

IB015 Neimperativní programování

Neimperativní programování v Prologu

Jiří Barnat

Logické programování

- Deklarativní programovací paradigma
- Mechanická dedukce nových informací na základě uvedených údajů a jejich vzájemných vztahů.
- Výpočet = logické dokazování.
- Nejpoužívanější (jediný) programovací jazyk

Prolog

Logické programování

- Deklarativní programovací paradigma
- Mechanická dedukce nových informací na základě uvedených údajů a jejich vzájemných vztahů.
- Výpočet = logické dokazování.
- Nejpoužívanější (jediný) programovací jazyk

Prolog

- Třešnička na dortu neimperativního programování.



Četnost použití Prologu

- Logické programování se masově nerozšířilo.
- Okrajový, specificky používaný programovací prostředek.
- <http://langpop.com/> (2011) – Prolog ani není uveden.

Typické oblasti použití Prologu

- Umělá inteligence, zpracování jazyka, rozvrhování.
- Řešení deklarativně zadaných úloh.

Moderní způsoby použití Prologu

- Vložené použití ve větší (i imperativní) aplikaci.
- Deduktivní databáze.

Logické výpočetní paradigma

- program = databáze faktů a pravidel + cíl
- výpočet = dokazování cíle metodou unifikace a SLD rezoluce
- výsledek = pravda/nepravda, případně seznam hodnot volných proměnných, pro které je cíl pravdivý vzhledem k databázi faktů a pravidel.

Příklad programu

- databáze faktů a pravidel

```
pocasi(prsi).
```

```
stav(mokro) :- pocasi(prsi).
```

- cíl a výsledek

```
?-stav(mokro).
```

```
true.
```

```
?-stav(X).
```

```
X = mokro.
```

SICStus Prolog

- "State-of-the-art" implementace prologu.
- Komerční produkt.
- <http://sicstus.sics.se/>

SWI-Prolog

- Relativně úplná a kompatibilní implementace.
- Freeware.
- <http://swi-prolog.org>

A další ...

- YAP: Yet Another Prolog
<http://www.dcc.fc.up.pt/~vsc/Yap/>

Úvodní intuitivní příklady

SWI-Prolog

- Spouštěno příkazem `swipl`.

Textové interaktivní prostředí

- Standardní výzva: `?-`
- Veškeré povely uživatele musí být zakončeny tečkou.

Základní povely

- Ukončení prostředí: `halt.`
- Návoděda: `help.`
- Načtení souboru `jmeno.pl`: `consult(jmeno).`
- Též alternativně: `[jmeno].`

Struktura jednoduchých příkladů

- Databáze fakt a pravidel uvedena v externím souboru.
- Dotazy kladené skrze interpreter.
- Přípona souboru `.pl` nebo `.pro`.

Notace fakt

- Fakta začínají malým písmenem a končí tečkou.
- Fakty jsou konstanty (nulární relace) a n-ární relace.
- Počet parametrů udáván u jména za lomítkem: `jmeno/N`.

Příklady

- `tohleJeFakt.`
- `tohleTaky(parametr1,parametr2,...,parametrN).`
- `fakt /* a /* zanoreny */ komentar */ .`

Databáze fakt

- `je_teplo.`
`neprsi.`
`kamaradi(vincenc,kvido). /* Znají se od mateřské školy. */`
`kamaradi(vincenc,ferenc). /* Poznali se na pískovišti. */`

Dotazy na databázi

- `?- je_teplo.`
`true.`
- `?- prsi.`
`ERROR: Undefined prsi/0.`
- `?- kamaradi(vincenc,kvido).`
`true.`
- `?- kamaradi(ferenc,vincenc).`
`false.`

Proměnné

- Jména začínají velkým písmenem nebo podtržítkem.
- Je možné je (mimojiné) využít v dotazech.
- Interpreter se pokusí najít vyhovující přiřazení.

Dotazy s využitím proměnných

- `?- kamaradi(vincenc,X).`
X = kvido ;
X = ferenc.
- `?- kamaradi(X,Y).`
X = vincenc,
Y = kvido ;
X = vincenc,
Y = ferenc.

Odpověď interpretru zakončená tečkou

- Indikuje, že nejsou další možnosti.

Odpověď interpretru nezakončená tečkou

- Výzva pro uživatele, zda chce hledání možných řešení ukončit (uživatel vloží tečku), nebo zda si přeje, aby bylo hledáno další řešení (uživatel vloží středník).

Porovnejte

- `?- kamaradi(vincenc,X).`
`X = kvido ; /* uživatel vložil středník */`
`X = ferenc.`
- `?- kamaradi(vincenc,X).`
`X = kvido . /* uživatel vložil tečku */`

Pravidla v databázi

- Zápis: $\text{clovek}(X) :- \text{zena}(X).$
- Význam: Pokud platí $\text{zena}(X)$, pak platí $\text{clovek}(X)$.

Disjunkce

- Zápis: $\text{clovek}(X) :- \text{zena}(X); \text{muz}(X).$
- Alternativní zápis:
 $\text{clovek}(X) :- \text{zena}(X).$
 $\text{clovek}(X) :- \text{muz}(X).$
- Význam: $(\text{zena}(X) \vee \text{muz}(X)) \implies \text{clovek}(X).$

Konjunkce

- Zápis: $\text{unikat}(X) :- \text{zena}(X), \text{muz}(X).$
- Význam: $(\text{zena}(X) \wedge \text{muz}(X)) \implies \text{unikat}(X).$

Databáze:

- `clovek(X) :- zena(X).`
`zena(bozena_nemcova).`

Příklady dotazů

- `?-zena(bozena_nemcova).`
`true.`
- `?-clovek(bozena_nemcova).`
`true.`
- `?-zena(jirik).`
`false.`
- `?- clovek(X).`
`X = bozena_nemcova.`

Rozsah platnosti proměnných

- Použití proměnné je lokalizováno na dané pravidlo.

Příklad

- ```
clovek(X) :- zena(X); muz(X).
unikat(X) :- zena(X), muz(X).
zena(bozena_nemcova).
zena(serena_will).
muz(jara_cimrman).
muz(serena_will).
```
- ```
?- clovek(X).
X = bozena_nemcova ;
X = serena_will ;
X = jara_cimrman ;
X = serena_will.
```

```
?- unikat(X).
X = serena_will.
```

Termy – Základní stavební kameny

Pozorování

- Fakta, pravidla a dotazy jsou tvořeny z termů.

Termy

- Atomy.
- Čísla.
- Proměnné.
- Strukturované termy.

Atomy

- Řetězce začínající malým písmenem, obsahující písmena číslice a znak podtržítka.
- Libovolné řetězce uzavřené v jednoduchých uvozovkách.

Příklady:

- Atomy: `pepa`, `'pepa'`, `'Pepa'`, `'2'`.
- Neatomy: `Pepa`, `2`, `ja a on`, `holmes&watson`.

Test na bytí atomem

- `atom/1` – Pravda, pokud parametr je nestrukturovaným atomem.

Čísla

- Celá i desetinná čísla, používá se desetinná tečka.

```
?- A is 2.5 * 1.3.
```

```
A = 3.25.
```

- Porovnání s aritmetickým vyhodnocením pomocí `:=`.

```
?- 4 := 3+1.
```

```
true.
```

```
?- 4 == 3+1.
```

```
false.
```

```
?- 4 = 3+1.
```

```
false.
```

- Aritmetické vyhodnocení a přiřazení pomocí `is`.

```
?- A is 2*3.
```

```
A = 6.
```

```
?- A == 2*3.
```

```
false.
```

```
?- A = 2*3.
```

```
A = 2*3.
```

Testy na bytí číslem

- `number/1` – Pravda, pokud je parametr číslo.
- `float/1` – Pravda, pokud je parametr desetinné číslo.
- `=\=/2` – Aritmetická neekvivalence.

Strukturované termy

- Funktor (název relace) následovaný sekvencí argumentů.
- Pro funktor platí stejná syntaktická omezení jako pro atomy.
- Argumenty se uvádějí v závorkách, oddělené čárkou.
- Mezi funktorem a seznamem argumentů nesmí být mezera.
- Argumentem může být libovolný term.
- Rekurze je možná.

Arita

- Počet argumentů strukturovaného termu.
- Identifikace strukturovaného termu: `funktor/N`.
- Stejný funktor s jinou aritou označuje jiný term.
- Je možné současně definovat termy `term/2` i `term/3`.

Unifikace v Prologu

Definice

- Dva termy jsou unifikovatelné, pokud jsou identické, anebo je možné zvolit hodnoty proměnných použitých v unifikovaných termech tak, aby po dosazení těchto hodnot byly termy identické.

Operátor $=/2$

- Realizuje unifikaci v Prologu.
- Lze zapisovat infixově.
- Binární operátor ne-unifikace: $\backslash=$, tj. ve standardní notaci $\backslash=/2$.

Příklady

- $?- =(slovo,slovo).$ $?- a(A,[ble,ble]) = a(b(c(d)),B).$
true. $A = b(c(d)),$
 $?- =(slovo,X).$ $B = [ble, ble].$
 $X = slovo.$

- 1) Pokud jsou `term1` a `term2` konstanty (atomy, čísla), pak se unifikují, jestliže jsou tyto termy shodné.
- 2) Pokud je `term1` proměnná a `term2` je libovolný term, pak se unifikují a proměnná `term1` je instanciována hodnotou `term2`. Podobně, pokud je `term2` proměnná a `term1` je libovolný term, pak se unifikují a proměnná `term2` je instanciována hodnotou `term1`.
- 3) Pokud jsou `term1` a `term2` strukturované termy tak se unifikují, pouze pokud mají stejný funktor a aritu, všechny korespondující páry argumentů se unifikují, a všechny instanciaci proměnných z vnořených unifikací jsou kompatibilní.
- 4) Nic jiného.

Příklady unifikace

- `?- snida(karel,livance) = snida(Kdo,Co).`
`Kdo = karel,`
`Co = livance.`
- `?- snida(karel,Jidlo) = snida(Osoba,marmelada).`
`Jidlo = marmelada,`
`Osoba = karel.`
- `?- cdcko(29,beatles,yellow_submarine) = cdcko(A,B,help).`
`false.`
- `?- fce(X,val) = fce(val,X).`
`X = val.`
- `?- partneri(eva,X) = partneri(X,vasek).`
`false.`
- `?- fce(X,Y) = fce(Z,Z).`
`X = Y, Y = Z.`

Příklad

- Uvažme následující dotaz na Prolog:
 $?- X = \text{otec}(X).$
- Jsou unifikovatelné termy, kde jeden term je proměnná a přitom je vlastním podvýrazem druhého termu?

Nekorektnost algoritmu

- Podle definice ne, neboť neexistuje hodnota této proměnné taková, aby po dosazení nastala identita termů.
- Dle algoritmu na přechozím slajdu, však k unifikaci dojde:
 $?- X = \text{otec}(X).$
 $X = \text{otec}(X).$

Poznámka

- Některé implementace Prologu mohou při této unifikaci cyklit.

Kontrola sebevýskytu

- Algoritmus je možné modifikovat tak, aby dával korektní odpověď. Pokud se samostatná proměnná vyskytuje jako podvýraz v druhém termu, termy se neunifikují.
- V praxi se často jedná o nadbytečný test, unifikace je velice častá operace, z důvodu výkonnosti se tento test vynechává.

`unify_with_occurs_check/2`

- Specifický operátor unifikace s testem na sebevýskyt.
- `?- unify_with_occurs_check(X,otec(X)).`
`false.`

Pozorování

- Unifikace je jeden z fundamentů logického programování.
- Pomocí unifikace můžeme odvozovat i sémantickou informaci.

Příklad

- ```
vertical(line(point(X,Y),point(X,Z))).
horizontal(line(point(X,Y),point(Z,Y))).

?- horizontal(line(point(2,3),point(12,3))).
true.

?- vertical(line(point(1,1),X)).
X = point(1, _G2240).
```

## Jak Prolog počítá

## Teorie

- Výpočet = dokazování.
- Kódování problému pomocí Hornových klauzulí.
- Dokazování Selektivní Lineární Definitní rezolucí.
- Při výpočtu Prologu se konstruuje a prochází SLD strom.

## V rámci IB015

- Princip výpočtu s využitím příkladů a neformální demonstrace postupu bez intelektuální zátěže odpovídajícího teoretického fundamentu.

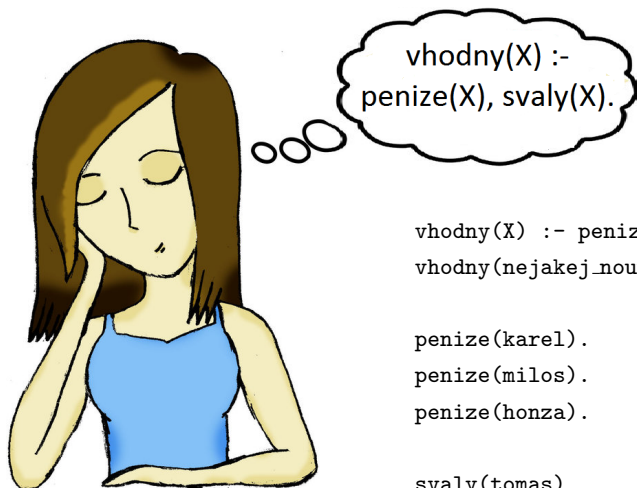
## Pozorování

- Při výpočtu Prolog vždy využívá fakta v tom pořadí, v jakém jsou uvedeny v programu.

## Příklad

- `players(flink,gta5).`  
`players(flink,world_of_tanks).`  
`players(flink,master_of_orion).`
- `?- players(flink,X).`  
`X = gta5 ;` `/* uživatel vložil ; */`  
`X = world_of_tanks ;` `/* uživatel vložil ; */`  
`X = master_of_orion.`
- `?- players(flink,X).`  
`X = gta5 .` `/* uživatel vložil . */`

# Příběh slečny Prology, aneb jak s pravidly



```
vhodny(X) :- penize(X), svaly(X).
vhodny(nejakej_nouma).
```

```
penize(karel).
penize(milos).
penize(honza).
```

```
svaly(tomas).
svaly(honza).
svaly(karel).
```

```
v(X) :- p(X), s(X).
```

```
v(nouma).
```

?- v(X)

```
p(karel).
```

```
p(milos).
```

```
p(honza).
```

```
s(tomas).
```

```
s(honza).
```

```
s(karel).
```

```
?-v(X).
```



# Příběh slečny Prology

```
v(X) :- p(X), s(X).
```

```
v(nouma).
```

```
p(karel).
```

```
p(milos).
```

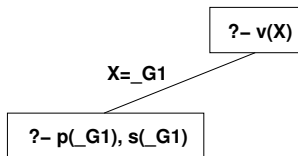
```
p(honza).
```

```
s(tomas).
```

```
s(honza).
```

```
s(karel).
```

```
?-v(X).
```



# Příběh slečny Prology

```
v(X) :- p(X), s(X).
```

```
v(nouma).
```

```
p(karel).
```

```
p(milos).
```

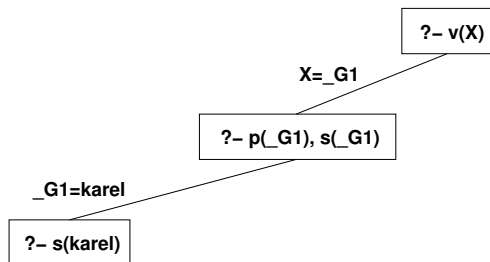
```
p(honza).
```

```
s(tomas).
```

```
s(honza).
```

```
s(karel).
```

```
?-v(X).
```



# Příběh slečny Prology

```
v(X) :- p(X), s(X).
```

```
v(nouma).
```

```
p(karel).
```

```
p(milos).
```

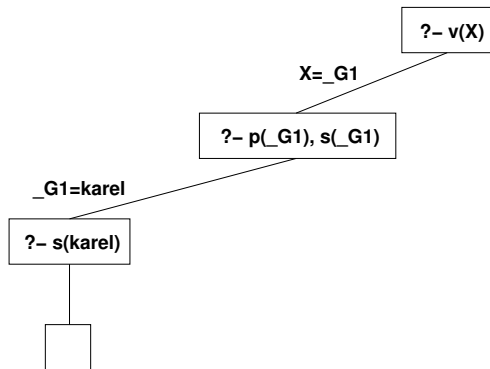
```
p(honza).
```

```
s(tomas).
```

```
s(honza).
```

```
s(karel).
```

```
?-v(X).
```



# Příběh slečny Prology

```
v(X) :- p(X), s(X).
```

```
v(nouma).
```

```
p(karel).
```

```
p(milos).
```

```
p(honza).
```

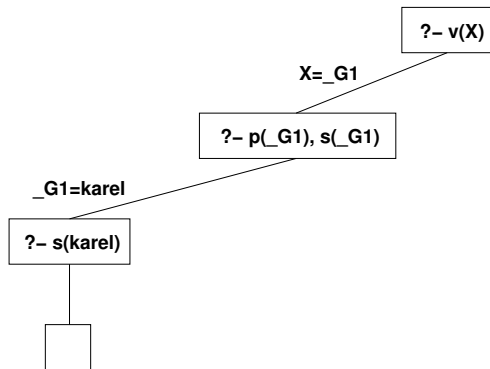
```
s(tomas).
```

```
s(honza).
```

```
s(karel).
```

```
?-v(X).
```

```
karel
```



# Příběh slečny Prology

```
v(X) :- p(X), s(X).
```

```
v(nouma).
```

```
p(karel).
```

```
p(milos).
```

```
p(honza).
```

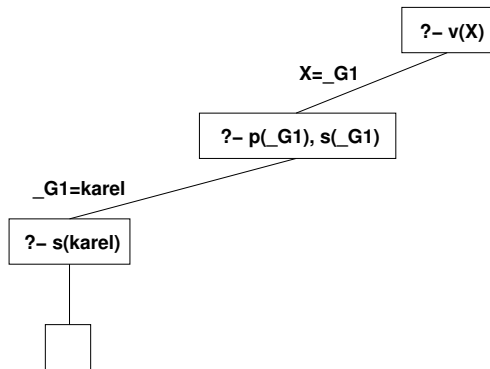
```
s(tomas).
```

```
s(honza).
```

```
s(karel).
```

```
?-v(X).
```

```
karel ; /* uživatel vložil ; */
```



# Příběh slečny Prology

```
v(X) :- p(X), s(X).
```

```
v(nouma).
```

```
p(karel).
```

```
p(milos).
```

```
p(honza).
```

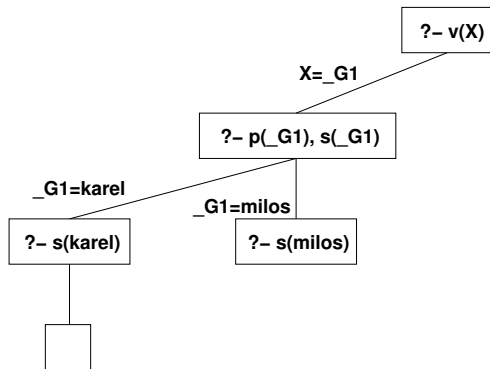
```
s(tomas).
```

```
s(honza).
```

```
s(karel).
```

```
?-v(X).
```

```
karel ; /* uživatel vložil ; */
```



# Příběh slečny Prology

```
v(X) :- p(X), s(X).
```

```
v(nouma).
```

```
p(karel).
```

```
p(milos).
```

```
p(honza).
```

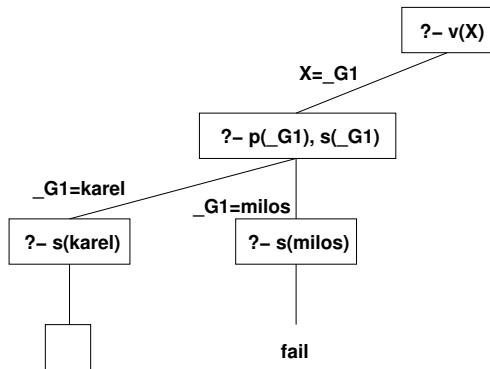
```
s(tomas).
```

```
s(honza).
```

```
s(karel).
```

```
?-v(X).
```

```
karel ; /* uživatel vložil ; */
```



# Příběh slečny Prology

```
v(X) :- p(X), s(X).
```

```
v(nouma).
```

```
p(karel).
```

```
p(milos).
```

```
p(honza).
```

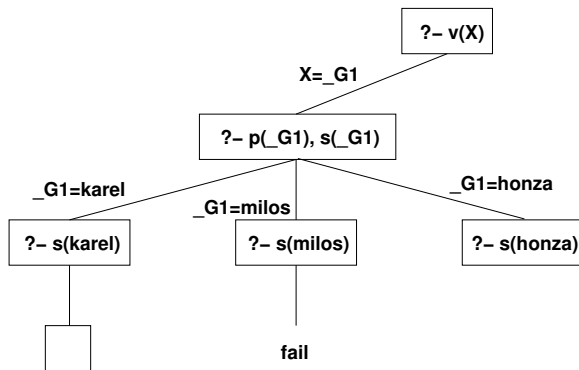
```
s(tomas).
```

```
s(honza).
```

```
s(karel).
```

```
?-v(X).
```

```
karel ; /* uživatel vložil ; */
```





# Příběh slečny Prology

```
v(X) :- p(X), s(X).
```

```
v(nouma).
```

```
p(karel).
```

```
p(milos).
```

```
p(honza).
```

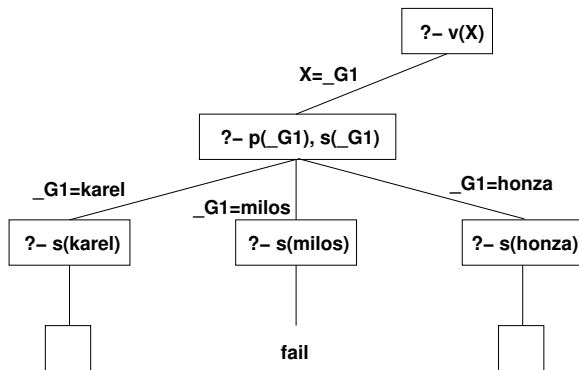
```
s(tomas).
```

```
s(honza).
```

```
s(karel).
```

```
?-v(X).
```

```
karel ; /* uživatel vložil ; */
```



# Příběh slečny Prology

```
v(X) :- p(X), s(X).
```

```
v(nouma).
```

```
p(karel).
```

```
p(milos).
```

```
p(honza).
```

```
s(tomas).
```

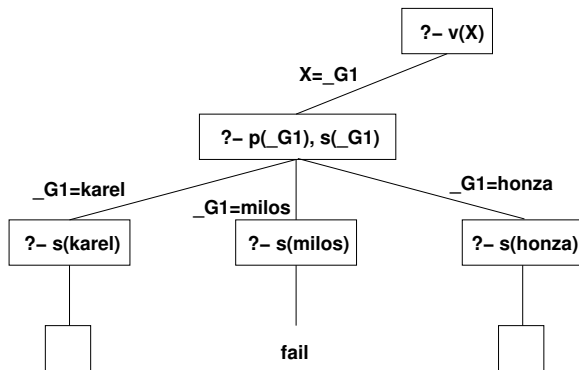
```
s(honza).
```

```
s(karel).
```

```
?-v(X).
```

```
karel ; /* uživatel vložil ; */
```

```
honza
```



# Příběh slečny Prology

```
v(X) :- p(X), s(X).
```

```
v(nouma).
```

```
p(karel).
```

```
p(milos).
```

```
p(honza).
```

```
s(tomas).
```

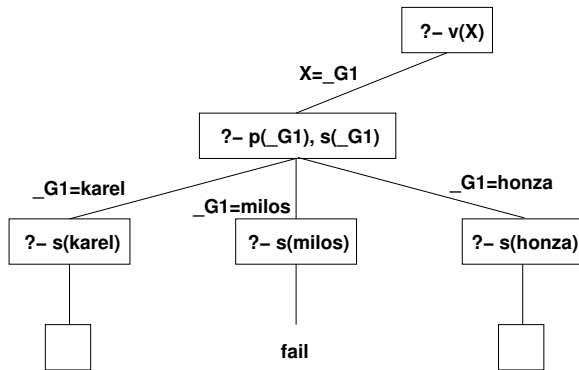
```
s(honza).
```

```
s(karel).
```

```
?-v(X).
```

```
karel ; /* uživatel vložil ; */
```

```
honza ; /* uživatel vložil ; */
```



# Příběh slečny Prology

```
v(X) :- p(X), s(X).
```

```
v(nouma).
```

```
p(karel).
```

```
p(milos).
```

```
p(honza).
```

```
s(tomas).
```

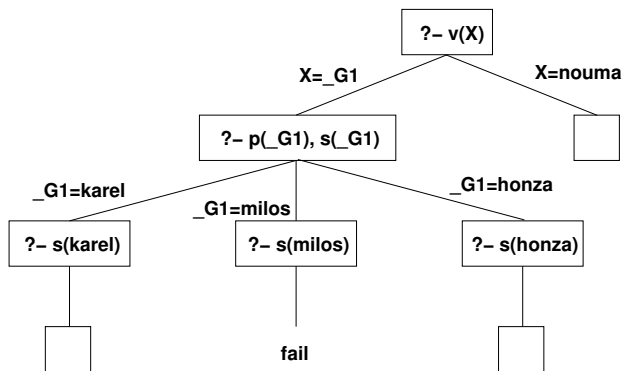
```
s(honza).
```

```
s(karel).
```

```
?-v(X).
```

```
karel ; /* uživatel vložil ; */
```

```
honza ; /* uživatel vložil ; */
```



# Příběh slečny Prology

```
v(X) :- p(X), s(X).
```

```
v(nouma).
```

```
p(karel).
```

```
p(milos).
```

```
p(honza).
```

```
s(tomas).
```

```
s(honza).
```

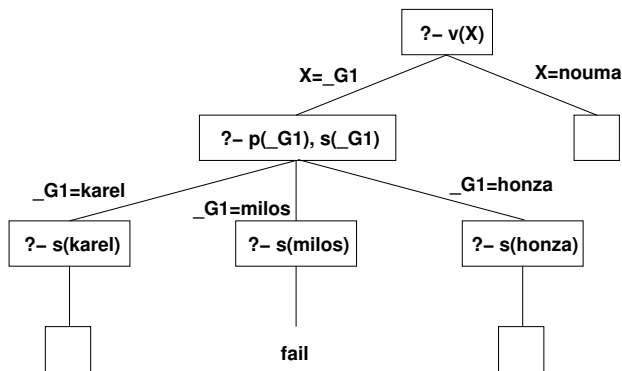
```
s(karel).
```

```
?-v(X).
```

```
karel ; /* uživatel vložil ; */
```

```
honza ; /* uživatel vložil ; */
```

```
nouma.
```



# Konstrukce výpočetního stromu

- Pro první podcíl v daném vrcholu se prohledává databáze faktů a pravidel vždy od začátku. Pro každou nalezenou vyhovující položku je vytvořen nový vrchol.
- Vrchol je vytvořen tak, že první podcíl se unifikuje s hlavou nalezené položky, a v nově vzniklém vrcholu je nahrazen tělem nalezené položky (fakta mají prázdné tělo, cíl je "vynechán").
- Hrana vedoucí do nového vrcholu je anotována unifikačním přiřazením. Proměnné vyskytující se v těle pravidla, které nejsou dotčeny unifikací, jsou označeny čerstvou proměnnou.
- Prázdné vrcholy (listy) značí úspěch, hodnoty hledaných proměnných vyplývají z unifikačních přiřazení na cestě od listu ke kořeni stromu.
- Vrcholy, pro které se nepodařilo nalézt položku v databázi vyhovující prvnímu podcíli jsou označeny podvrcholem **fail**.

?- f(G1)

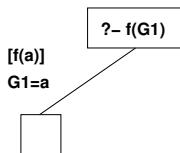
r(a,b).

r(a,c).

f(a).

f(X):-f(Y),r(Y,X).

# Výpočet s rekurzí



$r(a, b).$

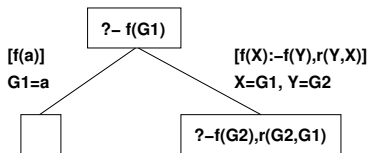
$r(a, c).$

$f(a).$

$f(X) :- f(Y), r(Y, X).$



# Výpočet s rekurzí



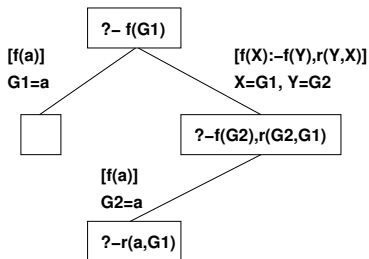
$r(a, b) .$

$r(a, c) .$

$f(a) .$

$f(X) :- f(Y), r(Y, X) .$

# Výpočet s rekurzí



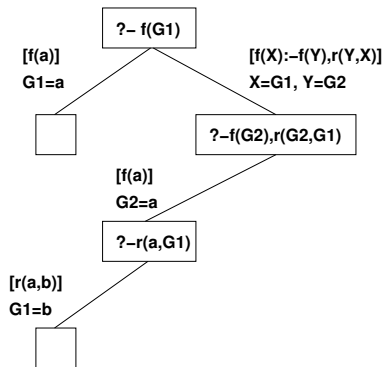
$r(a, b) .$

$r(a, c) .$

$f(a) .$

$f(X) :- f(Y), r(Y, X) .$

# Výpočet s rekurzí



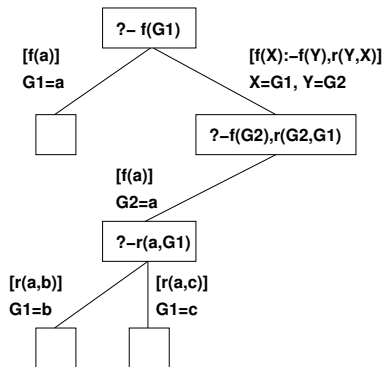
$r(a, b) .$

$r(a, c) .$

$f(a) .$

$f(X) :- f(Y), r(Y, X) .$

# Výpočet s rekurzí



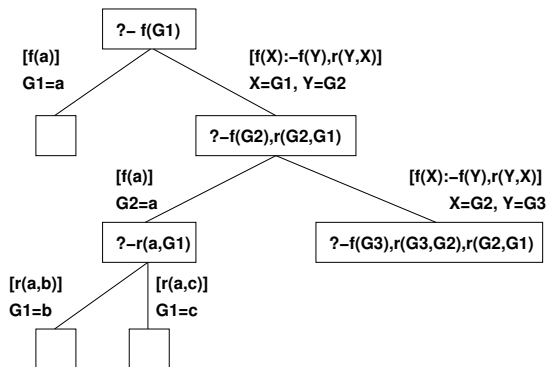
$r(a, b) .$

$r(a, c) .$

$f(a) .$

$f(X) :- f(Y), r(Y, X) .$

# Výpočet s rekurzí



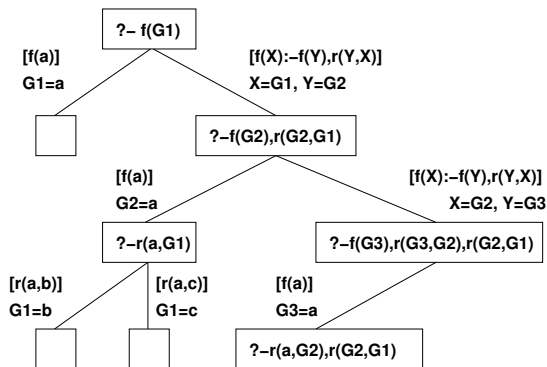
$r(a, b)$ .

$r(a, c)$ .

$f(a)$ .

$f(X) :- f(Y), r(Y, X)$ .

# Výpočet s rekurzí



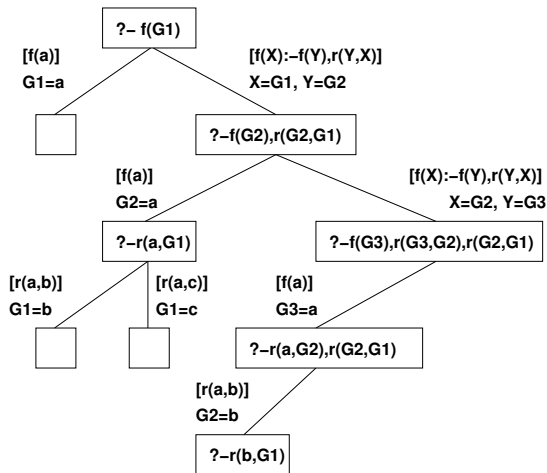
$r(a, b) .$

$r(a, c) .$

$f(a) .$

$f(X) :- f(Y), r(Y, X) .$

# Výpočet s rekurzí



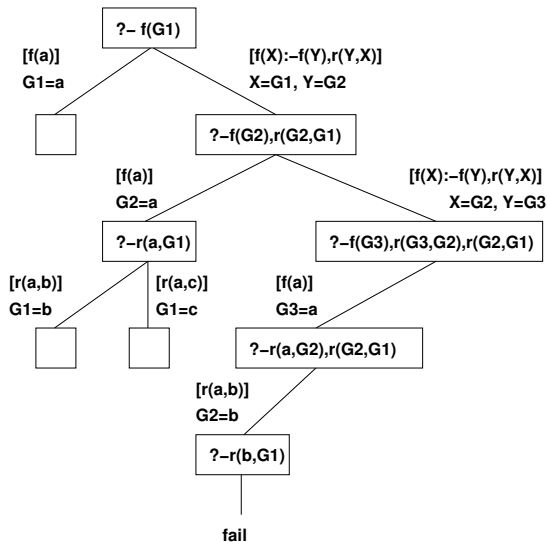
$r(a, b) .$

$r(a, c) .$

$f(a) .$

$f(X) :- f(Y), r(Y, X) .$

# Výpočet s rekurzí



$r(a, b)$ .

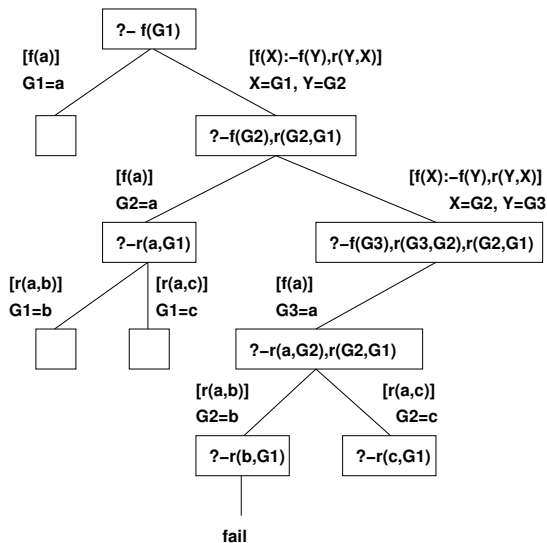
$r(a, c)$ .

$f(a)$ .

$f(X) :- f(Y), r(Y, X)$ .



# Výpočet s rekurzí



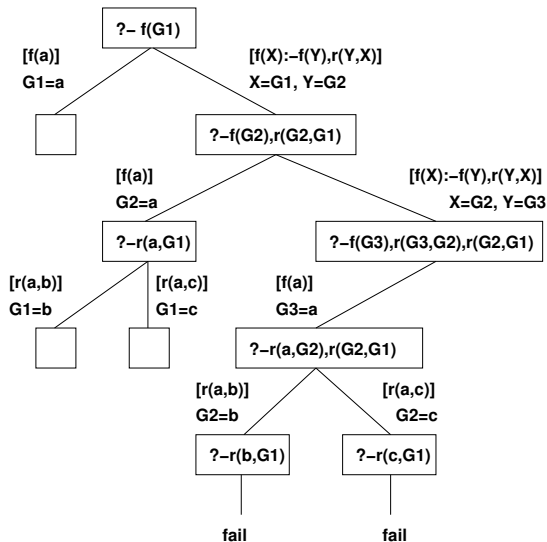
$r(a, b) .$

$r(a, c) .$

$f(a) .$

$f(X) :- f(Y), r(Y, X) .$

# Výpočet s rekurzí



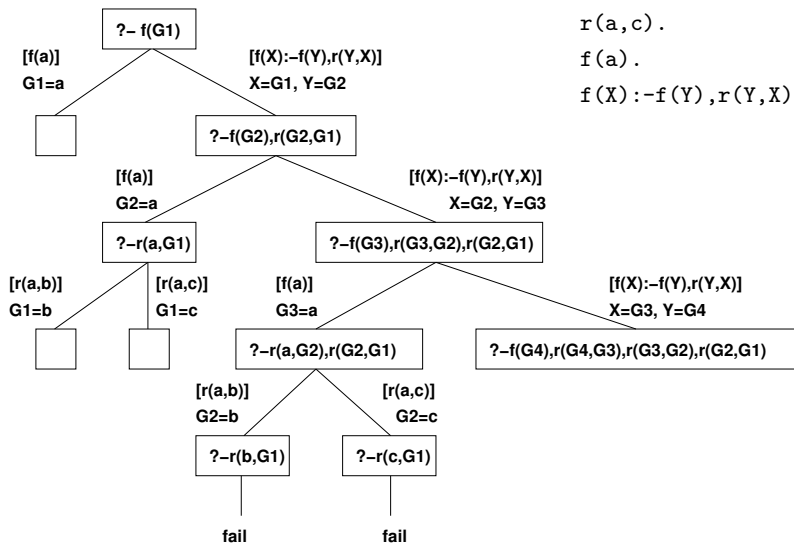
$r(a, b)$ .

$r(a, c)$ .

$f(a)$ .

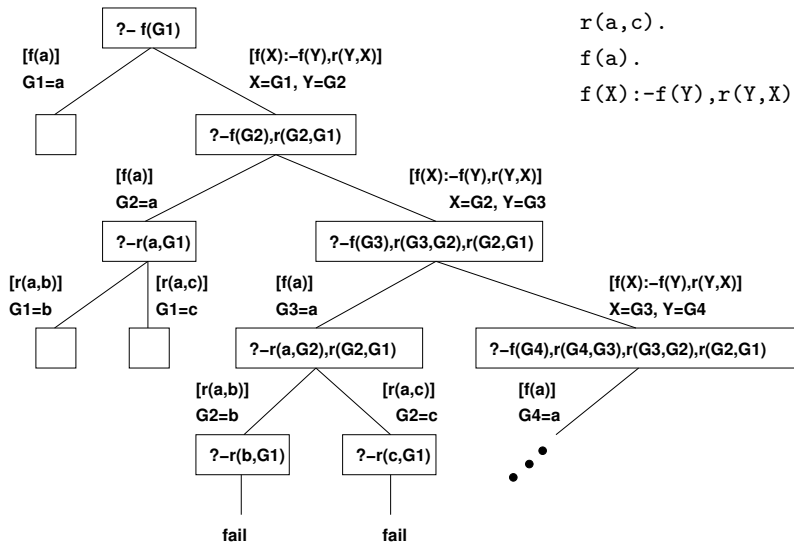
$f(X) :- f(Y), r(Y, X)$ .

# Výpočet s rekurzí



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

# Výpočet s rekurzí



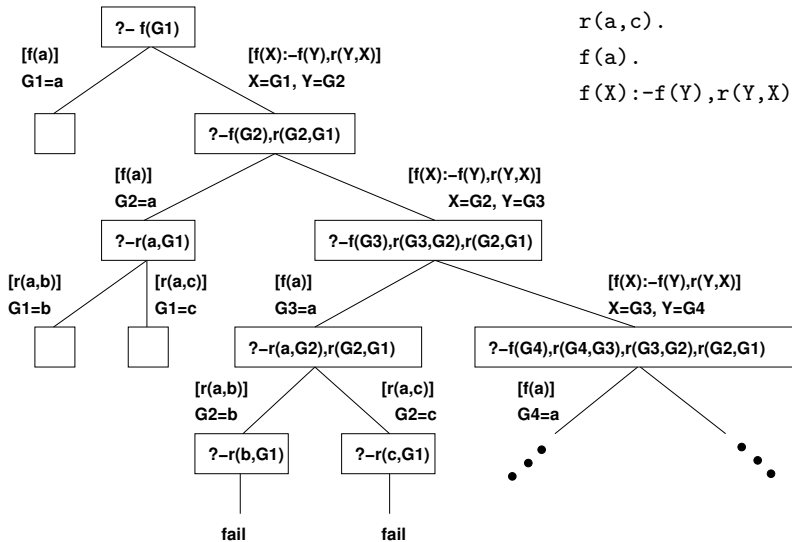
$r(a, b) .$

$r(a, c) .$

$f(a) .$

$f(X) :- f(Y), r(Y, X) .$

# Výpočet s rekurzí



## Pozorování

- Strom je procházen algoritmem prohledávání do hloubky.
- Výpočet nad nekonečnou větví stromu prakticky končí chybovou hláškou o nedostatečné velikosti paměti zásobníku.
- Použití levorekurzivních pravidel (první podcíl v těla pravidla je rekurzivní použití hlavy pravidla) často vede k nekonečné větvi, a to i v případě, kdy počet vyhovujících přiřazení na původní dotaz je konečný.
- Pořadí faktů a pravidel v databázi neovlivní počet úspěšných listů ve výpočetním stromu, ale ovlivní jejich umístění (tj. pořadí, ve kterém budou nalezeny a prezentovány uživateli.)

## Doporučení

- Používají se pravorekurzivní definice pravidel.
- Fakta se uvádějí před pravidly.

## Příklad

- S využitím znalostí prezentovaných v této přednášce naprogramujte nad databází měst:

```
mesto(prague).
```

```
mesto(viena).
```

```
mesto(warsaw).
```

```
mesto(roma).
```

```
mesto(paris).
```

```
mesto(madrid).
```

predikát `triMesta/3`, který je pravdivý pokud jako argumenty dostane tři různá města uvedené v databázi.

## Nekonečný výpočet

- Narhňte program v jazyce Prolog takový, aby ze zadání bylo zřejmé, že odpověď na určený dotaz je pravdivá, ovšem výpočet Prologu k tomuto výsledků nikdy nedospěje.