Тема 5. Операторы, работа с базой данных

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 меньше приоритета f

y – предикат с приоритетом выше либо равным приоритету f

Способы задания:

- инфиксные операторы трех типов: **xfx** **xfy** **yfx**
- префиксные операторы двух типов: **fx** **fy**
- постфиксные операторы двух типов: **xf** **yf**

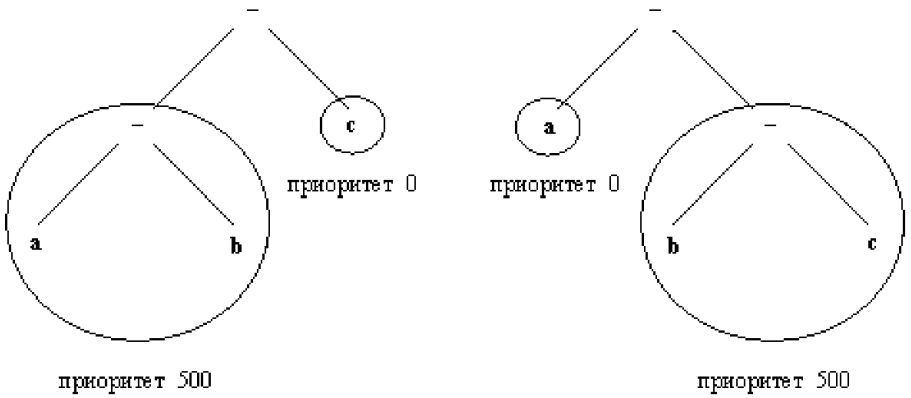
Пример:

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

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

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

Как показано на рисунке слева:



Если же поменять местами:

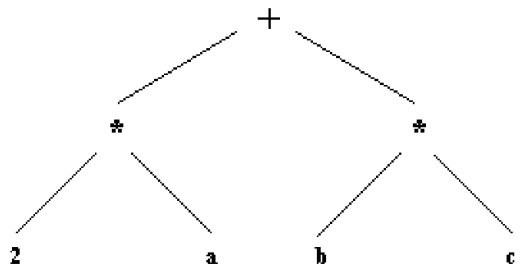
```
:- op(500, xfy, -).
```

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 Прологе используются 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 (больший номер приоритета). Спецификаторы же показывают способ последовательного выполнения операторов (кто и в каком порядке будет выполняться).

И если мы теперь определим в программе:

```
:- op(1000, fy, 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 добавления правил в программу их необходимо поместить в дополнительные скобки: `assertz((Правило))`.

Пример:

```
?- assertz(man(socrat)), assertz((fallible(X):-man(X))).
yes
```

```
?- fallible(socrat).
yes
?- fallible(X).
X = socrat
yes
?- listing.

% file: user_input
fallible(A) :- man(A).
man(socrat).
yes
```

Для корректного обращения к динамическому предикату его следует определить как динамический. Для этого в программе следует вызвать **dynamic(Имя_предиката/Арность_предиката)**, но некоторые версии Пролога могут обходиться без данного определения.

Пример с числами Фибоначчи. Текст программы:

```
:-dynamic(fibon/2).
fib(0, 1).
fib(1, 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 произойдёт 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.