IB013 Logické programování I

(průsvitky ze cvičení)

Hana Rudová

jaro 2012

Syntaxe logického programu

Term:

- univerzální datová struktura (slouží také pro příkazy jazyka)
- definovaný rekurzivně
- **konstanty:** číselné, alfanumerické (začínají malým písmenem), ze speciálních znaků (operátory)
- **proměnné**: pojmenované (alfanumerické řetězce začínající velkým písmenem), anonymní (začínají podtržítkem)
- **složený term**: funktor, arita, argumenty struktury jsou opět termy

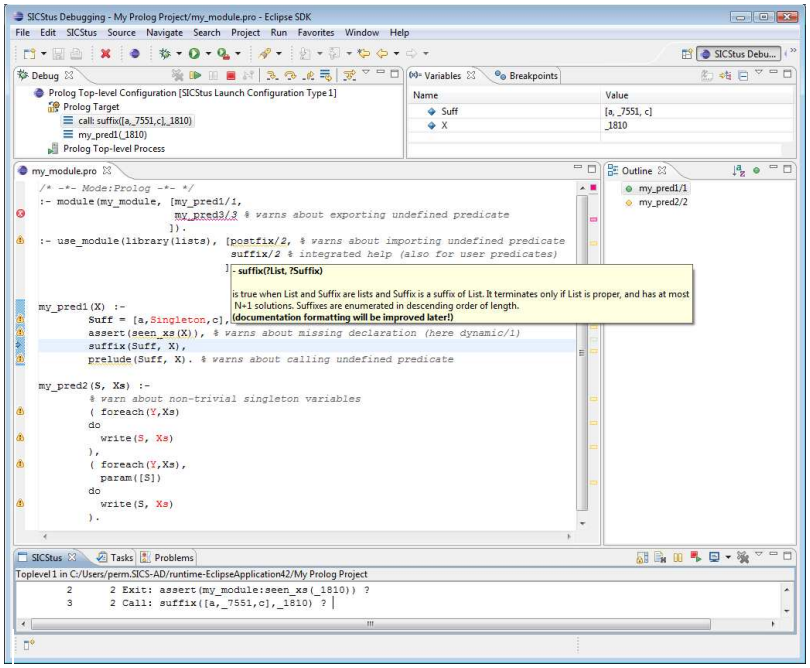
Backtracking, unifikace, aritmetika

Anatomie a sémantika logického programu

- **Program**: množina predikátů (v jednom nebo více souborech).
- **Predikát** (procedura) je seznam klauzulí s hlavou stejného jména a arity
- **Klauzule**: věty ukončené tečkou, se skládají z hlavy a těla.
  Prázdné tělo mají **fakta**, neprázdné pak **pravidla**, existují také klauzule bez hlavy – direktivy.
  Hlavu tvoří **literál (složený term)**, tělo seznam literálů.
  Literálům v těle nebo v dotazu říkáme **cíle**.
  Dotazem v prostředí interpretu se spouští programy či procedury.
  - př.
```
otec(Otec,Dite) :- rodic(Otec,Dite), muz(Otec).
rodic(petr, jana).
:- otec(Otec, jana).
```

Sémantika logického programu:

procedury ≡ databáze faktů a pravidel ≡ logické formule



SICStus Prolog: spouštění programu

- UNIX:

```
module add sicstus-4.1.3
eclipse          % používání IDE SPIDER
sicstus          % používání přes příkazový řádek
```

- MS Windows:
  - používání IDE SPIDER: z nabídky All Programs -> IDE -> Eclipse 3.7
  - příkazový řádek: z nabídky All Programs -> IDE -> SICStus Prolog VC9 4.2.0 nastavíme pracovní adresář pomocí File/Working directory, v případě potřeby nastavíme font Settings/Font a uložíme nastavení Settings/Save settings.
- Iniciální nastavení SICStus IDE v Eclipse pomocí Help->Cheat Sheets->Initial set up of paths to installed SICStus Prolog s cestou "C:\Program Files (x86)\SICStus Prolog VC9 4.2.0\bin\sicstus.exe"

  návod: `http://www.sics.se/sicstus/spider/site/prerequisites.html#SettingUp`

SICStus Prolog: konzultace

- **Otevření souboru:** File->Open File
- **Přístup k příkazové řádce pro zadávání dotazů:** SICStus->Open Toplevel
- **Načtení programu:** tzv. konzultace

  přímo z Menu: SICStus->Consult Prolog Code (okno s programem aktivní)

  nebo zadáním na příkazový řádek po uložení souboru (Ctrl+S)

```
?- consult(rodokmen).
```

  pokud uvádíme celé jméno případně cestu, dáváme jej do apostrofů

```
?- consult('D:\prolog\moje\programy\rodokmen.pl').
```

- V Eclipse lze nastavit Key bindings, pracovní adresář, ...

SICStus Prolog: spouštění a přerušení výpočtu

- **Spouštění programů/procedur/predikátů** je zápis dotazů na příkazové řádce (v okně TopLevel, kurzor musí být na konci posledního řádku s | ?-), př.

```
?- predek(petr,lenka).
?- predek(X,Y).
```

  Každý příkaz ukončujeme tečkou.

- **Přerušení a zastavení cyklícího programu**:

  pomocí ikony Restart Prolog z okna Toplevel

Příklad rodokmen

```
rodic(petr, filip).
rodic(petr, lenka).
rodic(pavel, jan).
rodic(adam, petr).
rodic(tomas, michal).
rodic(michal, radek).
rodic(eva, filip).
rodic(jana, lenka).
rodic(pavla, petr).
rodic(pavla, tomas).
rodic(lenka, vera).
```

```
muz(petr).
muz(filip).
muz(pavel).
muz(jan).
muz(adam).
muz(tomas).
muz(michal).
muz(radek).
```

```
zena(eva).
zena(lenka).
zena(pavla).
zena(jana).
zena(vera).
```

```
otec(Otec,Dite) :- rodic(Otec,Dite), muz(Otec).
```


Backtracking: příklady

V pracovním adresáři vytvořte program rodokmen.pl.
Načtěte program v interpretu (konzultujte).
V interpretu Sicstus Prologu pokládejte dotazy:

- Je Petr otcem Lenky?
- Je Petr otcem Jana?
- Kdo je otcem Petra?
- Jaké děti má Pavla?
- Ma Petr dceru?
- Které dvojice otec-syn známe?

Backtracking: příklady II

Predikát potomek/2:

```
potomek(Potomek,Predek) :- rodic(Predek,Potomek).
potomek(Potomek,Predek) :- rodic(Predek,X), potomek(Potomek,X).
```

Naprogramujte predikáty

- `prababicka(Prababicka,Pravnouce)`
- `nevlastni_bratr(Nevlastni_bratr,Nevlastni_sourozenec)`
  nápověda: využijte X \== Y (X a Y nejsou identické)

Backtracking: porovnání

Nahraďte ve svých programech volání predikátu `rodic/2` následujícím predikátem `rodic_v/2`

```
rodic_v(X,Y):-rodic(X,Y),print(X),print('? ').
```

Pozorujte rozdíly v délce výpočtu dotazu `nevlastni_bratr(filip,X)` při změně pořadí testů v definici predikátu `nevlastni_bratr/2`

- varianta 1: testy co nejdříve — správně
- varianta 2: všechny testy umístěte na konec — chybně

Co uvidíme po nahrazení predikátu `rodic/2` predikatem `rodic_v/2` v predikátech `nevlastni_bratr/2` a `nevlastni_bratr2/2` a spuštění?

Backtracking: prohledávání stavového prostoru

```
potomek(Potomek,Predek) :- rodic(Predek,Potomek).
potomek(Potomek,Predek) :- rodic(Predek,X), potomek(Potomek,X).
```

- Zkuste předem odhadnout (odvodit) pořadí, v jakem budou nalezeni potomci Pavly?

  ```
  :- potomek(X,pavla).
  ```

- Jaký vliv má pořadí klauzulí a cílu v predikátu potomek/2 na jeho funkci?

```
rodic(petr, filip).      rodic(petr, lenka).
rodic(pavel, jan).      rodic(adam, petr).
rodic(tomas, michal).      rodic(michal, radek).
rodic(eva, filip).      rodic(jana, lenka).
rodic(pavla, petr).      rodic(pavla, tomas).
rodic(lenka, vera).
```


Unifikace:příklady

Které unifikace jsou korektní, které ne a proč?

Co je výsledkem provedených unifikací?

1. a(X)=b(X)
2. X=a(Y)
3. a(X)=a(X,X)
4. X=a(X)
5. jmeno(X,X)=jmeno(Petr,plus)
6. s(1,a(X,q(w)))=s(Y,a(2,Z))
7. s(1,a(X,q(X)))=s(W,a(Z,Z))
8. X=Y,P=R,s(1,a(P,q(R)))=s(Z,a(X,Y))

Mechanismus unifikace I

Unifikace v průběhu dokazování predikátu odpovídá předávání parametrů při provádění procedury, ale je důležité uvědomit si rozdíly. Celý proces si ukážeme na příkladu predikátu suma/3.

```
suma(0,X,X).                        /*klauzule A*/
suma(s(X),Y,s(Z)):-suma(X,Y,Z).     /*klauzule B*/
```

pomocí substitučních rovnic při odvozování odpovědi na dotaz

```
?- suma(s(0),s(0),X0).
```


Mechanismus unifikace II

```
suma(0,X,X). /*A*/                suma(s(X),Y,s(Z)):-suma(X,Y,Z). /*B*/
?- suma(s(0),s(0),X0).
```

1. dotaz unifikujeme s hlavou klauzule B, s A nejde unifikovat (1. argument)

   ```
   suma(s(0),s(0),X0) = suma(s(X1),Y1,s(Z1))
   ==> X1 = 0, Y1 = s(0), s(Z1) = X0
   ==> suma(0,s(0),Z1)
   ```

2. dotaz (nový podcíl) unifikujeme s hlavou klauzule A, klauzuli B si poznačíme jako další možnost

   ```
   suma(0,s(0),Z1) = suma(0,X2,X2)
   X2 = s(0), Z1 = s(0)
   ==> X0 = s(s(0))
   X0 = s(s(0)) ;
   ```

2' dotaz z kroku 1. nejde unifikovat s hlavou klauzule B (1. argument)

no

Vícesměrnost predikátů

Logický program lze využít vícesměrně, například jako

- výpočet kdo je otcem Petra? `?- otec(X,petr).`
  kolik je 1+1? `?- suma(s(0),s(0),X).`
- test je Jan otcem Petra? `?- otec(jan,petr).`
  Je 1+1 2? `?- suma(s(0),s(0),s((0)) ).`
- generátor které dvojice otec-dítě známe? `?-otec(X,Y).`
  Které X a Y dávají v součtu 2? `?- suma(X,Y,s(s(0)) ).`

... ale pozor na levou rekurzi, volné proměnné, asymetrii, a jiné záležitosti

Následující dotazy

`?-suma(X,s(0),Z).` `?-suma(s(0),X,Z).`

nedávají stejné výsledky. Zkuste si je odvodit pomocí substitučních rovnic.

Aritmetika

Zavádíme z praktických důvodů, ale aritmetické predikáty již nejsou vícesměrné, protože v každém aritmetickém výrazu musí být všechny proměnné instaciovány číselnou konstantou.

Důležitý rozdíl ve vestavěných predikátech is/2 vs. =/2 vs. =:=/2

is/2: < konstanta nebo proměnná > is < aritmetický výraz >

výraz na pravé straně je nejdříve aritmeticky vyhodnocen a pak unifikován s levou stranou

=/2: < libovolný term > = < libovolný term >
levá a pravá strana jsou unifikovány

"=:="/2 "=\="/2 ">="/2 "=<"/2
< aritmetický výraz > =:= < aritmetický výraz >
< aritmetický výraz > =\= < aritmetický výraz >
< aritmetický výraz > =< < aritmetický výraz >
< aritmetický výraz > >= < aritmetický výraz >
levá i pravá strana jsou nejdříve aritmeticky vyhodnoceny a pak porovnány

Aritmetika: příklady

Jak se liší následující dotazy (na co se kdy ptáme)? Které uspějí (kladná odpověď), které neuspějí (záporná odpověď), a které jsou špatně (dojde k chybě)? Za jakých předpokladů by ty neúspěšné případně špatné uspěly?

1. X = Y + 1
2. X is Y + 1
3. X = Y
4. X == Y
5. 1 + 1 = 2
6. 2 = 1 + 1
7. 1 + 1 = 1 + 1
8. 1 + 1 is 1 + 1
9. 1 + 2 =:= 2 + 1
10. X \== Y
11. X =\= Y
12. 1 + 2 =\= 1 - 2
13. 1 <= 2
14. 1 =< 2
15. sin(X) is sin(2)
16. sin(X) = sin(2+Y)
17. sin(X) =:= sin(2+Y)

Nápověda: '='/2 unifikace, '=='/2 test na identitu, '=:='/2 aritmetická rovnost, '\=='/2 negace testu na identitu, '=\='/2 aritmetická nerovnost

Aritmetika: příklady II

Jak se liší predikáty s1/3 a s2/3? Co umí s1/3 navíc oproti s2/3 a naopak?

```
s1(0,X,X).
s1(s(X),Y,s(Z)):-s1(X,Y,Z).

s2(X,Y,Z):- Z is X + Y.
```

Závěr

Dnešní látku jste pochopili dobře, pokud víte

- jaký vliv má pořadí klauzulí a cílu v predikátu potomek/2 na jeho funkci,
- jak umisťovat testy, aby byl prohledávaný prostor co nejmenší (příklad nevlastni_bratr/2),
- k čemu dojde po unifikaci X=a(X),
- proč neuspěje dotaz ?- X=2, sin(X) is sin(2).
- za jakých předpokladů uspějí tyto cíle X=Y, X==Y, X=:=Y,
- a umíte odvodit pomocí substitučních rovnic odpovedi na dotazy suma(X,s(0),Z) a suma(s(0),X,Z).

Seznamy, řez

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]`

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
```

Predchudce a naslednik prvku X v seznamu S

```
hledej(S,X,Pred,Po) :- append( _S1, [ Pred,X,Po | _S2 ], 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 ) :-`
  DÚ: suffix(S1,S2)
- `last( X, S ) :-`
  `append([3,2], [6], [3,2,6]).` `X=6, S=[3,2,6]`
- `member( X, 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) :-`

  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 pamětovými 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, 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**

Akumulátor a sum_list(S,Sum)

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

s akumulátorem:
```

Výpočet faktoriálu `fact(N,F)`

s akumulátorem:


```
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: maximum

Je tato definice predikátu max/3 korektní?

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


Řez: member

Jaký je rozdíl mezi následujícími definicemi predikátů member/2. Ve kterých odpovědích se budou lišit? Vyzkoušejte např. pomocí 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).
```

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).

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

Vstup/výstup, databázové operace, rozklad termu

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.                             % vestavěný predikát
repeat :- repeat.
```

Predikáty pro vstup a výstup

```
| ?- read(A), read( ahoj(B) ), read( [C,D] ).
|: ahoj. ahoj( petre ). [ ahoj( 'Petre!' ), jdeme ].
A = ahoj, B = petre, C = ahoj('Petre!'), D = jdeme

| ?- write(a(1)), write('.'), nl, write(a(2)), write('.'), nl.
a(1).
a(2).
yes
```

- seeing, see, seen, read
- telling, tell, told, write
- see/tell(Soubor)
  - pokud Soubor není otevřený: otevření a aktivace
  - pokud Soubor otevřený: pouze aktivace (tj. udělá z něj aktivní vstupní/výstupní stream)
- standardní vstupní a výstupní stream: user

Příklad: vstup/výstup

Napište predikát uloz_do_souboru(Soubor), který načte několik fakt ze vstupu a uloží je do souboru Soubor.

```
| ?- uloz_do_souboru( 'soubor.pl' ).
|: fakt(mirek, 18).
|: fakt(pavel,4).
|: end_of_file.
yes
| ?- consult(soubor).
% consulting /home/hanka/soubor.pl...
% consulted /home/hanka/soubor.pl in module user, 0 msec
% 376 bytes
yes
| ?- listing(fakt/2). % pozor:listing/1 lze použít pouze při consult/1 (ne u compile/1)
fakt(mirek, 18).
fakt(pavel, 4).
yes
```


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: retract/1 lze použít pouze pro **dynamické klauzule** (přidané pomocí assert) a ne pro statické klauzule z programu
- Pozor: nadměrné použití těchto operací snižuje srozumitelnost programu

Databázové operace: příklad

Napište predikát vytvor_program/0, který načte několik klauzulí ze vstupu a uloží je do programové databáze.

```
| ?- vytvor_program.
|: fakt(pavel, 4).
|: pravidlo(X,Y) :- fakt(X,Y).
|: end_of_file.
yes
| ?- listing(fakt/2).
fakt(pavel, 4).
yes
| ?- listing(pravidlo/2).
pravidlo(A, B) :- fakt(A, B).
yes
| ?- clause( pravidlo(A,B), C). % clause/2 použitelný pouze pro dynamické klauzule
C = fakt(A,B) ?
yes
```

Konstrukce a dekompozice termu

- Konstrukce a dekompozice termu

  `Term =.. [ Funktor | SeznamArgumentu ]`

  ```
  a(9,e) =.. [a,9,e]
  Cil =.. [ Funktor | SeznamArgumentu ], call( Cil )
  atom =.. X ⇒ X = [atom]
  ```

- Pokud chci znát pouze funktor nebo některé argumenty, pak je efektivnější:

  `functor( Term, Funktor, Arita )` `functor( a(9,e), a, 2 )`

  `functor(atom,atom,0)` `functor(1,1,0)`

  `arg( N, Term, Argument )` `arg( 2, a(9,e), e)`

Rekurzivní rozklad termu

- Term je proměnná (`var/1`), atom nebo číslo (`atomic/1`) ⇒ konec rozkladu
- Term je seznam (`[_|_]`) ⇒
  procházení seznamu a rozklad každého prvku seznamu
- Term je složený (`=../2`, `functor/3`) ⇒
  procházení seznamu argumentů a rozklad každého argumentu
- Příklad: ground/1 uspěje, pokud v termu nejsou proměnné; jinak neuspěje

```
ground(Term) :- atomic(Term), !.          % Term je atom nebo číslo NEBO
ground(Term) :- var(Term), !, fail.       % Term není proměnná NEBO
ground([H|T]) :- !, ground(H), ground(T). % Term je seznam a ani hlava ani těl
                                          %      neobsahují proměnné NEBO
ground(Term) :- Term =.. [ _Funktor | Argumenty ], % je Term složený
                ground( Argumenty ).              % a jeho argumenty
                                                  % neobsahují proměnné

?- ground(s(2,[a(1,3),b,c],X)).        ?- ground(s(2,[a(1,3),b,c])).
no                                     yes
```


subterm(S,T)

Napište predikát `subterm(S,T)` pro termy S a T bez proměnných, které uspějí, pokud je S podtermem termu T. Tj. musí platit alespoň jedno z

- podterm S je právě term T NEBO
- podterm S se nachází v hlavě seznamu T NEBO
- podterm S se nachází v těle seznamu T NEBO
- T je složený term (`compound/1`) a S je podtermem některého argumentu T
  - otestujte `:- subterm(1,2).`
    pokud nepoužijeme (`compound/1`), pak tento dotaz cyklí
  - otestujte `:- subterm(a,[1,2]).` ověřte, zda necyklí (nutný červený řez níže)

```
| ?- subterm(sin(3),b(c,2,[1,b],sin(3),a)).                    yes
```


same(A,B)

Napište predikát `same(A,B)`, který uspěje, pokud mají termy A a B stejnou strukturu. Tj. musí platit právě jedno z

- A i B jsou proměnné NEBO
- pokud je jeden z argumentů proměnná (druhý ne), pak predikát neuspěje, NEBO
- A i B jsou `atomic` a unifikovatelné NEBO
- A i B jsou seznamy, pak jak jejich hlava tak jejich tělo mají stejnou strukturu NEBO
- A i B jsou složené termy se stejným funktorem a jejich argumenty mají stejnou strukturu

```
| ?- same([1,3,sin(X),s(a,3)],[1,3,sin(X),s(a,3)]).                yes
```

D.Ú. unify(A,B)

Napište predikát unify(A,B), který unifikuje termy A a B a provede zároveň *kontrolu výskytu* pomocí not_occurs(Var,Term).

```
| ?- unify([Y,3,sin(a(3)),s(a,3)],[1,3,sin(X),s(a,3)]).
X = a(3)      Y = 1       yes
```


not_occurs(A,B)

Predikát not_occurs(A,B) uspěje, pokud se proměnná A nevyskytuje v termu B. Tj. platí jedno z

- B je atom nebo číslo NEBO
- B je proměnná různá od A NEBO
- B je seznam a A se nevyskytuje ani v tělě ani v hlavě NEBO
- B je složený term a A se nevyskytuje v jeho argumentech

Definite-Clause Grammars (DCG)
Gramatiky uspořádaných klauzulí

Syntaktická analýza

Významná aplikace Prologu: syntaktická analýza

- ```
  sentence --> noun_phrase, verb_phrase.

  noun_phrase --> determiner, noun.
  noun_phrase --> noun.

  verb_phrase --> verb, noun_phrase.
  verb_phrase --> verb.

  determiner --> [the].
  determiner --> [a].

  noun --> [student].
  noun --> [dcg].

  verb --> [likes].
  ```
- ```
  | ?- sentence([a, student, likes, dcg]).
  yes
  ```

DCG a CFG

- DCG (DC gramatiky) jsou rozšířením bezkontextových gramatik (CFG)
- Implementace DCG využívá rozdílových seznamů

Formální podobnosti mezi DCG a CFG:

- CFG: pravidla tvaru $x \to y$, kde
  - $x \in N$ je neterminál
  - $y \in (N \cup T)^*$ je konečná posloupnost terminálů a neterminálů
- DCG: pravidla tvaru `<hlava> --> <tělo>`
  - `<hlava>` je opět neterminál
  - `<tělo>` je opět konečná posloupnost terminálů a neterminálů
- Pravidlo `<hlava> --> <tělo>` znamená, že
  - jedním z možných tvarů `<hlavy>` je `<tělo>`, neboli
  - `<hlavu>` je možno přepsat na `<tělo>`

Rozdíly a rozšíření DCG oproti CFG

- **Neterminál** může být téměř libovolný term, ovšem kromě seznamu, proměnné a čísla.
  - neterminál muže být složený term, tj. neterminálům lze přidat **argumenty**.
- **Terminál** může být libovolný term, s tím, že terminály a posloupnosti terminálů uzavíráme do hranatých závorek – jako **seznamy**.
  - hranaté závorky tedy odlišují terminály od neterminálů
- Pravá strana pravidla může obsahovat **dodatečné podmínky** v podobě prologovských podcílů. Tyto podmínky uzavíráme do složených závorek.
  - podmínky slouží jen pro testování, negenerují žádnou větnou formu
- Levá strana pravidla může dokonce vypadat i tak, že neterminál je následován posloupností terminálů.
- Tělo pravidla smí obsahovat řez.
  - nepodporováno všemi Prology

Příklad: gramatika

- ```
  sentence --> noun_phrase, verb_phrase.
  noun_phrase --> determiner, noun_phrase2.
  noun_phrase --> noun_phrase2.
  noun_phrase2 --> noun.
  noun_phrase2 --> adjective, noun_phrase2.
  verb_phrase --> verb.
  verb_phrase --> verb, noun_phrase.
  determiner --> [the].     noun --> [boy].
  determiner --> [a].       noun --> [song].
  verb --> [sings].         adjective --> [young].
  ```
- ```
  | ?- sentence(S,[]).
  S = [the,song,sings] ? ;
  S = [the,song,sings,the,song] ?
  | ?- sentence([the, young, boy, sings, a, song],[]).
  yes
  ```


Příklad: binární čísla

- DC gramatika `number` rozeznávající binární čísla:

```
number --> [0].
number --> [1].
number --> [0], number.
number --> [1], number.
```

```
| ?- number([0,1,0,1,1], []).                    yes
```

- Napište DCG `number2` pro rozpoznání binárních čísel bez vedoucích nul.
- Napište DCG `number3` rozpoznávající binární čísla, které jsou mocninou dvojky.

Příklad: neterminály s argumentem

- DC gramatika `digits` generuje binární čísla zapsaná jedinou číslicí:

```
digits --> same(0).
digits --> same(1).
same(N) --> [N].
same(N) --> [N], same(N).
```

```
| ?- digits([1,1,0,1], []).
no
| ?- digits([1,1,1], []).
yes
```

- Upravte kód tak, aby byly akceptovány jen korektní věty:

```
s --> np, vp.
np --> [zeny].
np --> [muzi].
vp --> [pracovali].
vp --> [pracovaly].

?- s([zeny, pracovali], []).
    yes
```

Nápověda: přidejte proměnnou pro rod (pro np a vp)

Generativní/rozpoznávací síla DCG: větší než CFG

- DCG dokáží generovat/rozpoznávat jazyky typu 0
- Cvičení: napište DCG gramatiku generující jazyk $a^n b^n c^n$
-
```
?- abc(X,[]).
X = [] ;
X = [a, b, c] ;
X = [a, a, b, b, c, c] ;
X = [a, a, a, b, b, b, c, c, c] ;
```

Nápověda: využijte a(s(s(s(0)))), b(s(s(s(0)))), c(s(s(s(0))))

Pomocné podmínky v těle pravidel

-
```
E -> T + E | T
T -> num

expr(X) --> term(A), [+], expr(B), {X is A+B}.
expr(X) --> term(X).
term(X) --> [X], {number(X)}.

?- expr(X, [1,+,2,+,2], []).      X = 5
```

- Cvičení: přidejte operaci násobení

```
E -> N + E | N
N -> T * N | T
T -> num
```


Komplexní vyhodnocování výrazů

```
E -> T + E | T - E | T
T -> F * T | F / T | F
F -> (E) | f

expr(X) --> term(Y), [+], expr(Z), {X is Y+Z}.
expr(X) --> term(Y), [-], expr(Z), {X is Y-Z}.
expr(X) --> term(X).

term(X) --> factor(Y), [*], term(Z), {X is Y*Z}.
term(X) --> factor(Y), [/], term(Z), {X is Y/Z}.
term(X) --> factor(X).

factor(X) --> ['('], expr(X), [')'].
factor(X) --> [X], {integer(X)}.

% vyhodnocení výrazu 3+(4/2)-(2*6/3)
?- expr(X,[3,+,'(',4,/,2,')',-,'(',2,*,6,/,3,')'],[]).      X = 1
```

Argument neterminálu je použit jako výstupní proměnná,
která v sobě nese hodnotu příslušného aritmetického výrazu.

Přepis do Prologu

Přepis do prologovského programu pomocí append/3:

- Větu reprezentujeme seznamem slov `[the,young,boy,sings,a,song]`
- **Pravidlová část** – neterminál chápeme jako unární predikát, jehož argumentem je ta větná složka, kterou daný neterminál popisuje

```
sentence(S) :- append(NP,VP,S),
               noun_phrase(NP), verb_phrase(VP).
...
```

- **Slovníková část** – zapisujeme ji pomocí faktů:

```
determiner([the]).          noun([boy]).
determiner([a]).            ...
```

Predikát append/3 zde *nedeterministicky* rozděluje aktuální větnou část na dva díly, což je velký zdroj neefektivnosti.

Lepší řešení poskytují *rozdílové seznamy*.

Přepis do Prologu pomocí rozdílových seznamů

- **Rozdílové seznamy** reprezentovány dvěma argumenty, první představuje neúplný seznam a druhý jeho zbytek append(S–S1, S1–S0, S–S0)
- Při volání predikátu S–S0 je spojením: S–S3, S3–S2, S2–S1, S1–S0
  sentence/2 je druhý argument prázdný; neúplný seznam tím uzavíráme tj. S0=[ ]

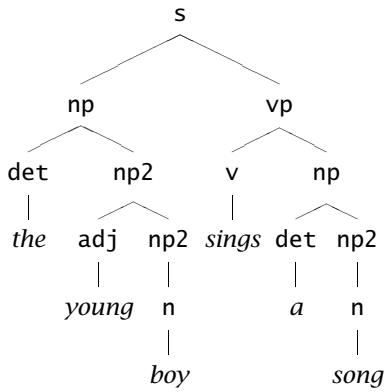
```
sentence(S,S0):- noun_phrs(S,S1), verb_phrs(S1,S0).
noun_phrs(S,S0):- determiner(S,S1), noun_phrs2(S1,S0).
noun_phrs(S,S0):- noun_phrs2(S,S0).
noun_phrs2(S,S0):- adjective(S,S1), noun_phrs2(S1,S0).
noun_phrs2(S,S0):- noun(S,S0).
verb_phrs(S,S0):- verb(S,S0).
verb_phrs(S,S0):- verb(S,S1), noun_phrs(S1,S0).

determiner([the|S],S).      noun([boy|S],S).
determiner([a|S],S).        noun([song|S],S).
verb([sings|S],S).          adjective([young|S],S).

?- sentence([the,young,boy,sings,a,song],[]).          yes
```


Derivační strom analýzy

```
?- sentence(Tree, [the,young,boy,sings,a,song],[]).
Tree=s( np( det(the), np2( adj(young), np2(n(boy) ) ) ),
        vp( v(sings), np( det(a), np2( n(song) ) ) ) )
```



Konstrukce derivačního stromu

- Neterminály opatříme argumentem:

```
sentence(s(NP,VP)) --> noun_phrase(NP), verb_phrase(VP).
noun(n(mama)) --> [mama].
noun(n(kralika)) --> [kralika].
verb(v(pekla)) --> [pekla].
```

- Doplňte gramatiku, aby platilo:

```
| ?- sentence(X, [mama,pekla,kralika], []).
X = s(np(n(mama)),vp(v(pekla),np(n(kralika))))      yes
```

Konstrukce derivačního stromu II.

Pokud však rozšíříme slovník:

```
noun(n(tata)) --> [tata].
verb(v(pekl)) --> [pekl].
```

Narazíme na problém se shodou podmětu a přísudku (mimo stávající problém „kralíka pekla máma"):

```
?- sentence(_,[tata, pekla, kralika],[]).
yes

?- sentence(_,[mama, pekl, kralika],[]).
yes
```

Proto rozšiřte neterminály o další argumenty (rod, pád)

Vestavěné nástroje

- operátor `-->` definován jako `?-op(1200,xfx,-->).`
- predikáty `phrase/2`, `phrase/3`, které slouží k jednoduché *tokenizaci*

```
?- phrase(abc,[a,b,c]).          % Yes
?- phrase(abc,[a,b,c,d],[d]).    % Yes
```


Logické programování s omezujícími podmínkami

Algebrogram

- Přiřaďte cifry 0, . . . 9 písmenům S, E, N, D, M, O, R, Y tak, aby platilo:
  - ```
      SEND
    + MORE
    ------
     MONEY
    ```
  - různá písmena mají přiřazena různé cifry
  - S a M nejsou 0
- **Proměnné:** S,E,N,D,M,O,R,Y
- **Domény:** [1..9] pro S,M [0..9] pro E,N,D,O,R,Y
- **1 omezení pro nerovnost**: `all_distinct([S,E,N,D,M,O,R,Y])`
- **1 omezení pro rovnosti**:

```
                1000*S + 100*E + 10*N + D          SEND
+               1000*M + 100*O + 10*R + E        + MORE
#=  10000*M + 1000*O + 100*N + 10*E + Y          MONEY
```
```

Jazykové prvky

Nalezněte řešení pro algebrogram

```
D O N A L D + G E R A L D = R O B E R T
```

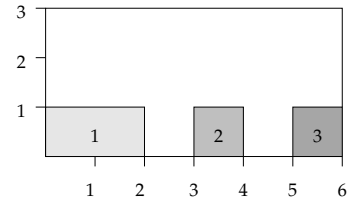
- Struktura programu

```
algebrogram( [D,O,N,A,L,G,E,R,B,T] ) :-
      domain(...),                      % domény proměnných
      all_distinct(...), ... #= ...,    % omezení
      labeling(...).                    % prohledávání stavového prostoru
```

- Knihovna pro CLP(FD) `:- use_module(library(clpfd)).`
- Domény proměnných `domain( Seznam, MinValue, MaxValue )`
- Omezení `all_distinct( Seznam )`
- Aritmetické omezení `A*B + C #= D`
- Procedura pro prohledávání stavového prostoru `labeling([],Seznam)`

Disjunktivní rozvrhování (unární zdroj)

- `cumulative([task(Start, Duration, End, 1, Id) | Tasks])`
- Rozvržení úloh zadaných startovním a koncovým časem (`Start`,`End`), dobou trvání (**nezáporné** `Duration`) a identifikátorem (`Id`) tak, aby se nepřekrývaly
  - příklad s konstantami:
    `cumulative`([task(0,2,2,1,1), task(3,1,4,1,2), task(5,1,6,1,3)])



  - `Start, Duration, End, Id` musí být doménové proměnné s konečnými mezemi nebo celá čísla

Plánování

Každý úkol má stanoven dobu trvání a nejdřívější čas, kdy může být zahájen.
Nalezněte startovní čas každého úkolu tak, aby se jednotlivé úkoly nepřekrývaly.

Úkoly jsou zadány následujícím způsobem:

```
% ukol(Id,Doba,MinStart,MaxKonec)
ukol(1,4,8,70).    ukol(2,2,7,60).    ukol(3,1,2,25).    ukol(4,6,5,55).
ukol(5,4,1,45).    ukol(6,2,4,35).    ukol(7,8,2,25).    ukol(8,5,0,20).
ukol(9,1,8,40).    ukol(10,7,4,50).   ukol(11,5,2,50).   ukol(12,2,0,35).
ukol(13,3,30,60).  ukol(14,5,15,70).  ukol(15,4,10,40).
```

Kostra řešení:

```
ukoly(Zacatky) :- domeny(Ukoly,Zacatky,Tasks),
                  cumulative(Tasks),
                  labeling([],Zacatky),
                  tiskni(Ukoly,Zacatky).

domeny(Ukoly,Zacatky,Tasks) :- findall(ukol(Id,Doba,MinStart,MaxKonec),
                                       ukol(Id,Doba,MinStart,MaxKonec), Ukoly),
                               nastav_domeny(Ukoly,Zacatky,Tasks).
```


Plánování: výstup

```
tiskni(Ukoly,Zacatky) :-
        priprav(Ukoly,Zacatky,Vstup),
        quicksort(Vstup,Vystup),
        nl, tiskni(Vystup).

priprav([],[],[]).
priprav([ukol(Id,Doba,MinStart,MaxKonec)|Ukoly], [Z|Zacatky],
        [ukol(Id,Doba,MinStart,MaxKonec,Z)|Vstup]) :-
        priprav(Ukoly,Zacatky,Vstup).

tiskni([]) :- nl.
tiskni([V|Vystup]) :-
        V=ukol(Id,Doba,MinStart,MaxKonec,Z),
        K is Z+Doba,
        format('   ~d: \t~d..~d \t(~d: ~d..~d)\n',
               [Id,Z,K,Doba,MinStart,MaxKonec] ),
        tiskni(Vystup).
```

Plánování: výstup II

```
quicksort(S, Sorted) :- quicksort1(S,Sorted-[]).

quicksort1([],Z-Z).
quicksort1([X|Tail], A1-Z2) :-
        split(X, Tail, Small, Big),
        quicksort1(Small, A1-[X|A2]),
        quicksort1(Big, A2-Z2).

split(_X, [], [], []).
split(X, [Y|T], [Y|Small], Big) :- greater(X,Y), !, split(X, T, Small, Big).
split(X, [Y|T], Small, [Y|Big]) :- split(X, T, Small, Big).

greater(ukol(_,_,_,_,Z1),ukol(_,_,_,_,Z2)) :- Z1>Z2.
```


Plánování a domény

Napište predikát nastav_domeny/3, který na základě datové struktury [ukol(Id,Doba,MinStart,MaxKonec)|Ukoly] vytvoří doménové proměnné Zacatky pro začátky startovních dob úkolů a strukturu Tasks vhodnou pro omezení cumulative/1, jejíž prvky jsou úlohy ve tvaru task(Zacatek,Doba,Konec,1,Id).

```
% nastav_domeny(+Ukoly,-Zacatky,-Tasks)
```


D.Ú. Plánování a precedence: precedence(Tasks)

Rozšiřte řešení předchozího problému tak, aby umožňovalo zahrnutí precedencí, tj. jsou zadány dvojice úloh A a B a musí platit, že A má být rozvrhováno před B.

```
% prec(IdA,IdB)
prec(8,7).  prec(6,12).  prec(2,1).
```

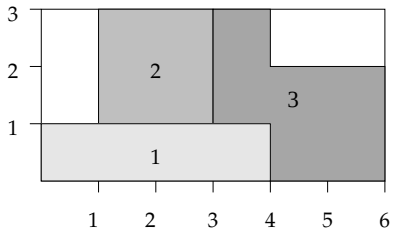
Pro určení úlohy v Tasks lze použít nth1(N,Seznam,NtyPrvek) z knihovny

```
:- use_module(library(lists)).
```


Kumulativní rozvrhování

- cumulative([task(Start,Duration,End,Demand,TaskId) | Tasks], [limit(Limit)])
- Rozvržení úloh zadaných startovním a koncovým časem (Start,End), dobou trvání (**nezáporné** Duration), požadovanou kapacitou zdroje (Demand) a identifikátorem (Id) tak, aby se nepřekrývaly a aby celková kapacita zdroje nikdy nepřekročila Limit
- Příklad s konstantami:

```
cumulative([task(0,4,4,1,1),task(1,2,3,2,2),task(3,3,6,2,3),task(4,2,6,1,4)],[limit(3)])
```



Plánování a lidé

Modifikujte řešení předchozího problému tak, že

- odstraňte omezení na nepřekrývání úkolů
- přidejte omezení umožňující řešení každého úkolu zadaným člověkem (každý člověk může zpracovávat nejvýše tolik úkolů jako je jeho kapacita)

```
ukoly(Zacatky) :-               % původně
        domeny(Ukoly,Zacatky,Tasks),
        cumulative(Tasks),
        labeling([],Zacatky),
        tiskni(Ukoly,Zacatky).

ukoly_lide(Zacatky) :-          % upravená verze
        domeny(Ukoly,Zacatky,Tasks),
        lide(Tasks,Lide),
        labeling([],Zacatky),
        tiskni_lide(Lide,Ukoly,Zacatky).
```


Plánování a lidé

```
% clovek(Id,Kapacita,IdUkoly)
% clovek Id zpracovává úkoly v seznamu IdUkoly
clovek(1,2,[1,2,3,4,5]).
clovek(2,1,[6,7,8,9,10]).
clovek(3,2,[11,12,13,14,15]).

lide(Tasks,Lide) :-
        findall(clovek(Kdo,Kapacita,Ukoly),clovek(Kdo,Kapacita,Ukoly), Lide),
        omezeni_lide(Lide,Tasks).

omezeni_lide([],_Tasks).
omezeni_lide([clovek(_Id,Kapacita,UkolyCloveka)|Lide],Tasks) :-
        omezeni_clovek(UkolyCloveka,Kapacita,Tasks),
        omezeni_lide(Lide,Tasks).
```


Plánování a lidé (pokračování)

Napište predikát omezeni_clovek(UkolyCloveka,Kapacita,Tasks), který ze seznamu Tasks vybere úlohy určené seznamem UkolyCloveka a pro takto vybrané úlohy sešle omezení cumulative/2 s danou kapacitou člověka Kapacita.

Pro nalezení úlohy v Tasks lze použít nth1(N,Tasks,NtyPrvek) z knihovny

```
:- use_module(library(lists)).
```


Všechna řešení, třídění, rozdílové seznamy

Všechna řešení

```
% z(Jmeno,Prijmeni,Pohlavi,Vek,Prace,Firma)
z(petr,novak,m,30,skladnik,skoda).  z(pavel,jirku,m,40,mechanik,skoda).
z(rostislav,lucensky,m,50,technik,skoda).  z(alena,vesela,z,25,sekretarka,skoda).
z(jana,dankova,z,35,asistentka,skoda).  z(hana,jirku,z,35,kucharka,zs_stara).
z(roman,maly,m,35,manazer,cs).  z(alena,novotna,z,40,ucitelka,zs_stara).
z(david,jirku,m,30,ucitel,zs_stara).  z(petra,spickova,z,45,uklizecka,zs_stara).
```

- Najděte jméno a příjmení všech lidí.

```
?- findall(Jmeno-Prijmeni, z(Jmeno,Prijmeni,_S,_V,_Pr,_F),L).
?- bagof( Jmeno-Prijmeni, [S,V,Pr,F] ^ z(Jmeno,Prijmeni,S,V,Pr,F) , L).

?- bagof( Jmeno-Prijmeni, [V,Pr,F] ^ z(Jmeno,Prijmeni,S,V,Pr,F) ,  L ).
?- bagof( Jmeno-Prijmeni, [V,Pr,F] ^ z(Jmeno,Prijmeni,_S,V,Pr,F) ,  L).
```

- Najděte jméno a příjmení všech zamestnanců firmy skoda a cs

```
?- findall( c(J,P,Firma), ( z(J,P,_,_,_,Firma), ( Firma=skoda ; Firma=cs ) ),
?- bagof( J-P, [S,V,Pr]^(z(J,P,S,V,Pr,F),( F=skoda ; F=cs ) ) ,  L ).
?- setof( P-J, [S,V,Pr]^(z(J,P,S,V,Pr,F),( F=skoda ; F=cs ) ) ,  L ).
```


Všechna řešení

Kolik žen a mužů je v databázi?

```
?- findall( c(P,J), z(P,J,z,_,_,_), L), length(L,N).
?- findall( c(P,J), z(P,J,m,_,_,_), L), length(L,N).

?- bagof(c(P,J), [Ve,Pr,Fi]^z(P,J,S,Ve,Pr,Fi), L), length(L,N).

?- findall( S-N, ( bagof(c(P,J), [Ve,Pr,Fi]^z(P,J,S,Ve,Pr,Fi), L),
                   length(L,N)
                 ), Dvojice ).
```


Všechna řešení: příklady

1. Jaká jsou příjmení všech žen?
2. Kteří lidé mají více než 30 roků? Nalezněte jejich jméno a příjmení.
3. Nalezněte abecedně seřazený seznam všech lidí.
4. Nalezněte příjmení vyučujících ze zs_stara.
5. Jsou v databázi dva bratři (mají stejné příjmení a různá jména) \= vs. @<
6. Které firmy v databázi mají více než jednoho zaměstnance?

bubblesort(S,Sorted)

Seznam S seřad'te tak, že

- nalezněte první dva sousední prvky X a Y v S tak, že X>Y,
  vyměňte pořadí X a Y a získate S1; swap(S,S1)
  a seřad'te S1 rekurzivně bubblesortem
- pokud neexistuje žádný takový pár sousedních prvků X a Y,
  pak je S seřazený seznam

```
bubblesort(S,Sorted) :-
        swap(S,S1), !,            % Existuje použitelný swap v S?
        bubblesort(S1, Sorted).
bubblesort(Sorted,Sorted).         % Jinak je seznam seřazený

swap([X,Y|Rest],[Y,X|Rest]) :-    % swap prvních dvou prvků
        X>Y.                       % nebo obecněji X@>Y, resp. gt(X,Y)
swap([X|Rest],[X|Rest1]) :-        % swap prvků až ve zbytku
        swap(Rest,Rest1).
```

quicksort(S,Sorted)

Neprázdný seznam S seřad'te tak, že — konec rekurze pro S=[]

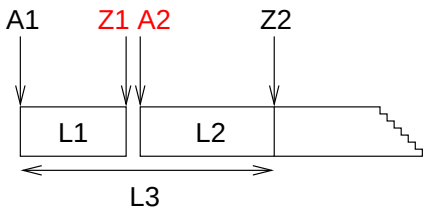
- vyberte nějaký prvek X z S; — např. vyberte hlavu S
  rozdělte zbytek S na dva seznamy Small a Big tak, že:
  v Big jsou větší prvky než X a v Small jsou zbývající prvky — split(X,Seznam,Small,Big)
- seřad'te Small do SortedSmall — rekurzivně quicksortem
- seřad'te Big do SortedBig — rekurzivně quicksortem
- setříděný seznam vznikne spojením SortedSmall a [X|SortedBig] — append

DÚ: insertsort(S,Sorted)

Neprázdný seznam S=[X|T] seřad'te tak, že — konec rekurze pro S=[]

- seřad'te tělo T seznamu S — rekurzivně insertsortem
- vložte hlavu X do seřazeného těla tak, — insert(X,SortedT,Sorted)
  že výsledný seznam je zase seřazený.
  Víme: výsledek po vložení X je celý seřazený 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( [1,2,3|Z1]-Z1, [4,5|Z2]-Z2, A1-[]).`
- `append( A1-Z1, Z1-Z2, A1-Z2 ).`
  `L1 L2 L3`
  `append( [1,2,3,4,5]-[4,5], [4,5]-[], [1,2,3,4,5]-[] ).`

reverse(Seznam, Opacny)

```
% kvadratická složitost
reverse( [], [] ).
reverse( [ H | T ], Opacny ) :-
        reverse( T, OpacnyT ),
        append( OpacnyT, [ H ], Opacny ).


% lineární složitost, rozdílové seznamy
reverse( Seznam, Opacny ) :- reverse0( Seznam, Opacny-[] ).
reverse0( [],  S-S ).
reverse0( [ H | T ], Opacny-OpacnyKonec  ) :-
         reverse0( T,  Opacny-[ H | OpacnyKonec]  ).
```

`quicksort` pomocí rozdílových seznamů

Neprázdný seznam S seřad'te tak, že

- vyberte nějaký prvek X z S;
  rozdělte zbytek S na dva seznamy Small a Big tak, že:
  v Big jsou větší prvky než X a v Small jsou zbývající prvky
- seřad'te Small do SortedSmall
- seřad'te Big do SortedBig
- setříděný seznam vznikne spojením SortedSmall a [X|SortedBig]

DÚ: `palindrom(L)`

Napište predikát palindrom(Seznam), který uspěje pokud se Seznam čte stejně zezadu i zepředu, př. [a,b,c,b,a] nebo [12,15,1,1,15,12]

Poděkování

Průsviky ze cvičení byly připraveny na základě materiálů dřívějších cvičících tohoto předmětu.

Speciální poděkování patří

- Adrianě Strejčkové

Další podklady byly připraveny

- Alešem Horákem
- Miroslavem Nepilem
- Evou Žáčkovou
- Janem Ryglem