

# **Изкуствен интелект / интелигентни системи**

*гл. асистент д-р Венета Табакова-Комсалова  
ФМИ, ПУ „П. Хилендарски“, Пловдив*

# Упражнение 5

## Ø Списъци

Един списък е последователност от елементи, затворени в скоби **[ ]**.

Например нека е даден списъкът **[витоша, мусала, шипка]**. Първият елемент от списъка се нарича глава или заглавна част (в примера витоша). Останалите елементи на списъка се наричат остатък или опашка (в примера [мусала, шипка]). Опашката се интерпретира отново като списък.

Елементите на списъците могат да бъдат произволни терми, включително и списъци.

Така например, списъкът `[a,[b,c],d]` се състои от три елемента - `a`, `[b,c]` и `d`. Един списък може да не съдържа елементи – нарича се празен списък и се означава `[ ]`.

С помощта на оператора `|` един непразен списък може да бъде разложен на глава и опашка `[глава|опашка]`.

Синтаксисът на Пролог позволява преди `|` да бъдат зададени повече от един елемент. Следващият `|` терм трябва винаги да бъде списък, вкл. Празен. Така например за горния пример съществуват следните алтернативни представяния:

`[витоша, мусала, шипка] =`  
`[витоша |[ мусала, шипка]] =`  
`[витоша, мусала |[шипка]] =`  
`[витоша, мусала, шипка|[ ]]`.

**Списък (определение)** – наредена последователност от 0,1 или повече елементи. Елементите могат да бъдат произволни терми, включително други списъци. (Списъците също са терми – специален вид структури.)

Примери:

$[]$  – празен списък

$[1]$  – списък с един елемент

$[a,b]$  – списък с два елемента

$[p(maria,ivan), 3]$  – списък с два елемента

$[a,b,f(N,k),1,[a,1]]$  – списък с пет елемента

- $[X | L]$  – списък

- $[1, 2, 3, 4, 5]$

Глава

опашка

**Обработка на списъци – чрез отделяне на глава и опашка.**

Глава – първият елемент на списъка

Опашка – списък от елементите без първия

Списък	Глава	Опашка
[a,b]	a	[b]
[1]	1	[]
[]	няма	няма
[X,f(Y),1]	X	[f(Y),1]
[[1,2],[3,4]]	[1,2]	[[3,4]]
[X+Y,p(sofia)]	X+Y	[p(sofia)]

Записът  $[X|Y]$  означава списък с глава  $X$  и опашка  $Y$ . При унификация на даден списък с  $[X|Y]$ , променливата  $X$  се свързва с главата на дадения списък, а променливата  $Y$  – с опашката му. Така е възможен достъп до елементите и обработка на списък.

Примери:

$$[1, 2, 3] = [1|[2,3]] = [1|[2|[3]]]$$
$$[1] = [1|[ ]]$$
$$[1,2] = [1|[2]]$$

## Унификация на списъци – примери.

?-  
[X|Y]=[a,b,c].

X=a, Y=[b,c]

?-  
[X,Y|Z]=[a,b].

X=a, Y=b, Z=[]

?- [X,Y|Z]=[a].

no

?-[X,\_,Y|Z]=[a,b,c,d,  
e].

X=a, Y=c, Z=[d,e]

?-[X,Y]=[a,b,c.d].

no

?-[X,Y]=[a,b].

X=a, Y=b

?-[X,a,X,f(X,a)|Y]=[Z,Z|L].

X=Z=a, Y=\_, L=[a,f(a,a)|Y]

?-[\_,X,7,Y,\_,f(1,abc)|Z]=[9  
9,abc,ABC,X,f(s,s,s),C,3,a  
bc|\_].

X=Y=abc, Z=[3,abc|\_],  
ABC=7, C=f(1,abc)

?-[X,7,[Y|L],34|Z]=[[a,b,c],  
ABC,X,34,a(1,2)|[x,y,z]].

X=[a,b,c], Y=a, L=[b,c],  
Z=[a(1,2),x,y,z],ABC=7

**Дефинициите на предикати, обработващи списъци са обикновено рекурсивни.**

Съществено свойство на списъците е, че те могат да се интерпретират като рекурсивни структурни данни. В съответствие с рекурсивната дефиниция, един списък е :

- Празен списък или
- Структура, състояща се от глава и опашка, която също е списък.

Такава дефиниция описва непосредствено характеристиките (обработката) на главата на списъка или на първите няколко елемента, а характеристиките (обработката) на опашката се описва чрез рекурсивно използване на същия предикат.



В съответствие с това определение можем да дефинираме рекурсивно основните операции върху списъци. В повечето случаи те се дефинират с помощта на две клаузи, както следва:

- Първата клауза описва базовия (нерекурсивен) случай – обикновено това е прилагане на операция върху празния списък или съответната обработка на първия елемент.
- Втората клауза описва рекурсивния случай – т.е. ефекта на операцията върху остатъка от списъка.

## Търсене на елемент в списъка

Нека наречем тази операция елемент/2. Предикатът успява, когато даден обект  $X$  е елемент от списъка  $L$ . Ще дефинираме предиката както следва:  $X$  е елемент от списъка  $L$ , ако:

- $X$  е първият елемент от списъка или
- $X$  е елемент от опашката.

Двата случая – базовия и рекурсивния – ще реализираме посредством двете клаузи:

елемент( $X$ , [ $X$ |Tail]).

елемент( $X$ , [ $Y$ |Tail]) :- елемент( $X$ , Tail).

В различни приложения предикатът елемент/2 се използва обикновено за:

- Тестване на принадлежност;
- Достъп до всички или определени елементи на един списък;
- Генериране на списъци, които съдържат точно определен елемент.

Нека разгледаме някои примери на използване на предиката.

Тест за принадлежност на един елемент (,c'):

?- елемент(c, [a, b, c]).

true.

Последователна обработка на елементите на списъка:

?- елемент(X, [a, b, c]).

X = a ;

X = b ;

X = c ;

false.

Последователно генерираме списъци, които съдържат един постоянен елемент (,a') като първи, втори, трети ... n-ти елемент.

?- елемент(a,L).

L = [a|\_6028] ;      L = [\_6026, a|\_6034] ;

L = [\_6026, \_6032, a|\_6040] ;

L = [\_6026, \_6032, \_6038, a|\_6046] ;

L = [\_6026, \_6032, \_6038, \_6044, a|\_6052]

## Свързване на списъци.

Тази операция ще дефинираме като предикат свържи/3. Предикатът се дефинира също рекурсивно като се специфицират следните два случая:

При свързване на празния списък [] с произволен списък L, получаваме L;

При свързване на непразен списък [X|L1] с произволен списък L2, получаваме списъка [X|L3], където L3 е свързването на списъците L1 и L2.

Изхождайки от тази дефиниция можем да дадем следното описание на предиката:

свържи([], L, L).

свържи([X|L1], L2, [X|L3]) :- свържи(L1, L2, L3).

Пример:

?- свържи([a], [b], X).

X = [a, b].

Интересно е да се отбележи, че така дефинираният предикат е симетричен, т.е. той може да се използва за инверсната операция, която в случая е *разлагане на списък на два отделни списъка*.

Пример: ?- свържи(L1,L2, [a,b]).

L1 = [],

L2 = [a, b] ;

L1 = [a],

L2 = [b] ;

L1 = [a, b],

L2 = [] ;

false.

## Отношение между терм и списък.

$\langle \text{терм} \rangle = .. \langle \text{списък} \rangle$  - списъкът е с глава функтора на терма и опашка списъка от аргументи на терма.

Примери:

?-g(том,котка,5) = ..[X|Y].

X=g, Y=[том,котка,5]

?- Y=[a,1,2,v,e].

Y=a(1,2,v,e)

# Проверка за принадлежност на даден елемент на СПИСЪК

% вграден

member(X,[X|\_]).

member(X,[\_|L]):-

member(X,L).

?- member(1,[1,2,3]).

yes

?- member(1,[4,1,2]).

yes

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

X = 1 ? ;

X = 2 ? ;

X = 3 ? ;

No

[a1, a2, a3, ..., an]

[b1, b2, b3, ..., bm]

[X | L1 ]

L2

[a1, a2, a3, ..., an, b1, b2, b3, ..., bm]

[X | a2, a3, ..., an, b1, b2, b3, ..., bm]

L1

L2

## Слепване на списъци

% вграден

append([],L2,L2).

append([X|L1],L2,[X|L3]):-

append(L1,L2,L3).

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

X = [1,2,3,4] ? ;

no

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

X = [1] ? ;

no

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

X = [3] ? ;

no

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

X = [1,2,3,4] ? ;

no

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

X = [1] ? ;

no

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

X = [3] ? ;

no

?- 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],

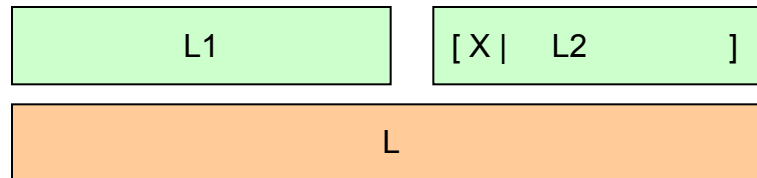
Y = [] ? ;

no

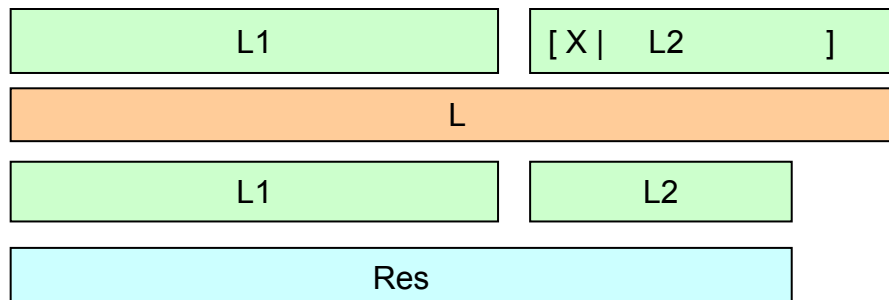


## Дефиниции чрез append

`member(X,L):- append(_,[X|_],L).`



`del(X,L,Res):- append(L1,[X|L2],L), append(L1,L2,Res).`



**Изтриване на  
първото срещане  
на елемент в  
списък**

```
del(X,[X|L],L).  
del(X,[Y|L],[Y|Res]):-  
    del(X,L,Res).
```

**Изтриване на всички  
срещания на елемент в  
списък**

```
del_all(X,[],[]).  
del_all(X,[X|L],Res):- del_all(X,L,Res).  
del_all(X,[Y|L],[Y|Res]):- del_all(X,L,Res).
```

```
?- del(1,[1,2,3],L).
```

```
L = [2,3] ? ;
```

```
no
```

```
?- del(1,[1,2,1,3],L).
```

```
L = [2,1,3] ? ;
```

```
L = [1,2,3] ? ;
```

```
no
```

```
?- del_all(1,[1,2,1,3],L).
```

```
L = [2,3] ? ;
```

```
L = [2,1,3] ? ;
```

```
L = [1,2,3] ? ;
```

```
L = [1,2,1,3] ? ;
```

```
no
```

## Изтриване на всички срещания на елемент в списък

```
del_all(X,[],[]).  
del_all(X,[X|L],Res):-  
    del_all(X,L,Res).  
del_all(X,[Y|L],[Y|Res]):-  
    X \= Y, del_all(X,L,Res).
```

```
?- del_all(1,[1,2,1,3],L).  
L = [2,3] ? ;  
no
```

## Дължина на списък

% вграден

length([ ], 0).

length([X|L],N):-  
    length(L,M),  
    N is M+1.

## Задачи:

1. Напишете предикат, който има два аргумента елемент и списък и е верен, ако елементът е последен елемент на списъка.

`last(X,[X]).`

`last(X,_|Z):-last(X,Z).`

*Особености:*

-Предикатът върви и когато първият аргумент е фиксирана променлива, а вторият свободна. Например при цел

`?-last(a,X).`

предикатът ще връща при преудовлетворяване следните стойности за X:

`[a] Yes.`

`_1|[a] Yes.`

и т. н.

Същественото е, че когато първият аргумент е свободна променлива, а вторият е фиксиран списък, предикатът ще се удовлетвори и ще върне във втория си аргумент последния му елемент.

2. Напишете предикат, който има два аргумента елемент и списък и е верен, ако елементът принадлежи на списъка, но не само на първо, а на произволно ниво.

```
member(Y,[Y|_]).  
member(Y,[_|Z]):-member(Y,Z).  
member(Y,[X|_]):-member(Y,X).
```

*Особености:*

При преудовлетворяване нещата стоят по същият начин, както при обикновения предикат `member`.

3. Напишете предикат, който има три аргумента първите два са елементите Ел1 и Ел2, а третият е списък и е верен, ако елементите Ел1 и Ел2 принадлежат на списъка в този ред и са последователни.

`sl(X,Y,[X,Y|_]).`

`sl(X,Y,[_|Z]):-sl(X,Y,Z).`

4. Напишете предикат, който има два аргумента-списъци и е верен, ако първият аргумент е подсписък на втория.

`match([],_).`

`match([X|L],[X|M]):-match(L,M).`

`sub([X|L],[X|M]):-match(L,M).`

`sub(L,[_|M]):-sub(L,M).`

`sub([],_).`

*Особености:*

Предикатът `match` проверява дали един списък е начало на друг.

5. Напишете предикат, който има два аргумента-списъци и е верен, ако първият аргумент е списък, в който елементите на втория са изброени в обратен ред.

```
rev(L1,L2):-rv(L1,[], L2).
```

```
rv([], L,L).
```

```
rv([X|L], L1,L2):-rv(L,[X|L1], L2).
```

### *Особености*

Предикатът използва помощен предикат, при който имаме една променлива в повече, която е работна, и в нея натрупваме текущия резултат.

6. Реализирайте предикатите за принадлежност и последен елемент чрез append.

```
last1(X,Y):-append(_, [X], Y).
```

```
member1(X,Y):-append(_, [X|_], Y).
```



7. Реализирайте предикат, който има два аргумента списъци, като вторият е пермутация на първия. Направете предиката така че при свободен втори аргумент и при всевъзможните преудовлетворявания във втория аргумент да се връщат всевъзможните пермутации на първия. `perm([],[]).`

```
perm(L,[H|T]):-append(V,[H|U],
L), append(V,U,W), perm(W,T).
```

8. Реализирайте, тъй нареченото, наивно сортиране на списък от естествени числа, което се състои в следното: генерирайте всевъзможните пермутации на един списък от числа, докато получите такава, която е сортирана. Използвайте вградения предикат `=<`, който има инфиксен запис.

```
sort1(L,S):-perm(L,S), is_sort(S).
```

```
is_sort(L):-iss(0,L).
```

```
iss(_,[]).
```

```
iss(N,[M|L]):-N =< M,iss(M,L).
```

9. Реализирайте quick sort за списъци. Използвайте вградения предикат  $=<$  и  $>$  които имат инфиксен запис.

hit(\_,[],[],[]).

hit(H,[A|X],[A|Y], Z):-A  $=<$  H, hit(H,X,Y,Z).

hit(H,[A|X], Z,[A|Y]):-A  $>$  H, hit(H,X,Z,Y).

qs([],[]).

qs([H|T], S):-hit(H,T,A,B), qs(A,A1), qs(B,B1),  
append(A1,[H|B1], S).

## 10. Пресмятане на сумата от елементите на списък от числа

`sum([X],X).`

`sum([X|L],N):-sum(L,M),N is M+X.`

`?- sum([1,2,3],T).`

`T=6`

% N елемент от списък

```
nth([X|_],1,X).
```

```
nth(_|L,N,X):- M is N-1, nth(L,M,X).
```

% Последен елемент от списък

```
last([X],X).
```

```
last(_|L,X):- last(L,X).
```

% Брой

```
count([ ],_,0).
```

```
count([X|L],X,N):-  
    count(L,X,K),  
    N is K+1.
```

```
count([Y|L],X,N):-  
    Y \= X,  
    count(L,X,N).
```

## Предикати за управление:

**Отсичане** - това е вграден предикат, който синтактично изглежда по следния начин: `cut` или за по-кратко **!**, който винаги се удовлетворява, но при опит за преудовлетворяване пропада. Извикан от тялото на някое правило, той означава "зарежи" възможните алтернативни решения, защото се интересуваме само дали можем да минем по точно този път. Т. е. когато в програмата се срещне отсичане, то то "отсича" пътя, който представлява редицата на доказателствата, така че следващата цел се съединява с предишната, или ако си представим дървото на извод то залепя всички възли от дясно, като непосредствени наследници на родителския възел на целта, от която е извикано. Или казано накратко, маха предшественика и всичките му наследници от ляво. Като при това когато Пролог срещне **!**, той се "сеща" как точно ще продължи механизмът на възврат и за тези цели, които не могат да се преудовлетворяват не пази маркери, т. е. с **!** освен че можем да ускорим програмата, защото режем някои пътища, пестим и памет.

Случаи, в които може да се използва !:

-ако сме сигурни, че възможните преудовлетворявания няма да доведат до получаването на нещо по-добро.

Например предикатът `member`, ако преудовлетворяваме, ще получаваме `yes` толкова пъти, колкото срещания имаме. Ако искаме просто еднократно да ни казва дали даден елемент се среща или не в даден списък, трябва да го запишем така:

```
member(X,[X|_]):-!.
```

```
member(X,_|L):-member(X, L).
```

-ако искаме да фиксираме пътя, по който трябва да минем. Например ако искаме да реализираме следната конструкция:

```
if a then b else c.
```

```
if:-a,! ,b.
```

```
if:-c.
```

**-в комбинация с fail.** fail е вграден предикат, който никога не успява. Тази конструкция се използва в случай, че искаме да окажем, че попаднем ли в дадена ситуация, непременно трябва да имаме отговор по.

Трябва да се внимава при употребата на !. Наистина той твърде много ускорява работата на програмата, но за съжаление доста усложнява разчитането ѝ, защото ние вече не работим чисто декларативно, което прави трудно откриването на евентуални грешки. За съжаление в Пролог, за разлика от чистото логическо програмиране, употребата на ! не може да се избегне.

**call**- това е предикат на един аргумент, който трябва да бъде цел за Пролог и се удовлетворява, ако целта се удовлетворява, иначе пропада.

**not**- това е предикат отново на един аргумент- цел за Пролог, който пропада, ако целта се удовлетворява, и успява иначе.

Внимание: това не е традиционното логическо отрицание, ако на `not` му подадем цел, при чието изпълнение попадаме в безкраен цикъл, тогава и `not` ще зацikli, а няма да върне `yes`. Как се реализира `not` в Пролог:

```
not(C):-call(C),!,fail.
```

```
not(C).
```

Когато не сме сигурни как ще отговори `not`, най-добре е да си спомним дефиницията му.

**repeat**- за разлика от `!` този предикат винаги се удовлетворява и преудовлетворява. Той служи за правенето на цикъл. Каква му е дефиницията:

```
repeat.
```

```
repeat:-repeat.
```



# Задачи

## Задача 1: Пет деца – таланти

В семейство има пет деца (братя и сестри), съответно на 4,5,6,7 и 8 години, които имат различни таланти. Едното от тях се казва Невена, а друго свири на пиано. Иванка е на 4 години и не разбира от математика. Детето, което е програмист е с една година по-голямо от Иван. Детето, което свири на китара е на 7 години. Йоана не е на 8 години. Станко е на 5 години и е по-малък от този, който разбира от литература. Кой на колко години е и в каква насока е талантът му?

# ПОДРОБНО ОПИСАНИЕ НА РЕШЕНИЕТО

deca(Ds)

Ds е списък от деца

В задачата е дадено:

1. В семейство има 5 деца, съответно на 4, 5, 6, 7 и 8 години, които имат различни таланти.
  2. Едно от тях се казва Невена,
  3. а друго свири на пиано, т.е Невена не свири на пиано
  4. Иванка е на 4 години и не разбира от математика.
  5. Детето, което е програмист е с една година по-голямо от Иван
  6. Детето, което свири на китара е на 7 години.
  7. Йоана не е на 8 години.
  8. Станко е на 5 години и е по-малък от този, който разбира от литература.
- Кой на колко години е и в каква насока е талантът му?

% Създаваме таблица за децата

:- use\_rendering(table,[header(d('Име', 'Талант',  
'Години'))]).

deca(Ds) :-

% всяко дете в списъка съдържа данни за:

% d(Name, Talent, Age)

length(Ds, 5), % дължина на списъка с деца Ds

%Проверка за принадлежност на даден елемент на списъка

member(d(ivanka,\_,4), Ds), % Иванка е на 4 години

member(d(\_,kitara,7), Ds),

% Детето, което свири на китара е на 7 години

member(d(ivan,\_,\_),Ds), % Дете Иван

member(d(nevena,\_,\_), Ds), % Дете Невена

member(d(stanko,\_,5), Ds), % Станко е на 5 години

```
member(d(_,programist,_), Ds),    % Дете - програмист
    member(d(yoana,_,_),Ds),    % Дете Йоана
    member(d(_,literatura,6),Ds),
% Детето, което разбита от литература е по-голямо от Станко и е на
% 6 години
    member(d(_,piano,_),Ds),    %Дете, което свири на пиано
    member(d(_,matematika,_),Ds),% Дете - математик
    member(d(_,_,8),Ds),        % Дете, което е на 8 години
    ((member(d(ivan,_,6),Ds), member(d(_,programist,7),Ds)));
% ако Иван е на 6 години, програмиста е на 7 години
    (member(d(ivan,_,7),Ds), member(d(_,programist,8),Ds))),
% ако Иван е на 7 години, програмиста е на 8 години
```

```
not(member(d(ivanka,matematika,_),Ds)),  
% Иванка не е математик
```

```
not(member(d(nevena,piano,_),Ds)),  
% Невена не свири на пиано
```

```
not(member(d(yoana,_,8),Ds)),  
% Йоана не е на 8 години
```


```
/* искаме списъка да бъде подреден по възрастта на  
децата */
```

```
Ds= [d(_,_,4),d(_,_,5),d(_,_,6),d(_,_,7),d(_,_,8)].
```

```
/* ?-deca(Deca). */
```

## **Задача 2. Задачата за купето**

Веднъж се случило в едно купе да пътуват астроном, поет, писател и драматург. Това са Алексеев, Борисов, Дмитриев и Константинов. Оказало се, че всеки един от тях е взел книга, написана от един от другите пътници в купето. Алексеев и Борисов потънали в четене, като предварително си разменили книгите. Поетът чете пиеса. Писателят, един много млад мъж, оставяйки книгата си казва, че никога и нищо не е чел по астрономия. Борисов е купил една от книгите на Дмитриев. Никой от пътниците не чете своя книга. Какво чете всеки един от тях?



Преди да попитаме машината за логически извод за решението на задачата е необходимо да се формализират фактите.

В тази задача са дадени 4 имена, 4 професии и 4 вида книги. Професията не е необходимо да се задава, защото тя може да се получи от книгата.

```

:- use_rendering(table,[header(patnik('Име', 'Чете', 'Купил', 'Написал'))]).
maje(alekseev). maje(borisov). maje(dmitriev). maje(konstantinov).
napisana_kniga(astronomiq). napisana_kniga(poeziq). napisana_kniga(piesa). napisana_kniga(drama).
%patnik(_Име, _Чете, _Купил, _Написал).
unikat([]).
unikat([A|As]):-maplist(dif(A),As), unikat(As).
proverka([]):-!.
proverka([patnik(_, AChete, AKupil, ANapisal)|X]):-!, not(AChete = ANapisal), not(AKupil = ANapisal),
proverka(X).
reshi(Reshenie):-
Reshenie = [patnik(A, AChete, AKupil, ANapisal), patnik(B, BChete, BKupil, BNapisal),
patnik(C, CChete, CKupil, CNapisal), patnik(D, DChete, DKupil, DNapisal)],
maje(A), maje(B), maje(C), maje(D), unikat([A, B, C, D]),
napisana_kniga(ANapisal),napisana_kniga(BNapisal),napisana_kniga(CNapisal),napisana_kniga(DNapisal),
unikat([ANapisal, BNapisal, CNapisal, DNapisal]),
napisana_kniga(AKupil),napisana_kniga(BKupil),napisana_kniga(CKupil),napisana_kniga(DKupil),
unikat([AKupil, BKupil, CKupil, DKupil]),
napisana_kniga(AChete),napisana_kniga(BChete),napisana_kniga(CChete),napisana_kniga(DChete),
unikat([AChete, BChete, CChete, DChete]),
% Алексиев и Борисов са си сменили кн.
member(patnik(alekseev, AlekseevChete, AlekseevKupil, _), Reshenie),
member(patnik(borisov, AlekseevKupil, AlekseevChete, _), Reshenie),
% Поетът чете пиеса
member(patnik( _, piesa,_, poeziq), Reshenie),
% Писателят не чете и съответно не е купил астрономия
not(member(patnik(_, astronomiq,_, piesa), Reshenie)),
not(member(patnik( _,_, astronomiq, piesa), Reshenie)),
% Борисов купил книга написана от Димитриев
member(patnik(dmitriev,_,_, DmitrievNapisal), Reshenie),
member(patnik(borisov, DmitrievNapisal,_,_), Reshenie),
% Всички четат чужда книга
proverka(Reshenie).

/* ЦЕЛ ?- reshi(X), X = [patnik(alekseev,_,_,_), patnik(borisov,_,_,_), patnik(konstantinov,_,_,_), patnik(dmitriev,_,_,_)], !. */

```