena mají přiřazena různé cifry
  - S a M nejsou 0
- `domain([E,N,D,O,R,Y], 0, 9), domain([S,M],1,9)`
-
```
                1000*S + 100*E + 10*N + D
+               1000*M + 100*O + 10*R + E
#=   10000*M + 1000*O + 100*N + 10*E + Y
```
- `all_distinct( [S,E,N,D,M,O,R,Y] )`
- `labeling( [S,E,N,D,M,O,R,Y] )`

Od LP k CLP I.

- CLP: rozšíření logického programování o omezující podmínky
- CLP systémy se liší podle typu domény
  - **CLP($\mathcal{A}$)** generický jazyk
  - **CLP(FD)** domény proměnných jsou konečné *(Finite Domains)*
  - **CLP($\mathbb{R}$)** doménou proměnných je množina reálných čísel
- Cíl
  - využít syntaktické a výrazové přednosti LP
  - dosáhnout větší efektivity
- **Unifikace v LP je nahrazena splňováním podmínek**
  - unifikace se chápe jako **jedna z** podmínek
  - `A = B`
  - `A #< B,   A in 0..9,   domain([A,B],0,9),   all_distinct([A,B,C])`

Od LP k CLP II.

- Pro řešení podmínek se používají **konzistenční techniky**
  - ***consistency techniques*, propagace omezení *(constraint propagation)***
  - omezení: `A in 0..2, B in 0..2, B #< A`

    domény po propagaci omezení `B #< A`: `A in 1..2, B in 0..1`
- Podmínky jsou deterministicky vyhodnoceny v okamžiku volání podmínky
- **Prohledávání doplněno konzistenčními technikami**
  - `A in 1..2, B in 0..1, B #< A`
  - po provedení `A #= 1` se z `B #< A` se odvodí: `B #= 0`
- **Podmínky jako výstup**
  - kompaktní reprezentace nekonečného počtu řešení, výstup lze použít jako vstup
  - dotaz: `A in 0..2, B in 0..2, B #< A`

    výstup: `A in 1..2, B in 0..1, B #< A`

Syntaxe CLP

- Výběr jazyka omezení
- CLP klauzule

  jako LP klauzule, ale její tělo může obsahovat omezení daného jazyka

```
p(X,Y) :- X #< Y+1, q(X), r(X,Y,Z).
```

- Rezoluční krok v LP
  - kontrola existence mgu mezi cílem a hlavou
- Krok odvození v CLP také zahrnuje
  - kontrola konzistence aktuální množiny omezení s omezeními v těle klauzule

⇒ Vyvolání dvou řešičů: unifikace + řešič omezení

Operační sémantika CLP

- CLP výpočet cíle G
  - $Store$ množina aktivních omezení ≡ **prostor omezení *(constraint store)***
  - inicializace $Store = \emptyset$
  - seznamy cílů v G prováděny v obvyklém pořadí
  - pokud narazíme na cíl s omezením c: $NewStore = Store \cup \{c\}$
  - snažíme se splnit c vyvoláním jeho řešiče
    - při neúspěchu se vyvolá backtracking
    - při úspěchu se podmínky v $NewStore$ zjednoduší propagací omezení
  - zbývající cíle jsou prováděny s upraveným $NewStore$
- CLP výpočet cíle G je úspěšný, pokud se dostaneme z iniciálního stavu $\langle G, \emptyset \rangle$ do stavu $\langle G', Store \rangle$, kde G' je prázdný cíl a $Store$ je splnitelná.

CLP(FD) v SICStus Prologu

Nejpoužívanější systémy a jazyky pro CP

- **Swedish Institute of Computer Science: SICStus Prolog** 1985
  - silná CLP(FD) knihovna, komerční i akademické použití pro širokou škálu platforem
- **IC-PARC, Imperial College London, Cisco Systems: ECL^{i}PS^e** 1984
  - široké možnosti kooperace mezi různými řešičemi: konečné domény, reálná čísla, `repair`
  - od 2004 vlastní Cisco Systems volně dostupné pro akademické použití, rozvoj na IC-PARC, platformy: Windows, Linux, Solaris
- **ILOG, omezující podmínky v C++** 1987
  - komerční společnost, vznik ve Francie, nyní rozšířená po celém světě
  - implementace podmínek založena na objektově orientovaném programování
- `http://www.fi.muni.cz/~hanka/bookmarks.html#tools`
  - cca 50 odkazů na nejrůznější systémy: Prolog, C++, Java, Lisp, . . .
  - COSYTEC: CHIP, PrologIA: Prolog IV, Siemens: IF Prolog,

    akademický: Mozart (objektově orientované, deklarativní programování, *concurrency*), . . .

CLP(FD) v SICStus Prologu

- CLP není dostupné v SWI Prologu
- CLP knihovna v ECLiPSe se liší
- Vestavěné predikáty jsou dostupné v separátním modulu (knihovně)
  `:- use_module(library(clpfd)).`
- Obecné principy platné všude
- Stejné/podobné vestavěné predikáty existují i jinde

Příslušnost k doméně: Range termy

- `?- domain( [A,B], 1,3).` `domain( +Variables, +Min, +Max)`
  `A in 1..3`
  `B in 1..3`
- `?- A in 1..8, A #\= 4.` `?X in +Min..+Max`
  `A in (1..3) \/ (5..8)`
- Doména reprezentována jako posloupnost intervalů celých čísel
- `?- A in (1..3) \/ (8..15) \/ (5..9) \/ {100}.` `?X in +Range`
  `A in (1..3) \/ (5..15) \/ {100}`
- Zjištění domény Range proměnné `Var`: `fd_dom(?Var,?Range)`
  - `A in 1..8, A #\= 4, fd_dom(A,Range).` `Range=(1..3) \/ (5..8)`
  - `A in 2..10, fd_dom(A,(1..3) \/ (5..8)).` `no`
- Range term: reprezentace nezávislá na implementaci

Příslušnost k doméně: FDSet termy

- FDSet term: reprezentace závislá na implementaci
- `?- A in 1..8, A #\= 4, fd_set(A,FDSet).` `fd_set(?Var,?FDSet)`
  `A in (1..3) \/ (5..8)`
  `FDSet = [[1|3],[5|8]]`
- `?- A in 1..8,A #\= 4, fd_set(A,FDSet),B in_set FDSet.` `?X in_set +FDSet`
  `A in (1..3) \/  (5..8)`
  `FDSet = [[1|3],[5|8]]`
  `B in (1..3) \/  (5..8)`
- FDSet termy představují nízko-úrovňovou implementaci
- FDSet termy nedoporučeny v programech
  - používat pouze predikáty pro manipulaci s nimi
  - omezit použití `A in_set [[1|2],[6|9]]`
- Range termy preferovány

Další `fd_...` predikáty

- `fdset_to_list(+FDset, -List)` vrací do seznamu prvky FDset
- `list_to_fdset(+List, -FDset)` vrací FDset odpovídající seznamu
- `fd_var(?Var)` je `Var` doménová proměnná?
- `fd_min(?Var,?Min)` nejmenší hodnota v doméně
- `fd_max(?Var,?Max)` největší hodnota v doméně
- `fd_size(?Var,?Size)` velikost domény
- `fd_degree(?Var,?Degree)` počet navázaných omezení na proměnné
  - mění se během výpočtu: pouze aktivní omezení, i odvozená aktivní omezení

Aritmetická omezení

- `Expr RelOp Expr`

  `RelOp -> #= | #\= | #< | #=< | #> | #>=`

  - `A + B #=< 3,`
    `A #\= (C - 4) * ( D - 5), A/2 #= 4`
- `sum(Variables,RelOp,Suma)`
  - `domain([A,B,C,F],1,3),`
    `sum([A,B,C],#= ,F)`
- `scalar_product(Coeffs,Variables,RelOp,ScalarProduct)`
  - `domain([A,B,C,F],1,6),`
    `scalar_product( [1,2,3],[A,B,C],#= ,F)`

Výroková omezení, reifikace

- **Výroková omezení** pozor na efektivitu
  - `\#` negace, `#/\` konjunkce, `#\/` disjunkce, `#<=>` ekvivalence, ...
  - `X #> 4 #/\ Y #< 6`
  - za předpokladu `A in 1..4`:

    `A#\= 3, A#\= 4` `A#\= 3 #/\ A#\= 4` A#=1 #\/ A#=2

    A in (inf..2)\/ (5..sup) A in (inf..2)\/ (5..sup) A in inf..sup

    tj. propagace disjunkce A#=1 #\/ A#=2 je příliš slabá (propagační algoritmy příliš obecné)
- **Reifikace** pozor na efektivitu
  - `Constraint #<=> Bool`

    `Bool in 0..1` v závislosti na tom, zda je `Constraint` splněn
  - příklad: `A in 1..10 #<=> Bool`
  - za předpokladu `X in 3..10, Y in 1..4, Bool in 0..1`

    porovnej rozdíl mezi X#<Y `X#<Y #<=> Bool`

    X = 3, Y = 4 X in 3..10, Y in 1..4, Bool in 0..1

Příklad: reifikace

- Přesně N prvků v seznamu S je rovno X: `exactly(X,S,N)`

```
exactly(_, [], 0).
exactly(X, [Y|L], N) :-
        X #= Y #<=> B,                          % reifikace
        N #= M+B,          % doménová proměnná místo akumulátoru
        exactly(X, L, M).
```

  - `| ?- domain([A,B,C,D,E,N],1,2), exactly(1,[A,B,C,D,E],N),A#< 2,B#< 2.`
    `A = 1, B = 1, C = 2, D = 2, E = 2, N = 2`
- Vyzkoušejte si
  - `greaters(X,S,N)`: přesně N prvků v seznamu S je větší než X
  - `atleast(X,S,N)`: alespoň N prvků v seznamu S je rovno X
  - `atmost(X,S,N)`: nejvýše N prvků v seznamu S je rovno X

Základní kombinatorická omezení

- `element(N,List,X)`
  - omezení na konkrétní prvek seznamu
- `all_distinct(List)`
  - všechny proměnné různé
- `serialized(Starts,Durations)`
  - disjunktivní rozvrhování
- `disjoint2(Rectangles)`
  - nepřekrývání obdélníků
- `cumulative(Starts,Durations,Resources,Limit)`
  - kumulativní rozvrhování

Výskyty prvků v seznamu

- `element(N,List,X)`
  - N-tý prvek v seznamu `List` je roven `X`
  - `| ?- A in 2..10, B in 1..3, element( N, [A,B], X ), X#< 2.`
    B = 1, N = 2, X = 1, A in 2..10

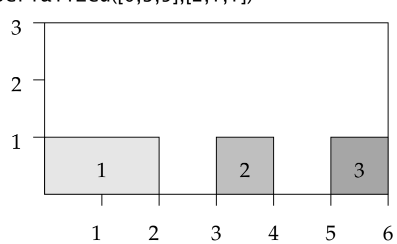
Všechny proměnné různé

- `all_distinct(Variables)`, `all_different(Variables)`
- Proměnné v seznamu `Variables` jsou různé
- `all_distinct` a `all_different` se liší úrovní propagace
  - `all_distinct` má úplnou propagaci
  - `all_different` má slabší (neúplnou) propagaci
- Příklad: učitelé musí učit v různé hodiny
  - `all_distinct([Jan,Petr,Anna,Ota,Eva,Marie])`
    `Jan = 6, Ota = 2, Anna = 5,`
    `Marie = 1, Petr in 3..4, Eva in 3..4`
  - `all_different([Jan,Petr,Anna,Ota,Eva,Marie])`
    `Jan in 3..6, Petr in 3..4, Anna in 2..5,`
    `Ota in 2..4, Eva in 3..4, Marie in 1..6`

učitel	min	max
Jan	3	6
Petr	3	4
Anna	2	5
Ota	2	4
Eva	3	4
Marie	1	6

Disjunktivní rozvrhování

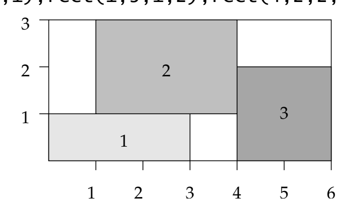
- `serialized(Starts,Durations)`
- Rozvržení úloh zadaných startovním časem (seznam `Starts`) a dobou trvání (seznam **nezáporných** `Durations`) tak, aby se nepřekrývaly
  - příklad s konstantami: `serialized([0,3,5],[2,1,1])`



  - příklad: vytvoření rozvrhu, za předpokladu, že **doba trvání hodin není stejná**
    - `D in 1..2, C = 3,`
      `serialized([Jan,Petr,Anna,Ota,Eva,Marie], [D,D,D,C,C,C])`

Nepřekrývání obdélníků

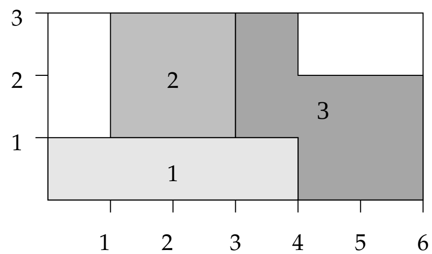
- `disjoint2(Rectangles)` `disjoint1(Lines)`
  `disjoint2( [Name(X, Delka, Y, Vyska) | _ ] )`
- umístění obdélníků ve dvourozměrném prostoru
  doménové proměnné `X,Y,Delka,Vyska` mohou být z oboru **celých čísel**
  - příklad s konstantami:
    `disjoint2([rect(0,3,0,1),rect(1,3,1,2),rect(4,2,2,-2)])`



  - příklad: vytvoření rozvrhu za předpokladu, že **učitelé učí v různých místnostech**
    `D in 1..2, C = 3,`
    `disjoint2( class(Jan,D,M1,1), class(Petr,D,M2,1), class(Petr,D,M3,1), ...]`

Kumulativní rozvrhování

- `cumulative(Starts,Durations,Resources,Limit)`
- Úlohy jsou zadány startovním časem (seznam `Starts`), dobou trvání (seznam `Durations`) a požadovanou kapacitou zdroje (seznam `Resources`)
- Rozvržení úloh tak, aby celková kapacita zdroje nikdy nepřekročila `Limit`
- Příklad s konstantami: `cumulative`([0,1,3],[4,2,3],[1,2,2],3)



Příklad: kumulativní rozvrhování

- Vytvořte rozvrh pro následující úlohy, tak aby nebyla překročena kapacita 13 zdroje, a minimalizujte celkovou dobu trvání

úloha	doba trvání	kapacita
t1	16	2
t2	6	9
t3	13	3
t4	7	7
t5	5	10
t6	18	1
t7	4	11

```
schedule(Ss, End) :-
    length(Ss, 7),
    Ds = [16, 6, 13, 7, 5, 18, 4],
    Rs = [ 2, 9, 3, 7, 10, 1, 11],
    domain(Ss, 0, 51),
    domain([End], 0, 69),
    after(Ss, Ds, End),    % koncový čas
    cumulative(Ss, Ds, Rs, 13),
    append(Ss, [End], Vars),
    labeling([minimize(End)], Vars).
after([], [], _).
after([S|Ss], [D|Ds], E) :-
    E #>= S+D, after(Ss, Ds, E).
| ?-  schedule(Ss, End).
Ss = Ss = [0,16,9,9,4,4,0], End = 22 ?
```


Vestavěné predikáty pro labeling

- Instanciace proměnné `Variable` hodnotami v její doméně

  `indomain( Variable )`

  hodnoty jsou instanciovány při backtrackingu ve vzrůstajícím pořadí

```
?- X in 4..5, indomain(X).
   X = 4 ? ;
   X = 5 ?

labeling( [] ).
labeling( [Var|Rest]  ) :-     % výběr nejlevější proměnné k instanciaci
     indomain( Var ),          % výběr hodnot ve vzrůstajícím pořadí
     labeling( Rest ).
```

- `labeling( Options, Variables )`

```
?- A in 0..2, B in 0..2, B#< A, labeling([], [A,B]).
```


Uspořádání hodnot a proměnných

- Při prohledávání je rozhodující **uspořádání hodnot a proměnných**
- Určují je **heuristiky výběru hodnot a výběru proměnných**

```
labeling( [] ).
labeling( Variables  ) :-
     select_variable(Variables,Var,Rest),
     select_value(Var,Value),
     ( Var #= Value,
       labeling( Rest )
     ;
       Var #\= Value ,          % nemusí dojít k instanciaci  Var
       labeling( Variables )   % proto pokračujeme se všemi proměnnými včetně  Var
     ).
```

- **Statické uspořádání:** určeno už před prohledáváním
- **Dynamické uspořádání:** počítá se během prohledávání

Výběr hodnoty

- Obecný princip výběru hodnoty: **první úspěch *(succeed first)***
  - volíme pořadí tak, abychom výběr nemuseli opakovat
  - `?- domain([A,B,C],1,2), A#=B+C.` optimální výběr A=2,B=1,C=1 je bez backtrackingu
- Parametry `labeling/2` ovlivňující výběr hodnoty   př. `labeling([down], Vars)`
  - up: doména procházena ve vzrůstajícím pořadí (default)
  - down: doména procházena v klesajícím pořadí
- Parametry `labeling/2` řídící, jak je výběr hodnoty realizován
  - `step`: volba mezi `X #= M, X #\= M` (default)
    - viz dřívější příklad u "Uspořádání hodnot a proměnných"
  - `enum`: vícenásobná volba mezi všemi hodnotami v doméně
    - podobně jako při `indomain/1`
  - `bisect`: volba mezi `X #=< Mid, X #> Mid`
    - v jednom kroku labelingu nedochází nutně k instanciaci proměnné

Výběr proměnné

- Obecný princip výběru proměnné: ***first-fail***
  - výběr proměnné, pro kterou je nejobtížnější nalézt správnou hodnotu
    pozdější výběr hodnoty pro tuto proměnnou by snadněji vedl k failu
  - vybereme proměnnou **s nejmenší doménou**
  - `?- domain([A,B,C],1,3), A#<3, A#=B+C.`   nejlépe je začít s výběrem A
- Parametry `labeling/2` ovlivňující výběr proměnné
  - `leftmost`: nejlevější (default)
  - `ff`: s (a) nejmenší velikostí domény   `fd_size(Var,Size)`
    (b) nejlevější z nich
  - `ffc`: s (a) nejmenší velikostí domény
    (b) největším množstvím omezením „čekajících" na proměnné   `fd_degree(Var,Size)`
    (c) nejlevější z nich
  - `min/max`: s (a) nejmenší/největší hodnotou v doméně proměnné
    (b) nejlevnější z nich   `fd_min(Var,Min)/fd_max(Var,Max)`

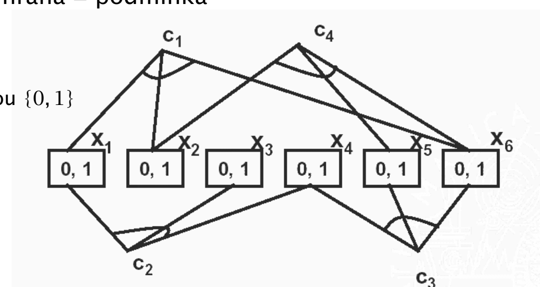
Hledání optimálního řešení   **(předpokládejme minimalizaci)**

- Parametry `labeling/2` pro optimalizaci: `minimize(F)/maximize(F)`
  - `Cena #= A+B+C, labeling([minimize(Cena)], [A,B,C])`
- **Metoda větví a mezí *(branch&bound)***
  - uvažujeme nejhorší možnou cenu řešení UB (např. cena už nalezeného řešení)
  - počítáme dolní odhad LB ceny částečného řešení
    LB je tedy nejlepší možná cena pro rozšíření tohoto řešení
  - procházíme strom a vyžadujeme, aby prozkoumávaná větev měla cenu $LB < UB$
    pokud je $LB \geq UB$, tak víme, že v této větvi není lepší řešení a odřízneme ji

Algoritmy pro řešení problému splňování podmínek (CSP)

Grafová reprezentace CSP

- **Reprezentace podmínek**
  - intenzionální (matematická/logická formule)
  - extenzionální (výčet k-tic kompatibilních hodnot, 0-1 matice)
- **Graf:** vrcholy, hrany (hrana spojuje dva vrcholy)
- **Hypergraf:** vrcholy, hrany (hrana spojuje množinu vrcholů)
- Reprezentace CSP pomocí **hypergrafu podmínek**
  - vrchol = proměnná, hyperhrana = podmínka
- Příklad
  - proměnné $x_1, \ldots, x_6$ s doménou $\{0, 1\}$
  - omezení $c_1 : x_1 + x_2 + x_6 = 1$
    $c_2 : x_1 - x_3 + x_4 = 1$
    $c_3 : x_4 + x_5 - x_6 > 0$
    $c_4 : x_2 + x_5 - x_6 = 0$



Binární CSP

- **Binární CSP**
  - CSP, ve kterém jsou pouze binární podmínky
  - unární podmínky zakódovány do domény proměnné
- **Graf podmínek** pro binární CSP
  - není nutné uvažovat hypergraf, stačí graf (podmínka spojuje pouze dva vrcholy)
- **Každý CSP lze transformovat na "korespondující" binární CSP**
- Výhody a nevýhody binarizace
  - získáváme unifikovaný tvar CSP problému, řada algoritmů navržena pro binární CSP
  - bohužel ale značné zvětšení velikosti problému
- Nebinární podmínky
  - složitější propagační algoritmy
  - lze využít jejich sémantiky pro lepší propagaci
    - příklad: `all_different` vs. množina binárních nerovností

Vrcholová a hranová konzistence

- **Vrcholová konzistence *(node consistency)* NC**
  - každá hodnota z aktuální domény V_i proměnné splňuje všechny unární podmínky s proměnnou V_i
- **Hranová konzistence *(arc consistency)* AC** pro **binární CSP**
  - **hrana** (V_i, V_j) **je hranově konzistentní**, právě když pro každou hodnotu x z aktuální domény D_i existuje hodnota y tak, že ohodnocení $[V_i = x, V_j = y]$ splňuje všechny binární podmínky nad V_i, V_j.
  - hranová konzistence je **směrová**
    - konzistence hrany (V_i, V_j) nezaručuje konzistenci hrany (V_j, V_i)

A [3..7] –A<B→ [1..5] B A [3..4] –A<B→ [1..5] B A [3..4] ⇄A<B [4..5] B

konzistence (A,B) konzistence (A,B) i (B,A)

  - **CSP** je **hranově konzistentní**, právě když jsou všechny jeho hrany (v obou směrech) hranově konzistentní

Je hranová konzistence dostatečná?

- Použitím AC odstraníme mnoho nekompatibilních hodnot
  - Dostaneme potom řešení problému? NE
  - Víme alespoň zda řešení existuje? NE
- `domain([X,Y,Z],1,2), X#\= Y, Y#\= Z, Z#\= X`
  - hranově konzistentní
  - nemá žádné řešení
- Jaký je tedy význam AC?
  - někdy dá řešení přímo
    - nějaká doména se vyprázdní ⇒ řešení neexistuje
    - všechny domény jsou jednoprvkové ⇒ máme řešení
  - v obecném případě se alespoň zmenší prohledávaný prostor

Řešení nebinárních podmínek

- S n-árními podmínkami se pracuje přímo
- Podmínka je **obecně hranově konzistentní (GAC)**, právě když pro každou proměnnou V_i z této podmínky a každou hodnotu $x \in D_i$ existuje ohodnocení zbylých proměnných v podmínce tak, že podmínka platí
  - `A + B = C, A in 1..3, B in 2..4, C in 3..7` je obecně hranově konzistentní
- Využívá se sémantika podmínek
  - speciální typy konzistence pro globální omezení
    - viz `all_distinct`
  - konzistence mezí
    - propagace pouze při změně nejmenší a největší hodnoty v doméně proměnné
- Pro různé podmínky lze použít různý druh konzistence
  - A#<B: hranová konzistence, konzistence mezí

Konzistenční algoritmus pro nebinární podmínky

- Algoritmus s **frontou proměnných** (někdy též nazýván AC-8)
  - opakovaně se provádí revize podmínek, dokud se mění domény

```
procedure Nonbinary-AC-3-with-Variables(Q)
while Q non empty do
      vyber a smaž V_j ∈Q
      for ∀C takové, že V_j ∈ scope(C) do
          W := revise(V_j,C)
             // W je množina proměnných jejichž, doména se změnila
          if ∃ V_i ∈ W taková, že D_i = ∅ then return fail
          Q := Q ∪ {W}
end  Non-binary-consistency
```

  - **rozsah omezení** $scope(C)$: množina proměnných, na nichž je C definováno
- Implementace
  - u každé proměnné je seznam **vybraných podmínek** pro propagaci
  - REVISE procedury pro tyto podmínky definuje uživatel v závislosti na typu podmínky

Revize podmínky pro hranovou konzistenci

- Jak udělat podmínku $c(V_j,V_i)$ na hraně (V_j,V_i) hranově konzistentní vůči V_j?
- Z domény D_j vyřadím takové hodnoty x, které nejsou konzistentní s aktuální doménou D_i (pro x neexistuje žádá hodnoty y v D_i tak, aby ohodnocení $V_j = x$ a $V_i = y$ splňovalo binární podmínku $c(V_j,V_i)$ mezi V_j a V_i)
-

```
procedure revise(V_j,c(V_j,V_i))
Deleted := false
for ∀x in D_j do
    if neexistuje y ∈ D_i takové, že (x,y) je konzistentní
    then D_j := D_j − {x}
         Deleted := true
    end if
return Deleted
end revise
```

- `domain([`V_1,V_2`],2,4),` V_1`#<` V_2 `revise(`V_1`,` V_1`#<` V_2`)` smaže 4 z D_1,D_2 se nezmění

Konzistence mezí

- ***Bounds consistency*** **BC**: slabší než obecná hranová konzistence
  - podmínka má **konzistentní meze (BC)**, právě když pro každou proměnnou V_j z této podmínky a každou hodnotu $x \in D_j$ existuje ohodnocení zbylých proměnných v podmínce tak, že je podmínka splněna a pro vybrané ohodnocení y_i proměnné V_i platí $\min(D_i) \leq y_i \leq \max(D_i)$
  - stačí propagace pouze při **změně minimální nebo maximální hodnoty (při změně mezí)** v doméně proměnné
- **Konzistence mezí pro nerovnice**
  - `A #> B => min(A) = min(B)+1, max(B) = max(A)-1`
  - příklad: `A in 4..10, B in 6..18, A #> B`
    `min(A) = 6+1` ⇒ `A in 7..10`
    `max(B) = 10-1` ⇒ `B in 6..9`
  - podobně: `A #< B`, `A #>= B`, `A #=< B`

Konzistence mezí a aritmetická omezení

- `A #= B + C => min(A) = min(B)+min(C), max(A) = max(B)+max(C)`
  `min(B) = min(A)-max(C), max(B) = max(A)-min(C)`
  `min(C) = min(A)-max(B), max(C) = max(A)-min(B)`
  - změna `min(A)`vyvolá pouze změnu `min(B)` a `min(C)`
  - změna `max(A)`vyvolá pouze změnu `max(B)` a `max(C), ...`
- Příklad: `A in 1..10, B in 1..10, A #= B + 2, A #> 5, A #\= 8`

  `A #= B + 2` ⇒ `min(A)=1+2, max(A)=10+2` ⇒ `A in 3..10`

  ⇒ `min(B)=1-2, max(B)=10-2` ⇒ `B in 1..8`

  `A #> 5` ⇒ `min(A)=6` ⇒ `A in 6..10`

  ⇒ `min(B)=6-2` ⇒ `B in 4..8` (nové vyvolání `A #= B + 2`)

  `A #\= 8` ⇒ `A in (6..7) \/ (9..10)` (meze stejné, k propagaci `A #= B + 2` nedojde)
- Vyzkoušejte si: `A #= B - C, A #>= B + C`

Globální podmínky

- Propagace je lokální
  - pracuje se s jednotlivými podmínkami
  - interakce mezi podmínkami je pouze přes domény proměnných
- Jak dosáhnout více, když je silnější propagace drahá?
- Seskupíme několik podmínek do jedné tzv. **globální podmínky**
- Propagaci přes globální podmínku řešíme
  speciálním algoritmem navrženým pro danou podmínku
- Příklady:
  - `all_different` omezení: hodnoty všech proměnných různé
  - `serialized` omezení:
    rozvržení úloh zadaných startovním časem a dobou trvání tak, aby se nepřekrývaly

Propagace pro `all_distinct`

učitel	min	max
Jan	3	6
Petr	**3**	**4**
Anna	2	5
Ota	**2**	**4**
Eva	**3**	**4**
Marie	1	6

- **U = {X2, X4, X5}, dom(U) = {2, 3, 4}**:

  `{2,3,4}` nelze pro `X1, X3, X6`

  `X1 in 5..6, X3 = 5, X6 in {1} \/ (5..6)`
- **Konzistence:** $\forall\{X_1,\ldots,X_k\}\subset V : card\{D_1\cup\cdots\cup D_k\}\geq k$
  stačí hledat **Hallův interval** I: velikost intervalu I je rovna
  počtu proměnných, jejichž doména je v I
- **Inferenční pravidlo**
  - $U=\{X_1,\ldots,X_k\}$, $dom(U)=\{D_1\cup\cdots\cup D_k\}$
  - $card(U)=card(dom(U))\Rightarrow\forall v\in dom(U),\forall X\in(V-U), X\neq v$
  - hodnoty v Hallově intervalu jsou pro ostatní proměnné nedostupné
- **Složitost**: $O(2^n)$ – hledání všech podmnožin množiny n proměnných (naivní)
  $O(n\log n)$ – kontrola hraničních bodů Hallových intervalů (1998)

Prohledávání + konzistence

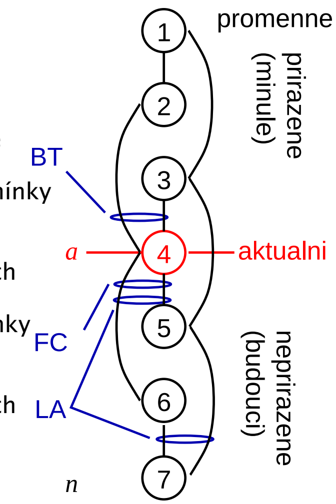
- Splňování podmínek **prohledáváním** prostoru řešení
  - podmínky jsou užívány pasivně jako test
  - přiřazuji hodnoty proměnných a zkouším co se stane
  - vestavěný prohledávací algoritmus Prologu: **backtracking**, triviální: **generuj & testuj**
  - úplná metoda (nalezneme řešení nebo dokážeme jeho neexistenci)
  - zbytečně pomalé (exponenciální): procházím i „evidentně" špatná ohodnocení
- **Konzistenční (propagační) techniky**
  - umožňují odstranění nekonzistentních hodnot z domény proměnných
  - neúplná metoda (v doméně zůstanou ještě nekonzistentní hodnoty)
  - relativně rychlé (polynomiální)
- Používá se **kombinace obou metod**
  - postupné přiřazování hodnot proměnným
  - po přiřazení hodnoty odstranění nekonzistentních hodnot konzistenčními technikami

Prohledávání s navracením

- Základní prohledávací algoritmus pro problémy splňování podmínek
- **Prohledávání stavového prostoru do hloubky**
- Dvě fáze prohledávání s navracením
  - **dopředná fáze**: proměnné jsou postupně vybírány, rozšiřuje se částečné řešení přiřazením konzistení hodnoty (pokud existuje) další proměnné
    - po vybrání hodnoty testujeme konzistenci
  - **zpětná fáze**: pokud neexistuje konzistentní hodnota pro aktuální proměnnou, algoritmus se vrací k předchozí přiřazené hodnotě
- Proměnné dělíme na
  - **minulé** – proměnné, které už byly vybrány (a mají přiřazenu hodnotu)
  - **aktuální** – proměnná, která je právě vybrána a je jí přiřazována hodnota
  - **budoucí** – proměnné, které budou vybrány v budoucnosti

Přehled algoritmů

- **Backtracking (BT)** kontroluje v kroku a podmínky
  $c(V_1, V_a), \ldots, c(V_{a-1}, V_a)$
  z minulých proměnných do aktuální proměnné
- **Kontrola dopředu (FC)** kontroluje v kroku a podmínky
  $c(V_a, V_{a+1}), \ldots, c(V_a, V_n)$
  z aktuální proměnné do budoucích proměnných
- **Pohled dopředu (LA)** kontroluje v kroku a podmínky
  $\forall l(a \le l \le n), \forall k(a \le k \le n), k \ne l : c(V_k, V_l)$
  z aktuální proměnné do budoucích proměnných
  a mezi budoucími proměnnými



Základní algoritmus prohledávání s navracením

- Pro jednoduchost proměnné očíslujeme a ohodnocujeme je v daném pořadí
- Na začátku voláno jako `labeling(G,1)`

procedure labeling(G,a)
if a > |uzly(G)| then return uzly(G)
for $\forall x \in D_a$ do
    if consistent(G,a) then   % consistent(G,a) je nahrazeno FC, LA, ...
        R := labeling(G,$a+1$)
        if R $\ne$ fail then return R
return fail
end labeling

Po přiřazení všech proměnných vrátíme jejich ohodnocení

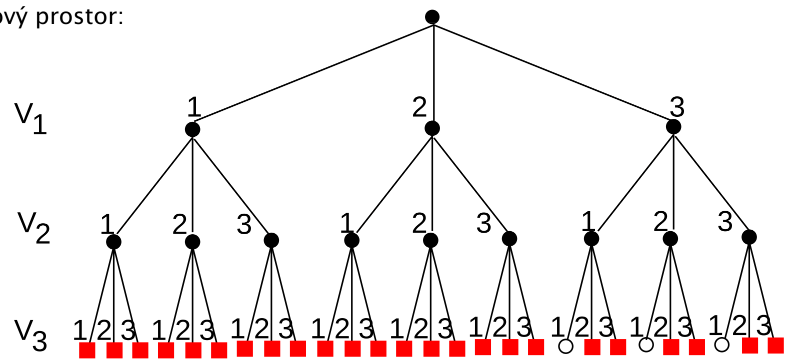
- Procedury `consistent` uvedeme pouze pro binární podmínky

Backtracking (BT)

- Backtracking ověřuje v každém kroku konzistenci podmínek vedoucích z minulých proměnných do aktuální proměnné
- Backtracking tedy zajišťuje konzistenci podmínek
  - na všech minulých proměnných
  - na podmínkách mezi minulými proměnnými a aktuální proměnnou
- procedure BT(G,a)
  Q:=$\{(V_i, V_a) \in$ hrany(G), $i < a\}$ % hrany vedoucí do minulých proměnných
  Consistent := true
  while Q není prázdná $\wedge$ Consistent do
    vyber a smaž libovolnou hranu (V_k, V_m) z Q
    Consistent := not revise(V_k, V_m) % pokud vyřadíme prvek, bude doména prázdná
  return Consistent
  end BT

Příklad: backtracking

- Omezení: V_1, V_2, V_3 in $1 \ldots 3$, $\quad V_1 \# = 3 \times V_3$
- Stavový prostor:



  - červené čtverečky: chybný pokus o instanciaci, řešení neexistuje
  - nevyplněná kolečka: nalezeno řešení
  - černá kolečka: vnitřní uzel, máme pouze částečné přiřazení

Kontrola dopředu *(FC – forward checking)*

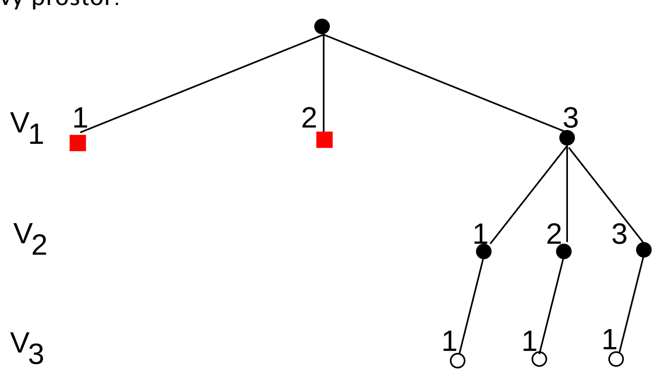
- FC je rozšíření backtrackingu
- FC navíc zajišťuje konzistenci mezi aktuální proměnnou a budoucími proměnnými, které jsou s ní spojeny dosud nesplněnými podmínkami
- procedure FC(G,a)

```
Q:={(V_i,V_a) ∈ hrany(G), i > a}     % přidání hran z aktuální proměnné do budoucích prom.
Consistent := true
while Q není prázdná ∧ Consistent do
      vyber a smaž libovolnou hranu (V_k,V_m) z Q
      if revise((V_k,V_m)) then
         Consistent := (|D_k| > 0)       % vyprázdnění domény znamená nekonzistenci
return Consistent
end FC
```

- Hrany z aktuální proměnné do minulých proměnných není nutno testovat

Příklad: kontrola dopředu

- Omezení: V_1, V_2, V_3 in $1 \ldots 3$, $\quad V_1 \# = 3 \times V_3$
- Stavový prostor:



Pohled dopředu *(LA – looking ahead)*

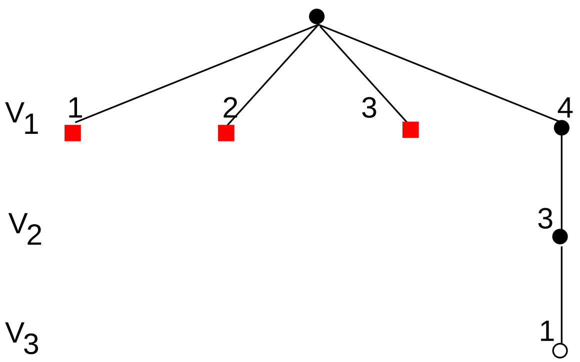
- LA je rozšíření FC, LA zajišťuje hranovou konzistenci
- LA navíc ověřuje i konzistenci všech hran mezi budoucími proměnnými
- procedure LA(G,a)

```
Q := {(V_i,V_a) ∈ hrany(G), i > a}        % začínáme s hranami do a
Consistent := true
while Q není prázdná ∧ Consistent do
      vyber a smaž libovolnou hranu (V_k,V_m) z Q
      if revise((V_k,V_m)) then
         Q := Q ∪ {(V_i,V_k)|(V_i,V_k) ∈ hrany(G), i ≠ k, i ≠ m, i > a}
         Consistent := (|D_k| > 0)
return Consistent
end LA
```

  - Hrany z aktuální proměnné do minulých proměnných opět netestujeme

Příklad: pohled dopředu

- Omezení: V_1, V_2, V_3 in $1 \ldots 4$, $\quad V_1\# > V_2$, $\quad V_2\# = 3 \times V_3$
- Stavový prostor:



Implementace Prologu

Literatura:

- Matyska L., Toman D.: Implementační techniky Prologu , Informační systémy, (1990), 21-59. `http://www.ics.muni.cz/people/matyska/vyuka/lp/lp.ht`

Opakování: základní pojmy

- Konečná množina klauzulí `Hlava :- Tělo` tvoří **program P**.
- **`Hlava`** je literál
- **`Tělo`** je (eventuálně prázdná) konjunkce literálů $T_1, \ldots T_a$, $a \geq 0$
- **Literál**
  je tvořen m-árním predikátovým symbolem (m/p) a m termy (argumenty)
- **Term** je konstanta, proměnná nebo složený term.
- **Složený term**
  a n termy na místě argumentů
- **Dotaz (cíl)** je neprázdná množina literálů.

Interpretace

Deklarativní sémantika:

Hlava platí, platí-li jednotlivé literály těla.

Procedurální (imperativní) sémantika:

```
Entry: Hlava::
{
   call T1
   ⋮
   call Ta
}
```

Volání procedury s názvem `Hlava` uspěje, pokud uspěje volání všech procedur (literálů) v těle.

Procedurální sémantika = podklad pro implementaci

Abstraktní interpret

Vstup: Logický program P a dotaz G.

1. Inicializuj množinu cílů S literály z dotazu G; S:=G
2. `while ( S != empty ) do`
3. Vyber A∈S a dále vyber klauzuli A':-B_1,...,B_n $(n \geq 0)$ z programu P takovou, že $\exists\sigma : A\sigma = A'\sigma$; σ je nejobecnější unifikátor.
   Pokud neexistuje A' nebo σ, ukonči cyklus.
4. Nahraď A v S cíli B_1 až B_n.
5. Aplikuj σ na G a S.
6. `end while`
7. Pokud S==empty, pak výpočet úspěšně skončil a výstupem je G se všemi aplikovanými substitucemi.
   Pokud S!=empty, výpočet končí neúspěchem.

Abstraktní interpret – pokračování

Kroky (3) až (5) představují **redukci** (logickou inferenci) cíle A.

Počet redukcí za sekundu (LIPS) == indikátor výkonu implementace

Věta

Existuje-li instance G' dotazu G, odvoditelná z programu P v konečném počtu kroků, pak bude tímto interpretem nalezena.

Nedeterminismus interpetu

1. **Selekční pravidlo:** výběr cíle A z množiny cílů S
   - neovlivňuje výrazně výsledek chování interpretu
2. **Způsob prohledávání stromu výpočtu:** výběr klauzule A' z programu P
   - je velmi důležitý, všechny klauzule totiž nevedou k úspěšnému řešení

Vztah k úplnosti:

1. Selekční pravidlo neovlivňuje úplnost
   - možno zvolit libovolné v rámci SLD rezoluce
2. Prohledávání stromu výpočtu do šířky nebo do hloubky

„Prozření" – automatický výběr správné klauzule

- vlastnost abstraktního interpretu, kterou ale reálné interprety nemají

Prohledávání do šířky

1. Vybereme všechny klauzule A_i', které je možno unifikovat s literálem A
   - nechť je těchto klauzulí q
2. Vytvoříme q kopií množiny S
3. V každé kopii redukujeme A jednou z klauzulí A_i'.
   - aplikujeme příslušný nejobecnější unifikátor
4. V následujících krocích redukujeme všechny množiny S_i současně.
5. Výpočet ukončíme úspěchem, pokud se alespoň jedna z množin S_i stane prázdnou.

- Ekvivalence s abstraktnímu interpretem
  - pokud jeden interpret neuspěje, pak neuspěje i druhý
  - pokud jeden interpret uspěje, pak uspěje i druhý

Prohledávání do hloubky

1. Vybereme všechny klauzule A'$_i$, které je možno unifikovat s literálem A.
2. Všechny tyto klauzule zapíšeme na zásobník.
3. Redukci provedeme s klauzulí na vrcholu zásobníku.
4. Pokud v nějakém kroku nenajdeme vhodnou klauzuli A', vrátíme se k předchozímu stavu (tedy anulujeme aplikace posledního unifikátoru σ) a vybereme ze zásobníku další klauzuli.
5. Pokud je zásobník prázdný, končí výpočet neúspěchem.
6. Pokud naopak zredukujeme všechny literály v S, výpočet končí úspěchem.

- Není úplné, tj. nemusí najít všechna řešení
- Nižší paměťová náročnost než prohledávání do šířky
- Používá se v Prologu

Reprezentace objektů

- Beztypový jazyk
- Kontrola „typů" za běhu výpočtu
- Informace o struktuře součástí objektu

Typy objektů

- **Primitivní objekty:**
  - konstanta
  - číslo
  - volná proměnná
  - odkaz (reference)
- **Složené (strukturované) objekty:**
  - struktura
  - seznam

Reprezentace objektů II

Příznaky (tags):

Objekt	Příznak
volná proměnná	FREE
konstanta	CONST
celé číslo	INT
odkaz	REF
složený term	FUNCT

Obsah adresovatelného slova: **hodnota** a **příznak**.

Primitivní objekty uloženy přímo ve slově

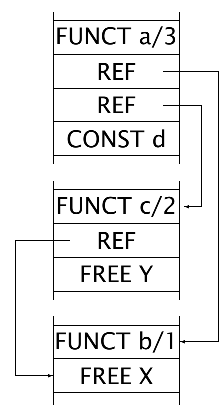
Složené objekty

- jsou instance termu ve zdrojovém textu, tzv. zdrojového termu
- zdrojový term bez proměnných ⟹ každá instanciace ekvivalentní zdrojovému termu
- zdrojový term s proměnnými ⟹ dvě instance se mohou lišit aktuálními hodnotami proměnných, jedinečnost zajišťuje kopírování struktur nebo sdílení struktur

Kopírování struktur

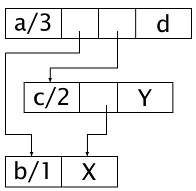
Příklad:

`a(b(X),c(X,Y),d),`



Kopírování struktur II

- Term F s aritou A reprezentován A+1 slovy:
  - funktor a arita v prvním slově
  - 2. slovo nese první argument (resp. odkaz na jeho hodnotu) :
  - A+1 slovo nese hodnotu A-tého argumentu
- Reprezentace vychází z orientovaných acyklických grafů:



- Vykopírována každá instance ⇒ **kopírování struktur**
- Termy ukládány na **globální zásobník**

Sdílení struktur

- Vychází z myšlenky, že při reprezentaci je třeba řešit přítomnost proměnných
- Instance termu

  `< kostra_termu; rámec >`

  - `kostra_termu` je zdrojový term s očíslovanými proměnnými
  - `rámec` je vektor aktuálních hodnot těchto proměnných
    - i-tá položka nese hodnotu i-té proměnné v původním termu

Sdílení struktur II

Příklad:

`a(b(X),c(X,Y),d)`

reprezentuje

`< a(b($1),c($1,$2),d) ; [FREE, FREE] >`

kde symbolem `$i` označujeme i-tou proměnnou.

Implementace:

`< &kostra_termu; &rámec >` (& vrací adresu objektu)

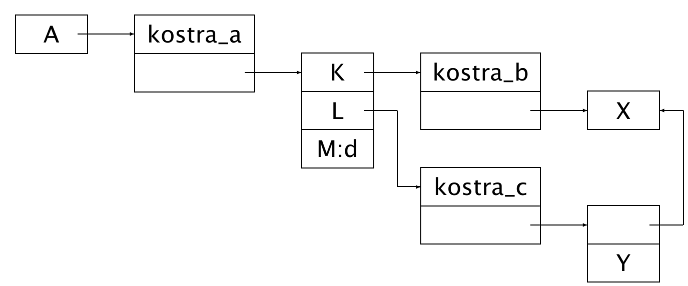
Všechny instance sdílí společnou `kostru_termu` ⇒ **sdílení struktur**

Srovnání: příklad

- Naivní srovnání: sdílení paměťově méně náročné
- Platí ale pouze pro rozsáhlé termy přítomné ve zdrojovém kódu
- Postupná tvorba termů:

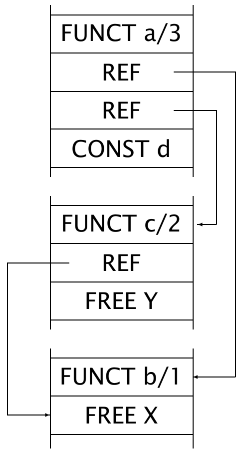
  `A = a(K,L,M), K = b(X), L = c(X,Y), M = d`

  - Sdílení termů:



Srovnání: příklad – pokračování

- Kopírování struktur: `A = a(K,L,M), K = b(X), L = c(X,Y), M = d`



tj. identické jako přímé vytvoření termu `a(b(X),c(X,Y),d)`

Srovnání II

- **Složitost algoritmů pro přístup k jednotlivým argumentům**
  - sdílení struktur: nutná víceúrovňová nepřímá adresace
  - kopírování struktur: bez problémů
  - jednodušší algoritmy usnadňují i optimalizace
- **Lokalita přístupů do paměti**
  - sdílení struktur: přístupy rozptýleny po paměti
  - kopírování struktur: lokalizované přístupy
  - při stránkování paměti – rozptýlení vyžaduje přístup k více stránkám
- Z praktického hlediska neexistuje mezi těmito přístupy zásadní rozdíl

Řízení výpočtu

- **Dopředný výpočet**
  - po úspěchu (úspěšná redukce)
    - jednotlivá volání procedur skončí úspěchem
  - klasické volání rekurzivních procedur
- **Zpětný výpočet (backtracking)**
  - po neúspěchu vyhodnocení literálu (neúspěšná redukce)
    - nepodaří se unifikace aktuálních a formálních parametrů hlavy
  - návrat do bodu, kde zústala nevyzkoušená alternativa výpočtu
    - je nutná obnova původních hodnot jednotlivých proměnných
    - po nalezení místa s dosud nevyzkoušenou klauzulí pokračuje dále dopředný výpočet

Aktivační záznam

- Volání (=aktivace) procedury
- Aktivace **sdílí společný kód**, liší se obsahem **aktivačního záznamu**
- Aktivační záznam uložen na **lokálním zásobníku**
- Dopředný výpočet
  - stav výpočtu v okamžiku volání procedury
  - aktuální parametry
  - lokální proměnné
  - pomocné proměnné (‘a la registry)
- Zpětný výpočet (backtracking)
  - hodnoty parametrů v okamžiku zavolání procedury
  - následující klauzule pro zpracování při neúspěchu

Aktivační záznam a roll-back

- Neúspěšná klauzule mohla nainstanciovat nelokální proměnné
  - `a(X) :- X = b(c,Y), Y = d.` `?- W = b(Z,e), a(W).` (viz instanciace Z)
- Při návratu je třeba obnovit (**roll-back**) původní hodnoty proměnných
- Využijeme vlastností logických proměnných
  - instanciovat lze pouze volnou proměnnou
  - jakmile proměnná získá hodnotu, nelze ji změnit jinak než návratem výpočtu

  $\Longrightarrow$ původní hodnoty všech proměnných odpovídají volné proměnné
- **Stopa** (trail): zásobník s adresami instanciovaných proměnných
  - ukazatel na aktuální vrchol zásobníku uchováván v aktivačním záznamu
  - při neúspěchu jsou hodnoty proměnných na stopě v úseku mezi aktuálním a uloženým vrcholem zásobníku změněny na „volná"
- **Globální zásobník**: pro uložení složených termů
  - ukazatel na aktuální vrchol zásobníku uchováván v aktivačním záznamu
  - při neúspěchu vrchol zásobníku snížen podle uschované hodnoty v aktivačním záznamu

Okolí a bod volby

Aktivační záznam úspěšně ukončené procedury nelze odstranit z lokálního zásobníku $\Longrightarrow$ **rozdělení aktivačního záznamu:**

- **okolí** (environment) – informace nutné pro dopředný běh programu
- **bod volby** (choice point) – informace nezbytné pro zotavení po neúspěchu
- ukládány na lokální zásobník
- samostatně provázány (odkaz na předchozí okolí resp. bod volby)

Důsledky:

- samostatná práce s každou částí aktivačního záznamu (optimalizace)
- alokace pouze okolí pro deterministické procedury
- možnost odstranění okolí po úspěšném vykonání (i nedeterministické) procedury (pokud okolí následuje po bodu volby dané procedury)
  - pokud je okolí na vrcholu zásobníku

Řez

- Prostředek pro ovlivnění běhu výpočtu programátorem
  - ```
    a(X) :- b(X), !, c(X).     a(3).
    b(1).     b(2).
    c(1).     c(2).
    ```
- Řez: neovlivňuje dopředný výpočet, má vliv pouze na zpětný výpočet
- Odstranění alternativních větví výpočtu

  ⇒ odstranění odpovídajících bodů volby
  - tj. odstranění bodů volby mezi současným vrcholem zásobníku a bodem volby procedury, která řez vyvolala (včetně bodu volby procedury s řezem)

  ⇒ změna ukazatele na „nejmladší" bod volby

⇒ Vytváření deterministických procedur

⇒ Optimalizace využití zásobníku

Warrenův abstraktní počítač, WAM I.

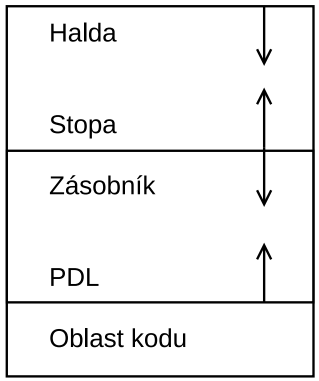
Navržen D.H.D. Warrenem v roce 1983, modifikace do druhé poloviny 80. let

Datové oblasti:

- **Oblast kódu** (programová databáze)
  - separátní oblasti pro uživatelský kód (modifikovatelný) a vestavěné predikátý (nemění se)
  - obsahuje rovněž všechny statické objekty (texty atomů a funktorů apod.)
- **Lokální zásobník *(Stack)***
- **Stopa *(Trail)***
- **Globální zásobník n. halda*(Heap)***
- **Pomocný zásobník *(Push Down List, PDL)***
  - pracovní paměť abstraktního počítače
  - použitý v unifikaci, syntaktické analýze apod.
    ```

Rozmístění datových oblastí

- Příklad konfigurace



- Halda i lokální zásobník musí růst stejným směrem
  - lze jednoduše porovnat stáří dvou proměnných srovnáním adres
    využívá se při zabránění vzniku visících odkazů

Registry WAMu

- **Stavové registry**:
  - **P** čitač adres (**P**rogram counter)
  - **CP** adresa návratu (**C**ontinuation **P**ointer)
  - **E** ukazatel na nejmladší okolí (**E**nvironment)
  - **B** ukazatel na nejmladší bod volby (**B**acktrack point)
  - **TR** vrchol stopy (**TR**ail)
  - **H** vrchol haldy (**H**eap)
  - **HB** vrchol haldy v okamžiku založení posledního bodu volby (**H**eap on **B**acktrack point)
  - **S** ukazatel, používaný při analýze složených termů (**S**tructure pointer)
  - **CUT** ukazatel na bod volby, na který se řezem zařízne zásobník
- **Argumentové registry**: `A1,A2,...` (při předávání parametrů n. pracovní registry)
- **Registry pro lokální proměnné**: `Y1,Y2,...`
  - abstraktní znázornění lok. proměnných na zásobníku

Typy instrukcí WAMu

- **`put` instrukce** – příprava argumentů před voláním podcíle
  - žádná z těchto instrukcí nevolá obecný unifikační algoritmus
- **`get` instrukce** – unifikace aktuálních a formálních parametrů
  - obecná unifikace pouze při `get_value`
- **`unify` instrukce** – zpracování složených termů
  - jednoargumentové instrukce, používají registr S jako druhý argument
  - počáteční hodnota S je odkaz na 1. argument
  - volání instrukce `unify` zvětší hodnotu S o jedničku
  - obecná unifikace pouze při `unify_value` a `unify_local_value`
- **Indexační instrukce** – indexace klauzulí a manipulace s body volby
- **Instrukce řízení běhu** – předávání řízení a explicitní manipulace s okolím

Instrukce put a get: příklad

Příklad: `a(X,Y,Z) :- b(f,X,Y,Z).`

```
get_var     A1,A5
get_var     A2,A6
get_var     A3,A7
put_const   A1,f
put_value   A2,A5
put_value   A3,A6
put_value   A4,A7
execute     b/4
```

Instrukce WAMu

`get` **instrukce**		`put` **instrukce**		`unify` **instrukce**	
`get_var`	`Ai,Y`	`put_var`	`Ai,Y`	`unify_var`	`Y`
`get_value`	`Ai,Y`	`put_value`	`Ai,Y`	`unify_value`	`Y`
`get_const`	`Ai,C`	`put_unsafe_value`	`Ai,Y`	`unify_local_value`	`Y`
`get_nil`	`Ai`	`put_const`	`Ai,C`	`unify_const`	`C`
`get_struct`	`Ai,F/N`	`put_nil`	`Ai`	`unify_nil`	
`get_list`	`Ai`	`put_struct`	`Ai,F/N`	`unify_void`	`N`
		`put_list`	`Ai`		

instrukce řízení		**indexační instrukce**			
`allocate`		`try_me_else`	`Next`	`try`	`Next`
`deallocate`		`retry_me_else`	`Next`	`retry`	`Next`
`call`	`Proc/N,A`	`trust_me_else`	`fail`	`trust`	`fail`
`execute`	`Proc/N`				
`proceed`		`cut_last`		`switch_on_term`	`Var,Const,List,Struct`
		`save_cut Y`		`switch_on_const`	`Table`
		`load_cut Y`		`switch_on_struct`	`Table`

WAM – indexace

- Provázání klauzulí: instrukce `XX_me_else`:
  - první klauzule: `try_me_else`; založí bod volby
  - poslední klauzule: `trust_me_else`; zruší nejmladší bod volby
  - ostatní klauzule: `retry_me_else`; znovu použije nejmladší bod volby po neúspěchu
- Provázání podmnožiny klauzulí (podle argumentu):
  - `try`
  - `retry`
  - `trust`
- „Rozskokové” instrukce (dle typu a hodnoty argumentu):
  - `switch_on_term Var, Const, List, Struct`
    výpočet následuje uvedeným návěstím podle typu prvního argumentu
  - `switch_on_YY`: hashovací tabulka pro konkrétní typ (konstanta, struktura)

Příklad indexace instrukcí

Proceduře

```
a(atom) :- body1.
a(1) :- body2.
a(2) :- body3.
a([X|Y]) :- body4.
a([X|Y]) :- body5.
a(s(N)) :- body6.
a(f(N)) :- body7.
```

odpovídají instrukce

```
a:    switch_on_term L1, L2, L3, L4
L2:   switch_on_const  atom :L1a
                       1    :L5a
                       2    :L6a
L3:   try              L7a
      trust            L8a
L4:   switch_on_struct s/1  :L9a
                       f/1  :L10a
L1:   try_me_else L5
L1a:  body1
L5:   retry_me_else L6
L5a:  body2
L6:   retry_me_else L7
L6a:  body3
L7:   retry_me_else L8
L7a:  body4
L8:   retry_me_else L9
L8a:  body5
L9:   retry_me_else L10
L9a:  body6
L10:  trust_me_else fail
L10a: body7
```


WAM – řízení výpočtu

- `execute Proc`: ekvivalentní příkazu goto
- `proceed`: zpracování faktů
- `allocate`: alokuje okolí (pro některé klauzule netřeba, proto explicitně generováno)
- `deallocate`: uvolní okolí (je-li to možné, tedy leží-li na vrcholu zásobníku)
- `call Proc,N`: zavolá Proc, N udává počet lok. proměnných (odpovídá velikosti zásobníku)
  Možná optimalizace: vhodným uspořádáním proměnných
  lze dosáhnout postupného zkracování lokálního zásobníku

  `a(A,B,C,D) :- b(D), c(A,C), d(B), e(A), f.`

  generujeme instrukce

  ```
  allocate
  call b/1,4
  call c/2,3
  call d/1,2
  call e/1,1
  deallocate
  execute f/0
  ```

WAM – řez

Implementace řezu (opakování): odstranění bodů volby mezi současným vrcholem zásobníku a bodem volby procedury, která řez vyvolala (včetně bodu volby procedury s řezem)

Indexační instrukce znemožňují v době překladu rozhodnout, zda bude alokován bod volby

- příklad: `?- a(X).` může být nedeterministické, ale `?- a(1).` může být deterministické

```
cut_last:   B := CUT          save_cut Y:   Y := CUT          load_cut Y:   B := Y
```

Hodnota registru B je uchovávána v registru CUT instrukcemi `call` a `execute`.
Je-li řez prvním predikátem klauzule, použije se rovnou `cut_last`. V opačném případě se použije jako první instrukce `save_cut Y` a v místě skutečného volání řezu se použije `load_cut Y`.

Příklad:
```
a(X,Z) :- b(X), !, c(Z).
a(2,Z) :- !, c(Z).
a(X,Z) :- d(X,Z).
```
odpovídá

```
save_cut Y2; get A2,Y1; call b/1,2; load_cut Y2; put Y1,A1; execute c/1
get_const A1,2; cut_last; put A2,A1; execute c/1
execute d/2
```


WAM – optimalizace

1. Indexace klauzulí
2. Generování optimální posloupnosti instrukcí WAMu
3. Odstranění redundancí při generování cílového kódu.

- Příklad: `a(X,Y,Z) :- b(f,X,Y,Z).`

  naivní kód (vytvoří kompilátor pracující striktně zleva doprava) vs.
  optimalizovaný kód (počet registrů a tedy i počet instrukcí/přesunů v paměti snížen):

```
get_var     A1,A5          |     get_var     A3,A4
get_var     A2,A6          |     get_var     A2,A3
get_var     A3,A7          |     get_var     A1,A2
put_const   A1,f           |     put_const   A1,f
put_value   A2,A5          |     execute     b/4
put_value   A3,A6          |
put_value   A4,A7          |
execute     b/4            |
```