

IB013 Logické programování

Hana Rudová

jaro 2013

Hodnocení předmětu

- **Průběžná písemná práce:** až 30 bodů (základy programování v Prologu)
 - pro každého jediný termín: **26.března**
 - alternativní termín pouze v případech závažných důvodů pro neúčast
 - vzor písemky na webu předmětu
- **Závěrečná písemná práce:** až 150 bodů
 - vzor písemky na webu předmětu
 - opravný termín možný jako ústní zkouška
- **Zápočtový projekt:** celkem až 40 bodů
- **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
- **Ukončení předmětu zápočtem:** zápočet udělen za zápočtový projekt

Základní informace

- **Přednáška:** účast není povinná, nicméně ...
- **Cvičení:** účast povinná
 - individuální doplňující příklady za zmeškaná cvičení
 - nelze při vysoké neúčasti na cvičení
 - skupina 01, sudá středa, první cvičení **6.března**, Hana Rudová
 - skupina 02, lichá středa, první cvičení **27.února**, Adriana Strejčková
- **Předpoklad:** znalost základů výrokové a predikátové logiky, např. z IB101
- **Web předmětu: interaktivní osnova v ISu**
 - průsvitky dostupné postupně v průběhu semestru
 - sbírka příkladů včetně řešení (zveřejněna cca do 8.3.)
 - harmonogram výuky, předběžný obsah výuky během semestru
 - elektronicky dostupné materiály
 - informace o zápočtových projektech

Rámcový obsah předmětu

Obsah přednášky

- základy programování v jazyce Prolog
- teorie logického programování
- logické programování s omezujícími podmínkami
- DC gramatiky

Obsah cvičení

- zaměřeno na praktické aspekty, u počítačů
- programování v Prologu
 - logické programování
 - logické programování s omezujícími podmínkami

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)

Průběžná písemná práce

- Pro každého jediný termín **26. 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
- Obsah: první čtyři přednášky a první dvě cvičení
- Oblasti, kterých se budou příklady zejména týkat
 - unifikace
 - seznamy
 - backtracking
 - optimalizace posledního volání
 - řez
 - aritmetika
- Ukázka průběžné písemné práce na webu

Zápočtové projekty

- Zadání, dotazy, odevzdávání: **Adriana Strejčková**
- Týmová práce na projektech, až 3 řešitelé
 - zápočtové projekty dostupné přes web předmětu
- Podrobné **pokyny k zápočtovým projektům** na webu předmětu
 - bodování, obsah předběžné zprávy a projektu
 - typ projektu: LP, CLP
- **Předběžná zpráva**
 - podrobné zadání
 - v jakém rozsahu chcete úlohu řešit
 - které vstupní informace bude program používat a co bude výstupem programu
 - scénáře použití programu (tj. ukázky dvojic konkrétních vstupů a výstupů)

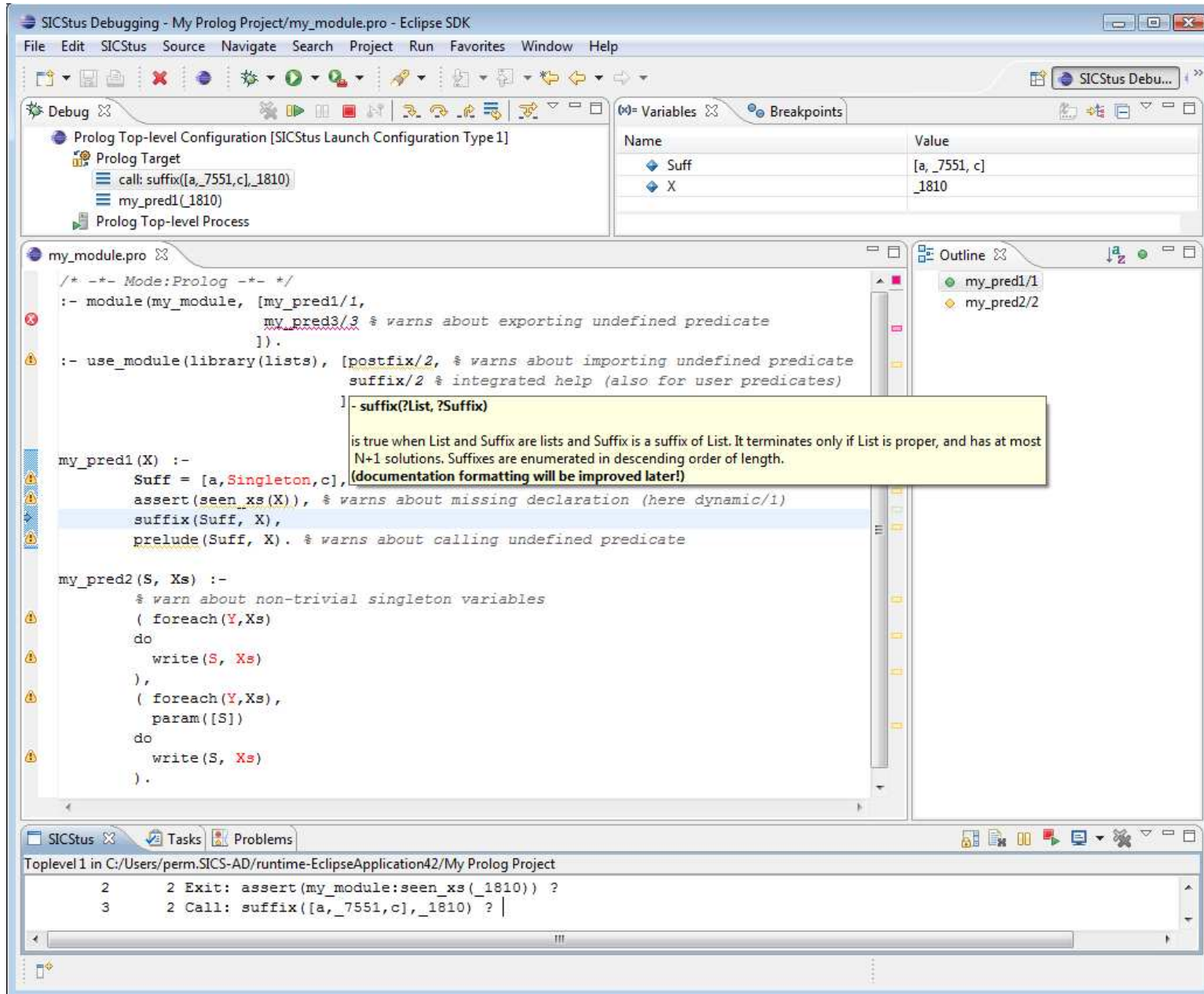
Časový harmonogram k projektům

- Zveřejnění zadání (většiny) projektů: **26. února**
- Zahájení registrace řešitelů projektu: **13. března, 19:00**
- Předběžná analýza řešeného problému: **12. dubna**
- Odevzdání projektů: **17. května**
- Předvádění projektů (po registraci): **23.května – 20.června**

Software: SICStus Prolog

- Doporučovaná implementace Prologu
- Dokumentace: <http://www.fi.muni.cz/~hanka/sicstus/doc/html>
- Komerční produkt
 - licence pro instalace na domácí počítače studentů
- Používané IDE pro SICStus Prolog SPIDER
 - dostupné od verze SICStus 4.1.3
 - <http://www.sics.se/sicstus/spider>
 - používá Eclipse SDK
- Podrobné informace dostupné přes web předmětu
 - stažení SICStus Prologu (sw + licenční klíče)
 - pokyny k instalaci (SICStus Prolog, Eclipse, Spider)

SICStus IDE SPIDER



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

Logické programování

Historie

- Rozvoj začíná po roce 1970
- Robert Kowalski – teoretické základy
- Alain Colmerauer, David Warren (*Warren Abstract Machine*) – implementace
- SICStus Prolog vyvíjen od roku 1985
- Logické programování s omezujícími podmínkami – od poloviny 80. let

Aplikace

- rozpoznávání řeči, telekomunikace, biotechnologie, logistika, plánování, data mining, business rules, ...
- SICStus Prolog — the first 25 years, Mats Carlsson, Per Mildner. *Theory and Practice of Logic Programming*, 12 (1-2): 35-66, 2012. <http://arxiv.org/abs/1011.5640>.

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:** seznam 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

- pozitivní – **dotaz je splnitelný a úspěš**
- negativní – **dotaz je nesplnitelný a neúspěš**

- Proměnné jsou během výpočtu **instanciovány** (= nahrazeny objekty)

- ?- clovek(novak, 18, Prace).

Prace = student

- 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 skládá 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

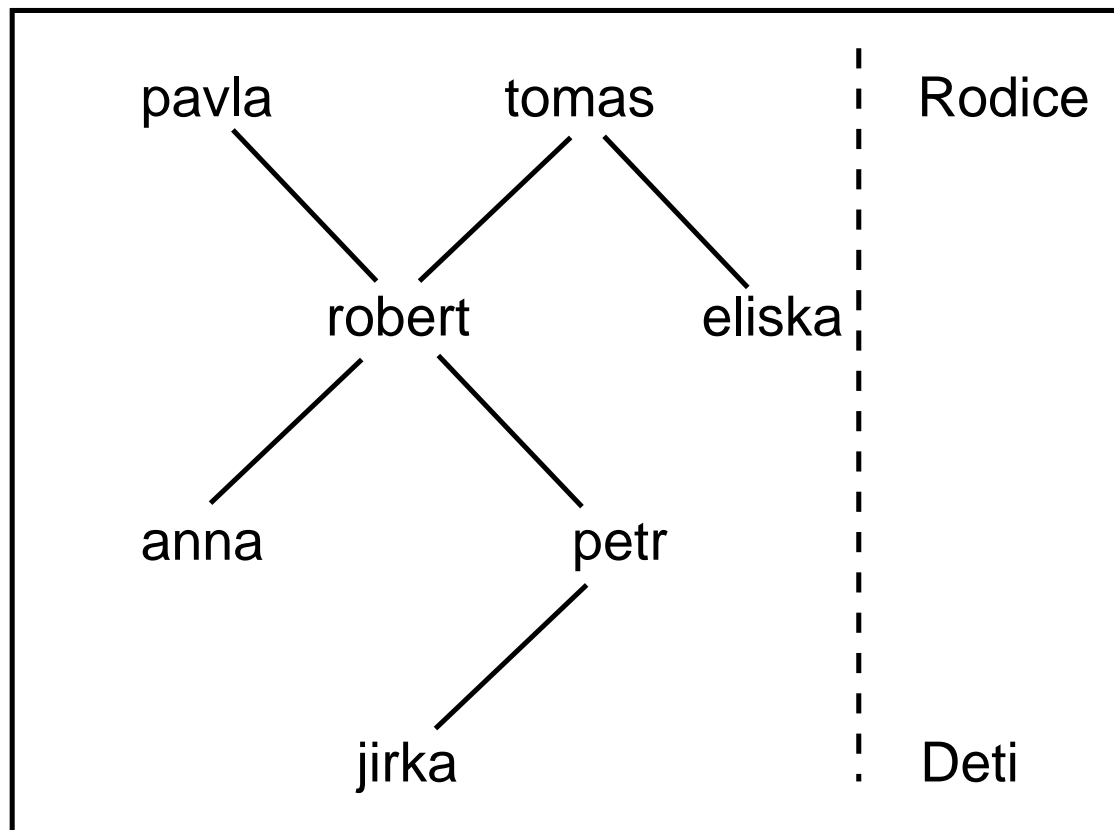
predek(X, Z) :- rodic(X, Z). % (1)

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

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

Příklad: rodokmen

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

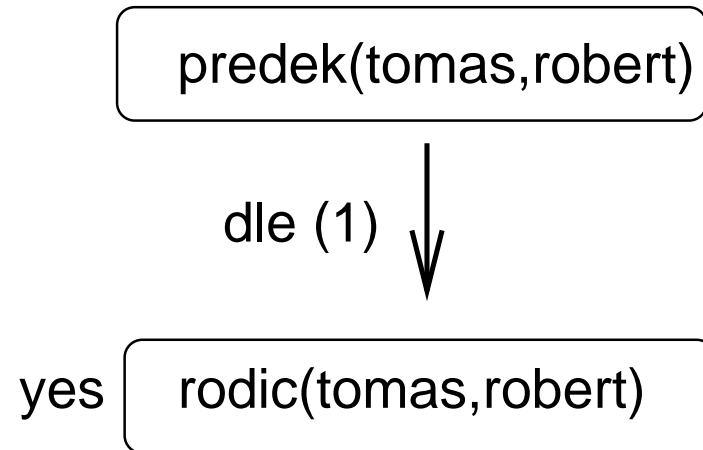


```
predek( X, Z ) :- rodic( X, Z ).           % (1)
```

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

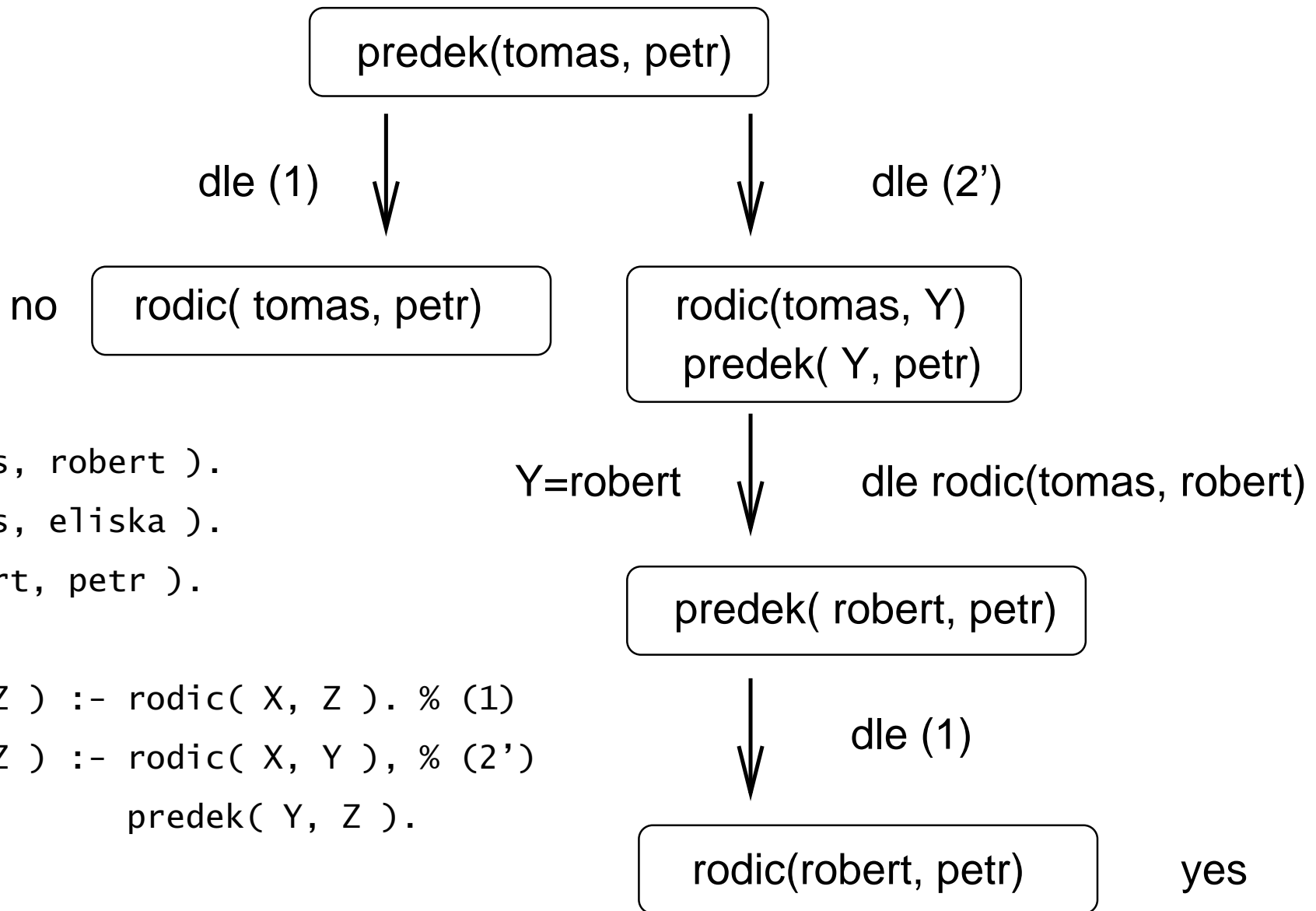
Výpočet odpovědi na dotaz ?- predek(tomas,robert)

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



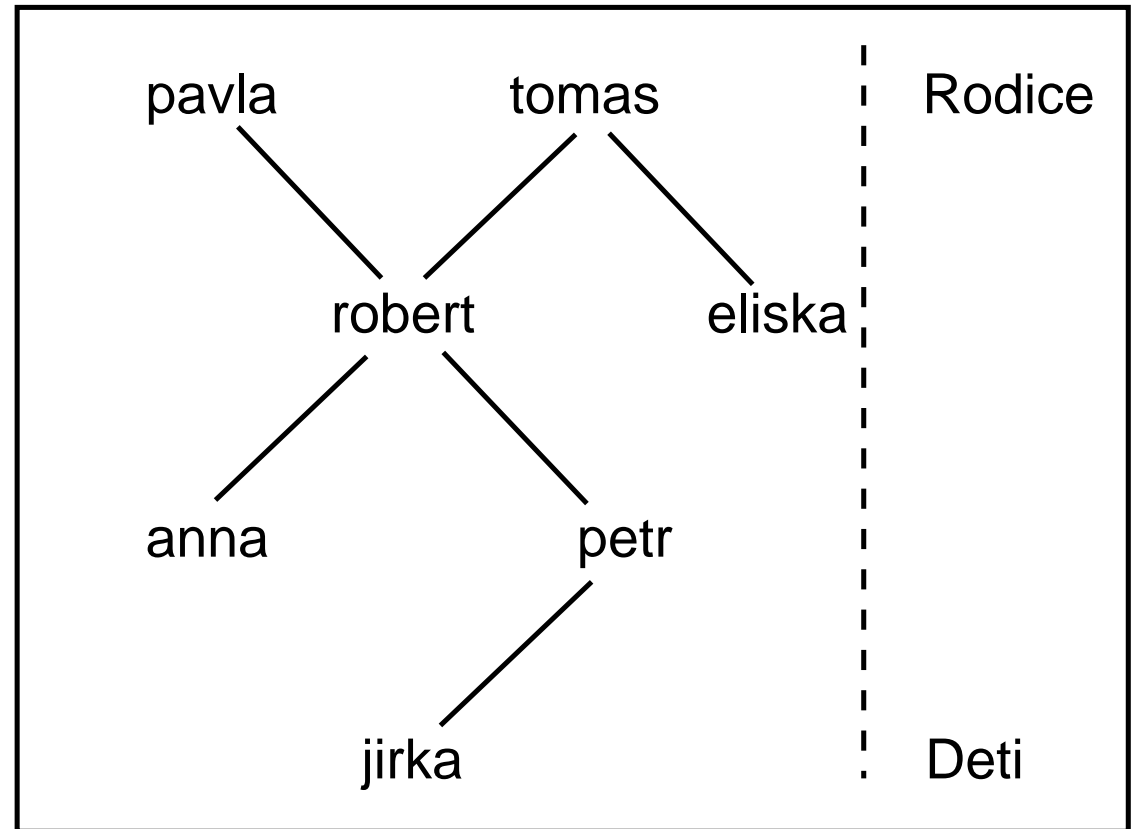
```
predek( X, Z ) :- rodic( X, Z ).           % (1)  
predek( X, Z ) :- rodic( X, Y ),         % (2')  
                    predek( Y, Z ).
```

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



Odpořěd' na dotaz s proměnnou

```
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(petr, Potomek) --> ???

Potomek=jirka

predek(robert, P) --> ???

1. P=anna, 2. P=petr, 3. P=jirka

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**
 - trojuheľnik(bod(4,2), bod(6,4), bod(7,1))
- **Hlavní funktor** termu – funktor v kořenu stromu odpovídající termu
 - trojuheľnik je hlavní funktor v trojuheľnik(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
 - $\text{datum}(D1, M1, 2003) = \text{datum}(1, M2, Y2)$ **operátor =**
 $D1 = 1, M1 = M2, Y2 = 2003$
- Hledáme **nejobecnější unifikátor** (*most general unifier (MGU)*)
 - jiné instanciaci? ... $D1 = 1, M1 = 5, Y2 = 2003$... není MGU
 - ?- $\text{datum}(D1, M1, 2003) = \text{datum}(1, M2, Y2), D1 = M1.$
- **Test výskytu** (*occurs check*)
 - ?- $X=f(X).$
 - $X = f(f(f(f(f(f(f(f(f(\dots))))))))))$

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:

$k = k \dots \text{yes}$, $k1 = k2 \dots \text{no}$, $A = k(2,3) \dots \text{yes}$, $k(s,a,l(1)) = A \dots \text{yes}$

$s(sss(2),B,ss(2)) = s(sss(2),4,ss(2),s(1)) \dots \text{no}$

$s(sss(A),4,ss(3)) = s(sss(2),4,ss(A)) \dots \text{no}$

$s(sss(A),4,ss(C)) = s(sss(t(B)),4,ss(A)) \dots A=t(B),C=t(B) \dots \text{yes}$

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

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 (viz ekvivalentní zápisy):

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

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

- $p :- q, r.$

- $p :- s, t, u.$

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) $c(1).$

?- $a(X).$

$a(X)$

dle (1) |

$c(Y), b(X,Y)$

dle (3) / $Y=2$ dle (4) / $Y=1$

$b(X,2)$

$b(X,1)$

no

dle (2) | $X=1$

yes

Vyzkoušejte si:

(1) $a(X) :- b(X,X), c(X).$

(3) $a(X) :- b(Y,X), c(X).$

(4) $b(2,2).$

(5) $b(2,1).$

(6) $c(1).$

Cvičení: 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.

Operátory, aritmetika

Operátory

- Infixová notace: $2*a + b*c$
- Prefixová notace: $+(*(2, a), *(b, c))$ priorita +: 500, priorita *: 400
 - prefixovou notaci lze získat predikátem `display/1`
 $:- display((a:-s(0), b, c)).$ $:- (a, , (s(0), , (b, c)))$
- **Priorita operátorů:** operátor s **nejvyšší** prioritou je hlavní funktor
- Uživatelsky definované operátory: zna
 $petr\ zna\ a\ lese.$ $zna(petr, a\ lese).$
- 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 spec. případů)

Typy operátorů

■ Typy operátorů

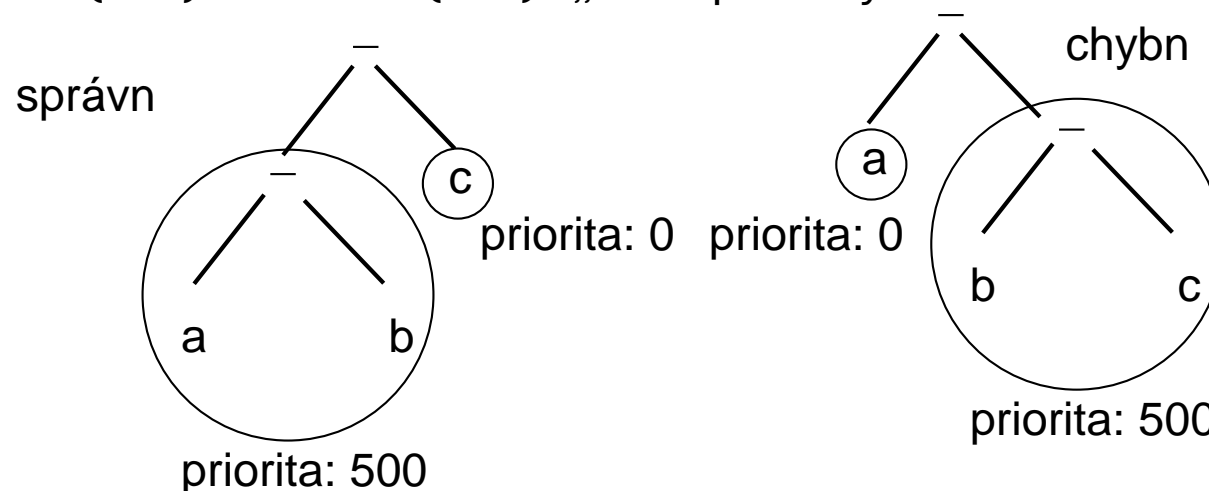
- infixové operátory: xfx , xfy , yfx
- prefixové operátory: fx , fy
- postfixové operátory: xf , yf

př. $xfx = yfx -$

př. $fx ?- fy -$

■ 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

- $?- X \text{ is } 1 + 2.$

$X = 3$ „is“ je speciální předdefinovaný operátor, který vynutí evaluaci

- porovnej: $N = (1+1+1+1+1)$ $N \text{ is } (1+1+1+1+1)$

- pravá strana musí být vyhodnotitelný výraz (bez proměnné)

- výraz na pravé straně je nejdříve aritmeticky vyhodnocen a pak unifikován s levou stranou
volání $?- X \text{ is } Y + 1.$ způsobí chybu

- Další speciální předdefinované operátory

$>$, $<$, $>=$, $=<$, $:=$ aritmetická rovnost, $=\backslash=$ 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
$X \neq Y$	X a Y nejsou unifikovatelné, (také $\neq X = Y$)
$X == Y$	X a Y jsou identické porovnej: $?- A == B. \dots$ no $?- A=B, A==B. \dots B = A$ yes
$X \neq Y$	X a Y nejsou identické porovnej: $?- A \neq B. \dots$ yes $?- A=B, A \neq B. \dots A$ no
$X is Y$	Y je aritmeticky vyhodnoceno a výsledek je přiřazen X
$X ::= Y$	X a Y jsou si aritmeticky rovny
$X \neq Y$	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 $?- f(pavel, g(b)) @< f(pavel, h(a)). \dots$ yes

Řez, negace

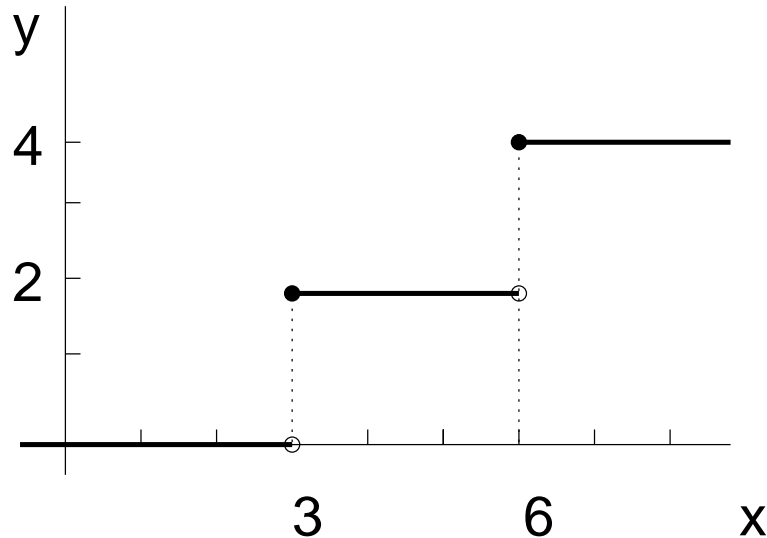
Řez a upnutí

$f(X,0) :- X < 3, !.$

$f(X,2) :- 3 \leq X, X < 6, !.$

$f(X,4) :- 6 \leq X.$

přidání **operátoru řezu** `,,!’’`



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

$f(X,0) :- X < 3, !. \quad \%(1)$

$f(X,2) :- X < 6, !. \quad \%(2)$

$f(X,4).$

$?- f(1,Y).$

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

Řez a ořezání

$f(X,Y) \text{ :- } s(X,Y).$

$s(X,Y) \text{ :- } Y \text{ is } X + 1.$

$s(X,Y) \text{ :- } Y \text{ is } X + 2.$

$?- f(1,Z).$

$Z = 2 ? ;$

$Z = 3 ? ;$

no

$f(X,Y) \text{ :- } s(X,Y), !.$

$s(X,Y) \text{ :- } Y \text{ is } X + 1.$

$s(X,Y) \text{ :- } Y \text{ is } X + 2.$

$?- f(1,Z).$

$Z = 2 ? ;$

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 $H :- T_1, T_2, \dots, T_m, !, \dots, T_n.$ je aktivována voláním cíle G , který je unifikovatelný s H . $G=h(X,Y)$
- V momentě, kdy je nalezen řez, existuje řešení cílů T_1, \dots, T_m $X=1, Y=1$
- **Ořezání:** při provádění řezu se už další možné splnění cílů T_1, \dots, T_m nehledá a všechny ostatní alternativy jsou odstraněny $Y=2$
- **Upnutí:** dále už nevyvolávám další klauzule, jejichž hlava je také unifikovatelná s G $X=2$

$?- h(X,Y).$ $h(1,Y) :- t1(Y), !.$ $h(2,Y) :- a.$ $t1(1) :- b.$ $t1(2) :- c.$	$h(X,Y)$ $X=1 \quad / \quad \backslash \quad X=2$ $t1(Y) \quad a \quad (\text{vynechej: upnutí})$ $Y=1 \quad / \quad \backslash \quad Y=2$ $b \quad c \quad (\text{vynechej: ořezání})$ $/$
---	--

Řez: návrat na rodiče

?- a(X).

(1) a(X) :- h(X,Y).

(2) a(X) :- d.

(3) h(1,Y) :- t1(Y), !, e(X).

(4) h(2,Y) :- a.

(5) t1(1) :- b.

(6) t1(2) :- c.

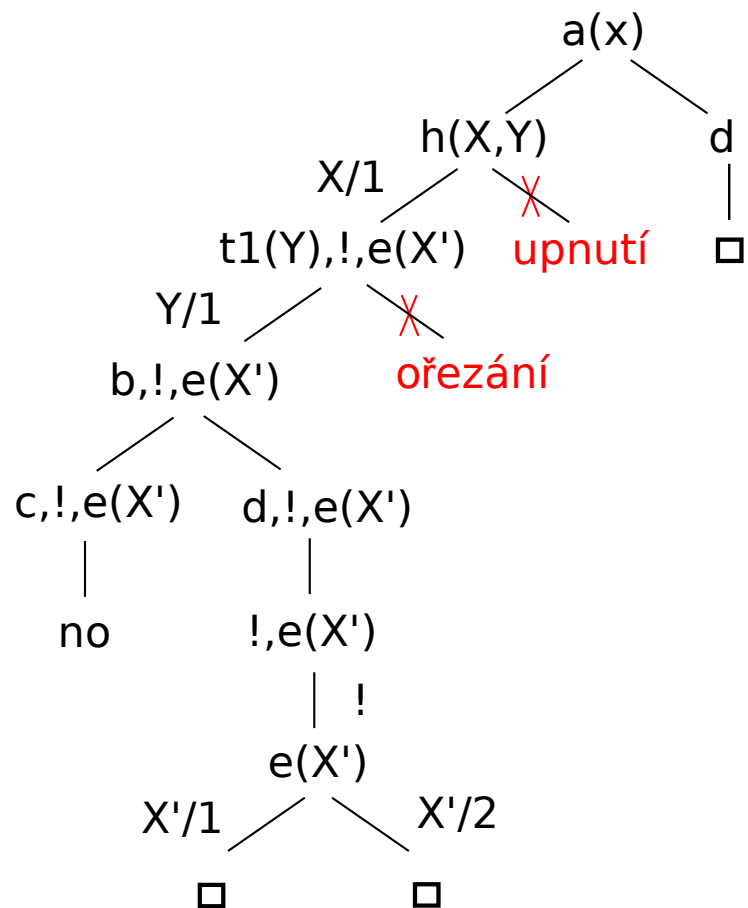
(7) b :- c.

(8) b :- d.

(9) d.

(10) e(1) .

(11) e(2) .



- Po zpracování klauzule s řezem se vracím až na rodiče této klauzule, tj. a(X)

Řez: příklad

```
c(X) :- p(X).
```

```
c(X) :- v(X).
```

```
p(1). p(2).
```

```
v(2).
```

```
?- c(2).
```

```
true ? ; %p(2)
```

```
true ? ; %v(2)
```

```
no
```

```
?- c(X).
```

```
X = 1 ? ; %p(1)
```

```
X = 2 ? ; %p(2)
```

```
X = 2 ? ; %v(2)
```

```
no
```

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

```
c1(X) :- v(X).
```

```
?- c1(2).
```

```
true ? ; %p(2)
```

```
no
```

```
?- c1(X).
```

```
X = 1 ? ; %p(1)
```

```
no
```

Řez: cvičení

1. Porovnejte chování uvedených programů pro zadané dotazy.

$a(X,X) :- b(X).$	$a(X,X) :- b(X), !.$	$a(X,X) :- b(X), c.$
$a(X,Y) :- Y \text{ is } X+1.$	$a(X,Y) :- Y \text{ is } X+1.$	$a(X,Y) :- Y \text{ is } X+1.$
$b(X) :- X > 10.$	$b(X) :- X > 10.$	$b(X) :- X > 10.$
		$c :- !.$

?- a(X,Y).

?- a(1,Y).

?- a(11,Y).

2. Napište predikát pro výpočet maxima $\text{max}(X, Y, \text{Max})$

Typy řezu

- Zlepšení efektivity programu: určíme, které alternativy nemá smysl zkoušet

Poznámka: na vstupu pro X očekávám číslo

- **Zelený řez:** odstraní pouze neúspěšná odvození

- $f(X,1) :- X \geq 0, !. \quad f(X,-1) :- X < 0.$

bez řezu zkouším pro nezáporná čísla 2. klauzuli

- **Modrý řez:** odstraní redundantní řešení

- $f(X,1) :- X \geq 0, !. \quad f(0,1). \quad f(X,-1) :- X < 0.$ bez řezu vrací $f(0,1)$ 2x

- **Červený řez:** odstraní úspěšná řešení

- $f(X,1) :- X \geq 0, !. \quad f(_X,-1).$ 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 $\backslash+$

- $\text{different}(X,Y) :- X = Y, !, \text{fail}.$ $\text{muz}(X) :- \text{zena}(X), !, \text{fail}.$
 $\text{different}(_X,_Y).$ $\text{muz}(_X).$
- Unární operátor $\backslash+$ P
 - jestliže P uspěje, potom $\backslash+$ P neuspěje
 $\backslash+(P) :- P, !, \text{fail}.$
 - v opačném případě $\backslash+$ P uspěje
 $\backslash+(_).$
- $\text{different}(X, Y) :- \backslash+ X=Y.$
- $\text{muz}(X) :- \backslash+ \text{zena}(X).$
- Pozor: takto definovaná negace $\backslash+P$ 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).`

`dobre(X),rozumne(X)`

| `dle (1), X/citroen`

`rozumne(citroen)`

| `dle (4)`

`\+ drahe(citroen)` | `dle (II)`

| `dle (I)`

`drahe(citroen),!, fail`



`yes`

| `no`

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).`

`rozumne(X), dobre(X)`

|
dle (4)

`\+ drahe(X), dobre(X)`

|
dle (1)

`drahe(X),!,fail,dobre(X)`

|
dle (3), X/bmw

`!, fail, dobre(bmw)`

|
`fail,dobre(bmw)`

|
no

Bezpečný cíl

- ?- \+ drahe(citroen). yes
- ?- \+ drahe(X). no
- ?- rozumne(citroen). yes
- ?- rozumne(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
- ?- drahe(X).
PTÁME SE: existuje X takové, že drahe(X) platí?
- ?- \+ 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)**
- TEDY: pro cíle s negací neplatí **existuje** X takové, že \+ drahe(X)
⇒ **negace jako neúspěch není ekvivalentní negaci v matematické logice**
- Negace jako neúspěch používá **předpoklad uzavřeného světa**
pravdivé je pouze to, co je dokazatelné

Predikáty na řízení běhu programu I.

- řez „!”

- fail: cíl, který vždy neuspěje true: cíl, který vždy uspěje

- $\backslash+$ P: negace jako neúspěch

$\backslash+$ P :- P, !, fail; true.

- once(P): vrátí pouze jedno řešení cíle P

once(P) :- P, !.

- Vyjádření **podmínky**: $P \rightarrow Q ; R$

- jestliže platí P tak Q $(P \rightarrow Q ; R) :- P, !, Q.$

- v opačném případě R $(P \rightarrow Q ; R) :- R.$

- příklad: $\text{min}(X, Y, Z) :- X \leq Y \rightarrow Z = X ; Z = Y.$

- $P \rightarrow Q$

- odpovídá: $(P \rightarrow Q; \text{fail})$

- příklad: $\text{zaporne}(X) :- \text{number}(X) \rightarrow X < 0.$

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`, proved' ji a otestuj, zda neskončit**

```
Hlava :- ...
```

```
    uloz_stav( StaryStav ),
```

```
    repeat,
```

```
        generuj( X ),           % deterministické: generuj, provadej, testuj
```

```
        provadej( X ),
```

```
        testuj( X ),
```

```
    !,
```

```
    obnov_stav( StaryStav ),
```

```
    ...
```

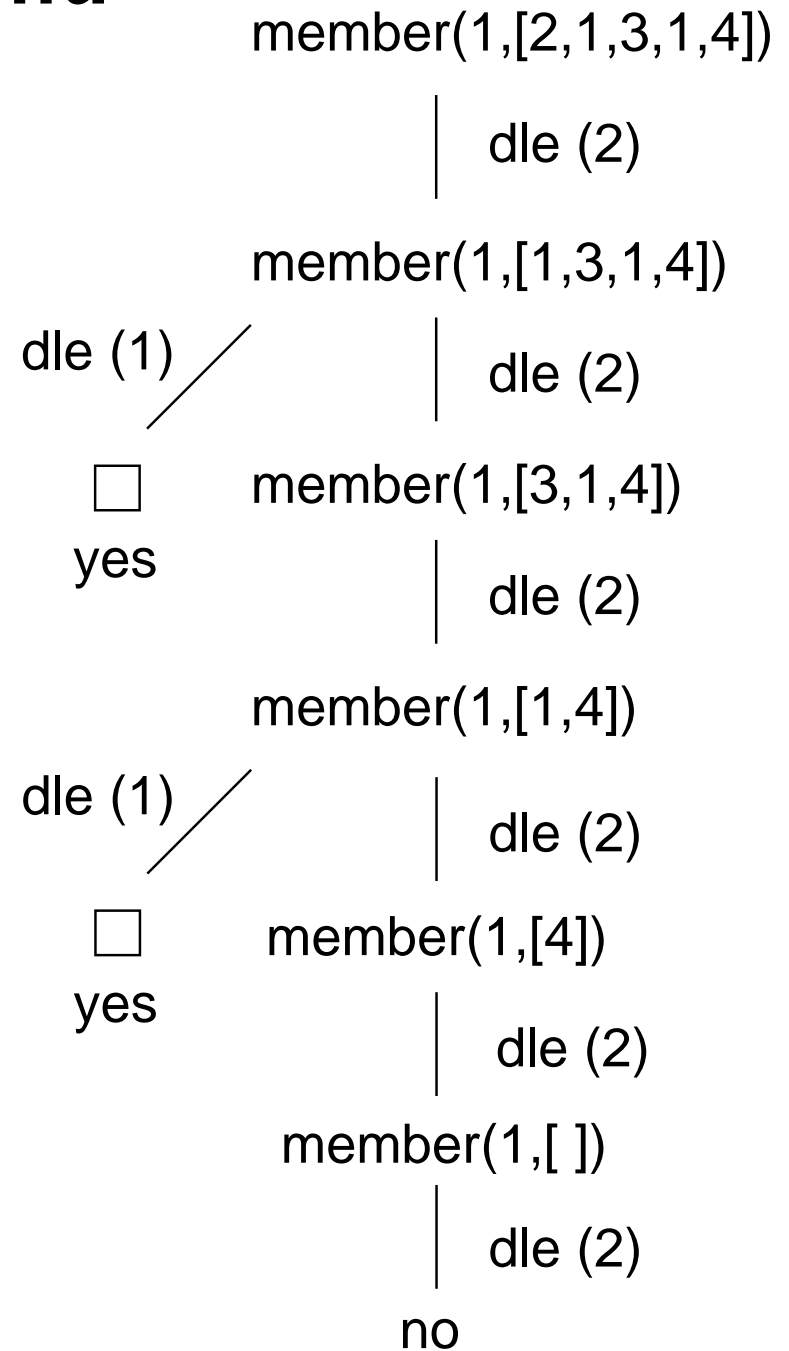
Seznamy

Reprezentace seznamu

- **Seznam:** $[a, b, c]$, prázdný seznam $[]$
- **Hlava (libovolný objekt), tělo (seznam):** $.(Hlava, TeĽo)$
 - všechny strukturované objekty stromy – i seznamy
 - funktor ".", dva argumenty
 - $.(a, .(b, .(c, []))) = [a, b, c]$
 - notace: $[Hlava | TeĽo] = [a|TeĽo]$
TeĽo je v $[a|TeĽo]$ seznam, tedy píšeme $[a, b, c] = [a|[b, c]]$
- Lze psát i: $[a,b|TeĽo]$
 - před "|" je libovolný počet prvků seznamu, za "|" je seznam zbývajících prvků
 - $[a,b,c] = [a|[b,c]] = [a,b|[c]] = [a,b,c|[]]$
 - pozor: $[[a,b] | [c]] \neq [a,b | [c]]$
- Seznam jako **neúplná datová struktura:** $[a,b,c|T]$
 - Seznam = $[a,b,c|T]$, $T = [d,e|S]$, Seznam = $[a,b,c,d,e|S]$

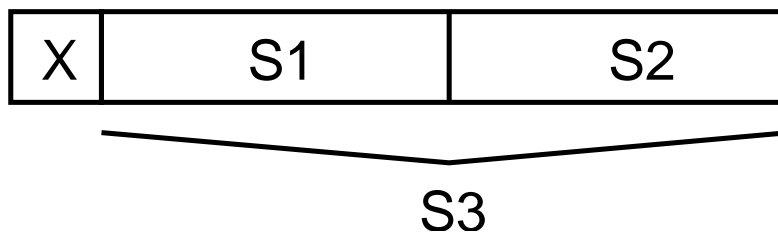
Prvek seznamu

- `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)
- Příklady použití:
 - `member(1, [2,1,3])`.
 - `member(X, [1,2,3])`.



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)`.



Cvičení: append/2

```
append( [], S, S ).      % (1)
```

```
append( [X|S1], S2, [X|S3] ) :- append( S1, S2, S3).  % (2)
```

```
:- append([1,2],[3,4],A).
```

```
  | (2)
```

```
  | A=[1|B]
```

```
  |
```

```
:- append([2],[3,4],B).
```

```
  | (2)
```

```
  | B=[2|C] => A=[1,2|C]
```

```
  |
```

```
:- append([], [3,4],C).
```

```
  | (1)
```

```
  | C=[3,4] => A=[1,2,3,4],
```

```
  |
```

```
yes
```

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 zajišťuje determinismus
```
- Tento typ rekurze lze převést na iteraci

# LCO a akumulátor

- Reformulace rekurzivní procedury, aby umožnila LCO
- Výpočet délky seznamu `length( Seznam, Deřka )`

```
length([], 0).
```

```
length([H | T], Deřka) :- length(T, Deřka0), Deřka is 1 + Deřka0.
```

- Upravená procedura, tak aby umožnila LCO:

```
% length(Seznam, ZapocitanaDeřka, CelkovaDeřka):
```

```
% CelkovaDeřka = ZapocitanaDeřka + ,,počet prvků v Seznam''
```

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

```
length([], Deřka, Deřka). % celková délka = započítaná délka
```

```
length([H | T], A, Deřka) :- A0 is A + 1, length(T, A0, Deřka).
```

- 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 poradí`  
`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`)

# reverse/2: cvičení

```
reverse(Seznam, OpacnySeznam) :- % (1)
 reverse(Seznam, [], OpacnySeznam).
```

```
reverse([], S, S). % (2)
```

```
reverse([H | T], A, Opacny) :- % (3)
 reverse(T, [H | A], Opacny).
```

---

? - reverse([1,2,3],0).

reverse([1,2,3],0) → (1)

reverse([1,2,3], [], 0) → (3)

reverse([2,3], [1], 0) → (3)

reverse([3], [2,1], 0) → (3)

reverse([], [3,2,1], 0) → (2)

yes      0=[3,2,1]

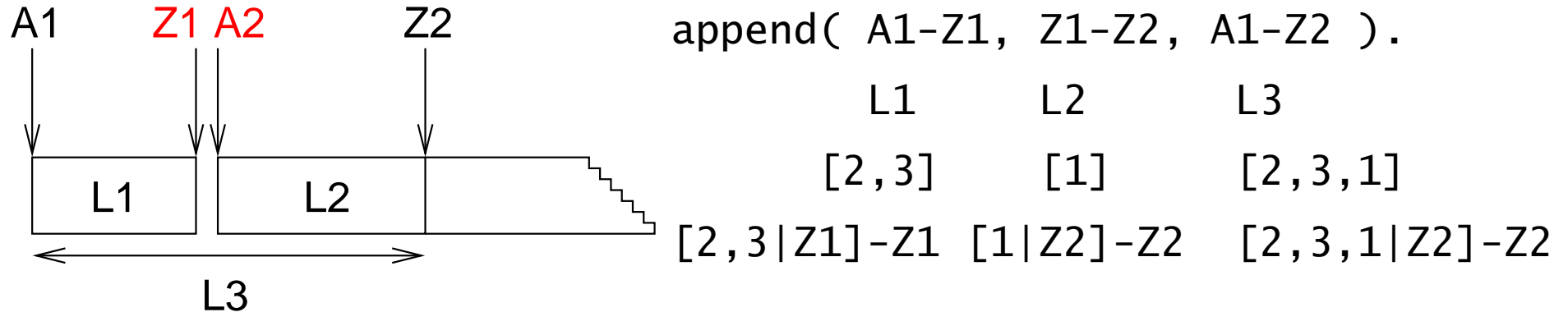
# Neefektivita při spojování seznamů

- Sjednocení dvou seznamů
- $\text{append}( [], S, S )$ .  
 $\text{append}( [X|S1], S2, [X|S3] ) \text{ :- append}( S1, S2, S3 )$ .
- $?\text{- append}( [2,3], [1], S )$ .  
postupné volání cílů:  
 $\text{append}( [2,3], [1], S ) \rightarrow \text{append}( [3], [1], S' ) \rightarrow \text{append}( [], [1], S'' )$
- 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( [2,3|Z1]-Z1, [1|Z2]-Z2, S )`.  
 $S = A1 - Z2 = [2,3|Z1] - Z2 = [2,3| [1|Z2] ] - Z2$   
 $Z1 = [1|Z2] \quad S = [2,3,1|Z2]-Z2$
- 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) :-
```

```
 reverse0(T, Opacny-[H | OpacnyKonec]).
```

rozdílové seznamy

(lineární)

Příklad: operace pro manipulaci s frontou

- test na prázdnotu, 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

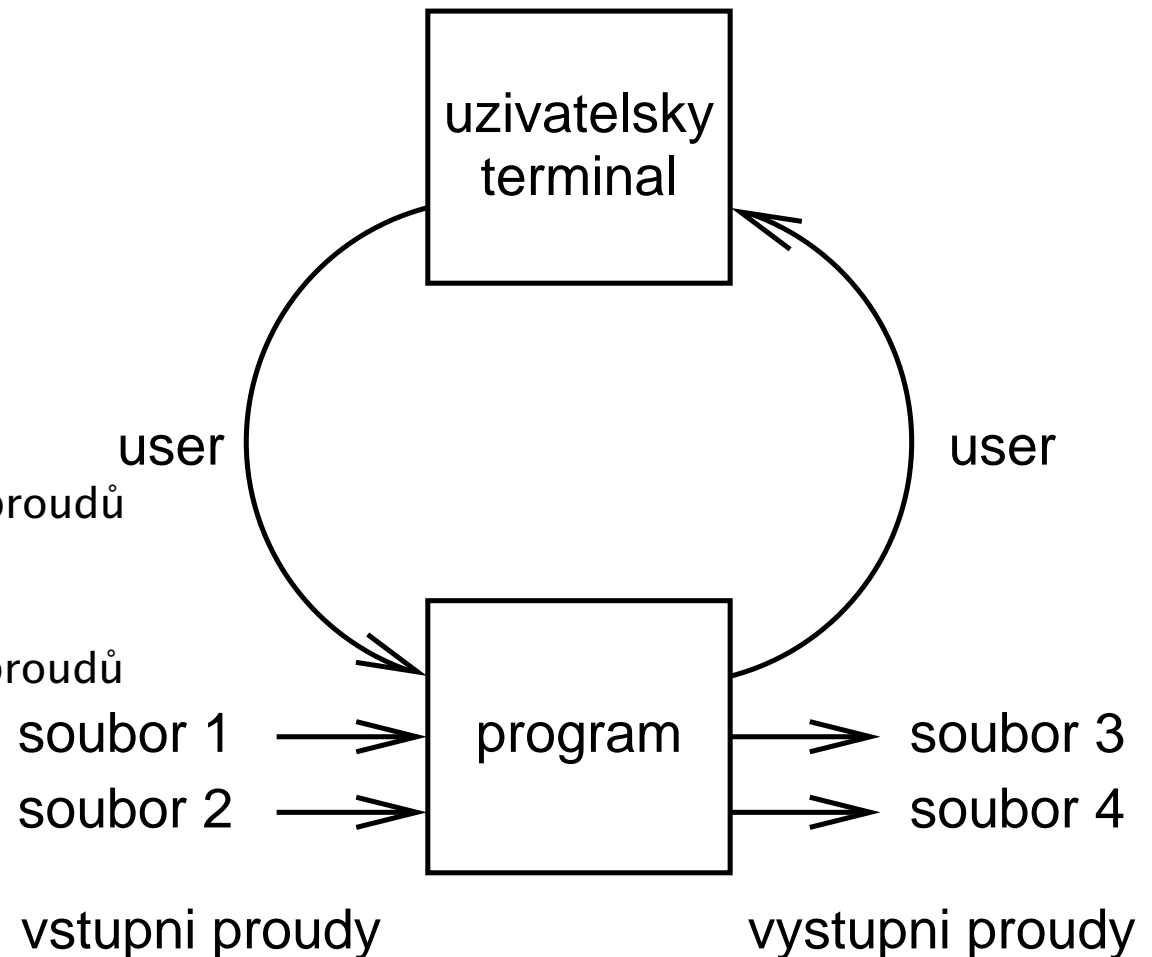
- **Caching**: odpovědi na dotazy jsou přidány do programové databáze
  - `?- solve( problem, Solution),  
asserta( solve( problem, Solution) ).`
  - `:- dynamic solve/2. % nezbytné při použití v SICStus Prologu`

- 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**



# 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: `n1`

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), % manipule 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'] ).`
- **Kompilace** kódu programu
  - ?- `compile( [program1, 'program2.pl'] ).`
  - ?- `[program].`
  - ?- `[user].`      **zadávání kódu ze vstupu** ukončené CTRL+D
  - 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 ]

# Všetchna 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: `X` je term

```
?- bagof(Dite-Vek, vek(Dite, Vek), Seznam).
```

```
Seznam = [petr-7,anna-5,tomas-5]
```

# Existenční kvantifikátor „ $\exists$ ”

- Přidání **existenčního kvantifikátoru** „ $\exists$ ”  $\Rightarrow$  hodnota proměnné nemá význam

?- bagof( Dite, Vek $\exists$ vek( 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 „ $\exists$ ” může být i seznam

?- bagof( Vek ,[Jmeno,Prijmeni] $\exists$ vek( Jmeno, Prijmeni, Vek ), Seznam ).

Seznam = [7,5,5]

# Všechna řešení III.

- `setof( X, P, S )`: rozdíl od `bagof`
  - `S` je uspořádaný podle `@<`
  - případné duplicity v `S` jsou eliminovány
- `findall( X, P, S )`: rozdíl 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

|                          |                                                                                         |
|--------------------------|-----------------------------------------------------------------------------------------|
| <code>var(X)</code>      | X je volná proměnná                                                                     |
| <code>nonvar(X)</code>   | X není proměnná                                                                         |
| <code>atom(X)</code>     | X je atom ( <code>pavel</code> , <code>'Pavel Novák'</code> , <code>&lt;--&gt;</code> ) |
| <code>integer(X)</code>  | X je integer                                                                            |
| <code>float(X)</code>    | X je float                                                                              |
| <code>atomic(X)</code>   | X je atom nebo číslo                                                                    |
| <code>compound(X)</code> | 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 = [80,97,118,101,108]`

?- `A="ano".`

`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]

Call =.. [ Funktor | SeznamArgumentu ], call( Call )

atom =.. X  $\Rightarrow$  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`)  $\Rightarrow$  konec rozkladu
- Term je seznam (`[_|_]`)  $\Rightarrow$  [] ... řešen výše jako `atomic`  
procházení seznamu a rozklad každého prvku seznamu
- Term je složený (`=../2`, `functor/3`)  $\Rightarrow$   
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)).
```

```
no
```

```
?- ground(s(2,[a(1,3),b,c])).
```

```
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` jsou termy 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 jména 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 klauzule se středníkem do více klauzulí
- Opatrné používání **operátoru řezu**
  - preferovat použití zeleného řezu (nemění deklarativní sémantiku)
  - červený řez používat v jasně definovaných konstruktech

negace:  $P, !, \text{fail}; \text{true}$   $\backslash + P$

alternativy:  $\text{Podmínka}, !, C_i11 ; C_i12$   $\text{Podmínka} \rightarrow C_i11 ; C_i12$
- Opatrné používání **negace „\+”**
  - negace jako neúspěch: negace není ekvivalentní negaci v matematické logice
- Pozor na **assert** a **retract**: snižují transparentnost chování programu

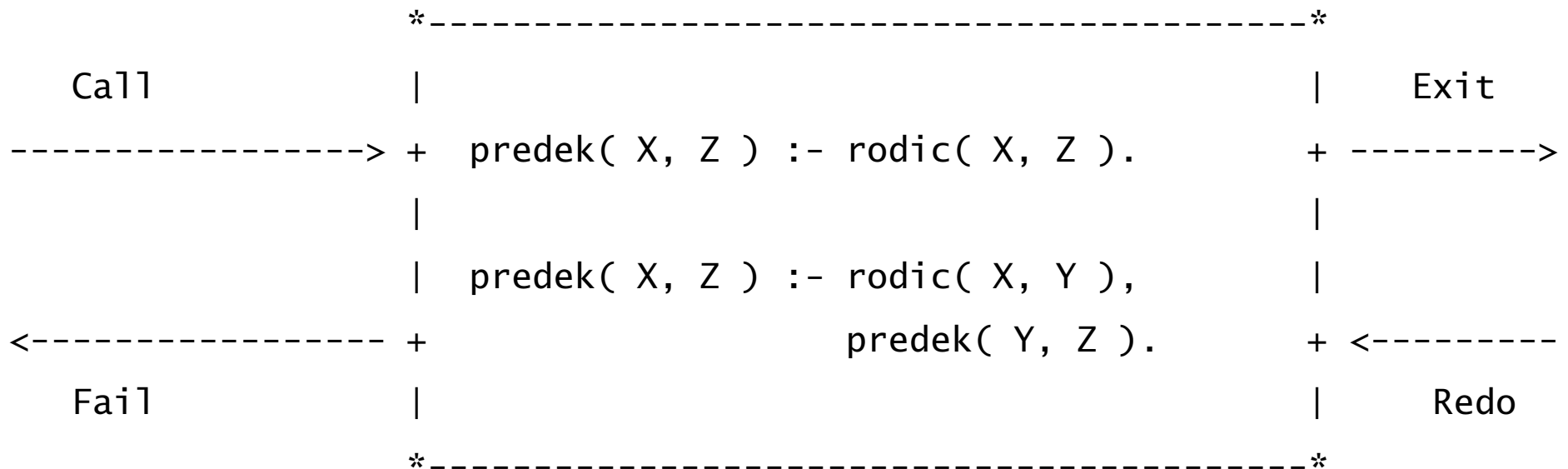


# Ladění

- 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 řídicího 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 našel alternativy k předchozímu řešení



# 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, nenumernické 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 nainstancovaným argumentem se používá hašovací tabulka zpřístupňující pouze odpovídající klauzule
  - `zamestnanec( Prijmeni, KrestniJmeno, Oddezeni, ...)`
  - `seznamy( [], ... ) :- ... .`  
`seznamy( [H|T], ... ) :- ... .`
- **Determinismus:**
  - rozhodnout, které klauzule mají uspět vícekrát, ověřit požadovaný determinismus

# Rezoluce v predikátové logice 1.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 nespílitelná  
 $\implies$  formule je vždy pravdivá

# Formule

- **literál**  $l$ 
  - **pozitivní literál** = atomická formule  $p(t_1, \dots, t_n)$
  - **negativní literál** = negace atomické formule  $\neg p(t_1, \dots, t_n)$ 
    - $t_1, \dots, t_n$  jsou termy
- **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**  $\iff$  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. konjunktivní 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**  $\iff$  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  $C_1 \cup C_2$

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

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

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

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  $C_1, \dots, C_n = C$  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

- důkaz pravdivosti 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  $\square$**

- Příklad:

$$F \dots \neg a \vee a$$

$$G = \neg F \dots a \wedge \neg a$$

$$G = \neg F \dots \{\{a\}, \{\neg a\}\}$$

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

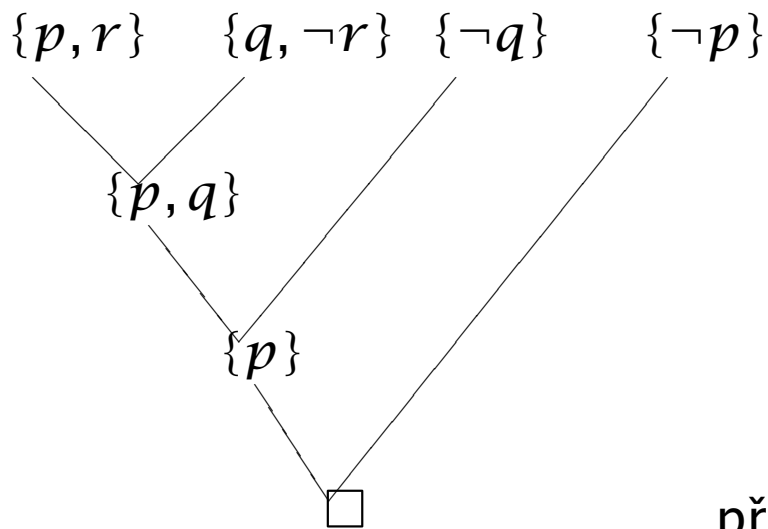
rezolventa  $C_1$  a  $C_2$  je  $\square$ , tj.  $F$  je vždy pravdivá

- rezoluční důkaz  $\square$  z formule  $G$  se nazývá **rezoluční vyvrácení formule  $G$** 
  - a tedy  $G$  je nepravdivá ve všech interpretacích, tj.  $G$  je nesplnitelná

# Strom rezolučního důkazu

- **strom rezolučního důkazu** klauzule  $C$  z formule  $G$  je binární strom:
  - kořen je označen klauzulí  $C$ ,
  - listy jsou označeny klauzulemi z  $G$  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:  $G = \{\{p, r\}, \{q, \neg r\}, \{\neg q\}, \{\neg p\}\}$        $C = \square$



**strom rezolučního vyvrácení**  
(rezoluční důkaz  $\square$  z  $G$ )

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

# Substituce

- **co s proměnnými? vhodná substituce a unifikace**
  - $f(X, a, g(Y)) < 1, f(h(c), a, Z) < 1, \quad X = h(c), Z = g(Y) \implies f(h(c), a, g(Y)) < 1$
- **substituce je libovolná funkce  $\theta$  zobrazující výrazy do výrazů tak, že platí**
  - $\theta(E) = E$  pro libovolnou konstantu  $E$
  - $\theta(f(E_1, \dots, E_n)) = f(\theta(E_1), \dots, \theta(E_n))$  pro libovolný funkční symbol  $f$
  - $\theta(p(E_1, \dots, E_n)) = p(\theta(E_1), \dots, \theta(E_n))$  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  $[X_1/\xi_1, \dots, X_n/\xi_n]$  kde  $X_i$  jsou proměnné a  $\xi_i$  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  $\sigma$  takové, že  $p\sigma = q\sigma$  nazýváme **unifikací** a příslušnou substituci **unifikátorem**.
- **Unifikátorem** množiny  $S$  literálů nazýváme substituce  $\theta$  takovou, že množina

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

má jediný prvek.

- příklad:  $S = \{ \text{datum}( D1, M1, 2003 ), \text{datum}( 1, M2, Y2) \}$

unifikátor  $\theta = [D1/1, M1/2, M2/2, Y2/2003]$

$S\theta = \{ \text{datum}( 1, 2, 2003 ) \}$

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

# Rezoluční princip v PL1

- základ:

- rezoluční princip ve výrokové logice 
$$\frac{C_1 \cup \{l\} \quad \{\neg l\} \cup C_2}{C_1 \cup C_2}$$
- 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

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

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

# Příklad: rezoluce v PL1

- příklad:  $C_1 = \{p(X, Y), q(Y)\}$      $C_2 = \{\neg q(a), s(X, W)\}$

- přejmenování proměnných:  $\rho = [X/Z]$

$$C_1 = \{p(Z, Y), q(Y)\} \quad C_2 = \{\neg q(a), s(X, W)\}$$

- nejobecnější unifikátor:  $\sigma = [Y/a]$

$$C_1 = \{p(Z, a), q(a)\} \quad C_2 = \{\neg q(a), s(X, W)\}$$

- rezoluční princip:  $C = \{p(Z, a), s(X, W)\}$

- vyzkoušejte si:

$$C_1 = \{q(X), \neg r(Y), p(X, Y), p(f(Z), f(Z))\}$$

$$C_2 = \{n(Y), \neg r(W), \neg p(f(W), f(W))\}$$

# Rezoluce v PL1

## ■ Obecný rezoluční princip v PL1

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

- kde  $\rho$  je přejmenováním proměnných takové, že množiny klauzulí  $\{A_1\rho, \dots, A_m\rho, C_1\rho\}$  a  $\{B_1, \dots, B_n, C_2\}$  nemají společné proměnné
- $\sigma$  je nejobecnější unifikátor množiny  $\{A_1\rho, \dots, A_m\rho, B_1, \dots, B_n\}$
- příklad:  $A_1 = a(X)$  vs.  $\{\neg B_1, \neg B_2\} = \{\neg a(b), \neg a(Z)\}$   
v jednom kroku potřebují vyrezolvovat zároveň  $B_1$  i  $B_2$

## ■ Rezoluce v PL1

- **korektní:** jestliže existuje rezoluční vyvrácení  $F$ , pak  $F$  je nespílitelná
- **úplná:** jestliže  $F$  je nespílitelná, 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 =  $\{S \mid S \text{ je splnitelná}\}$  NP úplný,  
nicméně: menší prohledávací prostor vede k rychlejšímu nalezení řešení
- strategie pro zefektivnění prohledávání  $\Rightarrow$  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 účastníci 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**  $S$

  - $\{\{p, q\}, \{\neg p, q\}, \{p, \neg q\}, \{\neg p, \neg q\}\}$ 

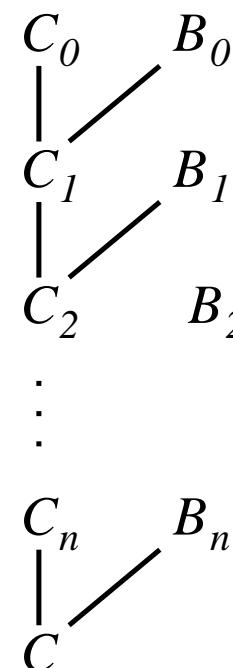
existuje rezoluční vyvrácení

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  $\langle C_0, B_0 \rangle, \dots, \langle C_n, B_n \rangle$  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 vyvrácení**  $S =$  lineární rezoluční důkaz  $\square$  z  $S$



# Lineární rezoluce II.

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

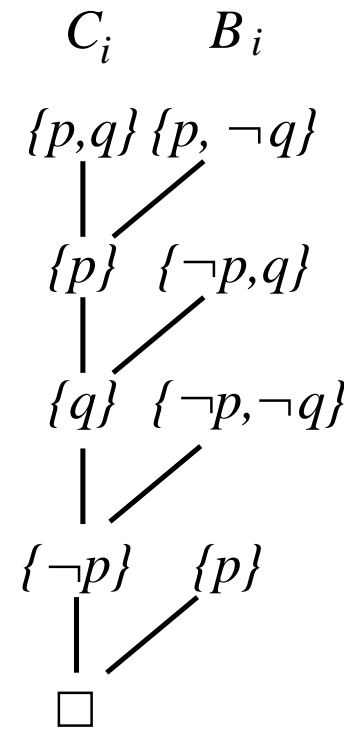
$$A_1 = \{p, q\}$$

$$A_2 = \{p, \neg q\}$$

$$A_3 = \{\neg p, q\}$$

$$A_4 = \{\neg p, \neg q\}$$

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



# Prologovská notace

- Klauzule v matematické logice

- $\{H_1, \dots, H_m, \neg T_1, \dots, \neg T_n\}$        $H_1 \vee \dots \vee H_m \vee \neg T_1 \vee \dots \vee \neg T_n$

- **Hornova klauzule:** nejvýše jeden pozitivní literál

- $\{H, \neg T_1, \dots, \neg T_n\}$        $\{H\}$        $\{\neg T_1, \dots, \neg T_n\}$

- $H \vee \neg T_1 \vee \dots \vee \neg T_n$        $H$        $\neg T_1 \vee \dots \vee \neg T_n$

- **Pravidlo:** jeden pozitivní a alespoň jeden negativní literál

- Prolog:  $H : - T_1, \dots, T_n.$       Matematická logika:  $H \Leftarrow T_1 \wedge \dots \wedge T_n$

- $H \Leftarrow T$        $H \vee \neg T$        $H \vee \neg T_1 \vee \dots \vee \neg T_n$       Klauzule:  $\{H, \neg T_1, \dots, \neg T_n\}$

- **Fakt:** pouze jeden pozitivní literál

- Prolog:  $H.$       Matematická logika:  $H$       Klauzule:  $\{H\}$

- **Cílová klauzule:** žádný pozitivní literál

- Prolog:  $: - T_1, \dots T_n.$       Matematická logika:  $\neg T_1 \vee \dots \vee \neg T_n$       Klauzule:  $\{\neg T_1, \dots \neg T_n\}$

# 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 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  $P$**  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  $P$  **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

⇒ 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 nespíitelná, právě když existuje rezoluční vyvrácení  $S$  pomocí **vstupní rezoluce**.
- **Korektnost** platí stejně jako pro ostatní omezení rezoluce
- **Úplnost LI-rezoluce pro Hornovy klauzule:**  
Necht'  $P$  je množina programových klauzulí a  $G$  cílová klauzule.  
Je-li množina  $P \cup \{G\}$  Hornových klauzulí nespíitelná,  
pak existuje rezoluční vyvrácení  $P \cup \{G\}$  pomocí LI-rezoluce.
  - vstupní rezoluce pro (obecnou) formuli sama o sobě není úplná  
 $\implies$  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-rezoluční ukážeme nespíitelnost  $P_1 \wedge \dots \wedge P_n \wedge (\neg G_1 \vee \dots \vee \neg G_m)$
  - 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
- Uspořádaná klauzule (*definite clause*) = posloupnost literálů
  - nelze volně měnit pořadí literálů
- Rezoluční princip pro uspořádané klauzule:

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

- **uspořádaná rezolventa:**  $\{\neg A_0, \dots, \neg A_{i-1}, \neg B_0\rho, \dots, \neg B_m\rho, \neg A_{i+1}, \dots, \neg A_n\}\sigma$
- $\rho$  je přejmenování proměnných takové, že klauzule  $\{A_0, \dots, A_n\}$  a  $\{B, B_0, \dots, B_m\}\rho$  nemají společné proměnné
- $\sigma$  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.

- Uspořádané klauzule

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

## Hornovy klauzule

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

- Příklad:

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

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

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

# LD-rezoluce

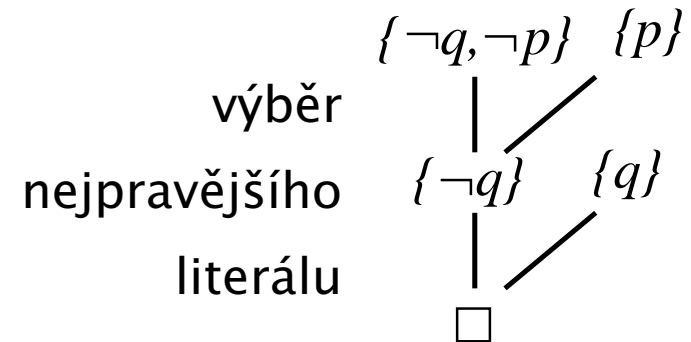
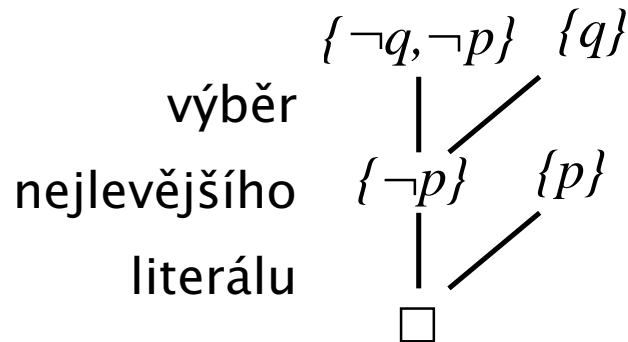
- **LD-rezoluční vyvrácení** množiny uspořádaných klauzulí  $P \cup \{G\}$  je posloupnost  $\langle G_0, C_0 \rangle, \dots, \langle G_n, C_n \rangle$  taková, že
  - $G_i, C_i$  jsou uspořádané klauzule
  - $G = G_0$
  - $G_{n+1} = \square$
  - $G_i$  je uspořádaná cílová klauzule
  - $C_i$  je přejmenování 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 z 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

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



- **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)

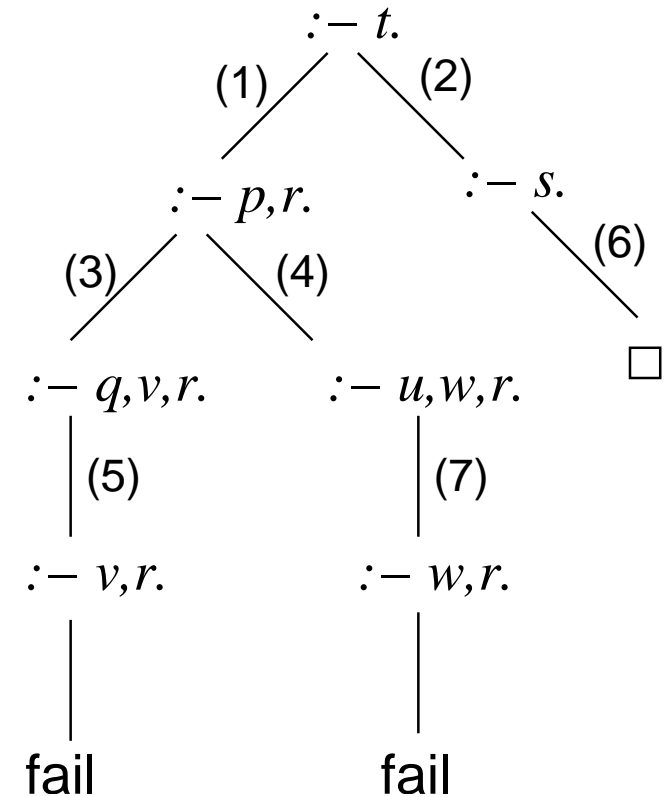
$p :- u, w.$  (4)

$q.$  (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  $G$
- kořenem stromu je cílová klauzule  $G$
- v uzlech stromu jsou rezolventy (rodiče uzlu a programové klauzule)
  - číslo vybrané programové klauzule pro rezoluci je v příkladu uvedeno jako ohodnocení hrany
- listy jsou dvojího druhu:
  - označené prázdnou klauzulí – jedná se o **úspěšné uzly** (*success 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

$: -a(Z).$

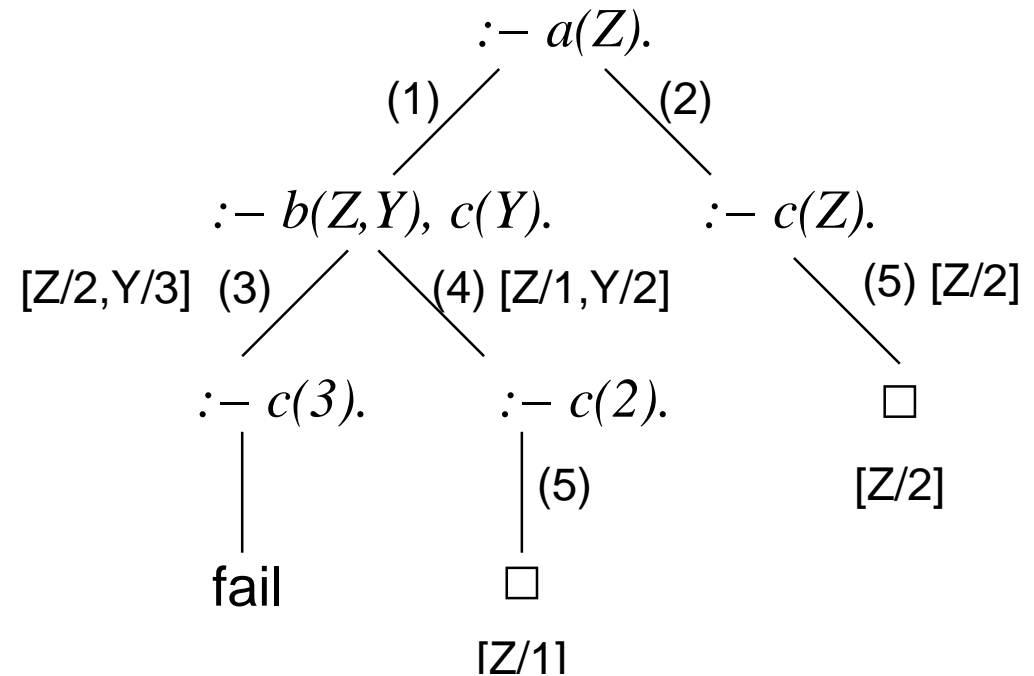
$a(X) : -b(X, Y), c(Y).$  (1)

$a(X) : -c(X).$  (2)

$b(2, 3).$  (3)

$b(1, 2).$  (4)

$c(2).$  (5)



Cvičení:

$p(B) : -q(A, B), r(B).$

$p(A) : -q(A, A).$

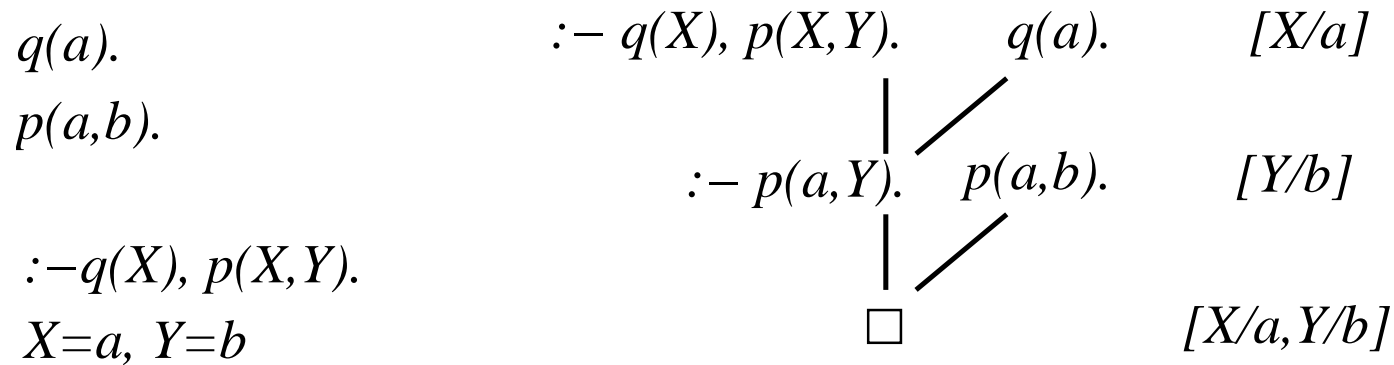
$q(a, a).$

$q(a, b).$

$r(b).$

ve výsledné substituci jsou pouze proměnné z dotazu  
 výsledné substituce jsou  $[Z/1]$  a  $[Z/2]$   
 nezajímá mě substituce  $[Y/2]$

# Výsledná substituce (*answer substitution*)



- Každý krok SLD-rezoluce vytváří novou unifikační substituci  $\theta_i$   
 $\Rightarrow$  potenciální instanciaci 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**

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

kde  $G = \{\neg G_1, \neg G_2, \dots, \neg G_n\}$  a  $\vec{X}$  je vektor proměnných v  $G$   
nesplnitelnost (1) je ekvivalentní tvrzení (2) a (3)

$$(2) P \vdash \neg G$$

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

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 řídicí struktury
- Použitelná výpočetní strategie = **prohledávání stromu do hloubky**
  - jednoduché řídicí struktury (zásobník)
  - lineární paměťová náročnost
  - **není ale úplná**: nenalezne vyvrácení i když existuje
    - procházení nekonečné větve stromu výpočtu
      - ⇒ na nekonečných stromech dojde k zacyklení
    - nedostaneme se tak na jiné existující úspěšné uzly

# SLD-rezoluce v Prologu: úplnost

- **Prolog**: prohledávání stromu do hloubky  
⇒ **neúplnost** použité výpočetní strategie

- Implementace SLD-rezoluce v Prologu

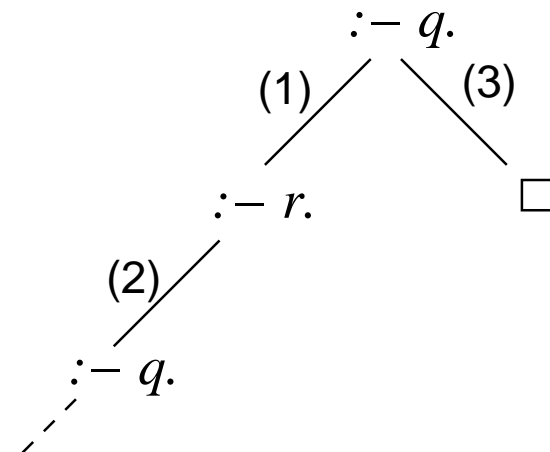
- **není úplná**

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:  $a(X, f(X))$ .

- by vedl k:  $[B/X], [X/f(X)]$

- Unifikátor pro  $g(X_1, \dots, X_n)$  a  $g(f(X_0, X_0), f(X_1, X_1), \dots, f(X_{n-1}, X_{n-1}))$

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

$$X_2 = f(f(X_0, X_0), f(X_0, X_0)), \dots$$

délka termu pro  $X_k$  exponenciálně narůstá

⇒ **exponenciální složitost** na ověření kontroly výskytu

- Test výskytu se **při unifikaci v Prologu neprovádí**

- Důsledek: ? –  $X = f(X)$  uspěje s  $X = f(f(f(f(f(f(f(f(f(...))))))))))$

# SLD-rezoluce v Prologu: korektnost

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

⇒ **není korektní**

(1)  $t(X) : -p(X, X). \quad : -t(X).$

$p(X, f(X)). \quad X = f(f(f(f(\dots)))))))))$       problém se projeví

(2)  $t : -p(X, X). \quad : -t.$

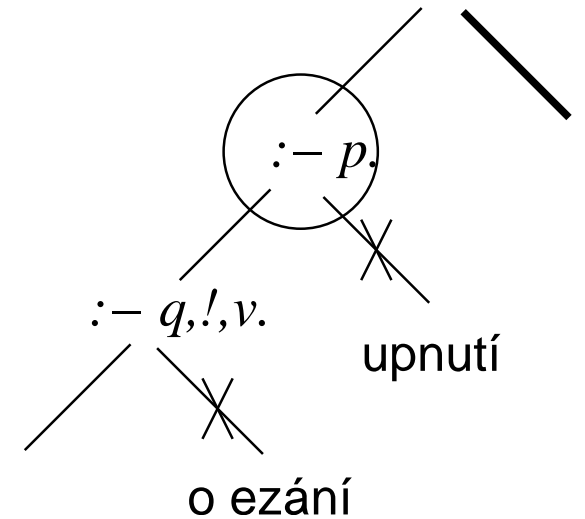
$p(X, f(X)). \quad \text{yes}$       dokazovací systém nehledá unifikátor pro  $X$  a  $f(X)$

- 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

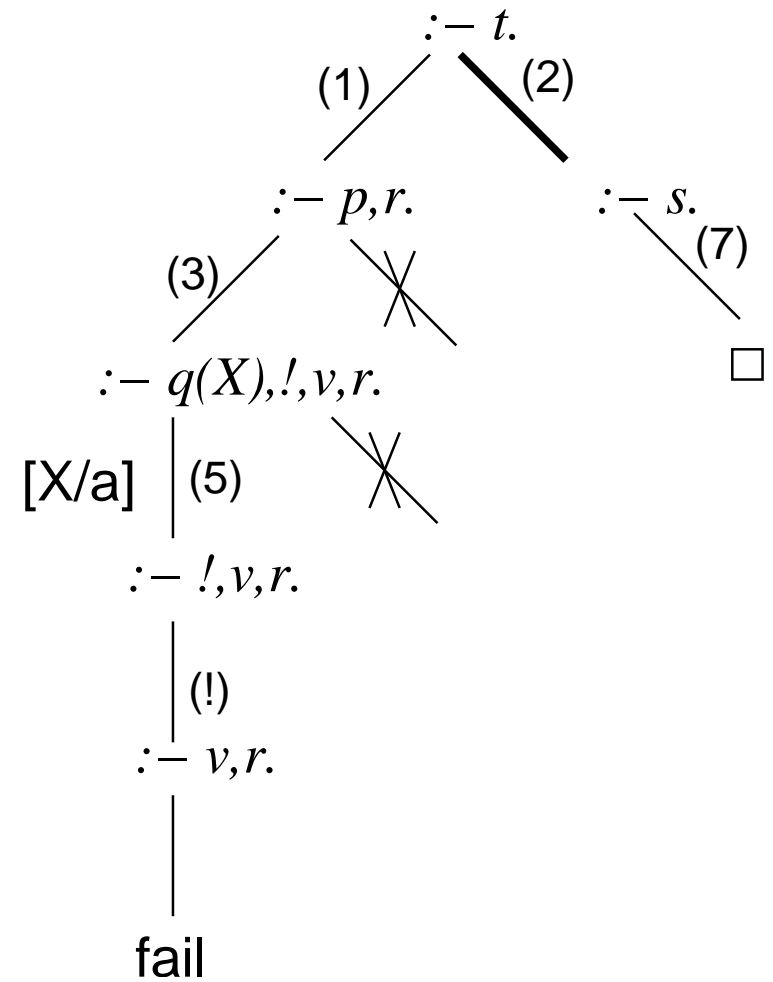
- nemá ale žádnou deklarativní sémantiku
- místo toho **mění implementaci programu**
- $p : -q, !, v.$
- **snažíme se splnit  $q$**   
řez se syntakticky chová jako kterýkoliv jiný literál
- pokud uspějí
  - ⇒ přeskočím řez a pokračuji jako by tam řez nebyl
- pokud ale **neuspějí (a tedy i při backtrackingu)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



upnutí  
ořezání

# Příklad: řez

- $t :- p, r.$  (1)
- $t :- s.$  (2)
- $p :- q(X), !, v, r.$  (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)

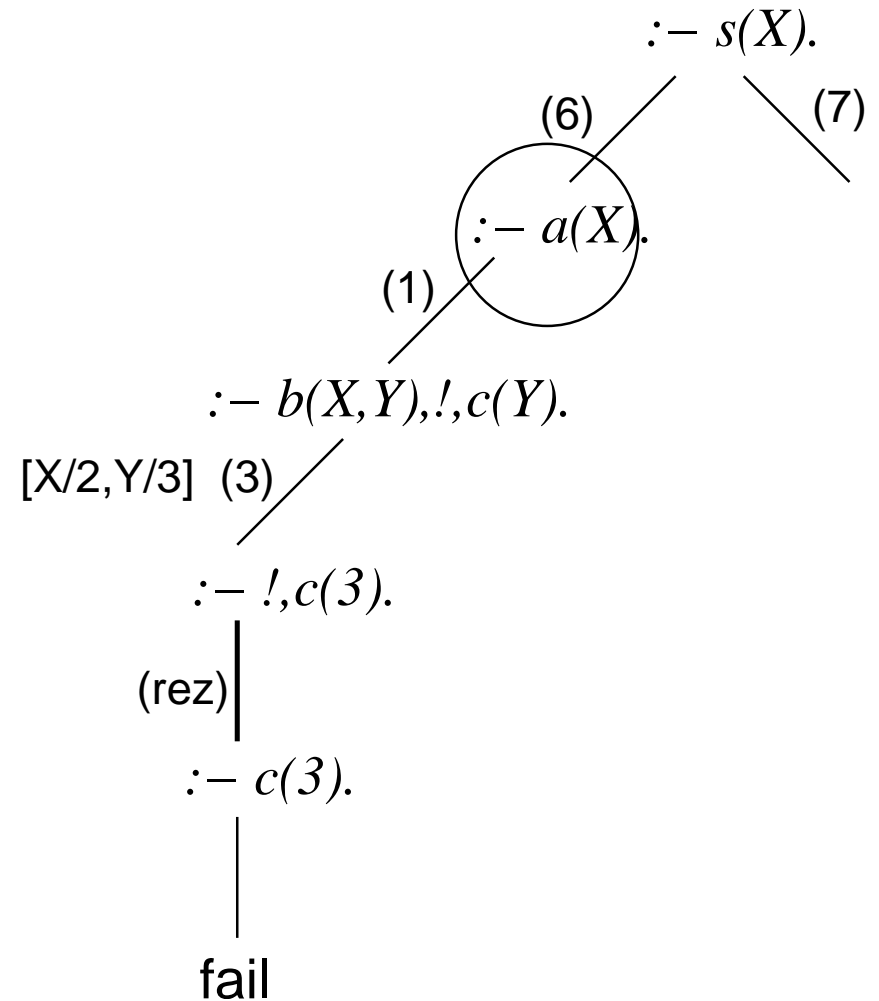
$p(B) :- q(A, B), r(B).$  (8)

$p(A) :- q(A, A).$  (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)

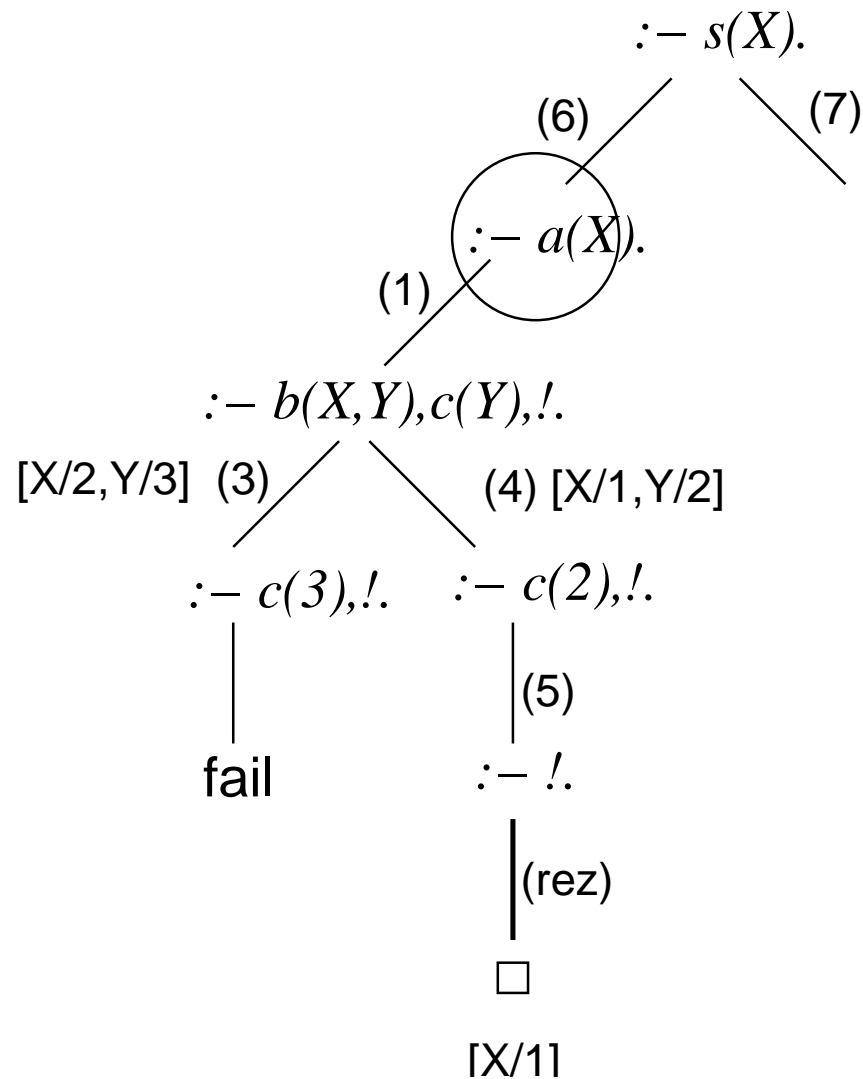
$p(B) : -q(A, B), r(B).$  (8)

$p(A) : -q(A, A).$  (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  $G^1$  odvodit nějakým rezolučním důkazem ze vstupní množiny  $P \cup \{G\}$ .

<sup>1</sup>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  $t_1, \dots, t_n$  přiřadí term  $f(t_1, \dots, t_n)$
  - 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

$\text{Tichy}(s(0)).$  % (1)

$\text{Tichy}(s(s(X))) \text{ :- } \text{Tichy}(X).$  % (2)

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

**Cvičení:** Napište minimální Herbrandův model pro následující logický program.

`muz(petr). muz(pavel). zena(o1ga). zena(jitka).`

`pary(X,Y) :- zena(X), muz(Y).`

Uved'te další model tohoto programu, který není minimální.

# 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  $M(P)$ .
- **Připomenutí: Operační sémantikou** logického programu  $P$  rozumíme množinu  $O(P)$  všech atomických formulí bez proměnných, které lze pro nějaký cíl  $G^1$  odvodit nějakým rezolučním důkazem ze vstupní množiny  $P \cup \{G\}$ .  
<sup>1</sup>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
  - $nad(X, Y) : \neg na(X, Y). \quad na(c, b).$   
 $nad(X, Y) : \neg na(X, Z), nad(Z, Y). \quad na(b, a).$
  - 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 vyjádřena 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):  $\frac{P \not\equiv A}{\neg A}$  (CWA)
- pro SLD-rezoluci:  $P \not\equiv nad(a, c)$ , 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 :  $\neg A$   
:  $\neg A$  má definitivně (konečně) neúspěšný SLD-strom      (*negation as failure, NF*)  
 $\neg A$
- normální cíl: cíl obsahující i negativní literály
  - :  $\neg nad(c, a), \neg nad(b, c)$ .
- rozdíl mezi CWA a NF
  - program  $nad(X, Y) : \neg nad(X, Y)$ , cíl :  $\neg \neg nad(b, c)$
  - neexistuje odvození cíle podle NF, protože SLD-strom :  $\neg nad(b, c)$  je nekonečný
  - existuje odvození cíle podle CWA, protože neexistuje vyvrácení :  $\neg nad(b, c)$
- 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**
  - $nad(X, Y) : \neg na(X, Y)$ .
  - $nad(X, Y) : \neg na(X, Z), nad(Z, Y)$ .
  - lze psát jako:  $nad(X, Y) : \neg (na(X, Y)) \vee (na(X, Z), nad(Z, Y))$ .
  - 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:  $na(X_1, X_2) : \neg X_1 = c, X_2 = b$ .
  - $na(X_1, X_2) : \neg X_1 = b, X_2 = a$ .
  - zúplnění:  $na(X_1, X_2) \leftrightarrow (X_1 = c, X_2 = b) \vee (X_1 = b, X_2 = a)$ .



# Zúplnění programu

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

# IF( $q, P$ )

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

- $q/n$  predikátový symbol programu  $P$   $na(b, a) : -g.$

- $X_1, \dots, X_n$  jsou „nové“ proměnné, které se nevyskytují nikde v  $P$

- Necht'  $C$  je klauzule ve tvaru

$$q(t_1, \dots, t_n) : -L_1, \dots, L_m$$

kde  $m \geq 0$ ,  $t_1, \dots, t_n$  jsou termy a  $L_1, \dots, L_m$  jsou literály.

Pak označme  $E(C)$  výraz  $\exists Y_1, \dots, Y_k(X_1 = t_1, \dots, X_n = t_n, L_1, \dots, L_m)$

kde  $Y_1, \dots, Y_k$  jsou všechny proměnné v  $C$ .

- Necht'  $def(q/n) = \{C_1, \dots, C_j\}$ .

Pak formuli  $IF(q, P)$  získáme následujícím postupem:

$$q(X_1, \dots, X_n) : -E(C_1) \vee E(C_2) \vee \dots \vee E(C_j) \quad \text{pro } j > 0 \text{ a}$$

$$q(X_1, \dots, X_n) : -\square \quad \text{pro } j = 0 [q/n \text{ není v programu } P]$$

# 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  $f/m$ :  $X_1 = Y_1 \wedge \dots \wedge X_m = Y_m \rightarrow f(X_1, \dots, X_m) = f(Y_1, \dots, Y_m)$

5. pro každý  $p/m$ :  $X_1 = Y_1 \wedge \dots \wedge X_m = Y_m \rightarrow (p(X_1, \dots, X_m) \rightarrow p(Y_1, \dots, Y_m))$

6. pro všechny různé  $f/m$  a  $g/n$ ,  $(m, n \geq 0)$ :  $f(X_1, \dots, X_m) \neq g(Y_1, \dots, Y_n)$

7. pro každý  $f/m$ :  $f(X_1, \dots, X_m) = f(Y_1, \dots, Y_m) \rightarrow X_1 = Y_1 \wedge \dots \wedge X_m = Y_m$

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:** Necht'  $P$  logický program a  $: \neg A$  cíl.  
Jestliže  $: \neg A$  má definitivně neúspěšný SLD-strom,  
pak  $\forall (\neg A)$  je logickým důsledkem  $\text{comp}(P)$  (nebo-li  $\text{comp}(P) \models \forall (\neg A)$ )
- **Úplnost NF pravidla:** Necht'  $P$  je logický program. Jestliže  $\text{comp}(P) \models \forall (\neg A)$ ,  
pak existuje definitivně neúspěšný SLD-strom  $: \neg A$ .
  - 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  $(\text{comp}(P) \models \forall (\neg A))$  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  $S_0, \dots, S_m$  ( $S_i \equiv$  **stratum**)
  - $p(\dots) : - \dots, q(\dots), \dots \in P, p \in S_k \implies q \in S_0 \cup \dots \cup S_k$
  - $p(\dots) : - \dots, \neg q(\dots), \dots \in P, p \in S_k \implies q \in S_0 \cup \dots \cup S_{k-1}$

# Stratifikované programy II

- program je *m*-stratifikovaný  $\iff m$  je nejmenší index takový, že  $S_0 \cup \dots \cup S_m$  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: 
$$\frac{:- C. \text{ má konečně neúspěšný SLD-strom}}{\neg C}$$
- 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*( $X$ ) :  $\neg \neg \text{blokovany}(X)$ .

*blokovany*( $X$ ) :  $\neg \text{na}(Y, X)$ .

*na*( $a, b$ ).

$:- \text{nahore}(c)$ .

*yes*

$:- \text{nahore}(c)$ .

$$\begin{array}{c} | \\ \hline \end{array} \text{ :- } \neg \text{blokovany}(c)$$

$:- \text{blokovany}(c)$ .

$$\begin{array}{c} | \\ \hline \end{array} \text{ :- } \text{na}(Y, c)$$

$$\begin{array}{c} | \\ \hline \end{array} \text{ FAIL}$$

$\Rightarrow$  **úspěšné odvození**

# SLDNF rezoluce: neúspěšné odvození

- NF pravidlo: 
$$\frac{:- C. \text{ má konečně neúspěšný SLD-strom}}{\neg C}$$
- 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

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

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

$na(_, _).$

$:- nahore(X).$

$no$

$$\begin{array}{c} :- nahore(X). \\ | \\ :- \neg blokovany(X). \end{array}$$

$:- blokovany(X).$

$$\begin{array}{c} | \\ :- na(Y, X). \\ | \\ \square \end{array}$$

$\Rightarrow$  neúspěšné odvození



# SLDNF rezoluce: uvázlé odvození

- NF pravidlo: 
$$\frac{:- C. \text{ má konečně neúspěšný SLD-strom}}{\neg C}$$
- 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**

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

$blokovany(X) : -na(Y, X).$

$na(a, b).$

$:- nahore(X).$

$:- nahore(X).$

$:- blokovany(X).$

|  
|  
|  
|  
|

$:- \neg blokovany(X).$

|  
|  
|  
|  
|

$:- na(Y, X).$

$[Y/a, X/b]$

□

$[X/b]$

$\Rightarrow$  **uvázlé odvození**

# Cvičení: SLDNF odvození

Napište množinu SLDNF odvození pro uvedený dotaz.

$\text{:- } a(B).$

$a(X) \text{ :- } b(X), \text{ \+ } c(X).$

$a(X) \text{ :- } d(X), Y \text{ is } X+1, \text{ \+ } c(Y), b(X).$

$b(1).$

$c(A) \text{ :- } d(A).$

$d(1).$

# SLD<sup>+</sup> odvození

- $P$  je normální program,  $G_0$  normální cíl,  $R$  selekční pravidlo:  
**SLD<sup>+</sup>-odvození**  $G_0$  je buď konečná posloupnost

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

nebo nekonečná posloupnost

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

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<sup>+</sup>-odvození může být:
  1. **úspěšné:**  $G_i = \square$
  2. **neúspěšné**
  3. **blokové:**  $G_i$  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
    - $0 \iff$  žádné  $SLD^+$ -odvození s pravidlem  $R$  není blokováno
    - $k + 1 \iff$  maximální úroveň cílů :  $\neg A$ ,  
které ve tvaru  $\neg A$  blokují  $SLD^+$ -odvození  $G_0$ , je  $k$
  - nekonečná úroveň cíle: **blokováné SLDNF odvození**
- **Množina SLDNF odvození** =  $\{(SLDNF \text{ odvození } G_0) \cup (SLDNF \text{ odvození } : \neg A)\}$ 
  - 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<sup>+</sup>-odvození**  $G_0$  je SLDNF odvození  $G_0$
- je-li SLD<sup>+</sup>-odvození  $\langle G_0; C_0 \rangle, \dots, G_i$  **blokováno na  $\neg A$** 
  - tj.  $G_i$  je tvaru :  $- L_1, \dots, L_{m-1}, \neg A, L_{m+1}, \dots, L_n$

pak

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

$G\theta$  je logickým důsledkem  $\text{comp}(P)$

- 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

$\Rightarrow \neg A$  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/books/materials.html>  
průsvitky ke knize
- Barták R. **Přednáška Omezující podmínky na MFF UK, Praha**.
  - <http://kti.ms.mff.cuni.cz/~bartak/podminky/index.html>
- **SICStus Prolog User's Manual**. Kapitola o CLP(FD).
  - <http://www.fi.muni.cz/~hanka/sicstus/doc/html/>
- **Příklady v distribuci SICStus Prologu**: cca 60 příkladů, zdrojový kód
  - `lib/sicstus-*/library/clpfd/examples/`

# 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
    - viz interaktivní osnova IS
  - PA167 Rozvrhování
    - <http://www.fi.muni.cz/~hanka/rozvrhovani>
    - zahrnuty CP techniky pro řešení rozvrhovacích problémů

# Omezení (*constraint*)

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

**Omezení**  $c$  na  $Y$  je podmnožina  $D_1 \times \dots \times D_k$

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

# Problém splňování podmínek (CSP)

- Dána
  - konečná množina **proměnných**  $X = \{x_1, \dots, x_n\}$
  - konečná množina hodnot (**doména**)  $D = \{D_1, \dots, D_n\}$
  - konečná množina **omezení**  $C = \{c_1, \dots, c_m\}$ 
    - 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:  $A \neq B, B \neq C$

# Řešení CSP

- **Částečné ohodnocení proměnných**  $(d_1, \dots, d_k), k < n$ 
  - některé proměnné mají přiřazenu hodnotu
- **Úplné ohodnocení proměnných**  $(d_1, \dots, d_n)$ 
  - všechny proměnné mají přiřazenu hodnotu
- **Řešení CSP**
  - úplné ohodnocení proměnných, které splňuje všechna omezení
  - $(d_1, \dots, d_n) \in D_1 \times \dots \times D_n$  je **řešení**  $(X, D, C)$ 
    - pro každé  $c_i \in C$  na  $x_{i_1}, \dots, x_{i_k}$  platí  $(d_{i_1}, \dots, d_{i_k}) \in c_i$
- 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}, ...
- **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
- **optimalizace:** ženy učí co nejdříve  
Anna+Eva+Marie  $\neq$  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) řídicí 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

- Přiřad'te 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
- $\text{domain}([E,N,D,O,R,Y], 0, 9), \text{domain}([S,M], 1, 9)$
- $$\begin{array}{r} 1000*S + 100*E + 10*N + D \\ + \quad 1000*M + 100*O + 10*R + E \\ \#= 10000*M + 1000*O + 100*N + 10*E + Y \end{array}$$
- $\text{all\_distinct}([S,E,N,D,M,O,R,Y])$
- $\text{labeling}([S,E,N,D,M,O,R,Y])$

# Od LP k CLP I.

- CLP: rozšíření logického programování o omezující podmínky
- CLP systémy se liší podle typu domény
  - $\text{CLP}(\mathcal{A})$  generický jazyk
  - $\text{CLP}(FD)$  domény proměnných jsou konečné (*Finite Domains*)
  - $\text{CLP}(\mathbb{R})$  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**
  - $A = B$
  - $A \#< B, \quad A \text{ in } 0..9, \quad \text{domain}([A,B],0,9), \quad \text{all\_distinct}([A,B,C])$

# Od LP k CLP II.

- Pro řešení podmínek se používají **konzistenční techniky**
  - *consistency techniques, propagace omezení (constraint propagation)*
  - omezení:  $A \text{ in } 0..2, B \text{ in } 0..2, B \#< A$   
domény po propagaci omezení  $B \#< A$ :  $A \text{ in } 1..2, B \text{ in } 0..1$
- Podmínky jsou deterministicky vyhodnoceny v okamžiku volání podmínky
- **Prohledávání doplněno konzistenčními technikami**
  - $A \text{ in } 1..2, B \text{ in } 0..1, B \#< A$
  - 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:  $A \text{ in } 0..2, B \text{ in } 0..2, B \#< A$   
výstup:  $A \text{ in } 1..2, B \text{ in } 0..1, B \#< A$

# 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 nejobecnějšího unifikátoru (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  $\equiv$  **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

# Systemy a jazyky pro CP

- **IBM ILOG CP** 1987
  - omezující podmínky v C++, Java nebo generickém modelovacím jazyku OPL
  - implementace podmínek založena na objektově orientovaném programování
  - špičkový komerční sw, vznikl ve Francii, nedávno zakoupen IBM
  - nyní nově volně dostupný pro akademické použití
- **Swedish Institute of Computer Science: SICStus Prolog** 1985
  - silná CLP(*FD*) knihovna, komerční i akademické použití
- **IC-PARC, Imperial College London, Cisco Systems: ECL<sup>i</sup>PS<sup>e</sup>** 1984
  - široké možnosti kooperace mezi různými řešiči: konečné domény, reálná čísla, repaír
  - od 2004 vlastní Cisco Systems volně dostupné pro akademické použití, rozvoj na IC-PARC, platformy: Windows, Linux, Solaris
- Mnoho dalších systémů: Choco, Gecode, Minion, Oz, SWI Prolog, ...

# CLP(*FD*) v SICStus Prologu

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



# 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`
  - POZOR: neplést `#=<` a `#>=` s operátory pro implikaci: `#<=` `#=>`
- `sum(Variables, RelOp, Suma)`
  - `domain([A,B,C,F], 1, 3)`, `sum([A,B,C], #=, F)`
  - `Variables` i `Suma` musí být doménové proměnné nebo celá čísla
- `scalar_product(Coeffs, Variables, RelOp, ScalarProduct)`
  - `domain([A,B,C,F], 1, 6)`, `scalar_product([1,2,3], [A,B,C], #=, F)`
  - `Variables` i `Value` musí být doménové proměnné nebo celá čísla, `Coeffs` jsou celá čísla
  - POZOR na pořadí argumentů, nejprve jsou celočíselné koeficienty, pak dom. proměnné
  - `scalar_product(Coeffs, Variables, #=, Value, [consistency(domain)])`
    - silnější typ konzistence
    - POZOR: domény musí mít konečné hranice

# Základní globální omezení

- `all_distinct(List)`
  - všechny proměnné různé
- `cumulative(...)`
  - disjunktivní a kumulativní rozvrhování
- `cumulatives(...)`
  - kumulativní rozvrhování na více zdrojů

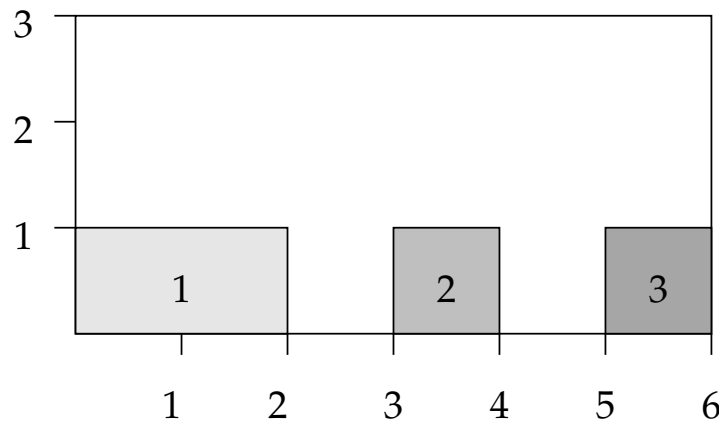
# Všechny proměnné různé

- `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 (unární zdroj)

- `cumulative([task(Start, Duration, End, 1, Id) | Tasks])`
- Rozvržení úloh zadaných startovním a koncovým časem (Start, End), dobou trvání (nezáporné Duration) a identifikátorem (Id) tak, aby se nepřekrývaly
  - příklad s konstantami: `cumulative([task(0,2,2,1,1), task(3,1,4,1,2), task(5,1,6,1,3)])`



- příklad: vytvoření rozvrhu, za předpokladu, že **doba trvání hodin není stejná**

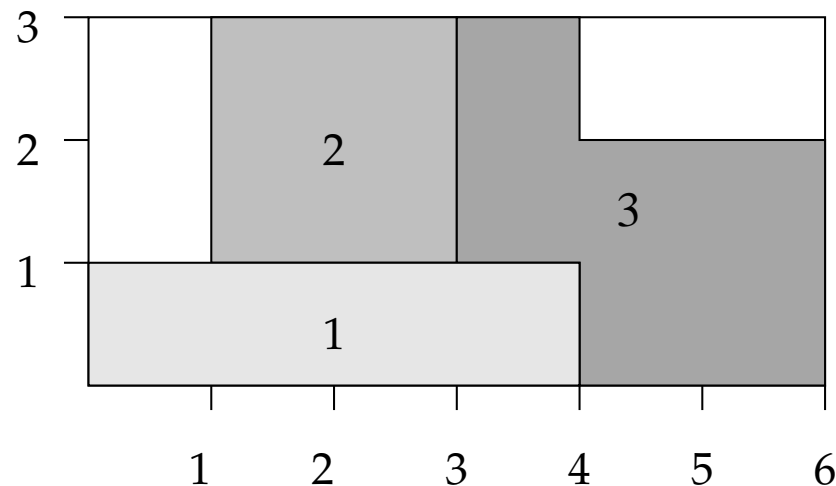
`JanE#= Jan+3, PetrE#= Petr+1, AnnaE#= Anna+2, ...`

```
cumulative(task(Jan, 3, JanE, 1, 1), task(Petr, 1, PetrE, 1, 2), task(Anna, 2, AnnaE, 1, 1),
task(Ota, 2, OtaE, 1, 4), task(Eva, 2, EvaE, 1, 5), task(Marie, 3, MarieE, 1, 6)])
```

# Kumulativní rozvrhování

- `cumulative([task(Start,Duration,End,Demand,TaskId) | Tasks], [limit(Limit)])`
- Rozvržení úloh zadaných startovním a koncovým časem (Start,End), dobou trvání (nezáporné Duration), požadovanou kapacitou zdroje (Demand) a identifikátorem (Id) tak, aby se nepřekrývaly a aby celková kapacita zdroje nikdy nepřekročila Limit
- Příklad s konstantami:

```
cumulative([task(0,4,4,1,1),task(1,2,3,2,2),task(3,3,6,2,3),task(4,2,6,1,4)], [limit(3)])
```





# Kumulativní rozvrhování s více zdroji

- Rozvržení úloh tak, aby se nepřekrývaly a daná kapacita zdrojů nebyla překročena (limit zdroje chápán jako horní mez – `bound(upper)`)
- `cumulatives([task(Start,Duration,End,Demand,MachineId)|Tasks],  
[machine(Id,Limit)|Machines],[bound(upper)])`
- Úlohy zadány startovním a koncovým časem (`Start,End`), dobou trvání (nezáporné `Duration`), požadovanou kapacitou zdroje (`Demand`) a požadovaným typem zdroje (`MachineId`)
- Zdroje zadány identifikátorem (`Id`) a kapacitou (`Limit`)
- Příklad:  
?- `domain([B,C],1,2),  
cumulatives([task(0,4,4,1,1),task(3,1,4,1,B), task(5,1,6,1,C)]  
[machine(1,1),machine(2,1)],  
[bound(upper)]).` `C in 1..2, B=2`

# Příklad: kumulativní rozvrhování

- Vytvořte rozvrh pro následující úlohy, tak aby nebyla překročena kapacita 13 zdrojů, 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       |

# Řešení: kumulativní rozvrhování

```
| ?- schedule(13, [16,6,13,7,5,18,4], [2,9,3,7,10,1,11], 69, Ss, End).
```

```
Ss = [0,16,9,9,4,4,0], End = 22 ?
```

```
schedule(Limit, Ds, Rs, MaxCas, Ss, End) :-
 domain(Ss, 0, MaxCas), End in 0..MaxCas,
 vytvor_u_lohy(Ss,Ds,Rs,1,Tasks),
 cumulative(Tasks, [limit(Limit)]),
 after(Ss, Ds, End), % koncový čas
 append(Ss, [End], Vars),
 labeling([minimize(End)],Vars).
```

```
vytvor_u_lohy([],[],[],_Id,[]).
```

```
vytvor_u_lohy([S|Ss], [D|Ds], [R|Rs], Id, [task(S,D,E,R,Id)|Tasks]):-
 NewId is Id+1,
 E #= S+D,
 vytvor_u_lohy(Ss,Ds,Rs, NewId,Tasks).
```

```
after([], [], _).
```

```
after([S|Ss], [D|Ds], End) :- E #>= S+D, after(Ss, Ds, End).
```

# Vestavěné predikáty pro labeling

- Instanciaci 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 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 (1) nejmenší velikostí domény fd\_size(Var,Size)  
(2) (pokud s nejmenší velikostí domény více, tak) nejlevější z nich

- ffc: s (1) nejmenší velikostí domény  
(2) největším množstvím omezením „čekajících“ na proměnné fd\_degree(Var,Size)  
(3) nejlevější z nich

- min/max: s (1) nejmenší/největší hodnotou v doméně proměnné  
(2) nejlevnější z nich fd\_min(Var,Min)/fd\_max(Var,Max)

# Hledání optimálního řešení

(předpokládejme minimalizaci)

- Parametry Labeling/2 pro optimalizaci:  $\text{minimize}(F)/\text{maximize}(F)$ 
  - $\text{Cena} \# = A+B+C$ ,  $\text{labeling}([\text{minimize}(\text{Cena})], [A,B,C])$
- **Metoda větví a mezí (*branch&bound*)**
  - algoritmus, který implementuje proceduru pro minimalizaci (duálně pro maximalizaci)
  - 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
    - přidává se tedy inkrementálně omezení  $LB \# < UB$  pro snižující se  $UB$  tak, jak nalézáme kvalitnější řešení



# **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

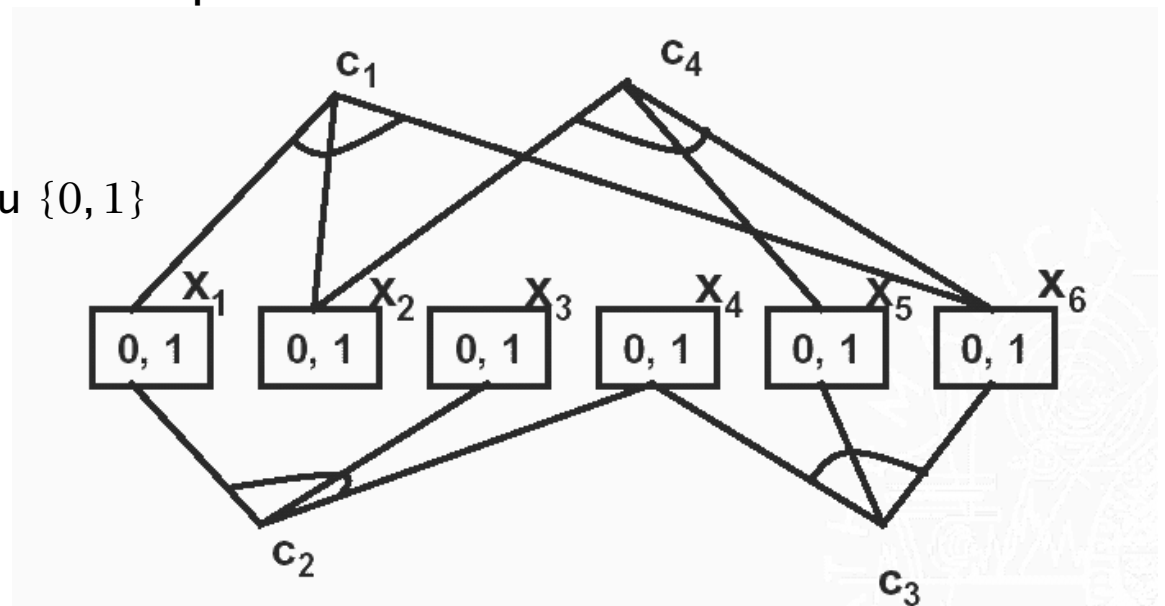
- proměnné  $x_1, \dots, x_6$  s doménou  $\{0, 1\}$

- 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

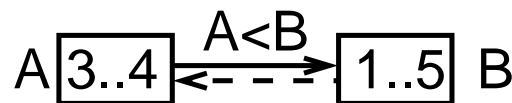
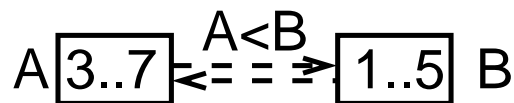
- získáváme unifikovaný tvar CSP problému, řada algoritmů navržena pro binární CSP
- bohužel ale značné zvětšení velikosti problému

## ■ Nebinární podmínky

- složitější propagační algoritmy
- lze využít jejich sémantiky pro lepší propagaci
  - příklad: `all_distinct` 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  $(V_j, V_i)$



konzistence (A,B)



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

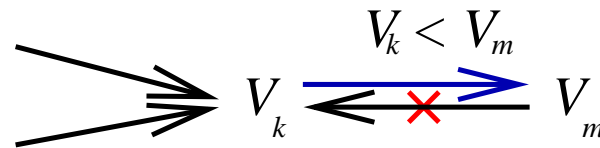
- CSP je hranově konzistentní, právě když jsou všechny jeho hrany (v obou směrech) hranově konzistentní

# Algoritmus revize hrany

- Jak udělat hranu  $(V_i, V_j)$  hranově konzistentní?
- Z domény  $D_i$  vyřadím takové hodnoty  $x$ , které nejsou konzistentní s aktuální doménou  $D_j$  (pro  $x$  neexistuje žádná hodnota  $y$  v  $D_j$  tak, aby ohodnocení  $V_i = x$  a  $V_j = y$  splňovalo všechny binární podmínky mezi  $V_i$  a  $V_j$ )
- procedure revise( $(i, j)$ )  
Deleted := false  
for  $\forall x$  in  $D_i$  do  
    if neexistuje  $y \in D_j$  takové, že  $(x, y)$  je konzistentní  
    then  $D_i := D_i - \{x\}$   
        Deleted := true  
    end if  
return Deleted  
end revise
- domain( $[V_1, V_2], 2, 4$ ),  $V_1 \# < V_2$     revise( $(1, 2)$ ) smaže 4 z  $D_1, D_2$  se nezmění

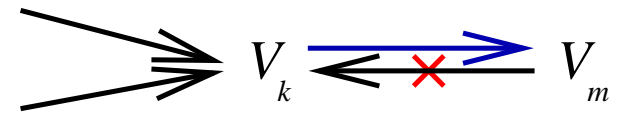
# Dosažení hranové konzistence problému

- Jak udělat CSP hranově konzistentní?
  - revize je potřeba opakovat, dokud se mění doména nějaké proměnné
  - efektivnější: opakování revizí můžeme dělat pomocí **fronty**
    - přidáváme do ní hrany, jejichž konzistence mohla být narušena zmenšením domény
- Jaké hrany přesně revidovat po zmenšení domény?
  - ty, jejichž konzistence může být zmenšením domény proměnné narušena jsou to hrany  $(i, k)$ , které vedou do proměnné  $V_k$  se zmenšenou doménou



- hranu  $(m, k)$  vedoucí z proměnné  $V_m$ , která zmenšení domény způsobila, není třeba revidovat (změna se jí nedotkne)

# Algoritmus AC-3



- procedure AC-3(G)

$Q := \{(i, j) \mid (i, j) \in \text{hrany}(G), i \neq j\}$     % seznam hran pro revizi

while Q non empty do

    vyber a smaž  $(k, m)$  z Q

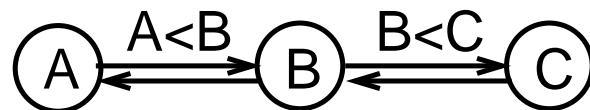
    if revise( $(k, m)$ ) then    % pridani pouze hran, ktere

$Q := Q \cup \{(i, k) \in \text{hrany}(G), i \neq k, i \neq m\}$     % dosud nejsou ve fronte

end while

end AC-3

- Příklad:



$A < B, B < C: (3..7, 1..5, 1..5) \xrightarrow{AB} (3..4, \underline{1..5}, 1..5) \xrightarrow{BA}$   
 $(3..4, \underline{4..5}, 1..5) \xrightarrow{BC} (3..4, 4, \underline{1..5}) \xrightarrow{CB} (3..4, 4, 5)$   
 $\xrightarrow{AB} (3, 4, 5)$

- Technika AC-3 je dnes asi nejpoužívanější, ale stále není optimální
- Jaké budou domény A, B, C po AC-3 pro:  $\text{domain}([A, B, C], 1, 10)$ ,  $A \neq B + 1$ ,  $C \# < B$

# Je hranová konzistence dostatečná?

- Použitím AC odstraníme mnoho nekompatibilních hodnot

- Dostaneme potom řešení problému?

NE

- Víme alespoň zda řešení existuje?

NE

- $\text{domain}([X, Y, Z], 1, 2)$ ,  $X \neq Y$ ,  $Y \neq Z$ ,  $Z \neq X$

- hranově konzistentní

- nemá žádné řešení

- Jaký je tedy význam AC?

- někdy dá řešení přímo

- nějaká doména se vyprázdní  $\Rightarrow$  řešení neexistuje

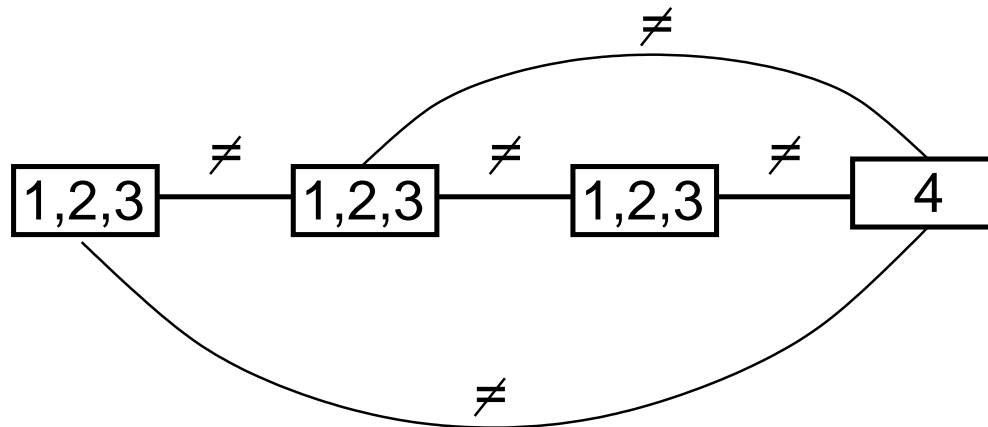
- všechny domény jsou jednoprvkové  $\Rightarrow$  máme řešení

- v obecném případě se alespoň zmenší prohledávaný prostor



# k-konzistence

- Mají NC a AC něco společného?
  - NC: konzistence jedné proměnné
  - AC: konzistence dvou proměnných
  - ... můžeme pokračovat
- CSP je **k-konzistentní** právě tehdy, když můžeme libovolné konzistentní ohodnocení  $(k-1)$  různých proměnných rozšířit do libovolné  $k$ -té proměnné



4-konzistentní graf

- Pro obecné CSP, tedy i pro nebinární podmínky

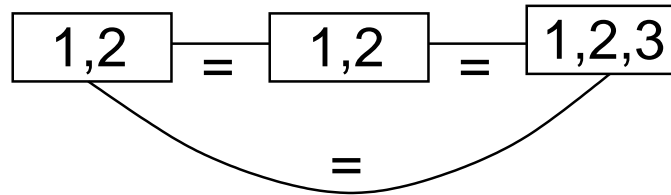
# Silná k-konzistence

3-konzistentní graf

(1, 1) lze rozšířit na (1, 1, 1)

(2, 2) lze rozšířit na (2, 2, 2)

(1, 3) ani (2, 3) nejsou konzistentní dvojice (nerozšiřujeme je)



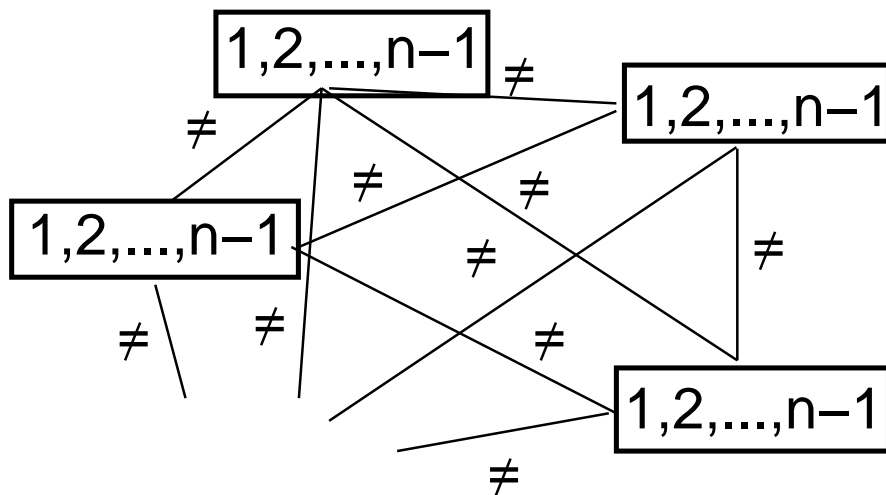
není 2-konzistentní

(3) nelze rozšířit

- CSP je **silně k-konzistentní** právě tehdy, když je j-konzistentní pro každé  $j \leq k$
- Silná k-konzistence  $\Rightarrow$  k-konzistence
- Silná k-konzistence  $\Rightarrow$  j-konzistence  $\forall j \leq k$
- k-konzistence  $\not\Rightarrow$  silná k-konzistence
- NC = silná 1-konzistence = 1-konzistence
- AC = (silná) 2-konzistence

# Konzistence pro nalezení řešení

- Máme-li graf s  $n$  vrcholy, jak silnou konzistenci potřebujeme, abychom přímo našli řešení?
  - silná  $n$ -konzistence je nutná pro graf s  $n$  vrcholy
    - $n$ -konzistence nestačí (viz předchozí příklad)
    - silná  $k$ -konzistence pro  $k < n$  také nestačí



graf s  $n$  vrcholy  
domény  $1..(n-1)$

silně  $k$ -konzistentní pro každé  $k < n$   
přesto nemá řešení

# Řešení nebinárních podmínek

- k-konzistence má exponenciální složitost, v reálu se nepoužívá
- 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 hodnou  $x \in D_i$  existuje ohodnocení zbylých proměnných v podmínce tak, že podmínka platí
  - $A + B \neq C$ ,  $A \in 1..3$ ,  $B \in 2..4$ ,  $C \in 3..7$  je obecně hranově konzistentní
- Využívá se sémantika podmínek
  - speciální typy konzistence pro globální omezení
    - viz `all_distinct`
  - konzistence mezi
    - propagace pouze při změně nejmenší a největší hodnoty v doméně proměnné
- Pro různé podmínky lze použít různý druh konzistence
  - $A \#< B$ : hranová konzistence, konzistence mezi

# 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( $(V, D, C)$ )

$Q := V$

while  $Q$  non empty do

vyber a smaž  $V_j \in Q$

for  $\forall C$  takové, že  $V_j \in scope(C)$  do

$W := revise(V_j, C)$

//  $W$  je množina proměnných jejichž, doména se změnila

if  $\exists V_i \in W$  taková, že  $D_i = \emptyset$  then return fail

$Q := Q \cup \{W\}$

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

# 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 hodnou  $x \in D_j$  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**
  - $A \#> B \Rightarrow \min(A) = \min(B)+1, \max(B) = \max(A)-1$
  - příklad:  $A \text{ in } 4..10, B \text{ in } 6..18, A \#> B$   
 $\min(A) = 6+1 \Rightarrow A \text{ in } 7..10$   
 $\max(B) = 10-1 \Rightarrow B \text{ in } 6..9$
  - podobně:  $A \#< B, A \#>= B, A \#=< B$

# Konzistence mezí a aritmetická omezení

- $A \# = B + C \Rightarrow \min(A) = \min(B) + \min(C), \max(A) = \max(B) + \max(C)$   
 $\min(B) = \min(A) - \max(C), \max(B) = \max(A) - \min(C)$   
 $\min(C) = \min(A) - \max(B), \max(C) = \max(A) - \min(B)$ 
  - změna  $\min(A)$  vyvolá pouze změnu  $\min(B)$  a  $\min(C)$
  - změna  $\max(A)$  vyvolá pouze změnu  $\max(B)$  a  $\max(C)$ , ...
- Příklad:  $A \text{ in } 1..10, B \text{ in } 1..10, A \# = B + 2, A \# > 5, A \# \setminus = 8$   
 $A \# = B + 2 \Rightarrow \min(A) = 1 + 2, \max(A) = 10 + 2 \Rightarrow A \text{ in } 3..10$   
 $\Rightarrow \min(B) = 1 - 2, \max(B) = 10 - 2 \Rightarrow B \text{ in } 1..8$   
 $A \# > 5 \Rightarrow \min(A) = 6 \Rightarrow A \text{ in } 6..10$   
 $\Rightarrow \min(B) = 6 - 2 \Rightarrow B \text{ in } 4..8$  (nové vyvolání  $A \# = B + 2$ )  
 $A \# \setminus = 8 \Rightarrow A \text{ in } (6..7) \setminus (9..10)$  (meze stejné, k propagaci  $A \# = B + 2$  nedojde)
- Vyzkoušejte si:  $A \# = B - C, A \# \geq B + C$

# 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_distinct` omezení: hodnoty všech proměnných různé
  - `serialized` omezení:  
rozvržení úloh zadaných startovním časem a dobou trvání tak, aby se nepřekrývaly



# Propagace pro all\_distinct

- $U = \{X_2, X_4, X_5\}$ ,  $dom(U) = \{2, 3, 4\}$ :

$\{2, 3, 4\}$  nelze pro  $X_1, X_3, X_6$

$X_1 \in 5..6, X_3 = 5, X_6 \in \{1\} \ \wedge \ (5..6)$

- **Konzistence:**  $\forall \{X_1, \dots, X_k\} \subset V : card\{D_1 \cup \dots \cup D_k\} \geq k$

stačí hledat **Hallův interval**  $I$ : velikost intervalu  $I$  je rovna

počtu proměnných, jejichž doména je v  $I$

- **Inferenční pravidlo**

- $U = \{X_1, \dots, X_k\}$ ,  $dom(U) = \{D_1 \cup \dots \cup D_k\}$

- $card(U) = card(dom(U)) \Rightarrow \forall v \in dom(U), \forall X \in (V - U), X \neq v$

- 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)

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

# 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 do hloubky

- Základní prohledávací algoritmus pro problémy splňování podmínek
- **Prohledávání stavového prostoru do hloubky (*depth first search*)**
- Dvě fáze prohledávání s navracením
  - **dopředná fáze:** proměnné jsou postupně vybírány, rozšiřuje se částečné řešení přiřazením konzistentní 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

# Základní algoritmus prohledávání do hloubky

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

```
procedure labeling(G, a)
```

```
if $a > |\text{uzly}(G)|$ then return uzly(G)
```

```
for $\forall x \in D_a$ do
```

```
 if consistent(G, a) then % consistent(G, a) je nahrazeno FC(G, a), LA(G, a)
```

```
 R := labeling(G, a + 1)
```

```
 if R \neq fail then return R
```

```
return fail
```

```
end labeling
```

Po přiřazení všech proměnných vrátíme jejich ohodnocení

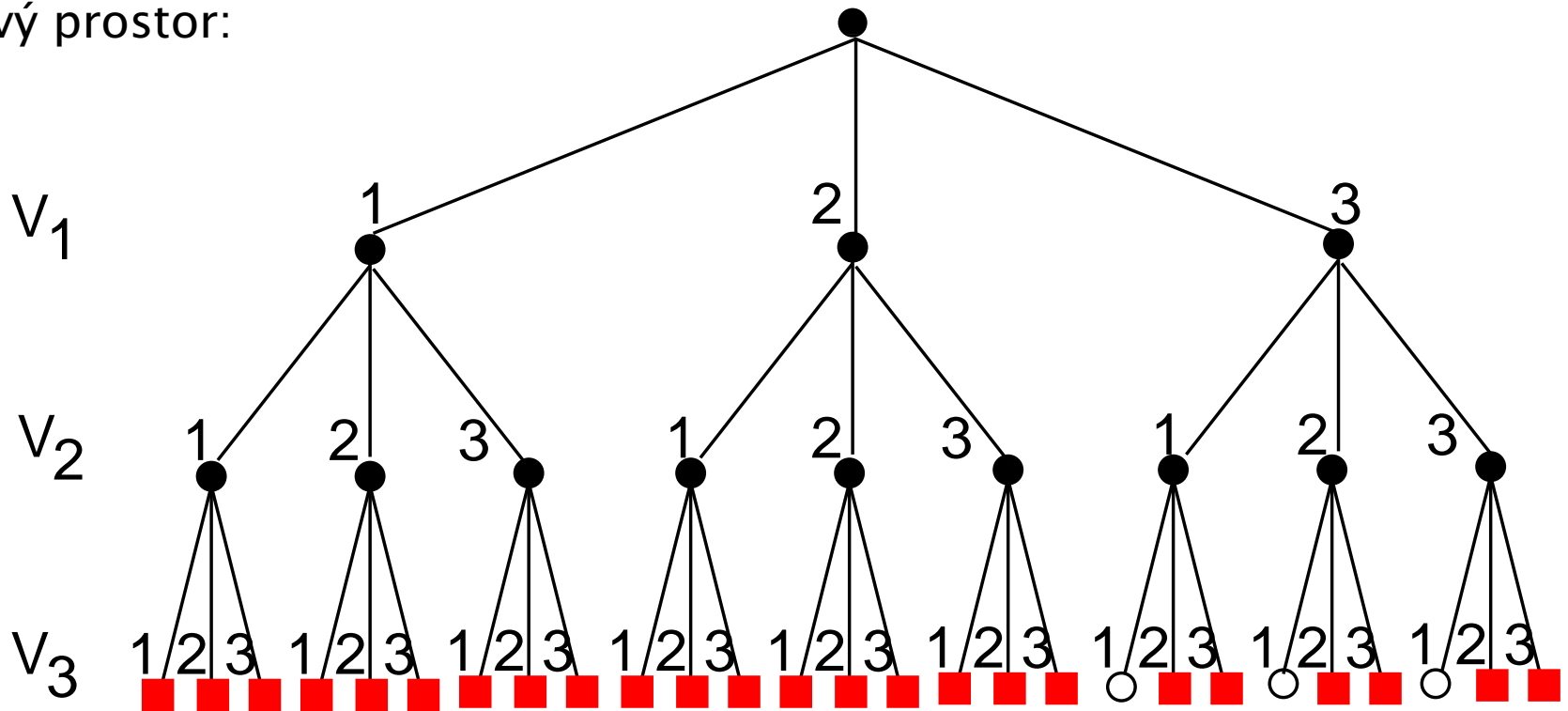
- Procedury `consistent` uvedeme pouze pro binární podmínky

# 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  $BT(G, a)$   
 $Q := \{(V_i, V_a) \in \text{hrany}(G), i < a\}$       % hrany vedoucí z minulých proměnných do aktuální  
 $\text{Consistent} := \text{true}$   
while  $Q$  není prázdná  $\wedge$   $\text{Consistent}$  do  
    vyber a smaž libovolnou hranu  $(V_k, V_m)$  z  $Q$   
     $\text{Consistent} := \text{not revise}(V_k, V_m)$       % pokud vyřadíme prvek, bude doména prázdná  
return  $\text{Consistent}$   
end  $BT$

# Příklad: backtracking

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



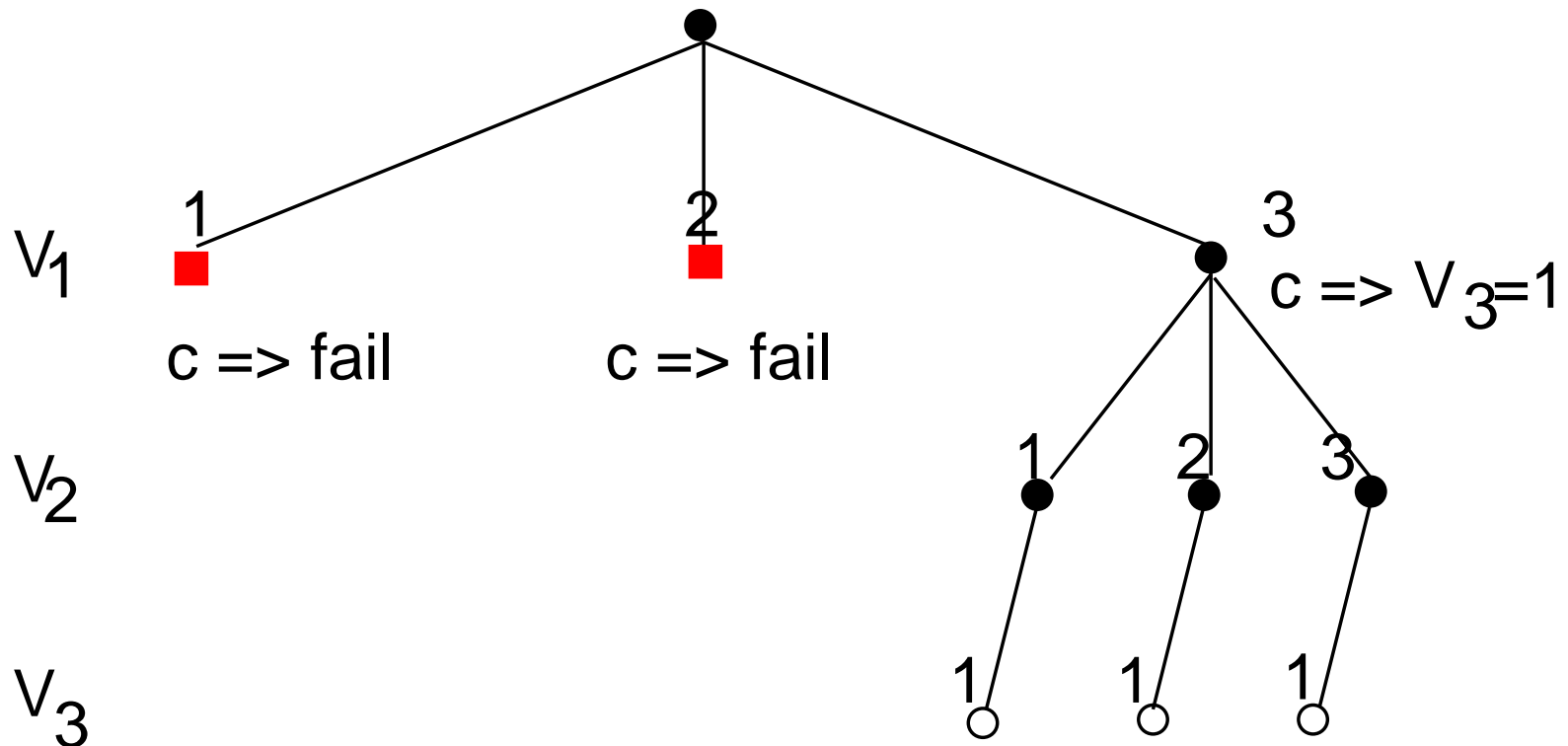
- červené čtverečky: chybný pokus o instanciaci, řešení neexistuje
- nevyplněná kolečka: nalezeno řešení
- černá kolečka: vnitřní uzel, máme pouze částečné přiřazení

# 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  $FC(G, a)$   
     $Q := \{(V_i, V_a) \in \text{hrany}(G), i > a\}$                       % přidání hran z budoucích do aktuální proměnné  
    Consistent := true  
    while Q není prázdná  $\wedge$  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 minulých proměnných do aktuální proměnné není nutno testovat

# Příklad: kontrola dopředu

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





# Pohled dopředu (*LA – looking ahead*)

- LA je rozšíření FC, navíc ověřuje konzistenci hran mezi budoucími proměnnými

- procedure  $LA(G, a)$

$Q := \{(V_i, V_a) \in \text{hrany}(G), i > a\}$       % začínáme s hranami do  $a$

Consistent := true

while  $Q$  není prázdná  $\wedge$  Consistent do

    vyber a smaž libovolnou hranu  $(V_k, V_m)$  z  $Q$

    if  $\text{revise}((V_k, V_m))$  then

$Q := Q \cup \{(V_i, V_k) \mid (V_i, V_k) \in \text{hrany}(G), i \neq k, i \neq m, i > a\}$

        Consistent :=  $(|D_k| > 0)$

return Consistent

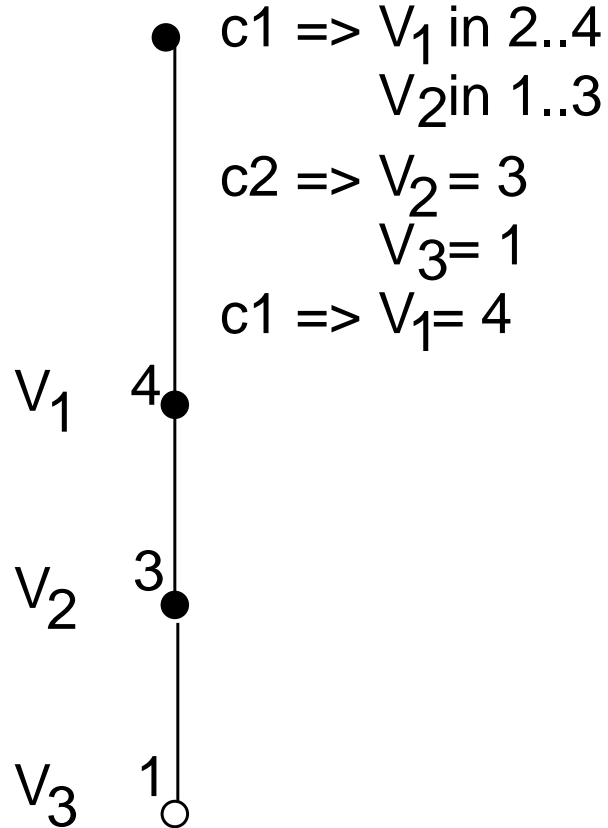
end LA

- Hrany z minulých proměnných do aktuální proměnné opět netestujeme
- Tato LA procedura je založena na AC-3, lze použít i jiné AC algoritmy
- **LA udržuje hranovou konzistenci:** protože ale  $LA(G, a)$  používá AC-3, musíme **zajistit iniciální konzistenci** pomocí AC-3 ještě před startem prohledávání

# Příklad: pohled dopředu (pomocí AC-3)

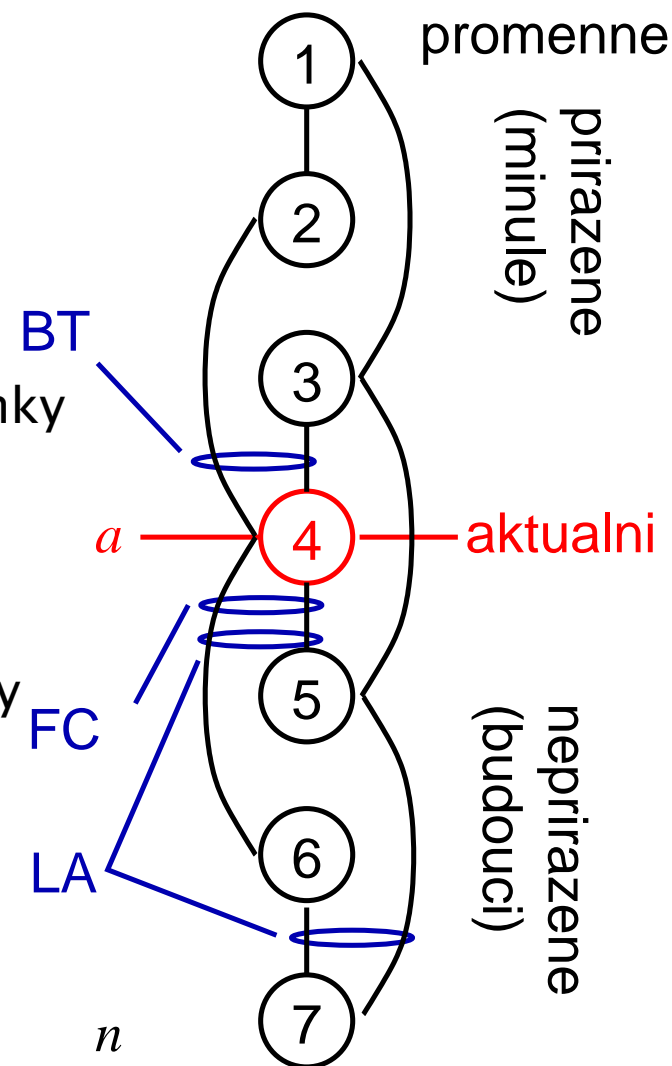
- Omezení:  $V_1, V_2, V_3 \in 1 \dots 4$ ,  $c1 : V_1\# > V_2$ ,  $c2 : V_2\# = 3 \times V_3$
- Stavový prostor

(spouští se iniciální konzistence se před startem prohledávání)



# Přehled algoritmů

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



# Cvičení

1. Jak vypadá stavový prostor řešení pro následující omezení

$A \in 1..4, B \in 3..4, C \in 3..4, B \neq C, A \neq C$

při použití kontroly dopředu a uspořádání proměnných A,B,C? Popište, jaký typ propagace proběhne v jednotlivých uzlech.

2. Jak vypadá stavový prostor řešení pro následující omezení

$A \in 1..4, B \in 3..4, C \in 3..4, B \neq C, A \neq C$

při použití pohledu dopředu a uspořádání proměnných A,B,C? Popište, jaký typ propagace proběhne v jednotlivých uzlech.

3. Jak vypadá stavový prostor řešení pro následující omezení

$\text{domain}([A,B,C],0,1), A \neq B-1, C \neq A*A$

při použití backtrackingu a pohledu dopředu a uspořádání proměnných A,B,C? Popište, jaký typ propagace proběhne v jednotlivých uzlech.

# Cvičení

1. Jaká jsou pravidla pro konzistenci mezí u omezení  $X \neq Y + 5$ ? Jaké typy propagací pak proběhnou v následujícím příkladě při použití konzistence mezí?

$X \text{ in } 1..20, Y \text{ in } 1..20, X \neq Y + 5, Y \neq 10.$

2. Ukažte, jak je dosaženo hranové konzistence v následujícím příkladu:

$\text{domain}([X, Y, Z], 1, 5), X \neq Y, Z \neq Y + 1 .$

# Implementace Prologu

## Literatura:

- Matyska L., Toman D.: Implementační techniky Prologu, Informační systémy, (1990), 21–59.  
<http://www.ics.muni.cz/people/matyska/vyuka/1p/1p.html>

# Opakování: základní pojmy

- Konečná množina klauzulí **Hlava** :- Tělo tvoří **program P**.
- **Hlava** je literál
- **Tělo** je (eventuálně prázdná) konjunkce literálů  $T_1, \dots, T_a, a \geq 0$
- **Literál**  
je tvořen  $m$ -árním predikátovým symbolem  $(m/p)$  a  $m$  termy (argumenty)
- **Term** je konstanta, proměnná nebo složený term.
- **Složený term**  
s  $n$  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 T_1
 ⋮
 call T_a
}
```

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  $A \in S$  a dále vyber klauzuli  $A' : -B_1, \dots, B_n$  ( $n \geq 0$ ) z programu  $P$  takovou, že  $\exists \sigma : A\sigma = A'\sigma$ ;  $\sigma$  je nejobecnější unifikátor.

Pokud neexistuje  $A'$  nebo  $\sigma$ , ukonči cyklus.

4. Nahrad'  $A$  v  $S$  cíli  $B_1$  až  $B_n$ .

5. Aplikuj  $\sigma$  na  $G$  a  $S$ .

6. `end while`

7. Pokud  $S == \text{empty}$ , pak výpočet úspěšně skončil a výstupem je  $G$  se všemi aplikovanými substitucemi.

Pokud  $S \neq \text{empty}$ , 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 interpretu

## 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  $A$ 
  - necht' je těchto klauzulí  $q$
2. Vytvoříme  $q$  kopií množiny  $S$
3. V každé kopii redukuje  $A$  jednou z klauzulí  $A'_i$ .
  - aplikujeme příslušný nejobecnější unifikátor
4. V následujících krocích redukuje 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  $\sigma$ ) 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

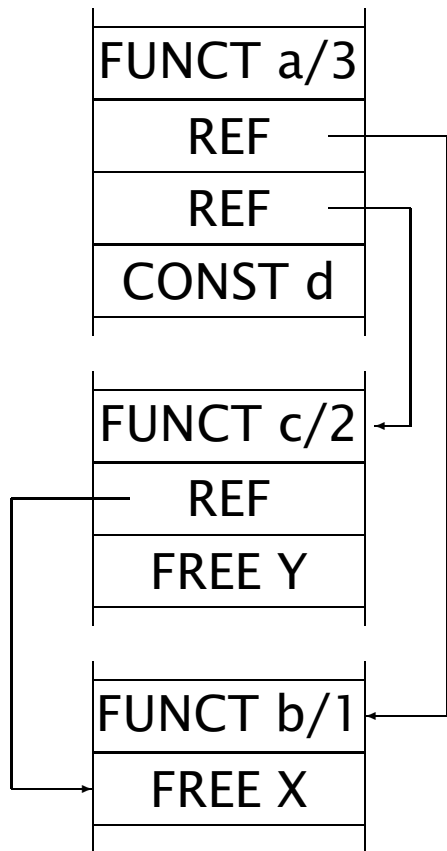
Složené objekty

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

# Kopírování struktur

Příklad:

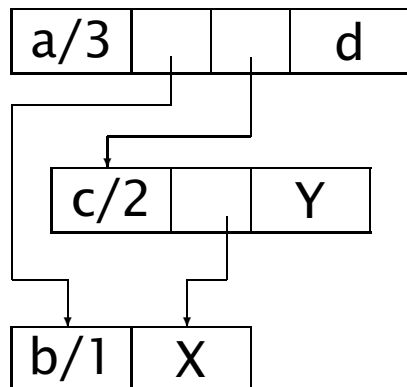
$a(b(X), c(X, Y), d)$ ,





# 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; rámeč >

- kostra\_termu je zdrojový term s očíslovanými proměnnými
- rámeč 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

$\langle a(b(\$1), c(\$1, \$2), d) ; [FREE, FREE] \rangle$

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

**Implementace:**

$\langle \&kostra\_termu; \&rámec \rangle$  (& vrací adresu objektu)

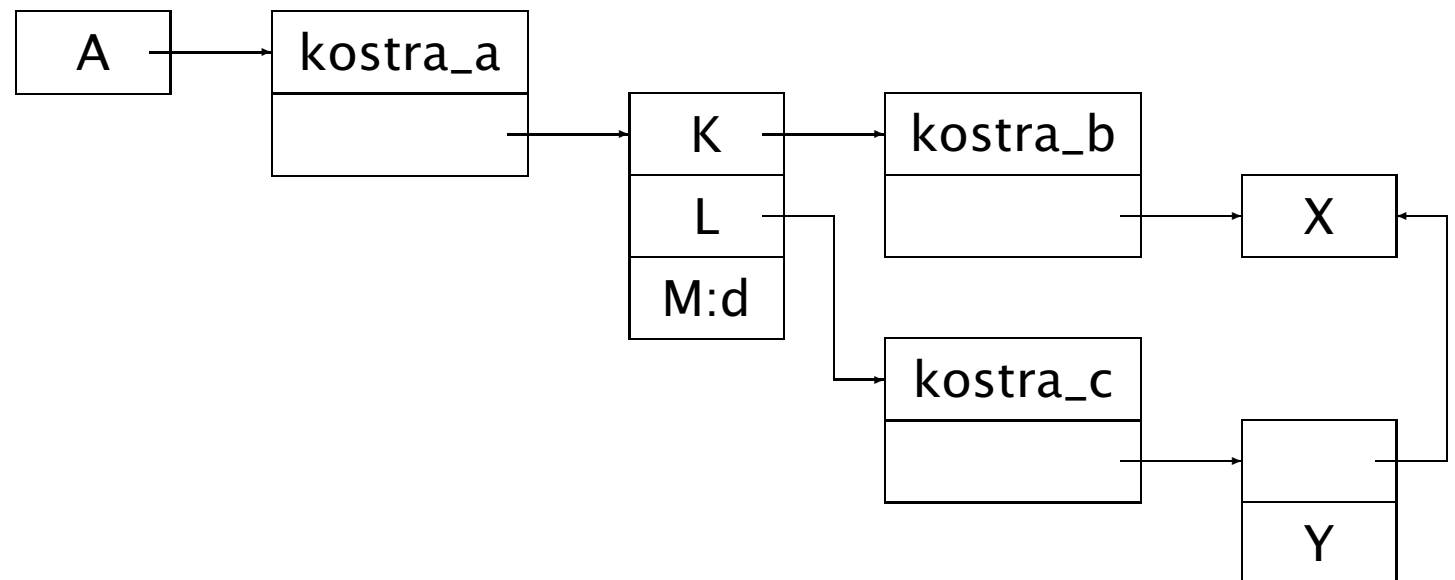
Všechny instance sdílí společnou  $kostru\_termu \Rightarrow$  **sdílení struktur**

# Srovnání: příklad

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

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

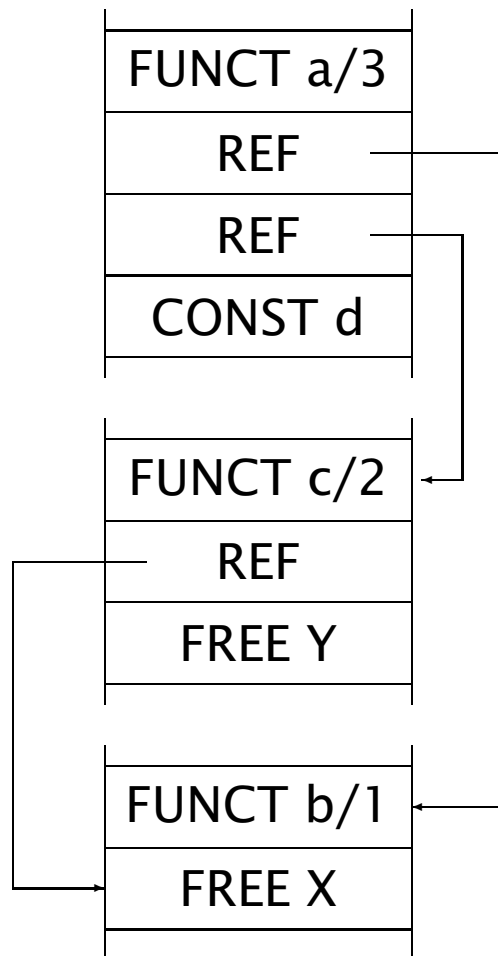
- Sdílení termů:



# 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 zůstala 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
  - $a(X) :- X = b(c, Y), Y = d. \quad ?- W = b(Z, e), a(W). \quad (\text{viz instanciacce } Z)$
- 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

⇒ 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  $\Rightarrow$  **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
  - $a(X) :- b(X), !, c(X). \quad a(3).$   
 $b(1). \quad b(2).$   
 $c(1). \quad c(2).$
- 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

# Interpret Prologu

Základní principy:

- klauzule uloženy jako termy
- **programová databáze**
  - pro uložení klauzulí
  - má charakter haldy
    - umožňuje modifikovatelnost prologovských programů za běhu (assert)
- klauzule zřetězeny podle pořadí načtení
  - triviální zřetězení

Vyhodnocení dotazu: volání procedur řízené unifikací

# Interpret – Základní princip

1. Vyber redukovaný literál („první“, tj. nejlevější literál cíle)
2. Lineárním průchodem od začátku databáze najdi klauzuli, jejíž hlava má stejný funktor a stejný počet argumentů jako redukovaný literál
3. V případě nalezení klauzule založ bod volby procedury
4. Založ dále okolí první klauzule (velikost odvozena od počtu lokálních proměnných v klauzuli)
5. Proved' unifikaci literálu a hlavy klauzule
6. Úspěch  $\Rightarrow$  přidej všechny literály klauzule k cíli („doleva“, tj. na místo redukovaného literálu).  
Tělo prázdné  $\Rightarrow$  výpočet se s úspěchem vrací do klauzule, jejíž adresa je v aktuálním okolí.
7. Neúspěch unifikace  $\Rightarrow$  z bodu volby se obnoví stav a pokračuje se v hledání další vhodné klauzule v databázi.
8. Pokud není nalezena odpovídající klauzule, výpočet se vrací na předchozí bod volby (krátí se lokální i globální zásobník).
9. Výpočet končí neúspěchem: neexistuje již bod volby, k němuž by se výpočet mohl vrátit.
10. Výpočet končí úspěchem, jsou-li úspěšně redukovány všechny literály v cíli.

# Interpret – vlastnosti

- Lokální i globální zásobník
  - při dopředném výpočtu roste
  - při zpětném výpočtu se zmenšuje

Lokální zásobník se může zmenšit při dopředném úspěšném výpočtu deterministické procedury.

- Unifikace argumentů hlavy – obecný unifikační algoritmus  
Současně poznačí adresy instanciovaných proměnných na stopu.
- „Interpret“:

```
interpret(Query, Vars) :- call(Query), success(Query, Vars).
interpret(_,_) :- failure.
```

- dotaz vsazen do kontextu této speciální nedeterministické procedury
- tato procedura odpovídá za korektní reakci systému v případě úspěchu i neúspěchu

# Optimalizace: Indexace

- Zřetězení klauzulí podle pořadí načtení velmi neefektivní
- Provázání klauzulí se stejným funktorem a aritou hlavy (tvoří jednu **proceduru**)
  - tj., **indexace procedur**
- Hash tabulka pro vyhledání první klauzule
- Možno rozhodnout (parciálně) determinismus procedury

# Indexace argumentů

$a(1) \text{ :- } q(1).$

$a(a) \text{ :- } b(X).$

$a([A|T]) \text{ :- } c(A,T).$

- Obecně nedeterministická
- Při volání s alespoň částečně instanciovaným argumentem vždy deterministická (pouze jedna klauzule může uspět)
- **Indexace podle prvního argumentu**

Základní typy zřetězení:

- podle pořadí klauzulí (aktuální argument je volná proměnná)
- dle konstant (aktuální je argument konstanta)
- formální argument je seznam (aktuální argument je seznam)
- dle struktur (aktuální argument je struktura)



# Indexace argumentů II

- Složitější indexační techniky
  - podle všech argumentů
  - podle nejvíce diskriminujícího argumentu
  - kombinace argumentů (indexové techniky z databází)
    - zejména pro přístup k faktům

# Tail Recursion Optimization, TRO

Iterace prováděna pomocí rekurze  $\Rightarrow$  lineární paměťová náročnost cyklů

**Optimalizace koncové rekurze (*Tail Recursion Optimisation*), TRO:**

Okolí se odstraní **před** rekurzivním voláním posledního literálu klauzule, pokud je klauzule resp. její volání deterministické.

Řízení se nemusí vracet:

- v případě úspěchu se rovnou pokračuje
- v případě neúspěchu se vrací na předchozí bod volby („nad“ aktuální klauzulí)
  - aktuální klauzule nemá dle předpokladu bod volby

Rekurzivně volaná klauzule může být volána přímo z kontextu volající klauzule.

# TRO – příklad

Program:

```
append([], L, L).
```

```
append([A|X], L, [A|Y]) :- append(X, L, Y).
```

Dotaz:

```
?- append([a,b,c], [x], L).
```

append volán rekurzivně 4krát

- bez TRO: 4 okolí, lineární paměťová náročnost
- s TRO: 1 okolí, konstatní paměťová náročnost

# Optimalizace posledního volání

TRO pouze speciální případ

obecné **optimalizace posledního volání** (*Last Call Optimization*), LCO

Okolí (před redukcí posledního literálu)

odstraňováno vždy, když leží na vrcholu zásobníku.

## Nutné úpravy interpretu

- disciplína směřování ukazatelů
  - vždy „mladší“ ukazuje na „starší“ („mladší“ budou odstraněny dříve)
  - z lokálního do globálního zásobníku

vyhneme se vzniku „visících odkazů“ při předčasném odstranění okolí

- „globalizace“ lokálních proměnných: lokální proměnné posledního literálu
  - nutno přesunout na globální zásobník
  - pouze pro neinstanciované proměnné

# 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 predikáty (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 čítač adres (Program counter)

  - CP adresa návratu (Continuation Pointer)

  - E ukazatel na nejmladší okolí (Environment)

  - B ukazatel na nejmladší bod volby (Backtrack point)

  - TR vrchol stopy (TRail)

  - H vrchol haldy (Heap)

  - HB vrchol haldy v okamžiku založení posledního bodu volby (Heap on Backtrack point)

  - S ukazatel, používaný při analýze složených termů (Structure 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
  - vykonávají činnost analogickou instrukcím `unify`
  - 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

# 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   |  |           |       |

# Instrukce WAMu

## get instrukce

|            |         |
|------------|---------|
| get_var    | Ai, Y   |
| get_value  | Ai, Y   |
| get_const  | Ai, C   |
| get_nil    | Ai      |
| get_struct | Ai, F/N |
| get_list   | Ai      |

## put instrukce

|                  |         |
|------------------|---------|
| put_var          | Ai, Y   |
| put_value        | Ai, Y   |
| put_unsafe_value | Ai, Y   |
| put_const        | Ai, C   |
| put_nil          | Ai      |
| put_struct       | Ai, F/N |
| put_list         | Ai      |

## unify instrukce

|                   |   |
|-------------------|---|
| unify_var         | Y |
| unify_value       | Y |
| unify_local_value | Y |
| unify_const       | C |
| unify_nil         |   |
| unify_void        | N |

## instrukce řízení

|            |           |
|------------|-----------|
| allocate   |           |
| deallocate |           |
| call       | Proc/N, A |
| execute    | Proc/N    |
| proceed    |           |

## indexační instrukce

|               |      |                  |                          |
|---------------|------|------------------|--------------------------|
| try_me_else   | Next | try              | Next                     |
| retry_me_else | Next | retry            | Next                     |
| trust_me_else | fail | trust            | fail                     |
| 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

Ize 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
```