

# IB013 Logické programování

*(průsvitky ze cvičení)*

Hana Rudová  
jaro 2013

## 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

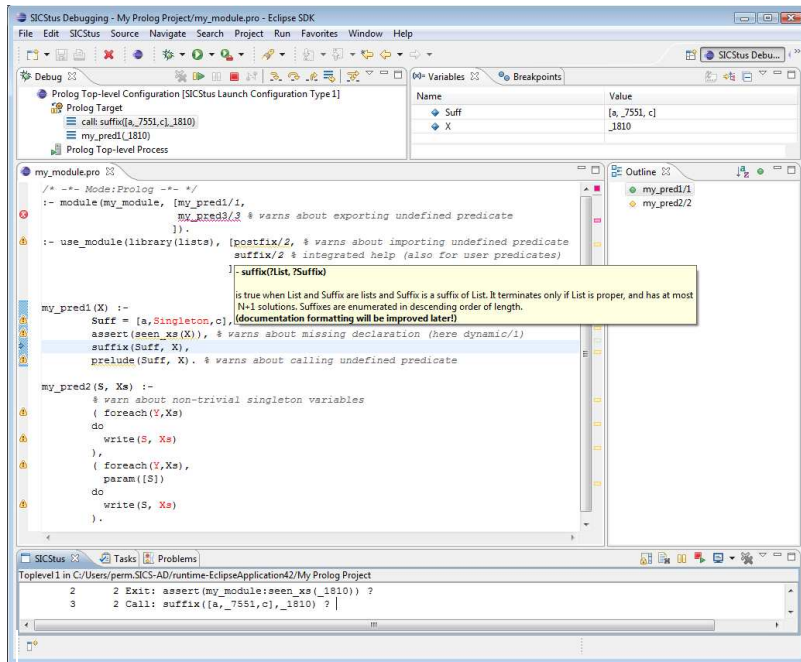
## Backtracking, unifikace, aritmetika

## 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  $\equiv$  databáze faktů a pravidel  $\equiv$  logické formule




## 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: spuštění programu

- **UNIX:**  
module add sicstus-4.1.3  
eclipse % používání IDE SPIDER  
sicstus % používání přes příkazový řádek
- **MS Windows:**
  - používání IDE SPIDER: C:\Eclipse3.7\eclipse.exe - Shortcut
  - příkazový řádek: z nabídky All Programs -> SICStus Prolog VC10 4.2.3  
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\SICStus Prolog VC10 4.2.3\bin\sicstus-4.2.3.exe"  
návod: <http://www.sics.se/sicstus/spider/site/prerequisites.html#SettingUp>

## SICStus Prolog: spuštění a přerušení výpočtu

- **Spuštění 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: 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 (pomocí rodic/2, muz/1, zena/1)

- prababicka(Prababicka,Pravnouce)
- nevlastni\_bratr(Nevlastni\_bratr,Nevlastni\_sourozenec)

nápověda: využijte  $X \backslash== Y$  (X a Y nejsou identické)  
Řešení:

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

## Backtracking: řešení příkladů II

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

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

```
| ?- 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?
X = filip,
Y = lenka ? ;
petr? petr? petr? petr? eva? jana? petr? eva? petr? petr? petr? jana? jana? pet
jana? pavel? pavel? pavel? pavel? adam? adam? adam? adam? pavla? pavla? adam?
pavla? tomas? tomas? tomas? tomas? michal? michal? michal? michal? eva? eva? pe
petr? eva? eva? petr? eva? jana? jana? petr? petr? jana? jana? petr? jana? pavl
pavla? adam? adam? pavla? pavla? adam? pavla? pavla? adam? pavla? pavla? pavla?
pavla? pavla? pavla? adam? pavla? pavla? pavla? pavla? lenka? lenka? lenka? ler
no
```

## Backtracking: porovnání

Nahrad'te 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` predikátem `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).
```

## Backtracking: řešení III

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

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

/\* varianta 1a \*/

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

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

/\* varianta 1b - jiné poradi odpovedi, neprimi potomci maji prednost \*/

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

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

/\* varianta 2a - leva rekurze ve druhe klauzuli,

na dotaz potomek(X, pavla) vypise odpovedi, pak cykli \*/

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

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

/\* varianta 2b - leva rekurze v prvni klauzuli,

na dotaz potomek(X, pavla) hned cykli \*/

## 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).

## 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 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)

$$\text{suma}(s(0), s(0), X0) = \text{suma}(s(X1), Y1, s(Z1))$$

$$\implies X1 = 0, Y1 = s(0), s(Z1) = X0$$

$$\implies \text{suma}(0, s(0), Z1)$$

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

$$\text{suma}(0, s(0), Z1) = \text{suma}(0, X2, X2)$$

$$X2 = s(0), Z1 = s(0)$$

$$\implies X0 = s(s(0))$$

$$X0 = s(s(0)) ;$$

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

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

D.Ú. Následující dotazy

$?- \text{suma}(X, s(0), Z)$ .                       $?- \text{suma}(s(0), X, Z)$ .

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

# 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 špatně uspěly?

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

Nápověda:  $'=:'$  unifikace,  $'=='$  test na identitu,  $'=:'$  aritmetická rovnost,  $'\setminus =='$  negace testu na identitu,  $'\setminus ='$  aritmetická nerovnost

# Aritmetika

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

Důležitý rozdíl ve vestavěných predikátech  $\text{is}/2$  vs.  $=/2$  vs.  $=:/2$

$\text{is}/2$ : < konstanta nebo proměnná >  $\text{is}$  < 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

$"=:"/2$   $"\setminus ="/2$   $">="/2$   $"<="/2$

< aritmetický výraz >  $=:$  < aritmetický výraz >

< aritmetický výraz >  $\setminus =$  < aritmetický výraz >

< aritmetický výraz >  $=$  < aritmetický výraz >

< aritmetický výraz >  $>=$  < aritmetický výraz >

levá i pravá strana jsou nejdříve aritmeticky vyhodnoceny a pak porovnány

# Aritmetika: příklady II

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

$s1(0, X, X)$ .

$s1(s(X), Y, s(Z)) :- s1(X, Y, Z)$ .

$s2(X, Y, Z) :- Z \text{ is } X + Y$ .

$s1/3$  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

# 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 odpovědi na dotazy  $\text{suma}(X,s(0),Z)$  a  $\text{suma}(s(0),X,Z)$ .

## Reprezentace seznamu

- **Seznam:**  $[a, b, c]$ , prázdný seznam  $[]$
- **Hlava (libovolný objekt), tělo (seznam):**  $.(H\text{lava}, T\text{elo})$ 
  - všechny strukturované objekty stromy - i seznamy
  - funktor ".", dva argumenty
  - $.(a, .(b, .(c, []))) = [a, b, c]$
  - notace:  $[ H\text{lava} \mid T\text{elo} ] = [a \mid T\text{elo}]$   
Telo je v  $[a \mid T\text{elo}]$  seznam, tedy píšeme  $[a, b, c] = [a \mid [b, c]]$
- Lze psát i:  $[a, b \mid T\text{elo}]$ 
  - před "|" je libovolný počet prvků seznamu, za "|" je seznam zbývajících prvků
  - $[a, b, c] = [a \mid [b, c]] = [a, b \mid [c]] = [a, b, c \mid []]$
  - pozor:  $[ [a, b] \mid [c] ] \neq [ a, b \mid [c] ]$
- Seznam jako **neúplná datová struktura:**  $[a, b, c \mid T]$ 
  - Seznam =  $[a, b, c \mid T]$ ,  $T = [d, e \mid S]$ , Seznam =  $[a, b, c, d, e \mid S]$

## Seznamy, řez

## 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
```

Předchudce a následník prvku X v seznamu S

```
h\edej(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 ) :- 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 paměťovými nároky
- Typický příklad, kdy je možné použití LCO:
  - procedura musí mít pouze jedno rekurzivní volání: **v posledním cíli poslední klauzule**
  - cíle předcházející tomuto rekurzivnímu volání musí být **deterministické**
  - `p( ... ) :- ...` % žádné rekurzivní volání v těle klauzule
  - `p( ... ) :- ...` % žádné rekurzivní volání v těle klauzule
  - ...
  - `p(... ) :- ..., !, p( ... ).` % řez zajišťuje determinismus
- Tento typ rekurze lze převést na iteraci

## LCO a akumulátor

- Reformulace rekurzivní procedury, aby umožnila LCO
- Výpočet délky seznamu `length( Seznam, Délka )`

```
length( [], 0 ).  
length( [ H | T ], Délka ) :- length( T, Délka0 ), Délka is 1 + Délka0.
```
- Upravená procedura, tak aby umožnila LCO:

```
% length( Seznam, ZapocitanaDélka, CelkovaDélka ):  
% CelkovaDélka = ZapocitanaDélka + ,,počet prvků v Seznam''  
  
length( Seznam, Délka ) :- length( Seznam, 0, Délka ). % pomocný predikát  
  
length( [], Délka, Délka ). % celková délka = započítaná délka  
length( [ H | T ], A, Délka ) :- A0 is A + 1, length( T, A0, Délka ).
```
- Přídavný argument se nazývá **akumulátor**

## Akumulátor a sum\_list(S, Sum)

```
?- sum_list( [2,3,4], Sum ).  
  
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 navrácí 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).	r1	?- X=1,r(X).
r(X):-p(X),write(r2).	X = 1 ? ;	?- X= -6, r(X)
r(X):-write(r3).	p1r2	r1
p(X):-write(p1).	X = 1 ? ;	X = -6 ? ;
p(X):-a(X),b(X),!,	a1b1r3	p1r2
c(X),d(X),write(p2).	X = 1 ? ;	X = -6 ? ;
p(X):-write(p3).	no	a1b2c1d1p2r2
a(X):-write(a1).	?- X=0,r(X).	d2p2r2
a(X):-write(a2).	r1	X = 3 ? ;
b(X):- X > 0, write(b1).	X = 0 ? ;	X = -6 ? ;
b(X):- X < 0, write(b2).	p1r2	X = -6 ? ;
c(X):- X mod 2 == 0, write(cc1).	X = 0 ? ;	d2p2r2
c(X):- X mod 3 == 0, write(cc2).	a1a2p3r2	X = 3 ? ;
d(X):- abs(X) < 10, write(d1).	X = 0 ? ;	no
d(X):- write(d2).	r3	X = -6 ? ;
	X = 0 ? ;	no
	no	

## Řez: maximum

Je tato definice predikátu max/3 korektní?

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

Není, následující dotaz uspeje: ?- 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.
```

Problém byl v definici, v první klauzuli se tvrdilo:  $X=Z \wedge X>=Y \Rightarrow true$   
správná definice je:  $X>=Y \Rightarrow Z=X$

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

**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.
```

## 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
```

## 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`

## 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.
```



## 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
Hana Rudová, Logické programování I, 15. května 2013 49 Vstup/výstup, databázové operace, rozklad termu
```

## 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(A,B) :- atomic(B), !, fail.                       % jen kvůli vyšší efektivitě
same([HA|TA], [HB|TB]) :- !, same(HA,HB), same(TA,TB).
same(A, B) :- A=.. [F|ArgA], B=.. [F|ArgB], same(ArgA,ArgB).
```

## 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
subterm(T,T) :- !.
subterm(S, [H|_]) :- subterm(S,H), !.
subterm(S, [_|T]) :- !, subterm(S,T).
subterm(S,T) :- compound(T), T=.. [_|Argumenty], subterm(S,Argumenty).
```

## 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
```

```
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(A,B) :- atomic(B), !, fail.
unify([HA|TA], [HB|TB]) :- !, unify(HA,HB), unify(TA,TB).
unify(A,B) :- A=.. [F|ArgA], B=.. [F|ArgB], 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 těle 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).

## 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,uklizicka,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 zaměstnanců 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, třídění, rozdílové seznamy

## 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?

```
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).
```

Hana Rudová, Logické programování I, 15. května 2013

57

Všechna řešení, třídění, rozdílové seznamy

## 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: split(X,Seznam,Small,Big)
- v Big jsou větší prvky než X a v Small jsou zbývající prvky
- 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, T, 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).
```

Hana Rudová, Logické programování I, 15. května 2013

59

Všechna řešení, třídění, rozdílové seznamy

## bubblesort(S, Sorted)

Seznam S seřad'te tak, že

- nalezněte první dva sousední prvky X a Y v S tak, že X>Y, swap(S,S1)
- vyměňte pořadí X a Y a získáte S1; rekurzivně bubblesortem
- a seřad'te S1
- 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).
```

Hana Rudová, Logické programování I, 15. května 2013

58

Všechna řešení, třídění, rozdílové seznamy

## 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.

```
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]).
```

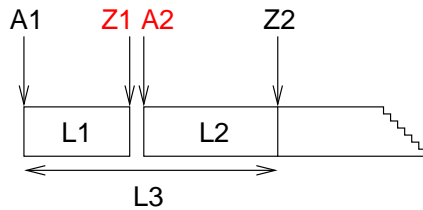
Hana Rudová, Logické programování I, 15. května 2013

60

Všechna řešení, třídění, rozdílové seznamy

## Rozdílové seznamy

- Zapamatování konce a připojení na konec: rozdílové seznamy
- $[a, b] \dots L1-L2 = [a, b|T]-T = [a, b, c|S]-[c|S] = [a, b, c]-[c]$
- Reprezentace prázdného seznamu: L-L



- $?- \text{append}([1,2,3|Z1]-Z1, [4,5|Z2]-Z2, A1-[]).$
  - $\text{append}(A1-Z1, Z1-Z2, A1-Z2).$
- L1      L2      L3
- $\text{append}([1,2,3,4,5]-[4,5], [4,5]-[], [1,2,3,4,5]-[]).$

## 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]

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

```
quicksort1([],Z-Z).
```

```
quicksort1([X|T], A1-Z2) :-
    split(X, T, Small, Big),
    quicksort1(Small, A1-[X|Z1]),
    quicksort1(Big, Z1-Z2).
```

```
append(A1-Z1, Z1-Z2, A1-Z2).
```

## 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).
```



## Logické programování s omezujícími podmínkami

## Algebrogram

- Přiřaďte cifry 0, ... 9 písmenům S, E, N, D, M, O, R, Y tak, aby platilo:

$$\begin{array}{r} \text{SEND} \\ + \text{MORE} \\ \hline \text{MONEY} \end{array}$$

- 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:**

$$\begin{array}{r} 1000*S + 100*E + 10*N + D \\ + 1000*M + 100*O + 10*R + E \\ \hline \neq 10000*M + 1000*O + 100*N + 10*E + Y \end{array} \quad \begin{array}{r} \text{SEND} \\ + \text{MORE} \\ \hline \text{MONEY} \end{array}$$

Hana Rudová, Logické programování I, 15. května 2013

66

Omezující podmínky

## 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)`

Hana Rudová, Logické programování I, 15. května 2013

67

Omezující podmínky

## Algebrogram: řešení

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

```
dona1d(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),
    10000*D + 10000*O + 1000*N + 100*A + 10*L + D +
    10000*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).
```

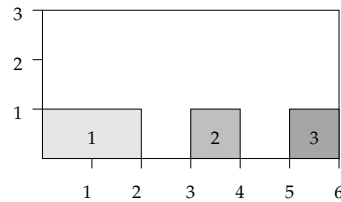
Hana Rudová, Logické programování I, 15. května 2013

68

Omezující podmínky

## 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: výstup

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

priprav([], [], []).
priprav([uko1(Id, Doba, MinStart, MaxKonec) | UkoLy], [Z | Zacatky],
        [uko1(Id, Doba, MinStart, MaxKonec, Z) | Vstup]) :-
    priprav(UkoLy, Zacatky, Vstup).

tiskni([]) :- nl.
tiskni([V | Vystup]) :-
    V = uko1(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

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

Úkoly jsou zadány následujícím způsobem:

```
% uko1(Id, Doba, MinStart, MaxKonec)
uko1(1, 4, 8, 70).    uko1(2, 2, 7, 60).    uko1(3, 1, 2, 25).    uko1(4, 6, 5, 55).
uko1(5, 4, 1, 45).    uko1(6, 2, 4, 35).    uko1(7, 8, 2, 25).    uko1(8, 5, 0, 20).
uko1(9, 1, 8, 40).    uko1(10, 7, 4, 50).    uko1(11, 5, 2, 50).    uko1(12, 2, 0, 35).
uko1(13, 3, 30, 60).    uko1(14, 5, 15, 70).    uko1(15, 4, 10, 40).
```

Kostra řešení:

```
ukoLy(Zacatky) :- domeny(UkoLy, Zacatky, Tasks),
                  cumulative(Tasks),
                  labeling([], Zacatky),
                  tiskni(UkoLy, Zacatky).

domeny(UkoLy, Zacatky, Tasks) :- findall(uko1(Id, Doba, MinStart, MaxKonec),
   uko1(Id, Doba, MinStart, MaxKonec), UkoLy),
                                  nastav_domeny(UkoLy, Zacatky, Tasks).
```

## 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(uko1(_,_,_,_,Z1), uko1(_,_,_,_,Z2)) :- Z1 > Z2.
```

## Plánování a domény

Napište predikát `nastav_domeny/3`, který na základě datové struktury `[uko1(Id,Doba,MinStart,MaxKonec)|Uko1y]` 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)`.

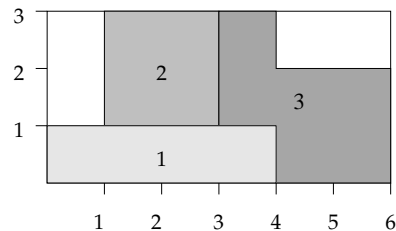
```
% nastav_domeny(+Uko1y,-Zacatky,-Tasks)

nastav_domeny([],[],[]).
nastav_domeny([uko1(Id,Doba,MinStart,MaxKonec)|Uko1y],[Z|Zacatky],
               [task(Z,Doba,K,1,Id)|Tasks]) :-
    MaxStart is MaxKonec-Doba,
    Z in MinStart..MaxStart,
    K #= Z + Doba,
    nastav_domeny(Uko1y,Zacatky,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)])
```



## D.Ú. Plánování a precedence: precedences(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).
```

## 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)

```
uko1y(Zacatky) :- % původně
                 domeny(Uko1y,Zacatky,Tasks),
                 cumulative(Tasks),
                 labeling([],Zacatky),
                 tiskni(Uko1y,Zacatky).
```

```
uko1y_lide(Zacatky) :- % upravená verze
                      domeny(Uko1y,Zacatky,Tasks),
                      lide(Tasks,Lide),
                      labeling([],Zacatky),
                      tiskni_lide(Lide,Uko1y,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)).
```

```
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]).
```

## Poděkování

Průsvivky 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