2.7 Внелогические предикаты управления поиском решений

Поиск решений Пролог-системой –полный перебор. Это может стать источником неэффективности программы.

2.7.1 Откат после неудач, предикат fail

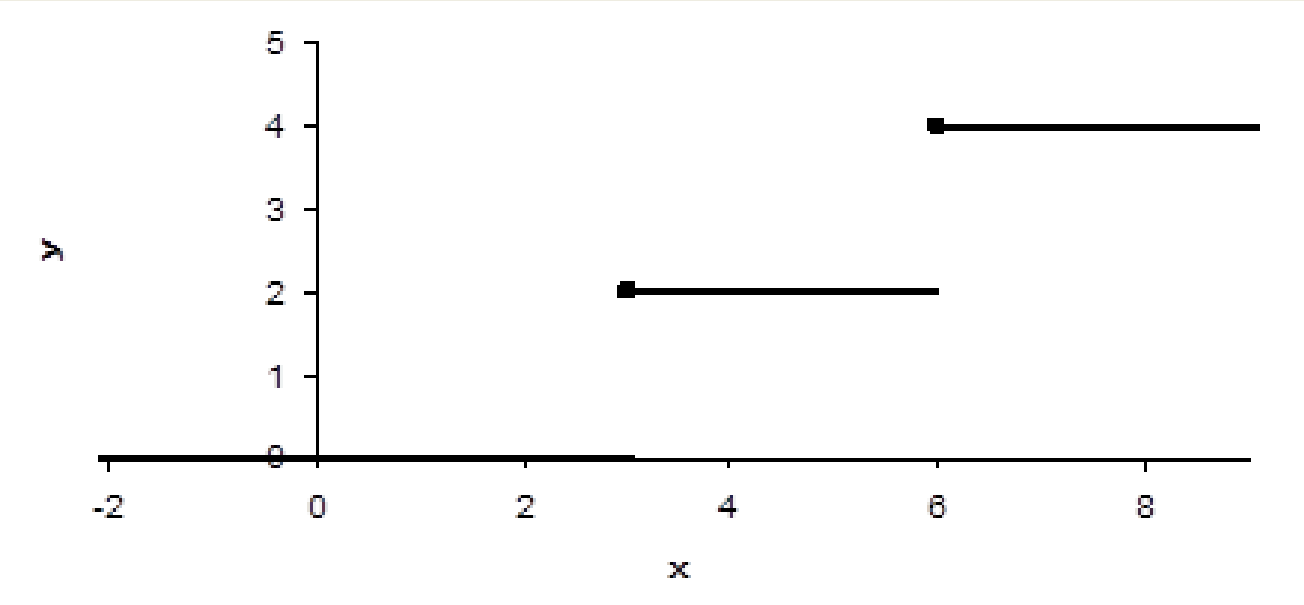
Предикат fail всегда неудачен, поэтому инициализирует откат в точки поиска альтернативных решений.

Пример:
Определим двуместный предикат сотрудник, который связывает ФИО и возраст сотрудника. Определим предикат, который выводит всех сотрудников до 40 лет.

```
кандидат(a,25).
кандидат(b,35).
кандидат(c,45).
кандидат(d,20).
кандидат(e,19).
кандидат(f,55).
до_40:-кандидат(X,Y),Y<40,writeln(X),fail.
до_40.
```

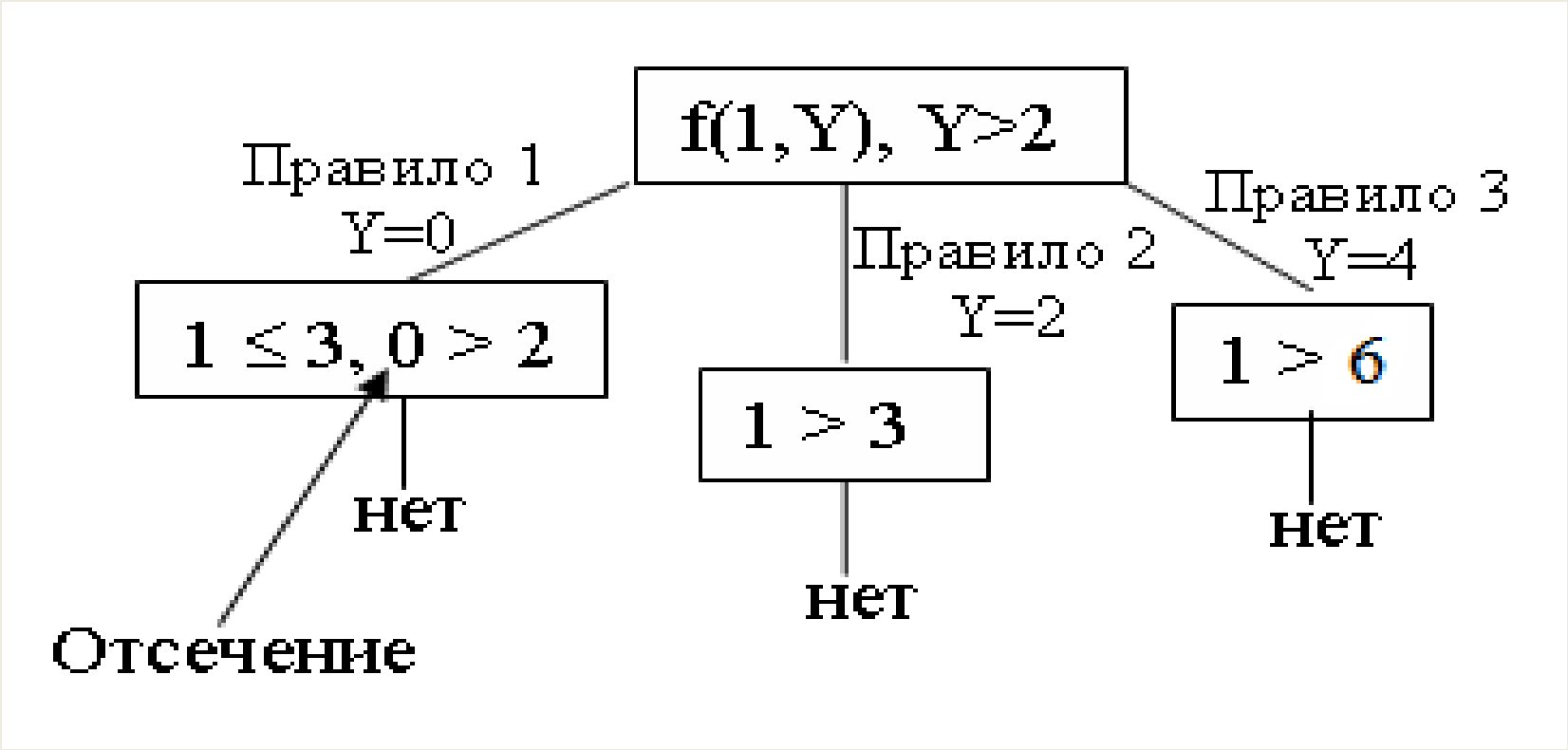
2.7.2 Ограничение перебора – отсечение

Пример1:



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

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):
f(1,Y), Y>2.
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).

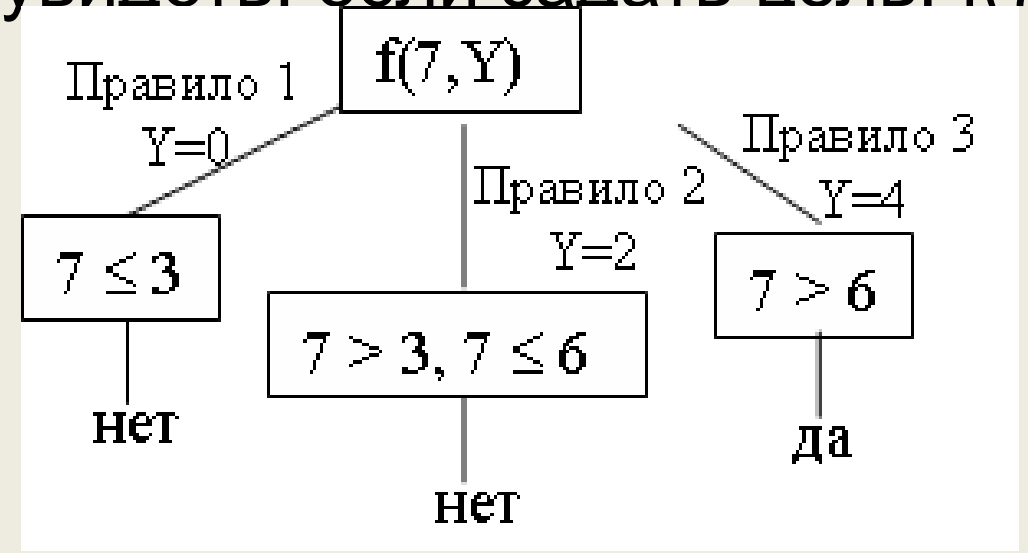
Добавим отсечения в наше определение функции:

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

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

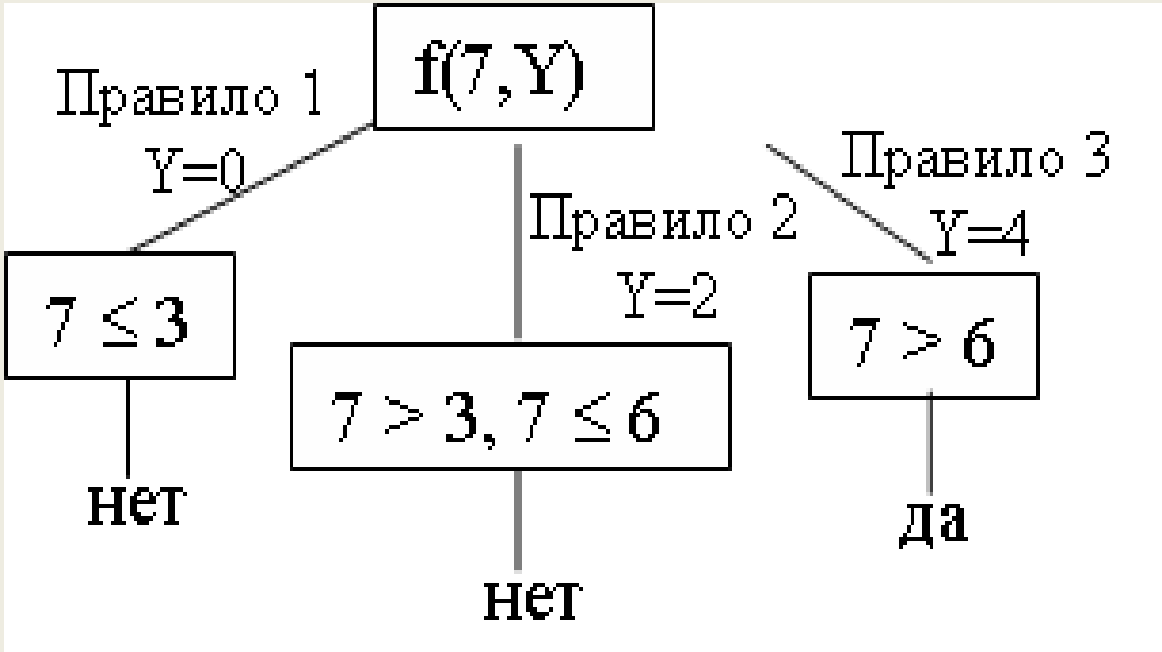
f(X,4):-X>6.

Еще один источник неэффективности можно увидеть. если задать цель: f(7,Y).



Еще один источник неэффективности можно увидеть, если задать цель:

f(7,Y).



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(X,4).
```

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:

$$H : -B_1, \ldots, B_k, !, \ldots, B_n$$

Если цели $B_1, \ldots, B_k$ успешны, то это решение замораживается, и другие альтернативы для этого решения больше не рассматриваются (отсекается правая часть дерева решений, которая находится выше $B_1, \ldots, B_k$).

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.

Пример2: Вычисление суммы ряда натуральных чисел 1, 2, ... N.

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

При цели sum(2,X), находится решение X=3, а после продолжения поиска решений уходит в бесконечную рекурсию.

Исправим:

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

При цели sum(-3,X) уходит в бесконечную рекурсию.

Исправим:

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

2.8 Циклы, управляемые отказом

Имеется встроенный предикат без аргументов repeat, который всегда успешен.

repeat.

repeat:-repeat.

Реализация цикла «до тех пор, пока»:

<голова правила>:- repeat,

<тело цикла>,

<условие выхода>,!.

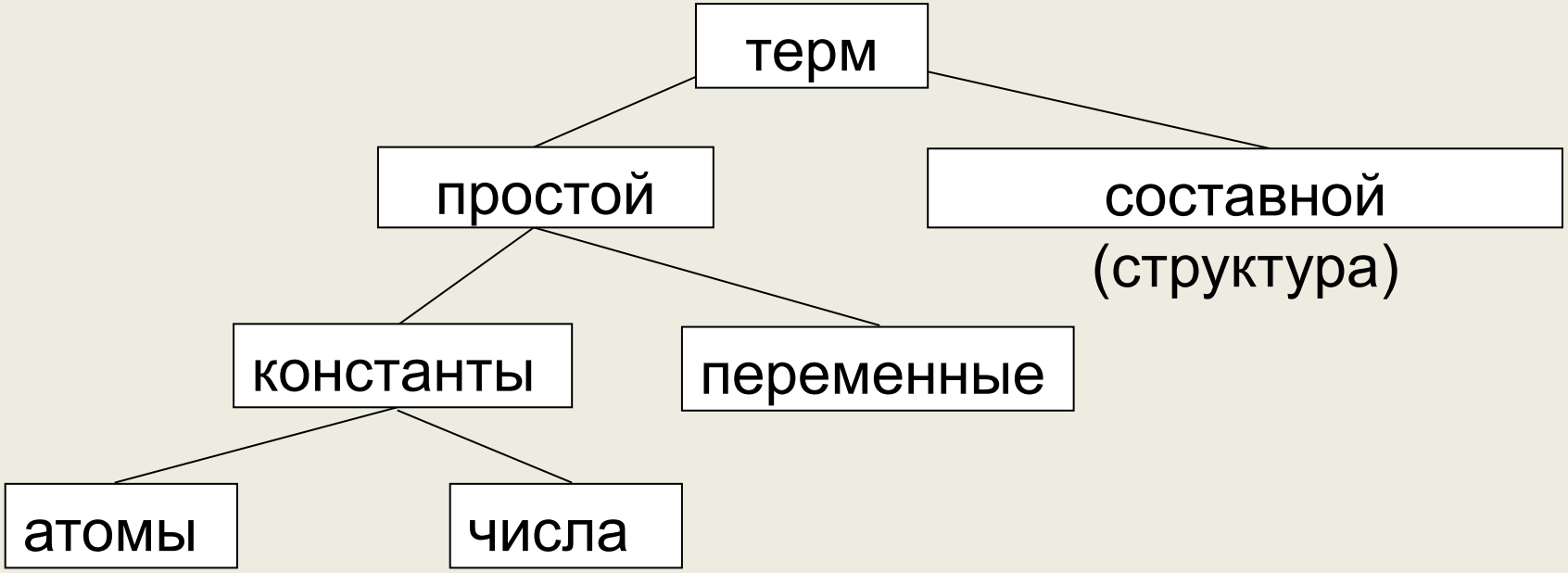
Пример:

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 слово ‘), эхо.
эхо:-repeat, read(S), writeln(S), проверка(S),!.
проверка(stop).
проверка(_):-fail.
```

2.9 Структуры

Данные в Прологе:



Составной терм:

$$f(t_1,t_2,...,t_n),$$

где f - имя n-арного функтора, t_i - аргументы.

дата(1,май,1998).

Каждый функтор и предикат определяется двумя параметрами: именем и арностью.

Арифметические выражения – составные термы, записанные в инфиксной форме. Но их можно также записать в префиксной форме.

Проверка типа терма:

var(Term) - свободная переменная

nonvar(Term) - несвободная (конкретизированная) переменная

number(Term) - целое или вещественное число

atom(Term) – атом

atomic(Term) - атом или число

ground(Term) -терм не содержит свободных переменных

2.10 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.

Например: [a,1,b,asd], [[1,2],[],x].

Пустой список: [].

2.10.1 Голова и хвост списка

Голова списка – первый элемент.

Хвост списка – часть списка без первого элемента

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

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

Пример: Сопоставление списков

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

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

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]	
[1,2,3]	[X,Y]	
[a,X|Y]	[Z,a]	

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]	
[a,X|Y]	[Z,a]	

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]	нет решений
[a,X|Y]	[Z,a]	

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]	нет решений
[a,X|Y]	[Z,a]	Z=a, X=a, Y=[ ]

2.10.2 Операции со списками

2.10.2.1 Принадлежность элемента списку

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

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

2.10.2.2 Соединение двух списков (аналог append)

append1([],L,L).

append1([H|T],L,[H|T1]):-append1(T,L,T1).

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.

2.10.2.3 Добавление и удаление элемента из списка

```
insert(X,L,[X|L]).
delete1([],_,[]):-!.
delete1([X|T],X,T):-!.
delete1([Y|T],X,[Y|T1]):-delete1(T,X,T1).
```

delete1 удаляет первое вхождение элемента. Встроенный предикат delete удаляет все вхождения элемента.

Одно вхождение элемента выполняет встроенный предикат:

select(<элемент>,<список1>,<список2>).

Его можно использовать также для добавления элемента в список

2.10.2.4 Деление списка на два списка по разделителю

Напишем предикат split, который будет делить список на две части, используя разделитель M. Определим предикат следующим образом: если элемент исходного списка меньше разделителя, то он помещается в первый результирующий список, иначе – во второй результирующий список.

```
split(M,[H1|T1],[H1|T],L2):-
H1@<M,!,split(M,T1,T,L2).

split(M,[H1|T1],L1,[H1|T2]):-split(M,T1,L1,T2).

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

2.10.2.5 Подсчет количества элементов в списке

```
count([],0).
count([_|T],N):-count(T,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

2.10.3 Быстрая сортировка списков (по неубыванию)

```
q_sort([H|T],Sort_list):-split(H,T,Less,More),
                         q_sort(Less,Sort_less),
                         q_sort(More,Sort_more),
                        append(Sort_less,[H|Sort_more], Sort_list).
q_sort([],[]).
```

Временная сложность алгоритма быстрой сортировки примерно $n \cdot \log n$

Есть встроенные предикаты: sort, msort