Функциональное и логическое программирование

Лекция 4

План лекции

1. Семантика Пролога
2. Внелогические предикаты управления поиском решений

2.8 Семантика Пролога

2.8.1 Порядок предложений и целей

Программу на Прологе можно понимать по-разному: с декларативной и процедурной точки зрения.

Декларативная семантика касается только отношений, описанных в программе, и определяет, что является ли поставленная цель достижимой и если да, то определяются значения переменных, при которых эта цель достижима.

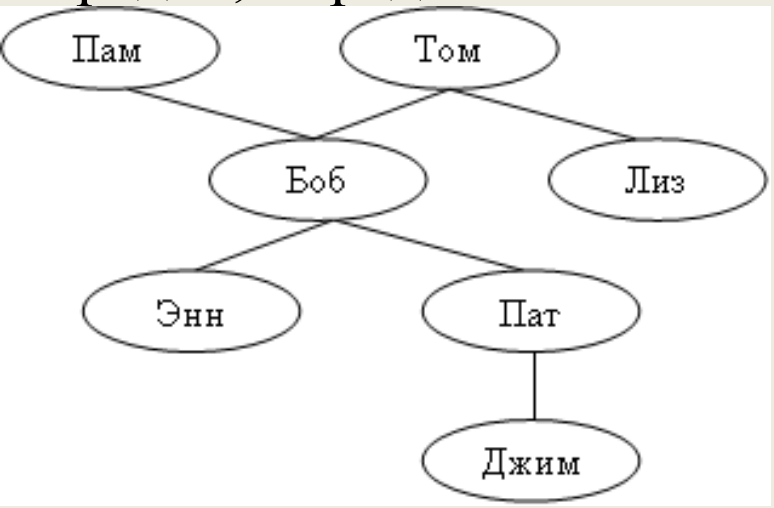
Процедурная семантика определяет, как должен быть получен результат, т.е. как Пролог-система отвечает на вопросы.

Для правила вида P:–Q,R. декларативная семантика определяет, что из истинности Q и R следует истинность P, а процедурная семантика определяет, что для решения P следует сначала решить Q, а потом R (важен порядок обработки целей).

P:- P. верно с декларативной т. зр.
бесконечный цикл с процед. т. зр

Пример 1:

Изменим порядок следования правил и подцелей в предикате предок, определенном на прошлой лекции:

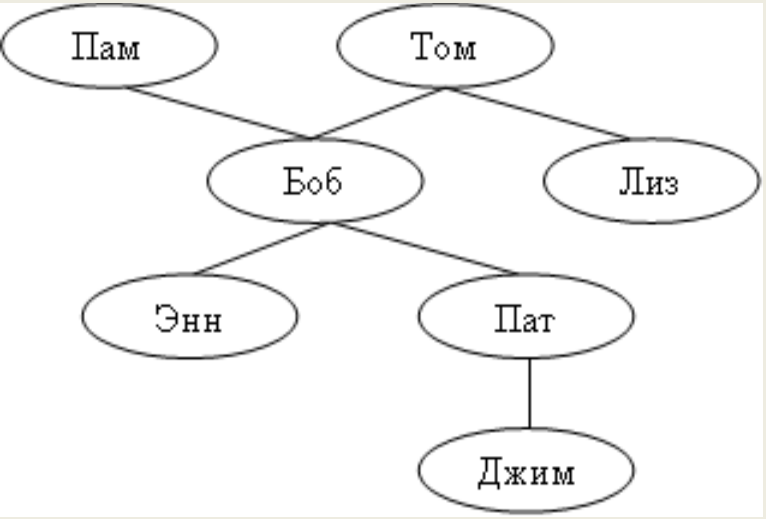


родитель(пам,боб).
родитель(том,боб).
родитель(том,лиз).
родитель(боб,энн).
родитель(боб,пат).
родитель(пат,джим).

.

предок(X,Y):-родитель(X,Y).
предок(X,Y):-родитель(X,Z),предок(Z,Y).

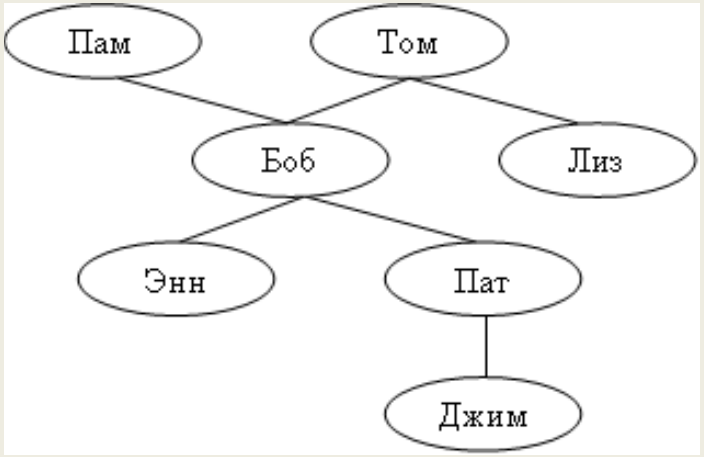
```
предок1(X,Y):-родитель(X,Z),предок1(Z,Y).
предок1(X,Y):-родитель(X,Y).

предок2(X,Y):-родитель(X,Y).
предок2(X,Y):-предок2(Z,Y),родитель(X,Z).

предок3(X,Y):-предок3(Z,Y),родитель(X,Z).
предок3(X,Y):-родитель(X,Y).
```



родитель(пам,боб).
родитель(том,боб).
родитель(том,лиз).
родитель(боб,энн).
родитель(боб,пат).
родитель(пат,джим).
предок(X,Y):-родитель(X,Y).
предок(X,Y):-родитель(X,Z),предок(Z,Y).

Посмотрим, как Пролог будет искать решения для разных вариантов определения предиката предок
?- предок(том,боб).

```
?- trace, предок(том,боб).
   Call: (11) предок(том, боб) ? creep
   Call: (12) родитель(том, боб) ? creep
   Exit: (12) родитель(том, боб) ? creep
   Exit: (11) предок(том, боб) ? creep
true .
```



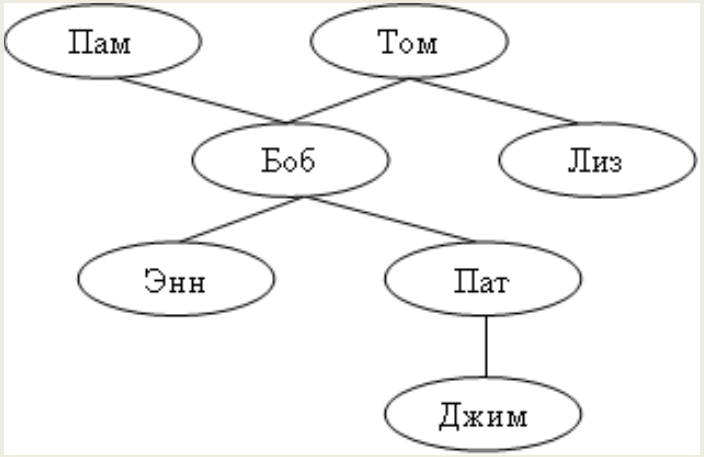
предок1(X,Y):-родитель(X,Z),предок1(Z,Y).
предок1(X,Y):-родитель(X,Y).
?- предок1(том,боб).

Ответ найден, но поиск долго шел.

```
[trace]  ?- предок1(том,боб).
   Call: (10) предок1(том, боб) ? creep
   Call: (11) родитель(том, _5450) ? creep
   Exit: (11) родитель(том, боб) ? creep
   Call: (11) предок1(боб, боб) ? creep
   Call: (12) родитель(боб, _5582) ? creep
   Exit: (12) родитель(боб, энн) ? creep
   Call: (12) предок1(энн, боб) ? creep
   Call: (13) родитель(энн, _5714) ? creep
   Fail: (13) родитель(энн, _5758) ? creep
   Redo: (12) предок1(энн, боб) ? creep
   Call: (13) родитель(энн, боб) ? creep
   Fail: (13) родитель(энн, боб) ? creep
   Fail: (12) предок1(энн, боб) ? creep
   Redo: (12) родитель(боб, _5978) ? creep
   Exit: (12) родитель(боб, пат) ? creep
```

. . . . .

```
   Redo: (11) предок1(лиз, боб) ? creep
   Call: (12) родитель(лиз, боб) ? creep
   Fail: (12) родитель(лиз, боб) ? creep
   Fail: (11) предок1(лиз, боб) ? creep
   Redo: (10) предок1(том, боб) ? creep
   Call: (11) родитель(том, боб) ? creep
   Exit: (11) родитель(том, боб) ? creep
   Exit: (10) предок1(том, боб) ? creep
true.
```

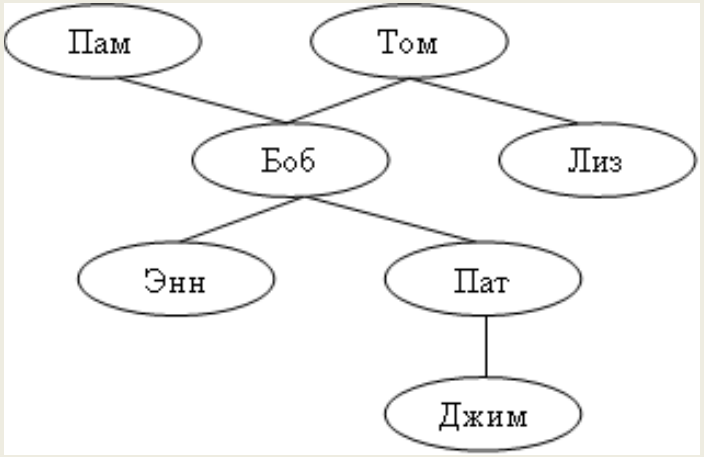


предок2(X,Y):-родитель(X,Y).
предок2(X,Y):-предок2(Z,Y),родитель(X,Z).
?- предок2(лиз,боб).

Переполнение стека рекурсивными вызовами

повтор

```
[trace]  ?- предок2(лиз,боб).
   Call: (10) предок2(лиз, боб) ? creep
   Call: (11) родитель(лиз, боб) ? creep
   Fail: (11) родитель(лиз, боб) ? creep
   Redo: (10) предок2(лиз, боб) ? creep
   Call: (11) предок2(_8682, боб) ? creep
   Call: (12) родитель(_8726, боб) ? creep
   Exit: (12) родитель(пам, боб) ? creep
   Exit: (11) предок2(пам, боб) ? creep
   Call: (11) родитель(лиз, пам) ? creep
   Fail: (11) родитель(лиз, пам) ? creep
   Redo: (12) родитель(_8946, боб) ? creep
   Exit: (12) родитель(том, боб) ? creep
   Exit: (11) предок2(том, боб) ? creep
   Call: (11) родитель(лиз, том) ? creep
   Fail: (11) родитель(лиз, том) ? creep
   Redo: (11) предок2(_9166, боб) ? creep
   Call: (12) предок2(_9210, боб) ? creep
   Call: (13) родитель(_9254, боб) ?
```



предок3(X,Y):-предок3(Z,Y),родитель(X,Z).
предок3(X,Y):-родитель(X,Y).
?- предок3(том,боб).

```
?- trace,предок3(том,боб).
   Call: (11) предок3(том, боб) ? creep
   Call: (12) предок3(_990, боб) ? creep
   Call: (13) предок3(_1034, боб) ? creep
   Call: (14) предок3(_1078, боб) ? creep
   Call: (15) предок3(_1122, боб) ?
```

. . . .

Ответ не будет получен

Таким образом, правильные с декларативной точки зрения программы могут работать неправильно. При составлении правил следует руководствоваться следующим:

- более простое правило следует ставить на первое место;
- по возможности избегать левой рекурсии.

2.8.2 Пример декларативного создания программы

Пример 2 (задача об обезьяне и банане):

Возле двери комнаты стоит обезьяна. В середине комнаты к потолку подвешен банан. Обезьяна голодна и хочет съесть банан, но не может до него дотянуться, находясь на полу. Около окна этой комнаты находиться ящик, которым обезьяна может воспользоваться.

Обезьяна может предпринимать следующие действия: ходить по полу, залазить на ящик, двигать ящик (если обезьяна находится возле ящика), схватить банан (если обезьяна находится на ящике под бананом).

Может ли обезьяна добраться до банана?

Мир обезьяны всегда находится в некотором состоянии, которое может изменяться со временем.

Состояние обезьяньего мира определяется четырьмя компонентами: горизонтальная позиция обезьяны, вертикальная позиция обезьяны (на ящике или на полу), позиция ящика, наличие у обезьяны банана (есть или нет). Объединим эти компоненты в структуру с функтором «состояние»:

Например, состояние(уокна, наполу, вцентре, нет).

Задачу можно рассматривать как игру для одного игрока – обезьяны. Формализуем правила этой игры: начальное состояние, цель игры и ходы игры.

Начальное состояние игры:

состояние(двери, полу, окна, кег)

Цель игры:

состояние(_, _, _, да)

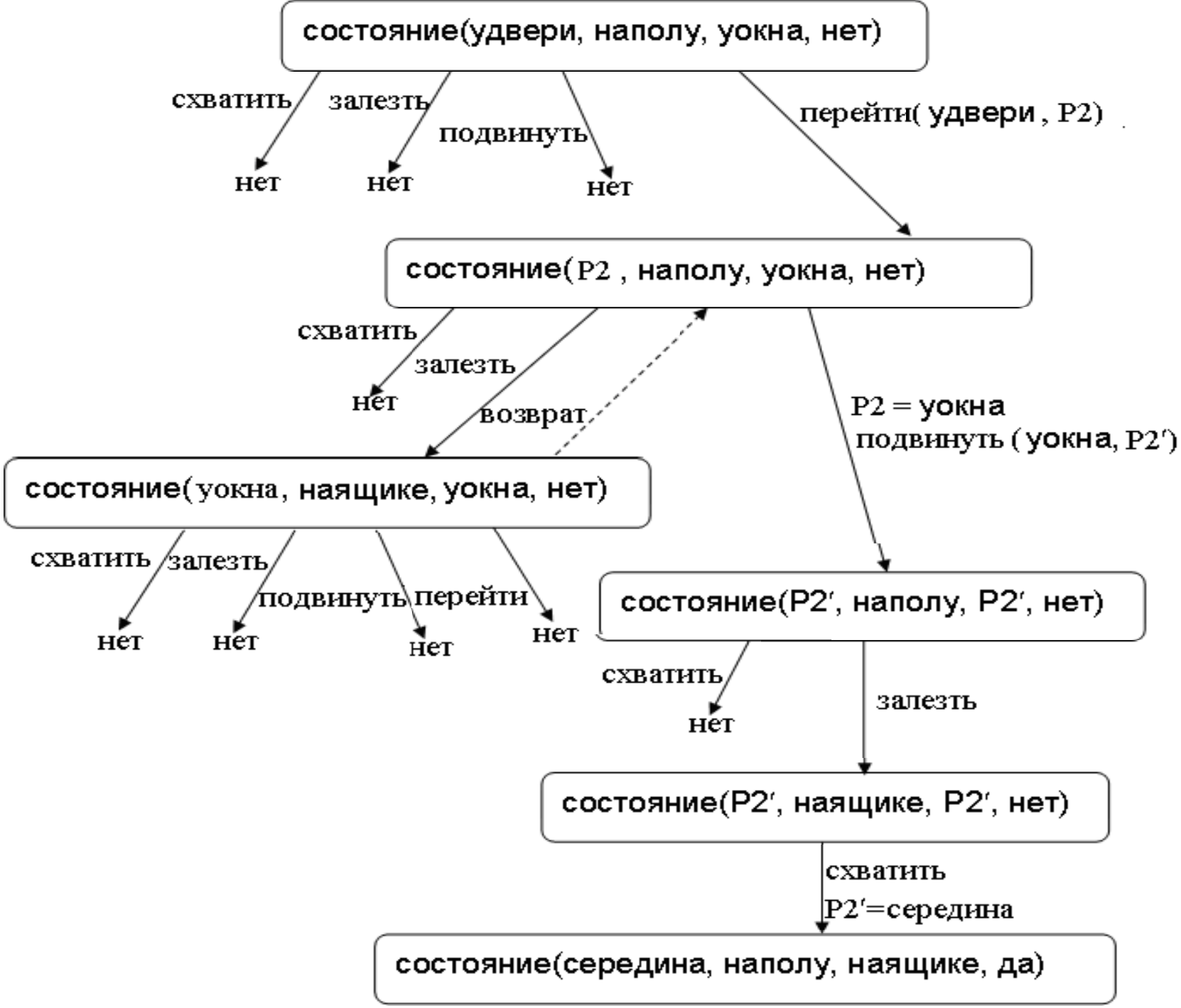
Ходы обезьяны: перейти в другое место, подвинуть ящик, залезть на ящик, схватить банан.

В результате хода меняется состояние обезьяньего мира, система переходит из одного состояния в другое. Для изменений определим предикат ход(состояние, действие, состояние).

Действия обезьяны: схватить, залезть, подвинуть, перейти.

```
goal:-может_достать(состояние(удвери,наполу,уокна,нет)).
ход(состояние(середина,наящике,середина,нет),
    схватить,
    состояние(середина,наящике,середина,да)).
ход(состояние(Р,наполу,Р,нет),
    залезть,
    состояние(Р,наящике,Р,нет)).
ход(состояние(Р1,наполу,Р1,нет),
    подвинуть(Р1,Р2),
    состояние(Р2,наполу,Р2,нет)).
ход(состояние(Р1,наполу,Р,нет),
    перейти(Р1,Р2),
    состояние(Р2,наполу,Р,нет)).
может_достать(состояние(_,_,_,да)).
может_достать(S1):-ход(S1,_,S2),
                   может_достать(S2).
```

```
[trace]  ?- goal.
   Call: (10) goal ? creep
   Call: (11) может_достать(состояние(удвери, наполу, уокна, нет)) ? creep
   Call: (12) ход(состояние(удвери, наполу, уокна, нет), _20778, _20780) ? creep
   Exit: (12) ход(состояние(удвери, наполу, уокна, нет), перейти(удвери, _20770), состояние(_20770, наполу, уокна, не
   Call: (12) может_достать(состояние(_20770, наполу, уокна, нет)) ? creep
   Call: (13) ход(состояние(_20770, наполу, уокна, нет), _20926, _20928) ? creep
   Exit: (13) ход(состояние(уокна, наполу, уокна, нет), залезть, состояние(уокна, наящике, уокна, нет)) ? creep
   Call: (13) может_достать(состояние(уокна, наящике, уокна, нет)) ? creep
   Call: (14) ход(состояние(уокна, наящике, уокна, нет), _21068, _21070) ? creep
   Fail: (14) ход(состояние(уокна, наящике, уокна, нет), _21112, _21114) ? creep
   Fail: (13) может_достать(состояние(уокна, наящике, уокна, нет)) ? creep
   Redo: (13) ход(состояние(_20770, наполу, уокна, нет), _21200, _21202) ? creep
   Exit: (13) ход(состояние(уокна, наполу, уокна, нет), подвинуть(уокна, _21192), состояние(_21192, наполу, _21192, н
   Call: (13) может_достать(состояние(_21192, наполу, _21192, нет)) ? creep
   Call: (14) ход(состояние(_21192, наполу, _21192, нет), _21348, _21350) ? creep
   Exit: (14) ход(состояние(_21192, наполу, _21192, нет), залезть, состояние(_21192, наящике, _21192, нет)) ? creep
   Call: (14) может_достать(состояние(_21192, наящике, _21192, нет)) ? creep
   Call: (15) ход(состояние(_21192, наящике, _21192, нет), _21490, _21492) ? creep
   Exit: (15) ход(состояние(середина, наящике, середина, нет), схватить, состояние(середина, наящике, середина, да))
   Call: (15) может_достать(состояние(середина, наящике, середина, да)) ? creep
   Exit: (15) может_достать(состояние(середина, наящике, середина, да)) ? creep
   Exit: (14) может_достать(состояние(середина, наящике, середина, нет)) ? creep
   Exit: (13) может_достать(состояние(середина, наполу, середина, нет)) ? creep
   Exit: (12) может_достать(состояние(уокна, наполу, уокна, нет)) ? creep
   Exit: (11) может_достать(состояние(удвери, наполу, уокна, нет)) ? creep
   Exit: (10) goal ? creep
true
```


состояние(удвери, наполу, уокна, нет)
схватить
залезть
подвинуть
перейти(удвери , P2)
нет
нет
нет
состояние(P2 , наполу, уокна, нет)
схватить
залезть
возврат
нет
P2 = уокна
подвинуть (уокна, P2′)
состояние(уокна , наящике, уокна, нет)
схватить
залезть
подвинуть
перейти
нет
нет
нет
нет
состояние(P2′, наполу, P2′, нет)
схватить
нет
залезть
состояние(P2′, наящике, P2′, нет)
схватить
P2′=середина
состояние(середина, наполу, наящике, да)

Для того чтобы ответить на вопрос, Пролог-системе пришлось сделать лишь один возврат. Причина такой эффективности – правильно выбранный порядок следования предложений, описывающих ходы.

Однако возможен и другой порядок, когда обезьяна будет ходить туда-сюда, не касаясь ящика, или бесцельно двигать ящик в разные стороны. Порядок предложений и целей важен в программе.

.

2.9 Внелогические предикаты управления поиском решений

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

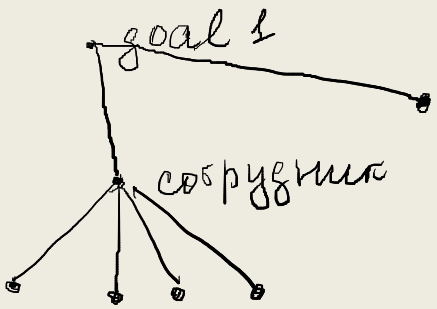
2.9.1 Откат после неудач, предикат fail

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(ф,25).
сотрудник(ы,42).
сотрудник(в,41).
сотрудник(а,30).
сотрудник(п,27).
сотрудник(р,55).
goal1:-сотрудник(X,A),A<40,writeln(X),fail.
```

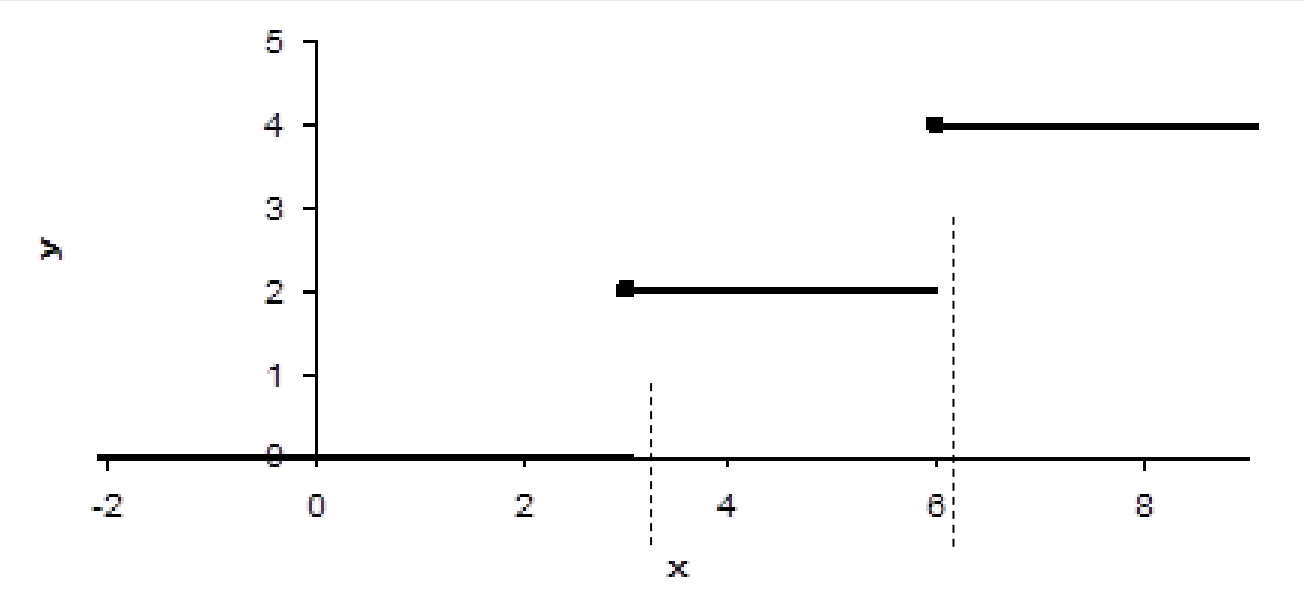
```
?- goal1.
ф
а
п
false.
```

```
сотрудник(ф,25).
сотрудник(ы,42).
сотрудник(в,41).
сотрудник(а,30).
сотрудник(п,27).
сотрудник(р,55).
goal1:-сотрудник(X,A),A<40,writeln(X),fail.
goal1.
```

```
?- goal1.
ф
а
п
true.
```

2.9.2 Ограничение перебора – отсечение

Пример 2:



Аналитическое задание функции:

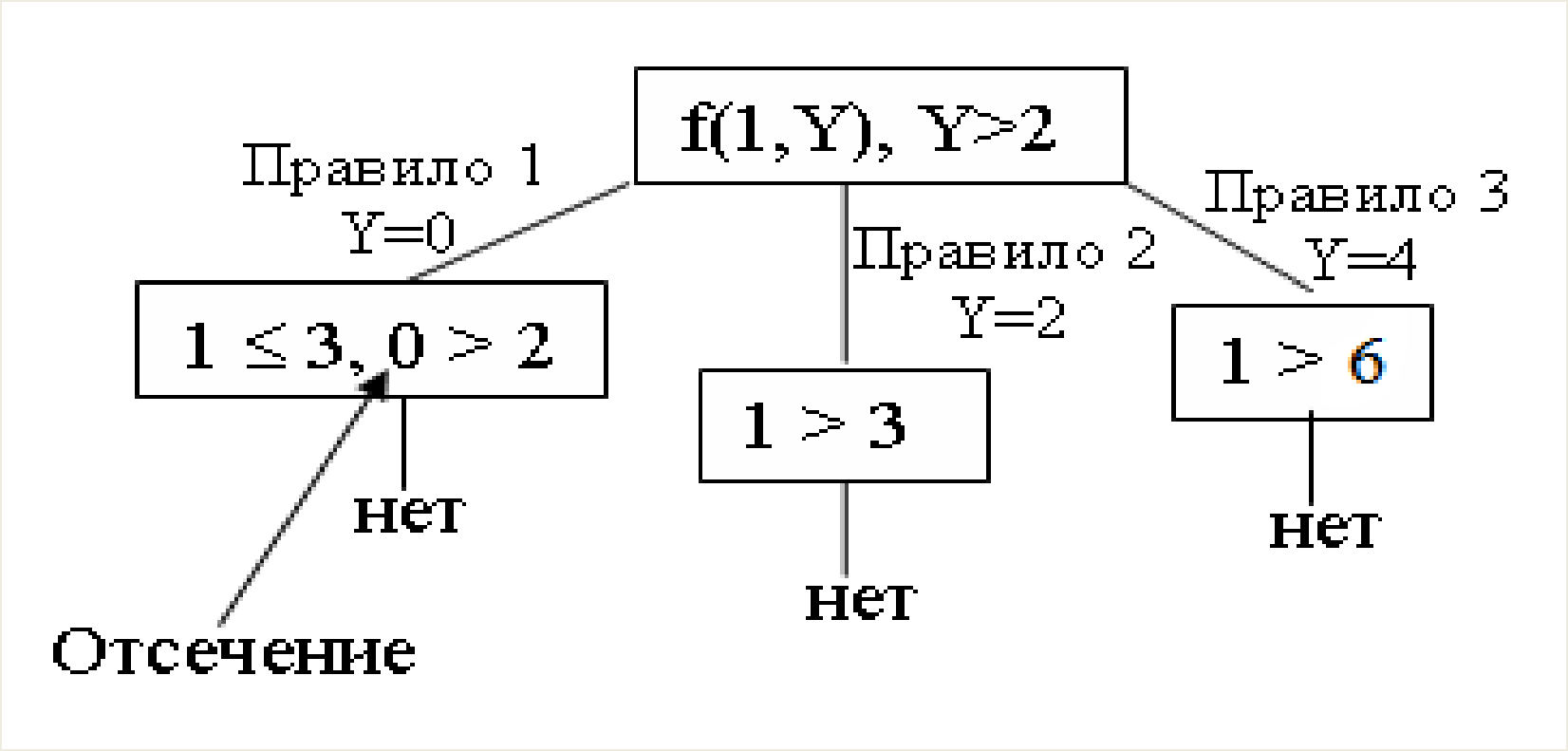
$$f(x) = \begin{cases} 0, & x \le 3 \\ 2, & 3 < x \le 6 \\ 4, & x > 6 \end{cases}$$

Программа на Прологе:

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

?- f(1,Y), Y>2.

```
[trace]  ?- f(1,Y),Y>2.
   Call: (11) f(1, _20924) ? 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, _20924) ? creep
   Call: (12) 1>3 ? creep
   Fail: (12) 1>3 ? creep
   Redo: (11) f(1, _20924) ? creep
   Call: (12) 1>6 ? creep
   Fail: (12) 1>6 ? creep
   Fail: (11) f(1, _20924) ? creep
false.
```



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.
```

?- f(1,Y), Y>2.

```
[trace]  ?- f(1,Y),Y>2.
   Call: (11) f(1, _22630) ? 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.
```

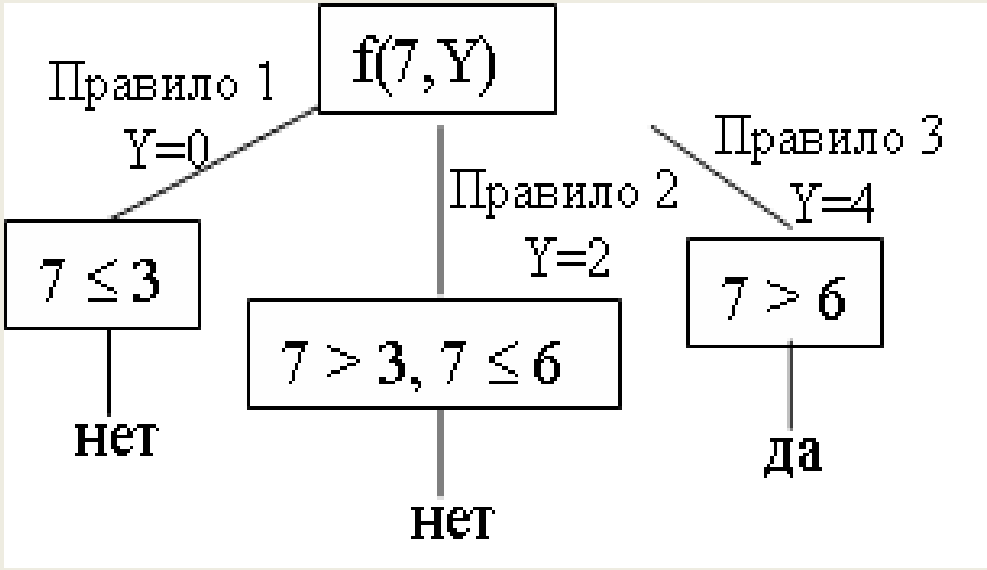
Теперь при поиске решения альтернативные ветви, соответствующие правилам 1 и 2, порождены не будут. Программа станет эффективнее.

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.

Можно увидеть еще один источник неэффективности.

?- f(7,Y).

```
[trace]  ?- f(7,Y).
   Call: (10) f(7, _24024) ? creep
   Call: (11) 7=<3 ? creep
   Fail: (11) 7=<3 ? creep
   Redo: (10) f(7, _24024) ? 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, _24024) ? creep
   Call: (11) 7>6 ? creep
   Exit: (11) 7>6 ? creep
   Exit: (10) f(7, 4) ? creep
Y = 4.
```



Новое определение функции:

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

```
[trace]  ?- f(7,Y).
   Call: (10) f(7, _25744) ? creep
   Call: (11) 7=<3 ? creep
   Fail: (11) 7=<3 ? creep
   Redo: (10) f(7, _25744) ? creep
   Call: (11) 7=<6 ? creep
   Fail: (11) 7=<6 ? creep
   Redo: (10) f(7, _25744) ? creep
   Exit: (10) f(7, 4) ? creep
Y = 4.
```

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

?- f(2,Y).

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

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

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:

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

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.

Пример3:

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

$1 + 2 + \ldots + N$

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

?-sum(2,X).

```
?- s(2,X).
X = 3 ;
ERROR: Stack limit (1.0Gb) exceeded
ERROR:   Stack sizes: local: 1.0Gb, global: 0.1Mb, trail: 1Kb
ERROR:   Stack depth: 11,178,061, last-call: 0%, Choice points: 3
ERROR:   Possible non-terminating recursion:
ERROR:     [11,178,061] user:s(-11178049, _35420)
ERROR:     [11,178,060] user:s(-11178048, _35440)
```

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

```
?- s(2,X).
X = 3.
```

?-sum(-3,X).

```
?- s(-3,X).
ERROR: Stack limit (1.0Gb) exceeded
ERROR:   Stack sizes: local: 1.0Gb, global: 27Kb, trail: 0Kb
ERROR:   Stack depth: 11,180,834, last-call: 0%, Choice points: 3
ERROR:   Possible non-terminating recursion:
ERROR:     [11,180,834] user:s(-11180827, _7042)
ERROR:     [11,180,833] user:s(-11180826, _7062)
```

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

```
?- s(-3,X).
false.
```

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».

```
goal2:-writeln('Слово-?'),дубль.
дубль:-repeat,
       read(S),
       проверка(S),!.
проверка(stop).
проверка(S):-writeln(S),fail.
```

```
?- goal2.
Слово-?
|: asd.
asd
|: rtyu.
rtyu
|: stop.

true.
```