1. **Предложения языка пролог.**

Любое предложение заканчивается символом точки.

Виды предложений:

- факт;

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 – родит(иван, петр).

- правило;

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: бабушка(X,Y):-родит(X,Z),родит(Z,Y),жен(X).

- вопрос.

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

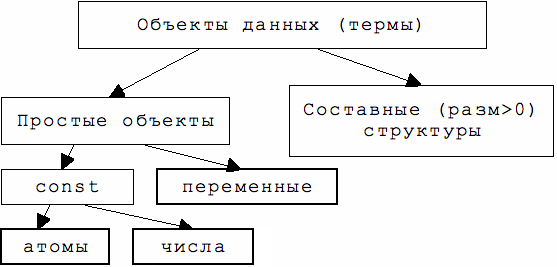
?-вопрос.

?-родит(иван, петр).

Получив подобный вопрос, Пролог-программа ответит yes (да), поскольку в ее базе данных имеется факт, описывающий отношение между указанными объектами.

1. **Термы пролога.**

Объекты данных – термы.



Атом – это любая последовательность букв, цифр, символов подчеркивания, начинающаяся со строчного символа.

Числа в Прологе бывают целыми и вещественными. Синтаксис целых чисел прост, как это видно из следующих примеров: 1, 1313, 0, -97. Вещественные числа можно представлять как в обычном формате, так и в экспоненциальном.

Переменная – последовательность букв, цифр, символов подчеркивания. Отличительной особенностью переменных в Прологе является то, что они всегда начинаются с прописного символа или с символа подчеркивания. Другая особенность переменных – их лексический диапазон ограничен рамками того предложения, в котором они присутствуют. Другими словами, если в одной и той же программе есть переменные с одним и тем же именем, то это суть разные объекты.

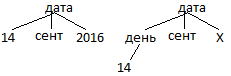
Если переменная встречается в предложения только один раз, то нет необходимости изобретать ей имя. Можно использовать так называемую "анонимную" переменную, которая записывается в виде одного символа подчеркивания.

Структурные объекты (или просто структуры) — это объекты, которые состоят из нескольких компонент. Эти компоненты, в свою очередь, могут быть структурами. Например, дату можно рассматривать как структуру, состоящую из трех компонент: день, месяц, год. Хотя они и составлены из нескольких компонент, структуры в программе ведут себя как единые объекты. Для того, чтобы объединить компоненты в структуру, требуется выбрать функтор.

дата (14, сент, 2016). % дата-компонента(функтор), 14-число, сент – атом, 2016 –число

дата (день(14), сент, X).

Все структурные объекты можно изображать в виде деревьев



Если одно и то же имя появляется в программе в двух различных смыслах, то пролог-система будет различать их по числу аргументов и интерпретировать это имя как два функтора. Это возможно потому, что каждый функтор определяется двумя параметрами:

(1) именем, синтаксис которого совпадает с синтаксисом атомов;

(2) арностью — т.е. числом аргументов.

1. **Процедурная семантика и декларативный смысл.**

Процедурная семантика связана с внутренней организацией Пролог-системы и определяет последовательность функционирования ЭВМ, ориентированной на достижение цели. Предложение G:-P,Q. с процедурной точки зрения оно интерпретируется так:

"чтобы решить задачу G, необходимо сначала выполнить процедуру P, а затем процедуру Q".

Запись предложения G:-Q,P определяет совершенно другую программу, состоящую в выполнении сначала процедуры Q, а затем - процедуры P. В некоторых случаях такая перестановка может привести к неверному результату. Таким образом, с процедурной точки зрения результат выполнения программы может зависеть от порядка предложений и предикатов в них.

1. **Списки, Базовые операции над списками.**

Список – это последовательность, составленная из произвольного числа элементов, например, иван, петр, мария, наталия. На языке Пролог это запишется следующим образом:

[иван, петр, мария, наталия].

1. элементы списка не индексированы и их число не фиксировано
2. список структура рекурсивная
3. элементом одного и того же списка могут быть объекты различного «типа», например, [1, иван, X1].
4. в качестве элемента списка может выступать любой прологовский терм, в том числе и список.

Пустой список [].

Для удобства обработки списков в Прологе введены два важных понятия: голова (head) и хвост(tail).

Голова – первый элемент списка, любой терм. Хвост – список, в том числе пустой.

Голова соединяется с хвостом при помощи специального функтора, 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.

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:

1. принадлежность элемента списку

member(El, [El|Tail]).

member(El, [Hail|Tail]).

?-member(a, [a, b, c]). %Yes

1. Сцепление (конкатенация, объединение) списков

append([],Lst, Lst).

append([H1|T1], Lst2, [H1|T3]):-append(T1, Lst2, T3).

?-append([a,b], [1,2], Lst), write(Lst). %[a,b,1,2]Yes.

1. Добавление элемента в список

add(El, Lst, [El|Lst]).

?-add(a, [b,c,d],V), write(V). % [a,b,c,d]Yes.

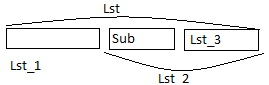
1. Удаление элемента из списка

delete(El, [El|Tail], Tail).

delete(El,[H|Tail],[H|Tail\_1]):-delete(El, Tail, Tail\_1).

?-delete(a,[1,2,3,a], Lst),write(Lst).

1. Подсписок



sublist(Sub, Lst):-append(Sub,Lst\_3,Lst\_2),append(Lst\_1,Lst\_2,Lst).

?-sublist([c,d],[a,b,c,d]).

1. **Операции сопоставления.**

Символ равенства (=) - операция сопоставления объектов и в случае, если она завершается успехом, переменные конкретизируются соответствующими значениями.

1. 2 терма сопоставимы в том случае, если они идентичны

?-а=а. %Yes

?-X=1. %Yes

?-X=Y. %Yes – наиболее общая конкретизация

?-10=X. %Yes

1. 2 терма сопоставимы в том случае, если их главные термы и их компоненты сопоставимы

?-X=дата(14, сент, 2016). %Yes

?-дата(14, Месяц, 2016) = дата(D, сент, Y),

% 1.?-дата=дата. Да

% 2.?-14=D. Да

%3.?Месяц=сент. Да

%4.?2016=Y. Да

write(Месяц), write(D), write(Y). %сент142016

Общие правила выяснения, сопоставимы ли два терма S и T:

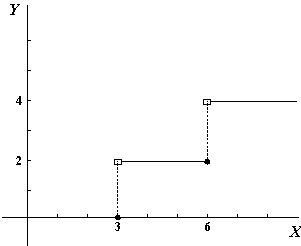
1) Если S и Т – простые объекты, то сопоставимы, если они идентичны.

2) S –переменная, Т – производный терм, то сопоставимы, если Т конкретизирует S.

3) S и Т – структурные объекты, то сопоставимы, если сопоставимы главные функторы и они покомпонентно сопоставимы.

1. **Перебор в Прологе Механизм прямой Трассировки и механизм возврата.**

Ничем не ограниченный перебор может стать источником неэффективности программы, поэтому иногда требуется его ограничить или исключить вовсе. Для этого в Прологе предусмотрена конструкция "отсечение".



Связь между X и Y можно определить с помощью следующих трех правил:

Правило 1: если X < 3, то Y = 0

Правило 2: если 3<= X и X < 6, то Y = 2

Правило 3: если 6 <= X, то Y = 4

На Прологе это можно выразите с помощью бинарного отношения f( X, Y) так:

f( X, 0) :- X < 3. % Правило 1

f( X, 2) :- 3 =< X, X < 6. % Правило 2

f( X, 4) :- 6 =< X. % Правило 3

В этой программе предполагается, конечно, что к моменту начала вычисления f( X, Y) X уже конкретизирован каким-либо числом; это необходимо для выполнения операторов сравнения.

Пролог производит доказательство конъюнкции целевых утверждений слева направо. При этом может встретиться целевое утверждение, согласовать которое не удается. Если такое случается, то происходит смещение влево до тех пор, пока не будет найдено целевое утверждение, которое может быть вновь согласовано, или не будут исчерпаны все предшествующие целевые утверждения. Если слева нет целевых утверждений, то конъюнкцию целевых утверждений согласовать нельзя. Однако, если предшествующее целевое утверждение может быть согласовано вновь, Пролог возобновляет процесс доказательства целевых утверждений слева направо, начиная со следующего справа целевого утверждения. Описанный процесс смещения влево для повторного согласования целевого утверждения и возвращения вправо носит название механизма возврата.

**Возврат**

Пролог производит доказательство конъюнкции целевых утверждений слева направо. При этом может встретиться целевое утверждение, согласовать которое не удается. Если такое случается, то происходит смещение влево до тех пор, пока не будет найдено целевое утверждение, которое может быть вновь согласовано, или не будут исчерпаны все предшествующие целевые утверждения. Если слева нет целевых утверждений, то конъюнкцию целевых утверждений согласовать нельзя. Однако, если предшествующее целевое утверждение может быть согласовано вновь, Пролог возобновляет процесс доказательства целевых утверждений слева направо, начиная со следующего справа целевого утверждения. Описанный процесс смещения влево для повторного согласования целевого утверждения и возвращения вправо носит название механизма возврата.

**Трассировка**

меньше(X.Y) :-

X<Y, write(X),

write ('меньше, чем'),write(Y).

меньше(Х.У) :-

Y<X, write(Y),

write ('меньше, чем'),write(X).

Целевое утверждение ?- меньше (5, 2). сопоставляется с головой первого утверждения при Х=5 и У=2. Однако не удается согласовать первый член конъюнкции в теле утверждения X<Y. Значит, Пролог не может использовать первое утверждение для согласования целевого утверждения меньше(5, 2). Тогда Пролог переходит к следующему утверждению, голова которого сопоставима с целевым утверждением. В нашем случае это второе утверждение. При значениях переменных Х=5 и Y=2 тело утверждения согласуется. Целевое утверждение меньше(5,2) доказано, и Пролог выдает сообщение "2 меньше, чем 5". Запрос ?-меньше (2, 2). сопоставляется с головой первого утверждения, но тело утверждения согласовать не удается. Затем происходит сопоставление с головой второго утверждения, но согласовать тело опять-таки оказывается невозможно. Поэтому попытка доказательства целевого утверждения меньше(2, 2) заканчивается неудачей.

Такой процесс согласования целевого утверждения путем прямого продвижения по программе мы называем прямой трассировкой (forward tracking).

1. **Отсечение в прологе. Виды отсечений.**

специальное целевое утверждение «!,», называемое отсечением. Отсечение реализуетяс следующим образом: после согласования ЦУ, стоящего перед отсечением, все предположения с тем же предикатом расположенные после отсечения не рассматриваются.

Удаление отсечений из программы может привести к изменению ее декларативного смысла. Но бывают также такие случаи, когда отсечение на него не влияло. Использование отсечений последнего типа требует меньшей осторожности, и поэтому такие отсечения иногда называют "зелеными отсечениями". С точки зрения наглядности программы такие отсечения "невинны" и их использование вполне приемлемо. При чтении программы их можно просто игнорировать. Напротив, отсечения, влияющие на декларативный смысл, называются "красными". Красные отсечения — это такие отсечения, которые делают программу трудной для понимания, и их нужно применять с особой осторожностью.

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

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

f( X, 4).

Примеры, использующие отсечение:

1. Вычисление максимума

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

max(X,Y,Y):-X<Y.

?-max(10,15,Y),write(Y). %15Yes.

1. Принадлежность элемента списку

member(El,[El|\_]):-!.

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

?-member(a, [a,c]). %Yes

1. Добавление элемента в список

add(El,Lst,[El|Lst]).

?-add(a, Lst1, Lst2), Lst1=[b,v,c], write(Lst2). %[a,b,v,c]Yes.

add(El,Lst,Lst):-member(El,Lst),!.

add(El,Lst,[El|Lst]).

?-add(a, Lst1, Lst2), Lst1=[b,a,c], write(Lst2).%No

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.**

Типовые предикаты - позволяющие определить к какому типу относится объект.

1. var( X) - Эта цель успешна, если X в текущий момент — не конкретизированная переменная.
2. nonvar( X) - Эта цель успешна, если X — терм, отличный от переменной, или если X — уже конкретизированная переменная.
3. atom( X) - Эта цель истинна, если X обозначает атом.
4. integer( X)- Цель истинна, если X обозначает целое.
5. atomic( X) - Цель истинна, если X обозначает целое или атом.
6. **Арифметические действия в Прологе.**

Отметим некоторые особенности стандарта языка при использовании арифметических действий.

1. для того, чтобы действительно вызвать какое-либо арифметическое действие, необходимо использовать оператор is.
2. Во-вторых, перед использованием переменных в арифметических действиях, они должны быть конкретизированы значениями.
3. 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.

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.

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.

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.

При работе с числовой информацией очень часто используются операции сравнения.

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.**

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.

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.

Следует отметить, что добавленный факт будет присутствовать в текущей БД до конца текущего сеанса работы. Однако он будет «не видимым» для пользователя, т. е. не будет отображаться в рабочем окне. Для того чтобы факт появился в рабочем окне, необходимо использовать предикат assert\_in.

К этой же группе предикатов относятся родственные предикаты asserta и assertz (последний в SP отсутствует). Предикат asserta добавляет предложение в начало БД, а assertz – в конец (аналогия assert).

Следует отметить, что из рабочего окна удаляемый факт не исчезнет. Для его «видимого» исчезновения необходимо использовать предикат retract\_in.

Для удаления всех вхождений данного факта в БД необходимо использовать предикат retractall(<the fact>).

Польза от таких предикатов очевидна, пользователь может динамически изменять БД в момент выполнения программы. Очевиден и тот факт, что неумелое, неграмотное их использование приведет к полному изменению исходной БД и как результат – непредсказуемые результаты. В отдельных случаях следует вовсе отказаться от использования предикатов этой группы.

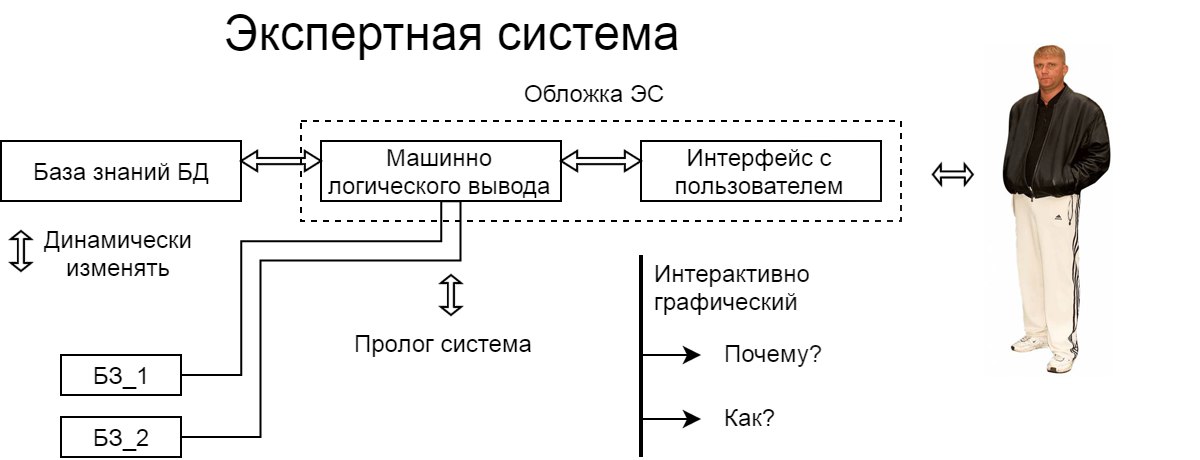
Интерфейс с программой осуществляется посредством предиката read(Term) или read(Term,String), первый аргумент которого – вводимый терм, т. е. переводимое слово, а второй (если он имеется) – надпись на заголовке диалогового окна.

1. **Пролог и ИИ.**
2. **Экспертные системы.**

пример применения Пролога в области искусственного интеллекта - экспертная система.

Функции ЭС:

1. Диалог с пользователем, интерфейс
2. Выбор из базы знаний



Экспертными системами обычно называют программы, которые могут заменить эксперта в какой-то предметной области. Мы построим классификационную экспертную систему, которая будет пытаться угадать загаданное человеком животное. Если загаданное человеком животное окажется неизвестно нашей программе, у нее будет возможность пополнить свою базу знаний новой информацией.

пределим два предиката внутренней базы данных, которые позволят нам хранить информацию о животных.

Один из них предназначен для хранения характеристик животных и будет иметь два аргумента: первый — номер свойства, второй — его словесное описание.

Небольшой базовый набор свойств может выглядеть, например, так:

cond(1,"кормит детенышей молоком").

cond(2,"имеет перья").

cond(3,"плавает").

cond(4,"ест мясо").

cond(5,"имеет копыта").

cond(6,"летает").

cond(7,"откладывает яйца").

cond(8,"имеет шерсть").

cond(9,"имеет полосы").

cond(10,"имеет пятна").

cond(11,"имеет черно-белую окраску").

cond(12,"имеет длинную шею").

cond(13,"имеет длинные ноги").

cond(14,"имеет щупальца").

Второй предикат будет хранить описание животных. Первый его аргумент — название животного, второй — список, элементами которого являются номера свойств, присущих данному животному.

Выглядеть эта база знаний может примерно следующим образом:

rule("гепард",[1,4,8,10]).

rule("тигр",[1,4,8,9]).

rule("жираф",[1,5,8,10,12,13]).

rule("зебра",[1,5,8,9,11]).

rule("страус",[2,14]).

rule("пингвин",[2,3,11]).

rule("орел",[2,6]).

rule("кит",[1,3,11]).

cond\_is(N,'1') /\* если загаданное животное имеет свойство с номером N \*/

cond\_is(N,'2') /\* если загаданное животное не имеет свойства с номером N \*/

Процесс отгадывания задуманного животного будет проходить следующим образом.

animals:–

rule(X,L),

check(L),

nl,write("Я думаю это ",X),

nl,write("Я прав? (1 — да, 2 — нет)"),

read\_true\_char(C),C='1',!.

animals:–

nl,write("Я не знаю, что это за животное"),nl,

nl,write("Давайте добавим его в мою базу

знаний."),nl,

update.

update:–

nl,write("Введите название животного:"),

readln(S),

add\_cond(L), /\* указываем свойства животного \*/

assert(rule(S,L),knowledge),

/\* добавляем информацию в базу

знаний\*/

save("animals.ddb",knowledge).

1. **Двоичные деревья, двоичные справочники.**

Другая рекурсивная структура – двоичные деревья, которые представляются элементом, принадлежащим корню дерева, а также левым и правым поддеревьями. Причем и левое, и правое поддеревья в свою очередь являются деревьями. Граничный случай – пустое дерево, обозначаемое как void. Пример определения двоичного дерева следующий:

binary\_tree(\_,\_,\_).

binary\_tree(Element,Left,Rigth):-binary\_tree(Left),binary\_tree(Rigth).

В отличие от списка для двоичного дерева характерна двойная рекурсия, то есть рекурсия по левому и правому поддеревьям. Это хорошо видно из второго предложения. Следует отметить, что эта программа уже не минимально рекурсивная.

Поставьте перед Пролог-системой вопрос

?- binary\_tree(a, binary\_tree(b,void,void), binary\_tree(c,void,void)).

и проанализируйте ответ.

Над деревьями определен ряд операций, наиболее важная из которых – принадлежность элемента дереву. Определение предиката может выглядеть следующим образом:

tree\_member(X,binary\_tree(\_,\_,\_)).

tree\_member(X,binary\_tree(\_,Left,\_)):-tree\_member(X,Left).

tree\_member(X,binary\_tree(\_,\_,Rigth)):-tree\_member(X,Rigth).

Здесь также имеет место двойная рекурсия во втором и третьем предложениях. Эти предложения разделены скорее для наглядности, чем по необходимости. Их можно объединить в одно, используя дизъюнкцию целей.

Декларативный смысл отношения достаточно очевиден: элемент принадлежит двоичному дереву, если он является корнем дерева. Если элемент не корень, то он принадлежит или левому или правому поддереву.

Имея подобный предикат, пользователь может проверить объект на «тип» дерева, например, задав вопрос:

?-tree\_member(b,binary\_tree(a, binary\_tree(b,void,void), binary\_tree(c,void,void))).

1. **Решение логических задачек, головоломок.**

Одно из основных применений языка Пролог – решение задач логики, головоломок, игровых задач.

Один из подходов решения логических задач рассмотрим на следующем примере.

*Три друга заняли первое, второе и третье места в соревнованиях универсиады. Друзья – разной национальности, зовут их по-разному, и любят они разные виды спорта.*

*Майкл предпочитает баскетбол и играет лучше, чем американец. Израильтянин Саймон играет лучше теннисиста. Игрок в крокет занял первое место.*

*Кто является австралийцем? Каким спортом занимается Ричард?*

Подобные логические головоломки изящно решаются посредством конкретизации значений подходящей структуры данных и выделения значения, приводящего к решению. Каждый ключ к решению преобразуется в факт относительно структуры данных. Это может быть сделано с использованием абстракции данных до определения точной формы структуры данных. Проанализируем первый ключ к разгадке: «Майкл предпочитает баскетбол и играет лучше, чем американец». Очевидно, речь идет о двух разных людях. Одного зовут Майкл, и занимается он баскетболом, в то время как второй – американец. Кроме того, Майкл лучше играет в баскетбол, чем американец. Предположим, что Друзья – структура данных, подлежащая конкретизации, тогда искомый ключ может быть выражен следующей конъюнкцией целей:

*играет\_лучше(Мужчина\_1, Мужчина\_2, Друзья),*

*имя(Мужчина\_1, Майкл), спорт(Мужчина\_1, баскетбол),*

*национальность(Мужчина\_2, американец).*

Аналогично второй ключ можно представить конъюнкцией целей:

*играет\_лучше(Мужчина\_1, Мужчина\_2, Друзья),*

*имя((Мужчина\_1,Саймон),*

*национальность(Мужчина\_1,израильтянин),*

*спорт(Мужчина\_2, теннис).*

Наконец, третий ключ к разгадке выразится следующим образом:

*первый(Друзья, Мужчина), спорт(Мужчина, крокет).*

Базовый предикат для решения головоломок:

*решить\_головоломку(Головоломка, Решение).*, где Решение является решением головоломки Головоломка. Головоломка представляется структурой головоломка(Ключи, Вопросы, Решение). Общую схему решения головоломки можно представить следующим образом:

*решить\_головоломку(головоломка(Ключи,Вопросы,Решение), Решение):- решить(Ключи), решить(Вопросы).*

*решить([]).*

*решить([Ключ|Ключи]):-решить(Ключ).*

Каждый человек имеет три атрибута и представляется структурой вида: друг(Имя, Страна, Спорт)., а в качестве структуры данных последовательность трех элементов, представляющим список: [друг(N1,C1,S1), друг(N2, C2, S2), друг(N3, C3, S3)]. Ключи и вопросы в данном примере предлагается составить самостоятельно. Для проверки: Майкл – австралиец, а Ричард играет в теннис.

Эта схема решения дает представление об общем подходе для решения головоломок. Часто подобную схему называют решателем головоломок.

1. **Представление графов в Прологе.**
2. **Как Пролог-система отвечает на вопросы пользователя.**
3. **S-выражения Лиспа, Примеры S-выражений.**

В языке Лисп как для вызова функций, так и для записи выражений принята единообразная форма записи, при которой имя функции и ее аргументы записываются внутри скобок. Общий формат записи следующий:

(name\_function arg\_1 arg\_2 … arg\_n), где

name\_function – имя функции;

arg\_I – аргументы.

Такая форма записи называется S-выражением.

Таким же образом записываются арифметические выражения, например, (+ 2 3), (- 4 3), (\* x (+ y z)), etc.

Характерной особенностью S-выражения является наличие скобок. Действительно, их количество и местоположение влияет на смысл и результат S-выражения. Кроме скобок в любом S-выражении имеются элементы, называемые атомами. Различают два типа атомов: символьные и числовые. Символьный атом состоит обычно из букв, но может содержать и другие символы (числа, символ подчеркивания), но обязательно должен содержать хотя бы один символ, отличающий его от числа. Символьный атом нельзя расчленять на составляющие символы. Единственная операция, допустимая над символами – операция сравнения. Числовой атом является последовательностью цифр, возможно следующих за знаком.

S-выражение может состоять из произвольной смеси символьных и числовых атомов. Простейшая форма S-выражения – атом.

Формы S-выражений:

* один атом
* список атомов
* список с подсписками

QUOTE или ' - блокирует вычисление значения S-выражения.

1. **Понятие строго функционального языка.**

Функциональное программирование – это способ составления программ, в которых единственным действием является вызов функции, единственным способом расчленения программ на части (модульность) является введение имени для функции, а единственным правилом композиции – оператор суперпозиции функции. Теоретической базой для функционального программирования является лямбда-исчисление Алонсо Черча. Не используются операторы (присваивания, цикла, передачи управления, etc.), блок-схемы, передачи управления.

1. Структуризация данных
2. Функции более высокого порядка
3. **Определение функции, безымянная лямбда-функция.**

Функция есть отображение (mapping), которое однозначно отображает одни значения на другие.

В языке Лисп как для вызова функций, так и для записи выражений принята единообразная форма записи, при которой имя функции и ее аргументы записываются внутри скобок. Общий формат записи следующий:

(name\_function arg\_1 arg\_2 … arg\_n), где

name\_function – имя функции;

arg\_I – аргументы.

Такая форма записи называется S-выражением.

Ключевое понятие, базис функционального программирования лямбда-функция.

В Лиспе лямбда-выражение имеет вид

*(lambda(x1,x2,x3,…xn) fn.).*

Символ lambda означает, что речь идет об определении функции. Символы xi – формальные параметры определения, a fn – тело функции. Телом функции является произвольная форма. Под формой следует понимать любое выражение, значение которого может вычислить интерпретатор Лиспа.

Функцию вычисления суммы двух чисел можно определить следующим образом: (lambda (x y)(+ x y)). Здесь x и y – формальные параметры, а (+ x y) – тело лямбда-выражения.

1. **Списки, операции над списками.**

Список в Лиспе это упорядоченная последовательность, элементами которой являются атомы либо списки. Списки заключаются в круглые скобки, элементы списка разделяются пробелами. Список всегда начинается с открывающей скобки и заканчивается закрывающей скобкой. Например, следующий список состоит из трёх символов и одного подсписка, который в свою очередь состоит из двух атомов:

  (x y (z 1) z)

  Список это многоуровневая структура данных, в которой открывающие и закрывающие скобки находятся в строгом соответствии.

Список, в котором нет ни одного элемента, называется пустым списком и обозначается () или символом nil. Пустой список - это не то же самое, что "ничего".

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.

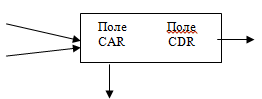


Рис. 1. Графическое представление точечной пары

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.

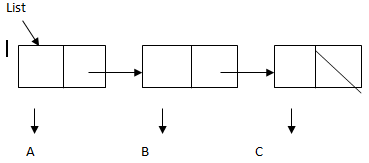


Рис. 2. Внутренне представление одноуровнего списка (A B C)

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.

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))).

1. **Точечное представление списков. Примеры списков точечной записи.**

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.

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.**

Символ — это имя, состоящее из букв, цифр и[ли] специальных знаков { [ < ! ? . - @ # $ % ^ & \* \_ + / > ] }. Символ обозначает произвольный объект или явление из прикладной области, вспомогательные объекты внутри программы, или некоторую структуру исходного кода. Имя функции — это тоже символ. Символ всегда имеет значение.

При помощи функции set символу можно присвоить или связать с ним некоторое значение. set вычисляет оба аргумента. Если перед первым аргументом нет апострофа, то с помощью функции set можно присвоить значение имени, которое получается путём вычисления.

Наряду с функцией set связать символ с его значением можно с помощью функции setq. Эта функция отличается от set тем, что она вычисляет только свой второй аргумент. Об автоматическом блокировании вычисления первого аргумента напоминает буква q в имени функции. При использовании функции setq отпадает надобность в знаке апострофа перед первым аргументом.

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

Рекурсия – один из приемов, который встречается практически во всех видах программирования. В декларативных языках рекурсия играет доминирующую роль. С одной стороны это связано с рекурсивной внутренней структурой декларативных языков. С другой – в нечисловом программировании часто используются структуры, рекурсивные по своей природе, например, списки, деревья, графы.

В математике рекурсивным началом можно считать натуральное число, если положить, что отправной точкой для любого натурального числа считать цифру 0. Любое натуральное число можно определить как функцию следования S от числа 0, т. е. 0, S(0), S(S(0)), … , S(S(S(…S(0)…))). Благодаря функции следования любое натуральное число представляется как рекурсивная структура с граничным условием, равным нулю. В языке Пролог напрямую запрограммировать функцию следования не представляется возможным, однако, используя математические операции можно ее промоделировать с определенной степенью верности.

Попытка определения натурального числа в терминах языка Пролог представлена ниже.

natural\_number(0).

natural\_number(X):- Y is X - 1, Y>=0, natural\_number(Y), write(X), nl.

?-natural\_number(5).

В Лиспе рекурсия может быть по значению или по аргументам. Если рекурсивный вызов является выражением, формирующим результат функции, то имеет место рекурсия по значению. Если же рекурсивный вызов является аргументом некоторой другой функции, формирующей окончательный результат, то имеет место рекурсия по аргументам.

Различают следующие формы рекурсии:

– простая рекурсия, если рекурсивный вызов встречается в некоторой ветви не более одного раза;

– параллельная рекурсия, если рекурсия встречается одновременно в нескольких аргументах функции;

– взаимная рекурсия, если при определении одной функции, вызывается другая, которая в свою очередь вызывает первую;

– рекурсия более высокого порядка, если в качестве аргумента рекурсивного вызова выступает рекурсивный вызов.

1. **Виды рекурсий в Лиспе.**

Основное средство функционального программирования – рекурсия. Любое определение, описывающее в терминах самого себя, является рекурсивным.

Основная идея рекурсивного определения состоит в том, что функцию можно свести к самой определяемой функции, но с более «простыми» аргументами. Вычисление такой функции заканчивается в тот момент, когда оно сводится к известным начальным значениям.

Рекурсивная функция всегда должна содержать, по крайней мере, одну терминальную ветвь – условие окончания. В тот момент, когда вычисления доходят до рекурсивной ветви, функционирующий процесс приостанавливается и запускается новый такой же процесс, но на новом уровне. Прерванный процесс запоминается в программном стеке, его выполнение откладывается до завершения нового процесса. В свою очередь, новый процесс может быть приостановлен другим процессом и т.д.

В Лиспе рекурсия может быть по значению или по аргументам. Если рекурсивный вызов является выражением, формирующим результат функции, то имеет место рекурсия по значению. Если же рекурсивный вызов является аргументом некоторой другой функции, формирующей окончательный результат, то имеет место рекурсия по аргументам.

Различают следующие формы рекурсии:

– простая рекурсия, если рекурсивный вызов встречается в некоторой ветви не более одного раза;

– параллельная рекурсия, если рекурсия встречается одновременно в нескольких аргументах функции;

– взаимная рекурсия, если при определении одной функции, вызывается другая, которая в свою очередь вызывает первую;

– рекурсия более высокого порядка, если в качестве аргумента рекурсивного вызова выступает рекурсивный вызов.

MEMBER проверяет, принадлежит ли элемент списку

(defun MEMBER1 (a l)

(cond ((null l) nil) ; l пуст?

((eql (car l) a) l) ; a найден?

(t (MEMBER1 a (cdr l))))); а - в хвосте?

APPEND объединяет два списка

(defun append1 (x y)

(cond ((null x) y)

(t (cons (car x) (append1 (cdr x) y)))))

REMOVE удаляет все вхождения данного элемента

(defun remove1 (a l)

(cond ((null l) nil)

((eql a (car l))

(remove1 a (cdr l)))

(t (cons (car l) (remove1 a (cdr l))))))

Reverse - обращает список

(defun reverse1 (l)

(cond ((null l) nil)

(t (append (reverse1 (cdr l))

(cons (car l) nil )))))

(print(reverse `(a b c))) -> (C B A)

1. ***Взаимная рекурсия.***

Взаимная рекурсия. Рекурсия является взаимной между одной и более функциями, если они вызывают друг друга. Для примера определим функцию реверсирования списка в виде двух взаимно рекурсивных функций следующим образом:

(defun reverse(lst)

(cond ((atom lst) lst) (t (permutation lst nil)))).

В определении функции reverse вызывается функция permutation, которая, как будет видно ниже, в свою очередь вызовет функцию reverse.

(defun permutation(lst res)

(cond ((null lst) res)

(t (permutation (cdr lst) (cons (reverse (car lst)) res)))).

1. ***Параллельная рекурсия.***

Рекурсия называется параллельной, если она встречается одновременно в нескольких аргументах функции. Рассмотрим параллельную рекурсию на примере преобразования списочной структуры в одноуровневый список:

(defun transform(lst)

(cond ((null lst) nil)

((atom lst) (cons lst nil))

(t (append (transform (car lst))

(transform (cdr lst)))))).

Здесь в качестве аргументов функции append выступают два рекурсивных вызова. Следует помнить, что параллельность в данном случае текстуальная, вычисление во времени происходит последовательно. Параллельная рекурсия применяется в том случае, если необходимо обработать список по всем уровням. В приведенном примере, если голова списка не является атомом (т.е. представляет собой списочную структуру), то по отношению к ней применяется рекурсивный вызов. Параллельная рекурсивная ветвь вызывает аналогичные вычисления относительно хвоста списка. Таким образом, внешне создается иллюзия параллелизма.

1. ***Рекурсия более высокого порядка.***

Более сложные вычисления можно осуществить с помощью рекурсии более высокого порядка. В качестве аргумента рекурсивного вызова может выступать рекурсивный вызов, в этом случае имеет место рекурсия более высокого порядка. Классическим примером рекурсии первого порядка является функция Аккермана, определение которой приведено ниже:

(defun ackermann(m n)

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

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

(t (ackermann (- m 1)

(ackermann m (- n 1)))))).

Эта функция не используется в практическом программировании. Ее вычисление достаточно сложно и время растет лавинообразно уже при малых значениях аргументов. Большее значение функция Аккермана имеет в теоретическом программировании при исследовании поведения рекурсивных функций

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

в Лиспе принята единообразная скобочная форма записи S-выражений, что позволяет сделать вывод, что смысл любого выражения зависит от его контекста. Другими словами, одно и то же выражение в одном случае может выступать как данное, а в другом – как функция.

Аргумент функции, значением которого является функция, называется функциональным аргументом, а функция, имеющая функциональный аргумент – функционалом. Функция, которая возвращает в качестве результата другую функцию, называется функцией с функциональным значением.

1. **Замыкания.**

Замыкание - это связь свободных переменных в некотором контексте.

Замыкание - это функция и контекст его применения. Сформированный на время вычисления контекст после окончания ее вычисления пропадает, и на него невозможно позже сослаться или вернуться к нему. Часто бывает полезным и необходимым, чтобы функция для продолжения вычислений могла запомнить связи и состояние более раннего контекста. Это достигается с помощью таких функциональных объектов, в которых вместе с самим описанием вычислений сохраняется контекст момента определения функционального объекта, защищённый от более позднего контекста вызова. Пару, состоящую из функции (лямбда-выражения) и контекста, называют замыканием или лексическим замыканием. Замыкание, как и имя функции или лямбда-выражение, можно использовать в качестве функционального аргумента

> (defun generate-even (\*evenum\*)

(function

(lambda()

(setq \*evenum\* (+ \*evenum\* 2)))))

> (setq gen-even-1 (generate-even 0))

> (funcall gen-even-1)

2

> (funcall gen-even-1)

4

> (funcall gen-even-1)

6

1. **Лисп в ИИ.**
2. **Декларативный смысл 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.

Декларативный смысл программы определяет, является ли данная цель истинной (достижимой) и, если да, при каких значениях переменных она достигается.

1. **Порядок предложений и целей в предложениях.**

Опасность бесконечного цикла

Рассмотрим следующее предложение: р :- р.

В нем говорится: "р истинно, если р истинно". С точки зрения декларативного смысла это совершенно корректно, однако в процедурном смысле оно бесполезно. Более того, для пролог-системы такое предложение может породить серьезную проблему. Рассмотрим вопрос:

?- р.

При использовании вышеприведенного предложения цель р будет заменена на ту же самую цель р; она в свою очередь будет заменена снова на р и т.д. В этом случае система войдет в бесконечный цикл, не замечая, что никакого продвижения в вычислениях не происходит.

Данный пример демонстрирует простой способ ввести пролог-систему в бесконечный цикл. Однако подобное зацикливание могло встретиться и в некоторых других программах.

предок( X, Z) :- родитель(X, Z).

предок( X, Z) :- родитель(X, Y), предок(Y, Z).

В соответствии с декларативной семантикой Пролога мы можем, не меняя декларативного смысла, изменить

1. порядок предложений в программе
2. порядок целей в телах предложений.

Возможны, поэтому, четыре варианта данной программы, все с одинаковым декларативным смыслом. Эти четыре варианта можно получить, если

1. поменять местами оба предложения
2. поменять местами цели в каждом из этих двух последовательностей предложений.

% Исходная версия

пред1( X, Z) :- родитель( X, Z).

пред1( X, Z) :- родитель( X, Y), пред1( Y, Z).

% Вариант а: изменение порядка предложений в исходной версии

пред2( X, Z) :- родитель( X, Y), пред2( Y, Z).

пред2( X, Z) :- родитель( X, Z).

% Вариант b: изменение порядка целей во втором предложении исходной версии

пред3( X, Z) :- родитель( X, Z).

пред3( X, Z) :- пред3( X, Y), родитель( Y, Z).

% Вариант с: изменение порядка предложений и целей в исходной версии

пред4( X, Z) :-пред4( X, Y), родитель( Y, Z).

пред4( X, Z):- родитель( X, Z).

В последнем случае пролог-система не сможет найти ответа. И выведет на терминал сообщение: "Не хватает памяти". пред2 довольно неэффективна по сравнению с пред1: пред2 производит значительно больший перебор и делает больше возвратов по фамильному дереву.

1. **.**
2. **Ввв**
3. **Ввв**
4. **Ввв**