IB013 Logické programování I

(průsvitky ze cvičení)

Hana Rudová

jaro 2010

Backtracking, unifikace, aritmetika

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

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.

Sémantika logického programu:
procedury ≡ databáze faktů a pravidel ≡ logické formule

Sicstus Prolog minimum I

- **Spuštění interpretu:** V unixu přidáme modul

  `module add sicstus`

  a spustíme příkazem

  `sicstus`

  Pracovním adresářem je aktuální (tam kde byl spuštěn).
  V MS Windows standardně z nabídky Start/Programs nebo pomocí ikony, nastavíme pracovní adresář pomocí File/Working directory, v případě potřeby nastavíme font Settings/Font a uložíme nastavení Settings/Save settings.

Sicstus Prolog minimum II

- **Načtení programu:** tzv. konzultace

  Editor není integrován, takže program editujeme externě ve svém oblíbeném editoru. Pak ho načteme z příkazové řádky v interpretu příkazem

  ```
  ?- consult(jmeno).
  ```

  nebo pomocí zkrácené syntaxe

  ```
  ?- [jmeno].  % (předpokládá se přípona .pl)
  ```

  pokud uvádíme celé jméno případně cestu, dáváme jej do apostrofů

  ```
  ?- ['D:\prolog\moje\programy\jmeno.pl'].
  ```

  V MS Windows lze také pomocí nabídky File/Consult

Sicstus Prolog minimum III

- **Spouštění programů/procedur/predikátů** je zápis dotazů, př.

```
?- muj_predikat(X,Y).
?- suma(1,2,Y), vypis('Vysledek je ',Y).
```

  Každý příkaz ukončujeme tečkou.

- **Přerušení a zastavení cyklícího programu**: Ctrl-C
- **Ukončení interpretu příkazem**

```
?- halt.
```

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: řešení příkladů

Středníkem si vyžádáme další řešení

```
| ?- otec(petr,lenka).
yes
| ?- otec(petr,jan).
no
| ?- otec(Kdo,petr).
Kdo = adam ? ;
no
| ?- rodic(pavla,Dite).
Dite = petr ? ;
Dite = tomas ? ;
no
| ?- otec(petr,Dcera),zena(Dcera).
Dcera = lenka ? ;
no
```

```
| ?- otec(Otec,Syn),muz(Syn).
Syn = filip,
Otec = petr ? ;
Syn = jan,
Otec = pavel ? ;
Syn = petr,
Otec = adam ? ;
Syn = michal,
Otec = tomas ? ;
Syn = radek,
Otec = michal ? ;
no
| ?-
```

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)`

Řešení:

```
prababicka(Prababicka,Pravnouce):-
        rodic(Prababicka,Prarodic),
        zena(Prababicka),
        rodic(Prarodic,Rodic),
        rodic(Rodic,Pravnouce).
```

Backtracking: řešení příkladů II

```
nevlastni_bratr(Bratr,Sourozenec):-
        rodic_v(X,Bratr),
        muz(Bratr),
        rodic_v(X,Sourozenec),
        /* tento test neni nutny,
           ale zvysuje efektivitu */
        Bratr \== Sourozenec,
        rodic_v(Y,Bratr),
        Y \== X,
        rodic_v(Z,Sourozenec),
        Z \== X,
        Z \== Y.
```

```
/* nevhodne umisteni testu -
vypocet "bloudi" v neuspesnych vetvich */
nevlastni_bratr2(Bratr,Sourozenec):-
        rodic_v(X,Bratr),
        rodic_v(X,Sourozenec),
        rodic_v(Y,Bratr),
        rodic_v(Z,Sourozenec),
        Y \== X,
        Z \== X,
        Z \== Y,
        muz(Bratr).
```

Backtracking: prohledávání stavového prostoru

- Zkuste předem odhadnout (odvodit) pořadí, v jakem budou nalezeni potomci Pavly?
- Jaký vliv má pořadí klauzulí a cílu v predikátu `potomek/2` na jeho funkci?
- 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`

Backtracking: řešení III

```
/* varianta 1a */
potomek(Potomek,Predek):-rodic(Predek,Potomek).
potomek(Potomek,Predek):-rodic(Predek,X),potomek(Potomek,X).

/* varianta 1b - jine poradi odpovedi, neprimi potomci maji prednost */
potomek(Potomek,Predek):-rodic(Predek,X),potomek(Potomek,X).
potomek(Potomek,Predek):-rodic(Predek,Potomek).

/* varianta 2a - leva rekurze ve druhe klauzuli,
na dotaz potomek(X,pavla) vypise odpovedi, pak cykli */
potomek(Potomek,Predek):-rodic(Predek,Potomek).
potomek(Potomek,Predek):-potomek(Potomek,X),rodic(Predek,X).

/* varianta 2b - leva rekurze v prvni klauzuli,
na dotaz potomek(X,pavla) hned cykli */
potomek(Potomek,Predek):-potomek(Potomek,X),rodic(Predek,X).
potomek(Potomek,Predek):-rodic(Predek,Potomek).
```

```
| ?- nevlastni_bratr(X,Y).
petr? petr? petr? petr? eva? petr? jana?
X = filip,
Y = lenka ? ;
petr? pavel? pavel? adam? adam? tomas? tomas? michal? michal? eva? eva? jana?
pavla? pavla? pavla? adam? pavla? pavla? pavla? pavla? pavla? pavla? lenka?
no
| ?- nevlastni_bratr2(X,Y).
petr? petr? petr? petr? eva? eva? petr? eva? petr? petr? petr? jana? eva? petr? jana?
X = filip,
Y = lenka ? ;
petr? petr? petr? petr? eva? jana? petr? eva? petr? petr? petr? jana? jana? petr?
jana? pavel? pavel? pavel? pavel? adam? adam? adam? adam?  pavla? pavla? adam?
pavla? tomas? tomas? tomas? tomas? michal? michal? michal? michal? eva? eva? petr?
petr? eva? eva? petr? eva? jana? jana? petr? petr? jana? jana? petr? jana? pavla?
pavla? adam? adam? pavla? pavla? adam? pavla? pavla? adam? pavla? pavla? pavla?
pavla? pavla? pavla? adam? pavla? pavla? pavla? pavla? lenka? lenka? lenka? lenka?
no
```

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))`

Neuspěje volání 1) a 3), ostatní ano, cyklické struktury vzniknou v případech 4),7) a 8) přestože u posledních dvou mají levá a pravá strana unifikace disjunktní množiny jmen proměnných.

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 unifkován 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.
```

s1/3 je vícesměrný - umí sčítat, odečítat, generovat součty, ale pracuje jen s nezápornými celými čísly

s2/3 umí pouze sčítat, ale také záporná a reálná čísla

Operátory

Definice operátorů umožňuje přehlednější infixový zápis binárních a unárních predikátů, příklad: definice op(1200,Y,':-') umožňuje zápis

```
a:-print(s(s(0))),b,c).
```

pro výraz

```
:-(a,,(print(s(s(0))),,(b,c))).
```

Prefixovou notaci lze získat predikátem display/1. Vyzkoušejte

```
display((a:-print(s(s(0))),b,c)).
display(a+b+c-d-e*f*g-h+i).
display([1,2,3,4,5]).
```

Definice standardních operátorů najdete na konci manuálu.

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]`

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 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 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**

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 navrací až k volající proceduře r/1, viz (1)
- alternativy vzniklé po provedení řezu se zachovávají - další možnosti predikátu c/1 viz (2) a (4)

```
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).

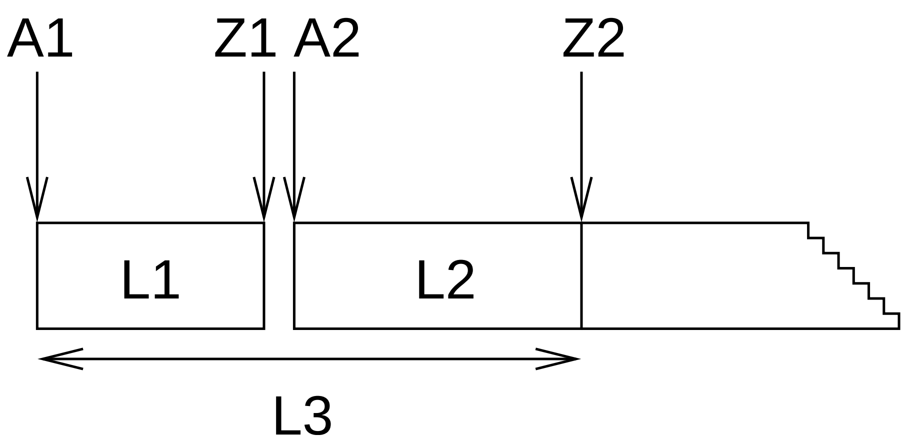
c(X):- X mod 2 =:= 0, write(c1).
c(X):- X mod 3 =:= 0, write(c2).

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=0,r(X).
r1
X = 0 ? ;
p1r2
X = 0 ? ;
a1a2p3r2
X = 0 ? ;
r3
X = 0 ? ;
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
```

Řez: maximum

Je tato definice predikátu max/3 korektní?

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

Není, následující dotaz uspěje: ?- `max(2,1,1).`

Uveďte 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 verzi se tvrdilo: X=Z ∧ X>=Y => Z=X

správná definice je: X>=Y => Z=X

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

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).
```

- 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? mem4(H,[K|_]) :- H==K. mem4(H,[K|T]) :- 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( _X, [], [] ).
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

Všechna řešení,

třídění, rozdílové seznamy

Všechna řešení

```
% z(Jmeno,Prijmeni,Pohlavi,Vek,Prace,Firma)
z(petr,novak,m,30,skladnik,skoda).  z(pavel,novy,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(lenka,merinska,z,35,ucetni,skoda).
z(roman,maly,m,35,manazer,cs).  z(alena,novotna,z,40,ucitelka,zs_stara).
z(david,novy,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,_,_,_,_),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 ).
```

- Najděte jméno a příjmení všech zamestnanců firmy skoda a cs

```
?- findall( c(J,P,Firma), ( z(J,P,_,_,_,Firma), ( Firma=skoda ; Firma=cs ) ),  L ).
?- 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: 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)?
6. Které firmy v databázi mají více než jednoho zaměstnance?

```
1. findall(Prijmeni, z(_,Prijmeni,z,_,_,_), L).
2. findall(Jmeno-Prijmeni, ( z(Jmeno,Prijmeni,_,Vek,_,_), Vek>30 ), L).
3. setof(P-J, [S,V,Pr,F]^z(J,P,S,V,Pr,F),  L ).
4. findall(Prijmeni, ( z(_,Prijmeni,_,_,P,zs_stara), (P=ucitel;P=ucitelka) ), L).
5. findall(b(J1-P,J2-P), ( z(J1,P,m,_,_,_),z(J2,P,m,_,_,_), J1@<J2 ), L).
6. findall(F-Pocet, ( bagof(P, [J,S,V,Pr]^z(J,P,S,V,Pr,F), L),
                       length(L,Pocet), Pocet>1
                     ), S).
```

bubblesort(S,Sorted)

Seznam S seřaďte 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 seřaďte 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|Rest1]) :-   % swap prvních dvou prvků
        X>Y.                      % nebo obecněji X@>Y, resp. gt(X,Y)
swap([Z|Rest],[Z|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

```
quicksort([],[]).
quicksort([X|T], Sorted) :- split(X, Tail, Small, Big),
                            quicksort(Small, SortedSmall),
                            quicksort(Big, SortedBig),
                            append(SortedSmall, [X|SortedBig], Sorted).
split(X, [], [], []).
split(X, [Y|T], [Y|Small], Big) :- X>Y, !, split(X, T, Small, Big).
split(X, [Y|T], Small, [Y|Big]) :- split(X, T, Small, Big).
```

DÚ: `insertsort(S,Sorted)`

Neprázdný seznam S=[X|T] seřaďte tak, že — konec rekurze pro S=[]

- seřaďte 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.

```
insertsort([],[]).

insertsort([X|T],Sorted) :-
      insertsort(T,SortedT),           % seřazení těla
      insert(X,SortedT,Sorted).        % vložení X na vhodné místo

insert(X,[Y|Sorted],[Y|Sorted1]) :-
      X > Y, !,
      insert(X,Sorted,Sorted1).
insert(X,Sorted,[X|Sorted]).
```

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]  ).
```

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]

```
palindrom(Seznam) :- reverse(Seznam,Seznam).
```

`quicksort` pomocí rozdílových seznamů

Neprázdný seznam S seřaďte 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
- seřaďte Small do SortedSmall
- seřaďte Big do SortedBig
- setříděný seznam vznikne spojením SortedSmall a [X|SortedBig]

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

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

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`
- 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).
|:
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
```

Implementace: vstup/výstup

```
uloz_do_souboru( Soubor ) :-
       seeing( StaryVstup ),
       telling( StaryVystup ),
       see( user ),
       tell( Soubor ),
       repeat,
          read( Term ),
          process_term( Term ),
          Term == end_of_file,
       !,
       seen,
       told,
       tell( StaryVystup ),
       see( StaryVstup ).

process_term(end_of_file) :- !.
process_term( Term ) :-
       write( Term ), write('.'), nl.
```

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).
|:
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
```

Databázové operace: implementace

```
vytvor_program :-
       seeing( StaryVstup ),
       see( user ),
       repeat,
          read( Term ),
          uloz_term( Term ),
          Term == end_of_file,
       !,
       seen,
       see( StaryVstup ).

uloz_term( end_of_file ) :- !.
uloz_term( Term ) :-
        assert( Term ).
```

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 tělo
                                          %      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 aT 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), není seznam (T\=[_|_]), a S je podtermem některého argumentu T.
  - pokud nepoužijeme (T\=[_|_]), pak dotaz :- subterm(a,[1]). cyklí!
  - pokud nepoužijeme (compound/1), pak dotaz :- subterm(1,2). cyklí!

```
| ?- subterm(sin(3),b(c,2,[1,b],sin(3),a)).                    yes
subterm(T,T) :- !.
subterm(S,[H|_]) :- subterm(S,H), !.
subterm(S,[_|T]) :- subterm(S,T),!.
subterm(S,T) :- compound(T), T\=[_|_], T=..[_|Argumenty], subterm(S,Argumenty).
```

`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
```

```
same(A,B) :- var(A), var(B), !.
same(A,B) :- var(A), !, fail.
same(A,B) :- var(B), !, fail.
same(A,B) :- atomic(A), atomic(B), !, A==B.
same([HA|TA],[HB|TB]) :- !, same(HA,HB), same(TA,TB).
same(A,B) :- A=..[FA|ArgA], B=..[FB|ArgB], FA==FB, same(ArgA,ArgB).
```

D.Ú. `unify(A,B)`

Napište predikát `unify(A,B)`, který unifikuje termy A a B a provede zároveň *kontrolu výskytu*.

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

unify(A,B) :- var(A), var(B), !,  A=B.
unify(A,B) :- var(A), !, not_occurs(A,B), A=B.
unify(A,B) :- var(B), !, not_occurs(B,A), B=A.
unify(A,B) :- atomic(A), atomic(B), !, A==B.
unify([HA|TA],[HB|TB]) :- !, unify(HA,HB), unify(TA,TB).
unify(A,B) :- A=..[FA|ArgA], B=..[FB|ArgB], FA==FB, unify(ArgA,ArgB).
```

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

```
not_occurs(_,B) :- atomic(B), !.
not_occurs(A,B) :- var(B), !, A\==B.
not_occurs(A,[H|T]) :- !, not_occurs(A,H), not_occurs(A,T).
not_occurs(A,B) :- B=..[_|Arg], not_occurs(A,Arg).
```

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

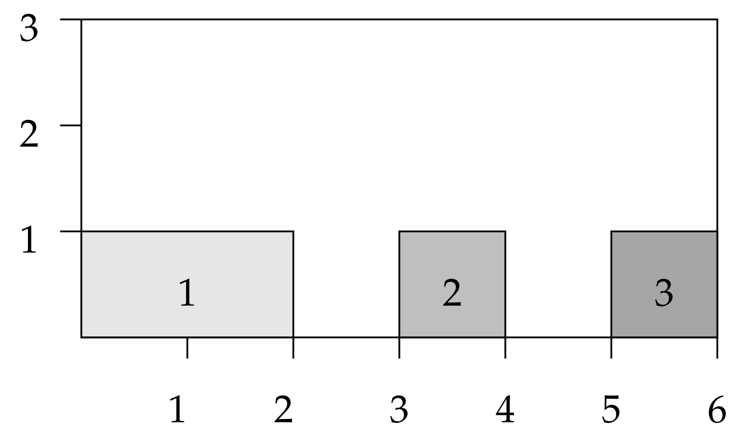
- Knihovna pro CLP(FD) `:- use_module(library(clpfd)).`
- Domény proměnných `domain( Seznam, MinValue, MaxValue )`
- Omezení `all_distinct( Seznam )`
- Aritmetické omezení `A*B + C #= D`
- Procedura pro prohledávání stavového prostoru `labeling([],Seznam)`

Algebrogram: řešení

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

donald(LD):-
        % domény
        LD=[D,O,N,A,L,G,E,R,B,T],
        domain(LD,0,9),
        domain([D,G,R],1,9),
        % omezení
        all_distinct(LD),
           100000*D + 10000*O + 1000*N + 100*A + 10*L + D +
           100000*G + 10000*E + 1000*R + 100*A + 10*L + D
        #= 100000*R + 10000*O + 1000*B + 100*E + 10*R + T,
        % prohledávání stavového prostoru
        labeling([],LD).
```

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).

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)

nastav_domeny([],[],[]).
nastav_domeny([ukol(Id,Doba,MinStart,MaxKonec)|Ukoly],[Z|Zacatky],
              [task(Z,Doba,K,1,Id)|Tasks]) :-
        MaxStart is MaxKonec-Doba,
        Z in MinStart..MaxStart,
        K #= Z + Doba,
        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).
```

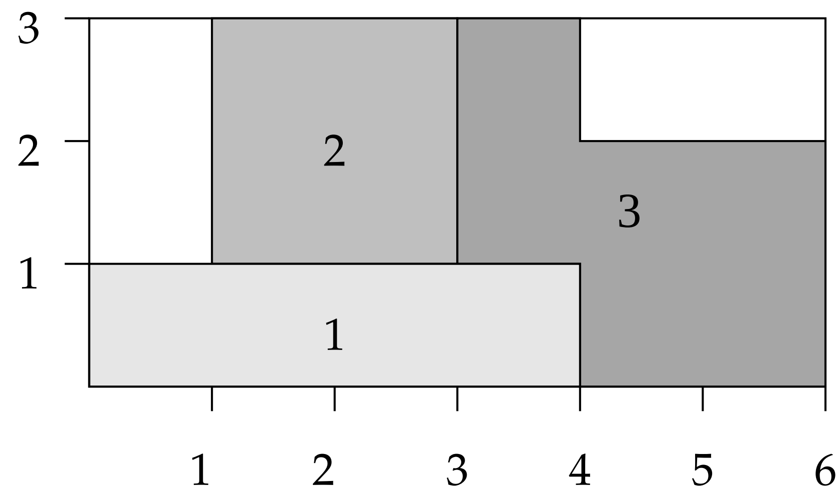
Pro určení úlohy v Tasks lze použít `nth1(N,Seznam,NtyPrvek)` z knihovny

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

precedence(Tasks) :- findall(prec(A,B),prec(A,B),P),
                     omezeni_precedence(P,Tasks).

omezeni_precedence([],_Tasks).
omezeni_precedence([prec(A,B)|Prec],Tasks) :-
        nth1(A,Tasks,task(ZA,DA,_KA,1,A)),
        nth1(B,Tasks,task(ZB,_DB,_KB,1,B)),
        ZA + DA #=< ZB,
        omezeni_precedence(Prec,Tasks).
```

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)

```
% 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)).

omezeni_clovek(UkolyCloveka,Kapacita,Tasks) :-
        omezeni_clovek(UkolyCloveka,Kapacita,Tasks,[]).

omezeni_clovek([],Kapacita,_Tasks,TasksC) :-
        cumulative(TasksC,[limit(Kapacita)]).
omezeni_clovek([U|UkolyCloveka],Kapacita,Tasks,TasksC) :-
        nth1(U,Tasks,TU),
        omezeni_clovek(UkolyCloveka,Kapacita,Tasks,[TU|TasksC]).
```

Stromy, grafy

Stromy

Uzly stromu Tree jsou reprezentovány termy

- tree(Left,Value,Right): Left a Right jsou opět stromy, Value je ohodnocení uzlu
- leaf(Value): Value je ohodnoceni uzlu
- Příklad:

```
      6
     / \
    /   \
   /     \
  2       8
 / \     /
1   4   7
   / \
  3   5
```

```
tree(tree(leaf(1), 2, tree(leaf(3),4,leaf(5)) ), 6, tree(leaf(7),8,[]) )
```

Stromy: hledáni prvku in(X,T)

Predikát in(X,T) uspěje, pokud se prvek X nachází ve stromu T.

Prvek X se nachází ve stromě T, jestliže

- X je listem stromu T, jinak leaf(X)
- X je kořen stromu T, jinak tree(Left,X,Right)
- X je menší než kořen stromu T, pak se nachází v levém podstromu T, jinak
- X se nachází v pravém podstromu T

```
in(X, leaf(X)) :- !.
in(X, tree(_,X,_)) :- !.
in(X, tree(Left, Root, Right) ) :-
        X<Root, !,
        in(X,Left).
in(X, tree(Left, Root, Right) ) :-
        in(X,Right).
```

Stromy: přidávání `add(T,X,TWithX)`

Prvek X přidej do stromu T jednou z následujících možností:

- pokud T = [], pak je nový strom leaf(X)
- pokud T=leaf(V) a X>V, pak vznikne nový strom s kořenem V, vpravo má leaf(X) (vlevo je [])

  pokud T=leaf(V) a X<V, pak vznikne nový strom s kořenem V, vlevo má leaf(X) (vpravo je [])
- pokud T=tree(L,V,_) a X>V, pak v novém stromě L ponechej a X přidej doprava (rekurzivně)

  pokud T=tree(_,V,R) a X<V, pak v novém stromě R ponechej a X přidej doleva (rekurzivně)

```
add([],X,leaf(X)) :- !.
add(leaf(V), X, tree([],V,leaf(X) ) :- X>V, !.
add(leaf(V), X, tree(leaf(X),V,[]) ) :- !.
add(tree(L,V,R), X, tree(L,V,R1)) :- X>V, !, add(R,X,R1).
add(tree(L,V,R), X, tree(L1,V,R)) :- add(L,X,L1).
```

Procházení stromů

Napište predikát traverse(Tree, List), který projde traversálně strom Tree. Seznam List pak obsahuje všechny prvky tohoto stromu.

Pořadí preorder: nejprve uzel, pak levý podstrom, nakonec pravý podstrom

```
?- traverse(tree(tree(leaf(1),2,tree(leaf(3),4,leaf(5))),6,
          tree(leaf(7),8,leaf(9))), [6,2,1,4,3,5,8,7,9]).
```

(preorder)

```
traverse(T,Pre):- t_pre(T,Pre,[]).

t_pre([],S,S).
t_pre(leaf(V),[V|S],S).
t_pre(tree(L,V,R),[V|S],S2):-
        t_pre(L,S,S1),
        t_pre(R,S1,S2).
```

```
          6
         / \
        /   \
       /     \
      2       8     % V=2, S=[1,4,3,5|S2]
     / \     / \    % S=[1|S1]
    1   4   7   9   % S1=[4,3,5|S2]
       / \
      3   5
```

Použit princip rozdílových seznamů

Procházení stromů

```
traverse(T,Pre):- t_pre(T,Pre,[]).

t_pre([],S,S).
t_pre(leaf(V),[V|S],S).
t_pre(tree(L,V,R),[V|S],S2):-
        t_pre(L,S,S1),
        t_pre(R,S1,S2).
```

```
          6
         / \
        /   \
       /     \
      2       8     % V=2, S=[1,4,3,5|S2]
     / \     / \    % S=[1|S1]
    1   4   7   9   % S1=[4,3,5|S2]
       / \
      3   5
```

Modifikuje algoritmus tak, aby byly uzly vypsány v pořadí inorder (nejprve levý podstrom, pak uzel a nakonec pravý podstrom), tj. [1,2,3,4,5,6,7,8,9]

```
traverse(T,In):- t_in(T,In,[]).

t_pre([],S,S).
t_in(leaf(V),[V|S],S).
t_in(tree(L,V,R),S,S2):-
        t_in(L,S,[V|S1]),
        t_in(R,S1,S2).
```

DÚ: Procházení stromu postorder

Modifikuje algoritmus tak, aby byly uzly vypsány v pořadí postorder (nejprve levý podstrom, pak pravý podstrom a nakonec uzel), tj. [1,3,5,4,2,7,9,8,6]

```
traverse_post(T,Post):-
        t_post(T,Post,[]).

t_pre([],S,S).
t_post(leaf(V),[V|S],S).
t_post(tree(L,V,R),S,S2):-
        t_post(L,S,S1),
        t_post(R,S1,[V|S2]).
```

```
          6
         / \
        /   \
       /     \
      2       8
     / \     / \
    1   4   7   9
       / \
      3   5
```

Reprezentace grafu

- Reprezentace grafu: pole následníků uzlů
- Grafy nebudeme modifikovat, tj. pro reprezentaci pole lze využít term
- (Orientovany) neohodnocený graf

```
graf([2,3],[1,3],[1,2]).          graf([2,4,6],[1,3],[2],[1,5],[4,6],[1,5]).

     1--2                             5--4
      \ |                             |  |
       \|                             6--1--2--3
        3
?- functor(Graf,graf,PocetUzlu).
?- arg(Uzel,Graf,Sousedi).
```

- (Orientovany) ohodnocený graf   [Soused-Ohodnoceni|Sousedi]

```
graf([2-1,3-2],[1-1,3-2],[1-2,2-2]).
graf([2-1,4-3,6-1],[1-1,3-2],[2-2],[1-3,5-1],[4-1,6-2],[1-1,5-2]).
```

Procházení grafu do hloubky

Napište predikát dfs(Uzel,Graf,Parents) pro procházení grafu Graf do hloubky z uzlu Uzel. Výsledkem je datová struktura Parents, která reprezentuje strom vzniklý při prohledávání do hloubky (pro každý uzel stromu známe jeho rodiče).

Datová struktura pro rodiče uzlů:

- při reprezentaci rodičů lze využít term s aritou odpovídající počtu uzlů
- iniciálně jsou argumentu termu volné proměnné
- na závěr je v N-tém argumentu uložen rodič N-tého uzlu
  (iniciální uzel označíme empty)

```
graf([2,3],[1,3],[1,2]).       graf([2,4,6],[1,3],[2],[1,5],[4,6],[1,5]).

     1--2    1--2                 5--4           5  4
      \ |     \                   |  |           |  |
       \|      \                  6--1--2--3     6--1--2--3
        3       3                           U=4: rodic(4, 1, 2, empty, 6, 1)
                U=2: rodic(2,empty,1)
```

Procházení grafu do hloubky: algoritmus I

Procházení grafu z uzlu Uzel

- Vytvoříme term pro rodiče (všichni rodiči jsou zatím volné proměnné)
- Uzel Uzel má prázdného rodiče a má sousedy Sousedi
- Procházíme (rekurzivně) všechny sousedy v Sousedi

```
dfs(Uzel,Graf,Parents) :-
        functor(Graf,graf,Pocet),
        functor(Parents,rodice,Pocet),
        arg(Uzel,Parents,empty),
        arg(Uzel,Graf,Sousedi),
        prochazej_sousedy(Sousedi,Uzel,Graf,Parents).
```

Procházení grafu do hloubky: algoritmus II

Procházení sousedů uzlu Uzel (pokud Uzel nemá sousedy, tj. Sousedi=[], končíme)

1. Uzel V je první soused
2. Zjistíme rodiče uzlu V ... pomocí arg(V,Parents,Rodic)
3. Pokud jsme V ještě neprošli (tedy nemá rodiče a platí var(Rodic)), tak
   (a) nastavíme rodiče uzlu V na Uzel
   (b) rekurzivně procházej všechny sousedy uzlu V
4. Procházej zbývající sousedy uzlu Uzel

```
prochazej_sousedy([],_,_,_).
prochazej_sousedy([V|T],Uzel,Graf,Parents) :- arg(V,Parents,Rodic),
                                 ( nonvar(Rodic) -> true
                                 ; Rodic = Uzel,
                                   arg(V,Graf,SousediV),
                                   prochazej_sousedy(SousediV,V,Graf,Parents)
                                 ),
                                 prochazej_sousedy(T,Uzel,Graf,Parents).
```

DÚ: Procházení grafu do šířky

Napište predikát bfs(U,G,P) pro procházení grafu G do šířky z uzlu U. Výsledkem procházení je datová struktura P, která reprezentuje strom vzniklý při prohledávání grafu G do šířky (pro každý uzel stromu známe jeho rodiče).

```
graf([2,3],[1,3],[1,2]).      graf([2,4,6],[1,3],[2],[1,5],[4,6],[1,5]).

    1--2   1--2               5--4          5--4
     \ |      |               |  |             |
      \|      |               6--1--2--3    6--1--2--3
       3      3                           U=4: rodic(4, 1, 2, empty, 4, 1)
          U=2: rodic(2,empty,2)
```

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