Оглавление

[1. Общие сведения о языке логического программирования Пролог (основные понятия и определения: декларативный язык, логическая программа, правила логического вывода, предметная область, объекты и отношения предметной области; предикаты - факты и правила, запросы). 2](#_Toc534818743)

[2. Декларативная интерпретация правил Пролог- программы. 5](#_Toc534818744)

[3. Процедурная интерпретация правил Пролог- программы. 8](#_Toc534818745)

[4. Общая схема выполнения запроса к Пролог- программе (подстановка, пример подстановки, использование переменных в предикатах и запросах, конъюнктивный запрос). 8](#_Toc534818746)

[5. Дерево решений Пролог- программы. 17](#_Toc534818747)

[6. Механизм поиска с возвратом. 17](#_Toc534818748)

[7. Механизм унификации; связанные и несвязанные переменные. 23](#_Toc534818749)

[8. Таблица унификации переменных; сцепленные переменные. 24](#_Toc534818750)

[9. Разделы Пролог- программы. Пролог- программы с внутренней целью. 25](#_Toc534818751)

[10. Стандартные домены Пролога. Пролог- программы с внешней целью. 29](#_Toc534818752)

[11. Методы организации итеративных правил: метод отката после неудачи. 31](#_Toc534818753)

[12. Методы организации итеративных правил: метод отсечения и отката. 32](#_Toc534818754)

[13. Методы организации итеративных правил. Метод повтора, определяемый пользователем. 34](#_Toc534818755)

[14. Рекурсивные правила. Общая схема рекурсивного выполнения предиката. 35](#_Toc534818756)

[15. Предикат «отсечение»; Использование в логических программах. 36](#_Toc534818757)

[16. Ложные логические программы. 37](#_Toc534818758)

[17. Рекурсия как один из подходов к решению сложных задач. Список как рекурсивно определяемая структура данных. 38](#_Toc534818759)

[18. Особенность использования списков (регулярность структуры, единый механизм доступа к элементам списка; разделение списка на «голову» и «хвост»; частные случаи списков). 39](#_Toc534818760)

[19. Оформление Пролог- программ, работающих со списками (на примере предиката, проверяющего вхождение отдельного элемента в список). 42](#_Toc534818761)

[20. Явное и неявное определение граничных условий при работе со списками (на примере предиката проверки вхождения элемента в список или предиката проверки совпадения двух списков). 43](#_Toc534818762)

[21. Декларативная интерпретация предиката слияния («присоединить») двух списков. 43](#_Toc534818763)

[22. Процедурная интерпретация предиката слияния («присоединить») двух списков. 43](#_Toc534818764)

[23. Варианты использования предиката слияния («присоединить») двух списков (определение префикса, суффикса, элемента списка). 44](#_Toc534818765)

[24. Компоновка данных в список (предикат «findall»). 44](#_Toc534818766)

[25. Структуры в Прологе. Описание альтернативных однодоменной и многодоменной структур. 45](#_Toc534818767)

[26. Структуры в Прологе. Доменное описание бинарного дерева. 47](#_Toc534818768)

[27. Структуры в Прологе. Префиксное представление бинарного дерева. 48](#_Toc534818769)

[28. Решение задач на графах: варианты описания графа на Прологе. 48](#_Toc534818770)

[29. Решение задач на графах (на примере задачи поиска пути в графе). 50](#_Toc534818771)

[30. Трассировка Пролог- программ (в терминах четырех видов происходящих событий: CALL, RETURN, REDO, FALL). 50](#_Toc534818772)

[31. Общие сведения о языке функционального программирования Лисп: основные понятия и определения. Правила записи программ на языке Лисп. 51](#_Toc534818773)

[32. Схема рекурсивного выполнения Лисп - программы. 52](#_Toc534818774)

[33. Основные функции обработки списков языка Лисп: CAR, CDR, CONS. 54](#_Toc534818775)

[34. Основные функции языка Лисп: ATOM, NULL, LIST. 55](#_Toc534818776)

[35. Машинное представление списочных структур. 56](#_Toc534818777)

[36. Логическая и физическая идентичность списков. 58](#_Toc534818778)

[37. Предикаты сравнения. 58](#_Toc534818779)

[38. Лисп- программы, рекурсивные по аргументу (на примере функции APPEND). 59](#_Toc534818780)

[39. Лисп- программы, рекурсивные по значению (на примере функции APPEND). 60](#_Toc534818781)

[40. Другие формы рекурсии: параллельная рекурсия. 60](#_Toc534818782)

[41. Другие формы рекурсии: взаимная рекурсия. 60](#_Toc534818783)

[42. Точечная нотация. 61](#_Toc534818784)

[43. Спискоразрушающие функции: RPLACA, RPLACD. 62](#_Toc534818785)

[44. Спискоразрушающие функции: NCONC, ATTACH. 62](#_Toc534818786)

[45. Функциональная абстракция, форма записи лямбда- выражений. 64](#_Toc534818787)

[46. Применяющие функционалы. 66](#_Toc534818788)

[47. Основные черты аппликативного программирования. 68](#_Toc534818789)

# Общие сведения о языке логического программирования Пролог (основные понятия и определения: декларативный язык, логическая программа, правила логического вывода, предметная область, объекты и отношения предметной области; предикаты - факты и правила, запросы).

Язык Пролог является представителем семейства языков логического программирования и в сравнении с традиционными языками программирования, предназначенными для записи алгоритмов, такими как Бейсик, Фортран, Паскаль, Си, обладает существенными особенностями:

• программа на Прологе не является алгоритмом, а представляет собой запись условия задачи на языке формальной логики (т.е.

это дескриптивный, описательный язык программирования);

• язык Пролог предназначен не для решения вычислительных или графических задач, а для решения логических задач, для моделирования процесса логического умозаключения человека; вычисления же и графические построения выполняются в Прологе как побочный продукт логического вывода;

• Пролог требует особого стиля мышления программиста, что затрудняет изучение его теми, кто уже привык к процедурному программированию, поэтому, так называемые, практические программисты не стремятся переходить на этот язык, что мешает росту популярности Пролога; однако во многих странах (Японии, Англии, Франции, Германии, Израиле и т.д.) расширяется практика применения Пролога в образовании как первого изучаемого языка программирования; переход к процедурным языкам типа Паскаля в этом случае трудностей не вызывает.

Все это позволяет отнести Пролог в существующем делении языков программирования на языки низкого и высокого уровня к языкам сверхвысокого уровня. В японском проекте создания в 90-х годах XX века компьютеров 5-го поколения (обладающих искусственным интеллектом) Пролог положен в основу аппаратной организации и разработки программного обеспечения. Нынешний Пролог, безусловно, не является окончательным вариантом языка программирования ЭВМ 5-го поколения и в ближайшие годы получит существенное развитие. По-видимому, он сыграет роль Бейсика дескриптивного программирования: его значение и возможности в популяризации и распространении идей логического программирования чрезвычайно велики.

Изучению языка Пролог очень способствует предшествующее изучение математической логики, понятийной системой которой он пользуется.

Программирование на Прологе включает в себя следующие этапы:

1) объявление фактов об объектах и отношениях между ними;

2) определение правил взаимосвязи объектов и отношений между ними;

3) формулировка вопроса об объектах и отношениях между ними.

Имена - это последовательности букв и цифр, начинающиеся с буквы (строчной !). Системы программирования на Прологе для компьютеров допускают использование лишь латинских строчных и прописных букв: а .. z, A .. Z. Использование русских строчных и прописных букв: а .. я, А .. Я не допускается. При практической работе с интерпретатором рекомендуется, чтобы смысл имен оставался понятным, использовать в качестве имен запись русских слов латинскими буквами. В данном параграфе мы будем записывать все имена русскими буквами, чтобы сделать смысл программ наиболее понятным. При запуске этих программ в «англо-язычных» системах программирования нужно заменять русские буквы в именах на латинские.

Типы данных включают переменные, атомарные значения и структуры (рис. 3.15).

Рис.3.1 5. Классификация типов данных Пролога

Примеры специальных атомов:

: - ( обозначающая импликацию),

? (вопрос, обозначающий отрицание),

! (предикат отсечения, рассматривается далее).

Переменные обозначаются последовательностью буквой и цифр, начинающейся с заглавной буквы. Особый вид переменной - анонимная переменная \_ , используемая в качестве аргумента предиката, когда конкретное значение переменной несущественно.

Структура - это конструкция, состоящая из имени структуры и заключенного в скобки списка ее аргументов, разделенных запятыми. Элементами структур могут быть числа, атомы, переменные, другие структуры и списки. Примеры структур: str(A,B,C), носит(юрий,пиджак).

Списки представляют собой объединение элементов произвольных видов, разделенных запятыми и заключенных в квадратные скобки. Списки отличаются от структур тем, что количество элементов может меняться при выполнении программы. Примеры списков: [1,3,5,7], [красный,желтый,зеленый].

Основная операция, выполняемая в языке Пролог, - это операция сопоставления (называемая также унификацией или согласованием). Операция сопоставления может быть успешной, а может закончиться неудачно. Определяется операция сопоставления так:

• константа сопоставляется только с равной ей константой;

• идентичные структуры сопоставляются друг с другом;

• переменная сопоставляется с константой или с ранее связанной переменной (и становится связанной с соответствующим значением);

• две свободные переменные могут сопоставляться (и связываться) друг с другом. С момента связывания они трактуются как одна переменная: если одна из них принимает какое-либо значение, то вторая немедленно принимает то же значение.

Примеры: 5 сопоставляется с 5, «имеет» сопоставляется с «имеет», «сергей» не сопоставляется с «юрий», «имеет(сергей,машина)» не сопоставляется с «имеет(сергей, телевизор)», «имеет(сергей,машина)» сопоставляется с «имеет(Х,машина)», в этом случае переменная Х получает в качестве значения атом «сергей».

Факты - это предикаты с аргументами-константами, обозначающие отношения между объектами или свойства объектов, именованные этими константами. Факты в программе считаются всегда и безусловно истинными и таким образом служат основой доказательства, происходящего при выполнении программы.

В Прологе все предложения программы - факты, правила, вопрос - заканчиваются точкой.

Отметим, что в Прологе вместо оператора присваивания имеется более общий и мощный механизм задания значений переменных. Переменные в Прологе получают свои значения в результате сопоставления с константами в фактах и правилах. До тех пор. пока переменная не получила какого-либо значения, она называется «свободной». Когда переменная примет значение, она становится «связанной». Однако, она остается связанной только в течение времени, необходимого для получения одного ответа на запрос, после этого Пролог «развязывает» ее, возвращается и ищет альтернативные решения. Это очень важный момент: нельзя хранить информацию, задавая значения переменных. Переменные служат частью процесса сопоставления, а не «хранилищем» информации. Область действия переменной -ровно одно предложение (правило или запрос программы).

# Декларативная интерпретация правил Пролог- программы.

Рассмотрим предложение

P :- Q, R.

где P, Q и R имеют синтаксис термов. Приведем некоторые способы декларативной интерпретации этого предложения:

P — истинно, если Q и R истинны.

Из Q и R следует P.

А вот два варианта его "процедурного" прочтения:

Чтобы решить задачу P, сначала реши подзадачу Q, а затем — подзадачу R.

Чтобы достичь P, сначала достигни Q, а затем R.

Таким образом, различие между "декларативным" и "процедурным" прочтениями заключается в том, что последнее определяет не только логические связи между головой предложения и целями в его теле, но еще и порядок, в котором эти цели обрабатываются.

Формализуем теперь декларативный смысл.

Декларативный смысл программы определяет, является ли данная цель истинной (достижимой) и, если да, при каких значениях переменных она достигается. Для точного определения декларативного смысла нам потребуется понятие конкретизации предложения. Конкретизацией предложения С называется результат подстановки в него на место каждой переменной некоторого терма. Вариантом предложения С называется такая конкретизация С, при которой каждая переменная заменена на другую переменную. Например, рассмотрим предложение:

имеетребенка( X) :- родитель( X, Y).

Приведем два варианта этого предложения:

имеетребенка( А) :- родитель( А, В).

имеетребенка( X1) :- родитель( X1, Х2).

Примеры конкретизаций этого же предложения:

имеетребенка( питер) :- родитель( питер, Z).

имеетребенка( барри) :- родитель( барри,

маленькая( каролина) ).

Пусть дана некоторая программа и цель G, тогда, в соответствии с декларативной семантикой, можно утверждать, что

Цель G истинна (т.е. достижима или логически следует из программы) тогда и только тогда, когда

(1) в программе существует предложение С, такое, что

(2) существует такая его (С) конкретизация I, что

(a) голова I совпадает с G и

(б) все цели в теле I истинны.

Это определение можно распространить на вопросы следующим образом. В общем случае вопрос к пролог-системе представляет собой список целей, разделенных запятыми. Список целей называется истинным (достижимым), если все цели в этом списке истинны (достижимы) при одинаковых конкретизациях переменных. Значения переменных получаются из наиболее общей конкретизации.

Таким образом, запятая между целями обозначает конъюнкцию целей: они все должны быть истинными. Однако в Прологе возможна и дизъюнкция целей: должна быть истинной, по крайней мере одна из целей. Дизъюнкция обозначается точкой с запятой. Например:

P :- Q; R.

читается так: P — истинно, если истинно Q или истинно R. То есть смысл такого предложения тот же, что и смысл следующей пары предложений:

P :- Q.

P :- R.

Запятая связывает (цели) сильнее, чем точка с запятой. Таким образом, предложение

P :- Q, R; S, T, U.

понимается как:

P :- ( Q, R); (S, T, U).

и имеет тот же смысл, что и два предложения

P :- Q, R.

P :- S, T, U.

# Процедурная интерпретация правил Пролог- программы.

Процедурная семантика определяет, *как* пролог-система отвечает на вопросы. Ответить на вопрос - это значит удовлетворить список целей. Этого можно добиться, приписав встречающимся переменным значения таким образом, чтобы

цели логически следовали из программы. Можно

сказать, что процедурная семантика Пролога - это

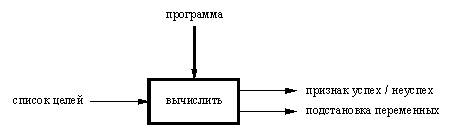
процедура вычисления списка целей

с учетом заданной программы. "Вычислить цели" это значит попытаться достичь их.

Назовем эту процедуру **вычислить**. Как показано на рис. 2.9, входом и выходом этой процедуры являются:

    входом - программа и список целей,

    выходом - признак успех/неуспех и подстановка переменных.



**Рис. 2. 9.**  Входы и выходы процедуры вычисления списка целей.

Смысл двух составляющих выхода такой:

(1)    Признак успех/неуспех принимает значение "да", если цели достижимы, и "нет" - в противном случае. Будем говорить, что "да" сигнализирует об *успешном* завершении и "нет" - о *неуспехе*.

(2)    Подстановка переменных порождается только в случае успешного завершения; в случае неуспеха подстановка отсутствует.

# Общая схема выполнения запроса к Пролог- программе (подстановка, пример подстановки, использование переменных в предикатах и запросах, конъюнктивный запрос).

Описание задачи на Турбо-Прологе состоит из трех основных компонентов:

Описание имен и структуры объектов, входящих в рассматриваемую задачу;

Описание имен отношений существующих между объектами;

Предложения, описывающие отношения между объектами в виде совокупности фактов и правил.

Программа на Турбо-Прологе состоит из следующих разделов, среди которых можно выделить три секции

директивы компилятора;

CONSTANTS - раздел описания констант;

DOMAINS - раздел описания областей определения;

DATABASE - раздел описания предикатов внутренней базы данных;

PREDICATES - раздел описания предикатов;

CLAUSES - раздел описания предложений;

GOAL - раздел описания внутренней цели.

Секции программы:

Секция определения объектов (DOMAINS)

Секция описания предикатов (PREDICATES)

Секция предложений (CLAUSES)

В программе не обязательно должны быть все эти разделы.

Пример: Программа, состоящая из одного описания цели:

goal

write("hello"), readchar( \_ ).

Эта программа выведет сообщение"hello" с помощью стандартного предиката write и будет ожидать нажатия пользователем любой клавиши (стандартный предикат readchar читает символ).

Как правило, программа содержит, по меньшей мере, разделы predicates и clauses.

Если программа запускается в среде разработки Турбо-Пролога, то раздел goal необязателен. При написании же программы, не зависящей от среды разработки, в ней необходимо указать внутреннюю цель.

В программе может быть несколько разделов описаний domains, predicates, database и clauses. Однако разделов goal не может быть в программе более одного.

Порядок разделов может быть произвольным, но при этом константы, области определения и предикаты должны быть определены до их использования. Однако в разделе domains можно ссылаться на области определения, которые будут объявлены позже.

Раздел описания констант

Раздел, озаглавленный зарезервированным словом constants, предназначен для описания констант. Объявление константы имеет вид:

<имя константы>=<значение>

Имя константы должно быть идентификатором, то есть оно может состоять из латинских букв, цифр и знака подчеркивания, причем не может начинаться с цифры.

Каждое определение константы должно размещаться в отдельной строке.

Пример: Определить константы.

constants

pi=3.14

path="c:\\prolog\\bgi"

В разделе описания констант можно использовать в качестве первого символа имени константы прописные символы, потому что в этом разделе прописные и строчные символы не различаются. Однако при использовании констант в разделе описания предложений следует использовать в качестве первого символа имени константы только строчные символы, чтобы Пролог-система не восприняла константу как переменную.

Разделов описания констант может быть несколько, но каждая константа должна быть определена до ее первого использования.

Раздел описания областей определения

Раздел описания областей определения или доменов является аналогом раздела описания типов в обычных императивных языках программирования и начинается с ключевого слова domains.

В Турбо-Прологе имеются стандартные области определения, которые не нужно указывать в разделе описания доменов. Основные стандартные домены - это:

integer - целое число (из промежутка -32768...32767);

real - действительное число (лежащее между ±1e-307...±1e308);

char - символ, заключенный в одиночные апострофы;

string - последовательность символов, заключенная в двойные кавычки;

symbol - символьная константа (начинающаяся со строчной буквы последовательность букв латинского алфавита, цифр и знаков подчеркивания или последовательность любых символов, заключенная в кавычки).

file – файл.

В разделе описания доменов объявляются любые нестандартные области определения, используемые в качестве аргументов предикатов.

Объявление домена имеет следующий вид:

<имя домена>=<определение домена> или

file=<имя файлового домена1>;...;<имя файлового доменаN>

Удобно использовать описание доменов для сокращения имен стандартных областей определения.

Пример: Использовать вместо ключевого слова integer односимвольное обозначение i.

domains

i=integer

Из доменов можно конструировать составные или структурные домены (структуры). Структура описывается следующим образом:

<имя структуры> = <имя функтора>(<имя домена первой компоненты>,..., <имя домена последней компоненты>) [;<имя функтора>(...)]\*

Каждая компонента структуры в свою очередь может быть структурой.

Пример: Предложить структуру, описывающую треугольник, содержащую в качестве компоненты структуру, описывающую точку на плоскости, имеющую две компоненты (координаты точки).

point = p(integer , integer)

triangle = tr(point , point , point)

В описание структуры могут входить альтернативы, разделенные символом ";" или ключевым словом "or".

Так, структуру, описывающую точку и на плоскости, и в пространстве, можно задать следующим образом:

point = p(integer , integer); p(integer , integer , integer).

Описание файлового домена имеет вид:

file = <символическое имя файла 1>;...;<символическое имя файла N>

Для представления данных в Турбо-Прологе, в отличие от стандартных алгоритмических языков программирования, используются не массивы, а списки. Списковый домен задается следующим образом:

<имя спискового домена> = <имя домена элементов списка>\*

Пример: Список целых чисел

list\_of\_integer = integer\*

Раздел описания предикатов внутренней базы данных

Раздел описания предикатов внутренней базы данных начинается с зарезервированного слова database и в нем описываются те предикаты, которые можно в процессе выполнения программы добавлять во внутреннюю базу данных или удалять оттуда. Описываются предикаты базы данных аналогично предикатам в разделе predicates.

Раздел описания предикатов

В разделе, озаглавленном зарезервированным словом predicates, содержатся описания определяемых пользователем предикатов. В традиционных языках программирования подобными разделами являются разделы описания заголовков процедур и функций. Описание n-местного предиката имеет следующий вид:

<имя предиката>(<имя домена первого аргумента>,...,

<имя домена n-го аргумента>).

Домены(области определения) аргументов должны быть либо стандартными, либо объявленными в разделе описания доменов. Следует обращать внимание на то, что имя предиката в Турбо-Прологе должно быть идентификатором, т.е. оно должно состоять только из латинских букв, цифр и символа подчеркивания, причем имя предиката не может начинаться с цифры.

Пример: Предикат, описывающий отношение "мама"

predicates

mother(string , string)

Это описание означает, что у предиката два аргумента, причем оба строкового типа:

mother("Наташа", "Даша").

Один предикат может иметь несколько описаний. Это используется, когда требуется, чтобы предикат работал с аргументами различной природы.

Пример: Предикат, проверяющий принадлежность элемента списку.

predicates

member(integer,integer\*)

member(real,real\*)

member(char,char\*)

member(string,string\*)

Такие описания означают, что у предиката два аргумента. При этом возможны четыре варианта использования этого предиката. Первый аргумент может быть целым числом, вещественным числом, символом или строкой, второй аргумент является списком из объектов соответствующего типа. При этом процедура, реализующая этот предикат в разделе описания предложений, будет единственной.

Кроме того, при описании предиката можно указать, будет он детерминированным или недетерминированным. Детерминированный предикат возвращает только одно решение, а недетерминированный предикат при помощи поиска с возвратом может давать много решений. Детерминированные предикаты менее требовательны к оперативной памяти и выполняются быстрее. Предикат отсечения cut , позволяет превращать недетерминированные предикаты в детерминированные.

Для того чтобы указать, что предикат является детерминированным (недетерминированным), следует перед его именем поместить зарезервированное слово determ (nondeterm). По умолчанию предикат считается детерминированным.

В Турбо-Прологе имеется директива компилятора check\_determ, которая принудительно включает проверку предикатов на детерминированность.

В Турбо-Прологе есть так называемые стандартные (встроенные) предикаты, которые не нужно описывать в разделе описания предикатов predicates. Все встроенные предикаты являются детерминированными.

Раздел описания предложений

Раздел описания предложений начинается со служебного слова clauses. Этот раздел можно считать основным разделом программы, потому что именно в нем содержатся факты и правила, реализующие пользовательские предикаты. Все предикаты, которые применяются в этом разделе и не являются стандартными предикатами, должны быть описаны в разделе описания предикатов или в разделе описания предикатов базы данных.

Предложения, у которых в заголовке указан один и тот же предикат, должны идти друг за другом. Такой набор предложений называется процедурой. Программу на Прологе принято оформлять по следующим правилам:

- между процедурами пропускается пустая строка;

- тело правила записывается со следующей строки, после строки, в которой был - заголовок, с отступом;

- каждую подцель записывают на отдельной строке, одну под другой.

Эти правила не являются обязательными, но они делают программу более читабельной.

Раздел описания целевого запроса

Раздел описания внутренней цели программы или внутрипрограммного целевого запроса начинается с зарезервированного слова GOAL. В соответствии с запросом осуществляется поиск только первого решения, а для получения всех возможных решений нужно предпринимать дополнительные действия.

Если раздел GOAL в тексте программы отсутствует, то после запуска программы Пролог-система выдает приглашение вводить запросы в диалоговом режиме (внешняя цель). При выполнении внешней цели Пролог-система ищет все решения, выводя все возможные значения для переменных, участвующих в запросе.

Программа, компилируемая в исполняемый файл, который можно запускать независимо от среды разработки, обязательно должна иметь внутреннюю цель. Внешнюю цель обычно применяют на этапе отладки программы.

Пример: Определить, кто является бабушкой, исходя из имеющихся сведений о том, кто является мамой.

domains /\* раздел описания областей определения \*/

s=string /\* синоним для строкового типа данных \*/

predicates /\* раздел описания предикатов \*/

mother(s,s) /\*мама\*/

grandmother(s,s) /\*бабушка\*/

clauses /\* раздел описания предложений \*/

mother("Наташа","Даша"). /\*"Наташа" является мамой для

"Даша"\*/

mother("Даша","Маша"). /\* "Даша" является мамой для

"Маша”\*/

grandmother( X , Y ) :- mother( X ,Z ), mother( Z , Y ).

/\* X является бабушкой Y, если найдется такой Z, что X является мамой для Z, а Z является мамой для Y \*/

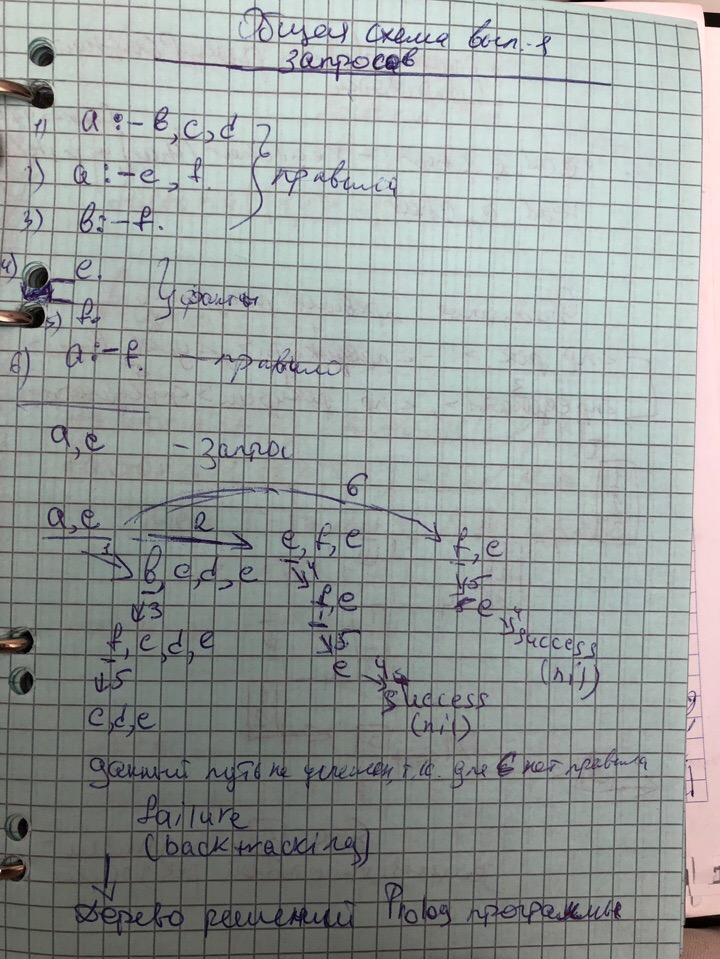
Поскольку раздел GOAL в программе отсутствует, то после запуска программы Пролог-система выведет приглашение на ввод внешней цели:

GOAL: grandmother(X , Y).

X=”Наташа” Y=”Маша”

Повтор предыдущей цели возможен при нажатии клавиши F8.

# Дерево решений Пролог- программы.



# Механизм поиска с возвратом.

Поиск с возвратом (backtracking) - это один из основных приемов поиска решений поставленной задачи в Prolog'е.

Каким образом работает поиск с возвратом? Это достаточно хорошо можно пояснить, вот на каком примере.

Предположим, для достижения некоторой цели человеку необходимо последовательно принять несколько решений и выполнить некоторые действия в соответствии с принятыми решениями. Первоначально человек без колебаний и раздумий принимает несколько решений, но при решении очередной проблемы у него возникают сомнения, поскольку возможных решений, предположим, имеется два, и человеку они кажутся одинаково правильными. Какое-либо из двух решений человек все равно принимает (но запоминает, в какой момент он сомневался, и какое из двух решений все же выбрал) и продолжает свое движение к поставленной цели.

Но, в какой-то момент оказывается, что решение, выбранное из двух, все же оказалось неправильным. Тогда человек вернется в точку принятия неверного решения, и пойдет по альтернативному пути. Не факт, что вновь выбранный путь окажется правильным, но человек попробует все возможные варианты нахождения решения.

Еще одна аналогия. Поиск с возвратом можно сравнить с поиском выхода из лабиринта. Нужно войти в лабиринт и на каждой развилке сворачивать влево, до тех пор, пока не найдется выход или тупик. Если впереди оказался тупик, нужно вернуться к последней развилке и свернуть направо, затем снова проверять все левые пути.

В конце концов, выход (если он есть) будет найден. Подобным образом работаем и механизм поиска с возвратом в языке Prolog.

Благодаря механизму поиска с возвратом Prolog в состоянии находить все возможные решения, имеющиеся для данной задачи.

Рассмотрим на примере, каким образом выполняется поиск всех возможных решений с применением поиска с возвратом.

PREDICATES

little (symbol)

middle (symbol)

big (symbol)

strong (symbol)

powerful (symbol)

CLAUSES

little (cat).

little (wolf).

middle (tiger).

middle (bear).

big (elephant).

big (hippopotamus).

strong (tiger).

powerful (Animal):- middle (Animal), strong (Animal).

powerful (Animal):- big (Animal).

Итак, обратимся к программе с запросом - какое животное можно назвать мощным?

Запрос будет выглядеть следующим образом:

Goal: powerful (Animal).

Проследим по шагам, каким образом будут находиться все возможные решения.

Доказательство цели, сформулированной в запросе, начинается с последовательного просмотра всех предложений, имеющихся в тексте программы. В данном примере цель powerful (Animal) может быть сопоставлена с заголовком первого правила вывода, что и происходит, но при этом помечается, что в тексте программы имеется еще одно правило точно с таким же заголовком, то есть устанавливается первая точка возврата (назовем ее \*1).

Так как было выбрано первое правило вывода, теперь необходимо последовательно доказать все цели, перечисленные в теле правила. Для доказательства цели middle (Animal) вновь начинается просмотр всех предложений, имеющихся в тексте программы, и находится факт middle (tiger). Но! Поскольку имеется еще один факт, описывающий животное средних размеров middle (bear). , устанавливается вторая точка возврата (назовем ее \*2). Переменная Animal получает значение tiger. Первая цель в теле правила успешно доказана.

Теперь выполняется переход к доказательству цели strong (tiger). Переменная Animal получила значение tiger при доказательстве предыдущей цели. Чтобы доказать цель strong (tiger), вновь начинается просмотр всех предложений, имеющихся в тексте программы, и находится факт strong (tiger), успешно доказывающий цель strong (tiger). Точка возврата не устанавливается, так как в тексте программы нет больше фактов strong, описывающих сильных животных.

Так как доказаны все цели в теле правила, считается успешно доказанной головная цель правила, и, следовательно, цель powerful (Animal), записанная в исходном запросе.

Найдено первое решение: Animal=tiger.

Поскольку должны быть найдены все возможные решения, вступает в действие поиск с возвратом, который возвращает выполнение программы к последней установленной точке возврата - \*2, то есть к цели middle (Animal), которая может быть передоказана. Вновь начинается просмотр всех предложений, но не с самого первого, а с того, на котором была установлена точка возврата \*2 и цель middle (Animal) успешно передоказывается фактом middle (bear). Следует отметить, что переменная Animal, получившая при нахождении первого решения значение tiger, потеряла это значение, когда поиск с возвратом вернулся к передоказательству цели middle (Animal), то есть, нет никаких препятствий к тому, чтобы переменная Animal получила теперь значение bear.

Точка возврата \*2 удаляется и вновь не устанавливается, так как нет более фактов middle, описывающих животных средних размеров. Итак, успешно передоказана первая цель в теле правила, и восстанавливается исходный порядок действий, то есть выполняется переход к доказательству второй цели в теле правила, но только теперь это цель strong (bear). Найти факт, доказывающий данную цель, не удается, то есть считается недоказанной вторая цель в теле правила, следовательно, вновь в действие вступает поиск с возвратом и происходит возврат к ближайшей точке возврата, а это точка \*1. Точка возврата \*1 свидетельствует о том, что вновь начинается просмотр предложений в тексте программы, но не с самого начала, а с предложения, помеченного этой самой точкой \*1. При просмотре обнаруживается, что цель в запросе powerful (Animal) может быть передоказана с помощью второго правила вывода.

Так как выполнен возврат к последней точке возврата, переменная Animal вновь теряет свое значение, и, так как больше возможностей для передоказательства цели в запросе powerful (Animal) нет, точка возврата \*1 более не устанавливается.

Теперь вновь повторяются действия, похожие на действия, происходившие, когда для доказательства использовалось первое правило вывода, только действий будет немного меньше, так как в теле второго правила всего одна цель.

Итак, цель запроса powerful (Animal) была сопоставлена с заголовком второго правила, что привело к необходимости доказать единственную цель в теле правила - big (Animal). Для этого вновь начинается просмотр предложений в тексте программы с самого начала и обнаруживается факт big (elephant). Вновь устанавливается точка возврата, назовем ее \*3, говорящая о том, что цель big (Animal) в дальнейшем может быть передоказана. Факт big (elephant). успешно доказывает цель big (Animal) и переменная Animal принимает значение elephant. Так как успешно доказаны все (в данном случае одна) цели в теле правила, считается успешно доказанной и заголовочная цель правила, что приводит к успеху в доказательстве цели в запросе, и вот оно, второе решение:

Animal=elephant.

Так как успешно найдено очередное решение, возобновляются действия по поиску следующих возможных решений, ведь есть точка возврата \*3, к которой можно вернуться и передоказать цель в теле правила big (Animal) (в процессе поиска с возвратом переменная Animal вновь теряет свое значение). Точка возврата \*3 свидетельствует о том, что вновь начинается просмотр предложений в тексте программы, но не с самого начала, а с предложения, помеченного точкой \*3. При просмотре обнаруживается, что цель в теле правила big (Animal) может быть передоказана с помощью факта big (hippopotamus). , что и делается, переменная Animal получает значение hippopotamus, точка возврата \*3 удаляется и более не устанавливается, так как больше нет фактов, описывающих больших животных, и считается найденным очередное, третье, решение:

Animal=hippopotamus.

Так как все возможные решения найдены, выполнение программы заканчивается.

Далее приводится пример трассировки (пошагового выполнения) только что рассмотренного примера.

Условные обозначения для трассировки:

CALL - цель, которую нужно доказать.

RETURN - цель, которая успешно доказана.

REDO - поиск с возвратом.

FAIL - неудача в доказательстве.

\* - точка возврата.

\_ - переменная, не имеющая значения.

CALL: powerful (\_)

CALL: middle (\_)

RETURN: \*middle ("tiger")

CALL: strong ("tiger")

RETURN: strong ("tiger")

RETURN: \*powerful ("tiger")

REDO: middle (\_)

RETURN: middle ("bear")

CALL: strong ("bear")

FAIL: strong ("bear")

REDO: powerful (\_)

CALL: big (\_)

RETURN: \*big ("elephant")

RETURN: powerful ("elephant")

REDO: big (\_)

RETURN: big ("hippopotamus")

RETURN: powerful ("hippopotamus")

Теперь можно сформулировать основные правила поиска с возвратом:

Цели должны быть доказаны по порядку, слева, направо.

Для доказательства некоторой цели предложения просматриваются в том порядке, в каком они появляются в тексте программы.

Для того, чтобы доказать головную цель правила, необходимо доказать цели в теле правила. Тело правила состоит, в свою очередь из целей, которые должны быть доказаны.

Цель считается доказанной, если с помощью соответствующих фактов доказаны все цели, находящиеся в листьевых вершинах дерева целей.

Для последнего правила следует пояснить, что называется деревом целей. Ход решения программы удобно представлять в виде дерева, которое называется деревом целей.

Пример дерева целей для ранее рассмотренного примера, для случая нахождения первого решения Animal=tiger.

Условные обозначения в дереве целей:

powerful (Animal) - цель, которую нужно доказать.

powerful (tiger) - цель, которая успешно доказана.

В данном дереве целей две цели: middle (Animal) и strong (tiger), находящиеся в листьевых вершинах, и обе они доказаны.

# Механизм унификации; связанные и несвязанные переменные.

УНИФИКАЦИЯ (отождествление) - механизм сопоставления целевого предиката с базой данных и конкретизации (сопоставления)

переменных.

Механизм унификации Турбо-Пролога использует и реализует

следующие правила при согласовании целей:

1) Свободные переменные могут быть унифицированы с любым

термом.

2) Атомы и константы могут быть унифицированы только сами с

собой.

3) Структурный терм может быть унифицирован с другим струк-

турным термом, если оба они имеют одинаковые функторы и

равное число компонент, при этом константы, используемые

в качестве компонент могут быть связаны только сами с со-

бой или со свободной переменной.

Унификация – это приведение выражений справа и слева от знака унификации к одному значению. При этом несвязанные значения из одного выражения будут заполнены значениями из другого выражения. Таким образом, если в операции унификации участвуют две переменных, и каждая связана со своим значением, то выполняется их сравнение. Если одна переменная не связана, то выполняется приравнивание ей значения. А если в унификации участвуют выражения с наличием несвязанных частей, то эти части принимают значения из таких же частей другого выражения. При этом ПроЛог пытается привести оба выражения к одному значению. Примеры.

Унификация как приравнивание.

Goal

А = 5,

А = Б

write(А, “,”, Б).

Здесь в первой строчке унификация несвязанной переменной со значением 5. Результат – значение переменной А связано с константой 5. Во второй строчке унификация несвязанной переменной В и связанной А приведет к передаче значения переменной А в переменную Б. ПроЛог выдаст на экран:

5,5

Унификация как сравнение

Goal

А = 5,

Б = “да”,

А = Б.

Здесь уже в унификации будут участвовать два полностью связанных значения, значит происходит простое сравнение значений. Результат выполнения ПроЛога:

No

Это говорит что унификация не успешна, то есть значения переменной А и Б неравны.

Переменная, которая получила какое-то значение и оказалась связанной с определенным объектом, называется связанной. Если переменная была конкретизирована каким-то значением и ей сопоставлен некоторый объект, то эта переменная уже не может быть изменена.

# Таблица унификации переменных; сцепленные переменные.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Константы  C2 | Опреде. Перем.  V2 | Неоперд. Перем.  F2 | Структуры  S2 |
| Константы  C1 | True, C1=C2 | True, C1=V2 | True, F2:=C1 | False |
| Определенные переменные  V1 | True, C2=V1 | True, V1=V2 | True, F2:=V1 | True, V1=S2 |
| Неопред. Перем.  F1 | True, F1:=C2 | True, F1:=F2 | True со сцеплением F1 и F2 | True, F1:=S2 |
| Структуры S1 | False | True, V2=S1 | True, F2:=S1 | True, при усл1 и усл2 |

Если Х и Y - неконкретизированные (свободные) переменные, то они сопоставимы, в этом случае говорят, что они сцеплены.

# Разделы Пролог- программы. Пролог- программы с внутренней целью.

Обычно программа на Прологе состоит из четырех основных программных разделов. К ним относятся:

раздел clauses (предложений);

раздел predicates (предикатов);

раздел domains (доменов);

раздел goal (целей).

Раздел clauses - это сердце Пролог-программы; именно в этот раздел записываются факты и правила, которыми будет оперировать Пролог, пытаясь разрешить цель программы.

Раздел predicates - это тот, в котором объявляются предикаты и домены (типы) их аргументов (вам не нужно объявлять предикаты, встроенные в Прологе).

Раздел domains служит для обьявления всех используемых нами доменов, не являющихся стандартными доменами Пролога (стандартные домены объявлять не нужно).

Раздел goal - это тот, в который вы помещаете цель Пролог-программы.

Раздел предложений

В раздел clauses (предложений) вы помещаете все факты и правила, составляющие вашу программу. Основное внимание на этом шаге было уделено рассмотрению предложений (фактов и правил) программы: что они означают, как их писать и т. д.

Если вы поняли, что собой представляют факты и правила и как их записывать в Прологе, то вы знаете, что все предложения для каждого конкретного предиката в разделе clauses должны располагаться вместе. Последовательность предложений, описывающих один предикат, называется процедурой.

Пытаясь разрешить цель, Пролог (начиная с первого предложения раздела clauses) будет просматривать каждый факт и каждое правило, стремясь найти сопоставление. По мере продвижения вниз по разделу clauses, он устанавливает внутренний указатель на первое предложение, являющееся частью пути, ведущего к решению. Если следующее предложение не является частью этого логического пути, то Пролог возвращается к установленному указателю и ищет очередное подходящее сопоставление, перемещая указатель на него (этот процесс называется поиск с возвратом).

Если в разделе clauses программы на Прологе вы описали собственный предикат, то вы обязаны объявить его в разделе predicates (предикатов); в противном случае Пролог не поймет, о чем вы ему "говорите". В результате объявления предиката вы сообщаете, к каким доменам (типам) принадлежат аргументы этого предиката.

Предикаты задают факты и правила. В разделе же predicates все предикаты просто перечисляются с указанием типов (доменов) их аргументов. Эффективность работы Пролога значительно возрастает именно из-за того, что вы объявляете типы объектов (аргументов), с которыми работают ваши факты и правила.

Как объявить пользовательский предикат

Объявление предиката начинается с имени этого предиката, за которым идет открывающая (левая) круглая скобка, после чего следует ноль или больше доменов (типов) аргументов предиката:

predicateName(argument\_type1 OptionalName1,

argument\_type2 OptionalName2,..,

argument\_typeN OptionalNameN)

После каждого домена (типа) аргумента следует запятая, а после последнего типа аргумента - закрывающая (правая) скобка. Отметим, что, в отличие от предложений в разделе clauses, декларация предиката не завершается точкой. Доменами (типами) аргументов предиката могут быть либо стандартные домены, либо домены объявленные вами в разделе domains. Можно указывать имена аргументов OptionalNameK - это улучшает читаемость программы, и не сказывается на скорости ее исполнения, т. к. компилятор их игнорирует.

Имена предикатов

Имя предиката должно начинаться с буквы, за которой может располагаться последовательность букв, цифр и символов подчеркивания. Регистр букв не имеет значения, однако мы не советуем вам использовать заглавные буквы в качестве первой буквы имени предиката. Имя предиката может иметь длину до 250 символов.

**Цель** - это формулировка задачи, которую программа должна решить. Цель состоит из взаимосвязанных предикатов. Структура цели точно такая же, как у правила или факта.

Цель используется для запуска процесса выполнения программы. Пролог пытается сопоставить цель с фактами и правилами программы. Если цель является фактом, таким как «любит(мэри, яблоки)», то Пролог отвечает True (истина) или False (ложь). Если цель содержит переменные, то Пролог выдает либо те их значения, которые приводят к решению, если оно существует, либо сообщение No solutions (решений нет).

Внешние цели заставляют внутренние унификационные подпрограммы Пролога найти все решения, удовлетворяющие цели. Внутренние цели заставляют внутренние унификационные подпрограммы останавливаться после первого успешного сопоставления цели.

В программе ["Синонимы"](http://www.verim.org/project/prolog/predikaty_v_prologe#programma_sinonimy) в самом теле программы мы использовали внутреннюю цель:

clauses

synonym(brave,daring).

*/\* внутренняя цель \*/*

goal

synonym(brave,X),

[write](http://pauillac.inria.fr/~deransar/prolog/bips.html)("A synonym for 'brave' [is](http://pauillac.inria.fr/~deransar/prolog/bips.html) "),

[nl](http://pauillac.inria.fr/~deransar/prolog/bips.html),

[write](http://pauillac.inria.fr/~deransar/prolog/bips.html)(" ",X,"."),

[nl](http://pauillac.inria.fr/~deransar/prolog/bips.html).

Само предложение, определяющее **цель**, состоит из пяти **подцелей**, разделенных запятыми.

Первая подцель:

synonym(brave,X)

**Х** здесь является **свободной переменной**, ее значение *не конкретизировано*. (Имя переменной обязательно начинается с большой буквы.) Говоря обычным языком, в данном предложении сформулирована такая цель:

*«Найти утверждение, использующее предикат synonym, такое, что первым объектом в нем является brave, и связать переменную Х с его вторым объектом».*

После запуска программы, Турбо-Пролог будет просматривать утверждения, содержащие synonym. Если таковое с объектом **brave** будет обнаружено, то **Х** примет значение второго объекта - **daring.**

**Второй подцелью** является печать следующей строки символов на экране:

A synonym for ‘brave’ is

Эта подцель образована при помощи предиката **write**, одного из многих «встроенных» предикатов Турбо-Пролога. Подобные предикаты не требуют специального описания в программе, их можно использовать сразу.

Встроенный предикат write в данной программе встречается в виде:

write("A synonym for 'brave' is "),

Двойные кавычки при этом применяются для ограничения символьной строки *A synonym for 'brave' is*; подобным образом должны выделяться все символьные строки. Предикат write может также содержать имена переменных, в этом случае кавычки не требуются.

Простейшим примером может служить:

write(X),

где **Х** - это имя переменной. Если Х принял значение daring, то write это daring и напечатает.

В обоих случаях, как символьная строка, так и переменная являются *аргументами* предиката write. Так же как и у других предикатов, аргументы write разделяются *запятыми*. Аргументы можно произвольно смешивать при условии соблюдения описываемых соглашений.

Компилятор Пролога поправит Вас, если Вы пропустите кавычки, или сделаете какую-либо другую ошибку того же порядка.

Пример смешанной записи аргументов:

write(“Today is the “,N,”th day of “,M,”, a “,D,”.”).

Этот предикат напечатает предложение

Today is the 19th day of August, a Tuesday.

(Сегодня 19 августа, вторник)

если значениями переменных N, M и D будут соответственно 19, August и Tuesday.

**Третья подцель** задается еще одним встроенным предикатом: **nl**. Предикат nl переводит курсор в начало следующей строки.

**Четвертая подцель** - печать всех трех объектов, заключенных в круглые скобки. Первый из них представляет собой обычную кавычку; второй, обозначенный как Х, - это daring; третий состоит из кавычки и точки.

**Пятая подцель**, еще один предикат nl, переводит курсор к началу строки, следующей за строкой с daring.

Предложение, описывающее цель, должно оканчиваться **точкой**; если точка будет опущена, компилятор выдаст сообщение об ошибке, и трансляция программы будет прекращена.

Результат выполнения программы:

A synonym for brave is 'daring'.

# Стандартные домены Пролога. Пролог- программы с внешней целью.

[Доменом](http://ermak.cs.nstu.ru/logprog/flp_book/glossary.html#6) в Prolog'е называют тип данных. В Prolog'е, как и других языках программирования, существует несколько стандартных доменов, перечислим их:

1. integer - целые числа.
2. real - вещественные числа.
3. string - строки (любая последовательность символов, заключенная в кавычки).
4. char - одиночный символ, заключенный в апострофы.
5. symbol - последовательность латинских букв, цифр и символов подчеркивания, начинающаяся с маленькой буквы или любая последовательность символов, заключенная в кавычки.

**Внешние цели в Прологе (Prolog)**

Если Вы запустите на счет программу, в которой будет отсутствовать описание цели или внутренняя цель, то Пролог (Prolog) попросит вас ввести цель с экрана. Такая цель является внешней целью. Программы Пролога с внешней целью называются интерактивными.

Цель применения внешних целей - дать пользователю полную свободу использования имеющихся данных. Программа в этом случае играет роль «нейтральной» базы данных.

Если, например, из программы «Синонимы» удалить раздел goal и запустить ее на счет, то на экране в окне возникнет приглашение Goal: .

Допустим, вы хотите задать вопрос: «Какое слово является синонимом слова brave?» Так же, как и при записи внутренней цели, переменная Q будет использоваться для представления значения, которое Пролог (Prolog) поставит ей в соответствие, используя утверждения базы данных. С ее помощью вопрос формулируется в виде:

synonym(brave,Q).

После выполнения внешней цели выполнение программы, однако, не завершается. Пролог (Prolog) просит ввести следующую внешнюю цель. Таким образом можно задать столько целей, сколько это представляется необходимым; чтобы остановить этот процесс нужно нажать клавишу Esc при выдаче очередного приглашения.

В формулировке запроса к программе можно использовать не только одну переменную, но и несколько. Если, к примеру, Вы введете две переменные, то программа выдаст все возможные комбинации значений, удовлетворяющие данным переменным.

Применение внешних целей бывает полезно при записи коротких целевых формулировок, а также для получения всего набора допустимых значений. Другое преимущество внешних целей - они позволяют адресовать базе данных совершенно произвольные запросы.

# Методы организации итеративных правил: метод отката после неудачи.

Вид правила, выполняющего повторение:

правило\_повторения : - <предикаты и правила>, fail.

Конструкция <предикаты и правила> в теле правила обозначает предикаты, содержащие несколько утверждений, а так же правила, определенные в программе. Встроенный предикат fail (неудача) вызывает откат, так что предикаты и правила выполняются еще раз.

Использование метода ОПН позволяет извлекать данные из каждого утверждения базы данных.

Например, утверждения *программы о служащих* содержат данные о служащих компании. Предикат базы данных имеет вид:

/\* служащий(имя, пол, отдел, почасовая\_оплата) \*/

employee(name,sex,department,pay\_rate)

Следующее правило выдает содержимое всей базы данных:

/\* выдать список служащих \*/

show\_all\_part\_time\_employees :- employee(Name,Sex,Dept,Pay\_rate), write(Name, “ “, Sex, “ “, Dept, “ “, Pay\_rate),nl, fail.

Переменные Name (имя), Sex (пол), Dept (отдел), Pay\_rate (почасовая оплата) не означены, и следовательно, все они могут получить значения. Результат выполнения этой цели - список всех служащих.

Добавив *дополнительные условия* на значения объектов для одной или более переменных предиката, можно извлекать данные только из определенных утверждений.

Например, необходимо получить список, содержащий данные только о служащих мужского пола. Для этого требуется, чтобы процесс сопоставления значений для Sex был успешным только для утверждений, содержащих M в позиции второго объекта. Так как константа сопоставима только сама с собой, то во время внутренней унификации постоянное значение M сопоставимо только с M. Правило, использующее это условие для выборки данных, имеет вид:

show\_male\_part\_time : employee(Name,”M”, Dept, Pay\_rate),

write(Name,Dept, “$”,Pay\_rate),

nl,

fail.

Альтернативной формой условия выборки по половому признаку является предикат равенства **Sex=M**. Используя этот предикат равенства, можно построить другое правило, имеющее тот же самый результат:

show\_male\_part\_time : employee(Name, Sex, Dept, Pay\_rate),

Sex=”M”,

write(Name,Dept, “$”, Pay\_rate),

nl,

fail.

Для этих правил некоторые подцели могут оказаться неуспешными из-за невозможности удовлетворить условию полового признака. Откат возникнет до момента, когда информация, содержащаяся в факте, будет выдана на экран.

Предикат fail не потребуется, если условия правил невозможно выполнить, т.е. подцель будет неуспешной сама по себе.

Предикат fail включен в правило для того, чтобы вызвать откат, если условия правила будут выполнены и все правило окажется *успешным*.

# Методы организации итеративных правил: метод отсечения и отката.

В некоторых ситуациях необходимо иметь доступ только к определенной части данных. Метод отсечения и отката (ОО) может быть использован для фильтрации данных, выбираемых из утверждений базы данных.

Предикат fail может вызвать откат к другим возможным вариантам решения задачи или подцели. Однако, для того, чтобы из базы данных выбирать данные, удовлетворяющие некоторым условиям, необходимо иметь средства управления откатом. Для этих целей Пролог имеет встроенный предикат **cut (отсечение)**. Предикат cut обозначается символом восклицательного знака (!). Этот предикат, вычисление которого *всегда завершается успешно*, заставляет внутренние унификационные подпрограммы «забыть» все указатели отката, установленные во время попыток вычислить текущую подцель.

Другими словами, предикат cut «устанавливает барьер», запрещающий выполнить откат ко всем альтернативным решениям текущей подцели. Однако последующие подцели могут создать новые указатели отката, и тем самым создать условия для поиска новых решений. Область действия предикат cut на них уже *не распространяется*. Но если **все** более поздние цели окажутся неуспешными, то барьер, установленный предикатом cut, заставит механизм отката отсечь все решения в области действия cut путем немедленного отката к другим возможным решениям вне области действия cut.

Метод отсечения и отката использует предикат fail для того, чтобы имитировать неуспешное вычисление и выполнять последующий откат, до тех пор, пока не будет обнаружено определенное условие. Предикат cut служит для устранения всех последующих откатов.

Метод отката и отсечения демонстрирует простая ситуация, в которой предикаты базы данных содержат несколько имен, как это имеет место для предикатов child (ребенок) в программе, формирующей список имен детей. Предположим, что необходимо выдать список имен до имени Diana включительно.

**Предикат cut** выполнит отсечение в указанном месте. **Предикат fail** используется для продолжения откатов и доступа к последовательности имен базы данных до элемента с именем Diana. Таким образом, поставленную задачу выполняет соответствующая комбинация предикатов cut и fail. Эта комбинация называется **методом отката и отсечения (ОО)**.

Правило, обеспечивающие генерацию всех имен (а не только некоторых) основано на методе ОПН и имеет вид:

show\_them\_all :- child(Name), write("Имя: ", Name), nl, fail.

Чтобы использовать предикат cut, необходимо определить некоторое условие **Name=Diana**. Правило, определяющее, это условие имеет вид:

make\_cut(Name) :- Name="Diana".

Это правило с последующим предикатом cut (!) образует правило make\_cut (сделать отсечение). Теперь выполнение программы будет неуспешным, а откаты будут повторяться до тех пор, пока Name не окажется равным Diana. В этот момент результат выполнения правила make\_cut будет успешным, что в свою очередь, вызовет выполнение предиката cut.

Таким образом, комбинируя правила отсечения с методом ОПН, можно получить ОО-правило:

show\_some\_of\_them :- child(Name), write("Имя: ", Name), nl, make\_cut(Name),!.

make\_cut(Name) :- Name="Diana".

Этот способ использования ОО-метода называется **методом ранжированного отсечения**.

# Методы организации итеративных правил. Метод повтора, определяемый пользователем.

**Метод повтора** (МП), как и Метод отсечения и отката (ОО), использует откат. Но в МП-методе выполнить откат возможно всегда в отличии от ОО-метода, где откат выполняются только после искусственного созданного неуспешного результата.

**Правило рекурсии общего вида** имеет более сложную структуру и является обобщением этих методов. Вид правила повтора, определяемого пользователем, следующий:

repeat. /\* повторить \*/

repeat :- repeat.

**Первый repeat** является утверждением, объявляющим предикат repeat истинным. Первый repeat не создает подцелей, поэтому данное правило всегда успешно. Однако, поскольку имеется еще один вариант для этого правила, то указатель отката устанавливается на первый repeat.

**Второй repeat** - это правило, которое использует само себя как компоненту (третий repeat). Второй repeat вызывает третий repeat, и этот вызов вычисляется успешно, так как первый repeat удовлетворяет подцели repeat. Следовательно, правило repeat так же всегда успешно.

Предикат repeat будет вычисляться успешно при каждой новой попытке его вызвать после отката. Факт в правиле будет использоваться для выполнения всех подцелей программы. Таким образом, **repeat** это рекурсивное правило, которое никогда не бывает неудачным. Правило repeat широко используется в качестве компоненты других правил.

**Пример**. Создадим *предикат*, который будет дублировать символ, введенный пользователем с клавиатуры. Завершиться этот процесс должен, когда *пользователь* введет некий ключевой символ, о котором мы заранее договоримся, что его появление означает окончание процесса дублирования символов. Давайте, для определенности, возьмем в качестве такого символа точку (.).

double\_char:–

repeat,

readchar(C), /\* читаем символ с клавиатуры

в переменную C \*/

write(C,C), nl,/\* выводим на экран значение

переменной C \*/

C=’.’,!, /\* сравниваем значение переменной C

с символом ‘.’\*/

nl,write("Была введена точка. Закончили.").

Первой подцелью нашего правила записан вызов предиката repeat. Он обеспечивает нам повторное выполнение следующих за ним подцелей. Можно эксперимента ради закомментировать *предикат* repeat, дабы убедиться, что в этом случае *цикла* не получится. Последующие подцели выполнятся всего один раз.

Далее, используя стандартный *предикат* readchar, осуществляем чтение символа с клавиатуры в переменную C. Посредством встроенного предиката write выводим два экземпляра введенного пользователем символа на экран; стандартным предикатом nl переводим *курсор* на следующую строку. Затем *значение* переменной C сравнивается с символом точка ( ‘.’ ). Это условие, которое, с одной стороны, обеспечивает *откат* на первую подцель (*предикат* repeat ), а с другой обеспечивает *выход* из *цикла*. Если поступивший с клавиатуры символ отличен от точки, подцель C=’.’ терпит неуспех. *Пролог*-система осуществляет *откат* на последнее *место*, *указатель* на которое записан в стеке точек возврата. Из подцелей, расположенных левее, альтернативы имелись только у первой подцели (предиката repeat ). Все повторяется еще раз. *Пользователь* вводит символ, он дважды отображается на экране, сравнивается с точкой. Процесс повторяется до тех пор, пока введенный символ не окажется точкой. В этом случае подцель C=’.’ окажется успешной, управление перейдет на подцели, расположенные правее. *Предикат* nl сдвинет *курсор* на начало следующей строки, стандартный *предикат* write выводит на экран сообщение: "Была введена точка. Закончили." — и процесс дублирования символов на этом завершается.

# Рекурсивные правила. Общая схема рекурсивного выполнения предиката.

Рекурсия - это средство для организации повторяющихся действий в Prolog'е. Рекурсивная процедура - это процедура, вызывающая сама себя до тех пор, пока не будет соблюдено некоторое условие, которое остановит рекурсию. Такое условие называют граничным. Рекурсия - хороший способ для решения задач, содержащих в себе подзадачу такого же типа.

Рекурсия является часто используемым приемом в программах на Прологе. Для рассмотренного выше примера базы знаний «Родственники» введем правило, описывающее отношение «предок» - «потомок» с помощью рекурсии.

|  |
| --- |
| % 1 параметр - имя отца или матери % 2 параметр - имя сына или дочери roditel('Вася', 'Вика'). roditel('Вася', 'Коля'). roditel('Коля', 'Юля'). roditel('Юля', 'Миша'). roditel('Юля', 'Маша'). roditel('Коля', 'Света'). roditel('Света', 'Рома'). roditel('Света', 'Леша'). % 1 параметр - предок % 2 параметр - потомок predok(A, B) :- roditel(A, B). predok(A, B) :- roditel(A, C), predok(C, B). |

На вопрос predok('Вася', 'Миша') ответ будет положительным.

**Второй пример** использования рекурсии – расчет факториала.

|  |
| --- |
| fact(1, 1):- !. % факториал единицы равен единице fact(N, F):- N1 is N - 1,  fact(N1, F1), % F1 равен факториалу числа на единицу меньшего N  F is F1 \* N. % факториал числа N равен произведению F1 на само число N |
|  |

На вопрос fact(6, F) ответ будет «F = 720».

Любая рекурсивная процедура в Прологе должна включать, как минимум два правила:

1) нерекурсивное правило, определяющее его вид в момент прекращения рекурсии;

2) рекурсивное правило. Первая подцель, вырабатывает новые значения аргументов, а вторая - вызов самого правила с новыми значениями аргументов.

# Предикат «отсечение»; Использование в логических программах.

Отличительной способностью ПроЛога является то, что при поиске решений на ПроЛоге выдается целый список подходящих значений. Поэтому, в ПроЛоге один предикат может быть представлен целым набором правил, каждое из которых может выдавать ответ. Пример. Пусть есть следующее описание предиката "число\_больше":

число\_больше(А,3):-А<3.

число\_больше(А,5):-А<5.

число\_больше(А,8):-А<8.

Тогда вызов решателя ПроЛога

Goal число\_больше(4,Больше).

Выдаст ответы: Больше = 5, 8.

Другими словами, согласно правилам в предикате "число\_больше" больше числа 4 есть два числа 5 и 8. Здесь видно, что набор правил работает как структура ветвления case в алгоритмических языках.

Теперь введем понятие отсечения. Отсечением в ПроЛоге называется механизм, который запрещает перебор правил данного предиката находящихся ниже текущего правила и запрещает механизм отката. Отсечение обозначается знаком "!". Пример. Если переписать предыдущий предикат "число\_болше" как:

число\_больше(А,3):-А<3,!.

число\_больше(А,5):-А<5,!.

число\_больше(А,8):-А<8,!.

Тогда вызов решателя ПроЛога

Goal число\_больше(4,Больше).

Выдаст ответ: Больше = 5.

Это происходит потому, что в каждом правиле предиката, после проверки на больше, идет оператор отсечения "!", что запрещает ПроЛогу последующий откат и поиск других вариантов ответа.

# Ложные логические программы.

Ложная программа – это программа, которая выдает неправильный ответ, то есть неправильно описан алгоритм.

*Пример ложной программы:*

Min(X,Y,X):- X<=Y,!.

Min(X,Y,Y).

*Пример верной программы:*

Min(X,Y,X):-X<=Y,!.

Min(X,Y,Y):-X>Y,!.

Первые два аргумента в данных программах – сравниваемые числа. Третий – предполагаемый результат.

Если в качестве входных параметров использовать комбинацию (1,0,0), то обе программы выведут правильный ответ (YES).

Но если ввести комбинацию (1,3,3), то первая программа (ложная) выведет неверный ответ (YES), т.к. во втором варианте предиката Min нет дополнительной проверки. Вторая программа выведет правильный ответ (No).

# Рекурсия как один из подходов к решению сложных задач. Список как рекурсивно определяемая структура данных.

Рекурсия - это средство для организации повторяющихся действий в Prolog'е. Рекурсивная процедура - это процедура, вызывающая сама себя до тех пор, пока не будет соблюдено некоторое условие, которое остановит рекурсию. Такое условие называют граничным.

*Пример рекурсии: найти факториал n!.*

PREDICATES

factorial (integer, integer)

CLAUSES

%факториал 0! равен 1

factorial (0, 1):- !.

%факториал n! равен факториалу (n-1)!, умноженному на n

factorial (N, Factorial\_N):- M=N-1, factorial (M, Factorial\_M),

Factorial\_N=Factorial\_M\*N.

GOAL

write ("Для какого числа Вы хотите найти факториал? "), readint (Number),

factorial (Number, Result), write (Number, "!=", Result).

Результат работы программы: 3!=6

Список — это структура данных, определяемая следующим образом:

* пустой список ([ ]) является списком;
* структура вида [H|T] является списком, если H — первый элемент списка (или несколько первых элементов списка, перечисленных через запятую), а T — список, состоящий из оставшихся элементов исходного списка.

Данное определение позволяет организовывать рекурсивную обработку списков, разделяя непустой список на голову и хвост.

# Особенность использования списков (регулярность структуры, единый механизм доступа к элементам списка; разделение списка на «голову» и «хвост»; частные случаи списков).

Пролог поддерживает связанные объекты, называемые **списками**. **Список** - это упорядоченный набор объектов, следующих друг за другом. Составляющие списка внутренне связаны между собой, поэтому с ними можно работать и как с группой (списком в целом), так и как с индивидуальными объектами (элементами списка).

Пролог позволяет выполнять со списком целый ряд операций:

* доступ к объектам списка;
* проверка на принадлежность к списку;
* разделение списка на два;
* слияние двух списков;
* сортировку элементов списка в порядке возрастания или убывания.

Списки бывают полезны при создании баз знаний (баз данных), экспертных систем, словарей.

**Список** является набором *объектов* одного и того же доменного типа. Объектами списка могут быть:

* целые числа;
* действительные числа;
* символы;
* символьные строки;
* структуры.

Порядок расположения элементов является *отличительной чертой* списка; те же самые элементы, упорядоченные иным способом, представляют уже совсем другой список. Порядок играет важную роль в процессе сопоставления.

Пролог допускает списки, элементами которых являются *структуры*. Если структуры принадлежат к *альтернативному домену*, элементы списка могут иметь разный тип. Такие списки используются для специальных целей.

Совокупность элементов списка заключается в квадратные скобки (**[]**), а друг от друга элементы отделяются *запятыми*. Примерами списков могут служить:

[1,2,3,6,9,3,4]

[3.2,4.6,1.1,2.64,100.2]

["YESTERDAY","TODAY","TOMORROW"]

Начало формы

Конец формы

Атрибуты списка

Объекты списка называются *элементами списка*. Список может содержать произвольное число элементов, единственным ограничением является лишь объем оперативной памяти.

Пролог требует, чтобы все элементы списка принадлежали к одному и тому же типу доменов: либо все элементы списка - целые числа, либо все - действительные, либо все - символы, либо - символьные строки.

В Прологе список [«JOHN WALKER»,3.50,45.50] некорректен ввиду того, что составлен из элементов разных типов. Списки структур являются исключением из правила.

Количество элементов в списке называется его **длиной**. Длина списка [«MADONNA»,»AND»,»CHILD»] равна 3. Длина списка [4.50,3.50,6.25,2.9,100.15] равна 5. Список может содержать всего один элемент и даже не содержать элементов вовсе:

[«summer»]

[]

Список, не содержащий элементов, называется *пустым* или *нулевым списком*.

Непустой список можно рассматривать как состоящий из двух частей:

1. первый элемент списка - его *голова*;
2. остальная часть списка - *хвост*.

Голова является элементом списка, хвост есть список сам по себе. Голова - это отдельное неделимое значение. Наоборот, хвост представляет из себя список, составленный из того, что осталось от исходного списка в результате «усекновения головы». Этот новый список зачастую можно делить и дальше.

Если список состоит из одного элемента, то его можно разделить на голову, которой будет этот самый единственный элемент, и хвост, являющийся пустым списком.

В списке [4.50,3.50,6.25,2.9,100.15] например, головой является значение 4.50, а хвостом - список [3.50,6.25,2.9,100.15] Этот список в свою очередь имеет и голову, и хвост. Голова - это значение 3.50, хвост - список [6.25,2.9,100.15]

**Таблица.** Головы и хвосты различных списков

|  |  |  |
| --- | --- | --- |
| **Список** | **Голова** | **Хвост** |
| [1,2,3,4,5] | 1 | [2,3,4,5] |
| [6.9,4.3,8.4,1.2] | 6.9 | [4.3,8.4,1.2] |
| [cat,dog,horse] | cat | [dog,horse] |
| [’S’,’K’,’Y’] | ‘S’ | [’K’,’Y’] |
| [«PIG»] | «PIG» | [] |
| [] | не определена | не определен |

Отличительной особенностью описания списка является наличие звездочки (\*) после имени домена элементов. Так запись bird\_name \* указывает на то, что это **домен списка**, элементами которого являются bird\_name, т. е. запись bird\_name \* следует понимать как список, состоящий из элементов домена bird\_name.

Описание в разделе domains, следовательно, может выглядеть либо как:

bird\_list = bird\_name \*

bird\_name = symbol

либо как

bird\_list = symbol \*

Домен bird\_list является доменом списка элементов типа symbol (списка птиц).

В разделе описания предикатов predicates требуется присутствия имени предиката, а за ним заключенного в круглые скобки имени домена.

birds(bird\_list)

# Оформление Пролог- программ, работающих со списками (на примере предиката, проверяющего вхождение отдельного элемента в список).

Для применения списков в программах на Прологе необходимо описать домен списка в разделе *domains*, предикаты, работающие со списками необходимо описать в разделе *predicates*, задать сам список можно либо в разделе *clauses* либо в разделе *goal*.

**Поиск элемента в списке**

Поиск элемента в списке является очень распространенной операцией. Поиск представляет собой просмотр списка на предмет выявления соответствия между объектом поиска и элементом списка. Если такое соответствие найдено, то поиск заканчивается успехом, в противном случае поиск заканчивается неуспехом. Стратегия поиска при этом будет состоять в рекурсивном выделении головы списка и сравнении ее с объектом поиска.

*Пример:*

*domains*

*list=integer\**

*predicates*

*member (integer, list)*

*clauses*

*member (Head, [Head |\_ ]).*

*member (Head, [\_ | Tail ]):- member (Head, Tail).*

*goal*

*member (3, [1, 4, 3, 2]).*

Правило поиска может сравнить объект поиска и голову текущего списка, эта операция записана в виде факта предиката member. Этот вариант предполагает наличие соответствия между объектом поиска и головой списка. Отметим, что хвост списка в данном случае не важен, поэтому хвост списка присваивается анонимной переменной. Если объект поиска и голова списка различны, то в результате исполнения первого предложения будет неуспех, происходит возврат и поиск другого правила или факта, с которыми можно попытаться найти соответствие. Для этой цели служит второе предложение, которое выделяет из списка следующий по порядку элемент, то есть выделяет голову текущего хвоста, поэтому текущий хвост представляется как новый список, голову которого можно сравнить с объектом поиска. В случае исполнения второго предложения, голова текущего списка ставится в соответствие анонимной переменной, так как значение головы с писка в данном случае не играет никакой роли.

# Явное и неявное определение граничных условий при работе со списками (на примере предиката проверки вхождения элемента в список или предиката проверки совпадения двух списков).

Программа проверки вхождения элемента в список

‘содерж’(H, [H|T]):-!.

‘содерж’(H, [R|T]):- ‘содерж’(H, T).

Первая альтернатива и есть граничное условие (явное) (когда голова списка совпадает с искомым элементом).

# Декларативная интерпретация предиката слияния («присоединить») двух списков.

'присоединить'([],Y,Y).  
'присоединить'([Х1|L1],Y,[Х1|L3]):- 'присоединить'(L1,Y,L3).

Пока первый список не опустеет, считываем его хвост поэлементно.

После этого записать второй список в конец третьего, потом с конца первого списка дописывать по элементу в начало третьего. Когда запишется последний элемент первого списка, третий список будет сформирован.

# Процедурная интерпретация предиката слияния («присоединить») двух списков.

'присоединить'([],Y,Y).  
'присоединить'([Х1|L1],Y,[Х1|L3]):- 'присоединить'(L1,Y,L3).

Разделяем первый список на голову и хвост. Считываем голову первого списка и рекурсивно вызываем предикат «Присоединить» для оставшегося хвоста первого списка (второй и третий аргументы не изменяем).

Когда первый список опустеет, сработает первая альтернатива и в третий список занесется второй. После этого по хвостовой рекурсии добавляем к голове третьего списка голову первого списка на каждом уровне рекурсии. Таким образом сформируется третий список, первая часть которого – первый список, а вторая – второй.

# Варианты использования предиката слияния («присоединить») двух списков (определение префикса, суффикса, элемента списка).

'присоединить'([],Y,Y).  
'присоединить'([Х1|L1],Y,[Х1|L3]):- 'присоединить'(L1,Y,L3).

Предикат «присоединить» так же можно использовать, чтобы определить совпадение суффикса, префикса, или элемента в списке.

*Суффикс:*

‘суффикс’ (ПС,С):- ‘присоединить’ (\_,ПС,С)

Программа действует следующим образом:

Рекурсивно считывает голову третьего списка, пока второй список не станет совпадать с оставшейся частью третьего списка, тогда программа выдаст сообщение об успешном нахождении.

*Префикс:*

‘префикc’ (ПС,С):- ‘присоединить’ (ПС,\_,С)

Программа действует следующим образом:

Рекурсивно считываем голову третьего и первого списка. Когда первый список опустеет, программа выдаст сообщение об успешном нахождении.

*Элемента списка:*

‘элемент’ (I,С):- ‘присоединить’ (\_,[I|L],С)

Программа определяет, принадлежит ли элемент списку.

# Компоновка данных в список (предикат «findall»).

Иногда, при программировании определенных задач, возникает необ-

ходимость собрать данные из базы данных в список для последующей их

обработки. Турбо-Пролог содержит встроенный предикат, позволяющий

справиться с этой задачей без каких бы то ни было хлопот. Таким предика-

том является предикат findall. Требуемый список представляется означенной

переменной, являющейся одним из объектов предиката.

Предописание встроенного предиката findall выглядит следующим

образом:

**findall(Variable\_name,Predicate\_expression,List\_name).**

**Variable\_name** обозначает здесь объект входного предиката

**Predicate\_expression**, а **List\_name** является именем переменной выходного

списка. Переменная должна относиться к домену списков, объявленному в

разделе domains.

Для пояснения только что сказанного рассмотрим предикат базы дан-

ных

football(name,points)

Этот предикат порождает 5 утверждений :

football("Ohio State",116.0). football("Michigan",121.0).

football("Michigan State",114.0). football("Purdue",99.0).

football("UCLA",122.0).

Эти утверждения содержат сведения об очках, набранных командами.

Предположим, что необходимо сложить все очки и усреднить их. Сбор очков

в список осуществляется при помощи встроенного предиката findall :

findall(Points,football(\_,Points),Point\_list)

Здесь Poits является свободной переменной для значений набранных коман-

дами очков, а Point\_list - списочной переменной, элементы которой принад-

лежат к тому же домену, что и Points, в данном случае, к домену real.

Сама работа предиката скрыта от глаз пользователя. В нашем приме-

ре findall просматривает все утверждения с предикатом football, начиная с

первого. Значение переменной Points (116), взятое из этого утверждения,

присваивается голове списка Point\_list. Остальные значения Points помеща-

ются в список на последующие позиции. По завершению работы findall

переменная Point\_list принимает значение

[116.0,121.0,114.0,99.0,122.0]

# Структуры в Прологе. Описание альтернативных однодоменной и многодоменной структур.

Cтруктурные объекты (или просто структуры) - это объекты, которые состоят из нескольких компонентов. Эти компоненты, в свою очередь, могут быть структурами.

Основной характеристикой структуры является ее размерность (или арность) - число термов в списке.

Структуры в программе ведут себя как единые объекты. Структура представляется на языке Пролог с помощью указания её функтора и компонент в следующем виде:

Имя структуры = функтор(компонента-1, ...., компонента-N), где в качестве функтора должен выступать атом, а компонентой может быть любой терм (в том числе и структура).

Если объекты структуры относятся к одному и тому же типу доменов, то такая структура называется однодоменной. Если объекты структуры относятся к разным типам доменов – многодоменная. Объекты могут быть константами, переменными или структурами.

Для того, чтобы объединить объекты в структуры, надо выбрать функтор. Каждый функтор определяется двумя параметрами: именем, синтаксис которого совпадает с именем предиката; арностью – т. е. числом аргументов.

Рассмотрим в структуру date:

date (1, september, 2002), где date – функтор, все аргументы данной многодоменной структуры являются константами.

date (Day, september,2002)

В этом примере date представляет функтор многодоменной структуры, аргументы которой являются переменные и константы.

likes ('Katja', fruits(banana, apples, oranges)),

А в этом примере likes – главный функтор, аргументы данной многодоменной структуры – константа и структура. Рассмотренную структуру в программе можно описать следующим образом:

domains

personal\_liking = fruits(type1,type2, type3)

type1, type2, type3 = symbol

predicates

likes(symbol, personal\_liking)

# Структуры в Прологе. Доменное описание бинарного дерева.

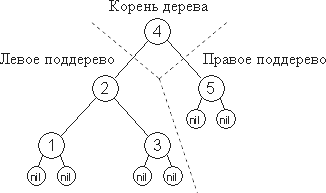
В Prolog'е можно определять рекурсивные типы данных. Например, можно определить дерево следующим образом:

DOMAINS

treetype=tree(integer, treetype, treetype); nil()

Такое определение говорит о том, что дерево является составным объектом, состоящим из трех составных частей: корня, принадлежащего домену integer и двух поддеревьев, принадлежащих домену treetype, так как именно этот домен и описывает структуру данных типа дерево. Так как дерево является составным объектом, его составные части объединяет функтор tree. Кроме того, дерево может находиться в двух состояниях: быть непустым (иметь хотя бы один корень) или пустым (не иметь ни одного корня). Пустое дерево описывается функтором nil без параметров. Если у функтора нет параметров, пустые скобки можно не указывать и записывать только имя функтора. Имена функторов tree и nil, домена treetype выбраны произвольно.

Узлы дерева, которые содержат значения 1, 3 и 5, не имеют левых и правых поддеревьев. Такие узлы можно назвать листьевыми узлами или листьевыми вершинами.



Такое определение позволяет записать следующую структуру данных:

tree (4,

tree (2,

tree (1, nil, nil),

tree (3, nil, nil)),

tree (5, nil, nil))

# Структуры в Прологе. Префиксное представление бинарного дерева.

Exp=

Aand(exp,exp)

Oor(exp,exp)

Imp(exp,exp)

Nnot(exp)

Str(string)

Int(integer)

Дерево:

Imp(aand(str(b),nnot(str(b))),oor(str(a),str(b)))

# Решение задач на графах: варианты описания графа на Прологе.

Графы используются во многих приложениях, например для представления отношений, ситуаций или структур задач. Граф определяется как множество вершин вместе с множеством ребер, причем каждое ребро задается парой вершин. Если ребра направлены, то их также называют дугами. Дуги задаются упорядоченными парами. Такие графы называются направленными. Ребрам можно приписывать стоимости, имена или метки произвольного вида, в зависимости от конкретного приложения.

В Прологе графы можно представлять различными способами. Один из них — каждое ребро записывать в виде отдельного предложения. Например, графы, показанные на рис. 9.18, можно представить в виде следующего множества предложений:

связь( а, b).

связь( b, с).

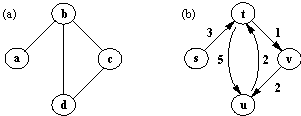
...

дуга( s, t, 3).

дуга( t, v, 1).

дуга( u, t, 2).

...



Другой способ — весь граф представлять как один объект. В этом случае графу соответствует пара множеств — множество вершин и множество ребер. Каждое множество можно задавать при помощи списка, каждое ребро — парой вершин. Для объединения двух множеств в пару будем применять функтор граф, а для записи ребра — функтор p. Тогда (ненаправленный) граф рис. 9.18 примет вид:

G1 = граф( [a, b, c, d],

[p( а, b), p( b, d), p( b, с), p( c, d)] )

Рис. 9.18. (а) Граф. (b) Направленный граф. Каждой дуге приписана ее стоимость.

Для представления направленного графа (рис. 9.18), применив функторы диграф и д (для дуг), получим

G2 = диграф( [s, t, u, v],

[д( s, t, 3), д( t, v, 1), д( t, u, 5), д( u, t, 2),

д( v, u, 2) ] )

Если каждая вершина графа соединена ребром еще по крайней мере с одной вершиной, то в представлении графа можно опустить множество вершин, поскольку оно неявным образом содержится в списке ребер.

Еще один способ представления графа — связать с каждой вершиной список смежных с ней вершин. В этом случае граф превращается в список пар, каждая из которых состоит из вершины- плюс ее список смежности. Наши графы (рис. 9.18), например, можно представить как

G1 = [ a->[b1, b->[a, c, d], c->[b, d], d->[b, c] ]

G2 = [s->[t/3], t->[u/5, v/l], u->[t/2], v->[u/2]]

Здесь символы '->' и '/' — инфиксные операторы.

Какой из способов представления окажется более удобным, зависит от конкретного приложения, а также от того, какие операции имеется в виду выполнять над графами. Вот типичные операции:

• найти путь между двумя заданными вершинами;

• найти подграф, обладающий некоторыми заданными свойствами.

Примером последней операции может служить построение основного дерева графа. В последующих разделах, мы рассмотрим некоторые простые программы для поиска пути в графе и построения основного дерева.

# Решение задач на графах (на примере задачи поиска пути в графе).

# Трассировка Пролог- программ (в терминах четырех видов происходящих событий: CALL, RETURN, REDO, FALL).

|  |  |
| --- | --- |
| CALL | Когда вызывается предикат, его имя и значения его параметров выводятся в окне Trace. |
| RETURN | Когда предикат (или правило) удовлетворяется, в окне Trace выводится RETURN и возвращаемый предикат. Если имеются другие предложения, соответствующие входным пара­метрам, то появляется изображение звездочки (\*), пока­зывающее, что имел место поиск с возвратом (недетермини­рованный). |
| FAIL | Когда предикат не удовлетворяется, перед именем неудачно Завершившегося предиката появляется слово FAIL. |
| REDO | REDO показывает, что произведен поиск с возвратом. Имя повторно выполняющегося предиката появляется в окне Trace наряду со значениями его параметров. |

# Общие сведения о языке функционального программирования Лисп: основные понятия и определения. Правила записи программ на языке Лисп.

Язык ЛИСП (LISP) был разработан в 1958 году американским ученым Джоном Маккарти как функциональный язык, предназначенный для обработки списков. (LISt Processing). Лисп был создан для работ по искусственному интеллекту и до сих пор остаётся одним из основных инструментальных средств в данной области.

Наряду с языком Ада, Лисп прошёл процесс фундаментальной стандартизации для использования в военном деле и промышленности, в результате чего появился диалект Common Lisp, впоследствии стандартизованный ANSI. Его реализации существуют для большинства платформ.

**Основу** ЛИСПа составляют символьные выражения, которые   
называются **S-выражениями** и образуют область определения  
для функциональных программ.

**S-выражение (Simbolic expresion)** - основная структура данных  
в ЛИСПе.

**(ДЖОН СМИТ 33 ГОДА)** \

S-ВЫРАЖЕНИЯ

**((МАША 21) (ВАСЯ 24) (ПЕТЯ 1))** /

S-выражение - это либо атом, либо список.

**Атомы** - это простейшие объекты Лиспа, из которых  
строятся остальные структуры.

Атомы бывают двух типов - **символьные и числовые**.

**Символьные атомы** - последовательность букв и цифр,  
при этом должен быть по крайней мере один символ   
отличающий его от числа.

**ДЖОН АВ13 В54 10А**

Символьный атом или **символ** - это не идентификатор   
переменой в обычном языке программирования.  
Символ как правило обозначает какой либо предмет, объект  
вещь, действие.

Символьный атом рассматривается как неделимое целое.

К символьным атомам применяется только одна операция:  
**сравнение**.

В МCL в состав символа могут входить:

**+ - \* / @ $ % ^ \_ \ <>**

**Числовые атомы** - обыкновенные числа

**124** **– 344** **4.5 3.055Е8**

Числа это константы.

Типы чисел зависят от реализации ЛИСПа

Атом - простейшее **S-выражение**.

В ЛИСПЕ **список** это последовательность элементов **(list)**.  
Элементами являются или атомы или списки.  
Списки заключаются в скобки, элементы списка разделяются  
пробелами.

**(банан)**1 атом

**(б а н а н)** 5 атомов

**(a b (c d) e)**4 элемента

Таким образом список - это многоуровневая или иерархическая структура данных, в которой открывающиеся и закрывающиеся скобки находятся в строгом соответствии.

**(+ 2 3)** 3 атома

**(((((первый) 2) второй) 4) 5)**2 элемента

Список, в котором нет ни одного элемента, называется **пустым  
списком** и обозначается "**()**" или символом **NIL**.

**NIL** - это и список и атом одновременно.

Пустой список играет такую же важную роль в работе со списками,  
что и ноль в арифметике.

Пустой список может быть элементом других списков.

**(NIL)**;список состоящий из атома NIL

**(())**;то же самое, что и (NIL)

**((()))**;- " -((NIL))

**(NIL ())** ;список из двух других списков

# Схема рекурсивного выполнения Лисп - программы.

Пусть имеется **произвольный** список, состоящий из чисел. Нужно подсчитать сумму чисел, входящих в список. Как это сделать? Длина списка (количество его элементов) заранее не известна. Читателям, знакомым с традиционными языками программирования, возможно, покажется, что нужно завести переменную для будущей суммы, а затем, перебирая элемент за элементом наш список, копить сумму. Процесс этот повторять до завершения списка. Так поступить действительно можно, но непонятно, как организовать циклическое повторение...   
  
На самом деле, поставленная задача допускает простое и элегантное решение. Обозначим нашу будущую функция **SUMLIST**. Тогда, если входной список **пуст**, то функция должна возвращать **нуль**. Это понятно. А если список **непуст**, то представляется вполне очевидным, что сумма элементов списка равна его первому элементу (**CAR**) плюс сумма элементов остатка (**CDR**). Другими словами, нашу функцию можно представить так: 

|  |
| --- |
| **(DEFUN SUMLIST (x) (COND ((NULL x) 0)**  **(T (+ (CAR x) (SUMLIST (CDR x))))))**  **==> sumlist** |

Поскольку эта функция - первый нетривиальный пример лисповской функции, опишем ее подробнее. Тело функции представляет собой **COND**-конструкцию из двух условий. Первое условие проверяет, не пуст ли список, поданный на вход функции. Если значение выражения **(NULL x)** истино (равно **T**), то функция возвращает нуль. В противном случае происходит переход к проверке второго условия. На месте второго условного выражения стоит **T** (условие всегда истино), поэтому вычисляется сумма **первого элемента** списка **x** и значения функции **SUMLIST**, примененной к списку **x** без первого элемента. Все! Задача решена.   
  
Пусть читатель обратит внимание на **лаконичность** определения функции. Всего две строки, причем **очень понятного кода**! В каком еще языке программирования такое возможно?!   
  
Чтобы убедиться в работоспособности нашей функции, попробуем вычислить сумму списка **(1 2 3 4 5)**: 

|  |
| --- |
| **(sumlist '(1 2 3 4 5))**  **==> 15** |

Функция работает. Можно убедиться, что функция будет работать правильно с любым одноуровневым списком чисел. Следует отметить, что функция **SUMLIST** вызывает сама себя. Такие функции называются **рекурсивными**.   
  
Может возникнуть вполне естственный вопрос: почему функция, вызывающая сама себя, не "зацикливается"? Это происходит потому, что в теле функции **первым условием** стоит проверка значения входного параметра на пустоту. Если значение входного параметра есть пустой список (**Nil**), то происходит выход из функции с возвратом значения **0**. А теперь пусть читатель обратит внимание на то, что повоторное обращение к функции **SUMLIST** происходит не к исходному списку, а к его **остатку** (после отделения головы). Последующее обращение происходит с остатком остатка и т.д. Поскольку список имеет конечное число элементов, то рано или поздно остаток окажется пустым списком. Это гарантирует завершение цепочки вызовов.   
  
Обычно тело рекурсивной функции состоит из проверки различных условий. Условие, гарантирующее выход, называется **терминальным условием**. Обычно, хотя и не обязательно, терминальное условие располагается в списке условий первым (как в функции **SUMLIST**). Что произойдет, если терминальное условие будет опущено? Ответ на этот вопрос дает следующая врезка: 

|  |
| --- |
| **(DEFUN BAD\_SUMLIST (x) (+ (CAR x) (BAD\_SUMLIST (CDR x))))**  **==> bad\_sumlist**  **(bad\_sumlist '(1 2 3))**  **Переполнение внутреннего стека**  **==> ERRSTATE** |

Здесь вводится определение неправильной рекурсивной функции (без терминальной ветви). Функция принимается ядром Лиспа, однако, попытка вычислить сумму даже простого списка из трех элементов вызывает ошибку с соответствующей диагностикой.

# Основные функции обработки списков языка Лисп: CAR, CDR, CONS.

CONS принимает 2 аргумента и возвращает новую cons-ячейку, содержащую 2 значения. Эти значения могут быть ссылками на объект любого типа. Если второе значение не NIL и не другая cons-ячейка, то ячейка печатается как два значения в скобках, разделённые точкой (так называемая точечная пара).

(cons 1 2) ==> (1 . 2)

Два значения в cons-ячейке называются CAR и CDR - по имени функций, используемых для доступа к ним:

(car (cons 1 2)) ==> 1

(cdr (cons 1 2)) ==> 2

Т.к. значения в cons-ячейке могут быть ссылками на любой объект, вы можете строить большие структуры, связывая cons-ячейки между собой. Списки строятся связыванием cons-ячеек в цепочку. Элементы списка содержатся в CAR cons-ячеек, а связи со следующими cons-ячейками содержатся в их CDR. Последняя ячейка в цепочке имеет CDR со значением NIL, которое представляет собой как пустой список, так и булево значение ложь.

CAR cons-ячейки является первым элементом списка, а CDR является ссылкой на другой список (который в свою очередь также является cons-ячейкой), содержащий остальные элементы, или значение NIL. Lisp-печаталка понимает это соглашение и печатает такие цепочки cons-ячеек как списки в скобках, нежели как точечные пары.

(cons 1 nil) ==> (1)

(cons 1 (cons 2 nil)) ==> (1 2)

(cons 1 (cons 2 (cons 3 nil))) ==> (1 2 3)

# Основные функции языка Лисп: ATOM, NULL, LIST.

При работе с выражениями необходимо иметь возможность проверить,

является ли выражение атомом или списком. Это может потребоваться, например, перед применением функций CAR и CDR, так как эти функции определены лишь для аргументов, являющихся списками. Базовый предикат АТОМ используется для идентифицирования лисповских объектов, являющихся атомами:

(АТОМ s-выражение)

Значением вызова АТОМ будет Т, если аргументом является атом, и () — в противном случае:

(atom 'x) 🡪 T

(atom '(a b c)) 🡪 ()

(atom '(Я программирую - следовательно существую)) 🡪 ()

(atom (cdr '(a b c))) 🡪 ()

(atom (car '(a b c))) 🡪 T

(atom (+ 5 6)) 🡪 T

(atom nil) 🡪 T

(atom ‘(nil)) 🡪 ()

NULL – проверяет является ли элемент пустым списком:

(NULL ‘()) 🡪 T

(NULL (CDR ‘(one two))) 🡪 nil

(NULL nil) 🡪 T

(NULL T) 🡪 nil

В основном при работе со списками вы не обязаны иметь дело с отдельными ячейками, т.к. существуют функции, которые создают списки и используются для манипулирования ими. Например, функция LIST строит cons-ячейки и связывает их вместе, следующие LISP-выражения эквивалентны CONS-выражениям, приведённым выше:

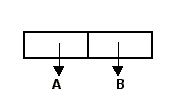
(list 1) ==> (1)

(list 1 2) ==> (1 2)

(list 1 2 3) ==> (1 2 3)

# Машинное представление списочных структур.

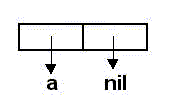
Оперативная память, в которой хранятся S-выражения, обычно делится на две больших области: **список объектов** и **область списочных ячеек**.   
  
В списке **объектов** хранятся **атомы**. Каждый атом занимает блок памяти **переменного** размера. В этом блоке хранится символьное изображение атома и ряд его дополнительных характеристик.   
  
Область списочных ячеек состоит из блоков **фиксированного** размера. Каждая списочная ячейка хранит **два адреса**, которые по историческим причинам называются **А-указатель** и **D-указатель**. Эти адреса могут указывать как на атомы (т.е. хранить адреса областей из списка объектов), так и на другие списочные ячейки.   
  
Для наглядного изображения списков существуют уже устоявшаяся традиция. Списочная ячейка предствляется прямоугольником, разделеным вертикальной линией на две равные части. В левой части хранится A-указатель, в правой - D-указатель. Атомы изображаются символами (буквами или цифрами). Теперь можно сказать, что точечная пара **(A . B)** - это одна списочная ячейка, в A-указателе которой находится адрес атома **A**, а в D-указателе - адрес атома **B**.   
  
Графически точечную пару изображают так:



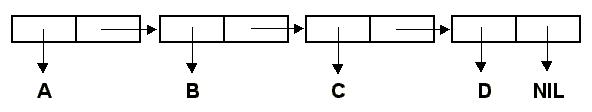
или так:

http://homelisp.ru/help/img/dotpaira.gif

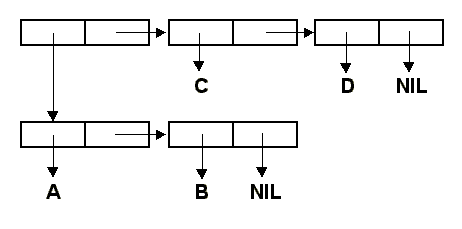
А как можно представить графически список **(A)**? Поскольку список **(A)** есть точечная пара **(A . Nil)**, то графическое изображение строится сразу:



Не представляет труда построить графическое изображение и более сложных списков. Так, например, список **(A B C D)** устроен так:



Полезно соотнести это представление с точечной записью списка **(A B C D)**, которое имеет вид **(A . (B . (C . (D . Nil))))**. Вот еще один пример внутреннего представленния списка более сложной структуры. Список **((A B) (C D))** хранится в памяти в следующем виде:



# Логическая и физическая идентичность списков.

Логически сравнивая списки, мы использовали предикат EQUAL, сравнивающий не физические указатели, а совпадение структурного построения списков и совпадение атомов, формирующих список. Предикат EQ можно использовать лишь для сравнения физического равенства двух символов. Во многих реализациях языка Лисп предикат EQ обобщен таким образом, что с его помощью можно определить физическое равенство двух выражений не зависимо от того, является ли он атомом или списком.

 Вызовы функции **eq** из следующего примера возвращают в качестве значения **false**, так как логически одинаковые аргументы в данном случае представлены в памяти физически различными ячейками:

**>(nil eq '((b c) a b c) '((b c) a b c))**

**false**

**>(nil eq list list2)**

**false**

# Предикаты сравнения.

**EQ, EQL, EQUAL**

Функция **EQ** принимает два аргумента. Она работает следующим образом:   
  
http://homelisp.ru/help/img/n1.gif Если значением первого и второго аргумента является **один и тот же** атом, то функция возвращает в качестве результата **атом T**;   
http://homelisp.ru/help/img/n1.gif Во всех остальных случаях функция возвращает **атом Nil**.   
**Все остальные случаи** включают ситуации, когда значение одного или обоих аргументов не есть атом. Функция **EQ** вернет **Nil** даже в случае, когда значением обоих аргументов является **одно и то же S-выражение**, но не атом!   
Функция **EQ** позволяет корректно сравнивать **только атомы**; для сравнения S-выражений (списков) служит другая функция - **EQUAL**. 

Предикат **[eql](http://filonenko-mikhail.github.io/cltl2-doc/ru/symbols.html" \l "x200-396379r379)** означает то же, что и **[eq](http://filonenko-mikhail.github.io/cltl2-doc/ru/symbols.html" \l "x200-396378r378)**, за исключением того, что если аргументы являются строковыми символами или числами одинакового типа, тогда сравниваются их значения. Таким образом **[eql](http://filonenko-mikhail.github.io/cltl2-doc/ru/symbols.html" \l "x200-396379r379)** говорит, являются ли два объекта «концептуально» одинаковыми, тогда как **[eq](http://filonenko-mikhail.github.io/cltl2-doc/ru/symbols.html" \l "x200-396378r378)** указывает, являются ли два объекта «реализационн)» одинаковыми. Предикат **[eql](http://filonenko-mikhail.github.io/cltl2-doc/ru/symbols.html" \l "x200-396379r379)** истинен, если его аргументы равны **[eq](http://filonenko-mikhail.github.io/cltl2-doc/ru/symbols.html" \l "x200-396378r378)**, или если это числа одинакового типа и с одинаковыми значениями, или если это одинаковые буквы.

Предикат **[equal](http://filonenko-mikhail.github.io/cltl2-doc/ru/symbols.html" \l "x200-396381r381)** истинен, если его аргументы это структурно похожие (изоморфные) объекты. Грубое правило такое, что два объекта равны **[equal](http://filonenko-mikhail.github.io/cltl2-doc/ru/symbols.html" \l "x200-396381r381)** тогда и только тогда, когда одинаково их выводимое представление.

Числа и буквы сравниваются также как и в **[eql](http://filonenko-mikhail.github.io/cltl2-doc/ru/symbols.html" \l "x200-396379r379)**. Символы сравниваются как в **[eq](http://filonenko-mikhail.github.io/cltl2-doc/ru/symbols.html" \l "x200-396378r378)**.(+ списки)

(eq ’a ’c) 🡪 nil

(eq ’a a) 🡪 T

(eq () nil) 🡪T

(eq T (atom ‘one)) 🡪T

(eq ’(abc) ‘(abc) ) 🡪nil

(equal ’(abc) ‘(abc) ) 🡪T

(= 3 3,0 ) 🡪T

(eql 3 3,0 ) 🡪nil

(eql ‘a ‘a ) 🡪T

(equal ’(abc) (cons ‘a ‘(bc))) 🡪T

# Лисп- программы, рекурсивные по аргументу (на примере функции APPEND).

(defun append1 (x y)

(cond ((null x) y)

(t (cons (car x)

(append1 (cdr x) y)))))

# Лисп- программы, рекурсивные по значению (на примере функции APPEND).

(defun append2 (x y)

(cond ((null x) y)

(t (append2 (cdr x)

(cons (car x) y)))))

# Другие формы рекурсии: параллельная рекурсия.

Параллельная рекурсия - тело определения функции function\_1 содержит вызов некоторой функции function\_2, несколько аргументов которой являются рекурсивными вызовами функции function\_1.

Рассмотрим примеры параллельной рекурсии. Для записи функции, которая будет копировать список в глубину, воспользуемся параллельной рекурсией.

> (defun full\_copy\_list (list)

(cond

;копией пустого списка является пустой список

((null list) nil)

;копией элемента-атома является элемент-атом

((atom list) list)

;копией непустого списка является список, полученный из копии головы

;и копии хвоста исходного списка

(t (cons (full\_copy\_list (car list)) (full\_copy\_list (cdr list))))))

FULL\_COPY\_LIST

> (full\_copy\_list '(((1) 2) 3))

(((1) 2) 3)

> (full\_copy\_list ())

NIL

# Другие формы рекурсии: взаимная рекурсия.

Взаимная рекурсия - в теле определения функции function\_1 вызывается некоторая функции function\_2, которая, в свою очередь, содержит вызов функции function\_1.

(defun function\_1 … (function\_2 … ) … ) (defun function\_2 … (function\_1 … ) … )

Пример взаимной рекурсии - реверс списка. Так как рекурсия взаимная, в примере определены две функции: reverse и rearrange. Функция rearrange рекурсивна сама по себе.

> (defun reverse (list)

(cond

((atom list) list)

(t (rearrange list nil)))))

REVERSE

> (defun rearrange (list result)

(cond

((null list) result)

(t (rearrange (cdr list) (cons (reverse (car list)) result)))))

REARRANGE

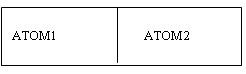
> (reverse '(((1 2 3) 4 5) 6 7))

(7 6 (5 4 (3 2 1)))

# Точечная нотация.

При реализации Лиспа в качестве единой универсальной базовой структуры для конструирования символьных выражений использовалась так называемая ***"точечная нотация"*** ( dot-nоtation ), согласно которой левая и правая части бинарного узла равноправны и могут хранить данные любой природы.

Бинарный узел, содержащий пару атомов ATOM1 и ATOM2,



можно представить как *запись* вида:

( ATOM1 . ATOM2 )

Если вместо атомов " ATOM1 ", " ATOM2 " рекурсивно подставлять произвольные *атомы*, затем построенные из них пары и так далее, то мы получим множество всех возможных составных символьных выражений – S-выражений.

***S-выражение*** - это или *атом* или заключенная в скобки пара из двух S-выражений, разделенных точкой.

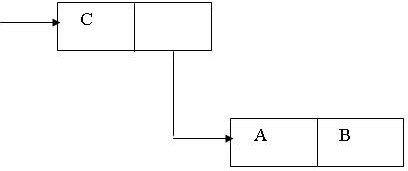
Все сложные данные создаются из одинаково устроенных блоков - бинарных узлов, содержащих пары объектов произвольного вида. Каждый бинарный узел соответствует минимальному блоку памяти.

Списки – это *подмножество* S-выражений, движение вправо *по* которым завершается атомом Nil.

(A . B)

https://www.intuit.ru/EDI/03_10_16_2/1475446789-12103/tutorial/260/objects/3/files/03_07.jpg

(C . (A . B))



# Спискоразрушающие функции: RPLACA, RPLACD.

Для изменения структуры уже имеющегося списка могут использоваться функции **[rplaca](http://filonenko-mikhail.github.io/cltl2-doc/ru/symbols.html" \l "x200-396824r824)** и **[rplacd](http://filonenko-mikhail.github.io/cltl2-doc/ru/symbols.html" \l "x200-396825r825)**. Данные функции изменяют *car* и *cdr* элементы cons-ячеек соответственно.

*[Функция]* **rplaca** *x* *y*

**(rplaca *x* *y*)** изменяет *car* элемент в cons-ячейке *x* на *y* и возвращает (модифицированную) *x*. *x* должен быть cons-ячейкой, но *y* может быть любым Lisp’овым объектом. Например:

(setq g ’(a b c))   
(rplaca (cdr g) ’d) ⇒ (d c)  
Теперь g ⇒ (a d c)

*[Функция]* **rplacd** *x* *y*

**(rplacd *x* *y*)** изменяет *cdr* элемент в cons-ячейке *x* на *y* и возвращает (модифицированную) *x*. *x* должен быть cons-ячейкой, но *y* может быть любым Lisp’овым объектом. Например:

(setq x ’(a b c))   
(rplacd x ’d) ⇒ (a . d)  
Теперь x ⇒ (a . d)

# Спискоразрушающие функции: NCONC, ATTACH.

Функция **NCONC** принимает два аргумента, значениями которых должны быть списки. Функция "склеивает" списки-аргументы в единый список (выполняя то же действие, что и [**APPEND**](http://homelisp.ru/help/lib_funct.html#APPEND). Отличие состоит в том, что **NCONC** имеет побочный эффект - присваивает значение результата первому аргументу.   
  
Вот примеры работы с функцией **NCONC**:

|  |
| --- |
| **(setq z1 '(1 2 3))**  **==> (1 2 3)**  **(setq z2 '(4 5 6))**  **==> (4 5 6)**  **(nconc z1 z2)**  **==> (1 2 3 4 5 6)**  **z1**  **==> (1 2 3 4 5 6)** |

Значение переменной **z1** изменилось (чего не было бы при использовании функции **APPEND**).

Функция **ATTACH** объединяет значения двух своих аргументов в точечную пару (аналогично **CONS**), но результат присваивается в качестве значения **значению** второго аргумента. Функции, которые не только возвращают значения, но и меняют состояние других объектов программы, называются функциями с **побочным эффектом**. Таким образом, **ATTACH** - функция с побочным эффектом.   
  
Вот примеры работы с функцией **ATTACH**:

|  |
| --- |
| **(setq c '(1 2 3))**  **==> (1 2 3)**  **(attach 'a c)**  **==> (a 1 2 3)**  **c**  **==> (a 1 2 3)**  **(setq z '(q w e))**  **==> (q w e)**  **(attach '(1 2) (cdr z))**  **==> ((1 2) w e)**  **z**  **==> (q (1 2) w e)** |

Видно, что первый вызова **ATTACH** приводит к тому, переменная **c** меняет свое значение. В последнем случае вызова **ATTACH** переменная **z** меняет свое значение таким образом, чтобы **(cdr z)** был бы равен результату **((1 2) w e)**.

# Функциональная абстракция, форма записи лямбда- выражений.

В Лиспе существует очень интересная конструкция - **безымянные функции**. Суть безымянной функции состоит в том, что задается алгоритм вычисления, но не задается имени функции. Безымянную функцию можно применить к списку аргументов и сразу получить результат. Синтаксис задания безымянной функции таков:

**(LAMBDA (список параметров) (тело функции))**

**Список параметров** - это одноуровневый атомный список (т.е. список, состоящий **только из атомов**). **Тело функции** - это S-выражение, зависящее от атомов, входящих в список параметров.   
  
Вот пример задания безымянной функции:

**(LAMBDA (x y) (+ (\* x x) (\* y y) ) )**

Легко видеть, что тело функции обеспечивает вычисление величины **x2+y2**. Как пользоваться безымянными функциями? Если передать приведенное выше выражение Лисп-машине, то мы получим обескураживающий результат:

|  |
| --- |
| **(LAMBDA (x y) (+ (\* x x) (\* y y)))**  **Не найдена функция LAMBDA**  **==> ERRSTATE** |

Впрочем, этот результат удивителен только на первый взгляд. Лисп просто пытается вычислить функцию **LAMBDA** с двумя аргументами: **(x y)** и **(+ (\* x x) (\* y y))**. Функции **LAMBDA** в системе нет, отсюда и ошибка. Чтобы вычислить значение безымянной функции, нужно составить S-выражение, представляющее собой **список**, головой которого является **вся LAМBDA-конструкция**, а последующими элементами - **фактические параметры**. Ниже показан правильный вызов безымянной функции:

|  |
| --- |
| **( (LAMBDA (x y) (+ (\* x x) (\* y y) ) ) 3 4 )**  **==> 25** |

Легко видеть, что приведенная выше **LAMBDA**-конструкция полностью эквивалентна традиционной именованной функции:

|  |
| --- |
| **(defun SQ (x y) (+ (\* x x) (\* y y)))**  **==> SQ**  **(SQ 3 4)**  **==> 25** |

Разница в том, что при задании именованной функции соответствующий атом (в примере - **SQ**) становится именем функции, при этом, определяющее выражение **(+ (\* x x) (\* y y))** сохраняется в теле функции. А безымянная функция просто вычисляется и "исчезает".   
  
Впрочем, **HomeLisp** позволяет использовать безымянные функции нетрадиционным образом: если присвоить какому-либо атому в качестве значения корректное **LAMBDA**-выражение, то появляется возможность использовать это **LAMBDA**-выражение без повторного задания:

|  |
| --- |
| **(setq sq2 '(LAMBDA (x y) (+ (\* x x) (\* y y) )))**  **==> (LAMBDA (x y) (+ (\* x x) (\* y y)))**  **sq2**  **==> (LAMBDA (x y) (+ (\* x x) (\* y y)))**  **(sq2 6 7)**  **==> 85** |

Последний вызов **(sq2 6 7)** очень похож на вызов функции **sq2**. Сходство, однако, чисто внешнее.

# Применяющие функционалы.

**Функции FUNCALL и APPLY.**

Среди всех мыслимых функционалов можно выделить один специфический вид - функционал, который **применяет** функциональный аргумент к остальным аргументам. Такие функционалы обычно называют **применяющими функционалами**.   
  
В ядро Лиспа встраиваются два стандартных функционала **FUNCALL** и **APPLY**. Рассмотрим их подробнее.   
  
Функция **FUNCALL** принадлежит классу **SUBR** и принимает произвольное количество аргументов. Эта функция применяет свой **первый** (функциональный) аргумент к оставшемуся списку аргументов. Это выглядит примерно так:

|  |
| --- |
| **(funcall '\* 1 2 3 4 5 6 7 8 9 10)**  **==> 3628800** |

Использование **FUNCALL** позволяет, например, давать именам функций синонимы:

|  |
| --- |
| **(setq сложить '+)**  **==> +**  **(funcall сложить 1 2 3)**  **==> 6** |

Более того, допустимо использовать имя стандартной функции для хранения ссылки на какую-либо другую функцию:

|  |
| --- |
| **(setq cons '\*)**  **==> \***  **(cons 2 3)**  **==> (2 . 3)**  **(funcall cons 2 3)**  **==> 6**  **(funcall 'cons 2 3)**  **==> (2 . 3)** |

Здесь атому **CONS** присвоено значение **\*** (что вполне допустимо!). Следующий вызов показывает, что традиционная функция **CONS** не теряет работоспособности (образуется точечная пара). А вот при вызове посредством **FUNCALL** производится замена **CONS** на **\*** - образуется произведение. В третьем вызове квотирование блокирует замену **CONS** на **\***, и вновь образуется точечная пара.

С функцией **FUNCALL** очень схожа функция **APPLY** (также принадлежащая классу **SUBR**). Отличие заключается в том, что у функции **APPLY** ровно два аргумента: первый - функциональный, а второй является списком произвольной длины. Вызов **APPLY** заключается в том, что вычисляется функция, заданная первым аргументом, со списком параметров, заданным вторым аргументом **APPLY**. Вот пример вызова этой функции:

|  |
| --- |
| **(setq mult '\*)**  **==> \***  **mult**  **==> \***  **(apply mult '(1 2 3 4 5 6 7 8 9 10))**  **==> 3628800** |

# Основные черты аппликативного программирования.

**Аппликативное программирование** — один из видов [декларативного программирования](https://ru.wikipedia.org/wiki/%D0%94%D0%B5%D0%BA%D0%BB%D0%B0%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5), в котором написание программы состоит в систематическом осуществлении применения одного объекта к другому. Результатом такого применения вновь является объект, который может участвовать в применениях как в роли [функции](https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)), так и в роли [аргумента](https://ru.wikipedia.org/wiki/%D0%90%D1%80%D0%B3%D1%83%D0%BC%D0%B5%D0%BD%D1%82_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) и так далее. Это делает запись программы математически ясной. Тот факт, что функция обозначается выражением, свидетельствует о возможности использования значений-функций — [функциональных объектов](https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%BE%D0%BD%D0%B0%D0%BB%D1%8C%D0%BD%D1%8B%D0%B9_%D0%BE%D0%B1%D1%8A%D0%B5%D0%BA%D1%82) — на равных правах с прочими объектами, которые можно передавать как аргументы, либо возвращать как результат вычисления других функций.

Модели аппликативного программирования основываются, как правило, на [комбинаторной логике](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%B1%D0%B8%D0%BD%D0%B0%D1%82%D0%BE%D1%80%D0%BD%D0%B0%D1%8F_%D0%BB%D0%BE%D0%B3%D0%B8%D0%BA%D0%B0) или [λ-исчислении](https://ru.wikipedia.org/wiki/%CE%9B-%D0%B8%D1%81%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5). В комбинаторной логике единственный метаоператор — аппликация, обеспечивающая применение одного объекта к другому, в λ-исчислении, кроме аппликации, есть метаоператор λ-абстракции, с помощью которого возможно построение функций по выражениям, которые, в свою очередь, можно применять к другим объектам. Таким образом, объекты в аппликативном программировании ведут себя как функциональные сущности, что относит его к [функциональной парадигме](https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%BE%D0%BD%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5), однако, в ограниченном смысле, так как имеют место некоторые особенности:

* число аргументных мест ([арность](https://ru.wikipedia.org/wiki/%D0%90%D1%80%D0%BD%D0%BE%D1%81%D1%82%D1%8C)) объекта заранее не фиксируется, но проявляет себя постепенно, во взаимодействиях с другими объектами;
* при конструировании составного объекта один из исходных объектов — функция, применяется к другому — аргументу, при этом в других контекстах они могут поменяться ролями, то есть функции и аргументы рассматриваются как объекты на равных правах;
* разрешается самоприменимость функций, то есть объект может применяться сам к себе.

**Аппликативный язык программирования** — [язык программирования](https://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F), который предназначен для поддержки разработки программ способом получения результата вычисления функции, зависящей от комбинации переменных. Конечный результат достигается последовательным применением функциональных преобразований к данным.

Примерами аппликативных языков программирования служат [функциональные языки](https://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%BE%D0%BD%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F) [Лисп](https://ru.wikipedia.org/wiki/%D0%9B%D0%B8%D1%81%D0%BF) и [ML](https://ru.wikipedia.org/wiki/ML). В языке [Haskell](https://ru.wikipedia.org/wiki/Haskell" \o "Haskell) эта парадигма программирования реализована в виде аппликативного [функтора](https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%BE%D0%BD%D0%B0%D0%BB%D1%8C%D0%BD%D1%8B%D0%B9_%D0%BE%D0%B1%D1%8A%D0%B5%D0%BA%D1%82), расширяющего возможности механизма функциональной абстракции высших порядков до многоместной.