ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ. ЯЗЫК PROLOG

Список рекомендуемой литературы

1. *Шрайнер П.А. Основы программирования на языке Пролог. Курс лекций. Учебное пособие — М.: нтернет-Ун-т Информ. Технологий, 2005.*
2. А.Н. Адаменко, А.М. Кучуков. Логическое программирование и Visual Prolog — СПб.: БХВ-Петербург, 2003.
3. Братко И. Программирование на языке Пролог для искусственного интеллекта. - М.: Мир, 1990.

В октябре 1981 года Японское министерство международной торговли и промышленности объявило о создании исследовательской организации — Института по разработке методов создания компьютеров нового поколения (Institute for New Generation Computer Technology Research Center). Целью данного проекта было создание систем обработки информации, базирующихся на знаниях и переходу к ЭВМ пятого поколения. Основная суть качественного перехода к пятому поколению ЭВМ заключалась в переходе от обработки данных к обработке знаний, отойдя при этом от фон-Неймановской архитектуры компьютеров. В 1991 году предполагалось создать первый прототип компьютеров пятого поколения (полностью задача не реализована). Этот проект послужил импульсом к развитию нового витка исследований в области искусственного интеллекта и вызвал взрыв интереса к логическому программированию. Думается, что и в настоящее время Prolog остается наиболее популярным языком искусственного интеллекта в Японии и Европе (в США, традиционно, более распространен другой язык искусственного интеллекта — язык функционального программирования Лисп).

Цель создания языка логического программирования – попытка автоматизировать выполнение логического вывода. Очевидно, что логическое программирование основано на математической логике, в которой используется язык формул. Формула в матлогике строится из высказываний и предикатов, объединенных знаками операций – конъюнкция, дизъюнкция, отрицание (базис) с кванторами: всеобщности и существования. (**Вопрос**: что такое высказывание и предикат?) Упрощение формул матлогики по определенным законам позволяет понять является ли формула тавтологией или нет, т.е. сделать логический вывод.

Математическая база Prolog – языка логического программирования это принцип резолюции, опубликованный (доказанный) лишь в 1965 г. (Дж. Робинсон). Принцип резолюции - это метод автоматического поиска доказательства теорем в исчислении предикатов первого порядка. Первая версия языка Prolog была написана на яз.Fortran в 1973г. в Марсельском универ-те.

Логическое программирование, которое принципиально работает не с данными, а со **знаниями**, поддерживается декларативными языками программирования. Prolog – это декларативный язык программирования. Он основан на системе правил, декларированных в программе, - знаниях. В отличие от императивного программирования, порядок действий в системе логического программирования определяется системой правил (т.е. особым способом обработки совокупности правил – алгоритмом **унификации**), а не определяется порядком записи правил в тексте программы.

Программируя в **императивном** стиле, программист должен объяснить компьютеру **как** нужно решать задачу (порядок действий). Программируя в декларативном стиле, программист должен описать **что** нужно решать, т.е. предметную область. Программа на **декларативном языке** представляет собой совокупность утверждений, описывающих

фрагмент предметной области (знания о предметной области) или сложившуюся ситуацию, а не порядок поиска решения.

Основным элементом языка является **терм. Терм – это:**

1. **Константа:**
   - Число (целое, вещественное),
   - Символьный атом (комбинация символов латинского алфавита, цифр и символа подчеркивания, начинающаяся со строчной буквы: aA, ab_2), используется для обозначения конкретного объекта предметной области или для обозначения конкретного отношения,
   - Строка: последовательность символов, заключенных в кавычки,
2. **Переменная:**
   - Именованная – обозначается комбинацией символов латинского алфавита, цифр и символа подчеркивания, начинающейся с прописной буквы или символа подчеркивания (X, A21, _X),
   - Анонимная - обозначается символом подчеркивания (_),
3. **Составной терм:**
   - Это средство организации группы отдельных элементов знаний в единый объект, синтаксически представляется: **f(t1, t2, …,tm)**, где f - функтор (функциональный символ) , t1, t2, …,tm – термы, в том числе и составные (их называют аргументами), (например: likes(judy, tennis) – знание о том, что judy любит tennis_ или еще, например: book(author(tolstoy, liev), war_and_peace) и т.д.). Аргументом или параметром составного терма может быть константа, переменная или составной объект. Число аргументов предиката называется его **арностью** или **местностью**. Составные термы с одинаковыми функторами, но разной арности, обозначают разные отношения.

С помощью термов и более сложных конструкций языка Prolog – **фактов** и **правил** «описываются» знания о предметной области, т.е. **база знаний**. Используя базу знаний, система Prolog будет делать логические выводы, отвечая на наши **вопросы**. Таким образом, **программа на Prolog представляет собой базу знаний и вопрос**.

База знаний состоит из предложений - CLAUSES (отдельных знаний или утверждений**): фактов и правил**. Каждое предложение заканчивается точкой.
Предложения бывают двух видов: факты и правила. Предложение более общего вида – **правило** имеет вид:

$$A :- B_1, \ldots , B_n.$$

A называется **заголовком правила**, а B_1,..., B_n – **телом правила**.

Факт – это частный случай правила. Факт – это предложение, в котором отсутствует тело (т.е. тело пустое).
Причем, A, B_1,... , B_n – это термы; символ "**:-**" это специальный символ-разделитель.

Заголовок содержит отдельное **знание** о предметной области (составной терм), а тело содержит условия истинности этого знания. Правило называют условной истиной, а факт, не содержащий тела – безусловной истиной.

Заголовок, как составной терм f(t1, t2, …,tm) , содержит **знание о том**, что между

аргументами: t1, t2, …,tm существует **отношение** (взаимосвязь, взаимозависимость). А **имя** этого **отношения** – это **f**. Например: likes(judy, tennis)

Если это факты (или правила) программы, то это записывается в разделе - CLAUSES:

likes(judy, tennis).
mother("Наташа", "Даша").

И далее другие факты. Программа может состоять только из фактов. Программа может состоять из фактов и правил – знаний. Одно знание может быть записано с помощью нескольких предложений.

Третьим специфическим видом предложений Prolog можно считать **вопросы**. Вопрос состоит только из тела – составного терма (или нескольких составных термов). Вопросы используются для выяснения выполнимости некоторого отношения между описанными в программе объектами. Система рассматривает вопрос как цель, к которой (к истинности которой) надо стремиться. Ответ на вопрос может оказаться логически положительным или отрицательным, в зависимости от того, может ли быть достигнута соответствующая цель.

Если составные термы, факты, правила и вопросы не содержат переменных, то они называются основными. Составные термы, факты, правила и вопросы в момент фиксации в программе могут содержать переменные, тогда они называются неосновными. Переменные в момент фиксации утверждений в программе, обозначая некоторый неизвестный объект из некоторого множества объектов, не имеют значения. Значения для переменных могут быть установлены Prolog-системой только в процессе поиска ответа на вопрос, т.е. реализации программы.

Программа на Prolog может содержать вопрос в программе (так называемая **внутренняя цель GOAL**). Если программа содержит внутреннюю цель, то после запуска программы на выполнение система проверяет достижимость заданной цели, исходя из базы знаний. Например:

GOAL

likes(judy, reading)

Или:
GOAL

likes(judy, tennis)

Ответ на поставленный вопрос система дает в логической форме – «Да» или «нет». Цель системы состоит в том, чтобы на поставленный вопрос найти возможность, исходя из базы знаний, ответить «Да». Вариантов ответить «Да» на поставленный вопрос может быть несколько. Система может (в нашем случае обучения - должна) быть настроена в режим получения всех возможных вариантов ответа «Да» на поставленный вопрос.

Факты, правила, и вопросы могут содержать переменные. Вообще – переменные предназначены для передачи значений «во времени и в пространстве». Переменные в факты и правила входят только с квантором всеобщности. А в вопросы переменные входят только с квантором существования. Поэтому символы кванторов в программе на Prolog не пишутся. Отметим, что именованные переменные уникальны в рамках предложения, а анонимная переменная – любая уникальна. В разных предложениях может использоваться одно имя переменной для обозначения разных объектов.

В процессе выполнения программы переменные могут связываться с различными объектами – **конкретизироваться**. Это относится только к именованным переменным.

Анонимные переменные не могут быть связаны со значением.

Говорят, что именованная переменная конкретизирована, если имеется объект, который в данный момент обозначает данная переменная (переменная связанная). Иначе переменная – не конкретизированная (переменная свободная). Конкретизация не связана с распределением памяти и присваиванием переменной значения. В логическом программировании все переменные рассматриваются как безтиповые, т.е. в процессе вычисления любая переменная может быть связана с объектом произвольной природы. В логическом программировании поддерживается механизм деструктивной конкретизации переменной. Т.е. используется идея **ре-конкретизации** переменной путем «отката» вычислительного процесса и отказа от выполненной ранее конкретизации. Это реализовано для возможности поиска нового значения для именованной переменной.

Поиск содержательного ответа на поставленный вопрос, с помощью имеющейся базы знаний, фактически заключается в поиске нужного знания, но какое знание понадобится – заранее неизвестно. Этот поиск осуществляется формально с помощью механизма **унификации**, встроенного в систему и не доступного программисту. Упрощенно, процесс унификации можно представить как формальный процесс сравнивания (сопоставления) терма вопроса с очередным термом знания. При этом, знания по умолчанию просматриваются сверху вниз, хотя такой порядок и не очевиден. В процессе сравнивания для переменных «подбираются», исходя из базы знаний, значения (для именованных переменных). И эти подобранные для переменных значения возвращаются в качестве побочного эффекта ответа на поставленный вопрос. Остается только один вопрос: как установить, что с помощью конкретного знания можно подтвердить истинность вопроса и как «подбираются» значения для переменных. Это выполняется по формальным правилам с помощью сопоставления термов

Ведем определения:

Пусть дан терм: $\mathbf{A(X_1, X_2, X_n)}$

Подстановкой называется множество пар, вида: $\{ X_i = t_i \}$,

где X_i – переменная, а t_i – терм.

Пусть $\Theta = \{ X_1 = t_1, X_2 = t_2, \dots, X_n = t_n \}$ – подстановка, тогда результат применения подстановки к терму обозначается: $A\Theta$. Применение подстановки заключается в замене **каждого** вхождения переменной X_i на соответствующий терм.

Терм **B** называется **примером** терма **A**, если существует такая подстановка Θ, что $B=A\Theta$.

Терм **C** называется **общим примером** термов **A** и **B**, если существуют такие подстановки Θ_1 и Θ_2, что $C = A\,\Theta_1$ и $C=B\,\Theta_2$

В Prolog существует понятие процедуры. **Процедурой** называется совокупность правил, заголовки которых имеют одно и то же имя и одну и ту же арность (местность), т.е. это совокупность правил, описывающих одно определенное отношение. Отношение, определяемое процедурой, называется **предикатом**.

Логический вывод. Простейшие правила логического вывода, когда программа состоит только из фактов.

Цель работы программы – определить является ли вопрос логическим следствием программы или нет, что выполняется с применения правил вывода. Правила вывода – это утверждения о взаимосвязи между допущениями и заключениями, которые с позиции

исчисления предикатов верны всегда.

Если программа состоит только из фактов, то может быть только 4 случая:

1. Факты в базе и вопрос – основные. В этом случае используется простое правило: совпадение - вопрос из программы выводится, если в программе есть тождественный факт;
2. Факты в базе основные, а вопрос – неосновной. В этом случае используется простое правило: обобщение (факта) – если есть такая подстановка, что вопрос QΘ логически следует из программы, то и вопрос Q следует из программы.
3. Факты в базе неосновные, а вопрос – основной. В этом случае используется простое правило: конкретизация (факта) – из утверждения P с квантором всеобщности выводится пример PΘ, которым является вопрос.
4. Факты в базе и вопрос – неосновные. Это наиболее общий случай. Для ответа необходимо найти общий пример для вопроса и факта. В этом случае используется два правила: : конкретизация (факта) – строится пример, а затем, с помощью правила – обобщение, из примера выводится вопрос. Например
   Факт: P(Y , b).

   конкретизация➔ P(a, b). обобщение➔ P(a, X)

   Вопрос: P(a , X)

Существенным недостатком в использовании этих правил вывода является необходимость «угадать» подстановку или пример терма. Кроме этого, переменные в факте и в вопросе могут стоять на одной позиции. Поэтому для выполнения логического вывода используется **механизм (алгоритм) унификации**, встроенный в систему.

Унификация – операция, которая позволяет формализовать процесс логического вывода (наряду с правилом резолюции). С практической точки зрения - это основной вычислительный шаг, с помощью которого происходит:

- Двунаправленная передача параметров процедурам,
- Неразрушающее присваивание,
- Проверка условий (доказательство).

В процессе работы система выполняет большое число унификаций. Процесс унификации запускается автоматически, но пользователь имеет право запустить его принудительно с помощью утверждения (немного нарушает форму записей): T1 = T2 . Унификация – попытка "увидеть одинаковость" – сопоставимость двух термов, может завершаться успехом или тупиковой ситуацией (неудачей). В последнем случае включается механизм отката к предыдущему шагу.

Итак, два терма **T1 и T2 унифицируемы успешно, если**:

1) T1 и T2 – одинаковые константы;

2) T1 – не конкретизированная переменная, а T2 - константа или составной терм, не содержащий в качестве аргумента T1. Тогда унификация успешна, причем T1 конкретизируется значением T2;

3) T1 и T2 не конкретизированные переменные. Их унификация успешна всегда, причем T1 и T2 становятся сцепленными переменными – т.е. двумя именами одного "объекта", возможно пока и не установленного. В этом случае при конкретизации одной переменной вторая получает такое же значение.

4) T1 и T2 составные термы. Унификация успешна при выполнении трех условий:

а) T1 и T2 имеют одинаковые главные функторы,

б) T1 и T2 имеют одинаковые арности (количество аргументов),

в) успешно унифицируется каждая пара их соответствующих компонент (аргументов).

Декларативная и процедурная семантика логической программы

В естественном языке каждое слово имеет некоторое значение (смысл). В логической программе смысл каждого имени (идентификатора) определяется набором утверждений, которые описывают его свойства и связи.

В императивных (процедурных) языках программирования **под семантикой** некоторой конструкции языка понимается поведение вычислительной системы при обработке этой конструкции. **Семантика** формулы в логике предикатов связана с приписыванием этой формуле истинностного значения.

Prolog одновременно является: и логическим языком, и языком программирования, реализуемым на императивно работающей технике. Поэтому к нему применимы обе трактовки понятия семантики. В **декларативной** модели – формулировки программы рассматриваются как знания – отношения, взаимосвязи между объектами, сформулированные в виде правил. Для этой модели порядок следования предложений в программе и условий в правиле не важен. Действительно, важна общая совокупность знаний, а не их порядок. **Процедурная** модель рассматривает правила как последовательность шагов, которые необходимо успешно выполнить для того, чтобы соблюдалось отношение, приведенное в заголовке правила. В процедурной трактовке логической программы, т.е. при ее выполнении становится **важен порядок**, в котором записаны предложения в процедурах и условия в предложениях (в теле правил). В идеале стиль программирования должен быть полностью декларативным. Реально порядок следования предложений в программе крайне существенно сказывается на **эффективности** работы системы при выполнении программы. Порядок до такой степени существен, что некоторые утверждения, верные в декларативном смысле, с процедурной точки зрения, могут оказаться неработающей программой!

Как было ранее сказано, Prolog одновременно является: и логическим языком, и языком программирования, реализуемым на императивно работающей технике. Поэтому в «чистом» логическом программировании порядок следования предложений в программе может быть произволен, но, в реальной программе на Prolog оказывается важен порядок процедур, предложений внутри процедуры, а также порядок условий в теле предложений, т.е. вообще порядок любых составных частей текста программы. От порядка предложений и условий в теле правил зависит порядок подбора знаний и порядок, в котором будут находиться ответы на вопросы. Порядок условий влияет на количество проверок, выполняемых программой при решении, объемы памяти, используемой системой Prolog во время работы и др.

Процедурная семантика программы (базы знаний) на Prolog (с декларативной точки зрения нет разницы в каком порядке знания попали в базу, или в Вашу голову– Вы или система обязаны дать ответ на вопрос) состоит в том, что совокупность знаний должна быть использована техникой в некотором порядке. А в каком? А Вы знаете, в каком порядке в Вашей голове анализируются знания? Вы об этом не думаете. Какая разница, в каком – важен вывод. Но, формально, системе надо установить порядок, в котором она должна работать. И такой порядок установлен – один для всех программ, т.е. для системы (хотя это далеко не единственно возможный порядок).

Напомним, что унификация двух термов – это основной шаг доказательства, назначение которого – подобрать нужное в данный момент правило. В процессе работы система выполняет большое число унификаций. **Унификация –** операция, которая позволяет формализовать процесс логического вывода (наряду с правилом резолюции). С практической точки зрения – это отдельный вычислительный шаг работы системы, который может завершиться успехом или тупиковой ситуацией (неудачей). Процесс унификации запускается автоматически, если есть что доказывать, то надо запускать алгоритм унификации. Пользователь имеет право запустить этот процкссс вручную, с помощью утверждения T1 = T2, включенного в текст программы. Остаются вопросы: для

каких термов запускать алгоритм унификации, и что делать дальше в том или другом случае, и как понять это на формальном уровне. Принят следующий порядок.

Общая схема согласования целевых утверждений

Инициация работы системы выполняется заданием вопроса – цели доказательства. Общая схема доказательства вопроса подчиняется следующему принятому жесткому порядку.

Пусть есть некоторая **программа P** и **вопрос G**. Решение задачи с помощью логической программы P начинается с задания вопроса G и завершается получением одного из двух результатов:

* успех согласования программы (базы знаний) и вопроса; в качестве побочного эффекта формируется подстановка, которая содержит значения переменных, при которых вопрос является примером программы (примеров может быть несколько);
* неудача – тупиковая ситуация (вывод делается исходя только из знаний заданной БЗ).

Порядок работы:

Вычисления с помощью конечной логической программы представляют собой пошаговое преобразование исходного **вопроса**. На каждом шаге имеется некоторая совокупность целей - утверждений, истинность (выводимость) которых надо доказать. Эта совокупность называется **резольвентой** - её состояние меняется в процессе доказательства (Для хранения резольвенты система использует стек). Успешное завершение работы программы достигается тогда, когда резольвента пуста. Преобразования резольвенты выполняются с помощью **редукции**.

Редукцией цели G с помощью программы P называется замена цели G телом того правила из P, заголовок которого унифицируется с целью (в заголовке правила зафиксировано знания). Такие правила будем называть сопоставимыми с целью, и система подбирает нужные с помощью алгоритма унификации.

Новая резольвента образуется в два этапа

1. в текущей резольвенте выбирается одна из подцелей (по стековому принципу - верхняя) и для неё выполняется редукция - замена подцели на тело найденного (подобранного, если удалось) правила (а как подбирается правило?),
2. затем, к полученной конъюнкции целей применяется подстановка, полученная как наибольший общий унификатор цели (выбранной) и заголовка сопоставленного с ней правила.

Если для редукции цели из резольвенты был выбран факт из БЗ, то *новая* резольвента будет содержать в конъюнкции на одну цель меньше, т.к. факт – частный случай правила с пустым телом. И, если задан простой вопрос (на первом шаге он попадает в резольвенту) и подобран для редукции факт, то произойдет немедленное его согласование. А если для простого вопроса подобрано правило, то число целей в резольвенте не уменьшится, т.к. цель будет заменена телом подобранного правила.

Механизм отката (backtracking) и дерево поиска решений

При использовании принципа не детерминизма при поиске решения, возможны тупиковые ситуации, для разрешения которых используется механизм отката, кроме этого мы хотим получить все возможные ответы, значит, получив один ответ, надо начать заново, чтобы получить другой ответ.

Для ответа на поставленный вопрос система должна подобрать нужное знание в БЗ, каждое из которых, зафиксировано в заголовке правила. И таких знаний может быть несколько. Причем, неудача при использовании одного такого правила, вовсе не означает неудачу при использовании другого. Поэтому предусмотрена возможность отказа от

сделанного выбора – **backtracking.** Т.е. недетерминизм обеспечивается пошаговым алгоритмом с возвратами – перебором методом «проб и ошибок».

На каждом шаге работы системы – поиске способа (нового) доказательства подцели, система начинает поиск знания с начала базы. И на каждом таком шаге может сложиться одна из трех ситуаций:

1. решение найдено полностью и окончательно, и алгоритм завершен,

2. имеется некоторое число дальнейших альтернативных возможностей, но какая приведет к решению – неизвестно, а система должна выбрать следующее действие формально, т.е. автоматически,

3. решение не найдено, и из данного состояния невозможен переход в новое состояние. В этом случае автоматически включается бэктрэкниг («обратная трассировка»). Происходит возврат к моменту (состоянию), где еще можно сделать другой альтернативный выбор, т.е. к предыдущему состоянию резольвенты и попытке ее преобразовать – пере- согласовать. Альтернативный выбор заключается в использовании другого знания для доказательства цели, а, чтобы не было повторов, система каждый раз помечает выбранное знание, т.е. ранее выбранное правило было уже отмечено. Т.о., при возврате отменяется последняя уже выполненная редукция (восстанавливается предыдущее состояние резольвенты) и система выполняет ре- конкретизацию переменных, которые были конкретизированы на предыдущем шаге. Поэтому, система не должна забывать: какое состояние резольвенты и переменных было ранее. Для этого система использует дополнительную память (а как там организована информация, как бы Вы организовали?), и, хотелось бы, чтобы этой памяти был минимум, а сколько будет возвратов, а, значит, сколько понадобится памяти – зависит от порядка утверждений в БЗ.

Для того чтобы представить себе наглядно порядок работы системы, строят дерево поиска решения. **Дерево поиска решения** – формализм для исследования всех возможных путей вычисления – схема, которую мы можем изобразить для представления себе порядка выбора знаний и последующих действий системы.

Корень дерева - вопрос G. *Вершины дерева* образуют резольвенты, которые в общем случае являются конъюнктивными. Для каждого утверждения программы (правила или факта), заголовок которого унифицируется с выделенной подцелью в резольвенте, имеется ребро. *На ребрах дерева записываются подстановки*, которые формируются в результате унификации выделенной подцели и заголовка сопоставленного ей правила. *Лист* дерева называется успешной вершиной, если резольвента ему соответствующая – пуста и безуспешной вершиной, если резольвента не пуста и нет утверждений в базе знаний, которые удается сопоставить с выделенной в этой вершине подцелью.

Система хранит резольвенту в виде стека, содержащего цели, которые надо будет доказать. Говорят, что при обработке резольвенты, стек растёт влево – т.к. в вершине стека находится левое условие тела выбранного на предыдущем шаге правила. Если резольвента не пуста – запускается алгоритм унификации, а если – пуста, это значит, что получен один, однократный ответ «Да» на поставленный вопрос, после чего включается механизм отката, в попытке найти другое решение с помощью другого знания. При этом, БЗ просматривается сверху вниз.

Управление порядком работы системы

В императивных языках программирования, программист полностью контролирует последовательность вычислений и управляет использованием аппаратных ресурсов. В логических программах система выполняет полный перебор всех возможных вариантов

решения, а возможности управления вычислениями – минимальны. У программиста есть право только изменить порядок следования правил в процедурах и порядок следования условий в теле правила. И этим надо максимально пользоваться.

Но существуют постановки задач, в которых есть бесперспективные пути поиска решений, в чем программист совершенно уверен. Для исключения соответствующих действий из рассмотрения, в Prolog включены два системных предиката: предикат отсечения и предикат fail.

Предикат отсечения **!** (cat) включается в конъюнкцию целей так же, как и другие предикаты и отсекает в определенном случае бесперспективные пути доказательства.

Предикат fail принудительно включает механизм отката.

В современных версиях языка существуют и другие системные предикаты, позволяющие создавать интерфейс программы, но они не влияют на логику работы программы, которую мы изучаем.

Представление списков

В силу особенностей задач искусственного интеллекта, в которых предполагается накопление и пере - структурирование знаний, целиком и полностью оправдано использование списков. При этом используются списки, элементы которых сами могут являться списками. Списки, как рекурсивно определенные структуры (определение давалось ранее), удобно обрабатывать рекурсивно. При обработке всех элементов списка, требуется уметь «увидеть» признак конца списка – пустой список (в Prolog []).

Исторически первая синтаксическая форма представления списков как термов (в Prolog) была основана на использовании бинарного функтора “точка” (точечная пара). Чтобы такая пара была списком, достаточно, чтобы 2-ой элемент был списком (пустым), первый элемент может иметь любую структуру. Синтаксическая форма записи списка в виде терма: **“.”(1, [ ])** или **“.”(a, “.”(b, [ ])) .**
Облегченный (лаконичный) синтаксис Prolog позволяет эти списки записать:

[1, []] или **[1]** , [a, b, []] или **[a, b]** . В Prolog – символы **[…]** – это синтаксис (признак) особого терма, ничего, кроме термов Prolog не использует! .
Элементы списка для Prolog это аргументы терма. С такими термами система работает как обычно!. Пустой список (**[ ]**) не образует никакой структуры, по сути, он является особым атомом, обозначающим пустой список. Система Prolog неявно подразумевает его в качестве терминального элемента списка.

В Prolog существует более общий способ доступа к элементам списка. Для этого используется метод разбиения списка на начало и остаток. **Начало** списка – это группа первых элементов, **не менее одного**. Остаток списка – обязательно список (может быть пустой). Для разделения списка на начало, и остаток используется вертикальная черта (|) за последним элементом начала. **Остаток это всегда один** (простой или составной) **терм**. Свойство атома: его нельзя разбить на части. Если начало состоит из одного элемента, то получим: голову и хвост. При таком способе разбиения – список можно разбить на части не единственным способом:
[a, b, c, d] ~ [a, b, c, d | []] ~ [a, b, c | [d]] ~ [a, b | [c, d]] ~ [a | [b, c, d]] других способов разбиения этого списка нет!

То, что второй элемент списка (хвост – остаток) должен быть списком – это свойство списка, а не функтора «точка». Поэтому интерпретатор не обнаруживает ошибки в записи: [a | b]. Ошибка может быть обнаружена только при попытке расчленить хвост. Т.е. не всякая конструкция, внешне использующая списковую нотацию, представляет собой список.

В списковых структурах переменными могут быть представлены и голова и хвост:
[X | Y] или
отдельные элементы: **[**a, X, b | **[Y** | []**]]**.

Каждая из переменных может быть конкретизирована термом, произвольной структуры. В результате, мы можем получить списковую или нет структуру. При этом, каждая переменная может обозначать только один объект (терм). Система Prolog должна подобрать такие термы для последующей конкретизации ими переменных. Как ранее было сказано, она это делает с помощью алгоритма унификации, пытаясь так разделить список на начало и остаток, чтобы унификация была успешна. Списки могут являться аргументами других термов: вопросов, заголовков правил, термов тела правил. Таким образом, очевидно, что система должна уметь сравнивать списки. Естественно, что бессмысленно сравнивать терм – список с термом, например студент. Имеет смысл сравнивать списки со списками.

Примеры унификации списков:

- [X] = [] – унификация невозможна;
- [X] = [a | []] → [X] = [a] – унификация успешна, X = a;

- [X] = [a, b, c] – унификация невозможна;
- [X, b] = [a, b] → [X | [b]] = [a | [b]]] – унификация успешна, {X = a};
- [[X], b] = [a, b] – унификация невозможна, т.к. разная структура 1-х элементов;
- [a | Y] = [a, b, c] → [a | Y] = [a | [b, c]] – унификация успешна {Y = [a | [b, c]]}
- [a | Y] = [a, b | [c]] → [a | Y] = [a | [b, c]] – унификация успешна {Y = [b, c]}

Методы обработки списков

Для обработки списка: его самого или его элементов – системе требуется знать как это сделать, т.е. требуется знание.

Приведем несколько примеров.

1). Предикат list, который позволит проверить: является ли аргумент списком.

Для распознавания, является ли терм L списком, создадим знание о том как это сделать – предикат **list(L):**

По определению **L –**является списком: если он пустой (правило I), или, если он не пустой, но его хвост – является списком (правило II), т.е. это надо проверить. Очевидно, что второе правило определяет рекурсию. После задания вопроса, например list([1, 2, 3]) – с помощью успешной унификации терма вопроса и терма заголовка правила (list([1, 2, 3]) = list(L)), L будет конкретизировано списком [1, 2, 3] и начнется его анализ:

```
I     list(L) :–  L=[].                 «Да», если список пуст, а если нет, то проверить II
II    list(L) :–  L=[H|T], list(T).     если список не пуст,  то проверить хвост
```

Здесь: H – голова, а T – хвост. Но после выделения головы и хвоста (в II), голова далее не используется, а значит ее значение возвращать не надо – можно использовать анонимную переменную. Т.е.:

```
I     list(L) :–  L=[].
II    list(L) :–  L=[_|T], list(T).
```

Далее можно заметить, что первые проверки в обоих правилах являются проверками применимости этих правил, значит, эту проверку можно перенести в заголовки:

```
I     list([]).
II    list( [_ | T] ) :– list(T).
```

Эта версия наиболее эффективна: при выборе правила одновременно выполняется проверка – пустой ли список (если выбрано правило I, являющееся фактом, тогда можно ответить «Да» – на поставленный вопрос), а если нет, то прямо при выборе правила II (с помощью алгоритма унификации) список делится на голову и хвост. И, при проверке истинности тела, происходит анализ хвоста, с помощью этой же процедуры (рекурсия). На каждом очередном шаге рекурсии работа будет происходить с хвостом.

Приведем **схему описания формальной работы системы**, при задании вопроса: list([1, 2, 3]) (процесс согласования цели list([1, 2, 3])).

Введены обозначения:

ТР - текущая резольвента; ТЦ - текущая цель; шаг№№ - шаг начинается с появления очередной резольвенты и заканчивается: либо новой резольвентой, либо выводом – успех или неудача; ПРI или ПРII – попытка применения соответствующего правила, рядом указываем **равенства для унификации** и результат (главные функторы одинаковы, арности одинаковы – **проверить надо унифицируемость аргументов**);

т.к. на каждом шаге рекурсии система создает новые экземпляры нужных переменных, то к имени переменной добавляем номер шага (T1, T2, ...). Используем процедуру:

```
I     list([]).
II    list( [_ | T] ) :– list(T).
```

И вопрос (GOAL): list([1,2,3])

ТР: list([1,2,3])

шаг1: ТЦ: list([1,2,3]) *поиск знания с начала базы знаний*

ПРI: [] = [1, 2, 3] унификация невозможна => возврат к ТЦ, метка переносится ниже
ПРII: [_|T1] = [1, 2, 3] успех (подобрано знание)=> {T1 = [2, 3]}, проверка тела ПРII
ТР: list([2,3]) (резольвента изменилась в 2 этапа)
шаг2: ТЦ: list([2,3]) *поиск знания с начала базы знаний*
ПРI: [] = [2,3] унификация невозможна => возврат к ТЦ, метка переносится ниже
ПРII: [_|T2] = [2,3] успех (подобрано знание) => {T2 = [3]}, проверка тела ПРII
ТР: list([3]) (резольвента изменилась в 2 этапа)
шаг3: ТЦ: list([3]) *поиск знания с начала базы знаний*
ПРI: [] = [3]] унификация невозможна => возврат к ТЦ, метка переносится ниже
ПРII: [_|T3] = [3] успех (подобрано знание) => {T3 = []}, проверка тела ПРII
ТР: list([]) (резольвента изменилась в 2 этапа)
шаг4: ТЦ: list([]) *поиск знания с начала базы знаний*
ПРI: [] = [] успех (подобрано знание) => пустое тело заменяет цель в резольвенте
ТР: пусто; успех – однократный ответ – «Да», *метка – на ПРI*
Отказ от найденного значения (откат), возврат к предыдущему состоянию резольвенты
ТР: list([])
шаг5: ТЦ: list([]) *поиск знания от метки ниже*
ПРII: [_|T5] = [] унификация невозможна =>неудача, *надо включить откат, но метка (метки) в конце процедуры* – других альтернатив нет => **система завершает работу** с единственным результатом – «Да».

2). Предикат member, который позволяет проверить: является ли X элементом L, т.е. необходимо иметь два аргумента. Очевидно, что X надо сравнить с головой, т.е. в списке L надо выделить голову и сравнить с X. Это можно сделать несколькими способами:

а) I member(X, L):– L=[H|T], X=H.
II member(X, L):– L=[H|T], member(X,T).

Но в I : хвост нас не интересует, а X=H можно перенести в предыдущее условие, переход к ПР II произойдет, если в ПР I неудачно. В ПР II: нас не интересует голова, а X надо искать в хвосте T – это следующий шаг рекурсии:

б) т.е. оптимизация:
I member(X, L):– L=[X,_].
II member(X, L):– L=[_|T], member(X,T).

Но разбиение L на части можно выполнить при подборе знания, т.е. в заголовке:

в) поэтому, дальнейшая оптимизация:
I member(X, [X|_]).
II member(X, [_|T]):– member(X, T).

Как видели в предыдущем примере, после получения утвердительного ответа на вопрос, с помощью ПРI, система все-таки анализирует возможность использования и ПРII, хотя очевидно, что уже можно ответить «Да» на поставленный вопрос. Можно на это повлиять:

г) используем отсечение – оптимальный вариант:
I member(X, [X|_]):– !.
II member(X, [_|T]):– member(X, T).

Если подобрано ПРI , то предикат ! станет текущей резольвентой, а он истинный, т.е. система получит «Да». При попытке же отменить ! система завершит использование процедуры, ПРII использоваться не будет.

д) без использования отсечения "оптимального" поведения можно добиться с помощью дополнительной проверки, но в этом случае все-таки происходит анализ ПРII:
I member(X, [X|_]).
II member(X,[H|T]):– X<>H, member(X, T).

3). Объединение списков – предикат **append.** Этот предикат должен используя два списка, например, L1 и L2 получить третий- результирующий список L, т.е. у предиката три аргумента (append(L1, L2, L)), пусть результат – это третий аргумент, тогда вопрос можно задать: append([a, b], [c, d], R). Хотим получить R=[a, b, c, d]. Рассмотрим возможные ситуации:

Если 1-й список – пустой, то результат равен второму списку (ПРI). Если первый список не пуст, то надо переписать **в результирующий список**, на место головы – голову первого списка (ПРII), а хвост результата еще не определен (Т3). После чего, **в голову хвоста результирующего списка** надо переписать голову хвоста 1-го списка (рекурсия), и т.д., пока первый список не станет пустым, после чего остатком результирующего списка должен стать второй список.

I append([], L2, L2).
II append([H|T], L2, [H|T3]):- append(T, L2, T3).

Частичное формирование результирующего списка (переписывание головы первого списка в голову результата – ПРII) обеспечиваем на этапе выбора соответствующего правила (заголовка). Отметим, что сцепление элементов в результирующем списке формально реализуется конкретизацией частей списка (причем не сразу всех). Здесь отсечение в первом правиле можно не использовать, т.к., в случае пустоты первого списка, второе правило выбрано быть не может (хотя попытка и происходит), т.к. первый список (он пустой), в этом случае, разбить на части нельзя.