Seznamy

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]`

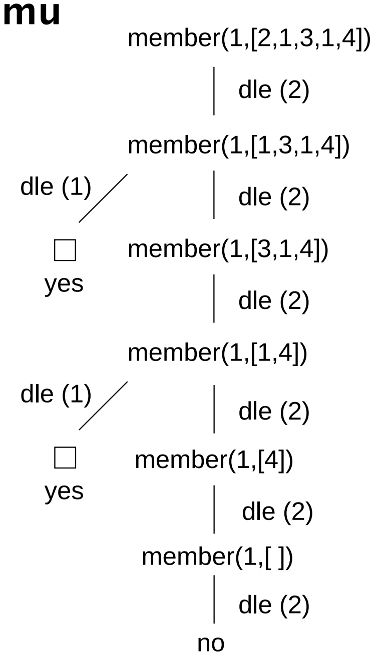
    `Telo` je v `[a|Telo]` seznam, tedy píšeme [a, b, c] = [a | [b, c]]
- Lze psát i: `[a,b|Telo]`
  - před "|" je libovolný počet prvků seznamu , za "|" je seznam zbývajících prvků
  - `[a,b,c] = [a|[b,c]] = [a,b|[c]] = [a,b,c|[]]`
  - pozor: `[ [a,b] | [c] ]` ≠ `[ a,b | [c] ]`
- Seznam jako **neúplná datová struktura:** `[a,b,c|T]`
  - `Seznam = [a,b,c|T], T = [d,e|S], Seznam = [a,b,c,d,e|S]`

Prvek seznamu

- `member( X, S )`
- platí: `member( b, [a,b,c] ).`
- neplatí: `member( b, [[a,b]|[c]] ).`
- X je prvek seznamu S, když
  - X je hlava seznamu S nebo

    `member( X, [ X | _ ] ).   %(1)`
  - X je prvek těla seznamu S

    ```
    member( X, [ _ | Telo ] ) :-
           member( X, Telo ).   %(2)
    ```
- Příklady použití:
  - `member(1,[2,1,3]).`
  - `member(X,[1,2,3]).`



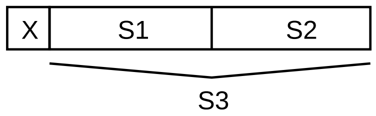
Spojení seznamů

- `append( L1, L2, L3 )`
- Platí: `append( [a,b], [c,d], [a,b,c,d] )`
- Neplatí: `append( [b,a], [c,d], [a,b,c,d] ),`

  `append( [a,[b]], [c,d], [a,b,c,d] )`
- Definice:
  - pokud je 1. argument prázdný seznam, pak 2. a 3. argument jsou stejné seznamy:

    `append( [], S, S ).`
  - pokud je 1. argument neprázdný seznam, pak má 3. argument stejnou hlavu jako 1.:

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



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
```


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 zajišt'uje determinismus
    ```
- Tento typ **rekurze lze převést na iteraci**

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 ).
  ```
- 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) ).
```
```

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 složitosti
- reverse pomocí akumulátoru s lineární složitostí

```
% reverse( Seznam, Akumulator, Opacny ):
%  Opacny obdržíme přidáním prvků ze Seznam do Akumulator v opacnem poradi
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)

reverse/2: **cvičení**

```
reverse( Seznam, OpacnySeznam ) :-          % (1)
        reverse( Seznam, [], OpacnySeznam).
reverse( [], S, S ).                        % (2)
reverse( [ H | T ], A, Opacny ) :-          % (3)
        reverse( T, [ H | A ], Opacny ).
```

```
? - reverse([1,2,3],O).
reverse([1,2,3],O) → (1)
    reverse([1,2,3], [], O) → (3)
        reverse([2,3], [1], O) →  (3)
          reverse([3], [2,1], O) → (3)
             reverse([], [3,2,1], O) →  (2)
                 yes     O=[3,2,1]
```


Neefektivita při spojování seznamů

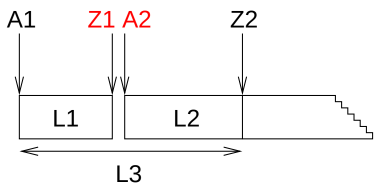
- Sjednocení dvou seznamů

```
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( A1-Z1, Z1-Z2, A1-Z2 ).
          L1       L2         L3
        [2,3]     [1]      [2,3,1]
[2,3|Z1]-Z1  [1|Z2]-Z2  [2,3,1|Z2]-Z2
```

```
?- append( [2,3|Z1]-Z1, [1|Z2]-Z2, S ).
S = A1 - Z2 =  [2,3|Z1] - Z2 = [2,3| [1|Z2] ] - Z2
Z1 = [1|Z2]     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  ) :-             rozdílové seznamy
         reverse0( T, Opacny-[ H | OpacnyKonec] ).                (lineární)
```

Příklad: operace pro manipulaci s frontou

- test na prázdnost, 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( _, _ ) :- !.
```