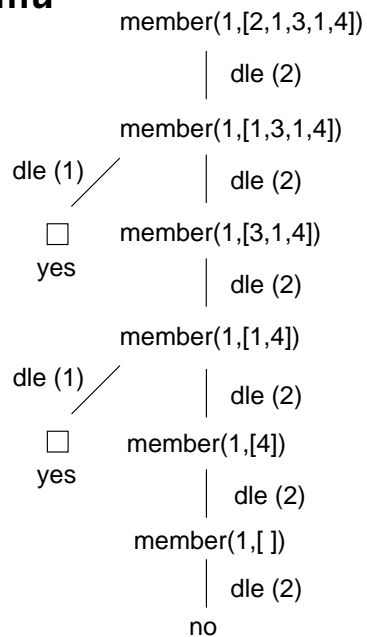


Seznamy

Prvek seznamu

- $\text{member}(X, S)$
- platí: $\text{member}(b, [a,b,c])$.
- neplatí: $\text{member}(b, [[a,b] | [c]])$.
- X je prvek seznamu S , když
 - X je hlava seznamu S nebo
 $\text{member}(X, [X | _])$. % (1)
 - X je prvek těla seznamu S
 $\text{member}(X, [_ | \text{Te}lo]) :- \text{member}(X, \text{Te}lo)$. % (2)
- Další příklady použití:
 - $\text{member}(X, [1,2,3])$.
 - $\text{member}(1, [2,1,3,1])$.



Reprezentace seznamu

- **Seznam:** $[a, b, c]$, prázdný seznam $[]$
- **Hlava (libovolný objekt), tělo (seznam):** $.(Hlava, Te\text{lo})$
 - všechny strukturované objekty stromy – i seznamy
 - funktor ".", dva argumenty
 - $.(a, .(b, .(c, []))) = [a, b, c]$
 - notace: $[Hlava | Te\text{lo}] = [a|Te\text{lo}]$
 $Te\text{lo}$ je v $[a|Te\text{lo}]$ seznam, tedy píšeme $[a, b, c] = [a | [b, c]]$
- Lze psát i: $[a,b|Te\text{lo}]$
 - 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:** $[a,b,c|T]$
 - $\text{Seznam} = [a,b,c|T]$, $T = [d,e|S]$, $\text{Seznam} = [a,b,c,d,e|S]$

Seznamy (pokračování)

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

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) ).
```

LCO a akumulátor

- Reformulace rekurzivní procedury, aby umožnila LCO
- Výpočet délky seznamu `length(Seznam, Delka)`
 - `length([], 0).`
 - `length([H | T], Delka) :- length(T, Delka0), Delka is 1 + Delka0.`
- Upravená procedura, tak aby umožnila LCO:
 - `% length(Seznam, ZapocitanaDelka, CelkovaDelka):`
 - `% CelkovaDelka = ZapocitanaDelka + ,,počet prvků v Seznam''`
 - `length(Seznam, Delka) :- length(Seznam, 0, Delka).` % pomocný predikát
 - `length([], Delka, Delka).` % celková délka = započítaná délka
 - `length([H | T], A, Delka) :- A0 is A + 1, length(T, A0, Delka).`
- Příkladový argument se nazývá **akumulátor**

Akumulátor jako seznam

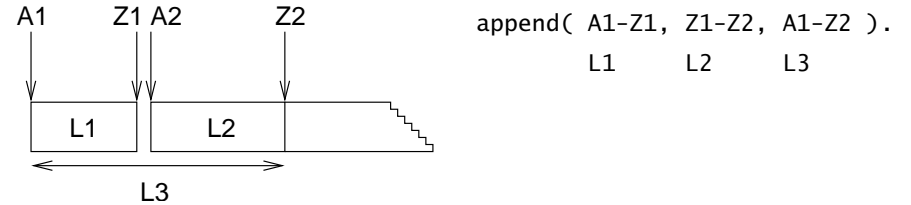
- Nalezení seznamu, ve kterém jsou prvky v opačném pořadí `reverse(Seznam, OpacnySeznam)`
 - `reverse([], []).`
 - `reverse([H | T], Opacny) :-`
 - `reverse(T, OpacnyT),`
 - `append(OpacnyT, [H], Opacny).`
 - naivní reverse s kvadratickou složitostí
- reverse pomocí akumulátoru s lineární složitostí
 - `% 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`)

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

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

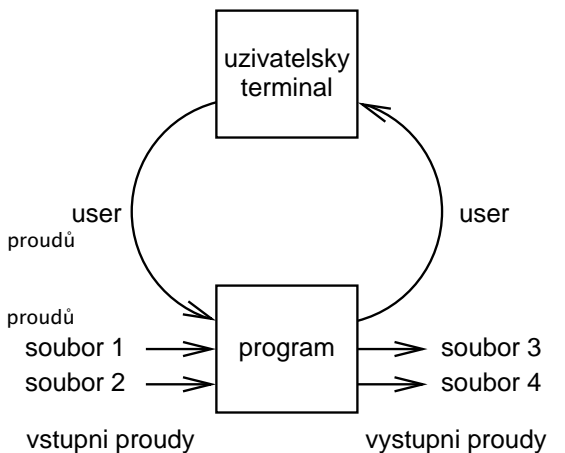
- program může číst data ze **vstupního proudu** (*input stream*)
- program může zapisovat data do **výstupního proudu** (*output stream*)

dva aktivní proudy

- aktivní vstupní proud
- aktivní výstupní proud

uživatelský terminál – user

- 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 ).
```

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.
```

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`

Čtení programu ze souboru

- **Interpretování** kódu programu

- `?- consult(program).`

- `?- consult('program.pl').`

- `?- consult([program1, 'program2.pl']).`

- `?- [program].`

- `?- [user].` **zadávaní kódu ze vstupu ukončené CTRL+D**

- **Kompilace** kódu programu

- `?- compile([program1, 'program2.pl']).`

- další varianty podobně jako u interpretování

- typické zrychlení: 5 až 10 krát

Všechna řešení

- Backtracking vrátí 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 ]
```

Všechna 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

```
?- bagof( Dite-Vek, vek( Dite, Vek ), Seznam ).  
Seznam = [petr-7,anna-5,tomas-5]
```

Existenční kvantifikátor „^”

- Přidání **existenčního kvantifikátoru** „^” ⇒ hodnota proměnné nemá význam

```
?- bagof( Dite, Vek^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 „^” může být i seznam

```
?- bagof( Vek, [Jmeno,Prijmeni]^vek( Jmeno, Prijmeni, Vek ), Seznam ).  
Seznam = [7,5,5]
```

Všechna řešení III.

- `setof(X, P, S)`: rozdíly od `bagof`

- S je uspořádaný podle @<
- případné duplicity v S jsou eliminovány

- `findall(X, P, S)`: rozdíly 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]
```