Aritmetika, seznamy, řez

Aritmetika

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?

- X = Y + 1
- X is Y + 1
- X = Y
- X == Y
- 1 + 1 = 2
- 2 = 1 + 1
- 1 + 1 = 1 + 1
- 1 + 1 is 1 + 1
- 1 + 2 =:= 2 + 1
- X \== Y
- X =\= Y
- 1 + 2 =\= 1 - 2
- 1 <= 2
- 1 =< 2
- sin(X) is sin(2)
- sin(X) = sin(2+Y)
- sin(X) =:= sin(2+Y)

Seznamy a append

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

Napište následující predikáty pomocí append/3:

- ```
  last( X, S ) :- append( _S1, [X], S).
  append([3,2], [6], [3,2,6]).      X=6, S=[3,2,6]
  ```
- ```
  prefix( S1, S2 ) :- append( S1, _S3, S2).
  DÚ: suffix(S1,S2)
  ```
- ```
  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

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).
```

```
| ?- 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(
r1
X = -6 ? ;
p1r2
X = -6 ? ;
a1b2c1d1p2r2
X = -6 ? ;
d2p2r2
X = -6 ? ;
c2d1p2r2
X = -6 ? ;
d2p2r2
X = -6 ? ;
r3
X = -6 ? ;
no
```



```
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 navracen
např. pomocí následujících dotazů (vždy s
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 alternativ predikátu r/1
- dokud nebyl proveden řez, alternativ predikátu a/1 se uplatňují, př. neúspěc b/1 v dotazu (3)
- při neúspěchu cíle za řezem se výpoče navrací až k volající proceduře r/1, viz (1
- alternativy vzniklé po provedení řezu s zachovávají - další možnosti predikát c/1 viz (2) a (4)

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,Z):-X>=Y,!,Z=X.
max(X,Y,Y):-Y>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|_]) :- !.                mem3(H,[K|_]) :- H==K.
mem2(H,[_|T]) :- mem2(H,T).        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).

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

Třídění, rozdílové seznamy

`bubblesort(S,Sorted)`

Seznam S setřiďte tak, že

- nalezněte první dva sousední prvky X a Y v S tak, že X>Y,
  vyměňte pořadí X a Y a získate S1;
  a setřiďte S1
- pokud neexistuje žádný takový pár sousedních prvků X a Y,
  pak je S setříděný seznam

```
swap([X,Y|Rest],[Y,X|Rest1]) :-
        X>Y.                      % nebo obecněji X@>Y pomocí gt(X,Y)
swap([Z|Rest],[Z|Rest1]) :-
        swap(Rest,Rest1).

bubblesort(S,Sorted) :-
        swap (S,S1), !,
        bubblesort(S1, Sorted).
bubblesort(Sorted,Sorted).
```

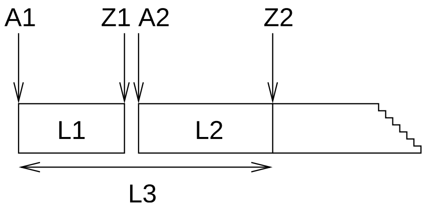

`quicksort(S,Sorted)`

Neprázdný seznam S setřiďte tak, že

- smažte 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
- setřiďte Small do SortedSmall
- setřiďte Big do SortedBig
- setříděný seznam vznikne spojením SortedSmall a [X|SortedBig]

```
quicksort([],[]).                  + ošetření případu, kdy S je prázdný seznam
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).
```


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, S ).`
- `append( A1-Z1, Z1-Z2, A1-Z2 ).`
  `           L1     L2     L3`

`reverse(Seznam, Opacny)`

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


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


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

quicksort pomocí rozdílových seznamů

Neprázdný seznam S setřiďte tak, že

- smažte 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
- setřiďte Small do SortedSmall
- setřiďte 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-Z1, Z1-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
| ?- [soubor].
% consulting /home/hanka/soubor.pl...
% consulted /home/hanka/soubor.pl in module user, 0 msec
% 376 bytes
yes
| ?- listing(fakt/2).
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: 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).
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 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 aT bez proměnných, které uspějí, pokud je S podtermem termu T. Tj. musí platit alespoň jedno z

- subterm S je právě term T NEBO
- subterm S se nachází v hlavě seznamu T NEBO
- subterm S se nachází v těle seznamu T NEBO
- T je složený term (`compound/1`), není seznam (`T\=[_|_]`),
  a S je podtermem některého argumentu T.

```
| ?- 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).
```


unify(A,B)

Napište predikát unify(A,B), který unifikuje termy A a B.

```
| ?- 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 pd 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( Cifry ) :-
      domain(...),
      constraints(...),
      labeling(...).
```

- 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([],[X1, X2, X3])`

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).
```


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,Doby),
                  serialized(Zacatky,Doby),
                  labeling([],Zacatky).

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

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

```
nastav_domeny([],[],[]).
nastav_domeny([U|Ukoly],[Z|Zacatky],[Doba|Doby]) :-
        U=ukol(_Id,Doba,MinStart,MaxKonec),
        MaxStart is MaxKonec-Doba,
        Z in MinStart..MaxStart,
        nastav_domeny(Ukoly,Zacatky,Doby).
```


Plánování a precedence

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 zjištění parametrů úlohy lze použít např. nth(N,Seznam,NtyPrvek) z knihovny

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

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

omezeni_precedence([],_Zacatky,_Doby).
omezeni_precedence([prec(A,B)|Prec],Zacatky,Doby) :-
        nth(A,Zacatky,ZA), nth(B,Zacatky,ZB), nth(A,Doby,DA),
        ZA + DA #< ZB,
        omezeni_precedence(Prec,Zacatky).
```

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 jeden úkol)

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

lide(Zacatky,Doby,Lide) :-
        findall(clovek(Kdo,IdUkoly),clovek(Kdo,IdUkoly), Lide),
        omezeni_lide(Lide,Zacatky,Doby).

omezeni_lide([],_Zacatky,_Doby).
omezeni_lide([Clovek|Lide],Zacatky,Doby) :-
        Clovek=clovek(_Id,IdUkoly),
        omezeni_clovek(IdUkoly,Zacatky,Doby),
        omezeni_lide(Lide,Zacatky,Doby).
```


Plánování a lidé (pokračování)

```
omezeni_clovek(IdUkoly,Zacatky,Doby) :-
        omezeni_clovek(IdUkoly,Zacatky,Doby,[],[]).

% omezeni_clovek(IdUkoly,Zacatky,Doby,ClovekZ,ClovekD)
omezeni_clovek([],_Zacatky,_Doby,ClovekZ,ClovekD) :-
        serialized(ClovekZ,ClovekD).
omezeni_clovek([U|IdUkoly],Zacatky,Doby,ClovekZ,ClovekD) :-
        nth(U,Zacatky,Z),
        nth(U,Doby,D),
        omezeni_clovek(IdUkoly,Zacatky,Doby,[Z|ClovekZ],[D|ClovekD]).
```

Rozšiřte řešení problému tak, aby mohl každý člověk zpracovávat několik úkolů dle jeho zadané kapacity.

```
% clovek(Id,Kapacita,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(Zacatky,Doby,Lide) :-
        findall(clovek(Kdo,Kapacita,IdUkoly),clovek(Kdo,Kapacita,IdUkoly), Lide
        omezeni_lide(Lide,Zacatky,Doby).

omezeni_lide([],_Zacatky,_Doby).
omezeni_lide([clovek(_Id,Kapacita,IdUkoly)|Lide],Zacatky,Doby) :-
        omezeni_clovek(IdUkoly,Kapacita,Zacatky,Doby),
        omezeni_lide(Lide,Zacatky,Doby).

omezeni_clovek(IdUkoly,Kapacita,Zacatky,Doby) :-
        omezeni_clovek(IdUkoly,Kapacita,Zacatky,Doby,[],[]).

omezeni_clovek([],Kapacita,_Zacatky,_Doby,ClovekZ,ClovekD) :-
        length(ClovekZ,Delka), listOf1(Delka,ListOf1),
        cumulative(ClovekZ,ClovekD,ListOf1,Kapacita).
omezeni_clovek([U|IdUkoly],Kapacita,Zacatky,Doby,ClovekZ,ClovekD) :-
        nth(U,Zacatky,Z), nth(U,Doby,D),
        omezeni_clovek(IdUkoly,Kapacita,Zacatky,Doby,[Z|ClovekZ],[D|ClovekD]).

listOf1(0,[]) :- !.
listOf1(D,[1|L]) :- D1 is D-1, listOf1(D1,L).
```


Všechna řešení, stromy, grafy

Všechna řešení

```
% z(Jmeno,Prijmeni,Sex,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,z,30,ucitel,zs_stara).  z(petra,spickova,z,45,reditelka,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 ) ),
?- bagof( J-P, [P,S,V,Pr]^(z(J,P,S,V,Pr,F),( F=skoda ; F=cs ) ) ,  L ).
?- setof( P-J, [P,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 učitelů 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,_,_,_,_),z(J2,P,_,_,_,_), J1@<J2, J1\=J2 ),L).
6. bagof(P,[J,S,V,Pr]^z(J,P,S,V,Pr,F),L),length(L,Pocet),Pocet>1.
```


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,Tree)`

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(Tree,X,TreeWithX)`

Prvek X přidej do stromu T jednou z

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

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

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

```
add([],X,leaf(X)) :- !.
add(leaf(V), X, T1) :-
        ( X>V, !, T1=tree([],V,leaf(X))
        ; X<V, T1=tree(leaf(X),V,[])
        ).
add(tree(L,V,R), X, tree(L1,V,R1)) :-
        ( X>V, !, L1=L, add(R,X,R1)
        ; X<V, R1=R, add(L,X,L1)
        ).
```


Procházení stromů

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

Použit princip rozdílových seznamů

```
        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_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).
```


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],[1,5,6],[4],[1,4]).

?- functor(Graf,graf,PocetUzlu).
?- arg(Uzel,Graf,Sousedi).
```

- (Orientovany) ohodnocený graf

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


Procházení grafu do hloubky

- 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 (iniciální uzel označíme empty)
- Procházení grafu z uzlu U
  - Vytvoříme term pro rodiče (všichni rodiči jsou zatím volné proměnné)
  - Uzel U má prázdného rodiče a má sousedy S
  - Procházíme (rekurzivně) všechny sousedy v S
- Procházení sousedů S uzlu U
  - Uzel V je první soused
  - Nastavíme rodiče uzlu V na uzel U
  - Pokud jsme V ještě neprošli (nemá rodiče), tak rekurzivně procházej všechny jeho sousedy
  - Procházej zbývající sousedy uzlu U

DFS: algoritmus

```
dfs(U,G,P) :-
        functor(G,graf,Pocet),
        functor(P,rodice,Pocet),
        arg(U,G,Sousedi),
        arg(U,P,empty),
        prochazej_sousedy(Sousedi,U,G,P).

prochazej_sousedy([],_,_,_).
prochazej_sousedy([V|T],U,G,P) :-
        arg(V,P,Rodic),
        ( nonvar(Rodic), !
        ;
          Rodic = U,
          arg(V,G,SousediV),
          prochazej_sousedy(SousediV,V,G,P)
        ),
        prochazej_sousedy(T,U,G,P).
```