3. СТРУКТУРИ В ПРОЛОГ

Въведение. Една от най-широко използваните в програмирането структури са списъците. Списъкът в Пролог е поредица от поставени в квадратни скоби елементи. Елементите могат да бъдат например цели числа или поредици от букви (string):

list1 = [1, 2, 3, 5, 4, 7].

list2 = [“Иво”,”Мая”,”Петя”,”Сашо”].

Всеки списък трябва да бъде деклариран в раздела за декларации, който в Пролог се означава с думата **domains**. За горните два списъка е необходимо да се запише

domains

list1 = integer *

list2 = symbol *

Звездичката означава, че списъкът съдържа неопределен брой елементи от посочения тип. Засега при използването на PIE този и други раздели за декларации и дефиниции не се използват. Те ще бъдат разгледани по-нататък в ръководството.

Всеки списък има глава и опашка. Първият елемент от списъка се нарича глава (head). Списъкът от всички останали елементи (без първия) се нарича опашка (tail). Обърнете внимание, че главата е елемент, а опашката е списък. Опашката може да е и списък от един елемент или празен списък.

Предикатите боравят със списъците, като ги проследяват отляво надясно, отделят главата и преобразуват опашката в нов списък. Ето няколко примера за действията на предиката number([1,2,3,5,4,7]).

При задаване на цел:

Goal: number([H|T]) се получава

H=1,

T=[2,3,4,5,6,7].

Goal: number([A, B, C | T])

A = 1

B = 2

C = 3

T=[4,5,6,7]

При Goal: number([A,B,C,D,E,F|T]) променливите от A до F получават стойности от 1 до 6, а опашката T=[7] е списък от последния елемент.

При Goal: number([A,B,C,D,E,F,G|T]) променливите от A до G получават стойности от 1 до 7, а опашката T=[] е празен списък.

При Goal: number([A,B,C,D,E,F,G,J,K|T]) резултатът е сигнал за грешка.

Забележете, че в клаузите задавахме променливи за главата и за опашката, като ги отделяме с права черта (|). Когато в списъка няма достатъчно променливи за посочения брой глави, се получава грешка.

Постановка. Ще съставим няколко удивиделно къси програми за обработка на списъци. Основната конструкция в тях е така наречената **рекурсия**. *Това е правило, в което клаузата глава е дефинирана в тялото си чрез същата тази клауза*. Това изглежда на пръв поглед странно, но е много мощно средство на езика Пролог. При работата с рекурсивни клаузи трябва строго да се спазват следните две изисквания:

1) рекурсията трябва да напредва, като намалява възможностите за търсене на решения;

2) рекурсията трябва да спре.

Пример 1. Да се изведат върху екрана елементите на списък в колонка един под друг.

Упътване**.** Най-елегантното решение е с използването на рекурсия. Условието за край на рекурсията е списъкът да остане празен. Първата клауза е клауза за край.

Правилото за разпечатване на елементите трябва да взема елемента, който е глава на списъка, да го извежда на екрана, да преминава на нов ред и отново да извиква същия предикат, но с опашката на списъка.

Програма.

```
write_list([]).
write_list([H|T]):- write(H),nl,write_list(T).
```

Goal: write_list([1,2,3,4,5])

Работа на Пролог. Разглежда се първата клауза write_list([1,2,3,4,5]). Тъй като списъкът не е празен, първата клауза не се унифицира (резултатът е false). Преминава се към втората клауза. Отделя се главата на списъка H=1. Изпълнява се нейното тяло като стойността на H се отпечатва (write(H)), преминава се на нов ред (nl) и се извиква рекурсивно отново предикатът write_list(T) с аргумент опашката на списъка.

Когато опашката стане празен списък, първата клауза се унифицира (резултатът е true) и програмата завършва.

Пример 2. Да се провери дали един елемент е член на даден списък.

Упътване. Предикатът, нека го наречем member, трябва да има два аргумента – елемента, за който ще проверяваме дали е член на списъка, и самия списък.

За да бъде член на списък един елемент, трябва да съвпада с главата на списъка или да се съдържа в опашката му. Следователно условието за край ще бъде разглежданият елемент да съвпадне с главата на списъка.

Правилото, което ще ни приближава към клаузата за край всеки път, ще съкращава списъка с един елемент, като отстранява главата на списъка и задава като нов аргумент опашката му.

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

Goal: member(5,[1,2,3,4,5,6,7])
True
1 Solution

Работа на Пролог. Първата клауза пропада, тъй като при изпълнението й се установява, че главата на зададения списък е числото 1, търсеното число е 5, а записът X и за глава, и за търсено число изисква равенство на двете. Втората клауза се активира без значение каква е стойността на главата (за глава е посочена анонимна променлива) и при изпълнението си извиква рекурсивно предикатът member, но с втори аргумент опашката на списъка. Така се отстранява първият елемент на списъка и се работи със списък от оставащите елементи [2,3,4,5,6,7]. При всяко рекурсивно извикване отпада първият от оставащите елементи на списъка.

Когато елементът съвпадне с пореден елемент-глава на списъка, първата клауза се унифицира, изпълнява се нейното тяло, което съдържа само предиката cut, според който не трябва да се търсят повече решения, и програмата завършва с True. Ако елементът не се открие в списъка, след изчерпване на елементите му програмата завършва с false.

Пример 3. Да се провери дали конкретен елемент се среща само веднъж в зададен списък.

Упътване. Заданието съдържа два компонента: (а) да се установи дали елементът се съдържа в списъка и (б) да се установи дали този елемент се съдържа само веднъж в даден списък (т.е. дали елементът е уникален).

Първата част вече е решена с програмката от предния пример. За решаването на втората част се съобразява, че един елемент е уникален за даден списък, ако съвпада с главата на списъка и не се съдържа в опашката му.

Програма.
```
member(X,[X|_]):-!.
member(X,[_|T]):- member(X,T).
unique_member(X,[X|T]) :- !, not(member(X,T)).
unique_member(X,[_|T]) :- unique_member(X,T).
```

Работа на Пролог. Проследяването може да се направи по следната трасировка на изпълнението.

```
Goal
unique_member(5,[1,5,3,7,5,6,5])
Trace is On
Trace: >> CALL:   unique_member(5,[1,5,3,7,5,6,5])
Trace: >> CALL:   unique_member(5,[5,3,7,5,6,5])
Trace: >> CALL:   member(5,[3,7,5,6,5])
Trace: >> CALL:   member(5,[7,5,6,5])
Trace: >> CALL:   member(5,[5,6,5])
Trace: >> RETURN: member(5,[5,6,5])
Trace: >> RETURN: member(5,[7,5,6,5])
Trace: >> RETURN: member(5,[3,7,5,6,5])
Trace: >> FAIL:   unique_member(5,[5,3,7,5,6,5])
Trace: >> FAIL:   unique_member(5,[1,5,3,7,5,6,5])
No solutions
```

Обърнете внимание как при извикванията на клаузите елементите на списъка изчезват един по един и как при връщането назад те се появявят в обратен ред. Машината за извод използва за тази цел структурата стек.

Пример 4. Да се състави програма за съединяването (конкатенация) на два списъка в трети списък, в който елементите на изходните списъци запазват подреждането си.

Упътване. За реализирането на програмата се построява триместен предикат append(list, list, list), който съединява списъците от първата и втората позиция и помества резултата в третата позиция на предиката.

Програмата ще бъде рекурсивна с условие за край, в което списъкът в първата позиция е празен списък. Правилото за достигане на клаузата за край ще съкращава списъка от първа позиция на предиката с по един елемент при всяко повикване и ще възстановява при всеки възврат тези елементи в обратен ред, като ги помества в главата на списъка от трета позиция на предиката. Това се реализира, като се използва едно и също име за променливата на главата на първия и третия списък.

Програма.

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

Да приведем няколко резултата от работата на тази програма.

Цели:

append([1,2,3],[4,5,6], Y)

Y= [1,2,3,4,5,6]

1 Solution

append(X,[5,6,7],[1,2,3,4,5,6,7])

X= [1,2,3,4]

1 Solution

append([1,2,3],Y,[1,2,3,4,5])
Y= [4,5]
1 Solution
append (X,Y,[1,2,3,4,5,6])
X= [], Y= [1,2,3,4,5,6]
X= [1], Y= [2,3,4,5,6]
X= [1,2], Y= [3,4,5,6]
X= [1,2,3], Y= [4,5,6]
X= [1,2,3,4], Y= [5,6]
X= [1,2,3,4,5], Y= [6]
X= [1,2,3,4,5,6], Y= []
7 Solutions

Този предикат илюстрира мощта на декларативното програмиране. Едни и същи клаузи могат да се използват за съвсем различни цели, някои от които дори не са и поставяни.

Работата на Пролог.

По зададени втори и трети списък append възстановява първия списък.

Trace Call On

append(X, [5,6,7],[1,2,5,6,7])
Trace: >> CALL: append(_,[5,6,7],[1,2,5,6,7])
Trace: >> CALL: append(_,[5,6,7],[2,5,6,7])
Trace: >> CALL: append(_,[5,6,7],[5,6,7])
Trace: >> RETURN: append([],[5,6,7],[5,6,7])
Trace: >> RETURN: append([2],[5,6,7],[2,5,6,7])
Trace: >> RETURN: append([1,2],[5,6,7],[1,2,5,6,7])
X= [1,2]
Trace: >> REDO: append([1,2],[5,6,7],[1,2,5,6,7])

При неизвестни първи и втори списък append намира всички възможни варианти на списъци, от които може да се получи третият списък.

append(X,Y, [1,2,3])
Trace: >> CALL: append(_,_,[1,2,3])
Trace: >> RETURN: append([],[1,2,3],[1,2,3])
X= [], Y= [1,2,3]
Trace: >> REDO: append([],[1,2,3],[1,2,3])
Trace: >> CALL: append(_,_,[2,3])
Trace: >> RETURN: append([],[2,3],[2,3])
Trace: >> RETURN: append([1],[2,3],[1,2,3])
X= [1], Y= [2,3]
Trace: >> REDO: append([1],[2,3],[1,2,3])
Trace: >> REDO: append([],[2,3],[2,3])
Trace: >> CALL: append(_,_,[3])
Trace: >> RETURN: append([],[3],[3])
Trace: >> RETURN: append([2],[3],[2,3])
Trace: >> RETURN: append([1,2],[3],[1,2,3])

```
X= [1,2], Y= [3]
Trace: >> REDO:   append([1,2],[3],[1,2,3])
Trace: >> REDO:   append([2],[3],[2,3])
Trace: >> REDO:   append([],[3],[3])
Trace: >> CALL:   append(_,_,[])
Trace: >> RETURN: append([],[],[])
Trace: >> RETURN: append([3],[],[3])
Trace: >> RETURN: append([2,3],[],[2,3])
Trace: >> RETURN: append([1,2,3],[],[1,2,3])
X= [1,2,3], Y= []
Trace: >> REDO:   append([1,2,3],[],[1,2,3])
Trace: >> REDO:   append([2,3],[],[2,3])
Trace: >> REDO:   append([3],[],[3])
Trace: >> REDO:   append([],[],[])
Trace: >> FAIL:   append(_,_,[])
Trace: >> FAIL:   append(_,_,[3])
Trace: >> FAIL:   append(_,_,[2,3])
Trace: >> FAIL:   append(_,_,[1,2,3])
4 Solutions
```

Задание

1. Проследете изпълнението на програмата от Пример 1 за различни списъци, например **Goal**: write_list([“Име1”,”Име2”,”Име3”,”Име4”]).

.

2. Проследете изпълнението на програмата от Пример 2 за различни списъци и за различни елементи, например **Goal:** member(8,[1,2,3,4,5]), **Goal:** member("Ira", ["Ina", "Iva", "Ira", "Inna"]) и др.

Защо е необходим предикатът cut(!) в този пример?

Забележка. Променете програмата от примера така, че да ви показва колко пъти конкретен елемент се съдържа в зададен списък.

3. Проследете изпълнението на програмата за различни списъци и за различни случаи.

4. Съставете програма за намиране на последния елемент на списък.

Забележка. Използвайте рекурсия. Условието за край е списъкът да има само глава; тя е последният елемент на списъка (last([X],X)). Прогресиращият предикат съкращава списъка (last([_|T],X):- last(T,X)).

Изпълнете и трасирайте изпълнението за няколко списъка.

5. Открийте предназначението на следната програма

```
n_element([X|_],1,Y) :- X=Y.
n_element([_|T],N,Y) :- M is N - 1, n_element(T,M,Y).
```

Пояснение. Изпълнението на предиката **M is N – 1** заставя M да получи стойност N – 1, където M и N са цели числа.

Ето един пример за изпълнение на програмата:

Goal n_element([1,2,8,3,6,7,9],5,Y)
Y= 6
1 Solution

6. Проведете няколко експеримента с програмата за съединяване на списъци. Опитайте се да получите от три, четири и пет списъка един общ списък.

Забележка. Преди да извършите сливането, си планирайте работата.

7**.** Да се състави програма за подреждане на елементите на един списък в обратен ред.

Упътване. Необходимо е прехвърлянето на главата на зададения списък за глава на новия списък да се извършва на правия ход на рекурсията.

Програма.

```
reverse(X,Y) :- change(X,[],Y).
change([],Y,Y).
change([H|T],X,Y) :- change(T,[H|X],Y).
```

Работата на Пролог.

Trace Calls On

reverse([1,2,3,4],Y)

```
Trace: >> CALL:   reverse([1,2,3,4],_)
Trace: >> CALL:   change([1,2,3,4],[],_)
Trace: >> CALL:   change([2,3,4],[1],_)
Trace: >> CALL:   change([3,4],[2,1],_)
Trace: >> CALL:   change([4],[3,2,1],_)
Trace: >> CALL:   change([],[4,3,2,1],_)
Trace: >> RETURN: change([],[4,3,2,1],[4,3,2,1])
Trace: >> RETURN: change([4],[3,2,1],[4,3,2,1])
Trace: >> RETURN: change([3,4],[2,1],[4,3,2,1])
Trace: >> RETURN: change([2,3,4],[1],[4,3,2,1])
Trace: >> RETURN: change([1,2,3,4],[],[4,3,2,1])
Trace: >> RETURN: reverse([1,2,3,4],[4,3,2,1])
Y= [4,3,2,1]
```

…
1 Solution

8. Да състави програма за намиране средната стойност на последователност от няколко елемента на списък.

Програма.

```
list_suma_pointer([],0,0).
list_suma_pointer([H|T],S,P) :- list_suma_pointer(T,S1,P1), S is S1+H, P is P1+1.
average(L,X):- list_suma_pointer(L,S,P), X is S/P.
```

Цел:

```
average([1,2,3,4,5],Y)
Y= 3
1 Solution
```