IB015 Neimperativní programování

Seznamy, Aritmetika a Efektivita výpočtu v Prologu

Jiří Barnat

Výpočty nad seznamy v Prologu

Seznam v Prologu

- Sekvence prvků (mohou být typově různorodé).
- Pro dekompozici na hlavu a tělo se používá znak `|`.
- Hlavu lze zobecnit na neprázdnou konečnou sekvenci prvků.

Příklady

- ```
  ?- [H|T] = [marek,matous,lukas,jan].
  H = marek,
  T = [matous, lukas, jan].
  ```
- ```
  ?- [X,Y|Z] = [1,2,3,4].
  X = 1,
  Y = 2,
  Z = [3, 4].
  ```
- ```
  ?- [X|Y|Z] = [1,2,3,4].
  ```
  ERROR
  ```

Anonymní proměnná

- Označená znakem podtržítko.
- Nelze se na ni odkázat v jiném místě programu.
- Při použití v unifikačním algoritmu neklade žádné omezení na kompatibilitu přiřazení hodnot jednotlivým proměnným.

Příklady unifikace s anonymní proměnnou

- `?- f(a,X)=f(X,b).` `false.` `?- f(a,X)=f(_,b).` `X = b.` `?- f(a,_)=f(_,b).` `true.`

```
?- f(a,X)=f(X,b).      ?- f(a,X)=f(_,b).      ?- f(a,_)=f(_,b).
false.                 X = b.                 true.
```

- Unifikací získejte 2. a 4. prvek seznamu `Seznam`:

```
[_,X,_,Y|_]= Seznam.
```

Pozorování

- Při vytváření programu v prologu, který má něco spočítat nebo vytvořit, postupujeme dle obecného pravidla transformace funkce `f(A) = B` na predikát `r(A,B)`.

Zadání

- Definujte predikát `a2b/2`, který transformuje seznam termů `a` na stejně dlouhý seznam termů `b`.

Příklad – rekurze na seznamech

Řešení

```
a2b([],[]).
a2b([a|Ta],[b|Tb]) :- a2b(Ta,Tb).
```

Použití

```
?- a2b([a,a,a,a],X).
X = [b,b,b,b].
```

```
?- a2b(X,[b,b,b,b]).
X = [a,a,a,a].
```

```
?- a2b(X,Y).
X = Y, Y = [] ;
X = [a], Y = [b] ;
X = [a, a], Y = [b, b] ;
X = [a, a, a], Y = [b, b, b].
```

Délka seznamu a test na bytí seznamem

length/2

- `length(L,I)` je pravda pokud délka seznamu `L` má hodnotu `I`.
- Prázdný seznam má délku 0.
- ```
  ?- length([a,ab,abc],X).
  X = 3.
  ```
- ```
  ?- length([[1,2,3]],X).
  X = 1.
  ```
- ```
  ?- length(X,2).
  X = [_G907, _G910].
  ```

Test na bytí seznamem

- `is_list/1` – Pravda, pokud parametr je seznam.
- ```
  ?- is_list( ['aha',[2,'b'],[],2.3] ).
  true.
  ```

Prvky seznamu – predikát member/2

member/2

- Zjištění přítomnosti prvku v seznamu.
- `member(X,L)` je pravda, pokud objekt X je prvkem seznamu L.
- Implementace:

```
member(X,[X|_]).
member(X,[_|T]) :- member(X,T).
```

Postup výpočtu:

- `?- member(lukas,[marek,matous,lukas,jan]).`

  $\rightsquigarrow$

  `?- member(lukas,[matous,lukas,jan]).`

  $\rightsquigarrow$

  `?- member(lukas,[lukas,jan]).`

  `true.`

Predikát member/2 – použití

Pozorování

- Volnou proměnnou lze použít jak v prvním, tak i v druhém argumentu. Pomocí predikátu je možné enumerovat prvky seznamu, ale také vynutit, že daný prvek je prvkem seznamu.

Příklady

- ```
  ?- member(X,[marek,matous,lukas,jan]).
  X = marek ;
  X = matous ;
  X = lukas ;
  X = jan ;
  false.
  ```
- ```
  ?- member(jan,[marek,matous,lukas,X]).
  X = jan ;
  false.
  ```

Konkatenace seznamů predikátem append

append/3

- Dotaz `append(L1,L2,L3)` se vyhodnotí na pravda, pokud seznam `L3` je zřetězením seznamů `L1` a `L2`.

Definice `append`

- ```
  append([],L,L).
  append([H1|T1],L2,[H1|T3]) :- append(T1,L2,T3).
  ```

Použití

- Test na to, zda $\texttt{L1} \cdot \texttt{L2} = \texttt{L3}$.
- Test na rovnost seznamů.
- Výpočet zřetězení dvou seznamů.
- Odvození prefixu, nebo sufixu v daném seznamu.
- Generování seznamů, jejichž zřetězení má daný výsledek.
  ```

Syntax se mění, myšlenka zůstává

Definice `append` v Prologu

```
append([],L,L).
append([H1|T1],L2,[H1|T3]) :- append(T1,L2,T3).
```

Definice `append` v Haskellu

```
append [] l = l
append (h1:t1) l2 = h1:t3 where t3 = (append t1 l2)
```

Řetězce v Prologu

Pozorování

- `'Ahoj'` není totéž co `"Ahoj"` .
- Řetězce znaků uvedených v uvozovkách jsou chápány jako seznamy čísel, které odpovídají ASCII kódům jednotlivých znaků řetězce.
- `"Ahoj" == [65,104,111,106].`

Automatická konverze na seznam čísel

- ```
  ?- append("Ale ","ne!",X).
  X = [65, 108, 101, 32, 110, 101, 33].
  ```

Konverze na řetězce

- Vybrané předdefinované predikáty vynutí prezentaci ve formě "řetězce".
  ```

Syntaktická ekvivalence (identita) - operátor ==

Syntaktická rovnost

- ```
  ?- 'pepa' == "Pepa".
  false.

  ?- 4 == 3+1.
  false.
  ```
  ```
  ?- 'mouka' == 'mouka'.
  true.

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

Pozor na syntaktické ekvivalenty

- ```
  ?- 'pepa' == pepa.
  true.

  ?- [97] == "a".
  true.
  ```

Aritmetika

SWI Prolog - Aritmetické typy

Celá čísla - integer

- Nativní typ, využívá knihovnu GMP.
- Velikost čísel omezena pouze velikostí dostupné paměti.

Desetinná čísla - float

- Nativní typ, odpovídá typu `double` z programovacího jazyka C.

Racionální čísla - rational

- Reprezentované s využitím složeného termu `rdiv(N,M)`.
- Výsledek vrácený operátorem `is/2` je kanonizován, tj. `M` je kladné a `M` a `N` nemají společného dělitele.

Konverze a unifikace

- `rdiv(N,1)` se konvertuje na celé číslo `N`.
- Automatické konverze ve směru od celých čísel k desetinným.
- Riziko vyvolání výjimky přetečení.
- Čísla různých typů se **neunifikují.**

Aritmetické funkce a operátory

Relační operátory

</2	menší než
>/2	větší než
=</2	menší nebo rovno
>=/2	větší nebo rovno
=:=/2	rovno
=\=/2	nerovno

Bitové operace

<</2	bitový posun vlevo
»/2	bitový posun vpravo
\//2	bitové OR
/\/2	bitové AND
\/1	bitová negace
xor/2	bitový XOR

Vybrané aritmetické funkce

-/1	unární mínus
+/1	znaménko plus
+/2	součet
-/2	rozdíl
*/2	součin
//2	dělení
**/2	mocnina
///2	celočíselné dělení
rem/2	zbytek po dělení //
div/2	dělení a zaokrouhlení dolů
mod/2	zbytek po dělení div
max/2	maximum
min/2	minimum
is/2	vyhodnocení a unifikace

Aritmetické operace v relacích

Pozorování

- Pro strukturovaný term, který dává do relace dva jiné termy, je možné nechat Prolog dohledat termy, pro které relace platí.
- ```
  rel(a,b).
  ?- rel(X,b).
  X = a.
  ```

Neplatí pro argumenty aritmetických operací

- Prolog při unifikaci a rezoluci nepočítá inverzní funkce, v okamžiku požadavku na takovouto operaci ohlásí interpret chybu (nedostatečná instanciace).
- Porovnejte:

`?- X is 3*3.`	`?- 9 is 3*X.`
`X = 9.`	**ERROR**
  ```

Příklady nedostatečné instanciace

Vyzkoušejte

- `?- 9 is X + 1.` **ERROR** `?- X > 3.` **ERROR** `?- X = 2, X > 3.` `false.`

Vysvětlete rozdílné chování

- Korektní definice predikátu pro výpočet délky seznamu.

```
length([],0).
length([_|T],N) :- length(T,X), N is X+1.
```

- Nevhodná definice predikátu pro výpočet délky seznamu.

```
length1([],0).
length1([_|T],N) :- N is X+1, length1(T,X).
```

- Rozdílné chování při výpočtu.

```
?- length([a,b],X).          ?- length1([a,b],X).
X = 2.                       ERROR
```

Instanciace parametrů

Pozorování

- Předdefinované predikáty mohou vyžadovat, aby některé parametry byly povinně instanciované, tzn. na jejich místě nelze použít proměnnou.

Používaná notace v dokumentaci

- +Arg: musí být instanciovaný parametr.
- -Arg: očekává se proměnná.
- ?Arg: instanciovaný parametr nebo proměnná.
- @Arg: parametr nebude vázán unifikací.
- :Arg: parametrem je název predikátu.

Módy použití

- Je-li binární predikát použit s dvěma instaciovanými parametry, říkáme, že predikát je použit v `(+,+)` módu.

Příklady z dokumentace

between(+Low, +High, ?Value)

- `Low and High are integers, High >=Low. If Value is an integer, Low=< Value=< High. When Value is a variable it is successively bound to all integers between Low and High. If High is inf or infinite between/3 is true iff Value>= Low, a feature that is particularly interesting for generating integers from a certain value.`

plus(?Int1, ?Int2, ?Int3)

- `True if Int3 =Int1 +Int2.` **At least two of the three arguments must be instantiated to integers.**

sort(+List, -Sorted)

- `True if Sorted can be unified with a list holding the elements of List, sorted to the standard order of terms. Duplicates are removed. The implementation is in C, using natural merge sort.`

  `The sort/2 predicate can sort a cyclic list, returning a non-cyclic version with the same elements.`

Efektivita – Tail rekurze

Rekurze a efektivita

Pozorování

- Uvažme následující definici predikátu `length`.

```
length([],0).
length([_|T],N) :- length(T,X), N is X+1.
```

- Nevýhodou této definice je, že při volání dochází k výpočtu před rekurzivním voláním (při rekurzivním sestupu) i po rekurzivním volání (při vynořování z rekurze).

Tail rekurze

- Definice, jež nevynucuje výpočet po rekurzivním volání, tj. rekurzivní cíl je jeden a je uveden jako poslední podcíl.
- Výsledek je znám při dosažení dna rekurzivního sestupu.
- Menší režie výpočtu, větší efektivita.
- Platí i ve světě imperativních programovacích jazyků.

Tail rekurze a akumulátory

Pozorování

- Tail-rekurzivní definice lze dosáhnout použitím akumulátoru.

Predikát `length` definován tail-rekurzivně

- Realizace pomocným predikátem s akumulátorem.

```
length(Seznam,Delka) :- accLen(Seznam,0,Delka).
```

- Definice pomocného predikátu.

```
accLen([],A,A).
accLen([_|T],A,L) :- B is A+1, accLen(T,B,L).
```

- Mód použití `accLen` je `(?,+,?)`.

Efektivita – Řez

Motivace pro operaci řezu

Pozorování

- Základem výpočtu logického programu je **backtracking**.
- Některé větve výpočtu nevedou k požadovanému cíli.
- Jistá kontrola nad způsobem prohledávání SLD stromu, by byla vhodná.

Dosavadní možnosti ovlivnění výpočtu

- Změna pořadí faktů v databázi.
- Změna pořadí podcílů v definici pravidla.

Vestavěný predikát ! (vykřičník)

Operátor řezu – `!/0`

- Vždy jako podcíl uspěje.
- Ovlivňuje způsob výpočtu (má vedlejší efekt).
- Eliminuje další volby, které by Prolog udělal při procházení výpočetního stromu, a to od okamžiku unifikace podcíle s levou stranou pravidla, ve kterém se predikát `!` vyskytuje, až do místa výskytu `!`.

Důsledky vedlejšího efektu

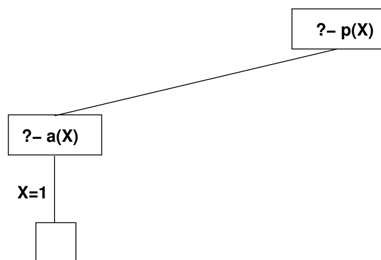
- Prořezává výpočetní strom.
- Rychlejší výpočet.
- Riziko odřezání větví výpočtu, které vedou k dalším (stejným, či jiným) řešením.

Příklad fungování řezu – bez řezu

?– p(X)

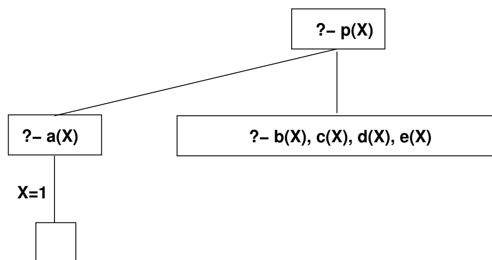
```
p(X) :- a(X).
p(X) :- b(X),c(X),
        d(X),e(X).
p(X) :- f(X).

a(1).
b(1). c(1).
d(1). d(2). e(2).
f(3).
b(2). c(2).
```

Příklad fungování řezu – bez řezu



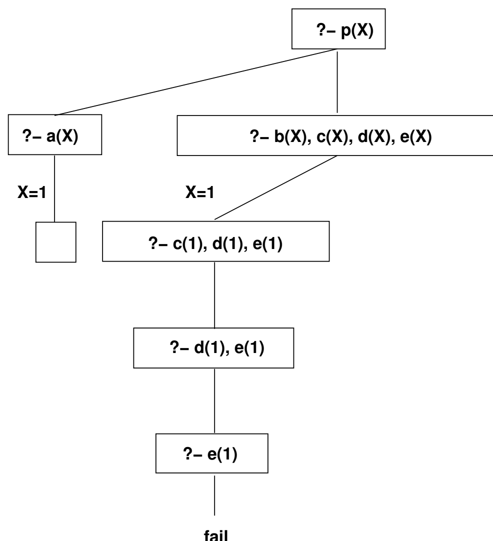

```
p(X) :- a(X).
p(X) :- b(X),c(X),
        d(X),e(X).
p(X) :- f(X).

a(1).
b(1). c(1).
d(1). d(2). e(2).
f(3).
b(2). c(2).
```

Příklad fungování řezu – bez řezu



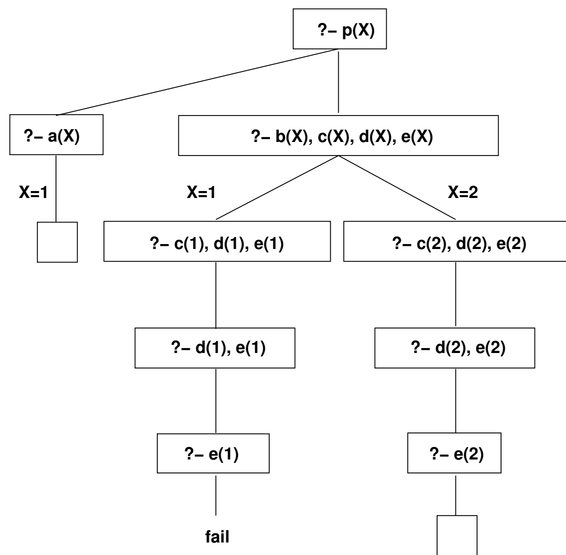

```
p(X) :- a(X).
p(X) :- b(X),c(X),
        d(X),e(X).
p(X) :- f(X).

a(1).
b(1). c(1).
d(1). d(2). e(2).
f(3).
b(2). c(2).
```

Příklad fungování řezu – bez řezu



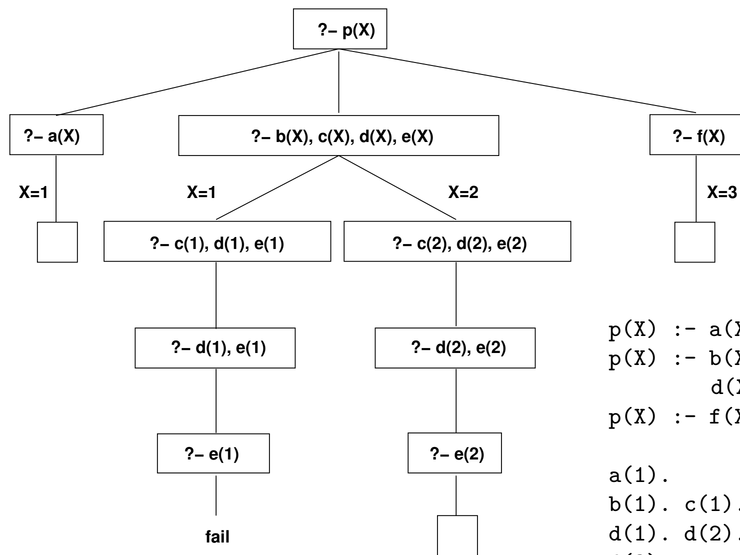

```
p(X) :- a(X).
p(X) :- b(X),c(X),
        d(X),e(X).
p(X) :- f(X).

a(1).
b(1). c(1).
d(1). d(2). e(2).
f(3).
b(2). c(2).
```

Příklad fungování řezu – bez řezu



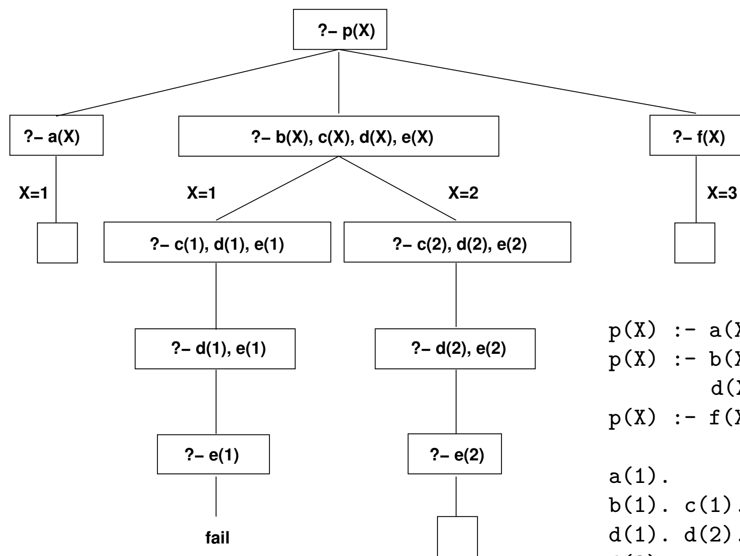

```
p(X) :- a(X).
p(X) :- b(X),c(X),
        d(X),e(X).
p(X) :- f(X).

a(1).
b(1). c(1).
d(1). d(2). e(2).
f(3).
b(2). c(2).
```

Příklad fungování řezu – bez řezu



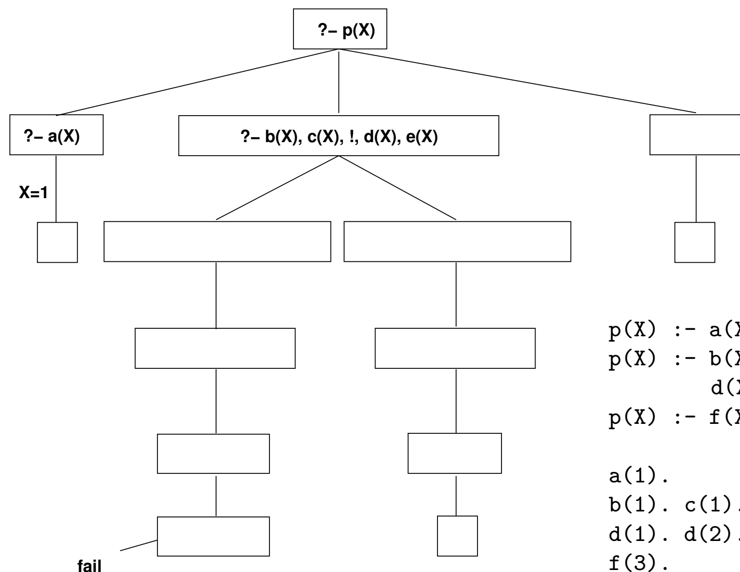

```
p(X) :- a(X).
p(X) :- b(X),c(X),
        d(X),e(X).
p(X) :- f(X).

a(1).
b(1). c(1).
d(1). d(2). e(2).
f(3).
b(2). c(2).
```

Příklad fungování řezu – s řezem



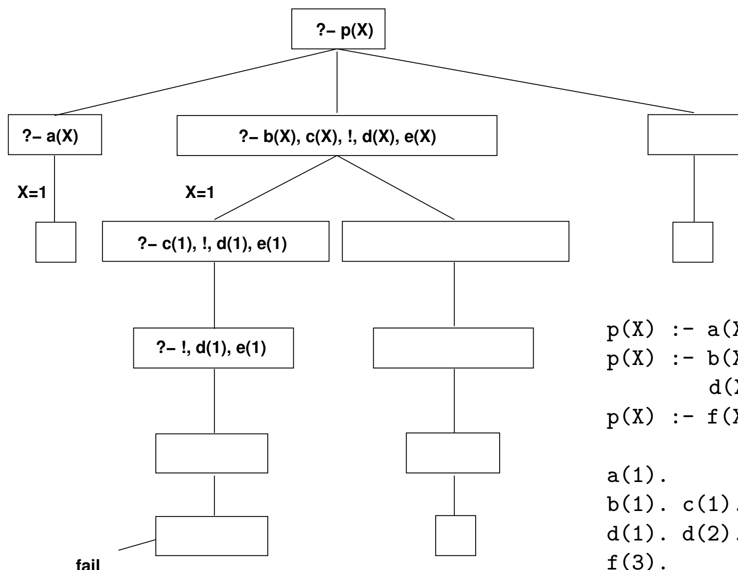

```
p(X) :- a(X).
p(X) :- b(X),c(X), !,
        d(X),e(X).
p(X) :- f(X).

a(1).
b(1). c(1).
d(1). d(2). e(2).
f(3).
b(2). c(2).
```

Příklad fungování řezu – s řezem



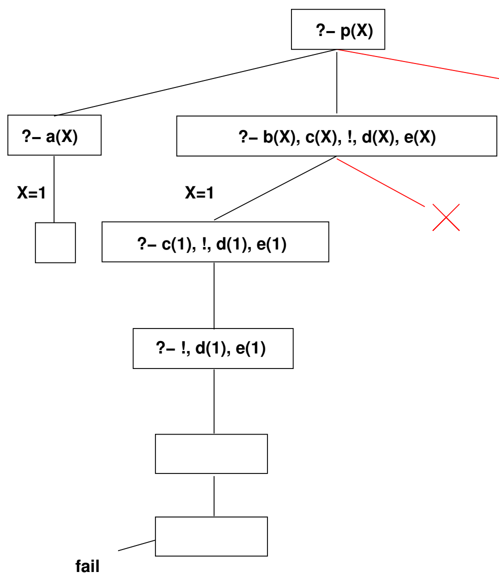

```
p(X) :- a(X).
p(X) :- b(X),c(X), !,
        d(X),e(X).
p(X) :- f(X).

a(1).
b(1). c(1).
d(1). d(2). e(2).
f(3).
b(2). c(2).
```

Příklad fungování řezu – s řezem



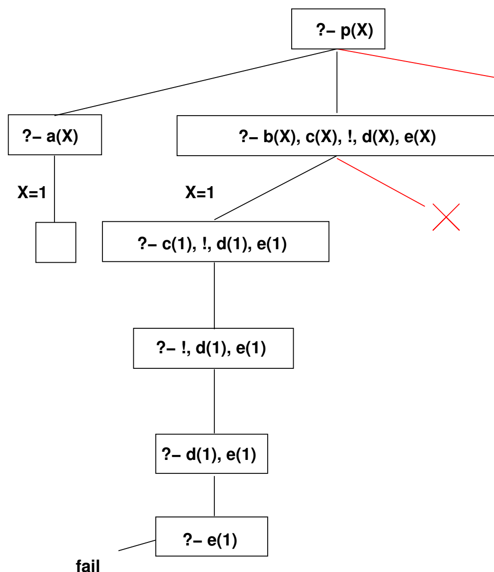

```
p(X) :- a(X).
p(X) :- b(X),c(X), !,
        d(X),e(X).
p(X) :- f(X).

a(1).
b(1). c(1).
d(1). d(2). e(2).
f(3).
b(2). c(2).
```

Příklad fungování řezu – s řezem




```
p(X) :- a(X).
p(X) :- b(X),c(X), !,
        d(X),e(X).
p(X) :- f(X).

a(1).
b(1). c(1).
d(1). d(2). e(2).
f(3).
b(2). c(2).
```

Příklad fungování řezu – s řezem




```
p(X) :- a(X).
p(X) :- b(X),c(X), !,
        d(X),e(X).
p(X) :- f(X).

a(1).
b(1). c(1).
d(1). d(2). e(2).
f(3).
b(2). c(2).
```

Vedlejší efekty – Upnutí

Popis

- Pokud se při řešení podcíle narazí v těle pravidla na operátor `!`, ostatní fakta a pravidla, se pro právě řešený cíl (ten, který se unifikoval s hlavou pravidla) neberou v potaz.

Příklad

- Porovnej chování následujících programů.

a)
```
a(X) :- X = 1.
a(X) :- X = 2.

?- a(X).
X = 1 ;
X = 2.
```

b)
```
a(X) :- X = 1, !.
a(X) :- X = 2.

?- a(X).
X = 1.
```

Popis

- Pokud se při řešení podcíle narazí v těle pravidla na operátor řezu, všechny unifikace vyplývající z podcílů vyskytujících se v těle pravidla před operátorem `!` se fixují (jiné možnosti unifikace těchto podcílů se neuvažují).

Porovnejte

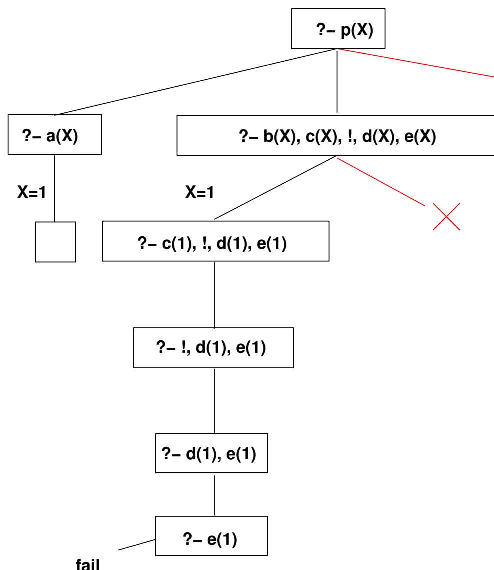
a)

```
a(X) :- X = 0.
a(X) :- X = 1.
b(X,Y) :- a(X), a(Y).

?- b(X,Y).
X = 0, Y = 0 ;
X = 0, Y = 1 ;
X = 1, Y = 0 ;
X = 1, Y = 1.
```

b)

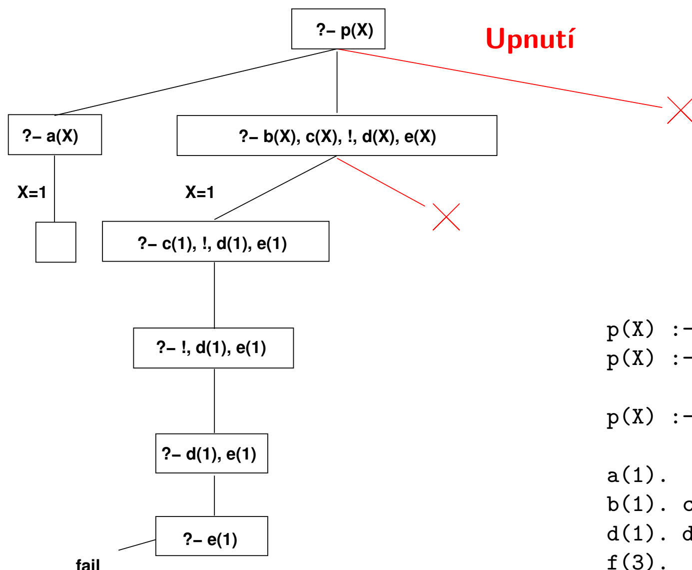
```
a(X) :- X = 0.
a(X) :- X = 1.
b(X,Y) :- a(X), !, a(Y).

?- b(X,Y).
X = 0, Y = 0 ;
X = 0, Y = 1.
```

Příklad fungování řezu – vedlejší efekty



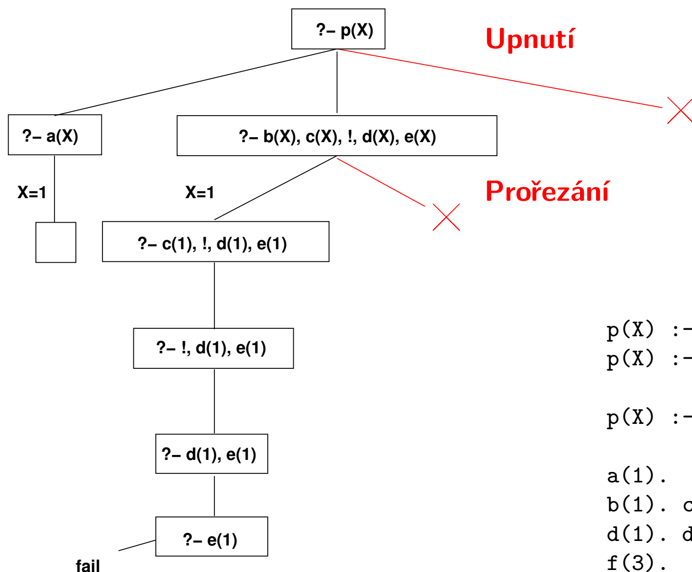

```
p(X) :- a(X).
p(X) :- b(X),c(X), !,
        d(X),e(X).
p(X) :- f(X).

a(1).
b(1). c(1).
d(1). d(2). e(2).
f(3).
b(2). c(2).
```

Příklad fungování řezu – vedlejší efekty




```
p(X) :- a(X).
p(X) :- b(X),c(X), !,
        d(X),e(X).
p(X) :- f(X).

a(1).
b(1). c(1).
d(1). d(2). e(2).
f(3).
b(2). c(2).
```

Příklad fungování řezu – vedlejší efekty




```
p(X) :- a(X).
p(X) :- b(X),c(X), !,
        d(X),e(X).
p(X) :- f(X).

a(1).
b(1). c(1).
d(1). d(2). e(2).
f(3).
b(2). c(2).
```

Příklad

Úkol

- Určete maximální možný výstup interpretru pro následující kód v Prologu na dotaz `?- b(X,Y).`

Zadání 1

```
a(X) :- X = 0.
a(X) :- X = 1,!.
a(X) :- X = 2.
b(X,Y) :- a(X),a(Y).
```

Příklad

Úkol

- Určete maximální možný výstup interpretru pro následující kód v Prologu na dotaz `?- b(X,Y).`

Zadání 1

```
a(X) :- X = 0.
a(X) :- X = 1,!.
a(X) :- X = 2.
b(X,Y) :- a(X),a(Y).
```

Řešení 1

```
X = 0, Y = 0 ;
X = 0, Y = 1 ;
X = 1, Y = 0 ;
X = 1, Y = 1.
```

Příklad

Úkol

- Určete maximální možný výstup interpretru pro následující kód v Prologu na dotaz `?- b(X,Y).`

Zadání 1

```
a(X) :- X = 0.
a(X) :- X = 1,!.
a(X) :- X = 2.
b(X,Y) :- a(X),a(Y).
```

Řešení 1

```
X = 0, Y = 0 ;
X = 0, Y = 1 ;
X = 1, Y = 0 ;
X = 1, Y = 1.
```

Zadání 2

```
a(X) :- X = 0.
a(X) :- X = 1,!.
a(X) :- X = 2.
b(X,Y) :- a(X), !, a(Y).
```

Příklad

Úkol

- Určete maximální možný výstup interpretru pro následující kód v Prologu na dotaz `?- b(X,Y).`

Zadání 1

```
a(X) :- X = 0.
a(X) :- X = 1,!.
a(X) :- X = 2.
b(X,Y) :- a(X),a(Y).
```

Řešení 1

```
X = 0, Y = 0 ;
X = 0, Y = 1 ;
X = 1, Y = 0 ;
X = 1, Y = 1.
```

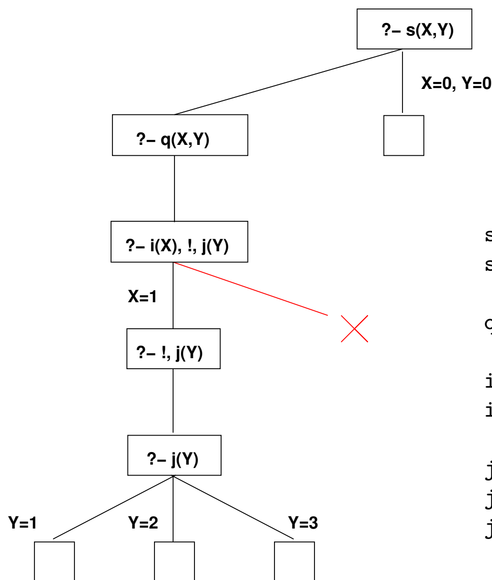
Zadání 2

```
a(X) :- X = 0.
a(X) :- X = 1,!.
a(X) :- X = 2.
b(X,Y) :- a(X), !, a(Y).
```

Řešení 2

```
X = 0, Y = 0 ;
X = 0, Y = 1.
```

Příklad fungování řezu – imunita nadřazených cílů




```
s(X,Y):- q(X,Y).
s(0,0).

q(X,Y):- i(X), !, j(Y).

i(1).
i(2).

j(1).
j(2).
j(3).
```

Typy řezů

Zelené řezy

- Odstraněním operátoru řezu se nemění sémantika programu (množina řešení po odstranění řezu je shodná).
- Řez je použit pouze z důvodů efektivity.
- Někdy se jako„modré“ označují řezy eliminující duplicity.

Červené řezy

- Odstraněním operátoru řezu se mění sémantika programu (po odstranění řezu, je možné nalézt další jiná řešení).

Obecná doporučovaná strategie

- Vyrobit funkční řešení bez řezů.
- Zvýšit efektivitu použitím „zelených“ řezů.
- Využít „červené řezy“ pouze pokud není vyhnutí, dobře okomentovat.

Checkpoint

Zadání

- V Prologu naprogramujte predikáty `encodeRLE/2` a `decodeRLE/2`, které budou realizovat RLE kódování seznamů.
- V RLE kódování je každá `n`-tice stejných po sobě jdoucích prvků `k` v seznamu nahrazena uspořádanou dvojicí `(n,k)`.

Příklad použití

- ```
  ?- encodeRLE([a,a,a,b,b,c,d,d,e],X).
  X = [(3,a),(2,b),(1,c),(2,d),(1,e)]
  ```
- ```
  ?- decodeRLE([(5,1),(1,2),(3,3)],[1,1,1,1,1,2,3,3,3]).
  true.
  ```