Тема 2. Рекурсивное определение правил и использование GNU Prolog

Написание программы «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 неуспешно.

```
(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 правила. 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 predecessor: 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 ещё не установлен, то запустите setup-gprolog-1.4.5.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 неё помещать тексты программ, которые будут загружаться в Пролог, (2) запускать GNU Prolog с использованием gprolog.exe прямо из папки BIN, тогда он будет ожидать, что все загружаемые файлы находятся в папке BIN.

Простейшие программы можно опробовать Online: https://www.jdoodle.com/execute-prolog-online (обязательно включите «Interactive mode» и «Focus View»).

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

Пролог имеет дружественный консольный интерфейс, поэтому в нём 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 неправильно.

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».
Задаём 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:

- Важное свойство переменных Пролога: они независимы в разных правилах и запросах, но в процессе доказательства их значение может быть изменено только при возврате.
- Т.к. большая часть доказательства в Прологе строится с использованием принципа рекурсии, то при доказательстве часто происходят возвраты (backtracking).
- «Нет» в Прологе означает «Я не могу доказать это».
- 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).
- Любая программа на языке Пролог может быть прочитана на естественном языке (см. пример с «Сократом»).

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 неправильно.

Переменная –

- последовательность латинских символов, цифр и знака подчеркивания, начинающаяся с заглавной латинской буквы;
- последовательность латинских символов, цифр и знака подчеркивания, начинающаяся со знака подчеркивания;
- знак подчеркивания. В последнем случае она называется **анонимной 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*.

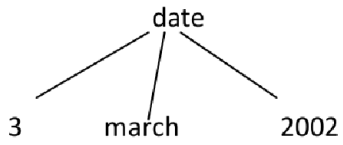
Примеры: Atom, VaRiAbLe, _345, _.

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 её 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*.

Вопрос к структуре:

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

Функтор (предикат) – 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 вопрос:

```
?- 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)**

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

Нумерация, как обычно – «для внутреннего использования» (Пролог её поймёт).

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
```

Напишем программу вычисления «чисел Фибоначчи» (fib):

$\mathbf{X_0 = 1, X_1 = 1, X_{N+2} = X_{N+1} + X_N}$.

Программа пишется по аналогии с вычислением факториала: fib(Number, Value).

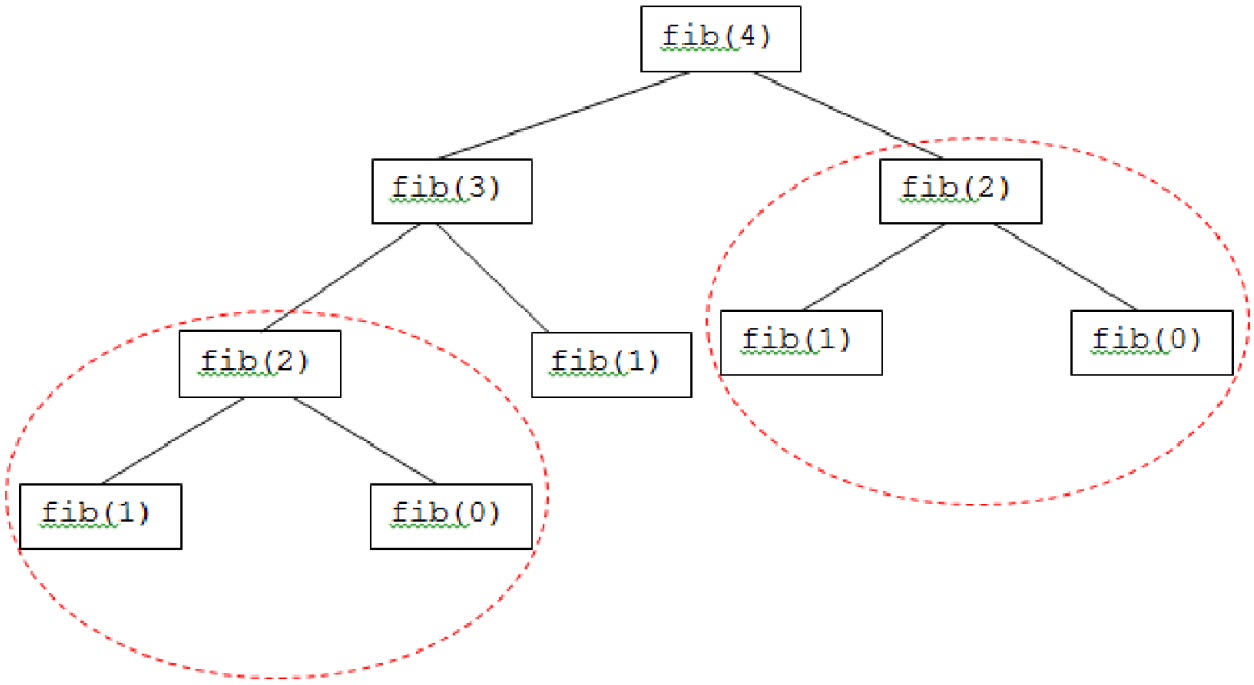
```
fib(0, 1).
fib(1, 1).
fib(N, V) :- N1 is N - 1, N2 is N - 2, fib(N1, V1), fib(N2, V2),
V is V1 + V2.
```

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 неэффективно.

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

Рассмотрим задачи, которые могут быть решены с использованием рекурсии:

1. Создайте предикат, вычисляющий неотрицательную степень целого числа.
2. Создайте предикат, вычисляющий по натуральному числу N сумму чисел от 1 до N.
3. Создайте предикат, вычисляющий по натуральному числу N сумму нечетных чисел, не превосходящих N.

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 «как может 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 = 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.

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**