1. Предложения языка Пролог.

Факт представляет собой некое утверждение, которое всегда истинно. Его не нужно выводить или доказывать, например, то, что Иван является родителем Петра, записано в виде факта – родитель(иван, петр).

Правило – более сложная конструкция, которую необходимо доказать, вывести из программы или согласовать с программой. Оно состоит из следующих основных частей: А:-В1, В2, В3, …, Вn,

где А – часть вывода, левая половина правила, голова правила;

:- - последовательность символов, читающаяся как «при условии, что»;

В1, В2, …, Вn – условная часть, правая часть, хвост правила. Правило в отличие от факта Пролог-программа обязана доказать. Пример введения правил и их использования будет рассмотрен ниже.

Вопрос является основным средством общения пользователя с Пролог-программой, т.е. своеобразным интерфейсом с программой. В общем виде вопрос записывается в следующей форме:

?-вопрос.

Здесь «вопрос» представляет собой имя отношения между объектами, например, применительно программы о родственных отношениях, Пролог-системе можно задать следующий вопрос:

?-родитель(мария, петр).

1. Термы Пролога.

**Объекты данных языка Пролог**

В Прологе различают следующие объекты данных:

- атомы;

- числа;

- переменные;

- сложные (составные) объекты

**Атомы**

В своих программах мы уже использовали атомы. В общем случаю атомы можно создавать тремя способами:

* Из цепочки букв, цифр и символа подчеркивания, начиная цепочку со строчной буквы:

родитель, мать, отец, х25, х\_24, иван, маша\_иванова;

* Из специальных символов:

===>, <---->, ….., ::=;

* Из произвольной последовательности символов, заключенные в одинарные кавычки:
* ‘Иван’, ‘Маша\_Иванова’, ‘Северо-Западнай район’.
* Областью действия атомов является вся программа.

**Числа**

* Числа в Прологе бывают целые и вещественные с возможным предшествующим знаком. Представление чисел не отличается от обычного представления числовых величин в императивных языках программирования.

**Переменные**

* Переменные в Прологе – это цепочки, состоящие из букв, цифр и символа подчеркивания. Переменные всегда начинаются с прописной буквы или с символа подчеркивания:
* Х, Результат, Путь, Голова\_1, Остаток\_списка, \_х24, \_33.

Если переменная встречается в предложении один раз, нет необходимости придумывать для нее имя. Можно использовать анонимную переменную, состоящую из одного символа подчеркивания, например, в предложении

имеет\_ребенка(Х):- родитель(X,Y).,

переменная Y участвует один лишь раз, то есть ее можно опустить

имеет\_ребенка(Х):- родитель(X,\_).

Важной особенностью переменных в Прологе является то, что лексический диапазон переменной ограничен тем предложением, в котором она используется. То есть, если в другом предложении предложение встретится переменная с таким же набором символов, то она будет представлять совершенно другой объект.

В Прологе нет глобальных переменных.

**Структурные объекты**

Структурные объекты или просто структуры – это объекты, которые состоят из нескольких компонент. Эти компоненты, в свою очередь, могут быть структурами. Например, сегодняшнюю дату можно рассматривать как структурный объект:

дата(8, сентября, 2021).

Все компоненты структуры являются простыми объектами – два числа и атом.

Другой пример:

дата(День, сентябрь, 2021).

На первом месте используется переменная.

Как было сказано, любой компонент структуры может сам являться структурой, например, дата(День, месяц(май, июнь, июль), 2021) и так далее.

Такой метод является простым и эффективным. Структуры очень удобно представлять в виде деревьев.

Не сложно заметить, что предложения языка Пролог представляют собой ничто иное, как те же самые структурные объекты. Например, факт, утверждающий, что Иван является родителем Петра – по своей сути есть структура:

родитель(иван, петр).

Еще одна отличительная особенность языка Пролог состоит в том, что в нем нет четкой границы между данными и собственно самой программой.

Рассмотренные объекты языка Пролог в дальнейшем мы будем называть **термами**.

1. Процедурная семантика и декларативный смысл.

**Декларативный и процедурный смысл Пролог-программ**

Отметим сразу, что эту тему мы будем поднимать в дальнейшем еще не один раз.

В любой программе на языке Пролог следует различать два смысла:

* декларативный и
* процедурный.

Декларативный смысл касается только отношений, определенных в программе. Например, в программе о родственных отношениях были определены отношения: родитель, мужчина, женщина, отец, мать, сестра, брат и так далее. Это говорит о том, что при некоторых условиях эти отношения достижимы из данной программы.

С определенной степенью уверенности можно утверждать, что в своих программах вы не описывали отношений типа зять, шурин, золовка и так далее. Это значит, что перечисленные отношения не достижимы из программы ни при каких условиях.

Другими словами, декларативный смысл определяет, что должно быть результатом работы программы.

С другой стороны, процедурный смысл определяет еще и как результат был получен, то есть, как отношения реально обрабатываются Пролог-системой.

Способность Пролог-системы прорабатывать многие процедурные особенности самостоятельно считается одним из специфических преимуществ Пролога.

Это свойство позволяет программисту отойти от процедурных деталей и акцентировать свое внимание на декларативных деталях, как более понимаемых человеком. Следует отметить, что результат программы в большей степени определяется ее декларативным смыслом.

1. Списки. Базовые операции над списками.

Списки являются одной из структур данных языка Пролог. Список – это последовательность, составленная из произвольного числа элементов, например, **иван, петр, мария, наталия**. На языке Пролог это запишется следующим образом:

**[иван, петр, мария, наталия].**

Отметим сразу, что список обязательно заключается в квадратные скобки, а элементы списка разделяются символом запятая.

В некотором смысле список является аналогом массива в алгоритмических языках программирования. Однако здесь есть существенные отличия. Во-первых, элементы списка не индексированы и их число не фиксировано, во-вторых, список структура рекурсивная, в-третьих, элементом одного и того же списка могут быть объекты различного «типа», например, [**1, иван, X1**]. В отношении последнего отличия необходимо добавить, что в качестве элемента списка может выступать любой *прологовский терм,* в том числе и список*.*

Можно также провести аналогию списков Пролога и динамических однонаправленных структур в алгоритмических языках. От последних их выгодно отличает отсутствие необходимости работать с адресной составляющей списка.

Для удобства обработки списков в Прологе введены два важных понятия: *голова (head) и хвост(tail).* По умолчанию головой списка считается первый элемент, в некоторых случаях, несколько первых элементов. Оставшиеся элементы относятся к хвосту. Тогда можно сказать, что список – это структура данных, определяемая следующим образом:

*- пустой список*, т.е. список, не содержащий элементов, обозначается как [];

- список, состоящий из головы и хвоста. В качестве головы может выступать любой прологовский терм, а в качестве хвоста обязательно список.

Голова соединяется с хвостом при помощи специального функтора, заимствованного из теоретических разделов рекурсивного программирования, это символ точки. В общем виде любой список можно представить в виде: **.(Голова, Хвост)**. Тогда исходный список запишется следующим образом:

**.(иван, .(петр, .(мария, .(наталия, [])))).**

Подобную структуру удобно представлять в виде дерева, как показано на рис. 1.

٭

иван ٭

петр ٭

мария ٭

наталия []

Рис. 1. Представление списка [иван, петр, мария, наталия]

в виде дерева.

Подобный пример показывает, как общий принцип структуризации объектов данных можно применить к спискам любой длины. Использование точки в практическом программировании в языке Пролог для представления списков не принято. Подобный способ очень расточителен, более того, при большой вложенности списков, он может оказаться причиной путаницы. Более лаконичная скобочная запись пользуется предпочтением.

Для отделения головы от хвоста в языке предусмотрено расширение нотации представления списков, это символ *вертикальной черты*, т. е. List = [Head | Tail ]. Этот символ имеет более общий смысл: можно перечислить любое число символов перед «|», а затем – список остальных элементов.

***Некоторые операции над списками***

Среди множества операций (действий) над списками рассмотрим наиболее важные.

***Проверка принадлежности элемента списку.***

Требуется определить предикат: member(X,List). Здесь List – произвольный список, X – объект того же типа, что и элементы списка List. Составление предиката основывается на следующих соображениях: X есть либо голова списка List, либо X принадлежит хвосту. Это может быть записано с помощью двух предложений, первое из которых есть простой факт (граничное условие или условие выхода из рекурсии), а второе – рекурсивное правило:

member(X,[X|\_]).

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

Знаком подчеркивания здесь обозначена анонимная переменная, значение которой в данном предложении не существенно.

Работу предиката можно объяснить следующим образом: при каждом рекурсивном вызове список будет короче, так.как. аргументом заголовка является структура [\_,Tail], а в рекурсивном вызове – список Tail. Рекурсивные вызовы продолжаются до тех пор, пока искомый элемент X не совпадет с «очередной» головой списка Tail, или список не окажется пустым. В этом случае предикат завершится ложно, поскольку для пустого списка нет своего правила.

Чаще всего этот предикат используют в двух случаях:

1. Переменная X не конкретизирована, а список List – конкретизирован. В этом случае в результате работы предиката переменная X конкретизируется значением элемента списка List. Используя механизм возвратов переменной X можно придать значения всех элементов списка List.

2. Переменная X и список List – конкретизированы. Значением предиката будет истина, если X совпадает с одним из элементов списка List, т. е. принадлежит этому списку. Если совпадений не обнаружено, предикат завершится ложью.

***Сцепление (конкатенация, объединение) списков***

Предикат для объединения двух списков определяется отношением conc(L1,L2,L3), где L1 и L2 – объединяемые списки, а L3 – результат. Графически объединение двух списков представлено на рис. 2. Следует отметить, что голова первого списка становится головой результирующего списка.

|  |  |
| --- | --- |
| H1 | L1 |
| H2 | L2 |
| H1 | L3 | |

Рис. 2. Объединение двух списков

Для определения отношения выделим два случая:

1) если первый список пуст, то второй и третий список представляет один и тот же список, т. е.

conc([],L,L);

2) если первый аргумент не пуст, то он имеет голову и хвост, т. е. [H1|L1]. Его сцепление со вторым списком – список [H1|L], где список T получен после сцепления L1 и L2, т. е.

conc([H1|L1],L2,[H1,L]):-conc(L1,L2,L3).

Пример использования этого предиката в среде Strawberry Prolog:

conc([],L,L).

conc([H1|T1],L2,[H1|T]):- conc(T1,L2,T ).

?-conc([a,s,d],[q,w,e],X), write(X),nl.

Compiling the file:

C:\VIP\VPI\EXAMPLES\and\_\and\_

0 errors, 0 warnings.

[a,s,d,q,w,e]

Yes.

Этот же предикат можно использовать в противоположном «направлении», т. е. не для объединения, а разъединения списка, как это показано в следующем фрагменте:

conc([],L,L).

conc([H1|T1],L2,[H1|T]):- conc(T1,L2,T ).

?-conc([a,s,d],X,[a,s,d,e,r,t]), write(X),nl.

Выполите этот фрагмент и оцените ответ Пролог-системы.

***Добавление элемента в список***

Этот предикат должен иметь три аргумента: добавляемый элемент, исходный список и результирующий список. Самый простой способ добавить элемент в список – это вставить его в самое начало, т. е. сделать головой списка. С учетом сказанного предикат запишется следующим образом:

add(X,L,[X|L]).

***Удаление элемента из списка***

Удаление элемента X из списка L можно определить в виде отношения away(X,L,L1), где L1 – это список L, из которого удалили X. Отношение строится на том соображении, что если X – голова списка, то результат отношения – хвост списка. Если X не голова списка, то его необходимо удалить из хвоста.

away(X,[X|T],T).

away(X,[Y|T],[Y|T1]):-away(X,T,T1).

1. Операция сопоставления.

X =:=Y величины X и Y совпадают (равны)

X =\= Y величины X и Y не равны

Следует отметить разницу между операторами = и =:=, первая вызовет сопоставление объектов и если они сопоставимы, приведет к конкретизации. Никаких вычислений при этом производиться не будет. Вместе с тем, вторая операция (=:=) вызовет арифметические действия, а конкретизации переменных производиться, не будет.

Сопоставление – это операция, допустимая на термами. Сопоставление не единственная операция, но без преувеличения можно сказать, что одна из основных.

Операция сопоставления обозначается символом ‘=‘. Отметим сразу то, что сопоставление не является простым сравнением термов между собой. Оно может производить содержательные вычисления.

Операция сравнения обозначается символами ‘ ‘ ‘=:=‘.

Пуст даны два терма. Будем говорить, что они сопоставимы, если:

* они идентичны или
* переменным в обоих термах можно приписать в качестве значений объекты таким образом, что после подстановки этих объектов в термы, последние стали идентичными.

Подстановку объектов на место переменных в дальнейшем мы будем называть конкретизацией переменных.

Например, термы дата(15,сентябрь,2021) и дата(День, Месяц, 2021сопоставимы. Одна из конкретизаций, которая делает термы идентичными является:

* День заменяется на 15;
* Месяц заменяется на сентябрь;
* 2021 сопоставимо с 2021
* Сопоставление – это процесс, на вход которого подается два терма, он проверяет, соответствуют ли эти термы друг другу. Если термы не сопоставимы, то весь процесс терпит неуспех.

Общие правила сопоставления термов. Пусть есть два произвольных терма S и T, можно утверждать, что они сопоставимы, если:

1. Если S и T – константы, то они сопоставимы, если они являются одним и тем же объектом. Например,

?- иван = иван.

?- 123 = 123 и

?- 123 =:= 123.

Но,

?- иван = 43. , или

?- петр = родитель.

завершатся неуспехом, так как это различные термы.

2. Если S – переменная, а T – произвольный объект, то они сопоставимы и переменной S приписывается значение T. Наоборот, если S - произвольный объект, а T - переменная, то они сопоставимы и переменной T приписывается значение S.

Например,

?- X=иван, write(X). % успех

?- 32=Y. % успех

3. Если S и T – структуры, то они сопоставимы, если

- S и T имеют одинаковый главный функтор;

- покомпонентно сопоставимы.

Например,

?- родитель(X, иван) = родитель(сергей, Y).

родитель = родитель

X = сергей

иван = Y

В целом процесс завершится успехом.

Операцию сопоставления можно использовать в своих программах в любой позиции, она рассматривается как отдельная цель, которая может завершиться успехом или нет.

Кроме того, операцию сопоставления очень активно использует сама Пролог-система, в частности, при попытке ответить на вопрос пользователя.

1. Перебор в Прологе. Механизм прямой трассировки и механизм возврата.

При согласовании целевого утверждения Пролог-система использует метод, известный под названием механизма возврата.

При попытке согласования целевого утверждения Пролог выбирает из базы данных первое предложение, голова которого сопоставима с целевым утверждением.

Если удается согласовать тело утверждения (правую часть правила), то цель достигнута из программы. Если нет, то Пролог переходит к следующему утверждению, и так до тех пор, пока не будет доказано (согласовано) целевое утверждение или не будет достигнут конец базы данных.

В качестве программы рассмотрим следующие утверждения:

less(X,Y):- X<Y, write(X),write(“меньше, чем”),

write(Y).

less(X,Y):- X>Y, write(Y),write(“меньше, чем”),

write(X).

Целевое утверждение

?-less(5,2).

сопоставляется с головой первого утверждения

при X=5, Y=2. Однако не удается согласовать первую цель конъюнкции X<Y в теле утверждения. Это предложение потерпело неуспех.

Пролог переходит к следующему предложению, в нашем случае это второе предложение, голова которого сопоставима с целевым утверждением.

После доказательства цели X>Y Пролог выдаст утвердительный ответ, поскольку все оставшиеся цели доказуемы по умолчанию.

Цель ?-less(5,5) не достижима из текущей базы данных никогда.

Рассмотренный процесс согласования (доказательства) целевого утверждения носит название *механизма прямой трассировки.* Даже если некоторая цель доказана с помощью метода прямой трассировки, пользователь может продолжить поиск других решений*.*

Если слева нет целевых утверждений, о конъюнкцию целей согласовать (доказать) нельзя. Однако, если предшествующие целевые утверждения согласовать удается, Пролог возобновляет процесс доказательств целей слева направо. Описанный процесс смещения влево для повторного согласования целей и возвращения вправо носит название *механизма возвратов*.

Рассмотрим еще раз программу родственных отношений.

родитель(света, петя).

родитель(сергей, петя).

родитель(петя, вася).

муж(петя).

муж(сергей).

отец(X,Y):- родитель(X,Y), муж(X).

Заставим Пролог-систему ответить на вопрос

?- отец(A,B), write(A), write(B).

Пролог найдет в базе данных утверждение отец(X,Y), голова которого согласуется с целевым утверждением:

?- отец(A,B) = отец(X,Y).

Процесс успешен и переменные сопоставятся соответственно A = B и X = Y.

Следующий шаг Пролог-системы – доказательство конъюнкции целей родитель(X,Y), муж(X).

Первая цель родитель(X,Y) согласуется с целью родитель(света, петя), в результате чего переменные конкретизируются X(A) = света, а Y(B) = петя.

Вторая цель муж(света) не достижима из текущей базы данных.

Пролог-система не ответит нет, она предпримет попытку(и) доказать цель родитель(X,Y) при другой конкретизации переменных. То есть произошел процесс смещения влево к предыдущей цели.

Пролог-система найдет очередное предложение родитель(сергей, петя), голова которого согласуется с целевым утверждением.

Результатом этого сопоставления будет очередная конкретизация переменных X(A) = сергей, Y(B) = петя. После чего Пролог-система возвратиться к доказательству цели муж(X). На этот раз процесс завершится успехом, так как муж(X) = муж(сергей) – достижимая цель.

Вкратце в этом и состоит суть механизма возврата.

Механизм возврата – полезный механизм, помогающий определить все возможные достижимые цели из данной программы. Однако, в некоторых случаях он может повлиять на эффективность программы или стать причиной зацикливания программы.

1. Отсечение в Прологе. Виды отсечений.

Для предотвращения бессмысленного перебора необходимо сообщить Пролог-системе, что не нужно осуществлять возврат из этой точки. Для этого в Прологе имеется конструкция, называемая отсечением – ‘!’.

Эта конструкция играет роль псевдоцели, которая всегда успешна.

Программа, использующая отсечение, выглядит следующим образом:

f(X,0):-X<3,!.

f(X,2):- X>3, X<6,!.

f(X,4):- X>6.

Символ ! Предотвращает возврат из тех точек программы, в которых он поставлен.

Теперь при ответе на вопрос ?-f(1,Y), 2<Y система попытается сделать возврат, но вернуться она сможет не далее точки, помеченной символом отсечения. Альтернативные утверждения, соответствующие 2 и 3 правилам, рассматриваться не будут.

1. Операторная запись.

Программист определяет новые операторы, вводя в программу особый вид предложений, которые иногда называют директивами. Такие предложения играют роль определений новых операторов. Определение оператора должно появиться в программе раньше, чем любое выражение, использующее этот оператор. Например, оператор родитель можно определить директивой

:- op( 600, xfx, родитель).

Такая запись сообщит Прологу, что мы хотим использовать " родитель " в качестве оператора с приоритетом 600 и типом 'xfx', обозначающий одну из разновидностей инфиксного оператора. Форма спецификатора 'xfx' указывает на то, что оператор, обозначенный через 'f', располагается между аргументами, обозначенными через 'х'.

Существуют три группы типов операторов, обозначаемые спецификаторами:

1. инфиксные операторы трех типов: xfx xfy yfx
2. префиксные операторы двух типов: fx fy
3. постфиксные операторы двух типов: хf yf

Спецификаторы выбраны с таким расчетом, чтобы нагляднее отразить структуру выражения, в котором 'f' соответствует оператору, а 'x' и 'y' представляют его аргументы.

Между 'x' и 'y' есть разница. Если аргумент заключен в скобки или не имеет структуры (является простым объектом), тогда его приоритет равен 0; если же он структурный, тогда его приоритет равен приоритету его главного функтора. С помощью 'x' обозначается аргумент, чей приоритет должен быть строго выше приоритета оператора (т e. его номер строго меньше номера приоритета оператора); с помощью 'y' обозначается аргумент, чей приоритет выше или равен приоритету оператора.

1. Типовые предикаты Пролога.
2. Арифметические действия в Прологе.

Язык Пролог относится к декларативным языкам, которые предназначены для обработки символьной информации. В редких случаях в языке используются средства для выполнения арифметических действий. Эти средства для различных версий сильно отличаются. Наиболее развитым математическим аппаратом снабжены более поздние версии языка, например, VP, который предназначен для решения широкого круга задач, а не только символьной обработки данных. В SP подобные средства более «скромные», однако, их вполне достаточно для решения большинства математических задач.

Отметим некоторые особенности стандарта языка при использовании арифметических действий. Во-первых, для того, чтобы действительно вызвать какое-либо арифметическое действие, необходимо использовать оператор **is**. Пожалуй, это единственный случай, когда производится какое-либо действие, а не задается отношение между объектами программы. Пролог-система «понимает» этот оператор как вызов встроенной процедуры, в качестве аргументов которой должны выступать числовые величины. Во-вторых, перед использованием переменных в арифметических действиях, они должны быть конкретизированы значениями. В-третьих, лексический диапазон переменной – то предложение, в котором она присутствует. Если в программе присутствуют в разных предложениях переменные с одинаковыми именами, то это разные объекты.

Рассмотрим несколько простых примеров использования арифметических действий.

***Пример 1.*** Сложение двух чисел.

Если Пролог-системе задать следующий вопрос:

?-12+34, то ответом системы будет

Compiling the file:

C:\VIP\VPI\EXAMPLES\and\_\and\_

0 errors, 0 warnings.

No.

Здесь не указано никаких действий, и Пролог-система выдает подобный ответ. Для того, чтобы система выполнила действие, поставим вопрос иначе:

?-X is 12+34, write(X), nl.

В этом случае ответ будет следующим:

Compiling the file:

C:\VIP\VPI\EXAMPLES\and\_\and\_

0 errors, 0 warnings.

46

Yes.

Ответ в этом случае полностью совпадает с истиной. Попробуйте заменить символ сложение на символы других арифметических операций и оцените результаты.

***Пример 2.*** Использование скобочных выражений.

Задачу можно усложнить, если использовать скобки для изменения приоритета арифметических операций, например,

?-X is (12+34)\*20//4, write(X), nl.

Ответом в данном случае будет число 230.

***Пример 3***. Использование переменных в арифметических выражениях.

Как уже отмечалось, перед использованием переменных в выражениях, они должны быть конкретизированы числовыми значениями. Это показано ниже.

?-X=23, Y=11, Z is X\*Y, write(Z), nl.

Рассмотрим последний вопрос к системе более подробно. Отметим ту особенность, что символ равенства (=) здесь не обозначает операцию присваивания, как в алгоритмических языках программирования. Это операция *сопоставления* объектов и в случае, если она завершается успехом, переменные *конкретизируются* соответствующими значениями, в данном случае числовыми. Оттранслируйте этот фрагмент и посмотрите результат.

В языке Strawberry Prolog кроме рассмотренных арифметических действий определены следующие операции над числовыми величинами:

+ сложение

- вычитание

\* умножение

/ вещественное деление

\*\* степень числа (экспонента)

// целочисленное деление

mod деление по модулю

rem остаток от целочисленного деления

abs абсолютное значение числа

sign знак числа

random случайное число, не превышающее значения аргумента.

Следует отметить, что в среде Strawberry Prolog можно оперировать как с целочисленными значениями, так и с вещественными. Причем вещественные числа можно представлять как в обычном формате, так и в экспоненциальном. Следующий фрагмент программы демонстрирует сказанное.

?- X=2300.345, Y=4.4e2, Z is X/Y, write(Z), nl.

Оттранслируйте фрагмент и получите результат.

Для вещественных чисел в Strawberry Prolog определены следующие функции: sin, cos, tg, arcsin, arccos, arctg, sinh, cosh, tgh, sqrt, log, log10, exp. Их названия и функциональные действия совпадают с известными функциями алгоритмических языков программирования. Следует помнить, что тригонометрические функции в качестве аргументов получают значения, измеряемые в радианах. Если есть потребность работы с аргументами, измеряемыми в градусах, необходимо сделать соответствующие преобразования, например:

?- X is sin(3.14\*30/180), write(X), nl.

При работе с числовой информацией очень часто используются операции сравнения. В Прологе и, в частности, в Strawberry Prolog имеется ряд операторов, позволяющих производить операции сравнения чисел:

X > Y X больше Y

X < Y X меньше Y

X >= Y X больше или равен Y

X =< Y X меньше или равен Y

X =:=Y величины X и Y совпадают (равны)

X =\= Y величины X и Y не равны

Следует отметить разницу между операторами = и =:=, первая вызовет сопоставление объектов и если они сопоставимы, приведет к конкретизации. Никаких вычислений при этом производиться не будет. Вместе с тем, вторая операция (=:=) вызовет арифметические действия, а конкретизации переменных производиться, не будет.

1. Работа с базами данных..

Программная среда Пролога состоит из двух частей:

* программного модуля Пролога;
* *базы данных* (БД) Пролога.

Программный модуль включает в себя интерпретатор (компилятор), драйвер ввода-вывода и набор системных процедур. Пользователь обычно не имеет возможности изменять программный модуль.

База данных Пролога содержит набор основных, заранее определенных предикатов. Кроме того, резервируется пространство для добавления пользовательских предикатов, представляющих собой программу на Прологе. Механизм добавления, удаления и обновления утверждений пользователя классифицируется следующим образом:

1. добавление и удаление утверждений;
2. добавление утверждений из файла.

Пролог программа представляется реляционной БД, то есть как описание некоторого множества отношений. Описание отношений в ней присутствует в явном виде (факты) или в неявном виде (правила). Более того, встроенные предикаты дают возможность корректировать эту БД в процессе выполнения программы. Предикаты для работы с БД таковы: assert, asserta, assertz, assert\_in, asserta\_in, retract, retractall. Имена предикатов и их количество зависит от реализации языка.

Предикат assert(<the fact>) – всегда успешен, а в качестве своего побочного эффекта добавляет факт к текущей базе данных. Предикат retract(<the fact>) – приводит к противоположному эффекту: удаляет сопоставимый c <the fact> факт из базы данных. Для уяснения принципов работы этих предикатов рассмотрим фрагмент программного кода.

1. Пролог и искусственный интеллект.
2. Экспертные системы.
3. Двоичные деревья, двоичные справочники.

Следующий рекурсивный тип данных, широко используемый в символьном программировании – двоичные (бинарные) деревья.

Бинарные деревья задаются тренарным функтором tree(Element, Left, Right), где

Element – элемент, находящийся в вершине дерев, Left – левое поддерево и Right – правое поддерево.

Пустое дерево будем обозначать через void.

Рассмотрим пример самого простого дерева.

Его можно описать следующим образом.

tree(a, tree(b, void,void), tree(c, void,void)).

Логические программы, работающие с деревьями, подобны программам, работающими со списками. Как и в случае натуральных чисел и списков, начнем с типового определения двоичного дерева.

binary\_tree(void).

binary\_tree(tree(Element,Left,Right)):-

binary\_tree(Left), binary\_tree(Right).

Следует сразу отметить что отношение с двойной рекурсией, то есть в теле рекурсивного правила имеются две цели с тем же предикатом, что и в заголовке правила.

Этот эффект возникает благодаря двойной рекурсивной природе бинарных деревьев.

Давайте, как в случае со списками, определим некоторые типовые операции над деревьями.

*Принадлежность элемента дереву.*

По аналогии со списками можно определить предикат, проверяющий вхождение элемента в дерево.

tree\_member(El,tree(El,L,R)).

tree\_member(El,tree(X,L,R)):- tree\_member(El,L).

tree\_member(El,tree(X,L,R)):- tree\_member(El,R).

Декларативный смысл отношения очевиден. Элемент принадлежит дереву, если он является вершиной дерева. Это граничное условие.

Если искомый элемент не является вершиной дерева, то он может принадлежать либо левому поддереву, либо правому поддереву.

*Изоморфность двух бинарных деревьев*

Две ветви бинарного дерева различны, но во многих приложениях это различие не существенно*.* Следовательно, возникает полезное изоморфизма, которое определяет, являются ли два дерева одинаковыми.

Два бинарных дерева Т1 и Т2 изоморфны, если Т2 может быть получено из Т2 изменением порядка ветвей в поддеревьях.

В случае непустых деревьев два дерева изоморфны, если элементы в вершинах совпадают, и/или оба левых поддерева и оба правых поддерева изоморфны, или

левое поддерево одного дерева изоморфно правому поддереву другого и два других поддерева тоже изоморфны.

Отношение изоморфности.

isotree(void,void).

iIsotree(tree(X,L1,R1), tree(X,L2,R2)):-

isotree(L1,L2),isotree(R1,R2).

iIsotree(tree(X,L1,R1), tree(X,L2,R2)):-

isotree(L1,R2),isotree(R1,L2).

*Замена в дереве всех вхождений элемента на другое значение*

Это отношение базируется на двух правилах:

- из пустого дерева можно получить только пустое дерево;

если дерево не пусто, то проверяется принадлежность заменяемого значение вершине дерева и если оно является вершиной, производится замена. После чего осуществляется переход на замену значения в левом и в правом поддеревьях.

tree\_replace(\_,\_,void,void).

tree\_replace(Old, New, tree(Old,L,R),tree(New,L1,R1)):- !,

tree\_replace(Old,New,L,L1),

tree\_replace(Old,New,R,R1).

tree\_replace(Old, New, tree(K,L,R),tree(K,L1,R1)):-

tree\_replace(Old,New,L,L1),

tree\_replace(Old,New,R,R1).

*Подсчет числа вершин в бинарном дереве*

Это отношение очень похоже на отношение для вычисления длины списка. Рассуждения те же самые: если дерево пустое, количество вершин равно нулю. Если не пустое, то вычисляется количество вершин левого и правого поддеревьев и плюс 1 – вершина в корне.

tree\_length(void,0).

tree\_length(tree(\_,L,R), N):-

tree\_length(L,N1),

tree\_length(R,N2),

N is N1+N2+1.

Несмотря на кажущуюся сложность бинарных деревьев, операции над ними не сложнее, чем аналогичные операции над списками.

*Количество листьев бинарного дерева*

Подобная задача – подсчет числа листьев. Напомню, что листом дерева считается любая вершина, не имеющая потомков.

Понятно, что в пустом дереве нет вершин, являющихся листьями. Это базис рекурсии. Второй базис будет заключаться в очевидном факте, что в дереве в котором одна вершина , количество листьев равно 1.

Шаг рекурсии – подсчет листьев в левом поддереве и подсчет листьев в правом поддереве.

tree\_leaves(void,0).

tree\_leaves(tree(\_,void,void), 1):- !.

tree\_leaves(tree(\_,L,R), N):-

tree\_leaves(L,N1),

tree\_leaves(R,N2),

N is N1+N2.

*Сумма чисел, расположенных в числовом бинарном дереве*

Это отношение похоже на предыдущие и основывается на том, что сумма числовых величин пустого дерева равна 0. Если дерево не пустое, то сумма равна сумме числовых величин левого и правого поддеревьев.

tree\_summa(void,0).

tree\_summa(tree(X,L,R), N):-

tree\_summa(L,N1),

tree\_summa(R,N2),

N is N1+N2+X.

*Высота бинарного дерева*

Высотой бинарного дерева считается наибольшая длина пути от вершины дерева до его листа.

Базис рекурсии прост – высота пустого дерева равна 0.

Шаг рекурсии состоит в вычислении высоты левого и правого поддеревьев, вычисления максимума из них плюс 1 (высота вершины).

tree\_hight(void, 0).

tree\_high(tree(\_,L,R),D):-

tree\_high(L,D1),

tree\_high(R,D2),

max(D1,D2,Max),

D is Max+1.

Существует особый вид бинарных деревьев – двоичные справочники. В двоичном справочнике все величины, которые меньше значения, находящегося в вершине, принадлежат левому поддереву, а те значения, которые больше – принадлежат правому поддереву.

Усовершенствуем предикат принадлежности элемента двоичному справочнику:

tree\_member(El,tree(El,\_,\_)):- !.

tree\_member(El,tree(K,L,\_)):-

El<K, !,

tree\_member(El,L).

tree\_member(El,tree(K,\_,R)):-

El>K, !,

tree\_member(El,R).

Это отношение можно существенно упростить, предложите свои варианты.

*Добавление в двоичный справочник нового значения*

Это отношение будет содержать два граничных условия и два рекурсивных правила.

Первое граничное условие говорит о том, что если исходное дерево пустое, то добавляемый элемент становится вершиной этого дерева.

Второе граничное условие – если вставляемое значение совпадает со значением в вершине дерева, то результат не будет отличаться от исходного дерева ( в двоичном справочнике все значения различны).

Два правила определяют поведение программы в случае не выполнения граничных условий.

tree\_insert(El,void, tree(El,void,void)).

tree\_insert(El,tree(El,L,R),tree(El,L1,R)):-!,

tree\_insert(El,L,L1).

tree\_insert(El,tree(El,L,R),tree(El,L,R1)):-

tree\_insert(El,L,R1).

*Генерация двоичного дерева*

Записывать двоичные деревья вручную достаточно сложно, определим предикат, который генерирует дерево самостоятельно.

tree\_gen(0, void).

tree\_gen(N,T):- random(100,X),

N1 is N-1,

tree\_gen(N1,T1), tree\_insert(X,T1,T).

1. Решение логических задач, головоломок.

*Предварительные понятия*

Рассмотрим задачу переупорядочивания кубиков, поставленных друг на друга, как показано на следующем слайде. На каждом шагу разрешается переставлять только один кубик. Кубик можно взять только тогда, когда его верхняя поверхность свободна. Кубик можно поставить на стол или на другой кубик.

Для того, чтобы построить требуемый план, мы должны отыскать последовательность ходов, реализующих данную задачу.

Эту задачу можно представить как проблему выбора среди множественных альтернатив.

В исходной ситуации альтернатива всего одна – поставить кубик С на стол. После того как кубик С поставлен на стол, можно выделить еще три альтернативы:

- поставить А на стол или

- поставить А на С или

- поставить С на А.

Ясно, что «поставить С на А» не следует рассматривать в серьез. Тем не менее, это один из допустимых ходов.

Как показывает рассмотренный пример, с задачами такого рода связано два типа понятий:

- проблемные ситуации

- разрешенный ходы или действия, преобразующие одни проблемные ситуации в другие.

Проблемные ситуации вместе с возможными ходами образуют направленный граф, называемый *графом пространства состояний.*

Пространство состояний для рассмотренного примера представлено на следующем слайде.

1. Представление графов в Прологе.

Ориентированный граф может быть задан в виде логической программы с помощью набора фактов вида:

edge(Node1,Node2).

Этот факт входит в программу, если в графе существует ребро, ведущее из вершины Node1 в вершину Node2.

Рассмотрим пример простого графа

Это пример простого ациклического графа.

Определим логическую программу, описывающую данный граф, а также предикат, позволяющий вычислить связность двух вершин в графе.

Последовательность фактов определяет собственно сам граф.

edge(a,b).

edge(c,d).

edge(a,c).

edge(d,e).

edge(b,d).

edge(f,g).

Ребро вида edge(a,b) задает направленность данного отношения.

Две вершины в графе считаются связными, если существует последовательность ребер, ведущая из первой во вторую. Таким образом, connected(Node1,Node2), является истинным, если вершины Node1 иNode2 связны.

connected(Node,Node).

connected(Node1,Node2):-

edge(Node1,Link), connected(Link, Node2).

Несложно заметить аналогию данной программы с программой о родственных отношениях, в частности, с отношениями родитель и предок.

Этот классически пример раскрывает суть программирования на языке Пролог. Сама Пролог-система использует подобный «алгоритм» при нахождении ответов на вопросы к программе. Такой метод нахождения ответов на поставленные вопросы мы будем называть методом поиска в глубину.

Этот метод рассматривает программу как ориентированный ациклический граф, в котором цели связан между собой последовательностью ребер. Если цель достижима из программы, то есть связана с некой целевой вершиной, Пролог-система дает утвердительный ответ.

В случае циклического графа Пролог-система может не найти положительного решения, и в худшем случае, может попасть в замкнутый цикл.

1. Как Пролог-система отвечает на вопросы пользователя.

**Как Пролог-система отвечает на вопросы пользователя?**

Первый, неформальный подход к этому вопросу. Вопрос к системе – это всегда последовательность, состоящая из одной или нескольких целей. Для ответа на вопрос пользователя система пытается достичь всех целей.

Достижение цели означает, что она логически следует из текущей базы данных или все отношения программы истины. Если в предложении имеются переменные, система отыскивает в программе объекты и подставляет их на места переменных, после чего предложения становятся истинными. Полученные конкретизации переменных сообщаются пользователю.

Если для некоторой конкретизации Пролог-система не в состоянии согласовать некоторую цель, то ее ответом пользователю будет “no”.

Таким образом, подходящей интерпретацией Пролог-программы в математических терминах будет следующая : Пролог-система рассматривает программу (факты, правила) как множество аксиом, а вопрос пользователя – как теорему.

Затем она пытается доказать эту теорему, то есть показать, что ее можно логически вывести из аксиом.

Рассмотрим классический пример. Допустим в программе имеются аксиомы:

Все люди смертны.

Сократ – человек.

В терминах языка Пролог:

смертен(Х):-человек(Х).

человек(сократ).

?- смертен(сократ).

yes.

Ответ системы очевиден, он достаточно просто выводится из программы.

Более сложный пример из программы о родственных отношениях:

?-предок(иван, петр).

yes.

Мы знаем, что родитель(иван, петр) – это факт. Известно, что факт во всех случаях – истина. Используя этот факт, и правило предок(X, Y):-родитель(X,Y)., Пролог- система отвечает также утвердительно.

Однако, в отличие от предыдущего примера, это вывод делается в результате «вывода», его нельзя найти непосредственно в программе, но можно вывести, опираясь на имеющиеся факты и правила (аксиомы). Подобный шаг вывода можно записать более коротко:

родитель(иван, петр) ==> предок(иван, петр).

При ответе на вопрос пользователя последовательность шагов может быть более сложной и запутанной. Такую последовательность называют цепочкой доказательств.

Более точный ответ на поставленный вопрос (заголовок) мы попытаемся дать уже сегодня.

1. S-выражения Лиспа. Примеры S-выражений

Для записи программ (функций) в Листе используется специальная форма символьных данных, введенная Маккартни, называемая S-выражениями.

Определим класс символьных выражений, которые называются S-выражениями и образуют класс определения функциональных программ, которые мы научимся конструировать самостоятельно.

Каждая из последующих строк содержит S-выражение:

(иван петр мария вася толя света катя)

((иван 40) (мария 37)(вася 10))

((кафедра вт) самая (лучшая (кафедра (в мире))))

Самым очевидным свойством символьного выражения является использование скобок. Действительно, использование скобок в S-выражениях является основополагающим.

В Лиспе с помощью S-выражений записывается абсолютно все: объявление структур данных, определение и вызовы функций, макросов, специальных форм, вычисление алгебраических выражений, и т. д. Другими словами, в Лиспе принята единообразная форма S-выражений.

1. Понятие строго функционального языка.

Запись выражений в этой форме показывает то, Питер Лэндин (\*) назвал *аппликативной структурой выражений* в математическом языке (и в языке программирования тоже).

Каждое выражение в таком языке может быть разбито на составляющие его части, каждая из которых является либо оператором, либо операндом.

Операнд обозначает значение, а операция – функцию. Структура выражения проста: она состоит из операции, применимой к операндам.

Любое выражение, которое дважды встречается в одном и том же контексте, должно иметь одно и тоже значение.

Язык, в котором это свойство сохраняется для всех его выражений, называется аппликативным. Мы чаще будем называть такой язык строго функциональным.

1. Определение функции, безымянная лямбда-функция.

**Имя и значение символа**

Значением константы является сама константа. Мы уже знаем, что перед такими объектами как t, nil, числа ставить апостроф не нужно. Как констант они обозначают самих себя (имя совпадает со значением).

Другие символы можно использовать в качестве переменных, связав с ними некоторые значения.

*Функция set*

При помощи функции set можно «присвоить» символу некоторое значение. Точнее будет сказать, связать символ с некоторым значением. Эта связь остается до следующего связывания или до окончания работы программы.

(set ‘sym 10)

После этого sym будет иметь значение 10 и им можно воспользоваться в различных выражениях, например,

(\* sym 2.3) 20.3

sym 10

*Функция setq*

Аналогичная функция setq позволяет выполнить связывание символа со значением, не вычисляя первый аргумент.

Об этом символизирует символ q (quote)в названии функции. В остальном эта функция аналогична выше рассмотренной.

>(setq functions ‘(car cdr cons atom equal))

*Обобщенная функция присваивания setf*

В Коммон Лиспе значение символа сохраняется в ячейке памяти, связанной с самим символом. Под ячейкой памяти понимаются поля списочной ячейки (позже).

Для записи значения в ячейку памяти можно применить функцию setf.

Общий формат функции setf:

(setf ячейка\_памяти значение)

>(setq x y) = (setf x y)

>(set x y) = (setf(symbol-value x) y)

Еще один интересный предикат – boundp, который позволяет поверить связность символа с определенным значением, например, (boundp ‘functions).

Последнее выражение примет значение t, поскольку ранее была осуществлена связь с символом functions.

Функции группы set не являются «чистыми» функциями, они имеют побочный эффект, смысл которого состоит в создании связи символа со значением. Если это не является принципиально важным, слово псевдо перед функцией будем опускать.

**Лямбда-выражение – параметризированные вычисления**

Определение функций и их вычисление в Лиспе основано на лямбда-исчислении (нотации) А. Черча, представляющем для этого простой и точный формализм.

В лямбда-исчислении Черча функция имеет следующий вид:

lambda(x1, x2, … , xN) fn, где

xi - формальные параметры определения;

fn – тело функции.

Входящий в состав формы список параметров называют *лямбда-списком*.

Телом функции может быть произвольная *форма*, значение которой может вычислить интерпретатор.

На языке Лисп лямбда-выражение:

(lambda (x1 x2 … xn) fn)

Как видите отличия не принципиальные.

Функцию, вычисляющую сумму квадратов двух чисел можно определить следующим образом:

(lambda (x)(+ (\* x x)(\* y y)))

Формальность параметров означает, что их можно заменить на любые другие символы, и это не отразится на вычислениях.

Следующий пример - функция list для двух аргументов:

(lambda (x y) (cons ( cons y nil))),

Здесь выражение или форма (cons ( cons y nil))) представляет собой тело лямбда-выражения.

Лямбда-вызов

Лямбда-выражения, рассмотренные ранее – это определение вычислений и параметров функции в чистом виде.

Для того, чтобы применить такую функцию к некоторым аргументам, нужно в вызове функции поставить лямбда определение на место имени функции:

(лямбда-выражение а1 а 2 … аN)

Здесь а1 а 2 … аN – формы, задающие фактические параметры, которые вычисляются как обычно.

Например, сложение двух чисел:

>( (lambda (x y)(+ x y)) 4 7)

Или функция list для двух аргументов:

>( (lambda (x y) (cons x (cons y nil))) ‘a ‘b)

Такие формы называют *лямбда-вызовами*.

1. Списки, операции над списками.

Основное назначение Лиспа – обработка списков. В языке имеется набор базовых функций (примитивов) для работы со списками. Различают следующие виды функций – разбора (селекторы), создания (конструкторы) и проверки (предикаты).

*Функции селекторы*. Первый элемент списка является его головой (head), оставшиеся элементы – хвостом (tail) списка. Отделить голову от списка можно с помощью функции car, общий формат которой представляется следующим образом:

**(car list)<enter>**

Результатом работы функции будет S-выражение или сообщение об ошибке, если аргумент не является списком.

Примеры использования функции car:

1. >(car ‘(a s d f)) ; вызов функции

>a ; ответ интерпретатора.

Комментарии к примеру. В первом предложении после вызова функции car, перед списком стоит символ апостроф (‘). Это специальный вид функции, который блокирует вычисления. Если его опустить, то интерпретатор воспримет первый элемент (символ а) списка как имя функции и попытается вызвать ее на исполнение. Эквивалентное действие (блокировку вычислений) выполняет функция quote, например для этого выражения эквивалентная форма записи будет выглядеть следующим образом: >**(car (quote (a s d f))).** Символ > – приглашение интерпретатора. Результат вызова функции, голова списка, представлен во второй строке.

2. >(car ‘cat)

error: bad argument type – CAT.

Здесь действительно аргумент функции является не списком, а атомом.

3. >(car ‘((a s d) e r t))

>(a s d).

Как видно из последнего примера, в качестве головы списка может выступать произвольный объект, в том числе и список.

Следующая функция cdr – отделение хвоста от списка– имеет следующий формат:

**(cdr list)<enter)**.

Результат работы функции – хвостовая часть списка.

Примеры использования функции cdr:

1. >(cdr ‘(a s d f))

>(s d f).

Следует отметить, что хвост списка также является списком.

2. >(cdr ‘(a))

>NIL.

Функция cdr определена только для списков и, из соображения удобства значением функции от пустого списка считается NIL. NIL – константа языка, обозначающая либо ложь, либо пустой список. В противоположность константа Т обозначает истину.

3. >(cdr ‘cat)

error: bad argument type – CAT

Выдача ошибки в данном случае уместна, так как аргумент не является списком.

*Функция конструктора.* Функция cons возвращает в качестве результата новый список, созданный из переданных ей в качестве аргументов элемента и старого списка, причем первый аргумент будет головой в результирующем списке. Общий формат функции следующий:

**(cons head old\_list)<enter>**

Одно замечание относительно второго аргумента – он обязательно должен быть списком. Примеры использования:

1. >(cons ‘a ‘(s d f))

>(a s d f).

Здесь видно, что функция cons конструирует из двух своих аргументов, первый из которых является произвольным объектом, в данном случае – символ, а второй – список, новый список – (a s d f).

2. >(cons (\* 2 3) ‘(+ 4 4)).

1. Внутреннее представление списков.

Оперативная память машины, на которой работает Лисп, логически разбивается на области, которые называются *списочными ячейками*. Списочная ячейка состоит из двух частей, поля CAR и CDR. Каждое поле содержит указатель на объект или другую списочную ячейку. Указатели между ячейками образуют цепочку, по которой можно из предыдущей ячейки попасть в следующую. Графически списочная ячейка представляется прямоугольником (рис. 1), разделенным на две части – поля CAR и CDR.

Поле Поле

CAR CDR

Рис. 1. Графическое представление точечной пары

Изображение одноуровнего списка состоит из последовательности ячеек, связанных друг с другом через указатели в правых частях ячеек (рис. 2).

Правое поле последней ячейки списка в качестве признака конца списка ссылается на пустой список (ячейка перечеркнута). Указатели полей CAR ячеек списка List ссылаются на атомарные структуры, в данном случае на атомы A, B, C.

List

A B C

Рис. 2. Внутренне представление одноуровнего списка (A B C)

При определении базового примитива CONS упоминалось, что ее вторым аргументом обязательно должен быть список. Сейчас это ограничение снимается и в дальнейшем результатом функции (cons ‘a ‘b) будет являться новая структура (a . b), называемая *точечной парой.* Точечная пара – это не список, а более общее символьное выражение. Название происходит от использования в ее записи символа точки, обязательно выделенного с обеих сторон пробелами. Выражение слева от точки (атом, список, другая точечная пара) соответствует полю CAR, а выражение с права от точки – полю CDR.

Точечная нотация позволяет расширить класс объектов, представляемых с помощью списков. Ситуацию сравнивают с использованием дробей или комплексных чисел в арифметике.

1. Точечное представление списков. Примеры списковой и точечной записи.

Любой список можно записать в точечной нотации. Преобразование можно осуществить на всех уровнях следующим образом: **(a1 a2 …an) ↔ (a1 . (a2 . …(an . NIL)…)).** Например, список **(a b c)** в точечной нотации представляется следующим образом **(a . (b . (c . NIL)))**.

Внимание! Не забывайте о пробелах слева и справа от символа точки.

Интерпретатор, получив подобное выражение, транслирует его к списочной структуре.

Точечные пары на практике используются при определении ассоциативных списков, в системном программировании или при умышленной экономии памяти машины, если того требует поставленная задача. Кроме того, использование точечной записи может несколько сократить объем вычислений.

1. Символ, значение символа, свойства символа.

**Символы используются для представления других объектов**

  В программировании на языке Лисп используются символы и построенные из них символьные структуры. В соответствии со словарным определением, символом (греч. "symbolon", знак) является опознавательный знак, условное обозначение, художественный образ, обозначающие какую-нибудь мысль, идею и т.п. В Лиспе понятие символа используется в более узком и точном смысле: под ним подразумевается запись или обозначение.  
  Символ - это имя, состоящее из букв, цифр и специальных знаков, которое обозначает какой-нибудь предмет, объект, вещь, действие из реального мира. В Лиспе символы обозначают числа, другие символы или более сложные структуры, программы (функции) и другие лисповские объекты. Например, символ "+" может обозначать определение действия сложения, "углерод-14" - изотоп углерода и т.п.  
  Примеры символов:

**x** **Символ** **defmacro** **STeP-1984**

**Символы в данном диалекте языка**

  Символы языка могут состоять из букв, цифр и некоторых других знаков, а именно

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

  В принципе, вопросительный и восклицательный знаки, так же как тильда (**~**), и квадратные скобки тоже могут входить в состав символов, но эти знаки лучше приберечь для специального использования.  
  Символы могут состоять как из прописных, так и из строчных букв, хотя в большинстве Лисп-систем, как и в описываемом диалекте, прописные и строчные буквы отождествляются и представляются прописными буквами. Поэтому, например:

**слово ≡ Слово ≡ СЛОВО**

1. Рекурсия в Прологе. Рекурсия по значению и по аргументу.

Процедура или функция может содержать вызов других процедур или функций. В том числе процедура может вызвать саму себя. В этом нет никакого парадокса, компьютер последовательно выполняет встретившиеся команды, и если встречается вызов процедуры, он выполняет эту процедуру. Без разницы, какая процедура дала команду это делать.

Рассмотрим пример рекурсивной процедуры:

procedure Rec(a: integer);

begin

if a>0 then Rec(a-1);

write(a);

end;

Процедура Rec вызывается с параметром a=3. В ней содержится вызов с параметром a=2. При этом предыдущий вызов еще не закончился, кроме того, в теле функции содержится еще один вызов и так до тех пор, пока параметр не будет равен нулю.

Не законченные вызовы процедуры заносятся в стек, и в момент достижения граничного условия, стек начинает «раскручиваться» в противоположном направлении.

Процесс продолжается до тех пор, пока стек обратных вызовов не окажется пустым.

Подобная процедура обработки рекурсивных вызовов используется в большинстве известных языков программирования, в Прологе в том числе.

Следует отметить, что Пролог рассчитан для работы именно с рекурсивными структурами данных, кроме того, внутренняя структура самого языка является рекурсивной.

Для понимания основных принципов рекурсии рассмотрим простые рекурсивные данные – натуральные числа.

Натуральные числа лежат в основе арифметики. Они строятся с помощью двух объектов – константного символ ‘0’ и унарной функции следования S. Таким образом, все натуральные числа рекурсивно представляются в виде 0, S(0), S(S(0)), … S(S..S(0)..). Используем сокращение Sn(0) для обозначения любого натурального числа n.

Определим предикат, описывающий натуральные числа на Прологе:

natural\_number(0).

natural\_number(X):- Y is X-1, natural\_number(Y).

?- natural\_number(13).

Ответом на заданный вопрос будет yes.

Функция следования S в этом отношении выполнена на основе операции вычитания.

Мы будем говорит о *рекурсии по значению*, когда рекурсивный вызов является выражением, определяющим результат функции.

Если же в качестве результата функции возвращается значение некоторой другой функции, а рекурсивный вызов участвует в вычислении аргументов для этой функции, будем говорить о *рекурсии по аргументам*.

1. Виды рекурсий в Лиспе.

**Простая рекурсия**

Функция является рекурсивной, если в ее определении содержится вызов смой этой функции.

Первый пример связан с простой рекурсией, при котором рекурсивный вызов встречается в некоторой ветви один лишь раз.

В качестве примера определим функцию copy, которая строит копию списка.

>(defun copy(lst)

(cond ((null lst) nil)

(t (cons (car lst) (copy (cdr lst))) )))

>(copy ‘(q w e r)) -> (q w e r)

Попросту говоря, эта функция строит копию списка, переданного ей в качестве параметра. Она является рекурсивной по аргументу (!).

Условное предложение в теле функции имеет две ветви: ветвь с условием окончания и ветвь с рекурсией, с помощью которой функция проходит по списку, укорачивая его по направлению cdr.

*Параллельная* рекурсия

Рекурсия называют параллельной, если она встречается одновременно в нескольких аргументах функции.

Общий формат параллельной рекурсии:

(defun f …

… (g … (f …) … (f …) …) … )

Аргументами некоторой функции являются рекурсивные вызовы функции f .

В качестве примера рассмотрим копирование списочной структуры на всех уровнях. Ранее мы рассмотрели копирование списка в направлении cdr. На возможные подсписки не обращалось внимания.

А теперь функция копирования:

>(defun copy-tree(lst)

(cond ((null lst) nil)

((atom lst) lst)

(t (cons (copy-tree (car lst))

(copy-tree (cdr lst)) ))))

Отличия функции copy-tree от функции copy-list есть, во-первых, рекурсия как по голове, так и по хвосту.

Во-вторых, здесь предполагается использование предиката atom, проверяющего атомарное дерево (лист дерева).

В- третьих, рекурсивный вызов представляет собой два аргумента функции cons.

Заметим, что функция рекурсивна по аргументу.

*Взаимная рекурсия*

Под взаимной рекурсией понимается, что в определении одной рекурсивной функции используется вызов некой другой функции, в определении которой используется вызов первой функции. Количество функций может быть больше двух.

Общий формат взаимной рекурсии следующий:

(defun f…

…(g …) …)

(defun g...

…(f …) …)

Классический пример взаимной рекурсии – реверсирование списка в котором основная функция пользуется услугами вспомогательной функции, а последняя, в свою очередь, вызывает основную.

>(defun rev(lst)

(cond ((atom lst) lst)

(t (perem lst nil))))

Вспомогательная функция перестановки:

>(defun perem(lst rez)

(cond ((null lst) rez)

(t (perem (cdr lst)

(cons (rev (car lst)) rez))) ))

Функция perem в процессе построения списка вызывает основную функцию с целью обращения подсписков исходного списка.

Кроме того вспомогательная функция рекурсивна сама по себе.

*Рекурсия более высокого порядка*

В теоретических разделах рекурсивного программирования выделяют понятие порядка (уровня) рекурсии. До чих пор мы рассматривали рекурсию нулевого порядка.

Далее рассмотрим рекурсию более высокого порядка, при которой аргументом рекурсивного вызова является рекурсивный вызов.

Классический пример функции более высокого порядка – функция Аккермана, пользующаяся славой «плохой» функции.

Она является функцией с рекурсией первого порядка. Несмотря на это, ресурсы машины «поедает» очень быстро.

>(defun ackerman(m n)

(cond ((= m 0) (+ n 1))

((= n 0) (ackerman (- m 1) 1))

(t (ackerman (- m 1)

(ackerman m (- n 1))) )))

Вычисление данной функции достаточно сложно, время вычислений растет лавинообразно даже при небольших значениях аргументов.

>ackerman(3 2)

27

При больших значениях аргументов машина может зациклится.

1. Функции более высокого порядка. Применяющие и отображающие функционалы.

Основываясь на едином представлении данных и программ (в форме s-выражения), функции можно передавать в качестве аргумента другую функцию, другими словами, определение функции или символ, представляющий собой определение функции.

Речь идет не о комбинации вызовов функций, например,

>(cons (car ‘(a b c)) ‘(q w e r))

в которой в качестве аргумента функции cons выступает результат вызова (car ‘(a b c)), а именно определение функции.

Аргумент, значением которого является функция, называется *функциональным аргументом.*

А функция, имеющая функциональный аргумент – *функционалом*.

Различия между понятиями «данные» и «функция» определяются не на основе их структуры, а в зависимости их использования.

Если аргумент используется в функции лишь как объект, то мы имеем дело с обыкновенным аргументом.

Если же он используется как средство, определяющее вычисления, то есть играет в вычислениях роль, например, лямбда выражения, которое применяется к другим аргументам, то мы имеем дело с функцией.

Одно и то же выражение в разных контекстах может выступать как обычный аргумент, а в других – как функциональный

Функциональный аргумент и функционал являются обобщением понятия функции.

Функция является функционалом, если в качестве ее аргумента используется лисповский объект типа (1), который интерпретируется как функция (2) в теле функционала.

Функциональным объектом может быть:

1. символьное имя, представляющее определение функции (системна функция или функция, определенная пользователем);

2. безымянное лямбда выражение:

3. замыкание.

**Функциональное** **значение**

Кроме функционального аргумента функционал может иметь функциональное значение, то есть в качестве результата возвращается определение функции.

Вызов функции с функциональным значением может находиться в двух различных позициях:

1. функционального аргумента в вызове функционала;

2. на месте имени функции в вызове функции (или функционала).

**Применяющие функционалы**

Одним из основных типов функционалов являются функции, позволяющие применять функциональный аргумент к его параметрам. Такие функционалы называют *аппликативными или применяющими*.

Применяющие функционалы родственны универсальной функции Лиспа eval. В то время как eval вычисляет значение произвольного

выражения (формы), применяющие функционалы вычисляет значение некоторой функции. Интерпретатор Лиспа eval вызывает в своем определении применяющий функционал apply, а apply в свою очередь вызывает eval.

*Применяющий функционал apply*

Apply является функцией двух аргументов, первый из которых представляет функцию, применяемую ко второму параметру, который передается списком. Общий формат следующий:

(apply fn lst), где lst = (x1 x2 … xN)

Приведем примеры:

> (apply '+ '(33 48))

81

> (apply 'cons '(a (b)))

(A B)

> (setq f '\*)

\*

> (apply f '(124 45))

5580

> (apply 'eval '((/ 3/5 7/9)))

27/35

Более интересные моменты посмотреть на примере лямбда выражений.

> (apply (lambda (x y)(\* x y)) '(2 3))

6 или

> (apply (lambda (x y)(cons x y)) '(a (q w e)))

(A Q W E)

> (apply (lambda (x y)(append x y)) '( (a b c) (1 2 3)))

(A B C 1 2 3)

Не запутайтесь в скобках, здесь это очень просто.

Еще раз о параметрах функции apply, их два – это функциональный аргумент и список, по отношению к которому применяется функциональный аргумент.

Более сложный пример представляет собой лямбда выражение в теле которого имеется применяющий функционал:

> ((lambda (x)(apply '\* x)) '(2 3))

6

Обратите внимание на количество аргументов, их позицию и передачу фактических параметров.

> ((lambda (x)(apply 'cons x)) '(a (s d f)))

(A S D F)

Или

> ((lambda (f x)(apply f x)) 'cons '(a (s d f)))

(A S D F)

Здесь передается два параметра лямбда выражения – функция cons и список

'(a (s d f))) по отношению к которому применяется функция.

Не запутайтесь в скобках и в блокировке вычислений.

Еще несколько примеров, в которых в качестве функционального аргумента выступает другой функционал.

> (apply '\* (apply '+ '(3 5 6)) '(9 6 4))

3024

Разберем подробнее …

И еще

> (apply (apply 'car '((cdr c d))) '((a s d)))

(S D)

Применяющий функционал funcall

Функционал funcall по своему действию аналогичен apply, но аргументы для вызываемой функции он принимает не списком, а по отдельности .

Общий формат следующий:

(funcall fn x1 x2…xN)

Например,

>(funcall ‘- 761 342)

329

Использование функционалов позволяет достичь в программе определенной гибкости. Известно, что с произвольным символом можно связать любое значение, в том числе и имя функции.

Предположим,

>(setq add ‘+)

Теперь символ add может выступать в роли функционального аргумента

> (funcall add 3 5)

8

Интерпретация имени зависит от синтаксической позиции.

Следующий пример

> (funcall (car '(\* + - /)) 4/7 5/9)

20/63

Таким образом появляется возможность использовать синонимы имен функции. С другой стороны, имя функции можно использовать как обыкновенную переменную.

> (setq cons '\*)

\*

> (funcall cons 8 29)

232

Важное замечание: в качестве функционального аргумента может выступать только «чистая» функция, то есть функция, не имеющая побочного эффекта.

Например,

> (funcall 'setq 'a 10)

error: bad function - #<FSubr-SETQ: #173fe523>

Это говорит о том, что функция setq не является чистой функцией.

В качестве функционального аргумента не может выступать макрос.

Некоторые функции, имеющие побочный эффект, все-таки можно передавать в качестве функций для функционалов, например структуроразрушающие rplaca и rplacd.

> (setq lst '(a b c d))

(A B C D)

> (funcall 'rplaca lst 'p)

(P B C D)

Как уже говорилось в качестве функционального аргумента может выступать не только предопределенная функция, но и функция пользователя. Вспомним пример с логической копией списка:

> (defun copy-list(lst)

(cond ((null lst) nil)

(t (cons (car lst)(copy-list (cdr lst))) )))

Вызов функции через функционал -

> (funcall 'copy-list '(1 2 3))

(1 2 3)

С функционалом funcall проблем меньше, чем с предыдущим функционалом. Рассмотрим несколько примеров использования лямбда выражений в данном функционале.

> (funcall (lambda(x y)(cons x y)) 'a '(q w e r))

(A Q W E R)

Меньше скобок, меньше вероятность сделать ошибку.

> (funcall (lambda(x y)(append x y)) '(a s d) '(q w e))

(A S D Q W E)

> (funcall (lambda(x y)(rplacd x y)) lst '(1 2 3))

(P 1 2 3)

Функционал funcall легче использовать в лямбда выражениях.

> ((lambda(x y)(funcall 'append x y)) '( a s d) '(c v b))

(A S D C V B)

> ((lambda(fn lst)(funcall fn lst)) 'reverse '(a s d f g))

(G F D S A) )

В последнем выражении функция, которую необходимо применить к аргументу, сама является аргументом лямбда выражения.

> (funcall (defun summa(x y)(+ x y)) 4 8)

12

Что примечательного в этом примере? Здесь на месте функционального аргумента стоит определение функции, которое применяется к двум числовым величинам. Заметим, что это не вызов, а именно определение функции, что важно для понимания разницы между функционалами и обычными функциями.

Примечательно и то, что данное определение будет работать дальше и вызов

> (summa 6 7)

13

воспримется интерпретатором как вызов полноценной функции.

*Отображающие функционалы*

Другой важный класс функционалов в практическом программировании на языке Лисп образуют отображающие функционалы или map-функции. Map-функционалы являются функциями, которые некоторым образом отображают список (последовательность) в новый список или порождают побочный эффект, связанный с этой последовательностью.

Имена map-функций начинаются на map и х вызов имеет вид:

(mapx fn l1 l2 … lN),

Где l1 l2 … lN – списки, а fn – функция от N аргументов. Как правило, map-функция применяется к одному списочному аргументу, то есть,

(mapx fn список)

Существует два основных типа mаp-функций. Один из них применяет функциональный аргумент fn к car последовательностям список. Другие применяют функциональный аргумент к cdr последовательностям списка. Кроме того они отличаются друг от друга способами формирования результата.

Рассмотрим основные типы map-функций.

*Отображающий функционал mapcar*

Этот функционал последовательно применяет функциональный аргумент к car элементам списка, то есть последовательно по отношению к каждому элементу списка.

Число параметров функционалов – число переменное, но большую популярность имеет работа с одним аргументом.

Общий формат mapcar следующий:

(mapcar fn ‘(x1 x2 x3 … xN))

что равносильно записи

(list (fn x1)(fn x2)(fn x3)…(fn xn))

В качестве значения функционала формируется список из результатов применения функционального аргумента.

Например,

> (setq lst '(a b c d e))

> (mapcar 'atom lst)

(T T T T T)

> (setq lst '(a b (c d) e))

> (mapcar 'atom lst)

(T T NIL T)

*Функционал maplist*

Функция maplist действует подобно mapcar, но действия осуществляются над cdr последовательностями списка, с учетом того, что изначально хвостом любого списка является собственно сам список. Например,

> (maplist 'car '(a s d f))

(A S D F)

> (maplist 'cdr '(a s d f))

((S D F) (D F) (F) NIL)

> (maplist 'reverse '(a s d f g))

((G F D S A) (G F D S) (G F D) (G F) (G))

Эту функцию, как и другие функционалы можно вызывать по другому

> (maplist (function reverse) '(1 2 3 4 5))

((5 4 3 2 1) (5 4 3 2) (5 4 3) (5 4) (5))

Функция function обозначает функциональную блокировку.

Как и в применяющих функционалах в отображающих также можно использовать безымянное лямбда выражение.

> (maplist (lambda(lst) 'reverse lst) '(w e r t))

((W E R T) (E R T) (R T) (T))

>(maplist (lambda(x) (car x)) '(q w e r))

Что здесь получится посмотрите и проанализируйте самостоятельно.

*Функционалы mapcan и mapcon*

Функции mapcan и mapcon являются аналогами mapcar и maplist. Отличие состоит в том, mapcan и mapcon не строят новый список, используя функцию list, а объединяют списки с помощью структуроразрушающей функции nconc

Пример функции nconc

> (setq lst '(a s d f))

(A S D F)

> lst

(A S D F)

> (nconc lst '(q w e r))

(A S D F Q W E R)

> lst

(A S D F Q W E R)

Исходный список lst был (a s d f), стал (A S D F Q W E R).

Функциями mapcar и mapcon нужно пользоваться очень осторожно. Чтобы эти функции могли применить nconc, их функциональные аргументы должны возвращать в качестве результатов списки.

Рассмотрим несколько примеров.

> (mapcan 'list '(a b c d))

(A B C D)

> (mapcan #'(lambda (x y) (if (null x) nil (list x y)))

'(nil nil nil d e)

'(1 2 3 4 5 6))

(D 4 E 5)

> (mapcan #'(lambda (x) (and (numberp x) (list x)))

'(a 1 b c 3 4 d 5))

(1 3 4 5)

Функция mapcon работает с хвостовыми частями списков

> (mapcon #'list '(1 2 3 4))

((1 2 3 4) (2 3 4) (3 4) (4))

Для следующего примера обратимся вначале к функции maplist

> (maplist 'append '( 1 2 3) '(q w e))

((1 2 3 Q W E) (2 3 W E) (3 E))

1. Замыкания.

**Замыкания**

*Функциональная блокировка*

С точки зрения эффективности работы интерпретатора, чтобы уже на этапе вызова функции функционала он мог отличить функциональный аргумент от обычного. Функциональный аргумент можно пометить с помощью специальной, предотвращающей вычисления формы – function.

Общий формат следующий:

(function функция)

Например:

> (function (lambda(x)(list x y)))

#<Closure: #15afad2b>

> (function car)

#<Subr-CAR: #15afe361>

Слово closure (closing, closure, close, shutdown, shutoff, dissolution) переводится как закрытие, в дальнейшем мы будем именовать как замыкание.

В отличие от обычной блокировки с помощью формы quote (‘) function называется функциональной блокировкой.

Как и обычную блокировку, функциональную блокировку можно записывать в сокращенном виде function = #’. Например, следующие записи эквивалентны:

> (funcall (function +) 2 3)

5

> (funcall #'+ 2 3)

5

Если нужно передать параметры как они есть, используется обычная блокировка. Quote достаточно и для передачи имени функции или лямбда выражения, если в них не участвуют свободные переменные. Работа со свободными переменными более сложная, поскольку их значения зависят от контекста вычислений.

В системных функциях свободные переменные не используются.

*Замыкание – это функция и контекст ее определения.*

Сформированный на время вычисления функции вычислительный контекст после ее вычисления исчезает и на него невозможно сослаться позже или вернуться к нему.

Часто бывает необходимо, чтобы функция для продолжения работы могла бы запомнить связи и состояние более раннего контекста. Это осуществляется с помощью специальных функциональных объектов, называемых *замыканием* (лексическим замыканием, фунаргом).

Замыкание, как имя функции или лямбда выражение, можно использовать в качестве функционального объекта.

*Связи свободных переменных*

В Коммон Лиспе замыкание создается с помощью формы function.

Например,

> (function (lambda(x)(+ x y)))

#<Closure: #15afaa7f>

В приведенном примере свободной (глобальной) переменой является переменная y.

Значение формальных (статических, лексических) параметров функции запоминать не имеет смысла, поскольку они получат новые значения в момент применения замыкания.

Если в замыкаемой функции нет свободных переменных, то форма function ни чем не отличается от формы quote.

Замыкание является обычным объектом языка Лисп и его можно сохранить как обычный объект.

Рассмотрим пример:

> (defun append-list(x)

(function (lambda(y)(append x y))))

APPEND-LIST

> (setq app-abc (append-list '(a b c)))

#<Closure: #15afa87e>

> (funcall app-abc '(1 2 3))

(A B C 1 2 3)

> (funcall app-abc '(3 4 5))

(A B C 3 4 5)