Лекция № 2

Основы логического программирования

The basics of logical programming

Основы логического программирования

Предпосылки

В начале 60-х гг. XX века Джон Робинсон реализовал на ЭВМ метод резолюции (исчерпывающий метод доказательства теорем).

Затем в начале 70-х гг. Роберт Ковальски (Robert Kowalski) разработал теорию использования логики в качестве языка программирования.

Он показал, что аксиому

A, если B_1 и B_2 и … и B_n

можно рассматривать как процедуру рекурсивного языка программирования. В этом случае ***A*** является заголовком процедуры, а набор B_i — телом процедуры.

В дополнение к декларативному пониманию

*"**A*** истинно, **если** истинны все B_i*“*

это утверждение допускает и следующее понимание:

"для решения (выполнения) ***A*** следует решить (выполнить)

B_1 и B_2 и … и B_n".

Основы логического программирования

В это же время Алэн Колмероэ (Alain Colmerauer) и его группа создали в университете Марсель-Экс (Франция) программу, написанную на Фортране и предназначенную для доказательства теорем. Эта программа, названная Prolog (от Programmation en Logique), включала в себя интерпретатор Ковальского.

Однако практическое применение язык Prolog получил лишь тогда, когда Дэвид Уоррен (David H. D. Warren) в 1977 г. создал его эффективную реализацию для вычислительной машины DEC-10. После этого Prolog-системы начали создаваться для многих машин.

In 1983, David H. D. Warren designed an abstract machine for the execution of Prolog consisting of a memory architecture and an instruction set. This design became known as the *Warren Abstract Machine* (*WAM*) and has become the de facto standard target for Prolog compilers.

Язык PROLOG

Основными областями применения языка Prolog являются:

- базы данных и базы знаний;
- экспертные системы и исследования в области искусственного интеллекта;
- общение с ЭВМ на естественном языке (естественно-языковые интерфейсы, машинный перевод, вопросно-ответные системы);
- построение планов действий роботов;
- составление расписаний;
- быстрое прототипирование прикладных программ;
- написание компиляторов, конверторов программ с одного языка в другой;
- системы автоматизированного проектирования (САПРы).

Язык PROLOG

Итак, **Пролог** — язык логического программирования (**ПРО**граммирование в **ЛОГ**ике).

В основе языка — результаты по автоматизации доказательства теорем в исчислении предикатов первого порядка.

Главное принципиальное отличие интерпретации программы на Прологе от классической процедуры автоматического доказательства теоремы в исчислении предикатов первого порядка состоит в том, что
(1) **аксиомы в базе данных упорядочены** и
(2) **порядок их следования весьма существен**, так как это учитывается алгоритмом, реализуемым Пролог-программой.

Другим существенным ограничением Пролога является то, что
(3) **в качестве логических аксиом используются формулы ограниченного класса — дизъюнкты Хорна**. Однако при решении многих практических задач этого достаточно для адекватного представления знаний.

Язык PROLOG

Дизъюнкт Хорна — дизъюнкция литералов с не более чем одним положительным литералом.

Пример дизъюнкта Хорна:

$$\neg P_1 \vee \neg P_2 \vee \ldots \vee \neg P_k \vee Q$$

Эта формула может быть преобразована в эквивалентную формулу с импликацией:

$$(P_1 \wedge P_2 \wedge \ldots \wedge P_k) \rightarrow Q,$$

Как было сказано выше дизъюнкт интерпретируется в обратном порядке: для решения (выполнения) Q следует решить (выполнить) $P_1, P_2, \ldots P_k$.

Поэтому используется обратный порядок записи:

$$Q \leftarrow (P_1 \wedge P_2 \wedge \ldots \wedge P_k)$$

Язык PROLOG

Очевидно, что поиск нужных для доказательства формул — комбинаторная задача и при увеличении числа аксиом число шагов вывода катастрофически быстро растет. Поэтому в реальных системах применяют всевозможные **стратегии, ограничивающие слепой перебор**.

В языке Пролог реализована **стратегия линейной резолюции**, предполагающая использование на каждом шаге в качестве одной из сравниваемых формул отрицание цели (теоремы) или ее «потомка», а в качестве другой — одну из аксиом логической программы.

При этом выбор той или иной аксиомы для сравнения может сразу или через несколько шагов завести в «тупик» Это вынуждает вернуться к точке, в которой производился выбор, чтобы испытать новую альтернативу, и т. д.

Язык PROLOG

Порядок просмотра альтернативных аксиом не произволен — его задает программист, располагая аксиомы в базе данных в определенном порядке.

Кроме того, в Прологе предусмотрены достаточно удобные (а во многих случаях просто необходимые) «встроенные» средства для запрещения возврата в ту или иную точку в зависимости от выполнения определенных условий.

Таким образом, процесс доказательства в Прологе существенно более прост и целенаправлен, чем в классическом «недетерминированном» методе резолюции, за счет применения определенных эвристик.

Язык PROLOG

Пролог-программа состоит из двух частей:

базы данных (соответствует множеству аксиом) и **последовательности целевых утверждений** (или запросов).

Аксиомы базы данных объединяются в подмножества (по общему предикату в левой части), в которых их порядок строго регламентирован, сами же подмножества не упорядочены.

База данных — это множество логических утверждений. По этой причине ее еще называют **базой знаний**.

Утверждения базы данных бывают двух типов:
факты и **правила**.

Факт — аналог аксиомы вида **T→ Q**,

где **T** — тождественно истинный предикат; **Q** — предикат.

Для обозначения факта используется запись вида «**Q.**».

Язык PROLOG

Правило – аналог аксиомы вида

$$P_1 \wedge ... \wedge P_k \rightarrow Q,$$

где P_i (условия) и **Q** (следствие) – предикаты.

В Прологе обычно применяется инверсная запись правила, заканчивающаяся точкой:

$$Q :- P_1, ..., P_к.$$

которое читается так: «цель **Q** удовлетворяется, если удовлетворяются подцели P_1 и ... и $P_к$» или проще:

«**Q, если P_1 и ... и $P_к$**».

Предикат **Q**, стоящий слева от знака «**:-**», называется **заголовком правила**, а предикаты условия составляют **тело правила**.

Особую роль в Прологе играют **целевые утверждения** (или запросы), для записи которых используется следующая нотация:

$$?- R_1, ..., R_n.$$

Язык PROLOG

Предикаты бывают 0-местные (без аргументов) или n-местные.

Если предикат 0-местный, он обозначается одним именем.

Предикат с аргументами обозначается именем, вслед за которым в скобках через запятые записываются все его аргументы.

Имя предиката — последовательность строчных или прописных букв, цифр и символов подчерка, начинающаяся со строчной буквы.

Аргумент предиката — синтаксическая конструкция, называемая термом.

Терм — это либо атом, либо число, либо переменная, либо составной терм (структура).

Примеры предикатов: **likes (beth, X), loves(vincent, mia).**

Язык PROLOG

Традиционно в языке Prolog **атом** вводится так, как выше было введено имя предиката, т.е. как последовательность строчных или прописных букв, цифр и символов подчерка, начинающаяся со **строчной** буквы.

В диалекте SWI Prolog определение атома расширено:

Атом — это:

1. Последовательность строчных или прописных букв, цифр и символов подчерка, начинающаяся со строчной буквы. (*Примеры:* butch, big_kahuna_burger, variant2)
2. Произвольная последовательность символов, заключённая в одинарные кавычки. (Примеры: ’ butch’, ’big kahuna burger’, ’The Gimp’, ’Five_Dollar_Shake’, ’&^%&#@$ &*’)
3. Последовательность спецсимволов. (Примеры: @=, ====>, ; , :-)

Язык PROLOG

Числа:

1. Действительные числа: 2.718, π, . . .
2. Целые числа: -2, -1, 0, 1, 2, . . .

Переменная — это последовательность букв и/или цифр, начинающаяся либо с **ПРОПИСНОЙ** буквы, либо со знака «_» (подчеркивания).

Примеры: X, Y, X_526, _tag, List, List24, Head, Tail, Голова, Хвост.

Переменная вида «_» называется **анонимной**.

Анонимные переменные используются в тех случаях, когда значение переменной несущественно (не используется).

Составной терм — это выражение вида $f(t_1,..,t_n)$,

где f — имя функтора (атом),

$t_1,..,t_n$ — термы.

Язык PROLOG

Итак, Пролог-программа состоит из множества фактов и правил, а решение задачи заключается в доказательстве (вычислении) некоторой цели (целей) на основе этого множества.

При этом **все утверждения** в Пролог-программе **заканчиваются точкой**, которая здесь означает, что данное предложение выражает законченную мысль.

Факты и правила в логической программе будут лежать "мертвым грузом" до тех пор, пока мы не зададим цель, которую хотим доказать, или, другими словами, не попросим Пролог подтвердить или вывести какой-нибудь факт.

Выполнение Пролог-программы

Пусть у нас есть множество утверждений:

```
likes (mary, apples).                    /* факт */
likes (beth, X) :-                       /* правило */
        likes (mary, X).
```

и цель, которую нужно вычислить:

```
?-likes (mary, apples).
```

Пролог сопоставит эту цель с фактом, имеющимся в программе, и ответит утвердительно, выдав сообщение **“Yes”**.

Если в качестве цели указать

```
?-likes (beth, apples).
```

Пролог будет сопоставлять термы цели с термами выбранных предикатов программы до тех пор, пока очередное сопоставление окажется неуспешным.

Выполнение Пролог-программы

Пролог всегда пытается сопоставить цель с заголовком правила точно так же, как он сопоставлял цель с фактом. Если ему это удается, то происходит редукция (замена) цели телом правила.

В данном случае Пролог попытается унифицировать цель **likes (beth, apples)** и голову правила **likes (beth, X).**

Пролог-программа:

```
likes (mary, apples).
likes (beth, X) :-
        likes (mary, X).
?-likes (beth, apples).
```

Цель и голова правила имеют сопоставимые предикатные термы и первые объекты. Что касается переменной **X**, она пока не имеет никакого значения, следовательно, она унифицируема с любым объектом (а значит и с **apples**). И это связывание действует до тех пор, пока либо цель не будет вычислена, либо нельзя будет выполнить никакого другого сопоставления с учетом этой унификации.

Выполнение Пролог-программы

В нашем случае будет **X = apples**, т.е. в результате сопоставления предикат **likes (beth, X)** будет иметь вид **likes (beth, apples)**. Таким образом, Пролог успешно сопоставил голову правила и цель, присвоил переменной **X** значение **apples**.

Пролог-программа:

```
likes (mary, apples).
likes (beth, X) :-
        likes (mary, X).
?-likes (beth, apples).
```

Теперь Пролог пытается выполнить условие **likes (mary, X)**, содержащееся в теле правила. Так как X имеет значение **apples**, для доказательства истинности всего правила необходимо доказать подцель **likes (mary, apples)**.

Таким образом, новая задача состоит в том, чтобы проверить, *любит ли Мэри яблоки*. Эта "подзадача" или подцель, была создана самим Прологом как шаг, направленный на решение исходной задачи.

Выполнение Пролог-программы

Итак, подцелью теперь является **likes (mary, apples)**.

Анализируя факты и правила программы, Пролог находит факт **likes (mary, apples)**, который, очевидно, сопоставим с подцелью.

Пролог-программа:

```
likes (mary, apples).
likes (beth, X) :-
        likes (mary, X).
?-likes (beth, apples).
```

Таким образом, подцель **likes (mary, apples)** успешно доказана. Следовательно, правило **likes (beth, X)** истинно, а значит, доказана истинность исходной цели **likes (beth, apples)**.

Если в качестве цели задать **likes (beth, oranges)**, то для программы эта цель будет неуспешной.

Выполнение Пролог-программы

При поиске решений в Прологе используется Бэктрекинг или откат.

Откат – это механизм, который используется для нахождения дополнительных фактов и правил, необходимых для вычисления цели, если текущая попытка вычислить цель (или подцель) оказалась неуспешной.

Пролог использует откат для того, чтобы пробовать новые пути для достижения цели. При этом он использует всю доступную ему информацию, которая в Пролог-программе представлена в виде фактов и правил. В программе могут быть правила, имеющие одинаковые левые части (заголовки), но разные правые части. Цель может быть составной, т.е. состоять из нескольких подцелей, соединенных между собой знаками конъюнкции.

Для иллюстрации отката и внутренней унификации рассмотрим небольшой пример, включающий как факты, так и правила.

Иллюстрация отката

```
     likes (mary, tomatoes).        /* Мэри любит помидоры */
(2) likes (mary, popcorn).          /* Мэри любит жареную кукурузу */
(3) likes (mary, apples).           /* Мэри любит яблоки */
(4) likes (beth, X) :-              /* Бет любит то,      */
          likes (mary, X),          /* что любит Мэри, */
          fruit (X).                /* если это фрукт.    */
(1) likes (beth, X) :-              /* Бет любит то,      */
          likes (mary, X),          /* что любит Мэри, */
          X = popcorn.              /* если это жареная кукуруза */
    fruit (pears).                  /* груши – это фрукты */
    fruit (apples).                 /* яблоки – это фрукты */
  vegetable (tomatoes).             /* помидоры – это овощи */
  vegetable (cucumbers).            /* огурцы – это овощи */

?- likes (beth, X).
```

Выполнение Пролог-программы

Пролог пытается вычислить цель, сопоставляя слева направо термы предикатов и аргументов цели с соответствующими элементами в фактах и заголовках правил. При этом некоторые подцели могут быть неуспешными, поэтому Прологу требуется способ "**запоминания точек**", в которых он может продолжить альтернативные попытки найти решение.

Прежде чем попробовать один из возможных путей доказательства подцели, Пролог помещает в программу специальный "**указатель**", определяющий точку, в которую может быть выполнен откат к альтернативному пути доказательства, если выбранный путь окажется неудачным. По мере того как Пролог успешно заканчивает свои попытки вычисления подцелей слева направо, он расставляет такие указатели во всех точках программы, в которых существуют альтернативы. Если некоторая подцель оказывается неуспешной, то выполняется откат к ближайшему указателю. С этой точки будет сделана попытка найти другое решение для неуспешной цели.

Методы организации выполнения Пролог-программ

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

fail — тождественно ложный предикат.

Выполнение предиката **cut** (**!**) 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.

Его можно интерпретировать следующим образом: **cut** запрещает использовать все альтернативные утверждения, лежащие ниже, а также все альтернативные выводы конъюнкции целей, находящиеся левее **cut** в утверждении; **cut** не влияет на выполнение целей, находящихся правее в утверждении. На этом участке может быть построено несколько решений и, естественно, допустим возврат.

Отрицание в Прологе

В языке Пролог используется ограниченная форма логического отрицания — **отрицание по невыполнимости (negation as failure)**.

Это понятие базируется на **гипотезе о замкнутости мира**. Суть ее сводится к предположению об истинности отрицания некоторой цели (утверждения), если не существует возможности доказательства этой цели.

Отрицание по невыполнимости достаточно просто определить с помощью уже введенных средств языка. Для этого достаточно воспользоваться предикатами **cut** и fail:

not(G):— G, !, fail.

not(G).

Как правило, большинство реализаций Пролога содержат предикат ***not*** в качестве системного.

Отрицание в Прологе

Отрицание является одной из фундаментальных логических операций, без которой невозможен полноценный логический вывод.

Для реализации отрицания в Пролог введен предикат ***not***. Он имеет формат:

not *(<утверждение>).*

Этот предикат истинен, если *<утверждение>* представляет собой цель, оказавшуюся при доказательстве неуспешной.

Отрицание в Прологе

Пусть, например, мы имеем предикат ***country/1***, служащий для задания страны Европы, и предикат ***border/2***, служащий для представления пар граничащих стран, а также набор утверждений

country ("Франция").

country ("Германия").

country ("Италия").

country ("Испания").

border ("Франция", "Германия").

border ("Франция", "Италия").

border ("Франция", "Испания").

Теперь, если мы захотим узнать все пары не граничащих между собой стран, то можем это сделать с помощью следующей цели:

?-country (C1), country (C2), not (border (C1, C2)).

Отрицание в Прологе

Следует заметить, что предикат ***not*** в Прологе не полностью соответствует отрицанию в математике, поэтому его нужно применять с осторожностью.

Например, если задать цель ***not (country ("Бельгия"))***, то система ответит утвердительно.

Это происходит потому, что система не доказывает данную цель напрямую. Вместо этого она пытается доказать противоположное утверждение, и если его доказать не удается, то считает что поставленная цель успешна.

Такое рассуждение основано на **предположении о замкнутости мира.** В соответствии с этим постулатом все, что в мире существует, либо явно указано в программе, либо может быть из нее выведено. И наоборот, если какое-либо утверждение не содержится в программе и не может быть из нее выведено, то оно не истинно, и, следовательно, истинно его отрицание.