Функциональное и логическое программирование

Лекция 6

2.9 Внелогические предикаты управления поиском решений

Поиск решений Пролог-системой – автоматический полный перебор всех вариантов с возвратом при неуспехе. Это полезный программный механизм, т.к. освобождает от необходимости программировать такой перебор. Но, с другой стороны, неограниченный перебор может стать источником неэффективности программы.

2.9.1 Откат после неудач, предикат fail

Предикат fail всегда неудачен, поэтому инициализирует откат в точки поиска альтернативных решений.

Пример 1:

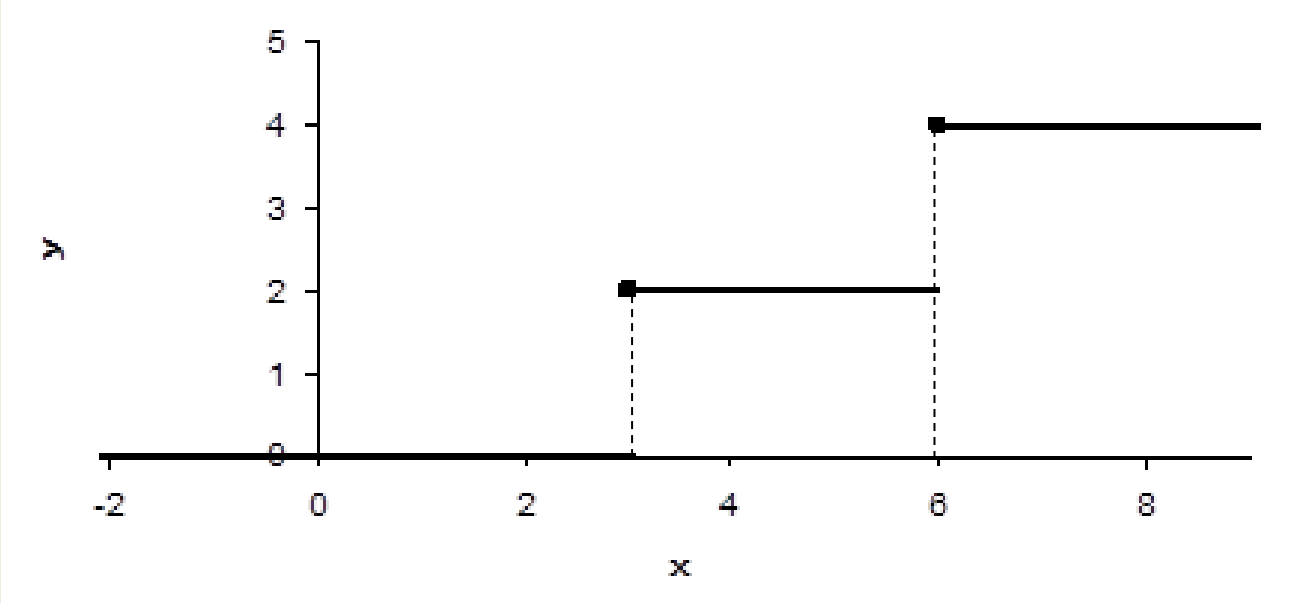
Определим двуместный предикат сотрудник, который связывает ФИО и возраст сотрудника.

Определим предикат, который выводит всех сотрудников до 40 лет.

```
сотрудник(a,25).
сотрудник(b,45).
сотрудник(c,28).
сотрудник(d,50).
сотрудник(e,36).
сотрудник(f,18).
до_40:-сотрудник(X,Y),Y<40,writeln(X),fail.
до_40.
```

2.9.2 Ограничение перебора – отсечение

Пример 2:



Аналитическое задание функции:

$$f(x) = \begin{cases} 0, & x \le 3 \\ 2, & 3 < x \le 6 \\ 4, & x > 6 \end{cases}$$

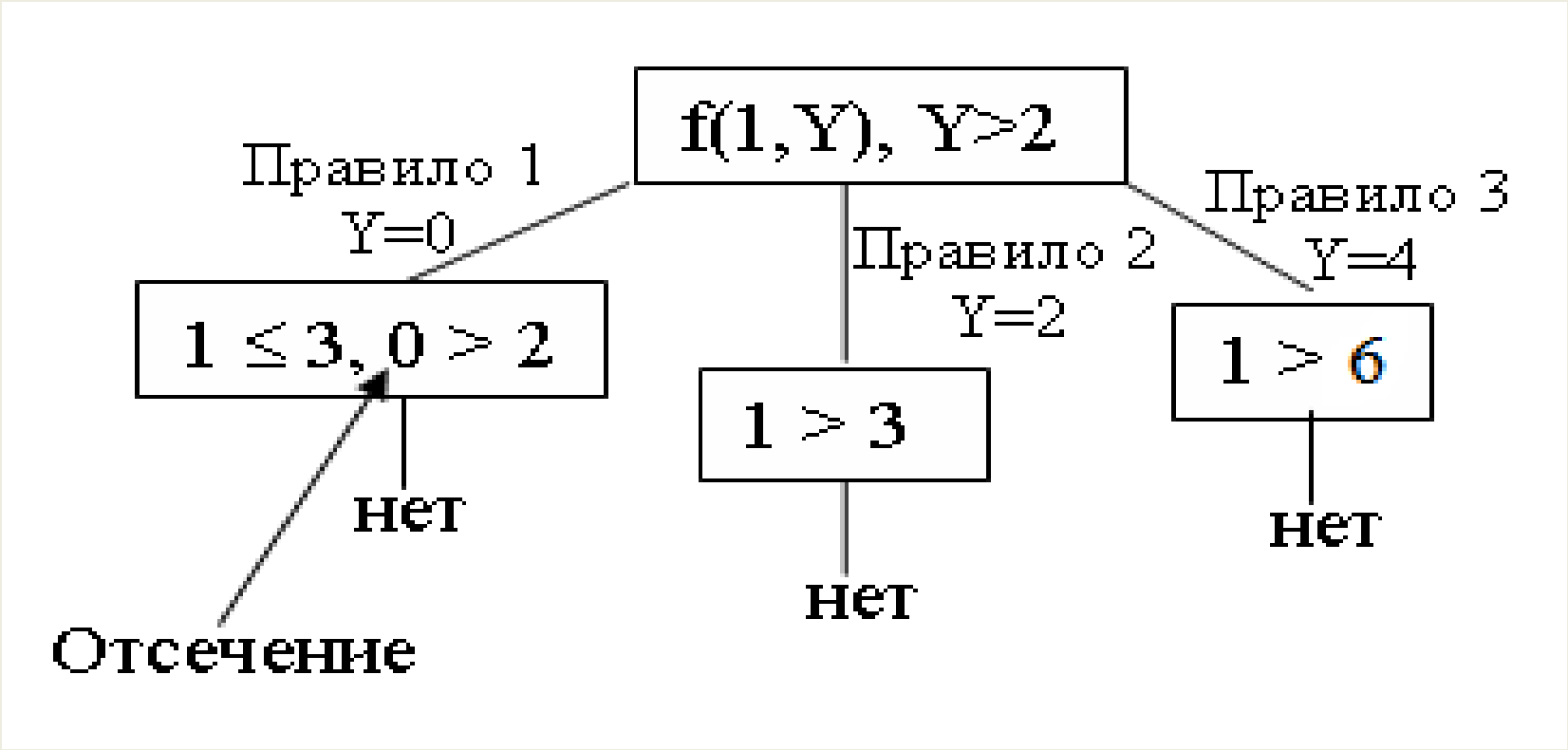
Предикат на Прологе:

```
f(X,0):-X=<3.
f(X,2):-X>3,X=<6.
f(X,4):- X>6.
```

Рассмотрим, как Пролог будет искать решение при цели (с помощью трассировки):

```
[trace]  ?- f(1,Y), Y>2.
   Call: (11) f(1, _35292) ? creep
   Call: (12) 1=<3 ? creep
   Exit: (12) 1=<3 ? creep
   Exit: (11) f(1, 0) ? creep
   Call: (11) 0>2 ? creep
   Fail: (11) 0>2 ? creep
   Redo: (11) f(1, _35292) ? creep
   Call: (12) 1>3 ? creep
   Fail: (12) 1>3 ? creep
   Redo: (11) f(1, _35292) ? creep
   Call: (12) 1>6 ? creep
   Fail: (12) 1>6 ? creep
   Fail: (11) f(1, _35292) ? creep
false.
```

Видно, что проверки условий во втором и третьем правилах излишни (условия в правилах являются взаимоисключающими).



О том, что правило 1 успешно становится известно в точке, обозначенной на рисунке словом «Отсечение». Из этой точке не надо делать возврат к правилам 2 и 3. Для запрета возврата используется предикат ! (отсечение).

Предикат ! всегда успешен и предотвращает возврат из тех точек программы, где он находится.

Добавим отсечения в определение функции:

```
f(X,0):-X=<3,!.
f(X,2):-X>3,X=<6,!.
f(X,4):- X>6.
```

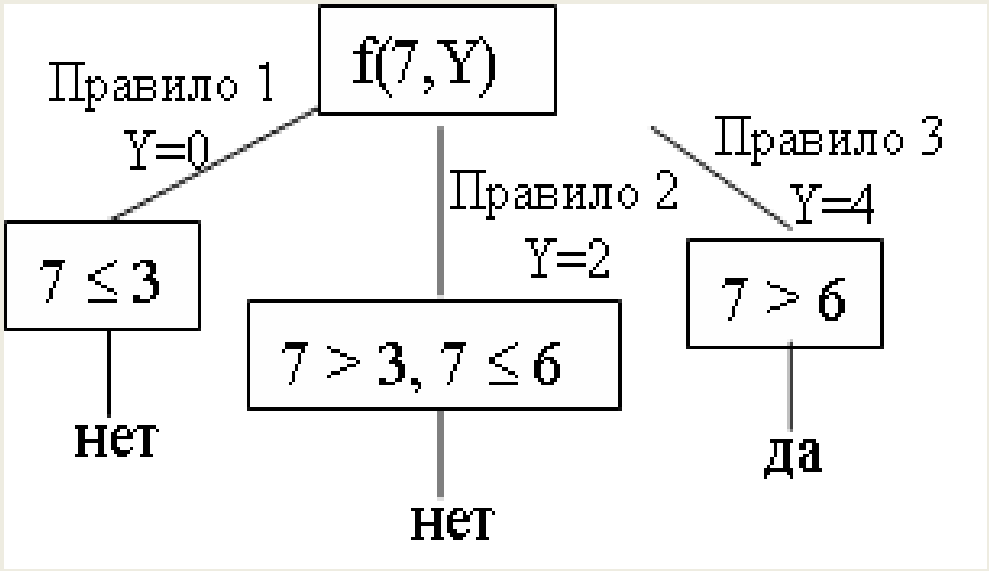
```
[trace]  ?- f(1,Y), Y>2.
   Call: (11) f(1, _5374) ? creep
   Call: (12) 1=<3 ? creep
   Exit: (12) 1=<3 ? creep
   Exit: (11) f(1, 0) ? creep
   Call: (11) 0>2 ? creep
   Fail: (11) 0>2 ? creep
false.
```

Теперь при поиске решения альтернативные ветви, соответствующие правилам 1 и 2, порождены не будут. Программа станет эффективнее.

Если убрать отсечения, программа выдаст тот же результат, хотя на его получение она затратит, скорее всего, больше времени. В данном случае отсечения изменили только процедурный смысл программы (теперь проверяется только левая часть дерева решений), не изменив ее декларативный смысл.

Еще один источник неэффективности можно увидеть, если задать цель:

```
[trace]  ?- f(7,Y).
   Call: (10) f(7, _7370) ? creep
   Call: (11) 7=<3 ? creep
   Fail: (11) 7=<3 ? creep
   Redo: (10) f(7, _7370) ? creep
   Call: (11) 7>3 ? creep
   Exit: (11) 7>3 ? creep
   Call: (11) 7=<6 ? creep
   Fail: (11) 7=<6 ? creep
   Redo: (10) f(7, _7370) ? creep
   Call: (11) 7>6 ? creep
   Exit: (11) 7>6 ? creep
   Exit: (10) f(7, 4) ? creep
Y = 4 ;
false.
```



Нет необходимости проверять условие 7>3, (уже проверили невыполнение условия 7≤3) и условия 7>6 (уже проверили невыполнение условия 7≤6).

Новое определение функции:

f(X,0):-X=<3,!.

f(X,2):-X=<6,!.

f(_,4).

Но если из этой программы убрать отсечения, то она будет не всегда правильно работать.

f(X,0):-X=<3.
f(X,2):-X=<6.
f(_,4).

?- f(2,Y).
Y = 0 ;
Y = 2 ;
Y = 4 ;
false.

Таким образом, теперь отсечения затрагивают декларативный смысл программы.

Отсечения, которые не затрагивают декларативный смысл программы, называются *зелеными*.

Отсечения, меняющие декларативный смысл программы называются *красными*. Их следует применять с большой осторожностью.

Часто отсечение является необходимым элементом программы - без него она правильно не работает.

Работа механизма отсечений:

$$Н : -В_1, \ldots, В_k, !, \ldots, В_n$$

Если цели $В_1, \ldots, В_k$ успешны, то это решение замораживается, и другие альтернативы для этого решения больше не рассматриваются (отсекается правая часть дерева решений, которая находится выше $В_1, \ldots, В_k$).

3 основных случая использования отсечения:

1. Указание интерпретатору Пролога, что найдено *необходимое правило* для заданной цели.
2. Указание интерпретатору Пролога, что необходимо *немедленно прекратить* доказательство конкретной цели, не пытаясь рассматривать какие-либо альтернативы.
3. Указание интерпретатору Пролога, что в ходе перебора альтернативных вариантов найдено *необходимое решение*, и нет смысла вести перебор далее.

Пример3:
Вычисление суммы ряда натуральных чисел 1, 2, ... N.

```
sum(1,1).
sum(N,S):-N1 is N-1,sum(N1,S1),S is S1+N.

?- sum(2,S).
S = 3 ;
ERROR: Stack limit (1.0Gb) exceeded
```

Добавим отсечение:

```
sum(1,1):-!.
sum(N,S):-N1 is N-1,sum(N1,S1),S is S1+N.

?- sum(2,S).
S = 3 ;
false.
```

?- sum(-3,S).
ERROR: Stack limit (1.0Gb) exceeded sum(X,_):-X<0,!,fail.

Окончательный вариант определения предиката sum:

```
sum(X,_):-X<0,!,fail.
sum(1,1):-!.
sum(N,S):-N1 is N-1,sum(N1,S1),S is S1+N.
```

2.10 Циклы, управляемые отказом

Имеется встроенный предикат без аргументов repeat, который всегда успешен.

Его определение:

repeat.

repeat:-repeat.

Реализация цикла «до тех пор, пока»:

<голова правила>:- repeat,

<тело цикла>,

<условие выхода>,!.

Пример:

Определим предикат, который считывает слово, введенное с клавиатуры, и дублирует его на экран до тех пор, пока не будет введено слово «stop».

```
goal:-writeln('Введите слова для дублирования'),эхо.
эхо:-repeat, read(Slovo), writeln(Slovo), проверка(Slovo),!.
проверка(stop).
проверка(_):-fail.
```

2.11 Списки

Список – упорядоченный набор объектов (термов). Список может содержать объекты разных типов, в том числе и списки. Элементы списка разделяются запятыми и заключаются в квадратные скобки.

Пример 1: `[]`, `[[a,b],c,d]`, `[1,a,[[[d]]]]`

2.11.1 Голова и хвост списка

Голова списка – первый элемент.
Хвост списка – часть списка без первого элемента.

Пример 2 (деление списка на хвост и голову):

Список	Голова	Хвост
[1,2,3,4]	1	[2,3,4]
[a]	a	[ ]
[ ]	не определено	не определен

Деление на голову и хвост осуществляется с помощью специальной формы представления списка: [Head|Tail].

Пример 3 (Сопоставление списков):

Список 1	Список 2	Результаты сопоставления
[X,Y,Z]	[1,2,3]	X=1, Y=2, Z=3
[5]	[X|Y]	X=5, Y=[]
[1,2,3,4]	[X,Y|Z]	X=1, Y=2, Z=[3,4]
[1,2,3]	[X,Y]	false
[a,X|Y]	[Z,a]	Z=a, X=a, Y=[]

2.11.2 Операции со списками
2.11.2.1 Принадлежность элемента списку

Пример 4:

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

При использовании можно задавать один или 2 аргумента.
Есть встроенный предикат member.

2.11.2.2 Соединение двух списков (аналог append)

Пример 5:

```
append1([],L2,L2).
append1([Head|Tail],L2,[Head|Tail1]):-append1(Tail,L2,Tail1).
```

Можно использовать предикат для следующих целей:

- слияние двух списков;
- получение всех возможных разбиений списка;
- поиск подсписков до и после определенного элемента;
- поиск элементов списка, стоящих перед и после определенного элемента;
- удаление части списка, начиная с некоторого элемента;
- удаление части списка, предшествующей некоторому элементу.

Есть встроенный предикат append.

Вопрос в Пролог-системе	Ответ Пролог-системы
append1([1,2],[3],L).	L=[1,2,3].
append1(L1,L2,[1,2,3]).	L1=[ ],L2=[1,2,3]; L1=[1],L2=[2,3]; L1=[1,2],L2=[3]; L1=[1,2,3],L2=[ ]; false
append1(Before,[3|After],[1,2,3,4,5]).	Before=[1,2],After=[4,5]; false
append1(_,[Before,3,After|_],[1,2,3,4,5]).	Before=2,After=4; false
append1(L1,[3|_],[1,2,3,4,5]).	L1=[1,2]; false
append1(_,[3|L2],[1,2,3,4,5]).	L2=[4,5]; false

2.11.2.3 Добавление и удаление элемента из списка

Пример 6 (добавление в начало списка, удаление первого вхождения заданного элемента):

```
insert(X,L,[X|L]).
select1(_,[],[]).
select1(X,[X|Tail],Tail).
select1(X,[Y|Tail],[Y|Tail1]):-select1(X,Tail,Tail1).
```

Предикат select1 можно использовать также для добавления элемента в список. Есть встроенный предикат select.

Пример 7 (удаление всех вхождений заданного элемента):

```
delete1([],_,[]):-!.
delete1([X|Tail],X,Tail1):-delete1(Tail,X,Tail1),!.
delete1([Y|Tail],X,[Y|Tail1]):-delete1(Tail,X,Tail1).
```

Есть встроенный предикат delete.

2.11.2.4 Деление списка на два списка по разделителю

Пример 8:

Деление списка на две части, используя разделитель M (если элемент исходного списка меньше разделителя, то он помещается в первый результирующий список, иначе – во второй результирующий список).

```
split(M,[Head|Tail],[Head|Tail1],L2):-
                                    Head<M,!,split(M,Tail,Tail1,L2).
split(M,[Head|Tail],L1,[Head|Tail2]):-split(M,Tail,L1,Tail2).
split(_,[],[],[]).
```

2.11.2.5 Подсчет количества элементов в списке

Пример 9:

```
count([],0).
count([_|Tail],N):-count(Tail,N1),N is N1+1.
```

Есть встроенный предикат length(L,N) – подсчет количества элементов N в списке L.

reverse(L1,L2) – обращение любого из списков-аргументов.

2.11.3 Сортировка списков (по неубыванию)
2.11.3.1 Сортировка вставкой

Пример 10:

Добавляем голову списка в нужное место отсортированного хвоста.

```
in_sort([],[]).
in_sort([X|Tail],Sort_list):-
                    in_sort(Tail,Sort_Tail),add(X,Sort_Tail,Sort_list).
add(X,[Y|Sort_list],[Y|Sort_list1]):-
                     X@>Y,!,add(X,Sort_list,Sort_list1).
add(X,Sort_list,[X|Sort_list]).
```

2.11.3.2 Пузырьковая сортировка

Пример 11:

Меняем местами соседние элементы до тех пор, пока есть неверно упорядоченные пары.

```
pu_sort(L,Sort_list):-swap(L,L1),!,pu_sort(L1,Sort_list).
pu_sort(L,L).
swap([X,Y|Tail],[Y,X|Tail]):-X@>Y.
swap([X|Tail],[X|Tail1]):-swap(Tail,Tail1).
```