Лекция № 3

Управление в PROLOG-программах

Control in PROLOG programs

Методы организации выполнения PROLOG -программ

Для управления процессом выполнения Пролог-программ используются два встроенных предиката **cut** и **fail** (отсечение и отказ/неудача).

fail — тождественно ложный предикат (*fail/0*). Его исполнение вырабатывает значение «неудача».

true — тождественно истинный предикат (*true/0*). Его исполнение всегда успешно.

cut — выполнение предиката *cut/0* (**!**) всегда завершается успешно, но сопровождается рядом побочных эффектов.

Введение предиката **cut** связано со стремлением получить более эффективные программы, предоставляя возможность сокращения дерева поиска решения за счет отсечения бесполезных ветвей.

Методы организации выполнения PROLOG -программ

Предикат **cut** можно интерпретировать следующим образом:

cut запрещает использовать все утверждения (правила и факты), лежащие в программе ниже, а также все альтернативные выводы конъюнкции целей, находящиеся левее **cut** в рассматриваемом утверждении.

cut не влияет на выполнение целей, находящихся правее в утверждении; на этом участке может быть построено несколько решений и, естественно, допустим возврат.

H :- B_1, …, B_m, !, B_{m+1} , ... , B_n.
H :- C_1, …, C_k.

…

H :- D_1, …, D_r.

Методы организации выполнения Пролог-программ

BAF-метод (Backtrack After Fail — возврат после отказа/неудачи).

Суть его заключается в организации повторяющихся отказов некоторой цели и возвратов: в результате – выполнение целей, лежащих левее цели, приводящей к отказу, будет возобновляться. Естественно, наиболее подходящим кандидатом постоянно отказывающей цели является предикат **fail**.

```
книга(толстой, "Война и мир").
книга(толстой, "Анна Каренинна").
книга(достоевский, "Идиот").
книга(достоевский,"Братья Карамазовы").
print_book :- книга(X,Y), write(X),
                write(". "), writeln(Y), fail.
```

С помощью этой программы можно вывести названия всех книг:

```
?– print_book.
```

Методы организации выполнения Пролог-программ

CAF-метод (Cut And Fail — отсечение и отказ/неудача).

В некоторых случаях при выполнении программы полезно ограничить поиск по базе данных программы. Повторяющееся выполнение целей можно организовать с помощью предиката **fail**.

Естественным расширением этого приема является не сканирование всей базы данных, а организация «забывания» не просмотренной еще ее части при выполнении некоторого условия.

Таким образом, для организации повторений можно, как и раньше, использовать возврат при неудаче, а с помощью **cut** запретить его при выполнении определенных условий.

Методы организации выполнения Пролог-программ

Допустим, мы хотим реализовать предикат ***max(X, Y, Max)***, который сравнивает два числа ***X*** и ***Y***, а результат записывает в переменную ***Max***.

Вариант 1.

```
max (X, Y, X) :- X >=Y.
max (X, Y, Y) :- X<Y.
?- max (5, 2, Z).
```

Вариант 2.

```
max (X, Y, X) :- X >=Y, !.
max (X, Y, Y).
```

Методы организации выполнения Пролог-программ

С помощью отсечения можно эффективно реализовать конструкцию, подобную оператору *case* в императивных языках программирования.

```
action(1) : - !, write("Вы нажали единицу.").
action(2) :- !, write("Вы нажали двойку.").
action(3) :- !, write("Вы нажали тройку.").
action(_) :- write("Я не знаю это число.").

my_goal:-writeln("Введите число от 1 до 3"),
        read(X),
        write(X),
        action(X).
```

Методы организации выполнения Пролог-программ

UDR-метод (User-Defined Repeat — повторения, управляемые пользователем).

В отличие от предыдущих двух методов, в которых количество повторений ограничено общим количеством неопробованных альтернатив, здесь повторения могут выполняться произвольное число раз. Для этой цели используется предикат ***repeat***:

```
repeat.
repeat :– repeat.          /* в SWI Prolog - встроенный предикат */
```

Общий вид циклического правила:

```
rule :– repeat,
        <тело цикла>,
        <условие выхода>,
         !.
```

Пример правила:

```
do.echo :– repeat,
           readln (Name),
           write (Name), nl.
           Name = '*',
           !.
```

Рекурсия в языке Prolog

В языке Пролог одним из основных методов управления выполнением программ является *рекурсия.*

Как правило, Пролог-программа представляет собой совокупность рекурсивных или взаимно рекурсивных определений.

Создадим отношение «быть предком», используя предикат «родитель». Для того чтобы один человек был предком другого человека, нужно, чтобы он либо был его родителем, либо являлся родителем другого его предка.

предок (Предок, Потомок):–
родитель(Предок, Потомок).
/* предком является родитель */
предок (Предок, Потомок):–
родитель(Предок, Человек),
предок(Человек, Потомок).
/* предком является родитель предка */

Рекурсивное правило

Рекурсивная процедура включает *базис* и *шаг рекурсии*.

Базис рекурсии — это предложение, определяющее некую начальную ситуацию или ситуацию в момент завершения рекурсии. Как правило, в этом предложении записывается некий простейший случай, при котором ответ получается сразу даже без использования рекурсии. Это предложение часто содержит условие, при выполнении которого происходит выход из рекурсии, или отсечение. (Базис может включать несколько предложений.)

Шаг рекурсии — это правило, тело которого содержит в качестве подцели вызов определяемого предиката.
Чтобы избежать зацикливания, определяемый предикат должен вызываться не от тех же параметров, которые указаны в заголовке правила. Параметры должны изменяться на каждом шаге так, чтобы в итоге либо сработал базис рекурсии, либо условие выхода из рекурсии, размещенное в самом правиле.

Рекурсивное правило

Рекурсивное правило общего вида можно представить с помощью следующей схемы:

<Рекурсивное правило> :–

<Список предикатов>,

< Предикат, определяющий условие выхода>,

< Список предикатов>,

< Рекурсивное правило>,

< Список предикатов>.

Предложения, построенные по рассмотренной схеме, принято называть GRR-правилами (General Recursive Rule — правило рекурсии общего вида).

Итеративное вычисление факториала

```
real  factorial (N: integer);
var  I: integer;
     F: real;
begin
     I := 0; F := 1;
     while  (I < N) do
          begin
          I := I+1; F := F*I
          end;
     factorial := F
end;
```

Рекурсивное вычисление факториала

Рекурсивное вычисление факториала:

```
factorial (1,1) :– !.          /* условие останова рекурсии */
factorial (N, FactN) :–  N1 is N - 1,
                         factorial (N1, FactN1),
                         FactN is N* FactN1.
```

?- *factorial* (5, F).

Рекурсивное вычисление числа Фибоначчи

```
fib (1, 1) :- !. /* первое число Фибоначчи равно единице */
fib (2, 1) :-!.  /* второе число Фибоначчи равно единице */
fib (N, F) :-
        N1 is N-1, fib(N1,F1),   /* F1 это N–1-е число Фибоначчи */
        N2 is N-2, fib(N2,F2),   /* F2 это N–2-е число Фибоначчи */
        F is F1+F2.              /* N-е число Фибоначчи равно
                                    сумме N–1-го и N–2-го чисел
                                    Фибоначчи */
```

Проблемы, связанные с рекурсией

предок (Предок, Потомок):–
родитель(Предок, Потомок).
/* предком является родитель */
предок (Предок, Потомок):–
предок(Человек, Потомок),
родитель(Предок, Человек).
/* предком является родитель предка */

Из-за того, что во втором правиле первая подцель является рекурсивным вызовом, происходит зацикливание.

Это явление называется *левосторонняя* рекурсия.

Работа данной процедуры заканчивается, когда переполняется стек.

Аналогично может переполняться стек и в представленной ранее процедуре рекурсивного вычисления факториала.

Хвостовая рекурсия

Рассмотрим правило ***A***, заданное в виде рекурсивного предложения

$$A :- B_1,\ B_2,\ \ldots,\ B_{n-1},\ A'.$$

где ***A'*** - рекурсивный вызов **A**.

Оптимизация потенциально применима к последнему обращению в теле предложения (обращению к ***A'***). При этом для вычисления цели ***A'*** может повторно использоваться область памяти, выделенная для вычисления порождающей цели ***A***.

Ключевая предпосылка для применения подобной оптимизации состоит в том, что с момента редуцирования цели ***A*** с помощью данного предложения до момента редуцирования последней цели ***A'*** не осталось нерассмотренных вариантов выбора. Другими словами, не осталось иных предложений для редуцирования цели ***A*** и не осталось вариантов вычисления целей левее ***A'***.

Хвостовая рекурсия

Таким образом, если рекурсивное правило не генерирует указателей отката и последняя подцель правила является рекурсивным вызовом самого правила, то Пролог устранит дополнительные расходы, вызываемые рекурсией.

Другими словами, хвостовую рекурсию можно определить следующим образом:
если рекурсивный вызов предиката A' эквивалентен исходному A, то рекурсия называется **хвостовой**.

Хвостовая рекурсия

Рассмотрим достаточно простую программу, представляющую собой хвостовую рекурсию, которая автоматически заменяется итерацией.

```
count (N) :- write (N),
             nl,
             NewN is N+1,
             count (NewN).
```

Данная программа печатает бесконечную последовательность чисел, начиная с числа, заданного в исходной цели.
В этом правиле тело состоит из четырех подцелей, последняя из которых представляет собой рекурсивный вызов самого правила. Так как первые три подцели являются встроенными детерминированными предикатами, то эквивалентность исходной цели ***count (N***) и подцели ***count (NewN)*** очевидна.

Примеры нехвостовой рекурсия

```
badcount1 (X) :- write (X), nl,
                 NewX is X + 1,
                 badcount1(NewX),
                 nl.
```

```
badcount2 (X) :- write (X), nl,
                 NewX is X + 1,
                 badcount2 (NewX).
badcount2(X):- X < 0,
                 write ("X is negative.").
```

```
badcount3 (X) :- write (X), nl,
                 NewX is X + 1,
                 check (NewX),
                 badcount3 (NewX).
check (Z) :- Z >= 0.
check (Z) :- Z < 0.
```

Приведение к хвостовой рекурсии

Чтобы рекурсия стала хвостовой необходимо, прежде всего, сделать **рекурсивный вызов последней подцелью** в теле правила.

Простая перестановка подцелей не всегда дает желаемого результата, так как часто необходимо сохранять промежуточные результаты, возникающие на каждой итерации. Для сохранения промежуточных результатов процедуры Пролога дополняются аргументами, называемыми ***накопителями***.

Обычно промежуточное значение соответствует результату вычисления при завершении итерации. Это значение сопоставляется результирующей переменной с помощью единичного предложения процедуры.

Накопители являются логическими переменными, а не ячейками памяти. В процессе итерации передается не адрес, а значение. Так как переменные обладают свойством "одноразовой записи", то измененное значение передается каждый раз.

Итерационное вычисление факториала

```
factorial (N, N, FactN, FactN) :- !.

factorial (I, N, P, FactN) :-
                                NewI is I + 1,
                                NewP is P * NewI,
                                factorial (NewI, N, NewP, FactN).

?- factorial (0, N, 1, FactN).
```

В этой программе используются два накопителя:
переменные ***I*** и ***P.***

Первый аргумент правила ***I*** соответствует счетчику цикла.
Третий аргумент правила ***P*** используется для накопления текущего значения произведения (факториала).