Тема 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 своё 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 неэффективен 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 будем использовать GNU Prolog (www.gprolog.org, версия 1.4.5 или выше). В нём 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).
```

Всё. Программа, состоящая из одной строки кода.

Вопрос:

```
?-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 вопросы.

```
?-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 вопрос?

```
?-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 вопрос «Кто является прародителем luk?»

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

Какой будет ответ?

Аналогично можно написать вопрос «Кто является правнуками tom?»

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

Таким образом:

- В Пролог легко определяются отношения зависимости. Например, родственные связи. Пользователь может легко выяснить какие именно зависимости определены в программе.
- Программы Пролога описываются в терминах **правил** (**фактов**).
- В качестве аргументов вопросов могут выступать как конкретные объекты, так и переменные.
- В результате выполнения вопроса 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 вопрос:

```
?-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:

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

Важно то, что Прологу абсолютно безразлично как именно Вы определите пол. Так как для него, что parent, что male, female, что sex одинаково непонятные 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 GNU Prolog 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 удаётся доказать тело правила.

Рассмотрим вопрос:

```
?-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 вопроса на языке Пролог, если существует несколько вариантов ответов, то для просмотра следующего варианта используют клавишу «точка с запятой».

Основные конструкции. ВОПРОСЫ (цели).

«Существует ли такое значение X, что A(X) истино ?»

?- A(X)

Для того чтобы программа, написанная на языке Пролог, начала работу, к ней нужно обратиться с запросом.

В этом примере смысл запроса (вопроса) рассматривается как утверждение о существовании такого значения X=a, при котором A(a) истинно. В логике подобные утверждения записываются с помощью кванторов существования.

Однако, в логическом программировании кванторы существования обычно не пишутся, но подразумеваются.

Вопрос может быть простым (состоящим из одной цели) или составным (состоящим из нескольких целей). Выполнение программы заключается в доказательстве целей, входящих в запрос. Программа считается успешно выполненной (завершенной), если доказаны цели, из которых состоит запрос.

Основные конструкции. ПРЕДИКАТЫ.

Предикат - это имя свойства или отношения между объектами с последовательностью аргументов.

При записи факта или цели с использованием некоторого предиката сначала записывается имя предиката, а затем в скобках, через запятую, его аргументы.

Примеры:

black (cat).

railway (novosibirsk, omsk).

Число аргументов предиката называется арностью предиката и обозначается black/1 (предикат black имеет один аргумент, его арность равна единице). Предикаты могут не иметь аргументов, арность таких предикатов равна 0.

Основные конструкции. ПЕРЕМЕННЫЕ.

Имя переменной должно начинаться с заглавной латинской буквы или символа подчеркивания, после которых могут следовать латинские буквы, цифры или символы подчеркивания.

Текущая цель: X=5

Если переменная X имеет значение (например, равное 6), то оператор равенства = работает как оператор сравнения. Если же переменная X свободна (не имеет никакого значения), то оператор равенства = работает как оператор присваивания переменной этого значения.

Работа с переменными в Прологе достаточна своеобразна. Если в других,

алгоритмических, языках программирования значение переменной, которое было ей присвоено, не изменяется до тех пор, пока не будет выполнено переприсваивание значения, то в Прологе переменная может получить некоторое значение в процессе поиска решения и потерять его, когда начнется поиск нового решения.

Переменная, не имеющая значения, называется **свободной**, переменная, имеющая значение - **конкретизированной**.

В Прологе нет оператора присваивания значения переменной, его роль выполняет оператор равенства =.

При этом совершенно неважно, слева или справа от знака равенства находится имя переменной. Главное, чтобы она была свободной. С точки зрения программы на Прологе следующие две строки совершенно одинаковы:

…, X=5, …

…, 5=X, …

Самое важное, чтобы переменная X не имела значения.

Вычисление в GNU Prolog производится предикатом **is**.

?- X is 2+2

X=4

Yes

У предиката is в левой части не унифицированная переменная, а в правой – выражение, в котором ВСЕ переменные унифицированы.

Из вышесказанного вытекает следующая особенность использования переменных в Прологе, нельзя записывать вот так:

…, X is X+5, …

В любом случае такая цель будет ошибочной. Действительно, если переменная X имеет, например, значение равное 10, то предыдущая цель сводится к доказательству цели:

…, 10 is 10+5, …

что, естественно, доказать нельзя.

Если же переменная X свободна, то нельзя к переменной, не имеющей никакого значения, прибавить 5, и присвоить эту неопределенность той же самой переменной. Как же тогда быть, если нужно изменить значение переменной? Ответ один - использовать новое имя, поскольку переменная, которая появляется в тексте программы впервые, считается свободной, и может быть конкретизирована некоторым значением:

..., Y is X+5, ...

Анонимные ПЕРЕМЕННЫЕ.

Еще существуют специальные переменные, называемые анонимными. Их имя состоит только из знака подчеркивания. Анонимные переменные используются в случаях, когда значение переменной несущественно, но переменная должна быть использована.

На слайде факты описывают родителей и их детей (первый аргумент - имя родителя, второй - имя ребенка). Теперь, если нужно узнать только имена родителей, но не нужны имена детей, к программе можно обратиться с внешним запросом, использовав анонимную переменную:

Пример:

parent('Владимир', 'Михаил').
parent('Владимир', 'Светлана').
parent('Анна', 'Михаил').
parent('Анна', 'Светлана').

Вопрос: Вопрос:
?- parent(Person,_). *?- parent(Person,Child).*
Решение (избыточное): Решение
Person='Владимир' *Person='Владимир', Child='Михаил'*
Person='Владимир' *Person='Владимир', Child='Светлана'*
Person='Анна' *Person='Анна', Child='Михаил'*
Person='Анна' *Person='Анна', Child='Светлана'*

Интерпретация правил логической программы.

нравятся_взаимно(X,Y) :- нравится(X,Y), нравится(Y,X).

Декларативная интерпретация правила

Введенное нами правило следует понимать так: "Для всех X и Y, **если** X - нравится Y и Y нравится X, **то** X,Y - нравятся взаимно".

Процедурная интерпретация правил

"Для ответа на вопрос нравятся ли X,Y взаимно друг другу, необходимо ответить на вопрос X нравится ли Y и вопрос Y нравится ли X". Каждое действие, связанное с ответом на вопрос, можно рассматривать как вызов процедуры, имя которой определяется функтором терма, а параметры - его аргументами.

Второй способ толкования правил предполагает расчленение его на простые компоненты в соответствии с действиями, выполнение которых необходимо для получения результата.

Каждое действие, связанное с ответом на вопрос, можно рассматривать как вызов процедуры, имя которой определяется функтором терма, а параметры – его аргументами.

Такое толкование называют процедурной интерпретацией правил. Процедурная интерпретация позволяет рассматривать выполнение логических программ с алгоритмической точки зрения, связывая его с последовательным вызовом процедур.

Правила вывода.

Обобщенное правило вывода modus ponens:

"Из B и A:-B следует A"

Все переменные правила связаны квантором всеобщности, поэтому правило A':- B_1' ,B_2',..., B_n' может быть примером правила A:- B_1 , B_2 ,..., B_n , если существует подстановка Q , такая, что AQ = A' и B_iQ = B' для всех $1 < i < n$.

ПРАВИЛО ВЫВОДА:

Из правила R = (A:- B_1 , B_2 ... B_n) и фактов B_1' , B_2' ,... B_n' выводится A' при условии, что A':- B_1' , B_2' ,..., B_n' является примером правила R.

Цель G является логическим следствием программы P , если в P найдется правило с основным примером A :- B_1, B_2 ,..., B_n , где $n > 0$, таким, что B_1 , B_2 ,..., B_n являются

логическими следствиями P и A является примером G.

Заметим, что цель G следует из программы P тогда и только тогда, когда G может быть выведена из P с помощью конечного числа применений обобщенного правила modus ponens.

В общем случае процедура получения ответа на вопрос отражает определение логического следования. В основе этой процедуры лежат многократно повторяемые действия поиска в программе правила, соответствующего текущей цели, и поиска подстановки для построения примера цели.

Выполнение логической программы.

Логическая модель, построенная на логике предикатов первого порядка, предполагает в общем случае пропозициональный взгляд на знание, когда в качестве элемента знания рассматривается некоторое утверждение и, соответственно, элементарное знание представляется атомарной формулой языка логики предикатов. Процесс вывода строится как последовательность шагов получения целевой формулы с помощью *правил вывода*, к которым относятся следующие классические правила:

1. Правило Modus Ponens:

$$\frac{A,\ A \Rightarrow B}{B}$$

если A – выводима и A влечет B, то B – выводимая формула.

2. Цепное правило:

$$\frac{A \Rightarrow B,\ B \Rightarrow C}{A \Rightarrow C},$$

если формулы $A \Rightarrow B$ и $B \Rightarrow C$ выводимы, то выводима и формула $A \Rightarrow C$.

3. Правило подстановки: если формула A(x) выводима, то выводима и формула A(B), в которой все вхождения литерала x заменены формулой B.

4. Правило резолюций: если выводимы формулы двух дизьюнктов, имеющих контрарную пару, $A \vee c$ и $B \vee \neg c$, то выводима формула дизьюнкта $A \vee B$, полученного из данных двух удалением контрарной пары.

Можно построить процесс вывода с использованием только правил подстановки и резолюции, так как правило Modus Ponens и цепное правило можно рассматривать как частные случаи правила резолюций, интерпретируя формулу $A \Rightarrow B$ как дизьюнкт $\neg A \vee B$.

Механизм вывода Пролога как раз и построен с использованием данной модели и порождает, по существу, процесс поиска в глубину.

Логическая программа начинает выполняться после того, как она вместе с вопросом подается на вход интерпретатора. Интерпретатор представляет собой программу, способную строить логические выводы, как правило, начиная с заданного вопроса, т.е. методом «сверху вниз". Если интерпретатору удается доказать выводимость вопроса из логической программы, то он выводит сообщение об удачном окончании и значения переменных, для которых получено решение. В противном случае он выводит сообщение о неудаче.

Основой построения интерпретатора является процедурная интерпретация правил.

Учитывая это обстоятельство, вместо термина логический вывод введем термин вычисление. Для того, чтобы определить этот термин, нам требуется ввести ряд новых понятий.

В общем случае вопрос представляет собой конъюнкцию термов. Доказательство выводимости из программы конъюнкции сводится к доказательству выводимости каждого элемента конъюнкции отдельно. В свою очередь доказательство выводимости терма из программы предусматривает поиск в программе правила, в заголовке которого используется тот же функтор, что и в рассматриваемом терме, а затем выполнение унификации. Если унификация проходит успешно, то можно переходить к доказательству выводимости тела правила, которое также представляет собой конъюнкцию термов.

Резольвентой называется конъюнкция целей, которую следует доказать на рассматриваемом шаге выполнения программы.

Редукция – это *операция*, связанная с заменой цели G телом того примера правила, из программы P, заголовок которого совпадает с данной целью.

Вначале резольвента совпадает с вопросом. Если после удачной унификации очередной цели, эту цель заменить телом выбранного правила, то получим резольвенту для очередного шага выполнения программы.

При этом заменяемая цель при редукции называется **снятой**, а добавляемые цели - **производными**.

Напомним, что переменные в каждом предложении связаны квантором всеобщности. Это означает, что значения переменных определены в области действия квантора и должны теряться при выходе из этой области. Такие переменные в программировании называют локальными. Чтобы избежать путаницы при использовании переменных с одинаковыми именами в разных правилах, условимся переименовывать переменные всякий раз, когда предложение выбирается для выполнения редукции. Среди новых имен не должно быть имен, ранее использованных при вычислении.

Иллюстрация механизма управления выполнением логической программы.

Иллюстрацией данного механизма управления служит следующий фрагмент программы экспертной системы, в котором в диалоге с пользователем определяется, что задуманное им животное – зебра. Для положительного заключения сначала проверяется, относится ли животное к травоядным и не является ли хищником. Для проверки того, что животное травоядное, на следующем уровне проверяется принадлежность к млекопитающим. И на последнем уровне данная принадлежность проверяется по наличию фактов 'Имеет волосы' или 'Может давать молоко' в базе данных или из ответов пользователя (реализация в примере не показана).

```
animal_is('зебра') :- it_is('травоядное'),
                      \+(it_is('хищник')),        %\+ означает not
                      positive('Имеет','темные полосы').
it_is('травоядное') :- it_is('млекопитающее'),
                      positive('Имеет','копыта').
it_is('травоядное') :- it_is('млекопитающее'),
                      positive('Может','есть траву').
it_is('хищник') :- positive('Может','есть мясо').
```

```
it_is('хищник') :- positive('Имеет','зубы'),
                   positive('Имеет','клыки').
it_is('млекопитающее') :-
                    positive('Имеет','волосы').
it_is('млекопитающее') :- positive('Может','давать молоко').
```

3 ситуации, требующие уточнения.

Первая ситуация возникает вследствие неопределенности выбора из резольвенты очередной цели для редукции. Чтобы внести определенность в работу интерпретатора обычно используют правило выбора, согласно которому в качестве очередной цели берется всегда **самая левая цель резольвенты**. Это правило часто называют *стандартной стратегией выбора цели*.

Вторая ситуация возникает оттого, что в определении не оговорены порядок поиска в программе P правила C_i, функтор в заголовке которого совпадает с функтором цели G_i . Эта неопределенность обычно снимается за счет того, что поиск осуществляется путем последовательного **просмотра правил в порядке их написания в программе**. Такой способ поиска называют *стандартной стратегией поиска*.

Третья ситуация обусловлена тем, что в программе могут содержаться несколько правил с одинаковыми функторами в заголовках, которые совпадают с функтором цели. Такие правила называют **альтернативными**. Если обозначить совокупность альтернативных правил $C_1,C_2,...,C_n$,то невозможность, например унификации цели G с заголовком правила C_i не означает еще невозможность доказательства вывода G из программы. Заключение о неудаче можно сделать только на основании невозможности унификации цели G ни с одним альтернативным правилом. Таким образом, при работе с альтернативными правилами необходимо предусмотреть действия, обеспечивающие повторные попытки унификации текущей цели с альтернативными правилами. Такие действия, обеспечивающие повторения называют возвратом или **бэктрекингом**.

Если унификация цели G_i оказывается невозможной для последнего альтернативного правила C_m, то необходимо вернуться к цели, рассмотренной на предыдущем шаге вычисления G_{i-1} и попытаться доказать ее выводимость, используя другое альтернативное правило (i-1)-го шага.

В общем случае возврат может произойти к любой из целей G_k, $k<i$, из которой с помощью нескольких последовательных редукций была получена цель G.

Выполнение возврата к предшествующей цели предполагает уничтожение значений переменных, полученных в результате унификации при выводе цели G_i из $G_к$.