Тезисы занятий по логическому программированию

Лекция 1. Введение в Пролог

Логическое программирование (PROgramming in LOGic – PROLOG) является одним из классических видов программирования (структурное, объектное, объектно-ориентированное, функциональное и т.д.).

В чистом виде – академический язык, достаточно простой, но этот курс позволяет отделить зерна от плевел: тех кто, хочет и умеет думать, от тех, кто не хочет либо не умеет. В свое время японцы планировали выпустить компьютер, работающий по технологии Пролога, планировалось к 1993 году получить «компьютеры пятого поколения». В Прологе была обнаружена серьезная проблема – он был предназначен для решения задач искусственного интеллекта. Большинство задач ИИ – переборные, а Пролог оказался не эффективен для решения переборных задач.

В настоящее время на практике используют в основном CLP-расширения Пролога (www.sics.se/sicstus). CLP – constraint logic programming, его мы рассмотрим в конце нашего курса. Главная их особенность в том, что программисту не надо задумываться над способом решения задачи, ему нужно только выполнить полностью корректное математическое описание задачи и задача будет решена автоматически. Конечно, решение получается не настолько эффективным, как при использовании специализированных решений, но прельщает своей универсальностью. Стоят CLP–расширения дорого. В настоящее время появились CLP-расширения для других языков программирования, например, для C++ и для Java (www.ilog.com). Мы на лабораторных работах будем использовать GNU Prolog (www.inria.fr, www.gprolog.org). В нем реализованы некоторые простые элементы CLP-программирования.

Список литературы:

Братко «Пролог Программирование для искусственного интеллекта»

Стерлинг, Шапиро «Искусство программирования на языке Пролог»

Первое знакомство начнем с примеров.

Факт Том является родителем Боба на Прологе может быть записан следующим образом:

```
parent(tom, bob).
```

Все. Программа, состоящая из одной строки кода.

Запрос:

```
?- parent(tom, bob).
Yes
```

Обратите внимание, что в отличие от других языков программирования Пролог **всегда** «всего лишь» отвечает Да или Нет. Это означает «Удалось доказать» или «Не удалось доказать». «Есть такой факт» – «Нет такого факта».

Дополним программу родственными узами.

```
parent(tom, bob).
parent(ann, bob).
parent(tom, liza).
parent(bob, mary).
parent(bob, luk).
parent(luk, kate).
```

Одно из важных условий: в конце **всегда** ставится точка.

И – на всякий случай – после parent не следует ставить пробел перед открывающей скобкой.

Как Пролог обрабатывает запросы.

```
?- parent(tom, liza).
```

Производится сравнение с первой строчкой – tom = tom, bob = liza → no.

Со второй строчкой tom = ann → no.

С третьей строчкой tom = tom, liza = liza → yes.

Пролог отвечает:

```
Yes
```

Конечно, может быть задан более интересный вопрос: Кто родитель liza?

```
?- parent(X, liza).
```

Обратите внимание на написание. X – переменная.

Производится сравнение с первой строчкой – X = tom, bob = liza → no.

Со второй строчкой X = tom, bob = liza → no.

С третьей строчкой X = tom, liza = liza → yes.

Пролог отвечает:

```
X=tom
Yes
```

Напишите запрос, который гласит «Кто потомок tom?»

Что ответит Пролог?

Какие будут варианты ответа на запрос?

```
?- parent(X, Y).
```

При просмотре **альтернативных решений** в Прологе следует нажимать клавишу «Точка с запятой», для окончания просмотра вариантов – «Ввод».

Конечно, можно написать и более сложный запрос «Кто является прародителем luk?»

```
?- parent(X, Y), parent(Y, luk).
```

Какой будет ответ?

Аналогично можно написать запрос «Кто является правнуками tom?»

Предложите вариант написания запроса «Верно ли, что bob и liza имеют общего родителя?».

Таким образом:

1. В Пролог легко определяются отношения зависимости. Например, родственные связи. Пользователь может легко выяснить какие именно зависимости определены в программе.
2. Программы Пролога описываются в терминах **правил (фактов).**
3. В качестве аргументов запросов могут выступать как конкретные объекты, так и переменные.
4. В результате выполнения запроса Пролог отвечает только Да или Нет. Т.е. доказательство прошло успешно (succeeded) или не успешно (failed).

Теперь решите несколько простых задачек:

Что Пролог ответит на запрос:

```
?- parent(X, kate).
?- parent(kate, X).
?- parent(X, luk), parent(X, mary).
```

Расширим нашу программу родственных связей дополнительными условиями.

Определим пол всех людей, представленных в программе.

Например:

```
female(kate).
male(tom).
```

Или по–другому:

```
sex(kate, feminine).
sex(tom, masculine).
```

Важно то, что Прологу абсолютно безразлично как именно Вы определите пол. Т.к. для него, что parent, что male, female, что sex одинаково не понятные слова. Обратите внимание, что в данном случае Пролог ничего не знает о том, ЧТО вы программируете. До настоящего момента мы не определили ни одного системного слова, или как они в Прологе называются – **предиката.**

Конечно, можно в программе оставить оба варианта, но при этом – вы сами можете запутаться, поэтому рекомендую определиться и использовать только один, хоть и любой вариант.

Другой вариант расширения.

Мы использовали понятие родитель – parent, давайте попробуем расширить программу понятием потомок – offspring.

```
offspring(bob, tom).
```

Конечно, бессмысленно переписывать все факты, тем более, что можно где–то ошибиться и получить бессмысленный результат.

Необходимо написать следующую программу: «X является потомком Y, если Y является родителем X».

```
offspring(X, Y) :- parent(Y, X).
```

Это первое полноценное **правило** в Прологе, с которым мы с вами познакомились.

В Прологе существует всего три конструкции описания программы: факты, правила и вопросы.

Факт: имя(параметры).

Вопрос: ?– тело вопроса.

Правило: имя(параметры) :– тело правила.

В правиле до знака :– находится голова правила, после – тело правила.

Факт отличается от правила тем, что описывает условие, которое всегда верно. Правило (голова правила) верно только в том случае, если удается доказать тело правила.

Рассмотрим запрос:

```
?- offspring(bob, X).
X=tom;
X=ann
Yes
```

Как Пролог это получил? Он просто при доказательстве offspring(bob, X) использовал правило и ему потребовалось доказать только правую часть правила – parent(X, bob).

Здесь важно отметить, что при выполнении запроса на языке Пролог, если существует несколько вариантов ответов, то для просмотра следующего варианта используют клавишу «точка с запятой».

Теперь вы сможете написать на Прологе программу «мама».

Домашнее задание:

Написать программы «сестра» (sister), «прародитель» (grandparent). В качестве исходных данных можно использовать все, что было представлено на сегодняшней лекции.

```
sister(X, Y) :- parent(Z, X), parent(Z, Y), female(X), female(Y).
grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
```

Лекция 2. Рекурсивное определение правил и использование GNU Prolog

Проверка домашнего задания и исправление ошибок студентов.

Написание программы «Прапрародитель».

Программа «Предшественник»:

```
predecessor(X, Y) :- parent(X, Y).
predecessor(X, Y) :- parent(X, Z), predecessor(Z, Y).
```

Все программы на Пролог можно прочесть на русском языке. В данном случае: X является предшественником Y, если X является родителем Y или X является родителем Z, который является предшественником Y.

Вопрос:

```
?- predecessor(tom, mary).
```

Yes

Рассмотрим, как Пролог будет доказывать это утверждение. Для этого пронумеруем все правила (факты parent от 1 до 6 и правила `predecessor` от 7 до 8). Обратите внимание, что при написании программ в Прологе *нумерация не используется.* Нумерацию мы добавили для себя, для удобства **трассировки** (пошагового выполнения) программы.

```
1. parent(tom, bob).
2. parent(ann, bob).
3. parent(tom, liza).
4. parent(bob, mary).
5. parent(bob, luk).
6. parent(luk, kate).
7. predecessor(X, Y) :- parent(X, Y).
8. predecessor(X, Y) :- parent(X, Z), predecessor(Z, Y).
```

В правиле 8 используется запятая. Запятая ⇔ «И».

Трассировка

Пролог анализирует правила с 1 по 6, но во всех случаях не совпадает имя.

Пролог использует правило 7, в котором совпало имя, при этом переменным присваиваются следующие значения: X = tom, Y = mary. Следующая промежуточная цель – parent(tom, mary). Для доказательства истинности правила 7 необходимо доказать истинность этой промежуточной цели.

(1) 7. X = tom, Y = mary → parent(tom, mary)

В Прологе промежуточная цель называется **резольвента.**

Пролог анализирует правила с 1 по 6, но во всех случаях не совпадает по крайней мере один атрибут. В правилах 7 и 8 не совпадает имя. Таким образом доказательство не успешно.

(1') 7. X = tom, Y = mary → parent(tom, mary) → **no**

Пролог использует правило 8, в котором совпало имя, при этом переменным присваиваются следующие значения: X = tom, Y = mary. Следующая промежуточная цель – parent(tom, Z).

(2) 8. X = tom, Y = mary → parent (tom, Z)

(3) 1. parent(tom, bob), Z = bob → **yes**

(2') 8. X = tom, Y = mary → parent(tom, Z), **Z = bob** → **predecessor(bob, mary)**

(4) 7. X = bob Y = mary → parent(bob, mary)

(5) 1. parent(bob, mary) = parent(tom, bob) no

(6) 2. parent(bob, mary) = parent(ann, bob) → no

(7) 3. parent(bob, mary) = parent(tom, liza) → no

(8) 4. parent(bob, mary) = parent(bob, mary) → yes

(4') 7. X = bob, Y = mary → parent (bob, mary) → **yes**

(2'') 8. X = tom, Y = mary → parent(tom, Z), Z = bob → predecessor(bob, mary) → **yes**

Здесь в круглых скобках показаны шаги трассировки, штрих у номера шага означает возврат к соответствующему правилу и продолжение его доказательства (на доске мы это обычно рисуем стрелками), после номера шага – номер используемого правили. Шаги трассировки с 5 по 8 показывают как Пролог пытается доказать резольвенту (в самом начале трассировки мы эти подробности пропустили).

В результате трассировки становится очевидна корректность работы программы.

В процессе трассировки мы присваивали переменным значения. В Прологе есть одна важная особенность, касающаяся присваивания значения переменным, которая отличает его от всех других языков программирования. **Если переменной присвоено значение, то это значение не может быть изменено в той же ветви доказательства.** При этом имя переменной имеет смысл только в рамках одного правила. Т.е. переменные X, встречающиеся в двух разных правилах – разные переменные X. (Можно как аналогию рассмотреть другие языки программирования – разные функции, например, на C++ и значение локальной переменной).

На примере программы grandparent: переменная X в рамках правила 7 – что в левой части, что в правой – одна и та же переменная. Однако, переменные X в правиле 7 и в правиле 8 – разные переменные и, в принципе, могут иметь разные названия.

Обратите внимание, что правило 8 в программе – рекурсивное. Это наиболее общий случай: большинство программ на языке Пролог строятся с использованием этого принципа программирования. Соответственно, в процессе доказательства наблюдаются **возвраты,** например, при доказательстве (1) – (1’) в рассмотренной выше программе. Так как доказательство происходит рекурсивно, то возвратов при доказательстве может быть множество. Важно, что именно при возврате происходит **переход на новую ветвь доказательства,** а значит переменные, которым в этой (той, из которой осуществляется возврат) «подветви» были присвоены значения, могут начать изменять свои значения.

Обратите внимание, что правила 7 и 8 можно поменять местами. При этом изменится и процесс доказательства цели, что можно описать в виде трассировки. Правила с 1 по 6, как и правила 7 и 8 представляют собой **процедуру.** Процедура – набор правил с одинаковой «головой».

Правила 7 и 8 могут быть объединены.

```
predecessor(X, Y) :- parent(X, Y); parent(X, Z),
predecessor(Z, Y).
```

Точка с запятой ⇔ «ИЛИ». Использовать это свойство следует с осторожностью.

Условный пример:

```
A :- B, C; D, E, F. ⇔ A :- (B, C); (D, E, F). ⇔
A :- B, C.
A :- D, E, F.
```

Для применения ваших знаний на практике следует воспользоваться GNU Prolog. Рассмотрим основные правила использования GNU Prolog.

Если Пролог еще не установлен, то запустите setup–gprolog–L2.16.exe и следуйте инструкциям по установке.

Учтите, что в результате установки GNU Prolog создает на рабочем столе ярлык для запуска, в котором не указана рабочая директория, поэтому все загружаемые тексты программ он будет искать на рабочем столе. Есть два возможных варианта решения: (1) прописать рабочую директорию и именно в нее помещать тексты программ, которые будут загружаться вПролог, (2) запускать GNU Prolog с использованием gprolog.exe прямо из папки BIN, тогда он будет ожидать, что все загружаемые файлы находятся в папке BIN.

Пролог ожидает, что все файлы называются **имя.pl.** Например, lab1.pl

Пролог имеет дружественный консольный интерфейс, поэтому в нем нет кнопки или пункта меню «Открыть...» Для открытия / загрузки файла необходимо выполнить следующую команду:

```
?- consult(имя_файла_без_расширения).
```

Другой вариант: имя файла без расширения в квадратных скобках:

```
?- [имя_файла_без_расширения].
```

Пример использования для файла «fact.pl»:

```
?- [fact].
compiling C:\soft\GNU-Prolog\bin\fact.pl for byte code...
C:\soft\GNU-Prolog\bin\fact.pl compiled, 1 lines read - 856 bytes written, 15 ms
```

В случае, если при компиляции текста программы обнаружены ошибки – будут отображены сообщения с ошибками с указанием строки и столбца, в которых обнаружена ошибка.

Для перезагрузки измененного текста в Прологе следует использовать команду reconsult(имя_файла), но в GNU Prolog она работает не правильно.

Для проверки загруженного в память текста программы можно воспользоваться предикатом `listing.`

Важно! В GNU Prolog текст программы и вывод на экран должны быть на английском языке, т.к. русские шрифты не поддерживаются.

Забегая вперед: write(X) – предикат вывода значения переменной X на экран.

При этом:

```
?- X = 2, write(X).
2
X = 2
yes
```

В то же время:

```
?- X = 2.
?- write(X).
16
yes
```

Т.е. присваивание значения переменной происходит только в рамках того же правила, при переходе к другому правилу – переменная имеет другое значение. В последнем примере – значение не определено. Фактически, это пример к тому свойству переменной, которое мы указали раньше.

Важное свойство Пролога: Пролог разработан как недетерминированный язык программирования. Соответственно, для его эффективного выполнения нужен недетерминированный компьютер, каковой в природе не существует. Поэтому Пролог осуществляет доказательство последовательно: сверху вниз по списку правил и фактов в программе и слева направо при доказательстве одного правила.

Другое важное свойство Пролога. Что такое «Да» и «Нет» в Прологе:

Да ⇔ Мне удалось это доказать

Нет ⇔ Мне НЕ удалось это доказать

Рассмотрим программу:

```
fallible(X) :- man(X).
man(socrates).
```

Читаем текст программы: «Все люди ошибаются. Сократ – человек».

Задаем вопрос:

```
?- fallible(socrates).
yes
```

«Ошибается ли Сократ?» – «Да»

```
?- fallible(platon).
no
```

«Ошибается ли Платон?» – «Нет»

Почему же Платон не ошибается? Да потому что он даже не человек! (Какой будет ответ на вопрос ?– `man(platon).` Пример показывает, что на вопрос «Ошибается ли Платон?» Пролог отвечает «Я не могу этого доказать».

Таким образом:

1. Важное свойство переменных Пролога: они независимы в разных правилах и запросах, но в процессе доказательства их значение может быть изменено только при возврате.
2. Т.к. большая часть доказательства в Прологе строится с использованием принципа рекурсии, то при доказательстве часто происходят возвраты (backtracking).
3. «Нет» в Прологе означает «Я не могу доказать это».
4. Важное отличие Пролога от структурных языков программирования – он является декларативным языком, т.е. сосредоточивается на том, ***что*** будет ответом на вопрос, а не на том, ***как*** этот ответ был получен (процедурные языки).
5. Любая программа на языке Пролог может быть прочитана на естественном языке (см. пример с «Сократом»).

Домашнее задание:

На основании программы «родственные связи» написать программу «Внук» (grandchild) и программу «Счастлив каждый, у кого есть ребенок» (happy).

```
grandchild(X, Y) :- grandparent(Y, X).
happy(X) :- parent(X, Y).
```

Несколько определений:

Атом – последовательность латинских символов, цифр и знака подчеркивания, начинающаяся с маленькой латинской буквы или произвольная последовательность символов, заключенная в одинарные кавычки. Примеры: tom, niL, n_32, 'South America'. На будущее, когда будем работать с файлами, то не читайте из файлов строки, начинающиеся с большой буквы – GNU Prolog их интерпретирует не правильно.

Переменная –

1. последовательность латинских символов, цифр и знака подчеркивания, начинающаяся с заглавной латинской буквы;
2. последовательность латинских символов, цифр и знака подчеркивания, начинающаяся со знака подчеркивания;
3. знак подчеркивания. В последнем случае она называется **безымянной переменной** и используется тогда, когда значение переменной для разработчика *безразлично.* При этом важно, что даже встречаясь в одном правиле безымянные переменные *могут иметь разные значения.*

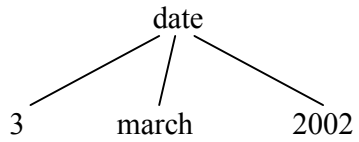
Примеры: Atom, VaRiAbLe, _345, _.

При загрузке текста программы в Пролог Вы, возможно, уже встречали сообщения «Предупреждение. Переменная X нигде не используется!!!». Это Пролог напоминает Вам, что Вы объявили переменную, но нигде ее не использовали. Если она Вам не нужна – используйте безымянную переменную, и Пролог не будет ругаться.

Структуры – составные термы языка Пролог.

Например, дата может быть представлена в виде структуры, состоящей из Дня, Месяца и Года. Для представления даты мы можем использовать **функтор:**

```
date(3, march, 2002).
```



Здесь date – имя, в скобках – параметры функтора. Напоминаю, что date я только что придумал, нет в Пролог такого слова. Т.е. я мог дату назвать как-нибудь по-другому, изменить порядок следования параметров. Главное в этом деле – самому не запутаться и *не запутать преподавателя.*

Запрос к структуре:

```
?- date(Day, march, 2002).
```

Функтор – составной терм, в качестве имени используется атом, за которым в скобках через запятую перечисляются параметры функтора. В качестве параметров могут выступать любые логические термы. Т.е. там могут быть атомы, переменные, функторы, математические выражения и т.п. Примеры: d(t(334, m), list).

Количество параметров функтора называется **арностью.**

Предположим, что мы описываем фигуры на плоскости. Они описываются с помощью точек. Возможный вариант описания – поточечный. Как это сделать – варианты есть?

Представление точки: `point(X, Y).`

Представление треугольника:

```
triangle(point(0, 0), point(0, 4), point(3, 0)).
```

Получился – «египетский треугольник» ($3^2+4^2=5^2$)

Для трехмерного представления мы можем использовать, например, `point(X, Y, Z).`

Важное отличие – разное количество параметров. Т.е. как и в других языках программирования, важно не только имя, но и количество параметров.

Как вы думаете, что ответит Пролог на запрос:

```
?- X=2+2.
X=2+2
yes
```

А его никто не просил посчитать. «Плюс» – такой же функтор и может быть представлен следующим образом `+(2, 2).`

Соответственно, `(a+b)*(c+d)` ⇔ `*(+(a, b), +(c, d))`

Писать лабораторные работы в этом стиле не рекомендую, но учитывать это следует.

Итак, вычисление в Прологе производится с помощью предиката is.

```
?- X is 2+2.
X=4
yes
```

У предиката is в левой части всегда находится **одна** не унифицированная переменная (ей не присвоено значение), а в правой – математическое выражение, в котором **все** переменные унифицированы.

Кстати, учитывая свойства переменных в языке Пролог **нельзя** писать следующее:

```
X is X+1
```

Знаки: +, –, *, / - как обычно

= - присваивание значения

>, <, >= - как обычно

=< - меньше либо равно
\= - не равно
Как вы думаете, чем отличается == от =:= ?
== - сравнение, =:= - сравнение с вычислением левой и правой части.
`2 == 1+1` → no, в то же время `2 =:= 1 + 1` → yes.

Рассмотрим программу «Вычисление факториала»: **fact(Number, Value)**

```
fact(0, 1).
fact(N, V) :- N>O, N1 is N-1, fact(N1, V1), V is V1*N.
```

Нумерация, как обычно – «для внутреннего использования» (Пролог ее не поймет).

Читаем по-русски: «Значение факториала от 0 равно 1. Значение факториала от N, если N > 0, равно произведению N на значение факториала от N – 1».

Обратите внимание, что программирование на Прологе напоминает **доказательство по индукции.** В данном примере: первое правило (факт) – база индукции, второе правило – индукционный переход от X_N к X_{N+1}.

Рассмотрим трассировку для запроса

```
?- fact(3, X).
```

Трассировка:

```
(1)  1. no
(2)  2. N = 3, V - ?, N1 = 2 → fact(2, V1)
(3)  1. no
(4)  2. N = 2, V - ?, N1 = 1 → fact(1, V1)
(5)  1. no
(6)  2. N = 1, V - ?, N1 = 0 → fact(0, V1)
(7)  fact (0, 1) → V1 = 1
(6') 2. N = 1, V - ?, N1 = 0 → fact(0, V1), V1 = 1, V = 1.
(4') 2. N = 2, V - ?, N1 = 1 → fact(1, V1), V1 = 1, V = 2.
(2') 2. N = 3, V - ?, N1 = 2 → fact(2, V1), V1 = 2, V = 6.
X = 6
yes
```

Напишите программу вычисления «чисел Фибоначчи» (fib). Для тех, кто не помнит:

$\mathbf{X_0} = 1$, $\mathbf{X_1} = 1$, $X_{N+2} = \mathbf{X_{N+1}} + X_N$. Программа пишется по аналогии с вычислением факториала: fib(Number, Value).

```
fib(0, 1).
fib(l, 1).
fib(N, V) :- N1 is N-1, N2 is N-2, fib(N1, V1), fib(N2, V2), V is V1+V2.
```

Важна последовательность элементов в третьем правиле. Важна последовательность строк в программе. Полезно добавить «N > 1» в третье правило, а на что влияет отсутствие «N > 1» в третьем правиле?

Рассмотрим трассировку программы для запроса:

```
?- fib(4, X).
```

При этом можно построить дерево вычислений (см. рис.).

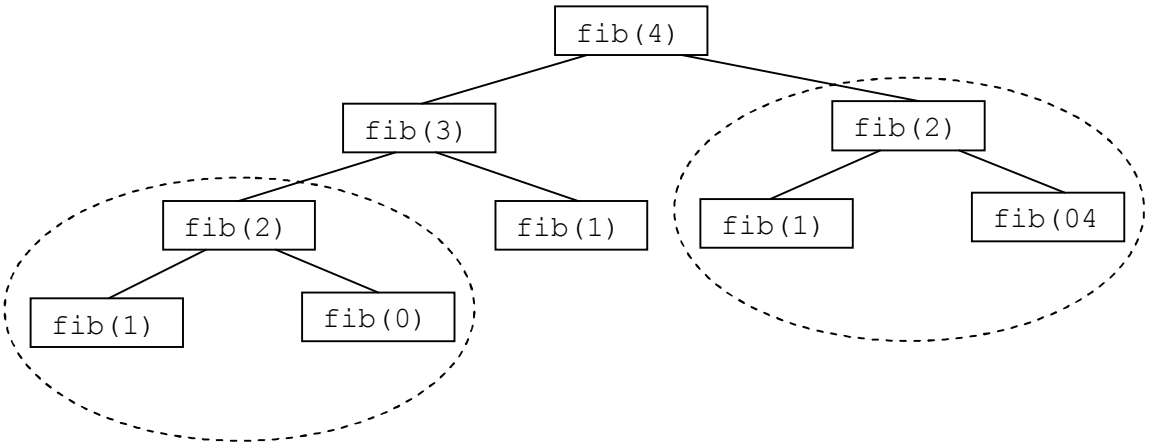
При этом пунктиром выделены повторяющиеся элементы дерева. Соответственно, вычисление проводилось не эффективно.

Домашнее задание: написать программу вычисления чисел Фибоначчи, которая бы вычисляла числа Фибоначчи за один проход (без повторных вычислений).

Рассмотрим программу: «Обезьяна и банан».

У нас есть доска размером 5x5 в координатах (1, 1) находится обезьяна, в координатах (4, 4) находится банан. Двигаться обезьяна может только вправо либо вверх. Необходимо написать программу, которая проверит: «может ли обезьяна добраться до банана».

5					
4				Банан	
3					
2					
1	Обезьяна				
	1	**2**	**3**	**4**	**5**



Возможный вариант решения:

```
1. start(1, 1).
2. stop(4, 4).
3. go :- start(X, Y), move(X, Y).
4. move(X, Y) :- stop(X, Y).
5. move(X, Y) :- X < 5, X1 is X+1, move(X1, Y).
6. move(X, Y) :- Y < 5, Y1 is Y+1, move(X, Y1).
```

Как обычно, нумерация – для нас, а не для Пролога. Правила 5 и 6 могут быть переписаны следующим образом:

```
5. move(X, Y) :- X < 5, X1 is X+1, Y1 is Y, move(X1, Y1).
6. move(X, Y) :- Y < 5, Y1 is Y+1, X1 is X, move(X1, Y1).
```

Оба варианта каждого правила в данном случае абсолютно идентичны.

Домашнее задание – написать трассировку данной программы и графически отобразить на рисунке (см. рисунок с обезьяной и бананом).

Лекция 3. Списки и отладка

Проверка домашнего задания.

Если на рисунке пронумеровать шаги программы, то получится следующее:

5					8,13,17
4				Банан,18	7,12,16
3				14	6,11,15
2				9	5,10
1	Обезьяна	1	2	3	4
	1	**2**	**3**	**4**	**5**

Почему Пролог ведет себя именно таким образом?

Что будет, если попросить его передоказать решение (найти альтернативы)?

Домашнее задание: написать программу «Обезьяна и банан» на бесконечном поле с эффективным поиском решения (без возвратов).

Список – последовательность логических термов, перечисленных через запятую и заключенных в квадратные скобки.

Пустой список – открывающая и закрывающая квадратные скобки без элементов внутри.

Примеры списков: [1, house, f(4)], [], [a, b, c].

Список делится на две части: **голову** и **хвост**.

Голова – первый(ые) элемент(ы) списка, хвост – остаток списка.

Для деления на голову и хвост используется вертикальная черта.

Пример деления:

Список	**Голова**	**Хвост**
`[a | b, c]`	`a`	`[b, c]`
`[a, b | c]`	`a, b`	`[c]`
`[a, b, c | ]`	`a, b, c`	`[]`

`[a, b, c]` ⇔ `[a | [b, c]]` ⇔ `[a, b | [c]]` ⇔ `[a, b, c |]`

Пустой список нельзя разделить на голову и хвост!!!

Хвост – всегда список!!!

Список – структура языка Пролог, которая может быть записана следующим образом:

.(Голова, Хвост)

[a, b, c] ⇔ .(a, .(b, .(c, [])))

[a] ⇔ .(a, [])

Но это академические знания, т.к. GNU Prolog их не понимает.

Рассмотрим полезный пример, который он понимает – «Вхождение элемента в список».

```
member(Elem, [Elem | _]).
member(Elem, [Head | Tail]) :- member(Elem, Tail).
```

Программа может быть прочитана следующим образом: «Элемент содержится в списке, если он находится в голове списка или в его хвосте».

Что ответит Пролог на следующие запросы? Выполните трассировку:

```
?- member(a, [b, a, c]).
?- member(a, [b, a, a]).
?- member(a, [b, c, X]).
?- member(X, [a, b, c]).
```

Программа «объединения двух списков» conc(List1, List2, ResultList):

```
?- conc([a, b], [c, d], [a, b, c, d]).
yes
```

Решение:

```
1. conc([], L, L).
2. conc([Head | Tail], L, [Head | NewTail]) :- conc(Tail, L,
NewTail).
```

Обратите внимание, что вторая строчка программы может быть переписана следующим образом:

```
2. conc([Head | Tail], L, ResultList) :- conc(Tail, L, NewTail),
ResultList = [Head | NewTail].
```

Выполните трассировку программы для следующих запросов:

```
?- conc([a, b], [c], Res).
?- conc(X, [c], [a, b, c]).
?- conc(X, Y, [a, b, c]).
```

Как можно записать программу **member** с использованием **conc** (в одну строчку)? Какой вариант будет быстрее работать?

```
member1(X, L) :- conc(L1, [X | L2], L).
```

Как можно написать программу добавления элемента в список:
add(Item, SourceList, DestList)?

```
add(X,L,[X|L]).
```

Напишите программу удаления элемента из списка: **del(Item, SourceList, DestList).**

```
del(X,[X|Tail],Tail).
del(X,[Y|Tail],[Y|Tail]) :- del(X,Tail,Tail).
```

Проверьте функцию del на запрос

```
?- del(X, [red, green, blue]).
```

Напишите программу удаления всех вхождений элемента в список:
delAll(Item, SourceList, DestList).

Необходимо написать программу генерации перестановок, которая при возврате и повторном передоказательстве должна генерировать все возможные перестановки элементов списка. Решение:

```
permutation([], []).
permutation(L, [X | P]) :- del(X, L, L1), permutation(L1, P).
```

Как это работает. База индукции: перестановка пустого списка – это пустой список. Индукционный переход: из списка L удаляется элемент (функция del начинает с удаления первого элемента и затем начинает удалять их последовательно один за другим), который затем помещается в голову перемешанного списка. Перемешивание начинается с конца списка.

```
?- permutation([red, green, blue], P).
```

Задача: есть список чисел. Необходимо получить этот список в упорядоченном по возрастанию виде. Решение:

```
ordered([]).
ordered([_]).
ordered([X, Y | T]) :- X<Y, ordered([Y | T]).
```

Пустой список и список из одного элемента – упорядочены. Если в голове списка есть два элемента, то список упорядочен, если первый элемент меньше второго, а список из второго и хвоста так же упорядочен.

Теперь задача сортировки решается в одну строчку:

```
sort(SourceList, DestList) :- permutation(SourceList, DestList),
ordered(DestList).
```

У метода сортировки два параметра: первый – исходный список, второй – упорядоченный. Сортировка считается завершенной, если после перестановки элементов в исходном списке получился упорядоченный по возрастанию список.

Усовершенствуем программу «обезьяна и банан», добавив вычисление пути, по которому обезьяна добралась до банана.

```
1. start(1, 1).
2. stop(4, 4).
3. go(Path):- start(X, Y), move(X, Y, [], Path).
4. move(X, Y, P, [m(X, Y) | P]) :- stop(X, Y).
5. move(X, Y, From, To) :- X<5, X1 is X+1,
move (X1, Y, [m(X, Y) | From], To).
6. move(X, Y, From, To) :- Y<5, Y1 is Y+1,
move (X, Y1, [m(X, Y) | From], To).
```

Для проверки корректности программы выполним к ней запрос:

```
?- go(P).
P = [m(4,4),m(4,3),m(4,2),m(4,1),m(3,1),m(2,1),m(1,1)] ?;
P = [m(4,4),m(4,3),m(4,2),m(3,2),m(3,1),m(2,1),m(1,1)] ?;
P = [m(4,4),m(4,3),m(3,3),m(3,2),m(3,1),m(2,1),m(1,1)] ?
yes
```

В данном примере программа передоказывалась дважды после первого успешного доказательства.

Почему это работает: в предикат move добавлены два параметра. Первый добавленный параметр инициализируется пустым списком и накапливает каждый вновь выполненный шаг, второй добавленный параметр – рекурсивно возвращает результат после того, как мы добрались до банана. Функтор m(X, Y) просто хранит координаты точки, в которую мы пришли.

Обратите внимание, что путь выдается в обратном порядке. Как можно сделать путь в правильном порядке?

```
1. start(1, 1).
2. stop(4, 4).
3. go(Path) :- start(X, Y),move(X, Y, [], Path).
4. move(X, Y, P, [m(X, Y) | P]) :- stop(X, Y).
5. move(X, Y, From, [m(X, Y) | To]) :- X<5, X1 is X+1,
move(X1, Y, From, To).
6. move(X, Y, From, [m(X, Y) | To]) :- Y<5, Y1 is Y+1,
move(X, Y1, From, To).
```

И тогда результат запроса будет другой:

```
?- go(P).
P = [m(1,1),m(2,1),m(3,1),m(4,1),m(4,2),m(4,3),m(4,4)]
yes
```

Обратите внимание, чем отличаются эти программы.

В первом случае мы сохраняем используемые координаты в процессе движения вглубь рекурсии, во втором случае мы делаем то же самое, но при возврате из рекурсии.

На всякий случай, для тех кто все еще не понял. Переписываю строчку 4 последнего примера:

```
4. move(X, Y, P, PReturn) :- stop(X, Y),  PReturn = [m(X, Y) | P].
```

Обе строчки идентичны. Просто в данном случае я ввел дополнительную переменную PReturn, появления которой в данном случае можно избежать.

Выполнение отладки в GBU Prolog:

trace – предикат, включающий трассировку.
При этом будут показываться все команды и все операции: call, fail, exit, redo, exception.
notrace – отключает трассировку.

spy(Спецификация_того_за_кем_наблюдаем) – отладка конкретного предиката.
Спецификации бывают следующих видов:

1. [PredSpec1, PredSpec2,…] – наблюдение за предикатами в списке.
2. Name – наблюдение за одним предикатом.
3. Name/Arity – наблюдение за предикатом с заданным именем и **арностью**.
4. Name/A1-A2 – наблюдение за предикатом с заданным именем и арностью из интервала от A1 до A2

nospy(Имя_предиката) – отключает наблюдение за предикатом.

Пример использования:

```
?-spy(conc).
Spypoint placed on conc/3
yes
?-spy(conc/3).
```

Для того, чтобы наблюдение за предикатом заработало, необходимо включить режим отладки.
debug – включает режим отладки.
nodebug – отключает режим отладки.

Для ограничения вывода информации в режимах отладки и трассировки используется предикат **leash**(Настройка).
В качестве настройки может выступать:

1. full – эквивалентно [call, exit, redo, fail, exception]
2. half – эквивалентно [call, redo]
3. loose – эквивалентно [call]
4. none – эквивалентно []
5. tight – эквивалентно [call, redo, fail, exception]

```
?-leash(full).
Using leashing stopping at [call, exit, redo, fail, exception]
ports
yes
```

Отладка включается автоматически при попытке доказательства с включенной трассировкой или при попытке debug доказательства предиката, за которым включено наблюдение.

В процессе трассировки / отладки можно выполнять следующие команды:
Enter – переход к следующей строке доказательства
a – прекратить доказательство
h/? – полная справка по командам отладки

Домашнее задание:
Напишите программу вычисления длины списка: **length(List, Length)**.
Напишите программу вхождения подсписка в начало списка: **prefix(SubList, MainList)**.
Напишите программу проверки вхождения одного списка в другой в качестве подсписка **sublist(SubList, TestList)**. Писать следует по аналогии с программой member.

```
length([], 0).
length([_ | Tail], N) :- length(Tail, N1), N is N1+1.
prefix(X, Y) :- conc(X, _, Y).
sublist(S, L) :- conc(L1, L2, L), conc(S, L3, L2).
```

Лекция 4. Управление перебором

Проверка домашнего задания.

Несколько полезных встроенных предикатов...

repeat – предикат, который всегда верен. Он всегда доказывается не зависимо от направления доказательства. Т.к. доказательство производится слева направо, то движение слева направо обозначим как успешное, обратное, как откат:

1. repeat → (доказывается успешно)

1. repeat → ← (попытка отката до repeat)
2. repeat →← → (доказывается успешно)

Предикат repeat может быть записан на Прологе в виде следующей программы:

```
repeat.
repeat :- repeat.
```

Что произойдет, если строчки поменять местами?

fail – предикат, который всегда НЕ верен. Он НИКОГДА не доказывается.

1. → fail (не доказывается)
2. ← fail (не доказывается)

Следствие: все, что будет перечислено через запятую после fail никогда не будет доказываться (мы туда просто не попадем).

Предикат fail может быть записан на Прологе в виде следующей программы:

```
fail :- 1 == 0.
```

Из repeat и fail может быть получен бесконечный цикл:

```
?- ..., repeat, ... , fail.
```

Из этого цикла программа выйдет по Out of memory, т.к. при доказательстве repeat генерируется новая ветвь доказательства, которая размещается в памяти.

Рассмотрим программу определения минимума из двух чисел:

```
1.min(X, Y, X) :- X<Y.
2.min(X, Y, Y).
```

Если X меньше Y, значит минимальный – X, иначе – минимальный – Y. Проверяем программу:

```
?- min(2, 3, Min).
Min = 2
yes
```

В чем ошибка в программе?

```
?- min(2, 3, Min), Min>2.
Min = 3
yes
```

Т.е. минимальное число равно 2? Правильное решение:

```
1. min(X, Y, X) :- X<Y.
2. min(X, Y, Y) :- X>=Y.
```

Второй вариант правильного решения с «отсечением»:

```
1. min(X, Y, X) :- X<Y, !.
2. min(X, Y, Y).
```

Оператор отсечения в Прологе обозначается восклицательным знаком – «!». Отсечение всегда доказывается при прямом доказательстве, а при попытке отката запрещает передоказательство (поиск альтернатив) того правила, в котором указано отсечение.

Как можно задать конструкцию **if-then-else** на Прологе:

```
A :- B, !, C.
A :- D.
```

Если удалось доказать B, то доказывается C, иначе доказывается D. Если убрать отсечение, то при неуспешном доказательстве C может произойти попытка передоказательства B, а это уже будет не if-then-else.

Предикат member(Item, List) давал возможность находить несколько решений по вхождению элемента в список. Можно его исправить, чтобы он искал только первое вхождение:

```
member(Elem, [Elem | _]) :- !.
member(Elem, [Head | Tail]) :- member(Elem, Tail).
```

Выделяют два вида отсечений: красные и зеленые. Деление условное.

Зеленые отсечения не влияют на логику выполнения программы, а только отсекают ветви перебора, в рамках которых решения быть не может. Красные – влияют и могут отсекать решения.

С использованием отсечения может быть описан предикат отрицания **not.**

```
not(X) :- X, !, fail.
not(_).
```

Что он делает: если X удается доказать, то not – не верен, т.к. сочетание отсечения и fail приведет к неуспешности доказательства not. Если X не удается доказать, то происходит поиск альтернативы первому правилу, которая оказывается верна, не зависимо от того, какой у нас X.

Пример:

```
?- X = 2, not(X == 3).
X = 2
yes
?- X = 2, not(X == 2).
no
```

Программа вычитания одного списка из другого **minus(L1, L2, Diff)**. Т.е. в Diff находятся только те элементы, которые встречаются в L1 и не встречаются в L2.

```
minus([],_, []).
minus([X | L1], L2, L) :- member(X, L2), !, minus(L1, L2, L).
minus([X | L1], L2, [X | L]) :- minus(L1, L2, L).
```

Если исходный список пуст, то и результат вычитания – пуст. Если «голова» исходного списка есть в результирующем, то про «голову» забываем и осуществляем вычитание из «хвоста». Если «головы» нет в списке, то при возврате нужно «голову» поместить в результирующий список.

Пример:

```
?- minus([a, b], [b], L).
L = [a]
yes
```

Краткая справка для выполнения лабораторных работ:

read(X) – предикат для чтения X из текущего входного потока.

write(X) – предикат вывода X в текущий выходной поток.

nl – выводит в выходной поток знак перевода каретки.

Пример использования:

```
?- write('Enter value: '), read(X), write('result = '), write(X).
Enter value: 12345.
result = 12345
X = 12345
yes
```

Обратите внимание, что после 12345 стоит **точка!!!** после которой выполнен **перевод каретки.**

Для работы с файлами:

see(ИмяФайла) – перенаправляет входной поток на чтение данных из файла.

seen – закрывает входной поток, если читали из файла.

see(user) – перенаправляет входной поток на чтение данных из стандартной консоли.

tell(ИмяФайла) – перенаправляет выходной поток на вывод данных в файл.

told – закрывает выходной поток, если выводили в файл.

tell(user) – перенаправляет выходной поток вывод данных в стандартную консоль.

Файлы могут обрабатываться только последовательно. Каждый запрос на чтение из входного файла приводит к чтению в текущей позиции текущего входного потока. После этого чтения текущая позиция, будет перемещена на следующий, еще не прочитанный элемент данных. Следующий запрос на чтение приведет к считыванию, начиная с этой новой текущей позиции. Если запрос на чтение делается в конце файла, то в качестве ответа на такой запрос выдается атом **end_of_file** (конец файла). **Считанную один раз информацию считать вторично невозможно.**

В файле data.txt находится 12345 *без точки и перевода каретки.* Результат запроса:

```
?- see('data.txt'), write('Enter value: read(X),
write('result = '), write(X), seen.
Enter value:
uncaught exception: error(syntax_error('data.txt:1 (char:6) or
operator expected after expression'), read/1)
```

В файле data.txt находится 12345. *с точкой без перевода каретки*. Результат запроса:

```
?- see('data.txt'), write('Enter value: '), read(X),
write('result = '), write(X), seen.
Enter value: result = end_of_file
X = end_of_file
yes
```

В файле data.txt находится 12345. *с точкой и переводом каретки*. Результат запроса:

```
?- see('data.txt'), write('Enter value: '), read(X),
write('result35'), write(X), seen.
Enter value: result=12345
X = 12345
yes
```

В файле данные должны быть с маленькой буквы или в одинарных кавычках!!!

Лекция 5. Операторы, работа с базой данных

Проверка домашнего задания.

Определение операторов:

:– op(Приоритет, Спецификатор, Имя_предиката).

Приоритет – число от 1 до 1200. Чем выше приоритет, тем меньше число.

Спецификатор использует **x, y** и **f.**

f – наш предикат, который мы определяем.

x – предикат с приоритетом строго выше приоритета f.

y – предикат с приоритетом выше либо равным приоритетом f.

Способы задания:

1. инфиксные операторы трех типов: **xfx xfy yfx**
2. префиксные операторы двух типов: **fx fy**
3. постфиксные операторы двух типов: **xf yf**

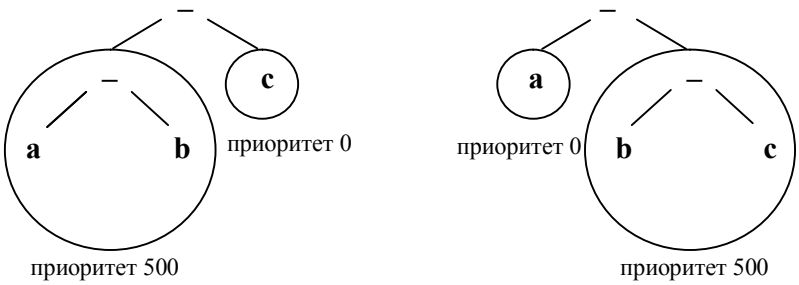
Пример:

```
:- op(500, yfx, -).
```

Такой способ определения гласит: предикат «минус» имеет приоритет 500 и слева от него может располагаться равный ему предикат, а справа – только меньший. Тогда, запись

`a - b - c` будет интерпретироваться как и `(a - b) - c`

Как показано на рисунке слева.



Если же поменять местами:

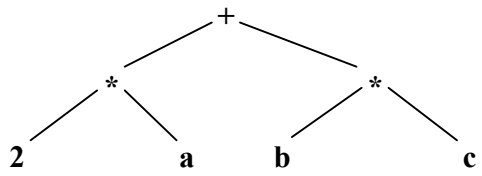
```
:- op(500, xfy, -).
```

Тогда будет верен рисунок справа и логика вычитания начнет не совпадать со всей той математикой, которой вас учили в школе.

Для того, чтобы корректно выполнялись математические операции в Пролог используется следующие приоритеты операций:

```
:- op(500, yfx, -).
:- op(500, yfx, +).
:- op(400, yfx, *).
:- op(400, yfx, /).
```

Тогда выражение `2*a+b*c` будет интерпретировано как показано на следующем рисунке:



Т.е. сначала выполняются операции с большим приоритетом (меньший номер приоритета), а затем операции с меньшим приоритетом. Спецификаторы же показывают способ последовательного выполнения операторов (кто и в каком порядке будет выполняться).

И если мы теперь определим в программе:

```
:- op(1000, fx, not).
```

Тогда сможем выполнить запрос:

```
?- X = 2, not X == 3.
X = 2
yes
```

Обратите внимание на отсутствие скобок у предиката **not.**

Домашнее задание необходимо определить все предикаты для следующего известного правила:

```
~ (A & B)  <===>  ~A v ~B
```

Который должен читаться как

```
эквивалентно(not(и(A, B)), или(not(A, not(B)))
```

assert(X) – добавляет факт X в программу.

retract(X) – удаляет факт X из программы.

Добавление имеет две модификации **assertz** – добавить в конец программы, **asserta** – добавить в начало программы.

Пример использования:

```
?- assertz(data(1)).
yes
?- data(X).
X = 1
yes
?- listing.
data(1).
yes
?- retract(data(_)).
yes
?- listing.
yes
```

Для корректного обращения к динамическому предикату его следует определить как динамический. Для этого в программе следует вызвать **dynamic(Имя_предиката/Арность_предиката).**

Пример с числами Фибоначчи. Текст программы:

```
:-dynamic(fibon/2).
fib(0, 1).
fib(l, 1).
fib(N, V) :- N1 is N-1, N2 is N-2, (fibon(N1, V1); fib(N1, V1)),
(fibon(N2, V2); fib(N2, V2)), V is V1+V2, asserta(fibon(N,V)).
```

При такой реализации решения количество рекурсивных вызовов **fib** существенно уменьшается. Почему вместо **fib** используется **fibon** для хранения данных? fibon используется, т.к. fib определен как статический предикат, а Пролог не позволит вносить изменения в статические предикаты. **Изменения можно вносить только** в **динамические предикаты!**

Предикат **abolish(Имя_предиката_Арность_предиката)** удаляет все вхождения предиката с данным именем и данной арностью.

Напишем аналогичный предикат удаления через retract: retractAll(X).

```
retractAll(X) :- retract(X), retractAll(X).
retractAll(_).
```

Он удаляет все вхождения X в нашу программу.

Задача: определить статические переменные в Пролог с использованием assert и retrtact.

Способ работы:

init(ИмяПеременной, Значение) – инициализация статической переменной заданным значением.

set(ИмяПеременной, Значение) – установка значения в переменную.

get(ИмяПеременной, Значение) – получение значения из переменной.

Решение:

```
init(Var, Val) :- assertz(variables(Var, Val)).
set(Var, Val) :- retract(variables(Var, _)),
assertz(variables(Var,Val)).
get(Var, Val) :- variables(Var, Val).
```

Пример использования:

```
?- init(t, 3), init(v, 4).
yes
?- set(t, data), get(t, T), get(v, V).
T = data
V = 4
yes
```

Естественно, повторная инициализация приведет к неправильной работе программы.

Предикаты read, write, assertz, asserta, retract, abolish являются внелогическими и в случае возврата повторно не доказываются.

Программа возведения числа в квадрат:

```
square :- repeat, nl, write('Enter X = '), read(X), (X = end, !;
Y is X*X, write('X*X = '), write(Y), fail).
```

Пользователь вводит числа – они возводятся в квадрат. Это происходит до тех пор, пока пользователь не введет end и программа корректно выйдет либо не введет вместо числа что–то другое и тогда произойдет exception.

```
?- square.
Enter X = 23.
X*X = 529
Enter X = 45.
X*X = 2025
Enter X = end.
yes
```

Для защиты от некорректного ввода хорошо бы проверить, что ввели число:

```
square :- repeat, nl, write('Enter X = '), read(X), (X = end, !;
number(X), Y is X*X, write('X*X = '),write(Y), fail).
```

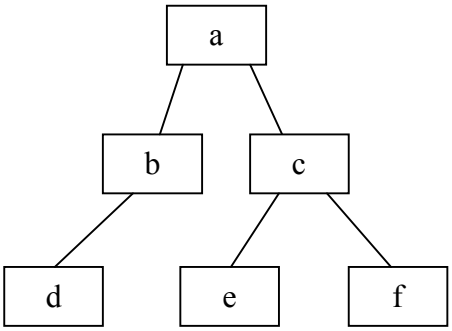
Тогда получим:

```
?- square.
Enter X = er.
Enter X = 345.
X*X = 119025
Enter X = end.
yes
```

Здесь: **number(X)** – встроенный предикат, проверяющий, что X является числом. Как его можно записать на языке Пролог (как мы это делали с fail и repeat)?

Бинарные деревья.

Бинарное дерево – дерево, имеющее максимум две ветки в каждом узле.



Варианты задания бинарных деревьев в Прологе:

1. a(b(d),c(e,f)).

2. Вводим понятие nil – пустое поддерево. Обозначим дерево предикатом btree с тремя параметрами: первый – корень, второй – левое поддерево, третий – правое поддерево.

```
btree(a, btree(b, btree(d, nil, nil), nil), btree(c, btree(e, nil, nil), btree(f, nil, nil)))
```

Второй вариант записи более громоздкий, но позволяет удобно писать программы.

Рассмотрим программу поиска элемента в дереве – аналог программы member, но уже для деревьев.

Назовем ее **in(Item, Tree).**

В качестве первого параметра – искомый элемент, в качестве второго параметра – поддерева вида btree(...)

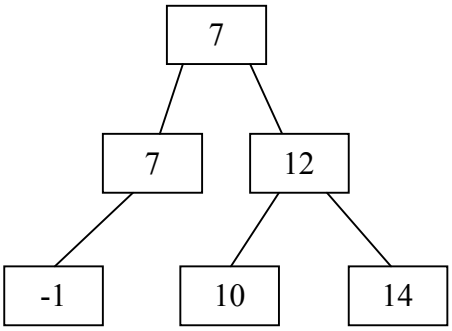
```
1. in(Item, btree(Item, _, _)).
2. in(Item, btree(_, Left, _)) :- in(Item, Left).
3. in(Item, btree(_, _, Right)) :- in(Item, Right).
```

Первое правило – база индукции – гласит: искомый элемент находится в корне дерева. Второе правило гласит: искомый элемент находится в левом поддереве. Третье правило – в правом поддереве.

Напишите трассировку для запроса:

```
?- in(b, btree(a, btree(b, nil, nil)), nil).
```

Представленная программа осуществляет поиск в глубину. Предположим, что у нас не просто бинарное дерево, а бинарный словарь: в узлах дерева находятся числа, причем все числа левого поддерева меньше либо равны корня, а все узлы правого – строго больше.



Необходимо усовершенствовать программу поиска вхождения элемента в дерево. Кстати, именно такие деревья строит СУБД Oracle при индексировании.

```
1. inS(Item, btree(Item, _, _)).
2. inS(Item, btree(Root, Left, _)) :- Item =< Root, in(Item, Left).
3. inS(Item, btree(Root, _, Right)) :- Item > Root, in(Item, Right).
```

Бинарное дерево можно представить в виде списка: **btree2list(Tree, List)**.

```
1. btree2list(Tree, List):- btree2list(Tree, [], List).
2. btree2list(nil, List, List).
3. btree2list(btree(Root, Left, Right), List, [Root | RList]) :-
btree2list(Left, List, List1), btree2list(Right, List1, RList).
```

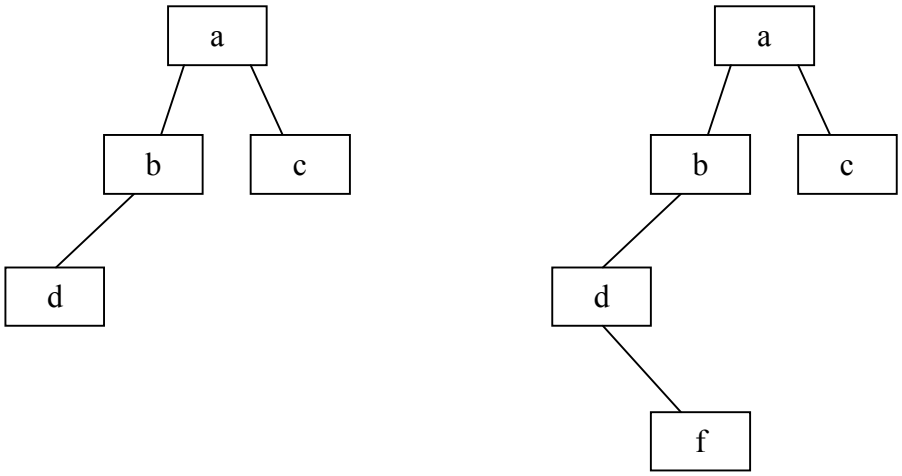
Первое правило направляет вызов предиката с тремя параметрами, у которого инициализируется один из параметров пустым списком. Второй – база индукции. Третий – сначала собирает результаты из одного списка, затем из другого, после чего добавляет к ним корень дерева.

Рассмотрим запрос и ответ Пролога в режиме трассировки:

```
?- btree21ist(btree(a, nil, nil), R).
     1    1   Call:  btree2list(btree(a, nil, nil), _20) ?
     2    2   Call:  btree2list(btree(a, nil, nil), [], _20) ?
     3    3   Call:  btree2list(nil, [], _117) ?
     3    3   Exit:  btree2list(nil, [], []) ?
     4    3   Call:  btree2list(nil, [], _77) ?
     4    3   Exit:  btree2list(nil, [], []) ?
     2    2   Exit:  btree2list(btree(a, nil, nil), [], [a]) ?
     1    1   Exit:  btree2list(btree(a, nil, nil), [a]) ?
R = [a]
yes
```

После каждого знака вопроса нажималась клавиша «Enter».

Домашнее задание: написать программу, которая проверяет, что бинарное дерево сбалансировано, т.е. в нем разница глубины деревьев не больше 1. Пример:



Левое дерево сбалансировано, правое дерево – не сбалансировано, т.к. разница между ветвью «c» и ветвью «f» равна 2.