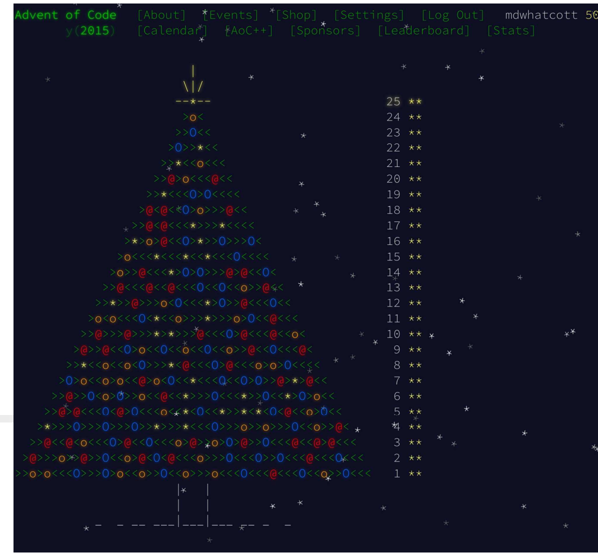
# Logické programovanie 2

Dnes bude:

- rekurzia na zoznamoch - append, reverse, flat, podzoznam, ...
- unifikácia a spôsob výpočtu LP
- čo je logická premenná ?
- nedeterministické programy (kombinatorika)
- prvý backtracking a algebrogramy
  - magické číslo, SEND+MORE=MONEY, 8-dám, ...
- Constraint Logic Programming - alternatívny pohľad
  - magické číslo, SEND+MORE=MONEY, 8-dám, ...

Cvičenie

- obľúbená kombinatorika, variácie, kombinácie, s a bez opakovania
- backtracking




# Rekapitulácia

Minule bolo:

- program je konečná množina (Hornových) klauzúl tvaru (implikácie):
  A.
  alebo
  $A\text{:-}B_1, B_2, \ldots, B_n.$
- klauzula predstavuje všeobecne kvantifikovanú implikáciu,
- dotaz (cieľ) je tvaru $?\text{-}B_1, B_2, \ldots, B_n$ (a obsahuje hľadané premenné),
- v programe deklarujeme tvrdenia o pravdivosti predikátov - čo sa z programu nedá odvodiť, neplatí bez toho, aby sme to deklarovali,
- premenné začínajú veľkým písmenom alebo _
- funkčné a predik.symboly začínajú malým písmenom,
- Prolog má zoznamy s konštruktormi [] a [H|T]
- Prolog je beztypový jazyk (pozná čísla, atómy-termy, zoznamy),
- klauzule sa skúšajú v textovom poradí (od hora dole),
- klauzule sa skúšajú všetky, ktoré môžu byť použiteľné, nie ako Haskell
- Prolog vráti všetky riešenia problému, ktoré nájde,
- Prolog nemusí nájsť riešenie, ak sa zacyklí

# Kvíz o zoznamoch

List Pattern

- Haskell (x:xs)
- Prolog [X|Xs]
- [X,Y|Tail]
  [X,Y,Z|[]]

- neprázdny zoznam
  neprazdnyZoznam([_|_]).
- aspoň dvojprvkový zoznam
  asponDvojPrvkovyZoznam([_,_|_]).
- tretí prvok
  tretiPrvok([_,_,T|_],T).
- posledný prvok zoznamu
  posledny([X],X).
  posledny([_,Y|Ys],X):-posledny([Y|Ys],X).
- tretí od konca
  tretiOdKonca(Xs,T) :- reverse(Xs,Ys),tretiPrvok(Ys,T).
- prostredný prvok zoznamu, ak existuje
  prostredny(Xs,T):-append(U,[T|V],Xs),length(U,L), length(V,L).

# Rekurzia vs. iterácia

Prolog nemá (ako Haskell):

- notáciu [1..n]
- list comprehension
- analógie foldr, foldl

- vygeneruj zoznam čísel od 1 po N pre zadané N

- prvý pokus (rekurzívne riešenie) – [N, ..., 1]

```
nAzJedna(0,[]).
nAzJedna(N,[N|X]):-N>0,N1 is N-1,nAzJedna(N1,X).
```

**?- nAzJedna(4,L).**
**L = [4, 3, 2, 1] ;**

- druhý pokus (iteratívne riešenie) – [1, ..., N]

```
jednaAzN(N,Res):-jednaAzN(N,[],Res).
jednaAzN(0,Acc,Res):-Acc=Res.
jednaAzN(N,Acc,Res):-N>0,N1 is N-1,jednaAzN(N1,[N|Acc],Res).
```

... alebo počítajme *dohora*

**?- jednaAzN(5,L).**
**L = [1, 2, 3, 4, 5] ;**

- tretí korektný pokus (nájdem v knižnici)

```
:- use_module(library(lists)).   % toto je import library(lists), default je in
```

https://www.swi-prolog.org/pldoc/man?section=lists

**?- numlist(1,7,List).**
**List = [1, 2, 3, 4, 5, 6, 7].**

# Snapshot do knižnice
## (library(list))

Módy:

- \+ musí byť známy
- ? nemusí byť známy

?- append(X,Y,[1,2,3]).
X = [],Y = [1, 2, 3] ;
X = [1],Y = [2, 3] ;
X = [1, 2],Y = [3] ;
X = [1, 2, 3],Y = [] ;
false.

?- append(X,[1,2,3]).
ERROR: Arguments are not sufficiently instantiated

**member**(*?Elem, ?List*)

True if *Elem* is a member of *List*. The SWI-Prolog definition differs from the classical one. Our definition avoids unpacking each list element twice and provides determinism on the last element. E.g. this is deterministic:

```
member(X, [One]).
```

**author**
Gertjan van Noord

**append**(*?List1, ?List2, ?List1AndList2*)

*List1AndList2* is the concatenation of *List1* and *List2*

**append**(*+ListOfLists, ?List*)

Concatenate a list of lists. Is true if *ListOfLists* is a list of lists, and *List* is the concatenation of these lists.

| | |
|---|---|
| ***ListOfLists*** | must be a list of *possibly* partial lists |

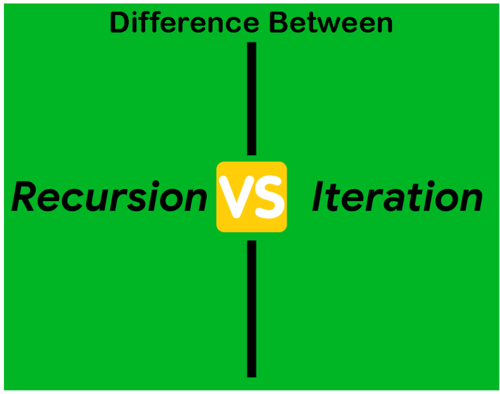
**prefix**(*?Part, ?Whole*)

True iff *Part* is a leading substring of *Whole*. This is the same as `append(Part, _, Whole)`.

**select**(*?Elem, ?List1, ?List2*)

Is true when *List1*, with *Elem* removed, results in *List2*. This implementation is determinsitic if the last element of *List1* has been selected.

# Konverzia zoznamu cifier na číslo




- konverzia zoznamu cifier na číslo

```
zoznamToInt([],0).
zoznamToInt([X|Xs],C) :- zoznamToInt(Xs,C1), C is 10*C1+X.
```

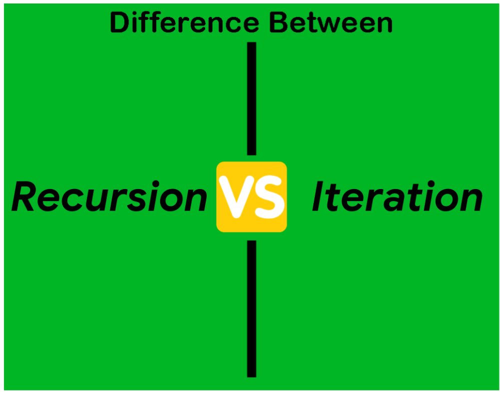
```
?- zoznamToInt([1,2,3,4],X).
X = 4321 ;
```

- konverzia čísla na zoznam cifier

```
intToZoznam(0,[]).
intToZoznam(C,[X|Xs]) :- C > 0,
                X is C mod 10, C1 is C // 10, intToZoznam(C1,Xs).
```

```
?- intToZoznam(4321,X).
X = [1, 2, 3, 4] ;
```

# Konverzia s akumulátorom




- akumulátorová verzia konverzie zoznamu cifier na číslo

```
zoznamToInt2(X,Res) :- zoznamToInt2(X,0,Res).
zoznamToInt2([],Acc,Res) :- Res = Acc.   % pomocný predikát/3
zoznamToInt2([X|Xs],Acc,Res) :- Acc1 is 10*Acc+X,
                                zoznamToInt2(Xs,Acc1,Res).
```

```
?- zoznamToInt2([1,2,3,4],X).
X = 1234 ;
```

- akumulátorová verzia konverzie čísla na zoznam cifier

```
intToZoznam2(X,Res) :- intToZoznam2(X,[],Res).
intToZoznam2(0,Acc,Acc).                 % pomocný predikát/3
intToZoznam2(C,Acc,Res) :- C > 0,
      X is C mod 10, C1 is C // 10, intToZoznam2(C1,[X|Acc],Res).
```

```
?- intToZoznam2(1234,X).
X = [1, 2, 3, 4] ;
```

# Zoznamová rekurzia

spojenie zoznamov, rekurzívna definícia predikátu append/3

- append([], Ys, Ys).
  append([X | Xs], Ys, [X | Zs]) :- append(Xs, Ys, Zs).

- ?- append([1,2],[a,b],[1,2,a,b]).

yes

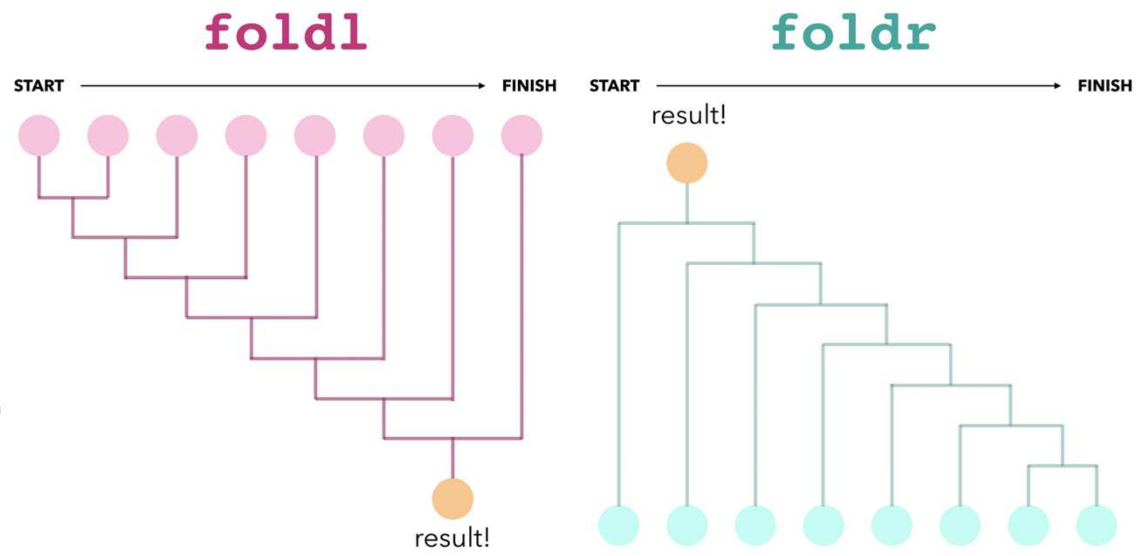
- ?- append([1,2,3],[4,5],V).

V = [1,2,3,4,5]

- ?- append(X, Y, [1,2,3]).

X = [], Y = [1,2,3] **;**
X = [1], Y = [2,3] **;**
X = [1,2], Y = [3] **;**
X = [1,2,3], Y = [] **;**
**no**

# reverse




- reverse rekurzívny
  reverse([], []).
  reverse([X|Xs], Y) :- reverse(Xs, Ys), append(Ys, [X], Y).

- akumulátorová verzia
  reverse(X, Y) :- reverse(X, [], Y).

  reverse([], Acc, Acc).
  reverse([X | Xs], Acc, Z) :- reverse(Xs, [X | Acc], Z).

```
reverse([1,2,3],Acc)
    reverse([1,2,3],[],Acc)
        reverse([2,3],[1],Acc)
            reverse([3],[2,1],Acc)
                reverse([],[3,2,1],Acc)
                    Acc = [3,2,1]
```

https://www.joachim-breitner.de/blog/753-Drawing_foldl_and_foldr

*unifikácia v Prologu je to,*
*čo volanie metódy v Jave*

# Unifikácia

unifikácia určuje, či klauzula je použiteľná na riešenie problému (cieľa)

Príklady (neformálny úvod):

- cieľ: C=append([1,2],[3,4],V), klauzula: H=append([X|Xs],Ys,[X|Zs]):-…
  riešenie: substitúcia θ = { X/1, Xs/[2], Ys/[3,4], V/[1|Zs] }
  lebo keď dosadíme Cθ=Hθ=append([1,2],[3,4],[1|Zs])

- cieľ: C=append([],[3,4],Zs2), klauzula: H=append([],Ys,Ys):-…
  riešenie: substitúcia θ = { Zs2/Ys, Ys/[3,4] }
  lebo keď dosadíme Cθ=Hθ=append([],[3,4],[3,4])

Unifikácia cieľa C a hlavy klauzule H je substitúcia θ taká, že Cθ= Hθ

- nemusí existovať (keď klauzula nie je použiteľná na redukciu cieľa):
  C=append([1,2], …   H=append([], …)
- ak existuje, zaujíma nás najvšeobecnejšia,
  napr. ak C=pred(X,Y) a H=pred(W,W):-…,
  - potom θ = { X/2, Y/2, W/2 } je príliš *konkrétna*
  - najvšeobecnejšia je θ = { X/W, Y/W }

# ?-append([1,2],[3,4],Zs).

```
append([],Ys,Ys)
append([X|Xs],Ys,[X|Zs])<-append(Xs,Ys,Zs)
```

$$\leftarrow \underline{append([1,2],[3,4],Zs)}$$

$$\downarrow \quad \theta_1 = \{X1/1, Xs1/[2], Ys1/[3,4], Zs/[1|Zs1],\}$$

$$\leftarrow \underline{append([2],[3,4],Zs1)}$$

$$\downarrow \quad \theta_2 = \{X2/2, Xs2/[], Ys2/[3,4], Zs1/[2|Zs2]\}$$

$$\leftarrow \underline{append([],[3,4],Zs2)}$$

$$\downarrow \quad \theta_3 = \{Ys3/[3,4], Zs2/[3,4]\}$$

$$\square$$

Výsledok: $append([1,2],[3,4],Zs)\theta_1\theta_2\theta_3 = append([1,2],[3,4],[1,2,3,4])$

# ?-append([1,2],Ys,Zs).

```
append([],Ys,Ys)
append([X|Xs],Ys,[X|Zs])<-append(Xs,Ys,Zs)
```

$$\leftarrow \underline{append([1,2],Ys,Zs)}$$

$$\theta_1 = \{X1/1, Xs1/[2], Ys/Ys1, Zs/[1|Zs1],\}$$

$$\leftarrow \underline{append([2],Ys1,Zs1)}$$

$$\theta_2 = \{X2/2, Xs2/[], Ys1/Ys2, Zs1/[2|Zs2]\}$$

$$\leftarrow \underline{append([],Ys2,Zs2)}$$

$$\theta_3 = \{Ys2/Zs2\}$$

$$\square$$

Výsledok: $append([1,2],Ys,Zs)\theta_1\theta_2\theta_3 = append([1,2],Zs2,[1,2|Zs2])$

```
append([], Ys, Ys).
append([X | Xs], Ys, [X | Zs]) :- append(Xs, Ys, Zs).
```

# ?-append(Xs,Ys,[1,2,3]).

$\leftarrow \underline{append(Xs, Ys, [1,2,3])}$

$\theta_1 = \{X1/1, Zs1/[2,3], Xs/[1|Xs1], Ys/Ys1\}$

$\theta_1 = \{Xs/[], Ys/[1,2,3], Ys1/[1,2,3]\}$

$\square$

$append([], [1,2,3], [1,2,3])$

$\leftarrow \underline{append(Xs1, Ys1, [2,3])}$

$\theta_2 = \{X2/2, Zs2/[3], Xs1/[2|Xs2], Ys1/Ys2\}$

$\theta_2 = \{Xs1/[], Ys1/[2,3], Ys2/[2,3]\}$

$\square$

$append([1], [2,3], [1,2,3])$

$\leftarrow \underline{append(Xs2, Ys2, [3])}$

$\theta_3 = \{X3/3, Zs3/[], Xs2/[3|Xs3], Ys2/Ys3\}$

$\theta_3 = \{Xs2/[], Ys2/[3], Ys3/[3]\}$

$\square$

$append([1,2], [3], [1,2,3])$

$\leftarrow \underline{append(Xs3, Ys3, [])}$

$\theta_4 = \{Xs3/[], Ys3/[], Ys4/[]\}$

$\square$

$append([1,2,3], [], [1,2,3])$

```
append([], Ys, Ys).
append([X | Xs], Ys, [X | Zs]) :- append(Xs, Ys, Zs).
```

# ?-append(Xs,[3,4],Zs).

$$\leftarrow \underline{append(Xs,[3,4],Zs)}$$

$$\theta_1 = \{Xs/[X1|Xs1], Ys1/[3,4], Zs/[X1|Zs1]\}$$

$$\theta_1 = \{Xs/[\,], Ys1/[3,4], Zs/[3,4]\}$$

$$\leftarrow \underline{append(Xs1,[3,4],Zs1)}$$

$$\square$$

$$append([\,],[3,4],[3,4])$$

$$\theta_2 = \{Xs1/[X2|Xs2], Ys2/[3,4], Zs1/[X2|Zs2]\}$$

$$\theta_2 = \{Xs1/[\,], Ys2/[3,4], Zs1/[3,4]\}$$

$$\leftarrow \underline{append(Xs2,[3,4],Zs2)}$$

$$\square$$

$$append([X1],[3,4],[X1,3,4])$$

$$\theta_3 = \{Xs2/[X3|Xs3], Ys3/[3,4], Zs2/[X3|Zs3]\}$$

$$\theta_3 = \{Xs2/[\,], Ys3/[3,4], Zs2/[3,4]\}$$

$$\leftarrow \underline{append(Xs3,[3,4],Zs3)}$$

$$\square$$

$$append([X1,X2],[3,4],[X1,X2,3,4])$$

nekonečný výpočet

# Spy points, debug

- Špiónovať môžete vlastú definíciu, nie štandardný predikát

SWI-Prolog -- d:/borovan/PARA/PrednaskyPARA/Kod/PR11/prolog2.pl

File Edit Settings Run Debug Help

```
[debug]  ?- spy(reverse/3).
% Spy point on reverse/3
true.

[debug]  ?- reverse([1,2,3],Xs).
 * Call: (8) reverse([1, 2, 3], _4632) ? creep
 * Call: (9) reverse([2, 3], _4872) ? creep
 * Call: (10) reverse([3], _4872) ? creep
 * Call: (11) reverse([], _4872) ? creep
 * Exit: (11) reverse([], []) ? creep
   Call: (11) lists:append([], [3], _4880) ? creep
   Exit: (11) lists:append([], [3], [3]) ? creep
 * Exit: (10) reverse([3], [3]) ? creep
   Call: (10) lists:append([3], [2], _4886) ? creep
   Exit: (10) lists:append([3], [2], [3, 2]) ? creep
 * Exit: (9) reverse([2, 3], [3, 2]) ? creep
   Call: (9) lists:append([3, 2], [1], _4632) ? creep
   Exit: (9) lists:append([3, 2], [1], [3, 2, 1]) ? creep
 * Exit: (8) reverse([1, 2, 3], [3, 2, 1]) ? creep
Xs = [3, 2, 1].

[trace]  ?-
```

# Vlastné kontrolné výpisy

```
areverse(X, Y) :- reverse(X, [], Y).

% -- najjednoduchší spôsob pre debug
reverse(Xs, Ys, _) :- write('Xs='), write(Xs), write(', Ys='),
write(Ys), nl, fail.

reverse([], Acc, Acc).
reverse([X | Xs], Acc, Z) :- reverse(Xs, [X | Acc], Z).
```

```
?- areverse([1,2,3],Xs).
Xs=[1,2,3], Ys=[]
Xs=[2,3], Ys=[1]
Xs=[3], Ys=[2,1]
Xs=[], Ys=[3,2,1]
Xs = [3, 2, 1].
```

Prolog je netypovaný jazyk
je možné urobiť zoznam čohokoľvek,
teda aj zoznamov

# flat alias sploští

```
notAList(X):-atomic(X),X \= [].
```

Sploštenie heterogénneho zoznamu s viacerými úrovňami do jedného zoznamu všetkých prvkov

- naivné riešenie

```
flat([X|Xs],Ys):-flat(X,Ys1),flat(Xs,Ys2), append(Ys1,Ys2,Ys).
flat(X,[X]):-atomic(X),X \= [].        % notAList(X)
flat([],[]).
```

```
?- flat([1,[2,[],[3,[4]]]], X).
X = [4, 3, 2, 1] ;
```

- akumulátorové riešenie (odstraňujeme append):

```
flat1(X,Y):-flat(X,[],Y).
flat1([X|Xs],Ys1,Ys):-flat1(X,Ys1,Ys2),flat1(Xs,Ys2,Ys).
flat1(X,Ys,[X|Ys]):-atomic(X),X \= []. % notAList(X)
flat1([],Ys,Ys).
```

```
?- flat1([1,[2,[],[3,[4]]]], X).
X = [4, 3, 2, 1] ;
```

# Prefix a sufix zoznamu

**prefix**(*?Part, ?Whole*)

True iff *Part* is a leading substring of *Whole*. This is the same as `append(Part, _, Whole)`.

- začiatok zoznamu, napr. ?-prefix([1,a,3],[1,a,3,4,5])

```
prefix([], _).
prefix([X|Xs], [Y|Ys]) :- X = Y, prefix(Xs, Ys).
prefix([X|Xs], [X|Ys]) :- prefix(Xs, Ys).
```

- koniec (chvost) zoznamu ?-sufix([3,4,5],[1,2,3,4,5])

```
sufix(Xs,Xs).
sufix(Xs,[_|Ys]):-sufix(Xs,Ys).
```

- koniec zoznamu, ak už poznáme reverse

```
sufix(Xs,Ys):-reverse(Xs,Xs1), reverse(Ys,Ys1), prefix(Xs1,Ys1).
```

# Podzoznam zoznamu

- súvislý podzoznam, napr. ?-sublist([3,4,5],[1,2,3,4,5,6])

```
sublist1(X,Y) :- append(_,X,V),append(V,_,Y).
sublist2(X,Y) :- append(V,_,Y),append(_,X,V).
```

```
?-
sublist2(X,[1,2,3]).
X = [] ;
X = [1] ;
X = [] ;
X = [1, 2] ;
X = [2] ;
X = [] ;
X = [1, 2, 3] ;
X = [2, 3] ;
X = [3] ;
X = [] ;
false.
```

- ešte raz súvislý podzoznam, keď poznáme sufix, prefix, ...

```
sublist3(Xs,Ys):-prefix(W,Ys),sufix(Xs,W).
sublist4(Xs,Ys):-sufix(W,Ys),prefix(Xs,W).
```

- nesúvislý podzoznam, tzv. vybratá podpostupnosť

```
subseq([X|Xs],[X|Ys]):-subseq(Xs,Ys).
subseq([X|Xs],[_|Ys]) :- subseq([X|Xs],Ys).
subseq([],_).
```

```
?- subseq(X,[1,2,3]).
X = [1, 2, 3] ;
X = [1, 2] ;
X = [1, 3] ;
X = [1] ;
X = [2, 3] ;
X = [2] ;
X = [3] ;
X = [].
```

# Práca so zoznamom

- definujte predikát nth1(I,Xs,X), ktorý platí, ak Xs[I] = X

```
nth1(1,[X|_],X).
nth1(I,[_|Ys],X):-nth1(I1,Ys,X), I is I1+1.
```

**nth1**(*?Index, ?List, ?Elem*)

Is true when *Elem* is the *Index*'th element of *List*. Counting starts at 1.

```
?- nth1(I,[a,b,c],b).
I = 2 ;
```

```
?- nth1(I,[a,b,c],X).
X = a, I = 1 ;
X = b, I = 2 ;
X = c, I = 3 ;
```

```
?- nth1(I,[1,2,3,4],Elem,Rest).
I = Elem, Elem = 1, Rest = [2, 3, 4] ;
I = Elem, Elem = 2, Rest = [1, 3, 4] ;
I = Elem, Elem = 3, Rest = [1, 2, 4] ;
I = Elem, Elem = 4, Rest = [1, 2, 3] ;
```

# Práca so zoznamom

- definujte predikát **select**(X,Y,Z), ktorý vyberie všetky možné prvky X zo zoznamu Y, a výsledkom je zoznam Z

```
select(X,[X|Xs],Xs).
select(X,[Y|Ys],[Y|Zs]):-select(X,Ys,Zs).
```

```
?- select(X,[a,b,c],Z).
X = a, Z = [b, c] ;
X = b, Z = [a, c] ;
X = c, Z = [a, b] ;
```

**select**(*?Elem, ?List1, ?List2*)

Is true when *List1*, with *Elem* removed, results in *List2*. This implementation is determinsitic if the last element of *List1* has been selected.

- definujte predikát **delete**(X,Y,Z) ak Z je Y-[X]

```
delete(X,Y,Z):-select(X,Y,Z).
```

- definujte predikát **insert**(X,Y,Z), ktorý vsunie prvok X do zoznamu Y (na všetky možné pozície), výsledkom je zoznam Z

```
insert(X,Y,Z):-select(X,Z,Y).
```

# Permutácie

- definujte predikát perm(X,Y), ktorý platí, ak zoznam Y je permutáciou zoznamu X
  perm(Xs,[H|Hs]):-select(H,Xs,W),perm(W,Hs).
  perm([],[]).

```
?- perm([1,2,3,4],Xs).
Xs = [1, 2, 3, 4] ;
Xs = [1, 2, 4, 3] ;
Xs = [1, 3, 2, 4] ;
Xs = [1, 3, 4, 2]
```

- iná verzia, miesto select/delete robíme insert
  perm2([],[]).
  perm2([X|Xs],Zs):-perm2(Xs,Ys),insert(X,Ys,Zs).

**permutation**(*?Xs, ?Ys*) *[nondet]*

True when *Xs* is a permutation of *Ys*. This can solve for *Ys* given *Xs* or *Xs* given *Ys*, or even enumerate *Xs* and *Ys* together. The predicate permutation/2 is primarily intended to generate permutations. Note that a list of length N has N! permutations, and unbounded permutation generation becomes prohibitively expensive, even for rather short lists (10! = 3,628,800).

https://www.swi-prolog.org/pldoc/doc/_SWI_/library/lists.pl?show=src#permutation/2

```
perm([], []).
perm(List, [First|Perm]) :-
    select(First, List, Rest),
    perm(Rest, Perm).
```

# Kombinácie

- definujte predikát comb(X,Y), ktorý platí, ak zoznam X je kombináciou prvkov zoznamu Y
- ```
  comb([],_).
  comb([X|Xs],[X|T]):-comb(Xs,T).
  comb([X|Xs],[_|T]):-comb([X|Xs],T).
  comb(Xs,[_|T]):-comb(Xs,T).
  ```
- to bolo nedávno ako subseq ☺
- definujte predikát comb(K,X,Y), ktorý platí, ak zoznam X je K-prvkovou kombináciou prvkov zoznamu Y
- ```
  comb(0,[],_).
  comb(K,[X|Xs],[X|T]):-K1 is K-1, comb(K1,Xs,T).
  comb(K,[X|Xs],[_|T]):-comb(K,[X|Xs],T).
  ```

```
?- L=[_,_,_,_],
   comb(L,[1,2,3,4,5,6]).

L = [1, 2, 3, 4] ;
L = [1, 2, 3, 5] ;
L = [1, 2, 3, 6] ;
L = [1, 2, 4, 5] ;
L = [1, 2, 4, 6] ;
L = [1, 2, 5, 6] ;
L = [1, 3, 4, 5] ;
L = [1, 3, 4, 6] ;
L = [1, 3, 5, 6] ;
L = [1, 4, 5, 6] ;
L = [2, 3, 4, 5] ;
L = [2, 3, 4, 6] ;
L = [2, 3, 5, 6] ;
L = [2, 4, 5, 6] ;
L = [3, 4, 5, 6] ;
```

```
?-
comb(2,L,[1,2,3,4,5,6]).
L = [1, 2] ;
L = [1, 3] ;
L = [1, 4] ;
L = [1, 5] ;
L = [1, 6] ;
L = [2, 3] ;
L = [2, 4] ;
L = [2, 5] ;
L = [2, 6] ;
L = [3, 4] ;
L = [3, 5] ;
L = [3, 6] ;
L = [4, 5] ;
L = [4, 6] ;
L = [5, 6] ;
```

# Backtracking

štandardný predikát between(X,Y,Z) platí, ak Z je z intervalu [X;Y]

numlist(X,Y,L),member(Z,L).

- definujme predikát myBetween(Od,Do,X), ktorý platí, ak X je z intervalu [Od;Do] pričom vygeneruje všetky čísla X z tohoto intervalu

```
myBetween(Od,Do,Od):-Od =< Do.
myBetween(Od,Do,X):-Od<Do,Od1 is Od+1, myBetween(Od1,Do,X).
```

**?- myBetween(1,10,X).**
**X = 1;2;3;4;5;6;7;8;9;10;**
**No**

**between**(*+Low, +High, ?Value*)

*Low* and *High* are integers, *High* >=*Low*. If *Value* is an integer, *Low* =<*Value* =<*High*. When *Value* is a variable it is successively bound to all integers between *Low* and *High*. If *High* is `inf` or `infinite`[112] between/3 is true iff *Value* >=*Low*, a feature that is particularly interesting for generating integers from a certain value.

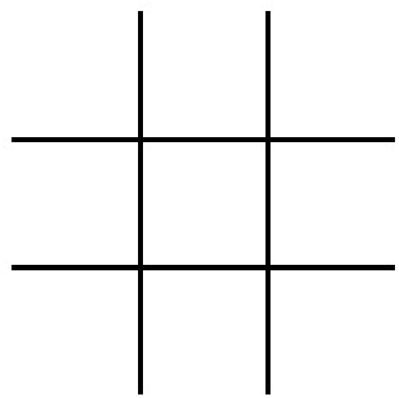
- použitie between – SQRT je celá časť z odmocniny z N

```
mySqrt(N,SQRT) :- between(1,N,SQRT),
                  SQRT2 is SQRT*SQRT, SQRT2 =< N,
                  SQRT1_2 is (SQRT+1)*(SQRT+1), SQRT1_2 > N.
```

**?- mySqrt(17,X).**
**X = 4 ;**

# Bactracking
## (ľahký úvod)

- vložte 6 kameňov do mriežky 3x3, tak aby v žiadnom smere (riadok, stĺpec, uhlopriečka) neboli tri.



- pri najivnom prehľadávaní - všetkých možností je 2^9 = 512
- ak poznáme kombinácie bez opakovania - možností je už len 9 nad 6, teda 9 nad 3, čo je 84

# Haskell to Prolog

- v Haskelli sme mali:

```
isOk:: [Int] -> Bool
isOk xs =  not (subset' [0,1,2] xs) && not (subset' [3,4,5] xs) && not (subset' [6,7,8] xs) &&
           not (subset' [0,3,6] xs) && not (subset' [1,4,7] xs) && not (subset' [2,5,8] xs) &&
           not (subset' [0,4,8] xs) && not (subset' [2,4,6] xs)
```

- v Prologu nič ľahšie:

```
isOk(Xs):- not(subseq([0,1,2],Xs)), not(subseq([3,4,5], Xs)), not(subseq([6,7,8], Xs)),
           not(subseq([0,3,6], Xs)), not(subseq([1,4,7], Xs)), not(subseq([2,5,8], Xs)),
           not(subseq([0,4,8], Xs)), not(subseq([2,4,6], Xs)).
% - daj mi všetky 6 prvkové podmnožiny 0..8, že sú okay.
?- Cs=[_,_,_,_,_,_],comb(Cs,[0,1,2,3,4,5,6,7,8]), isOk(Cs).
?- Cs=[I1,I2,I3,I4,I5,I6],comb(Cs,[0,1,2,3,4,5,6,7,8]), isOk(Cs).
Cs = [0, 1, 3, 5, 7, 8];
Cs = [1, 2, 3, 5, 6, 7];
```

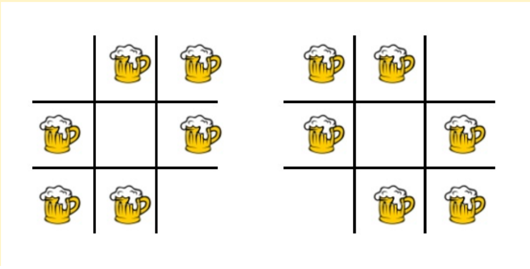
# Prolog to eCLIPse

(constraint logic programming)

```
isOk(Xs):- Xs[1]+Xs[2]+Xs[3] #<3,  Xs[4]+Xs[5]+Xs[6] #<3, Xs[7]+Xs[8]+Xs[9] #< 3,
           Xs[1]+Xs[4]+Xs[7] #<3,  Xs[2]+Xs[5]+Xs[8] #<3, Xs[3]+Xs[6]+Xs[9] #<3,
           Xs[1]+Xs[5]+Xs[9] #<3,  Xs[3]+Xs[5]+Xs[7] #<3.

isOk2(Xs):- (for(I,0,2), param(Xs) do Xs[1+3*I]+Xs[2+3*I]+Xs[3+3*I] #<3 ),
            (for(I,0,2), param(Xs) do Xs[1+I]+Xs[4+I]+Xs[7+I] #<3 ),
            Xs[1]+Xs[5]+Xs[9] #<3, Xs[3]+Xs[5]+Xs[7] #<3.

threeXthree(Cs) :-
    dim(Cs,[9]),
    Cs::0..1,
    % 6 #= Cs[1] + Cs[2] + Cs[3] + Cs[4] + Cs[5] + Cs[6] + Cs[7] + Cs[8] + Cs[9],
    6 #= sum(Cs[1..9]),
    isOk2(Cs),
    labeling(Cs),
    writeln(Cs).
```



```
?- threeXthree(Cs), fail.
[](0, 1, 1, 1, 0, 1, 1, 1, 0)
[](1, 1, 0, 1, 0, 1, 0, 1, 1)
```

```
SEND
+  MORE
======
 MONEY
```

# Send More Money

(algebrogram)

```
s(S,E,N,D,M,O,R,Y):
    cifra0(D),
    cifra0(E),D\=E,
    Y is (D+E) mod 10,
    Y\=E,Y\=D,
    Pr1 is (D+E) // 10,
    cifra0(N),N\=D,N\=E,N\=Y,
    cifra0(R),R\=D,R\=E,R\=Y,R\=N,
    E is (N+R+Pr1) mod 10,
    Pr2 is (N+R+Pr1) // 10,
    cifra0(O), O\=D,O\=E,O\=Y,O\=N,O\=R,
    N is (E+O+Pr2) mod 10,
    Pr3 is (E+O+Pr2) // 10,
    cifra0(S), S\=D,S\=E,S\=Y,S\=N,S\=R,S\=O,
    cifra0(M), M\=0,M\=D,M\=E,M\=Y,M\=N,M\=R,M\=O,M\=S,
    O is (S+M+Pr3) mod 10,
    M is (S+M+Pr3) // 10,
write(' '),write(S),write(E),write(N),write(D), nl,
write('+'),write(M),write(O),write(R),write(E),
nl,
write(M),write(O),write(N),write(E),write(Y),nl.
cifra0(0).
cifra0(X):-cifra(X).
```

```
s(S,E,N,D,M,O,R,Y).
 9567
+1085
10652

S = 9
E = 5
N = 6
D = 7
M = 1
O = 0
R = 8
Y = 2
```

# Send More Money

(constraint logic programming)

http://eclipseclp.org/

```
:- lib(ic).

sendmore(Digits) :-
    Digits = [S,E,N,D,M,O,R,Y],
    Digits :: [0..9],            % obor hodnôt
    alldifferent(Digits),        % všetky prvky zoznamu musia byť rôzne
    S #\= 0,  M #\= 0,           % úvodne cifry nemôžu byť 0
    (1000*S + 100*E + 10*N + D) + (1000*M + 100*O + 10*R + E)
          #= 10000*M + 1000*O + 100*N + 10*E + Y,
    labeling(Digits),                     % generovanie možností
    writeSolution(Digits).                % výpis riešenia

writeSolution([S,E,N,D,M,O,R,Y]) :-
    write(' '),write(S),write(E),write(N),write(D), nl,
    write('+'),write(M),write(O),write(R),write(E), nl,
    write(M),  write(O),write(N),write(E),write(Y),nl.
```

# Magické

- 381 je magické, lebo
  3 je delitel'né 1,
  38 je delitel'né 2,
  381 je delitel'né 3.

- magicke(X):-magicke(X,0,0).

- ```
  magicke([],_,_).
  magicke([X|Xs],Cislo,N) :-  Cislo1 is 10*Cislo+X,
                              N1 is N+1,
                              0 is Cislo1 mod N1,
                              magicke(Xs,Cislo1,N1).
  ```

- uplneMagicke(X) :- magicke(X), member(1,X), member(2,X), ...
  ```

ifthenelse(C,T,E):-C->T;E
ifthen(C,T):-C->T

# Ako nájdeme úplne magické

cifra(1).cifra(2).cifra(3).cifra(4).cifra(5).cifra(6).cifra(7).cifra(8).cifra(9).

- *technika "generuj a testuj"*

```
umag(X) :- cifra(C1), cifra(C2), cifra(C3), cifra(C4), cifra(C5),
           cifra(C6), cifra(C7), cifra(C8), cifra(C9),
           uplneMagicke([C1,C2,C3,C4,C5,C6,C7,C8,C9]),
           zoznamToInt2([C1,C2,C3,C4,C5,C6,C7,C8,C9],X).
```

- *technika backtracking*

if

then

```
umagiccifra(X):- length(X,9) -> zoznamToInt2(X,Y),write(Y),nl
           ;
           cifra(C),not(member(C,X)),
           append(X,[C],Y),
           magicke(Y),
           umagiccifra(Y).
```

else

?- umagic9([]).
381654729

# Magické
(constraint logic programming)

http://eclipseclp.org/

```
:- lib(ic).

uplneMagicke(Digits) :-
    Digits = [_,_,_,_,_,_,_,_,_],
    Digits :: [1..9],                  % obor hodnot
    alldifferent(Digits),              % vsetky prvky zoznamu musia byt rozne
    magicke(Digits),
    labeling(Digits).                  % generovanie moznosti

magicke(X):-magicke(X,0,0).
magicke([],_,_).
magicke([X|Xs],Cislo,N) :-
     Cislo1 #= 10*Cislo+X,
     N1 is N+1,
     Cislo1 / N1 #= _,
     magicke(Xs,Cislo1,N1).
```

# Master Mind

(hra Logic)

Hra MasterMind sa hráva vo viacerých verziách. Najjednoduchšia je taká, že hádate 4-ciferné číslo pozostávajúce z neopakujúcich sa cifier od 1 do 6. Napríklad, ak hádate utajené číslo 4251, hádajúci položí dotaz 1234, tak dostane odpoveď, koľko cifier ste uhádli (t.j. 3, lebo 1,2,4), a koľko je na svojom mieste (t.j. 1, lebo 2 je na "svojom" mieste v dotaze). Odpoveď je teda 3:1.

Definujte predikát mm(Utajene,Dotaz,X,Y), ktorý pre známe Utajene a Dotaz v tvare zoznamov [4,2,5,1] a [1,2,3,4] určí odpoveď X:Y, t.j, X=3 a Y=1.

Z rozohranej partie MasterMind ostal len zoznam dotazov a odpovedí hádajúceho vo formáte zoznamu, napr. P=[dotaz([1,2,3,4],3,1),dotaz([4,3,2,1],3,2)]. Definujte predikát findMM(P,X), ktorý pre zadaný zoznam dotazov P nájde všetky možné utajené čísla X, ktoré vyhovujú odpovediam na tieto dotazy.

Napr. X = [4,2,5,1] ale aj ďalšie.

**Hádané číslo** **[2,3,6,4]**
[1, 2,3,4] 3:1,
[3, 2,1,5] 2:0
[6, 4,3,1] 3:0

# Master Mind 1

MasterMind … koľko cifier ste uhádli, a koľko je na svojom mieste

- predikát spočíta počet *právd* v zozname:

```
countTrue([],0).
countTrue([C|Cs],N) :- countTrue(Cs,N1),
                       (C -> N is N1+1        % ak pravda,+1
                        ;
                        N is N1).             % inak nič
```

- 

```
mm([C1,C2,C3,C4],[Q1,Q2,Q3,Q4],X,Y) :-
      C = [C1,C2,C3,C4],
      countTrue([member(Q1,C),member(Q2,C),
                 member(Q3,C),member(Q4,C)],X),
      countTrue([C1=Q1,C2=Q2,C3=Q3,C4=Q4],Y).
```

Master Mind ešte príde...

# Master Mind 2

definujte predikát findMM(P,X), ktorý pre zadaný zoznam dotazov P nájde všetky možné utajené čísla X, ktoré vyhovujú odpovediam.

```
findMM(Qs, Code) :-                  % Qs-zoznam dotazov s odpoveďami
    Code1 = [_,_,_,_],               % hádaš štvorciferné číslo
    comb(Code1,[1,2,3,4,5,6]),       % ... 4-kombinácie množiny {1..6}
    perm(Code1,Code),                % ... a to rôzne poprehadzované
    checkMM(Qs,Code).                % ... že všetky dotazy platia

checkMM([],_).
checkMM([dotaz(Q,X,Y)|Qs],Code) :-
    mm(Q,Code,X,Y), checkMM(Qs,Code).
```

```
?-findMM([
    dotaz([1,2,3,4],3,1),
    dotaz([3,2,1,5],2,0),
    dotaz([6,4,3,1],3,0)],C).

C = [1, 6, 4, 2] ;
C = [2, 1, 6, 4] ;
C = [2, 3, 6, 4] ;
No
```

# Master Mind

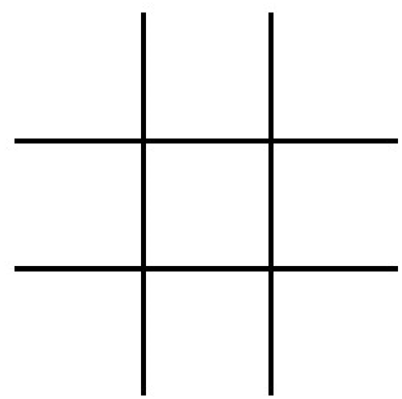
(constraint logic programming)

http://eclipseclp.org/

```
findMM(Qs, Code) :-
    Code = [_,_,_,_],
    Code :: [1..6],
    alldifferent(Code),
    labeling(Code),
    checkMM(Qs,Code).
```

# Bactracking
## (ľahký úvod)

- vložte 6 kameňov do mriežky 3x3, tak aby v žiadnom smere (riadok, stĺpec, uhlopriečka) neboli tri.



- pri naivnom prehľadávaní - všetkých možností je 2^9 = 512
- ak poznáme kombinácie bez opakovania - možností je už len 9 nad 6, teda 9 nad 3, čo je 84

# Haskell to Prolog

- v Haskelli sme mali:

```
isOk:: [Int] -> Bool
isOk xs =   not (subset' [0,1,2] xs) && not (subset' [3,4,5] xs) && not (subset' [6,7,8] xs) &&
            not (subset' [0,3,6] xs) && not (subset' [1,4,7] xs) && not (subset' [2,5,8] xs) &&
            not (subset' [0,4,8] xs) && not (subset' [2,4,6] xs)
```

- v Prologu nič ľahšie:

```
isOk(Xs):- not(subseq([0,1,2],Xs)), not(subseq([3,4,5], Xs)), not(subseq([6,7,8], Xs)),
           not(subseq([0,3,6], Xs)), not(subseq([1,4,7], Xs)), not(subseq([2,5,8], Xs)),
           not(subseq([0,4,8], Xs)), not(subseq([2,4,6], Xs)).
% - daj mi všetky 6 prvkové podmnožiny 0..8, že sú okay.
?- Cs=[_,_,_,_,_,_], comb(Cs,[0,1,2,3,4,5,6,7,8]), isOk(Cs).
?- Cs=[I1,I2,I3,I4,I5,I6],comb(Cs,[0,1,2,3,4,5,6,7,8]), isOk(Cs).
Cs = [0, 1, 3, 5, 7, 8];
Cs = [1, 2, 3, 5, 6, 7];
```

# Prolog to eCLIPse

(constraint logic programming)

#

nie je hashtag ale **constraint**
obmedzenie, podmienka, vzťah

```
isOk(Xs):- Xs[1]+Xs[2]+Xs[3] #<3,  Xs[4]+Xs[5]+Xs[6] #<3, Xs[7]+Xs[8]+Xs[9] #< 3,
           Xs[1]+Xs[4]+Xs[7] #<3,  Xs[2]+Xs[5]+Xs[8] #<3, Xs[3]+Xs[6]+Xs[9] #<3,
           Xs[1]+Xs[5]+Xs[9] #<3,  Xs[3]+Xs[5]+Xs[7] #<3.

           % pokus o cyklus v logickej paradigme vyzerá celkom tragicky...
isOk2(Xs):- (for(I,0,2), param(Xs) do Xs[1+3*I]+Xs[2+3*I]+Xs[3+3*I] #<3 ),
            (for(I,0,2), param(Xs) do Xs[1+I]+Xs[4+I]+Xs[7+I] #<3 ),
            Xs[1]+Xs[5]+Xs[9] #<3, Xs[3]+Xs[5]+Xs[7] #<3.

threeXthree(Cs) :-
    dim(Cs,[9]),
    Cs::0..1,
    % 6 #= Cs[1] + Cs[2] + Cs[3] + Cs[4] + Cs[5] + Cs[6] + Cs[7] + Cs[8] + Cs[9],
    6 #= sum(Cs[1..9]),
    isOk2(Cs),
    labeling(Cs), % backtrack
    writeln(Cs).
```

```
?- threeXthree(Cs), fail.
[](0, 1, 1, 1, 0, 1, 1, 1, 0)
[](1, 1, 0, 1, 0, 1, 0, 1, 1)
```

# Send More Money

(algebrogram)

```
 SEND
+ MORE
======
 MONEY
```

```
cifra(1). cifra(2). cifra(3). cifra(4).cifra(5).
cifra(6). cifra(7). cifra(8). cifra(9).
cifra(X):-between(1,9,X).
```

```
s(S,E,N,D,M,O,R,Y):-
    cifra0(D),
    cifra0(E),D\=E,
    Y is (D+E) mod 10,
    Y\=E,Y\=D,
    Pr1 is (D+E) // 10,
    cifra0(N),N\=D,N\=E,N\=Y,
    cifra0(R),R\=D,R\=E,R\=Y,R\=N,
    E is (N+R+Pr1) mod 10,
    Pr2 is (N+R+Pr1) // 10,
    cifra0(O), O\=D,O\=E,O\=Y,O\=N,O\=R,
    N is (E+O+Pr2) mod 10,
    Pr3 is (E+O+Pr2) // 10,
    cifra0(S), S\=D,S\=E,S\=Y,S\=N,S\=R,S\=O,
    cifra0(M), M\=0,M\=D,M\=E,M\=Y,M\=N,M\=R,M\=O,M\=S,
    O is (S+M+Pr3) mod 10,
    M is (S+M+Pr3) // 10,
    write(' '),write(S),write(E),write(N),write(D),nl,
    write('+'),write(M),write(O),write(R),write(E),
    nl,
    write(M),write(O),write(N),write(E),write(Y),nl.
cifra0(0).
cifra0(X):-cifra(X).
```

```
?- s(S,E,N,D,M,O,R,Y).
 9567
+1085
10652
S = 9
E = 5
N = 6
D = 7
M = 1
O = 0
R = 8
Y = 2
```

**Toto prepíšeme na cvičení**

# VINGT+CINQ+CINQ=TRENTE

(algebrogram – moje riešenie)

```
 VINGT
+ CINQ
+ CINQ
======
TRENTE
```

```
alldiff([]).
alldiff([X|Xs]):- not(member(X,Xs)), alldiff(Xs).

%- scitovanie po stlpcoch
sumCol(Cifra1,Cifra2,Cifra3,Cifra,Prenos,NovyPrenos):-
    NovyPrenos is (Cifra1+Cifra2+Cifra3+Prenos)//10,
                    Cifra is (Cifra1+Cifra2+Cifra3+Prenos) mod 10.

puzzle([V,I,N,G,T,C,Q,R,E]):-
  cifra(T),cifra(Q), alldiff([T,Q]),  sumCol(0,T,Q,Q,E,Pr1), alldiff([E,T,Q]),

  cifra(G),cifra(N),alldiff([G,N,E,T,Q]),  sumCol(Pr1,G,N,N,T,Pr2),

  cifra(I),alldiff([I,G,N,E,T,Q]), sumCol(Pr2,N,I,I,N,Pr3),

  cifra(C),alldiff([C,I,G,N,E,T,Q]),  sumCol(Pr3,I,C,C,E,Pr4),

  cifra(V),alldiff([V,C,I,G,N,E,T,Q]),  sumCol(Pr4,V,0,0,R,T),

  write(' '),write(V),write(I),write(N),write(G),write(T),nl,
  write('  '),      write(C),write(I),write(N),write(Q),nl,
  write('  '),      write(C),write(I),write(N),write(Q),nl,
  write(T),write(R),write(E),write(N),write(T),write(E),nl.
```

```
?- puzzle([V,I,N,G,T,C,Q,R,E]).
 94851
  6483
  6483
------
107817
```

# VINGT+CINQ+CINQ=TRENTE

(algebrogram – iný prístup)

```
solve(V,I,N,G,T,C,Q,E,R) :-
    select(T,[0,1,2,3,4,5,6,7,8,9], R1),    % T∈[1..9],        R1=[1..9] \\ [T]
     select(Q, R1, R2),                     % Q∈[1..9] \\ [T], R2=[1..9] \\ [T,Q]
      sumCol(T,Q,Q,E,0,Pr),                 % 10*Pr+E = T+Q+Q+0
       ;;select(E, R2,R3),
        select(G, R3,R4),
         select(N, R4,R5),
          sumCol(G,N,N,T,Pr,Pr2),
           select(I, R5,R6),
            sumCol(N,I,I,N,Pr2, Pr3),
             select(C,R6,R7),
              sumCol(I,C,C,E,Pr3,Pr4),
               select(V,R7,R8),
                sumCol(V,0,0,R,Pr4,Pr5),
                 T = Pr5,
                  select(R,R8,_), not(T = 0),
    write(' '),write(V),write(I),write(N),write(G),write(T),nl,
    write('  '),        write(C),write(I),write(N),write(Q),nl,
    write('  '),        write(C),write(I),write(N),write(Q),nl,
    write(T),write(R),write(E),write(N),write(T),write(E),nl.
```

```
 VINGT
+ CINQ
+ CINQ
======
TRENTE
```

```
?- solve(V,I,N,G,T,C,Q,E,R).
  94851
   6483
   6483
---------
107817
```