

Seznamy (pokračování)

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
```

Hana Rudová, Logické programování I, 14. března 2011

2

Seznamy

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, Délka)`

```
length( [], 0 ).
length( [ H | T ], Délka ) :- length( T, Délka0 ), Délka is 1 + Délka0.
```
- Upravená procedura, tak aby umožnila LCO:

```
% length( Seznam, ZapocitanaDélka, CelkovaDélka ):
%      CelkovaDélka = ZapocitanaDélka + ,,počet prvků v Seznam''

length( Seznam, Délka ) :- length( Seznam, 0, Délka ). % pomocný predikát
length( [], Délka, Délka ).                          % celková délka = započítaná délka
length( [ H | T ], A, Délka ) :- A0 is A + 1, length( T, A0, Délka ).
```
- 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) ).
```

Neefektivita při spojování seznamů

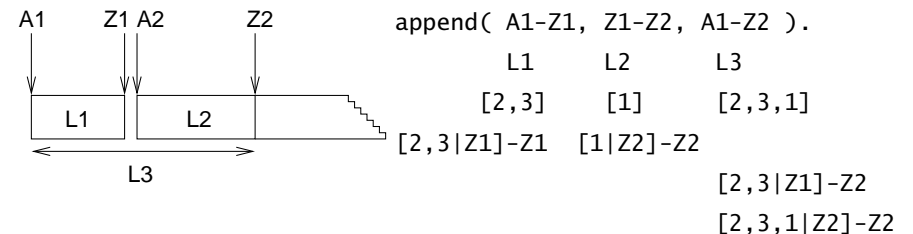
- Sjednocení dvou seznamů
- `append([], S, S).`
`append([X|S1], S2, [X|S3]) :- append(S1, S2, S3).`
- `?- append([2,3], [1], S).`
postupné volání cílů:
`append([2,3], [1], S) → append([3], [1], S') → append([], [1], S'')`
- Vždy je nutné projít celý první seznam

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`)

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 ).
```

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

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

kvadratická složitost

akumulátor (lineární)

rozdílové seznamy

(lineární)

Příklad: operace pro manipulaci s frontou

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

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?, ...

Vestavěné predikáty

Databázové operace

- Databáze: specifikace množiny relací
- Prologovský program: **programová databáze**, kde jsou relace specifikovány explicitně (fakty) i implicitně (pravidly)
- Vestavěné predikáty pro změnu databáze během provádění programu:
 - assert(Klauzule) přidání Klauzule do programu
 - asserta(Klauzule) přidání na začátek
 - assertz(Klauzule) přidání na konec
 - retract(Klauzule) smazání klauzule unifikovatelné s Klauzule
- Pozor: nadměrné použití těchto operací snižuje srozumitelnost programu

Příklad: databázové operace

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

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

Vstup a výstup

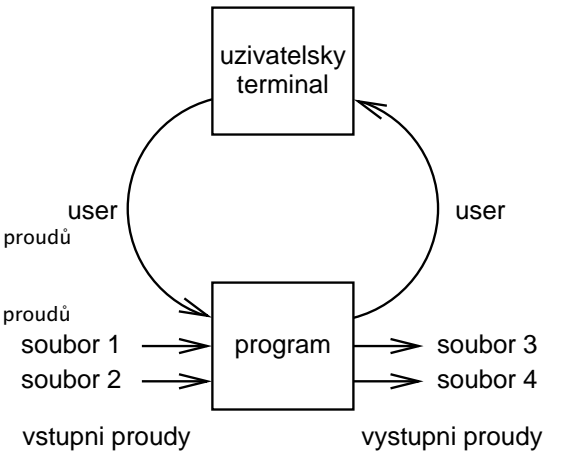
- program může číst data ze **vstupního proudu** (*input stream*)
- program může zapisovat data do **výstupního proudu** (*output stream*)

- dva **aktivní proudy**

- aktivní vstupní proud
- aktivní výstupní proud

- **uživatelský terminál** – user

- datový vstup z terminálu
chápán jako jeden ze vstupních proudů
- datový výstup na terminál
chápán jako jeden z výstupních proudů



Vstupní a výstupní proudy: vestavěné predikáty

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

```
cteni( Soubor ) :- see( Soubor ),  
                  cteni_ze_souboru( Informace ),  
                  see( user ).
```
- **uzavření** aktivního vstupního/výstupního proudu: `seen/told`
- **zjištění** aktivního vstupního/výstupního proudu: `seeing(S)/telling(S)`

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

Sekvenční přístup k textovým souborům

- **čtení dalšího termu:** `read(Term)`
 - při čtení jsou termy odděleny tečkou
 - | `?- read(A), read(ahoj(B)), read([C,D]).`
 - |: `ahoj. ahoj(petre). [ahoj('Petre!'), jdeme].`
 - A = ahoj, B = petre, C = ahoj('Petre!'), D = jdeme
 - po dosažení konce souboru je vrácen atom `end_of_file`
 - **zápis dalšího termu:** `write(Term)`
 - `?- write(ahoj).` `?- write('Ahoj Petre!').`
- nový řádek na výstup: `n`
N mezer na výstup: `tab(N)`- **čtení/zápis dalšího znaku:** `get0(Znak), get(NeprazdnyZnak)/put(Znak)`
 - po dosažení konce souboru je vrácena `-1`

Příklad čtení ze souboru

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

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

Čtení programu ze souboru

- **Interpretování** kódu programu
 - ?- consult(program).
 - ?- consult('program.pl').
 - ?- consult([program1, 'program2.pl']).
- **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