

Функциональное и логическое программирование

Лекция 5

2.8 Семантика Пролога

2.8.1 Порядок предложений и целей

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

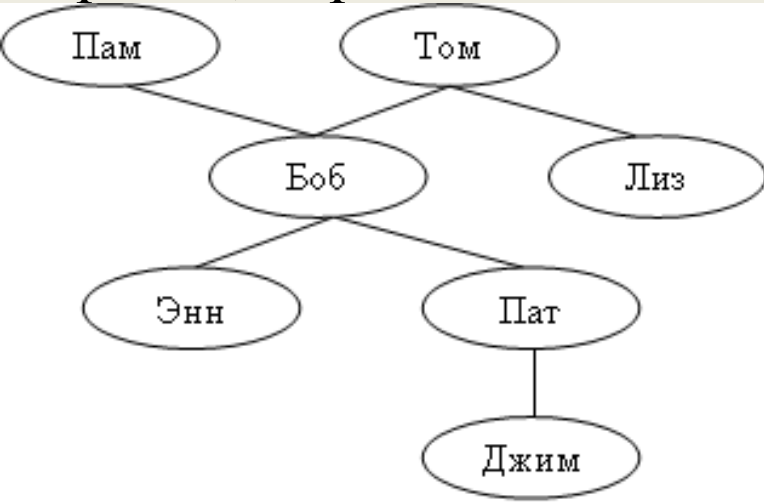
Декларативная семантика касается только отношений, описанных в программе, и определяет, что является ли поставленная цель достижимой и если да, то определяются значения переменных, при которых эта цель достижима.

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).

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).

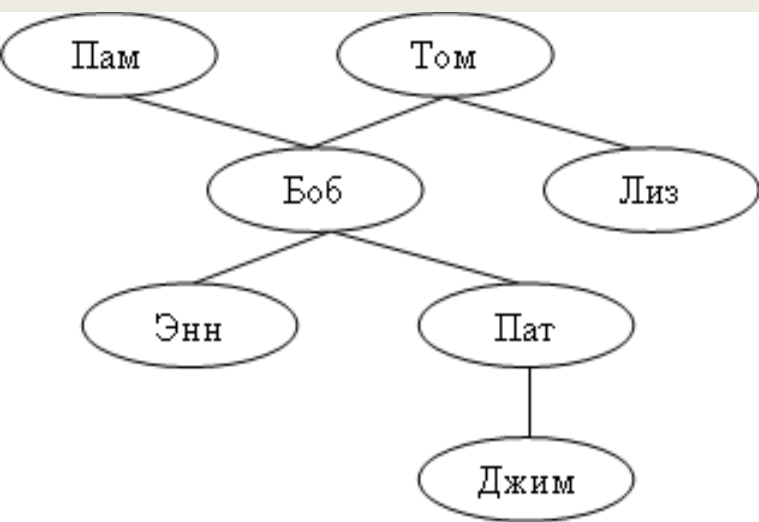
предок1(X,Y):-родитель(X,Y).

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).

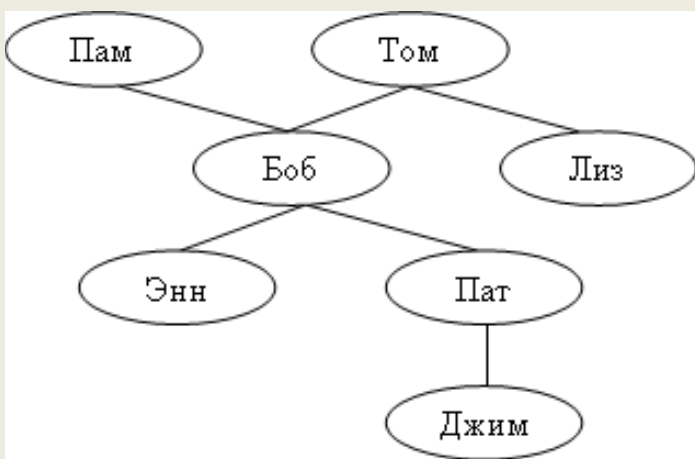
Call: (10) предок(том, боб) ? creep

Call: (11) родитель(том, боб) ? creep

Exit: (11) родитель(том, боб) ? creep

Exit: (10) предок(том, боб) ? creep

true .



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

предок1(X,Y):-родитель(X,Y).

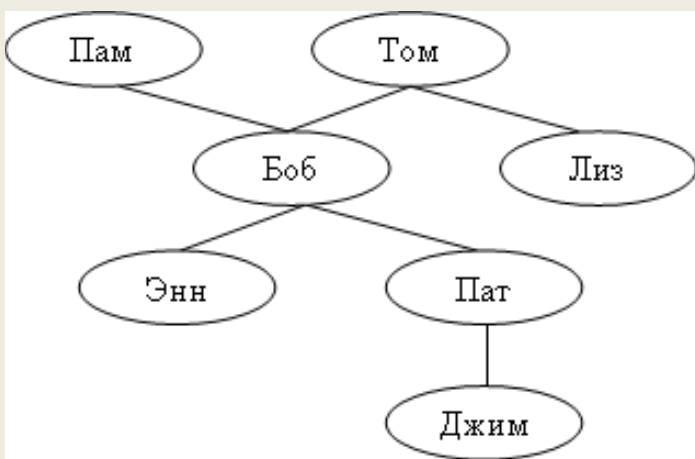
?- предок1(том,боб).

```

Call: (11) родитель(том, _33600) ? creep
Exit: (11) родитель(том, боб) ? creep
Call: (11) предок1(боб, боб) ? creep
Call: (12) родитель(боб, _36032) ? creep
Exit: (12) родитель(боб, энн) ? creep
Call: (12) предок1(энн, боб) ? creep
Call: (13) родитель(энн, _38464) ? creep
Fail: (13) родитель(энн, _38464) ? 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, _36032) ? creep
Exit: (12) родитель(боб, пат) ? creep
Call: (12) предок1(пат, боб) ? creep
Call: (13) родитель(пат, _45756) ? creep
Exit: (13) родитель(пат, джим) ? creep
Call: (13) предок1(джим, боб) ? creep
Call: (14) родитель(джим, _48188) ? creep
Fail: (14) родитель(джим, _48188) ? creep
  
```

```

Fail: (13) предок1(джим, боб) ? creep
Redo: (12) предок1(пат, боб) ? creep
Call: (13) родитель(пат, боб) ? creep
Fail: (13) родитель(пат, боб) ? creep
Fail: (12) предок1(пат, боб) ? creep
Redo: (11) предок1(боб, боб) ? creep
Call: (12) родитель(боб, боб) ? creep
Fail: (12) родитель(боб, боб) ? creep
Fail: (11) предок1(боб, боб) ? creep
Redo: (11) родитель(том, _33600) ? creep
Exit: (11) родитель(том, лиз) ? creep
Call: (11) предок1(лиз, боб) ? creep
Call: (12) родитель(лиз, _61960) ? creep
Fail: (12) родитель(лиз, _61960) ? 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).

Решение не может быть найдено!

[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(10174, боб) ? creep

Call: (12) родитель(_10174, 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(_10174, боб) ? creep

Exit: (12) родитель(том, боб) ? creep

Exit: (11) предок2(том, боб) ? creep

Call: (11) родитель(лиз, том) ? creep

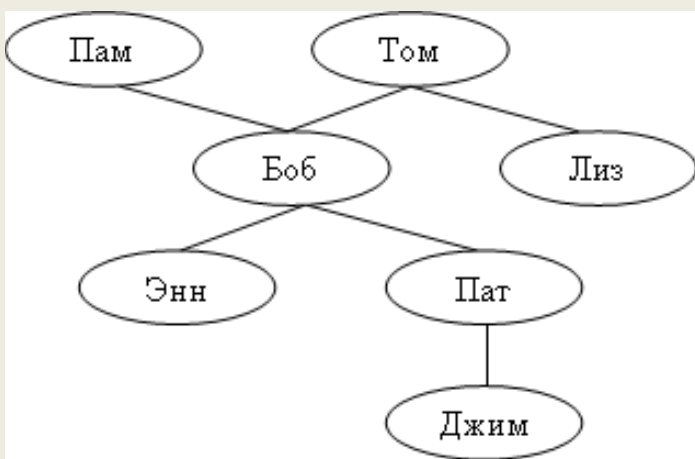
Fail: (11) родитель(лиз, том) ? creep

Redo: (11) предок2(_10174, боб) ? creep

Call: (12) предок2(19896, боб) ? creep

Call: (13) родитель(_19896, боб) ? creep

Exit: (13) родитель(пам, боб) ?



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

предок3(X,Y):-родитель(X,Y).

?- предок3(том,боб).

[trace] ?- предок3(том,боб).

Call: (10) предок3(том, боб) ? creep

Call: (11) предок3(_28704, боб) ? creep

Call: (12) предок3(_29516, боб) ? creep

Call: (13) предок3(_30328, боб) ? creep

Call: (14) предок3(_31140, боб) ? Unknown option (h for help)

Call: (14) предок3(_31140, боб) ? Unknown option (h for help)

Call: (14) предок3(_31140, боб) ? Unknown option (h for help)

Call: (14) предок3(_31140, боб) ? Unknown option (h for help)

Call: (14) предок3(_31140, боб) ?

Решение не будет найдено.

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 находится 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(у окна, на полу, в центре, 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 (удвери, напоцу, уокна, кет).

Цель игры:

состояние (-, -, -, ga).

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

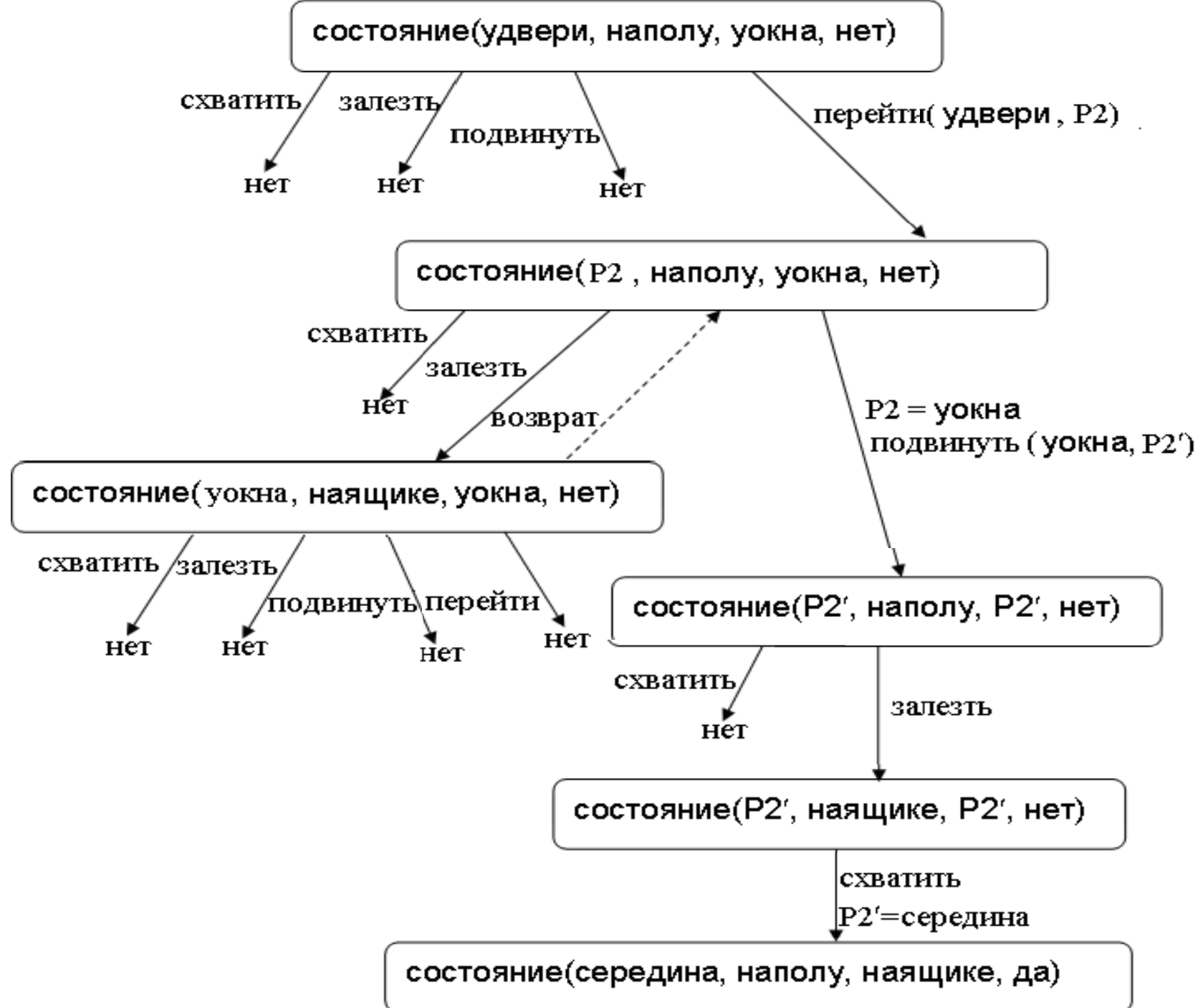
В результате хода меняется состояние обезьяньего мира, система переходит из одного состояния в другое. Для изменений определим предикат ход(состояние, действие, состояние).

Действия обезьяны: схватить, залезть, подвинуть, перейти.

```

goal:-может_схватить(состояние(удвери,наполу,уокна,нет)).
может_схватить(состояние(_,_,да)).
может_схватить(S):-ход(S,_S1),может_схватить(S1).
ход(состояние(вцентре,наящике,вцентре,нет),схватить,состояние(вцентре,наящике,вцентре,да)).
ход(состояние(P,наполу,P,нет),залезть,состояние(P,наящике,P,нет)).
ход(состояние(P,наполу,P,нет),подвинуть,состояние(P1,наполу,P1,нет)).
ход(состояние(P1,наполу,P,нет),перейти,состояние(P2,наполу,P,нет)).
[trace] ?- goal.
Call: (10) goal ? creep
Call: (11) может_схватить(состояние(удвери, наполу, уокна, нет)) ? creep
Call: (12) ход(состояние(удвери, наполу, уокна, нет), _31640, _31564) ? creep
Exit: (12) ход(состояние(удвери, наполу, уокна, нет), перейти, состояние(_32382, наполу, уокна, нет)) ? creep
Call: (12) может_схватить(состояние(_32382, наполу, уокна, нет)) ? creep
Call: (13) ход(состояние(_32382, наполу, уокна, нет), _34086, _34010) ? creep
Exit: (13) ход(состояние(уокна, наполу, уокна, нет), залезть, состояние(уокна, наящике, уокна, нет)) ? creep
Call: (13) может_схватить(состояние(уокна, наящике, уокна, нет)) ? creep
Call: (14) ход(состояние(уокна, наящике, уокна, нет), _36532, _36456) ? creep
Fail: (14) ход(состояние(уокна, наящике, уокна, нет), _37346, _36456) ? creep
Fail: (13) может_схватить(состояние(уокна, наящике, уокна, нет)) ? creep
Redo: (13) ход(состояние(_32382, наполу, уокна, нет), _38966, _34010) ? creep
Exit: (13) ход(состояние(уокна, наполу, уокна, нет), подвинуть, состояние(_39708, наполу, _39708, нет)) ? creep
Call: (13) может_схватить(состояние(_39708, наполу, _39708, нет)) ? creep
Call: (14) ход(состояние(_39708, наполу, _39708, нет), _41412, _41336) ? creep
Exit: (14) ход(состояние(_39708, наполу, _39708, нет), залезть, состояние(_39708, наящике, _39708, нет)) ? creep
Call: (14) может_схватить(состояние(_39708, наящике, _39708, нет)) ? creep
Call: (15) ход(состояние(_39708, наящике, _39708, нет), _43858, _43782) ? creep
Exit: (15) ход(состояние(вцентре, наящике, вцентре, нет), схватить, состояние(вцентре, наящике, вцентре, да)) ? creep
Call: (15) может_схватить(состояние(вцентре, наящике, вцентре, да)) ? creep
Exit: (15) может_схватить(состояние(вцентре, наящике, вцентре, нет)) ? creep
Exit: (14) может_схватить(состояние(вцентре, наящике, вцентре, нет)) ? creep
Exit: (13) может_схватить(состояние(вцентре, наполу, вцентре, нет)) ? creep
Exit: (12) может_схватить(состояние(уокна, наполу, уокна, нет)) ? creep
Exit: (11) может_схватить(состояние(удвери, наполу, уокна, нет)) ? creep
Exit: (10) goal ? creep
true .

```



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.

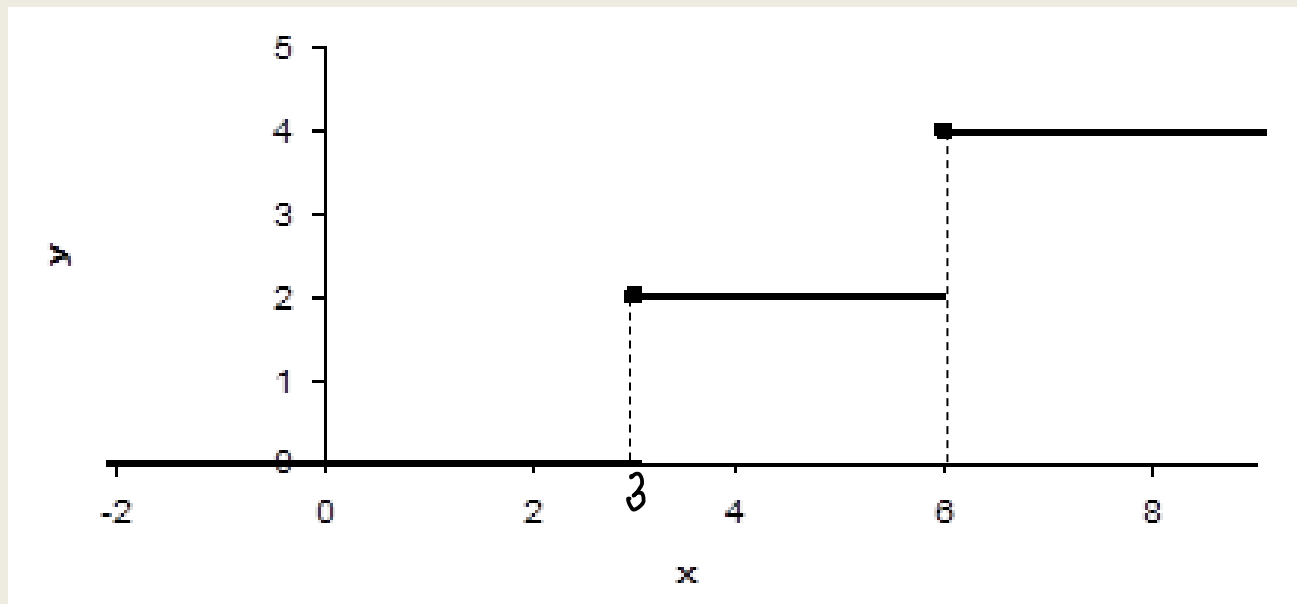
Определим предикат, который выводит всех сотрудников до 40 лет.

```
сотрудник(я,25).  
сотрудник(ч,45).  
сотрудник(с,23).  
сотрудник(м,41).  
сотрудник(и,20).  
goal1:-сотрудник(Ч,В),В<40,writeln(Ч),fail.  
goal1.
```

```
| goal1.  
я  
с  
и  
true.
```


2.9.2 Ограничение перебора – отсечение

Пример 2:



Аналитическое задание функции:

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

Программа на Прологе:

$f(X,0):-X \leq 3.$

$f(X,2):-X > 3, X \leq 6.$

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

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

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

Call: (11) $f(1, _14160)$? creep

Call: (12) $1 \leq 3$? creep

Exit: (12) $1 \leq 3$? creep

Exit: (11) $f(1, 0)$? creep

Call: (11) $0 > 2$? creep

Fail: (11) $0 > 2$? creep

Redo: (11) $f(1, _14160)$? creep

Call: (12) $1 > 3$? creep

Fail: (12) $1 > 3$? creep

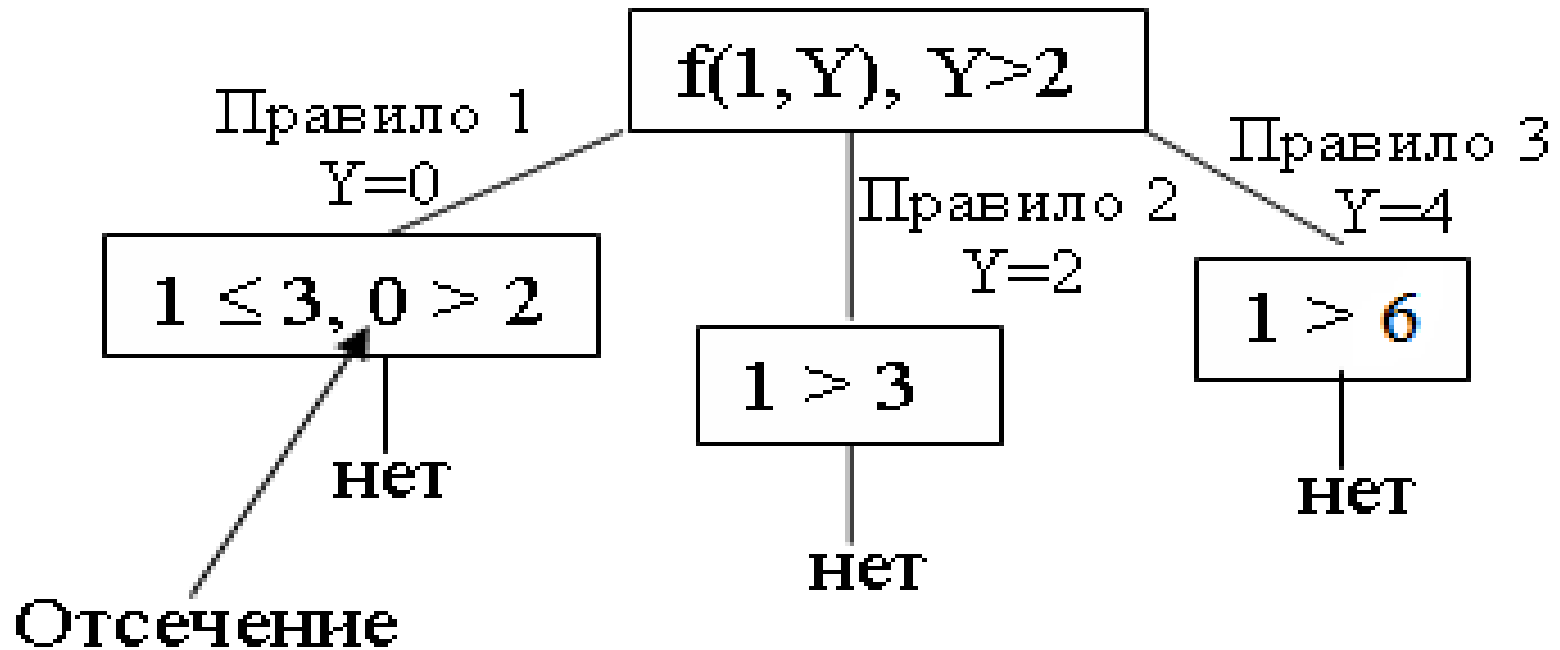
Redo: (11) $f(1, _14160)$? creep

Call: (12) $1 > 6$? creep

Fail: (12) $1 > 6$? creep

Fail: (11) $f(1, _14160)$? creep

false.



О том, что правило 1 успешно становится известно в точке, обозначенной на рисунке словом «Отсечение». Из этой точки 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, _26698) ? 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, _33566) ? creep

Call: (11) 7=<3 ? creep

Fail: (11) 7=<3 ? creep

Redo: (10) f(7, _33566) ? creep

Call: (11) 7>3 ? creep

Exit: (11) 7>3 ? creep

Call: (11) 7=<6 ? creep

Fail: (11) 7=<6 ? creep

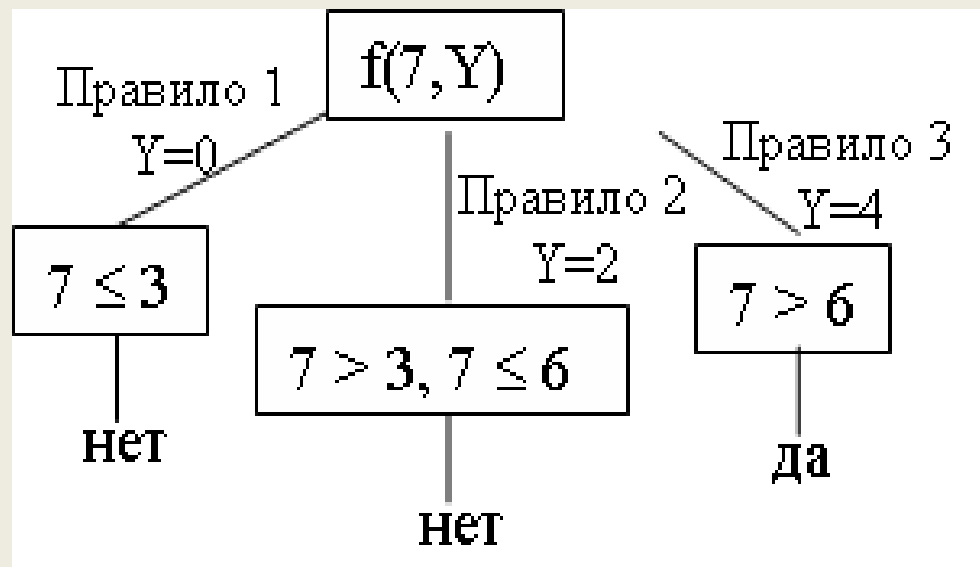
Redo: (10) f(7, _33566) ? creep

Call: (11) 7>6 ? creep

Exit: (11) 7>6 ? creep

Exit: (10) f(7, 4) ? creep

Y = 4.



Новое определение функции:

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

```
[trace] ?- f(7,Y).  
Call: (10) f(7, _45358) ? creep  
Call: (11) 7=<3 ? creep  
Fail: (11) 7=<3 ? creep  
Redo: (10) f(7, _45358) ? creep  
Call: (11) 7=<6 ? creep  
Fail: (11) 7=<6 ? creep  
Redo: (10) f(7, _45358) ? 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(2,Y).  
Y = 0  
Unknown action: : (h for help)  
Action?  
Unknown action: ж (h for help)  
Action? ;  
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:

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

Если цели B_1, \dots, 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, \dots, 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 $1, 2, \dots, N$. $1+2+\dots+N$

$s(1,1).$

$s(N,S):-N1 \text{ is } N-1,s(N1,S1),S \text{ is } S1+N.$

?-sum(2,X).

?- s(5,S).

S = 15

Unknown action: : (h for help)

Action? ;

ERROR: Stack limit (1.0Gb) exceeded

ERROR: Stack sizes: local: 0.9Gb, global: 78.0Mb, trail: 0Kb

ERROR: Stack depth: 10,185,804, last-call: 0%, Choice points: 3

ERROR: Possible non-terminating recursion:

ERROR: [10,185,804] user:s(-10185789, _20437630)

ERROR: [10,185,803] user:s(-10185788, _20437650)

$s(1,1):-!$

$s(N,S):-N1 \text{ is } N-1,s(N1,S1),S \text{ is } S1+N.$

?- s(5,S).

S = 15.

?-sum(-3,X).

?- s(-3,S).

ERROR: Stack limit (1.0Gb) exceeded

ERROR: Stack sizes: local: 0.9Gb, global: 77.8Mb, trail: 0Kb

ERROR: Stack depth: 10,190,924, last-call: 0%, Choice points: 3

ERROR: Possible non-terminating recursion:

ERROR: [10,190,924] user:s(-10190917, _20384356)

ERROR: [10,190,923] user:s(-10190916, _20384376)

s(N,_):-N<0,!,fail.

s(1,1):-!.

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

?- s(-3,S).

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

```
?- goal2.  
Слово?  
|: a.  
a  
|: s.  
s  
|: s.  
s  
|: g.  
g  
|: stop.  
  
true.
```