

Seznamy, řez

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
```

Předchudce a následník prvku X v seznamu S

```
hldej(S,X,Pred,Po) :- append( _S1, [ Pred,X,Po | _S2 ], S)
```

Reprezentace seznamu

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

Seznamy a append

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

Napište následující predikáty pomocí append/3:

- `prefix(S1, S2) :- append(S1, _S3, S2).`
DÚ: `suffix(S1,S2)`
- `last(X, S) :- append(_S1, [X], S).`
`append([3,2], [6], [3,2,6]).` X=6, S=[3,2,6]
- `member(X, S) :- append(S1, [X|S2], S).`
`append([3,4,1], [2,6], [3,4,1,2,6]).` X=2, S=[3,4,1,2,6]
DÚ: `adjacent(X,Y,S)`
- `% sublist(+S,+ASB)`
`sublist(S,ASB) :- append(AS, B, ASB),`
`append(A, S, AS).`

POZOR na efektivitu, bez append lze často napsat efektivněji

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**

Akumulátor a `sum_list(S, Sum)`

?- `sum_list([2,3,4], Sum)`.

bez akumulátoru:

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

```
sum_list( [H|T], Sum ) :- sum_list( T, SumT ),
                        Sum is H + SumT.
```

s akumulátorem:

```
sum_list( S, Sum ) :- sum_list( S, 0, Sum ).
```

```
sum_list( [], Sum, Sum ).
```

```
sum_list( [H|T], A, Sum ) :- A1 is A + H,
                            sum_list( T, A1, Sum).
```

Výpočet faktoriálu `fact(N, F)`

s akumulátorem:

```
fact( N, F ) :- fact( N, 1, F ).
```

```
fact( 1, F, F ) :- !.
```

```
fact( N, A, F ) :- N > 1,
                  A1 is N * A,
                  N1 is N - 1,
                  fact( N1, A1, F ).
```

```

r(X):-write(r1).
r(X):-p(X),write(r2).
r(X):-write(r3).

p(X):-write(p1).
p(X):-a(X),b(X),!,
    c(X),d(X),write(p2).
p(X):-write(p3).

a(X):-write(a1).
a(X):-write(a2).

b(X):- X > 0, write(b1).
b(X):- X < 0, write(b2).

c(X):- X mod 2 == 0, write(c1).
c(X):- X mod 3 == 0, write(c2).

d(X):- abs(X) < 10, write(d1).
d(X):- write(d2).

```

Prozkoumejte trasy výpočtu a navracení např. pomocí následujících dotazů (vždy si středníkem vyžádejte navracení):

(1) $X=1, r(X)$. (2) $X=3, r(X)$.
(3) $X=0, r(X)$. (4) $X=-6, r(X)$.

- řez v predikátu p/1 neovlivní alternativy predikátu r/1
- dokud nebyl proveden řez, alternativy predikátu a/1 se uplatňují, př. neúspěch b/1 v dotazu (3)
- při neúspěchu cíle za řezem se výpočet navrácí až k volající proceduře r/1, viz (1)
- alternativy vzniklé po provedení řezu se zachovávají - další možnosti predikátu c/1 viz (2) a (4)

Řez: maximum

Je tato definice predikátu max/3 korektní?

```

max(X,Y,X):-X>=Y,!.
max(X,Y,Y).

```

Není, následující dotaz uspěje: $?- \text{max}(2,1,1)$.

Uved'te dvě možnosti opravy, se zachováním použití řezu a bez.

```

max(X,Y,X):-X>=Y.
max(X,Y,Y):-Y>X.

max(X,Y,Z):-X>=Y,!,Z=X.
max(X,Y,Y).

```

Problém byl v definici, v první klauzuli se tvrdilo: $X=Z \wedge X>=Y \Rightarrow \text{true}$
správná definice je: $X>=Y \Rightarrow Z=X$

Při použití řezu je třeba striktně oddělit vstupní podmínky od výstupních unifikací a výpočtu.

```

r(X):-write(r1).
r(X):-p(X),write(r2).
r(X):-write(r3).

p(X):-write(p1).
p(X):-a(X),b(X),!,
    c(X),d(X),write(p2).
p(X):-write(p3).

a(X):-write(a1).
a(X):-write(a2).

b(X):- X > 0, write(b1).
b(X):- X < 0, write(b2).

c(X):- X mod 2 == 0, write(c1).
c(X):- X mod 3 == 0, write(c2).

d(X):- abs(X) < 10, write(d1).
d(X):- write(d2).

| ?- X=1,r(X).
r1
X = 1 ? ;
p1r2
X = 1 ? ;
a1b1r3
X = 1 ? ;
no

| ?- X=3,r(X).
r1
X = 3 ? ;
p1r2
X = 3 ? ;
a1b1c2d1p2r2
X = 3 ? ;
d2p2r2
X = 3 ? ;
r3
X = 3 ? ;
no

| ?- X=-6,r(X).
r1
X = -6 ? ;
p1r2
X = -6 ? ;
a1b2c1d1p2r2
X = -6 ? ;
d2p2r2
X = -6 ? ;
c2d1p2r2
X = -6 ? ;
d2p2r2
X = -6 ? ;
r3
X = -6 ? ;
no

| ?- X=0,r(X).
r1
X = 0 ? ;
p1r2
X = 0 ? ;
a1a2p3r2
X = 0 ? ;
r3
X = 0 ? ;
no

```

Řez: member

Jaký je rozdíl mezi následujícími definicemi predikátů member/2. Ve kterých odpovědích se budou lišit? Vyzkoušejte např. pomocí $\text{member}(X, [1,2,3])$.

```

mem1(H, [H|_]).
mem1(H, [_|T]) :- mem1(H,T).

mem2(H, [H|_]) :- !.
mem2(H, [_|T]) :- mem2(H,T).

mem3(H, [K|_]) :- H==K.
mem3(H, [K|T]) :- H\==K, mem3(H,T).

```

- mem1/2 vyhledá všechny výskyty, při porovnávání hledaného prvku s prvky seznamu může dojít k vázání proměnných (může sloužit ke generování všech prvků seznamu)
- mem2/2 najde jenom první výskyt, taky váže proměnné
- mem3/2 najde jenom první výskyt, proměnné neváže (hledá pouze identické prvky)

Dokážete napsat variantu, která hledá jenom identické prvky a přitom najde všechny výskyty? $\text{mem4}(H,[K|_]) :- H==K. \text{mem4}(H,[K|T]) :- \text{mem4}(H,T).$

Řez: delete

```
delete( X, [X|S], S ).
```

```
delete( X, [Y|S], [Y|S1] ) :- delete(X,S,S1).
```

Napište predikát `delete(X,S,S1)`, který odstraní všechny výskyty `X` (pokud se `X` v `S` nevyskytuje, tak predikát uspěje).

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

```
delete( X, [X|S], S1 ) :- !, delete(X,S,S1).
```

```
delete( X, [Y|S], [Y|S1] ) :- delete(X,S,S1).
```

Seznamy: intersection(A,B,C)

DÚ: Napište predikát pro výpočet průniku dvou seznamů.

Nápověda: využijte predikát `member/2`

DÚ: Napište predikát pro výpočtu rozdílu dvou seznamů. Nápověda: využijte predikát `member/2`