Министерство образования и науки Российской Федерации
Севастопольский государственный университет
Институт информационных технологий и управления в
технических системах

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

к выполнению лабораторных работ
на тему «Основы программирования на языке Visual Prolog»
по дисциплине «Логическое программирование»
для студентов направления
09.03.01 – «Информатика и вычислительная техника»

Севастополь
2015

УДК 519.6

Методические указания к выполнению лабораторных работ на тему «Основы программирования на языке Visual Prolog» по дисциплине «Логическое программирование» для студентов направления 09.03.01 – «Информатика и вычислительная техника»

дневной формы обучения/ Сост. А.А. Брюховецкий, Д.Ю. Воронин, Ю.П. Николаева – Севастополь: Изд-во СевГУ, 2015. — 59 с.

Целью методических указаний является оказание помощи студентам при подготовке к лекциям и выполнении лабораторных работ на тему «Основы программирования на языке Visual Prolog» по дисциплине «Логическое программирование»

Методические рекомендации рассмотрены и утверждены на заседании кафедры КВТ (протокол №1 от 30 августа 2014 г.).

Допущено учебно-методическим центром и научно-методическим советом СевГУ в качестве методических указаний.

Рецензент: Лелеков С.Г., к.ф-м.н., доцент кафедры кибернетики и вычислительной техники.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Цель – настоящие методические указания предназначены для изучения основных возможностей среды и типовых приемов логического программирования. Приведены необходимые теоретические сведения об основах логического программирования и типовые примеры решения задач, представлены варианты индивидуальных заданий для лабораторных работ в среде ПРОЛОГ и порядок их выполнения, а также контрольные вопросы для самостоятельной работы студентов. Особое внимание уделено вопросам разработки рекурсивных процедур, обработки списков, динамических баз данных и бинарных деревьев.

Пролог — язык программирования, в котором основные конструкции заимствованы из логики, начавшей свое развитие еще во времена Аристотеля более 2,2 тыс. лет назад. Логика же предоставляет точный язык для явного выражения целей, знаний и предположений. Она позволяет сделать заключение об истинности или ложности одних утверждений исходя из знаний об истинности или ложности других. Однако положения традиционной логики при создании языков программирования стали использовать сравнительно недавно.

Осознание того, что вычисление – часть случайного логического вывода привело к возникновению логического программирования, первая реализация которого была осуществлена в семидесятые годы в виде системы Пролог. Суть идеи – представить в качестве программы формальное описание предметной области, а затем сформулировать необходимую для решения задачу в виде цели или утверждения. Построение же решения этой задачи в виде вывода в этой системе предложить самой машине. Последнее возможно, поскольку нужный алгоритм решения (поиска) осуществляется решателем, строящим вывод по определенной стратегии. При такой постановке основная задача программиста сводится к описанию предметной области в виде системы логических формул и отношений на них, которые с достаточной степенью полноты описывают задачу. Этот подход стал возможен благодаря тому, что были получены достаточно эффективные методы автоматического поиска доказательств.

ЛАБОРАТОРНАЯ РАБОТА №1. ВЫПОЛНЕНИЕ И ТРАССИРОВКА ЛОГИЧЕСКИХ ПРОГРАММ В СРЕДЕ PIE

Цель работы: овладеть навыками трассировки логических программ на примере приложения PIE (Prolog Inference Engine - Машина Вывода Пролога), которое включено в комплект системы программирования Visual Prolog. PIE является "классическим" интерпретатором, используя который можно изучать и экспериментировать с Прологом без описания обьектов программы на уровне обьявления предикатов, типов данных, аргументов и т.д.

Для выполнения данной работы необходимо запустить проект PIE, который включен в состав приложений Visual Prolog. Интерфейс среды PIE показан на рисунке ниже:

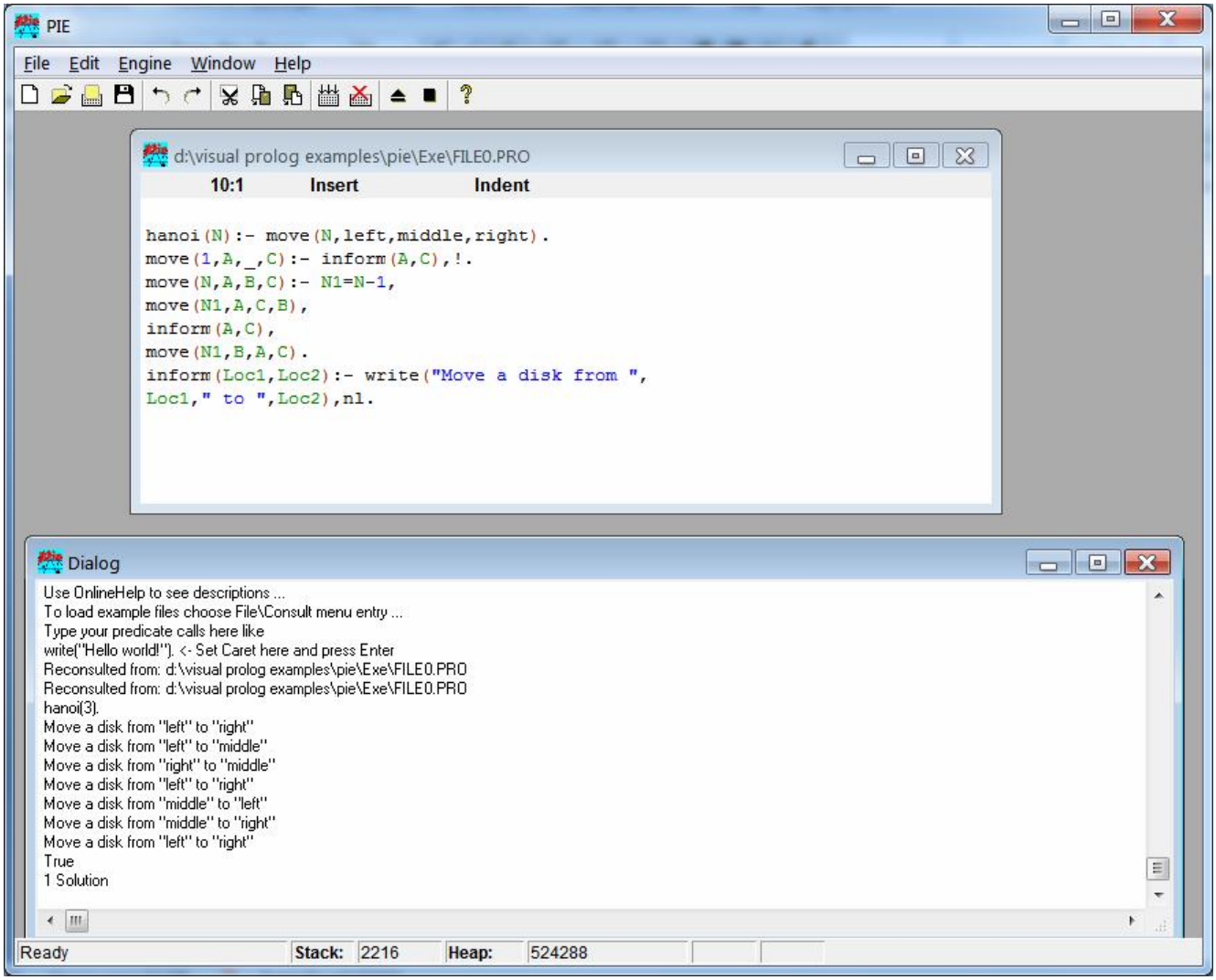


Рисунок 1 – Интерфейс среды PIE

Интерфейс содержит два окна: редактирования программы и диалога. Во втором окне пользователь может вводить запросы к программе. Кроме этих объектов имеется меню и панель инструментов.

Порядок выполнения работы каждой из приведенных ниже задач следующий:

1. Загрузить программу в окно редактирования.
2. Задать необходимые исходные данные.
3. Включить режим трассировки.
4. Сформировать цель и выполнить программу.
5. Проанализировать полученный результат.
6. Оформить отчет по работе.

Пример выполнения лабораторной работы

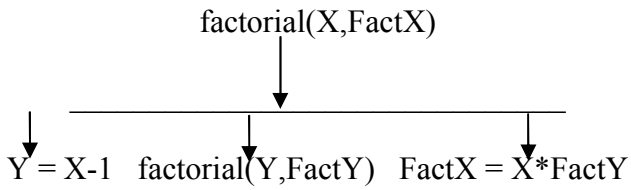
Задача 1. Программа вычисления факториала.

Текст программы

```
% predicates
%  factorial(integer, real)
 % clauses
  factorial(1, 1) :- !.
  factorial(X, FactX) :-
    Y = X-1,
    factorial(Y, FactY),
```

```
    FactX = X*FactY.
goal:- factorial(5, FactX),write(FactX),nl.
```

Представление программы в виде дерева



Пример выполнения программы в режиме трассировки

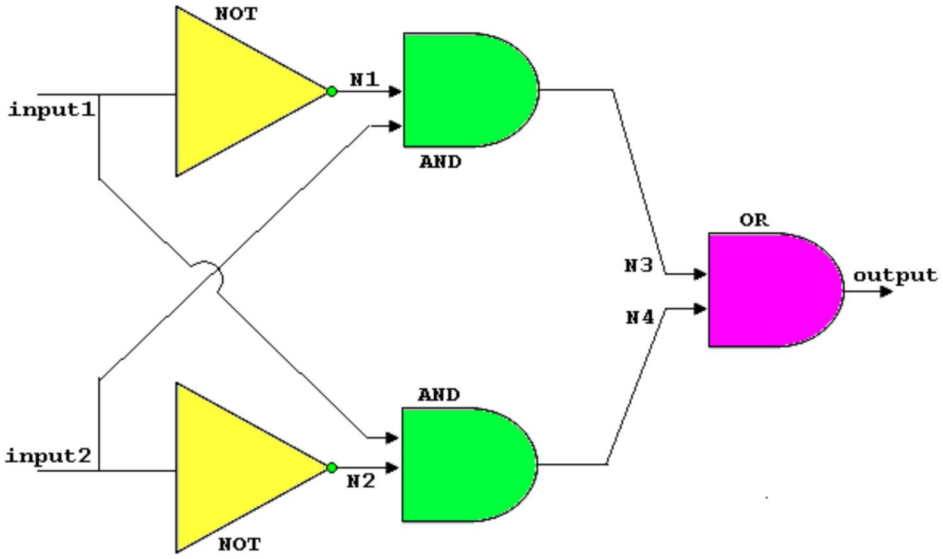
```
factorial(3,X)
Trace: >> CALL:   factorial(3,X$33)
Trace: >> CALL:   Y$34 = 2
Trace: >> RETURN: 2 = 2
Trace: >> CALL:   factorial(2,FACTY$35)
Trace: >> CALL:   Y$36 = 1
Trace: >> RETURN: 1 = 1
Trace: >> CALL:   factorial(1,FACTY$37)

Trace: >> REDO:   factorial(3,6)
Trace: >> REDO:   6 = 6
Trace: >> FAIL:   6 = 6
Trace: >> REDO:   factorial(2,2)
Trace: >> REDO:   2 = 2
Trace: >> FAIL:   2 = 2
Trace: >> REDO:   factorial(1,1)
Trace: >> RETURN: factorial(1,1)
Trace: >> CALL:   FACTY$35 = 2
Trace: >> RETURN: 2 = 2
Trace: >> RETURN: factorial(2,2)
Trace: >> CALL:   X$33 = 6
Trace: >> RETURN: 6 = 6
Trace: >> RETURN: factorial(3,6)
X = 6.

Trace: >> FAIL:   factorial(1,1)
Trace: >> REDO:   1 = 1
Trace: >> FAIL:   1 = 1
Trace: >> FAIL:   factorial(2,FACTY$35)
Trace: >> REDO:   2 = 2
Trace: >> FAIL:   2 = 2
Trace: >> FAIL:   factorial(3,X$33)
1 Solution
```

Задача 2. Моделирование элементов аппаратуры.

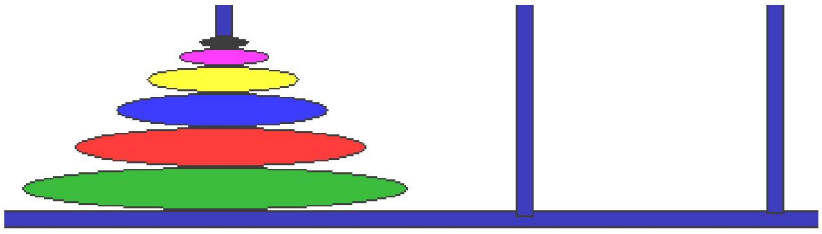
Любая логическая цепь может быть представлена в **Visual Prolog** при помощи предикатов, где они описывают соотношения между входными и выходными сигналами. Основные элементы логики могут быть описаны с помощью не только внешних, но и внутренних связей. В качестве примера сконструируем элемент «исключающее или» из элементов **AND**, **OR** и **NOT**, а затем проверим его работу с помощью полученной программы. Проинтерпретировав результат как таблицу истинности, вы увидите, что указанный элемент действительно выполняет требуемую задачу.



```
domains
d = integer
predicates
not_(D,D)
and_(D,D,D)
or_(D,D,D)
xor(D,D,D)
clauses
not_(1,0).       not_(0,1).
and_(0,0,0).     and_(0,1,0).
and_(1,0,0).     and_(1,1,1).
or_(0,0,0).      or_(0,1,1).
or_(1,0,1).      or_(1,1,1).
xor(Input1,Input2,Output):- not_(Input1,N1),not_(Input2,N2),
        and_(Input1,N2,N3),
        and_(Input2,N1,N4),
        or_(N3,N4,Output).
goal
xor(Input1,Input2,Output)
```

Задача 3. ***Ханойские башни.***

Решение задачи "*Ханойские башни*" – классический пример рекурсии. Эта древняя игра состоит из набора деревянных дисков и трех стержней, прикрепленных к основанию. Каждый диск имеет свой диаметр, а в середине отверстие, достаточное, чтобы диск можно было надеть на стержень. В начале игры все диски надеты на левый стержень в том порядке, как это показано на рисунке. Цель игры заключается в том, чтобы перенести все диски с левого на правый стержень, по одному диску за раз, при этом ни один больший диск нельзя ставить сверху на меньший по диаметру.



Эту задачу можно легко выполнить для одного или двух дисков, но, когда число дисков становится больше трех, задача усложняется. Используем язык **Пролог** для ее решения. Для этой игры есть простая стратегия:

- **один** диск перемещается непосредственно;
- **N** дисков переносятся в три этапа:
  - перенести **N-1** дисков на средний стержень;
  - перенести последний диск на первый стержень;
  - перенести **N-1** дисков со среднего стержня на правый.

В нашей программе есть три предиката:

- предикат **hanoi** с одним параметром, указывающим, со сколькими дисками мы играем;
- предикат **move**, с помощью которого мы описываем перенос **N** дисков с одного стержня на другой, используя третий стержень в качестве промежуточного места для дисков;
- предикат **inform**, который указывает на действие, выполняемое с конкретным диском.

Проанализируем программу.

```
domains
loc =right;middle;left
predicates
hanoi(integer)
move(integer,loc,loc,loc)
inform(loc,loc)
clauses
hanoi(N):- move(N,left,middle,right).
move(1,A,_,C):- inform(A,C),!.
move(N,A,B,C):- N1=N-1,
        move(N1,A,C,B),
        inform(A,C),
        move(N1,B,A,C).
inform(Loc1,Loc2):- write("Move a disk from ",
Loc1," to ",Loc2),nl.
goal
hanoi(3).
```

В результате получим ответ:

```
Move a disk from left to right
Move a disk from left to middle
Move a disk from right to middle
Move a disk from left to right
Move a disk from middle to left
Move a disk from middle to right
Move a disk from left to right
yes
```

Задание на лабораторную работу.

1. Изучить основные возможности среды PIE и выполнить три программы приведенные выше и исследовать процесс их выполнения в режиме трассировки.
2. Выполнить варианты заданий в соответствии с порядковым номером N в списке группы:

   I=N(MOD 11)+1, J=(I+4)(MOD 11)+1, K=(J+4)(MOD 11)+1.
3. Подготовить отчет по установленной форме.

Содержание отчета о выполнении индивидуального задания.

1. Постановка задачи.
2. Представление структур данных и структуры программ в виде деревьев И/ИЛИ в соответствии с заданным вариантом.
3. Примеры обработки запросов. Представление доказательства цели в виде дерева И/ИЛИ.
4. Анализ результатов работы программ в режиме трассировки.
5. Выводы по работе.

Варианты индивидуальных заданий.

Задание№1.

```
% Реализовать  программу выполнения все четырех арифметических операций.
 % predicates
 % operation(symbol, real, real)
% clauses
```

```
 operation("+",X,Y):-Z=X+Y,
  write(X,"+",Y,"=",Z),  nl.
 operation("-",X,Y):-Z=X-Y,
  write(X,"-",Y,"=",Z),  nl.
 operation("*",X,Y):-Z=X*Y,
  write(X,"*",Y,"=",Z),  nl.
  operation("/",X,Y):-Z=X/Y,
  write(X,"/",Y,"=",Z),  nl.
goal:-    X=11, nl, Y=7,  nl,
          operation("+",X,Y),
          operation("-",X,Y),
          operation("*",X,Y),
          operation("/",X,Y).
```

Задание№2. Найти спутники Марса.

```
% domains
 % name=symbol
% predicates
% star(name)
 % planet(name)
 % revolve(name,name)
 % satellite(name,name).
% clauses
 star("sun").
 planet("earth").
 planet("mars").
 revolve("earth","sun").
 revolve("mars","sun").
 revolve("moon ","earth").
 revolve("fobos","mars").
 revolve("demos","mars").
 satellite(X,Y):-planet(Y),revolve(X,Y).
goal:- satellite(X,"mars"),
 write(X,"  satellite mars"),nl,fail.
```

Задание№3. Преобразование градусов в радианы.

```
 % Constants
   % Predicates
 % get_vals ( real )
 % list_vals ( real, real, real, real )
 % run
  % Clauses

 get_vals(Deg):-
      Deg < 361,
      Rad = Deg /(18000 / 314),
      list_vals(Rad,Deg),
      Newangle = Deg + 30,
      get_vals(Newangle).

 list_vals(A,B):-
      write("радианы  градусы"), nl, write(A,"  ",B),nl.
```

```
 run:- get_vals(0).
```

Задание№4. Выдать страны с населением больше 16 млн.

```
 % Predicates
 %  country(symbol,real)
 % (name  ,population)
 % print_countries

 % Clauses
   country(england,37).
   country(france,23).
   country(germany,16).
   country(denmark,24).
   country(canada,73).
   country(chile,25).

   print_countries :-
   country(X,P),       P > 16,          % население больше 16 million
       write(X),  nl, fail.
  goal:- print_countries.
```

Задание№5. Реализация логической функции «сложение по модулю 2».

```
  %domains
 %  D = integer
%predicates
  % not_(D, D)
  % and_(D, D, D)
  % or_(D, D, D)
  % xor(D, D, D)

% clauses
   not_(1, 0).       not_(0, 1).
   and_(0, 0, 0).    and_(0, 1, 0).
   and_(1, 0, 0).    and_(1, 1, 1).
   or_(0, 0, 0).     or_(0, 1, 1).
   or_(1, 0, 1).     or_(1, 1, 1).

   xor(Input1, Input2, Output) :-
      not_(Input1, N1), not_(Input2, N2),
      and_(Input1, N2, N3), and_(Input2, N1, N4),
      or_(N3, N4, Output).
 goal:- xor(In1, In2, Out),
          write("in1=",In1,"in2=",In2,"out=",Out), nl,fail.
```

Задание№6. Пример программы «родственные отношения».

```
mother("Bill", "Lisa").    % 4 факта
father("Bill", "John").
father("Pam", "Bill").
father("Jack", "Bill").

parent(Child, Parent) :-    % правило1
```

```
mother(Child, Parent);      % обратить внимание на точку с запятой!
father(Child, Parent).

grandFather(GrandChild, GrandFather) :-    %правило2
father(GrandChild, Father),
father(Father, GrandFather).

rr1:-father("Pam","Bill").     % 3 цели

rr2:-father("Bill",X),write("X= ",X),nl.

rr3:- father(AA, BB), parent(BB, CC),
          write("AA= ",AA),write("BB= ",BB),write("CC= ",CC),nl,fail.

rr4:- grandFather(GC, GFa),write("child= ",GC),nl,
        write("grand= ",GFa),nl, fail.
```

Задание№7. Сгенерировать ряд чисел с шагом 1.

```
% predicates
  % count(real)
  % clauses
   count(N) :-
      write(N), nl,
      NewN = N+1,
      count(NewN).

goal:- count(1).
```

Задание№8. Вычислить факториал числа X.

```
 % predicates
 %  factorial(integer, real)
  % clauses

   factorial(1, 1) :- !.
   factorial(X, FactX) :-
      Y = X-1,
      factorial(Y, FactY),
      FactX = X*FactY.
   goal:- factorial(5, FactX),write(FactX),nl.
```

Задание№9. Вычислить степень числа через произведение.

```
 % Predicates
 % power_of_two ( integer,integer,integer )
  % Clauses
  power(_,0,_) :- !.          % stop after A^Expon is processed
  power(A,Expon,Z) :-
                                  Exp1 = Expon - 1 ,R = A*Z,
                                  write("  ",Exp1,"   ",R,),nl,power(A, Exp1,R).
  goal:- power(2,5,1).
```

Задание№10.

```
% Реализовать программы вычисления суммы следующих рядов:
% 1 + 2 + 3 + ... + 9 + 10
% 2 + 4 + 6 + ... + 14 + 16
% 10 + 9 + 8 + ... + 2 + 1
% 1 + 3 + 5 + ... + 13 + 15
% domains
% number,sum=integer
% predicates
% sum(number,sum)
% clauses
            sum(11,0).
            sum(Number, Sum) :- New_number=Number+1,
                        sum(New_number,Partial_sum),
                  Sum=Number+Partial_sum.
 goal:-  write("summa :  "),
       sum(1,Sum),  write(Sum).
```

Задание №11. Числа Фиббоначи

```
% ряд чисел фиббоначи
 % predicates
  %   fib(integer,integer)
% clauses

    fib(1,1):-!.
    fib(2,1):-!.

    fib(N,F):- N1=N-1, N2=N-2,
      fib(N1,F1),
      fib(N2,F2),
      F=F1+F2.

goal:- fib(7,X),write(X),nl.
```

ЛАБОРАТОРНАЯ РАБОТА №2. ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ ПРОЛОГ

Цель работы: изучить основы синтаксиса языка Пролог, выработать навыки работы с интерактивной системой Пролог, научиться оформлять отношения между данными на языке Пролог на примере родственных отношений между членами семьи.

Теоретические сведения.

1. Базовые понятия логики предикатов

Пролог является языком исчисления предикатов. Предикат – это логическая формула от одного или нескольких аргументов. Можно сказать, что предикат – это функция, отображающая множество произвольной природы в множество {ложно, истинно}.

Обозначаются предикаты F(x), F(x, y), F(x, y, z) и т. д.

Одноместный предикат F(x), определенный на предметной области M, является истинным, если у объекта x есть свойство F, и ложным – если этого свойства нет. Двухместный предикат F(x, y) является истинным, если объекты x и y находятся в отношении F. Многоместный предикат F(x1 , x2 , x3 ,..., xN) задает отношение между элементами x1, x2, ..., xN и интерпретируется как обозначение высказывания: «Элементы x1 , x2, xN находятся между собой в отношении F».

При разработке логических программ предикаты получают обычно названия, соответствующие семантике описываемой предметной области.

Примеры предикатов:

хищник (X)

супруги (X,Y)

фио (X,Y,Z)

Предикаты, которые нельзя разбить на отдельные компоненты, называют атомарными. Сложные формулы строятся путем комбинирования атомарных предикатов логическими соединителями И, ИЛИ и НЕ. Одноместный предикат при подстановке в него значения переменной становится «нульместным», т.е. высказыванием-предложением, которое является истинным или ложным:

хищник (тигр).

При подстановке в n-местный предикат конкретного значения его «местность» становится равной n–1. Таким образом, каждое высказывание порождается некоторым предикатом, а каждый предикат соответствует множеству высказываний. Задача Пролог-программы заключается в том, чтобы доказать, является ли заданное целевое утверждение следствием из имеющихся фактов и правил.

2. Основные конструкции Пролога

Факт в Прологе **-** это некоторое утверждение, определяющее отношение между объектами или описывающее свойства объекта. Общая форма записи факта имеет следующий вид:

<имя отношения> (имя_объекта_1,имя_объекта_2,..., имя_объекта_N).

Необходимо соблюдать следующие правила:

– имена всех отношений и объектов должны начинаться со строчной буквы;

– сначала записывается имя отношения, затем через запятую записываются имена объектов, а весь список имен объектов заключается в круглые скобки;

– каждый факт должен заканчиваться точкой;

– имена объектов в скобках могут перечисляться произвольно, но в одинаковом порядке для одноименных фактов.

Примеры нескольких отношений «любит» на естественном языке:

Джон любит яблоки.

Катя любит кошек.

и их эквиваленты в прологе:

likes(john, apples).

likes(kate, cats).

Факты, помимо отношений, могут выражать и свойства, например «Снег белый», «Джоана – учительница».

white(snow). teacher(johana).

Атом - имя, число без знака или символ.

Аргумент - имя объекта в круглых скобках.

Объект - название отдельного элемента в конструкции Пролога.

Домен - диапазон и тип значений, определенных для базисного типа данных.

Предикат - утверждение о наличии связи между объектами посредством задания имени отношения перед круглыми скобками и доменов его аргументов.

3. Правила в Прологе

Правило - утверждение о связи некоторого факта с другими фактами. Общая форма записи правила имеет вид:

<заголовок правила>:- <тело правила>.

Заголовок представляет собой предикат. Тело состоит из термов, которые могут быть связаны между собой "," или ";". (","- означает И,";"- означает ИЛИ). Между телом и заголовком стоит символ ":-", который означает ЕСЛИ.
Например, правило, состоящее из двух термов может описано следующим образом: **likes(tom,kathy) :- likes(kathy,computer), likes(kathy,apples).**
На естественном языке это означает : "Тому нравится Кэти, если Кэти нравится компьютер и яблоки."

4. Переменные в Прологе

В Прологе переменные помогают записывать общие факты и правила и задавать вопросы. Утверждение на естественном языке «Джон любит то же, что и Том», может быть записано при помощи переменной Thing в Прологе:

likes(john, Thing) :- likes(tom, Thing).

То есть: john любит объект Thing если tom любит этот же объект Thing.

При задании переменной в Прологе первый символ должен быть заглавной буквой или символом подчеркивания. Объекты john и tom начинаются со строчной буквы, так как они не являются переменными – это идентификаторы, имеющие постоянное значение.

5. Вопросы в Прологе

Однократно задав в Прологе базу данных фактов и правил, мы можем использовать запросы (query), позволяющие выяснять отношения между фактами. Пусть в нашей программе заданы следующие факты:

likes(john, apples).
likes(john, computers).
likes(kate, cats).

Тогда Прологу можно задать, например, вопрос «Любит ли Катя кошек»:

? – likes(kate, cats).

Ответом на этот запрос Пролога будет Yes (Да).

На вопрос «Любит ли Катя яблоки»

? – like(kate, apples).

ответом будет No (Нет), так как в базе данных нет соответствующего факта.

Можно задать Прологу вопрос, который звучал бы на естественном языке, например, «Что нравится Джону?». Для записи этого вопроса используются переменные:

? – like(john, What).

Ответом будет:
What=apples
What=computer
2 Solutions. (2 решения)

6. Универсальные правила

При помощи переменных могут быть выражены не только правила, но и так называемые «универсальные факты». Факт **умножить(0, X, 0)**, например, объединяет все факты, утверждающие, что 0, умноженный на любое число, дает 0. Как и в случае вопроса, можно добиться, чтобы два неопределенных объекта, обозначенных переменными, совпадали – для этого нужно использовать имя одной и той же переменной. Факт **плюс(0, X, X)** означает, что при всех значениях X, 0 плюс X равно X. Фраза «каждый любит себя» может быть записана в Прологе как **любит(X, X).**

7. Составные цели: конъюнкция и дизъюнкция. Общие переменные

Говоря о Прологе, кроме термина «вопрос» используется также термин «цель», являющийся более общим. Трактовка вопросов как целей такова: когда Прологу дается запрос, на самом деле перед ним ставится цель для выполнения. («Найти ответ на вопрос, если он существует»). Цель, состоящая из двух и более частей, называется **сложной целью**, а каждая часть сложной цели называется **подцелью**.

Составные цели можно использовать для поиска решения, в котором обе подцели A и B истины (конъюнкция), разделяя подцели запятой. Можно также искать решения, для которых истина хотя бы одна из подцелей A и B (дизъюнкция), разделяя цели точкой с запятой.

Составные цели представляют интерес в тех случаях, когда имеются одна и более **общие переменные**, то есть переменные, входящие в в две различных подцели общей цели. Областью переменной в составной цели является вся конъюнкция или дизъюнкция. Таким образом, вопрос «*p(X),q(X)»* означает «Существует ли такое *X*, что *p(X)* и *q(X)* одновременно?». Как и в простых вопросах, переменные в конъюнктивных вопросах неявно связаны квантором существования. Примером конъюнктивной цели может служить цель: **? – likes(Person, swimming), likes(Person, reading).** Требуется найти все объекты (переменная Person), которые любят чтение и плавание. Дизъюнктивная цель **? – likes(Person, swimming); likes(Person, reading).** ставит задачу найти всех, кто любит плавание или чтение, или и то и другое одновременно.

8. Пример программы «Родственные отношения».

На рисунке 1 представлен фрагмент «дерева» семейных отношений.

Тот факт, что Том является родителем Боба, можно записать на Прологе так:

родитель(том, боб).

Здесь мы выбрали «родитель» в качестве имени отношения, том и боб - в качестве аргументов этого отношения. Мы записываем такие имена, как том, начиная со строчной буквы, так как они представляют собой идентификаторы, а не переменные. Все дерево родственных отношений, представленных на рисунке 1 описывается следующей пролог-программой:

родитель(пам, боб).
родитель(том, боб).
родитель(том, лиз).
родитель(боб, энн).
родитель(боб, пат).
родитель(пам, джим).

Эта программа содержит шесть предложений. Каждое предложение объявляет об одном факте наличия отношения родитель. После ввода такой программы в Пролог-систему последней можно будет задавать вопросы, касающиеся отношения «родитель». Например, является ли Боб родителем Пат? Этот вопрос можно передать Пролог-системе следующим образом:

?- родитель(боб, пат).

Найдя этот факт в программе, система ответит **yes** (да) Другим вопросом мог бы быть такой:

?- родитель(лиз, пат).

Система ответит **no** (нет), поскольку в программе ничего не говорится о том, является ли Лиз родителем Пат. Программа ответит "нет" и на вопрос

?- родитель(том, бен).

потому, что имя Бен в программе не упоминается.

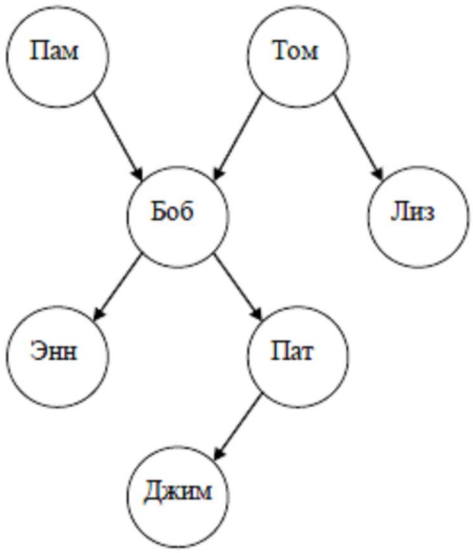


Рисунок 1. Пример дерева родственных отношений

Можно задавать и более интересные вопросы. Например: "Кто является родителем Лиз?"

?- родитель(X, лиз).

На этот раз система ответит не просто "да" или "нет". Она скажет нам, каким должно быть значение **X** (ранее неизвестное), чтобы вышеприведенное утверждение было истинным. Поэтому мы получим ответ:

X = том

Вопрос "Кто дети Боба?" можно передать пролог-системе в такой форме:

?- родитель(боб, X).

В этом случае возможно несколько ответов:

X = энн X = пат.

Нашей программе можно задавать и более общие вопросы: "Кто чей родитель?" Приведем другую формулировку этого вопроса: Найти X и Y такие, что X - родитель Y. На Прологе это записывается так:

?- родитель(X, Y).

Система будет по очереди находить все пары вида "родитель-ребенок". По мере того, как мы будем требовать от системы новых решений, они будут выводиться на экран одно за другим до тех пор, пока все они не будут найдены. Ответы выводятся следующим образом:

X=пам **Y**= боб**;**

X=том **Y**= боб**;**

X=том **Y**= лиз**;**

. . .

Нашей программе можно задавать и еще более сложные вопросы, скажем, кто является родителем родителя Джима? Поскольку в нашей программе прямо не сказано, что представляет собой отношение родитель родителя, такой вопрос следует задавать в два этапа, как это показано на рисунке 2.

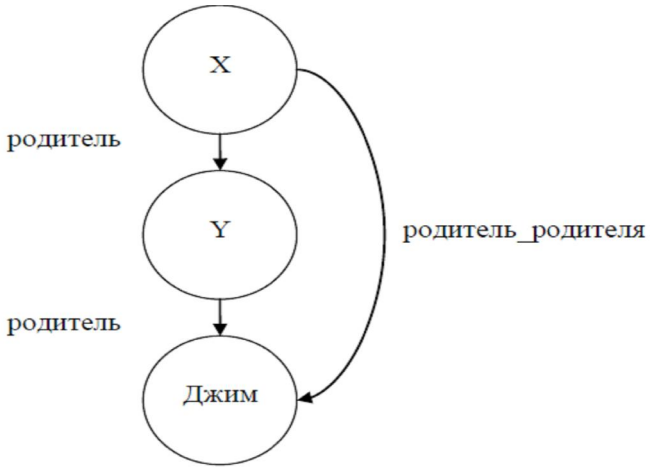


Рисунок 2. Отношение «родитель_родителя», выраженное через композицию двух отношений «родитель»

Кто родитель Джима? Предположим, что это некоторый Y. Кто родитель Y? Предположим, что это некоторый X. Такой составной вопрос на Прологе записывается в виде последовательности двух простых вопросов:

?- родитель(Y, джим**), родитель(X, Y).**

Ответ будет:

X=боб **Y**= пат

Наш составной вопрос можно интерпретировать и так: "Найти X и Y, удовлетворяющие следующим двум требованиям":

родитель (Y, джим**)** и **родитель(X,** У**)**

Если мы поменяем порядок этих двух требований, то логический смысл останется прежним:

родитель (X, Y) и **родитель (Y,** джим**)**

Этот вопрос можно задать нашей пролог-системе и в такой форме:

?- родитель(X, Y), родитель(Y, джим**).**

При этом результат будет тем же. Таким же образом можно спросить: "Кто внуки Тома?"

?- родитель**(том, X),** родитель**(X, Y).**

Система ответит так:

X= боб **Y** = энн**;**

X= боб **Y** = пат

Следующим вопросом мог бы быть такой: "Есть ли у Энн и Пат общий родитель?" Его тоже можно выразить в два этапа:

Какой X является родителем Энн?

Является ли (тот же) X родителем Пат?

Соответствующий запрос к пролог-системе будет тогда выглядеть так:

?- родитель(X, энн**), родитель(X,** пат**).**

Ответ:

X = боб

Определения отношений, в которых используется само отношение, называется рекурсивным. Рекурсия – один из фундаментальных механизмов программирования на Прологе. Рекурсию можно применять для достижения такого же эффекта, какой реализуется при употреблении итеративных управляющих конструкций (циклов) в процедурных языках. Примером использования рекурсии может служить определение отношения «предок». Данное отношение можно выразить с помощью двух правил. Первое правило будет определять

непосредственных предков, а второе – отдаленных. Первое правило легко сформулировать через отношение «родитель»:

предок(X, Z) :- родитель(X, Z).

Аналогичным образом можно попытаться сформулировать второе правило:

предок(X, Z) :- родитель(X, Z).
предок(X, Z) :- родитель(X, Y), родитель(Y, Z).
предок(X, Z) :- родитель(X, Y1), родитель(Y1, Y2), родитель(Y2, Z).
…

Подобное описание отношения будет работать только в определенных пределах, то есть обнаруживать предков лишь до определенной глубины, поскольку длина цепочки людей между предком и потомком ограничена длиной предложений в определении отношения.

Подобные отношения целесообразно описывать с помощью рекурсии. Правило будет выглядеть следующим образом:

Для всех X и Z, X – предок Z, если существует Y, такой, что X – родитель Y и Y – предок Z. или на языке Пролог:

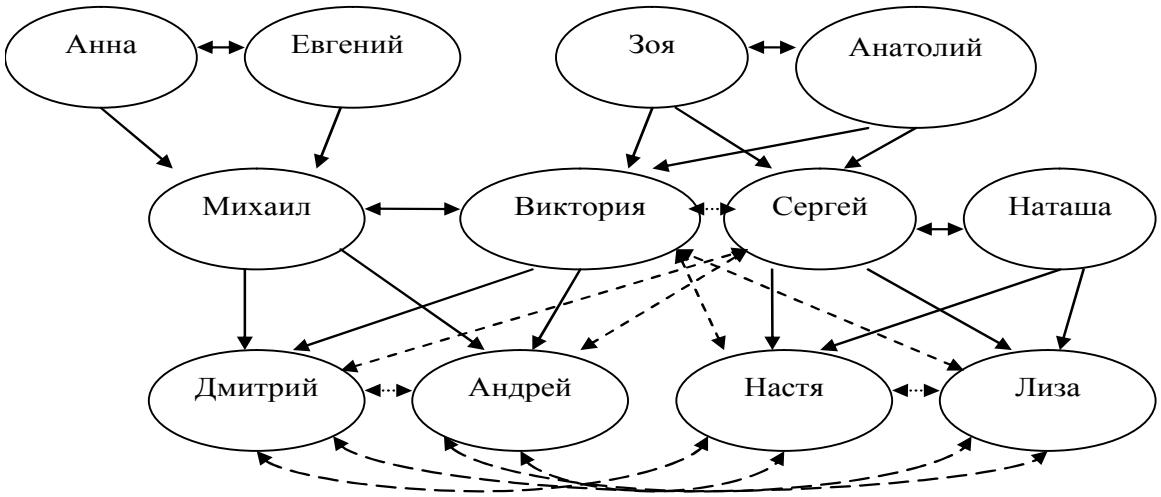
предок(X, Z) :- родитель(X, Y), предок(Y, Z).

Таким образом, определение отношения «предок» будет выглядеть следующим образом:

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

Пример выполнения лабораторной работы

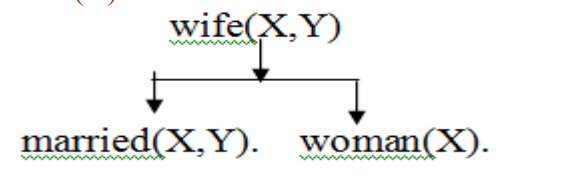
Задан следующий граф родственных отношений:



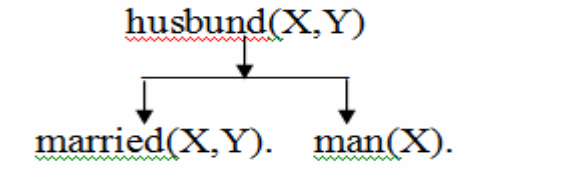
Условное обозначение	Отношение
	состоит в браке
	вертикальная стрелка – родитель, ребёнок, сын/дочь
	дядя/тётя, племянник/племянница
	брат/сестра
	- двоюродный брат/сестра

Представление программы в виде дерева

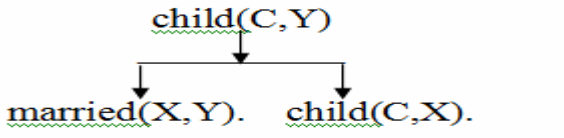
А) правило «жена»:
wife(X,Y):-married(X,Y),woman(X).



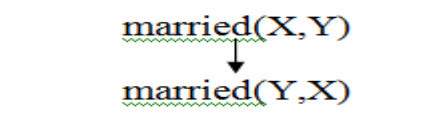
Б) правило «муж»:
husband(X,Y):-married(X,Y),man(X).



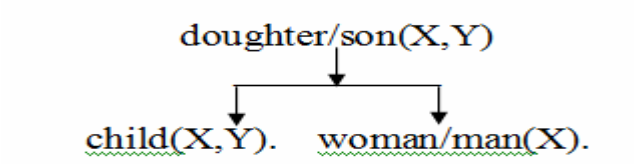
В) правило «ребёнок»:
child(C,Y):-married(X,Y),child(C,X)



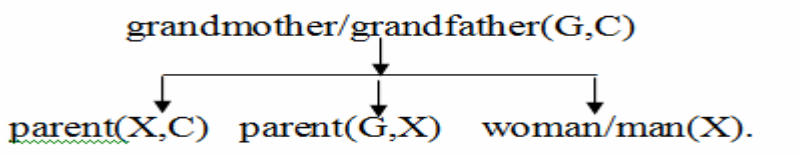
Г) правило «состоит в браке»:
married(X,Y):-married(Y,X).



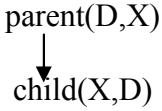
Д,Е) правило «дочь/сын»:
doughter(X,Y):-child(X,Y),woman(X).
son(X,Y):-child(X,Y),man(X).



Ж,З) правило «бабушка/дедушка»:
grandmother(G,C):-parent(X,C),parent(G,X),woman(G).
grandfather(G,C):-parent(X,C),parent(G,X),man(G).

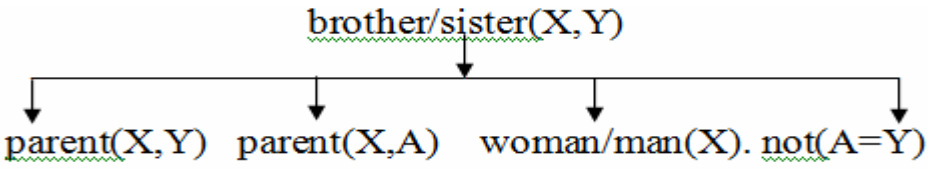


И) правило «родитель»:
parent(D,X):-child(X,D).



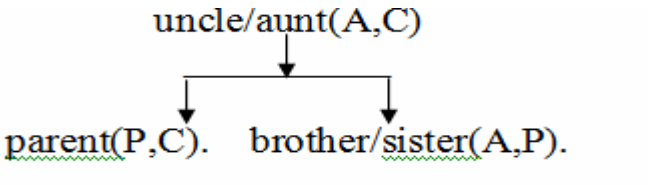
К,Л) правило «брат/сестра»:

brother(A,Y):-parent(X,Y),parent(X,A),man(A), not(A=Y).
sister(A,Y):- parent(X,Y),parent(X,A),woman(A) not(A=Y).



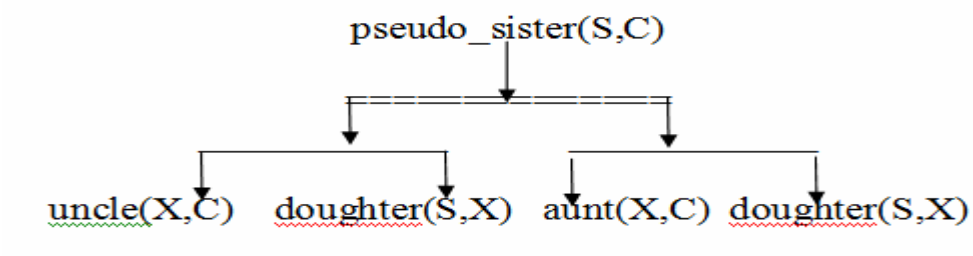
М,Н) правило «дядя/тётя»:

uncle(A,C):-parent(P,C),brother(A,P).
aunt(A,C):- parent(P,C),sister(A,P).



О) правило «двоюродная сестра»:

pseudo_sister(S,C):-uncle(X,C),doughter(S,X).
pseudo_sister(S,C):-aunt(X,C),doughter(S,X).



Текст программы:

```
man(evgeniy).
woman(anna).
man(misha).
woman(viktoria).
man(dmitry).
man(andrey).

married(anna,evgeniy).
child(misha,anna).

married(viktoria,misha).
child(dmitry,viktoria).
child(andrey,viktoria).

woman(zoia).
man(anatoly).
man(sergey).
woman(natasha).
woman(nastia).
woman(liza).

married(zoia,anatoly).
```

```
child(sergey,zoia).
child(viktoria,zoia).
married(sergey,natasha).
child(nastia,natasha).
child(liza,natasha).

wife(X,Y):-married(X,Y),woman(X).  %жена
husband(X,Y):-married(X,Y),man(X). % муж

child(C,Y):-married(X,Y),child(C,X). % Если является ребёнком одного из родителей,
                    % то автоматически является ребёнком второго родителя
doughter(X,Y):-child(X,Y),woman(X).
son(X,Y):-child(X,Y),man(X).
grandmother(G,C):-parent(X,C),parent(G,X),woman(G). %бабушка
grandfather(G,C):-parent(X,C),parent(G,X),man(G).   %дедушка

parent(D,X):-child(X,D). %родитель, если является ребёнком

brother(A,Y):-parent(P,Y),parent(P,A),man(A).   %брат
sister(X,Y):-child(X,P),child(Y,P),woman(X).  %сестра

uncle(U,C):-parent(P,C),brother(U,P). %дядя
aunt(A,C):-child(C,P),sister(A,P). %тётя

pseudo_sister(S,C):-uncle(X,C),doughter(S,X).
pseudo_sister(S,C):-aunt(X,C),doughter(S,X).
```

Пример выполнения программы в режиме трассировки

1. Конъюнкция нескольких вопросов: Миша является родителем Дмитрия И Андрей является братом Дмитрия И Анна является бабушкой Андрея И Евгений является дедушкой Дмитрия. Все вопросы имеют положительный ответ.

parent(misha,dmitry), brother(andrey,dmitry), grandmother(anna,andrey), grandfather(evgeniy,dmitry).

```
Trace: >> CALL:   parent("misha","dmitry")
Trace: >> CALL:   child("dmitry","misha")
Trace: >> CALL:   married(X$0,"misha")
Trace: >> RETURN: married("viktoria","misha")
Trace: >> CALL:   child("dmitry","viktoria")
Trace: >> RETURN: child("dmitry","viktoria")
Trace: >> RETURN: child("dmitry","misha")
Trace: >> RETURN: parent("misha","dmitry")
Trace: >> CALL:   brother("andrey","dmitry")
Trace: >> CALL:   parent(P$1,"dmitry")
Trace: >> CALL:   child("dmitry",P$1)
Trace: >> RETURN: child("dmitry","viktoria")
Trace: >> RETURN: parent("viktoria","dmitry")
Trace: >> CALL:   parent("viktoria","andrey")
Trace: >> CALL:   child("andrey","viktoria")
Trace: >> RETURN: child("andrey","viktoria")
Trace: >> RETURN: parent("viktoria","andrey")
Trace: >> CALL:   man("andrey")
Trace: >> RETURN: man("andrey")
Trace: >> RETURN: brother("andrey","dmitry")
Trace: >> CALL:   grandmother("anna","andrey")
Trace: >> CALL:   parent(X$2,"andrey")
Trace: >> CALL:   child("andrey",X$2)
Trace: >> RETURN: child("andrey","viktoria")
Trace: >> RETURN: parent("viktoria","andrey")
Trace: >> CALL:   parent("anna","viktoria")
Trace: >> CALL:   child("viktoria","anna")
Trace: >> CALL:   married(X$3,"anna")
Trace: >> FAIL:   married("sergey","anna")
Trace: >> FAIL:   child("viktoria","anna")
Trace: >> FAIL:   parent("anna","viktoria")
Trace: >> REDO:   parent("viktoria","andrey")
Trace: >> REDO:   child("andrey","viktoria")
Trace: >> CALL:   married(X$4,X$2)
Trace: >> RETURN: married("anna","evgeniy")
Trace: >> CALL:   child("andrey","anna")
Trace: >> CALL:   married(X$5,"anna")
Trace: >> FAIL:   married("sergey","anna")
Trace: >> FAIL:   child("andrey","anna")
Trace: >> REDO:   married("anna","evgeniy")
Trace: >> RETURN: married("viktoria","misha")
Trace: >> CALL:   child("andrey","viktoria")
Trace: >> RETURN: child("andrey","viktoria")
Trace: >> RETURN: child("andrey","misha")
Trace: >> RETURN: parent("misha","andrey")
Trace: >> CALL:   parent("anna","misha")
Trace: >> CALL:   child("misha","anna")
Trace: >> RETURN: child("misha","anna")
Trace: >> RETURN: parent("anna","misha")
Trace: >> CALL:   woman("anna")
```

```
    Trace: >> RETURN: woman("anna")
Trace: >> RETURN: grandmother("anna","andrey")
Trace: >> CALL:   grandfather("evgeniy","dmitry")
Trace: >> CALL:   parent(X$6,"dmitry")
Trace: >> CALL:   child("dmitry",X$6)
Trace: >> RETURN: child("dmitry","viktoria")
Trace: >> RETURN: parent("viktoria","dmitry")
Trace: >> CALL:   parent("evgeniy","viktoria")
Trace: >> CALL:   child("viktoria","evgeniy")
Trace: >> CALL:   married(X$7,"evgeniy")
Trace: >> RETURN: married("anna","evgeniy")
Trace: >> CALL:   child("viktoria","anna")
Trace: >> CALL:   married(X$8,"anna")
Trace: >> FAIL:   married("sergey","anna")
Trace: >> FAIL:   child("viktoria","anna")
Trace: >> REDO:   married("anna","evgeniy")
Trace: >> FAIL:   married("sergey","evgeniy")
Trace: >> FAIL:   child("viktoria","evgeniy")
Trace: >> FAIL:   parent("evgeniy","viktoria")
Trace: >> REDO:   parent("viktoria","dmitry")
Trace: >> REDO:   child("dmitry","viktoria")
Trace: >> CALL:   married(X$9,X$6)
Trace: >> RETURN: married("anna","evgeniy")
Trace: >> CALL:   child("dmitry","anna")
Trace: >> CALL:   married(X$10,"anna")
Trace: >> FAIL:   married("sergey","anna")
Trace: >> FAIL:   child("dmitry","anna")
Trace: >> REDO:   married("anna","evgeniy")
Trace: >> RETURN: married("viktoria","misha")
Trace: >> CALL:   child("dmitry","viktoria")
Trace: >> RETURN: child("dmitry","viktoria")
Trace: >> RETURN: child("dmitry","misha")
Trace: >> RETURN: parent("misha","dmitry")
Trace: >> CALL:   parent("evgeniy","misha")
Trace: >> CALL:   child("misha","evgeniy")
Trace: >> CALL:   married(X$11,"evgeniy")
Trace: >> RETURN: married("anna","evgeniy")
Trace: >> CALL:   child("misha","anna")
Trace: >> RETURN: child("misha","anna")
Trace: >> RETURN: child("misha","evgeniy")
Trace: >> RETURN: parent("evgeniy","misha")
Trace: >> CALL:   man("evgeniy")
Trace: >> RETURN: man("evgeniy")
Trace: >> RETURN: grandfather("evgeniy","dmitry")
True
1 Solution
```

2. Вопрос индивидуального задания: является ли Настя двоюродной сестрой Дмитрия?

```
pseudo_sister(nastia,dmitry).
Trace: >> CALL:   pseudo_sister("nastia","dmitry")
Trace: >> CALL:   uncle(X$66,"dmitry")
Trace: >> CALL:   parent(P$67,"dmitry")
Trace: >> CALL:   child("dmitry",P$67)
Trace: >> RETURN: child("dmitry","viktoria")
Trace: >> RETURN: parent("viktoria","dmitry")
Trace: >> CALL:   brother(X$66,"viktoria")
Trace: >> CALL:   parent(P$68,"viktoria")
Trace: >> CALL:   child("viktoria",P$68)
Trace: >> RETURN: child("viktoria","zoia")
Trace: >> RETURN: parent("zoia","viktoria")
Trace: >> CALL:   parent("zoia",X$66)
Trace: >> CALL:   child(X$66,"zoia")
Trace: >> RETURN: child("sergey","zoia")
Trace: >> RETURN: parent("zoia","sergey")
Trace: >> CALL:   man("sergey")
Trace: >> RETURN: man("sergey")
Trace: >> RETURN: brother("sergey","viktoria")
Trace: >> RETURN: uncle("sergey","dmitry")
Trace: >> CALL:   doughter("nastia","sergey")
Trace: >> CALL:   child("nastia","sergey")
Trace: >> CALL:   married(X$69,"sergey")
Trace: >> CALL:   married("sergey",X$69)
Trace: >> RETURN: married("sergey","natasha")
Trace: >> RETURN: married("natasha","sergey")
Trace: >> CALL:   child("nastia","natasha")
Trace: >> RETURN: child("nastia","natasha")
Trace: >> RETURN: child("nastia","sergey")
Trace: >> CALL:   woman("nastia")
Trace: >> RETURN: woman("nastia")
Trace: >> RETURN: doughter("nastia","sergey")
Trace: >> RETURN: pseudo_sister("nastia","dmitry")
True
1 Solution
```

Индивидуальное задание и методика выполнения работы

Выполнить вариант заданий **I=N(MOD 30)+1.**

В ходе выполнения работы требуется:

1. Изобразить граф, иллюстрирующий описываемые родственные отношения.

2. Составить программу, которая описывает родственные отношения.

3. В качестве фактов описать унарные отношения: мужчина, женщина; и бинарные: состоят_в_браке, родитель_ребенок. Записать правила, которые определяют следующие отношения: муж, жена, родитель, ребенок, сын, дочь, брат, сестра, дядя, тетя, бабушка, дедушка.

4. Подобрать тестовые данные, проверяющие все полученные отношения. Тестовые данные должны содержать операции «И», «ИЛИ», «НЕ».

5. Выполнить задание по индивидуальному варианту (Приложение А), дополнив граф родственных отношений таким образом, чтобы указанные в варианте виды родственных отношений были проиллюстрированы графом.

6. Использовать приложение PIE (Prolog Inference Engine - Машина Вывода Пролога), который включен в комплект системы программирования Visual Prolog для отладки программы «родственные отношения».

Содержание отчета о выполнении индивидуального задания

1. Постановка задачи.

2. Представление структур данных и структуры программ в виде деревьев И/ИЛИ в соответствии с заданным вариантом.

3. Примеры обработки запросов. Представление доказательства цели в виде дерева И/ИЛИ.

4. Анализ результатов работы программ в режиме трассировки.

5. Выводы по работе.

Контрольные вопросы

1. Что такое «факт» в Прологе? Как записываются факты?
2. Что такое «правила» в Прологе? Для чего они используются?
3. В каких конструкциях пролога используются переменные?
4. Как строятся вопросы в Прологе? Какие типы вопросов используются в Прологе?
5. Что такое «рекурсия» в Прологе? В каких случаях она может быть использована?

ЛАБОРАТОРНАЯ РАБОТА №3. СТРУКТУРА ПРОГРАММЫ НА ЯЗЫКЕ ПРОЛОГ. ИНТЕРФЕЙС СРЕДЫ РАЗРАБОТКИ

Цель работы: Изучить основные режимы работы интерфейса Visual Prolog, структуру программы и ее разделы. Ознакомиться с понятием «рекурсия».

Теоретические сведения.

1. Декларации и определения

В Visual Prolog, перед написанием кода для тела правила мы должны сначала объявить о существовании такого предиката компилятору. Аналогично, перед использованием любых доменов они должны быть объявлены и представлены компилятору. Причиной такой необходимости в предупреждениях является попытка как можно раньше обнаружить возможность исключений периода исполнения.

Под *"исключениями периода исполнения (runtime exceptions)"* мы понимаем события, возникающие только во время исполнения программы. Например, если Вы вознамерились использовать целое число в качестве аргумента функтора, а вместо этого вы по ошибке использовали вещественное число, то в процессе исполнения возникла бы ошибка *периода исполнения* и программа в этом случае завершилась бы неуспешно. Когда Вы объявляете предикаты или домены, которые определены, то появляется своего рода позиционная грамматика (какому домену принадлежит какой аргумент), доступная компилятору. Все это автоматически ведет к тому, что компилятор **должен** получать точные инструкции по поводу предикатов и доменов в виде соответствующих объявлений, которые должны предшествовать определениям.

2. Объектная ориентированность

Visual Prolog является полностью объектно-ориентированным языком. Весь код, который разрабатывается для программы, помещается в класс. Это происходит само собой, даже если Вы не интересуетесь объектными свойствами языка. Кроме того, мы можем использовать общедоступные предикаты, находящиеся в других классах.

3. Ключевые слова

Программа на Visual Prolog, представляемая кодом, разделяется ключевыми словами на **секции** разного вида путем использования ключевых слов, предписывающих компилятору, какой код генерировать. К примеру, есть ключевые слова, обозначающие различие между **декларациями** и **определениями** предикатов и доменов. Обычно, каждой секции предшествует ключевое слово. Ключевых слов, обозначающих окончание **секции**, нет. Наличие другого ключевого слова обозначает окончание предыдущей **секции** и начало другой. Исключением из этого правила являются ключевые слова **implement** и **end implement**. Код, содержащийся между этими ключевыми словами, есть код, который относится к конкретному классу. Основными ключевыми словами являются следующие: **implement** и **end**

implement Среди всех ключевых слов, это единственные ключевые слова, используемые парно. Visual Prolog рассматривает код, помещенный между этими ключевыми словами, как код, принадлежащий одному классу. За ключевым словом **implement** обязательно должно следовать имя класса.

Open.- Это ключевое слово используется для расширения **области видимости** класса. Оно должно быть помещено вначале кода класса, сразу после ключевого слова **implement** (с именем класса и, возможно именем интерфейса).

Constants.- Это ключевое слово используется для обозначения секции кода, которое определяет неоднократно используемые значения, применяемые в коде. Например, если строковый литерал "PDC Prolog" предполагается использовать в различных местах кода, тогда можно единожды определить мнемоническое (краткое, легко запоминаемое слово) для использовании в таких местах:

```
constants
pdc="PDC Prolog".
```

Определение константы завершается точкой (.). В отличие от переменных Пролога константы должны начинаться со строчной буквы (нижний регистр).

Domains. Это ключевое слово используется для обозначения секции, объявляющей домены, которые будут использованы в коде. Синтаксис таких объявлений позволяет порождать множество вариантов объявлений доменов, используемых в тексте программы. Вообще говоря, объявляется функтор, который будет использоваться в качестве домена и ряд доменов, которые используются в качестве его аргументов.

class facts. Это ключевое слово представляет секцию, в которой объявляются факты, которые будут в дальнейшем использоваться в тексте программы. Каждый факт объ-является как имя, используемое для обозначения факта, и набор аргументов, каждый из которых должен соответствовать либо стандартному (предопределенному), либо объявленному домену.

class predicates. Эта секция содержит объявления предикатов, которые определяются в тексте программы в разделе clauses. Объявление предиката - это имя, которое присваивается предикату, и набор аргументов, каждый из которых должен соответствовать либо стандартному (предопределенному), либо объявленному домену.

Clauses. Этот раздел содержит конкретные определения объявленных в разделе **class predicates** предикатов, причем синтаксически им соответствующие.

Goal. Этот раздел определяет главную точку входа в программу на языке системы Visual Prolog. Будучи компилятором, система отвечает за генерацию эффективного исполняемого кода написанной программы. Этот код исполняется не в то же время, когда компилятор порождает код. Поэтому компилятору надо знать заранее точно, с какого предиката начнется исполнение программы. Благодаря этому, когда программа вызывается на исполнение, она начинает работу с правильной начальной точки. Для того, чтобы это стало возможным, введен специальный раздел, обозначенный ключевым словом Goal. Его можно представлять как особый предикат, не имеющий аргументов. Это предикат - именно тот предикат, с которого вся программа начинает исполняться.

4. Составные домены

Составные домены создаются с использованием встроенных и других составных или списковых доменов. В системе Visual Prolog имеются следующие основные встроенные домены:

- integer – целые
- real - вещественные
- string - строковые
- symbol –символьные
- char - знаковые
- sring8 - строковые 8-разрядные
- pointer - указатели
- binary - двоичные (бинарные)

- boolean – булевские.

Составные домены позволяют рассматривать наборы данных как единое.

Рассмотрим, например, дату "October 15, 2009. Она состоит из трех единиц информации – месяца, дня и года – но было бы полезно рассматривать дату как единое целое в виде древовидной структуры:

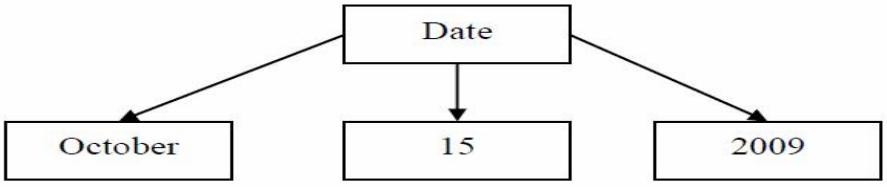


Рисунок 3. Составной домен «дата».

Это можно сделать путем объявления домена date_cmp, содержащего данные date:
domains date_cmp = date(string Month, unsigned Day, unsigned Year).
и затем использовать простую запись, то есть:
D = date(“October”, 15, 2009).
Это выглядит как факт Пролога, но это не так – это всего лишь значение, которое можно обрабатывать почти также, как строки или числа. Такая структура начинается с имени, обычно называемым функтор (functor) (в данном случае - date), за которым следуют три аргумента. Функтор в Visual Prolog не имеет ничего общего с функциями в языках программирования. Функтор не вызывает никаких вычислений, это всего лишь имя, которое идентифицирует составную величину и объединяет свои аргументы. Аргументы составной величины могут быть, в свою очередь, составными. К примеру, Вы можете думать о чьем-либо дне рождения как о структуре данных:

birthday(person("Per", "Schultze"), date("Apr",14,1960)).

В этом примере видны две подчасти составного значения birthday: аргумент person("Per", "Schultze") и аргумент date("Apr", 14, 1960). Функторами этих величин являются person и date.

6. Создание консольного проекта в среде Visual Prolog

Для создания консольного приложения в среде Visual Prolog нужно выбрать пункт меню **Project->New…**. В появившемся диалоговом окне следует задать имя проекта. Имя проекта используется, кроме того, как имя исполняемого файла, создаваемого системой в каталоге ...\Exe основного каталога проекта. В поле «UI Strategy» (User Interface Strategy – стратегия создания пользовательского интерфейса) надо выбрать «Console» для создания консольного приложения. Для сохраненного проекта появляются **Проектное окно (Project Window)**, которое содержит информацию о составе проекта и **Окно сообщений (Messages Window)**, которое будет содержать сообщения о состоянии и процессах, протекающих в среде.

Прежде, чем вносить какие-то изменения, проект необходимо «построить» (build), то есть откомпилировать и связать (compile and link). В меню Build (Постро-ить) содержатся команды для построения, компиляции и вызова исполняемого ре-зультата проекта. Проектное **Дерево (Project Tree)** в **Проектном окне (Project Window)** ото-бражает структуру файлов текущего проекта. Visual Prolog использует следующие соглашения для именований файлов:

- .ph файлы - заголовки пакетов (package headers). Пакет является набором классов и интерфейсов, которые используются совместно.
- .pack файлы - packages. Они содержат исполняемые разделы или кон-кретизации файлов, перечисленных в соответствующих .ph файлах.
- .i файл содержит интерфейс (interface).
- .cl файл содержит декларацию класса (class declaration).

- .pro файл содержит имплементацию класса (class implementation).

7. Средства ввода/вывода в Visual Prolog

Для работы с потокам ввода/вывода в Visual Prolog может быть использован стандартный класс StdIO. Основные методы класса:

- read : () -> _ Term. Считывает из входного потока терм.
- readChar : () -> char Char. Считывает из входного потока переменную типа Char.
- readString : (charCount NumberOfChars) -> string String. Читает из входного потока количество символов, равное NumberOfChars и записывает в переменную типа String.
- nl : (). Записывает в поток вывода символ перевода на новую строку.
- write : (...). Выводит в выходной поток свои аргументы.
- writef : (string Format [formatString], ...). Выводит в выходной поток аргументы, форматируя их в соответствии со строкой формата formatString.

Для того, чтобы считывать из входного потока переменную, имеющую тип составного домена, применяется метод hasDomain() вместе с методом stdIO:read().

Пример 1. Чтение и вывод символа

```
read_char1() :- Char_data = stdIO::readChar(), stdIO::writef("Char is %: ", Char_data).
```

Пример 2. Чтение и вывод даты

Файл mail.cl – декларации классов:

```
class main
open core
predicates
classInfo : core::classInfo.
predicates
 run : core::runnable.
 end class main
```

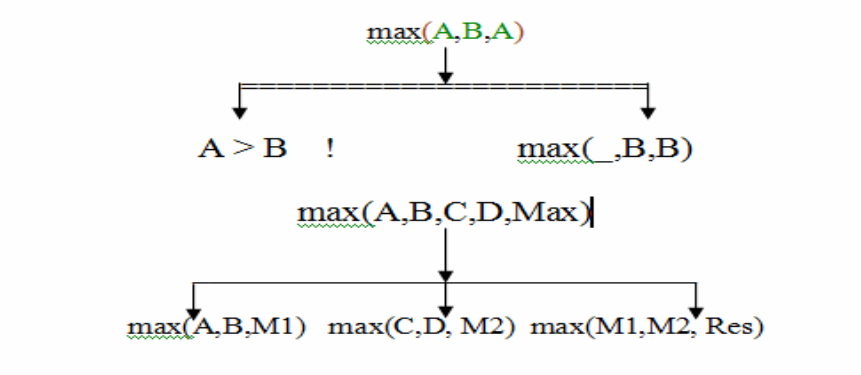
Файл mail.pro – основной текст исполняемого модуля:

```
implement main
open core
constants className = "main".
classVersion = "".
clauses classInfo(className, classVersion).
domains date_cmp = date(string Month, unsigned Day, unsigned Year).
class predicates
date_input: ().
clauses  date_input() :- stdIO::write("Input the first date"),
 stdIO::nl, hasDomain(date_cmp, Date1),
 Date1 = stdIO::read(), stdIO::writef("The 1st date is %", Date1),
 stdIO::nl, stdIO::write("Input the second date"),
 stdIO::nl, hasDomain(date_cmp, Date2),
 console::clearInput(), Date2 = stdIO::read(),
 stdIO::writef("The 2nd date is %", Date2),
 stdIO::nl.

 clauses run():- console::init(), date_input(), fail.
 run():- stdIO::write("End of test...").
end implement main
goal
 mainExe::run(main::run).
```

Пример выполнения лабораторной работы. Программа нахождения максимального из четырёх чисел.

Представление программы в виде дерева



Описание структуры предикатов

max : (integer A,integer B,integer C, integer D, integer Res) procedure (i,i,i,i,o).
A,B,C,D – исходные числа, целые;
Res – результат, максимальное из четырёх.
max : (integer A, integer B, integer Res) procedure(i,i,o).
A,B – исходные числа, целые;
Res – максимальное из двух.

Текст программы

```
        % Файл main.pro
implement main
    open core

    class predicates
        max : (integer A,integer B,integer C, integer D, integer Res) procedure (i,i,i,i,o).
        max : (integer A, integer B, integer Res) procedure(i,i,o).
    clauses
        classinfo("L2", "4.0").

   max(A,B,C,D,Res):- max(A,B,M1),max(C,D,M2),max(M1,M2,Res).
   max(A,B,A):- A>B,!.
   max(_,B,B).

        run():- console::init(),stdio::write("Введите 4 числа"), stdio::nl,
          max(stdio::read(),stdio::read(),stdio::read(),stdio::read(), Max), stdio::write("Max= ",Max), stdio::nl.
end implement main

goal
    mainExe::run(main::run).
```

Результат выполнения программs представлен на рисунке 2. В программе поиска максимального значения из четырёх чисел задействуются функции ввода/вывода стандартной библиотеки stdio. После ввода четырёх цифр программа выдаст максимальное из них.

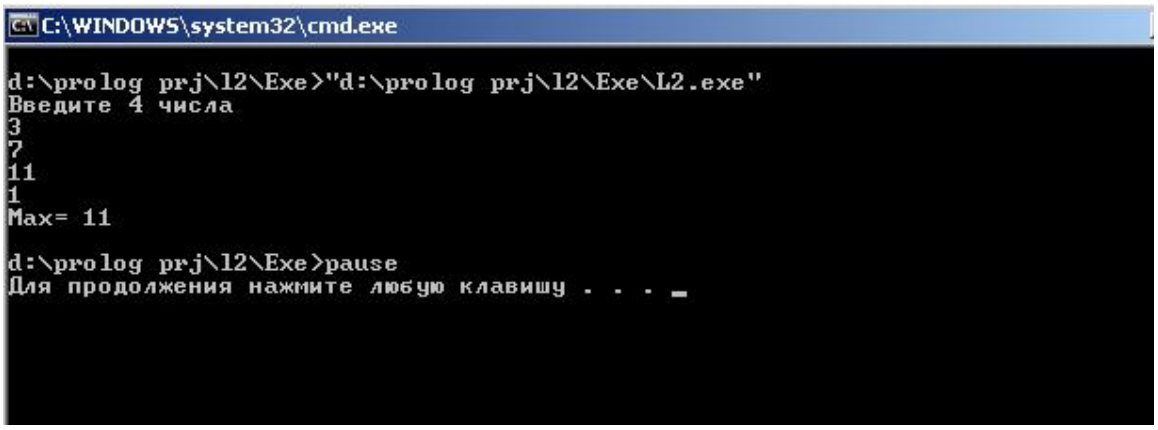


Рисунок 2 – Результат выполнения программы поиска максимального числа.

Индивидуальное задание и методика выполнения работы

В ходе выполнения работы требуется:

1. Выполнить задание согласно индивидуальному варианту **I=N(MOD 11)+1.** (Приложение Б). Для выполнения работы воспользоваться средой Visual Prolog, создав консольный проект. Использовать в проекте операции ввода/вывода.
2. Оформить задание лабораторной работы №1 в виде проекта Visual Prolog.

Содержание отчета о выполнении индивидуального задания

1. Постановка задачи.
2. Представление структур данных и структуры программ в виде деревьев И/ИЛИ в соответствии с заданным вариантом.
3. Примеры обработки запросов. Представление доказательства цели в виде дерева И/ИЛИ.
4. Анализ результатов работы программ в режиме трассировки.
5. Выводы по работе.

Контрольные вопросы

1. Какие основные ключевые слова используются в Пролог-программах? Каково их назначение?
2. Из каких основных секций состоит Пролог-программа?
3. Что такое «составные домены» Пролога? Каково их назначение, как они объявляются в Прологе?

ЛАБОРАТОРНАЯ РАБОТА №4. РЕКУРСИВНАЯ ОБРАБОТКА ДАННЫХ

Цель работы: Изучить основные режимы работы интерфейса Visual Prolog, структуру программы и ее разделы. Ознакомиться с понятием «рекурсия».

Теоретические сведения.

1. Методы организации рекурсии

Правило, содержащее само себя в качестве компоненты, называется правилом рекурсии. Правила рекурсии так же как правила повтора реализуют повторное вы-полнение задач. Они весьма эффективны, например, при формировании запросов к базе данных, а также при обработке таких доменных структур, как списки.

Пример 1. Правило рекурсии:

```
class predicates
 write_string : ().
clauses
 write_string() :- stdIO::write("hello"), stdIO::nl, write_string().
```

Это правило состоит из трех компонент. Первые две выдают строку «hello» и переводят курсор на начало следующей строки экрана. Третья - это само правило. Так как оно содержит само себя, то чтобы быть успешным, правило write_string должно удовлетворять само себе. Это приводит снова к вызову операции выдачи на экран строки и смещение курсора на начало новой строки экрана. Процесс продолжается бесконечно и в результате строка выдается на экран бесконечное число раз. Избежать возникновения бесконечной рекурсии можно. Для этого следует ввести предикат завершения, содержащий условие выхода. Формулировка условия выхода на русском языке для правила write_string может иметь вид: "Продолжать печать строки до тех пор, пока счетчик печати не превысит число 7. После чего остановить процесс". Определение условий выхода и включение их в правило ре-курсии является очень важным элементом программирования на Прологе.

Программа из примера 2 демонстрирует простое правило рекурсии, в кото-рое включено условие выхода.

Пример 2. Программа циклически считывает символ, введенный пользователем: если этот символ не #, то он выдается на экран, если этот символ - #, то программа за-вершается. Правило рекурсии имеет вид:

```
read_a_character() :-
 Char_data = stdIO::readchar(),
  Char_data <> '#',
   stdIO::write(Char_data),
   read_a_character().
```

Первая компонента правила есть встроенный предикат Пролога, обеспечивающий считывание символа. Значение этого символа присваивается переменной Char_data. Следующее подправило проверяет, является ли символ символом #. Если нет, то подправило успешно, символ выдается на экран и рекурсивно вызывается read_a_character. Этот процесс продолжается до тех пор, пока внутренняя проверка не обнаружит недопустимый символ #. В этот момент обработка останавливается, и программа завершается.

2. Метод обобщенного правила рекурсии

Обобщенное правило рекурсии содержит в теле правила само себя. Рекурсия будет конечной, если в правило включено условие выхода, гарантирующее окончание его работы. Тело правила состоит из утверждений и правил, определяю-щих задачи, которые должны быть выполнены. Ниже в символическом виде дан общий вид правила рекурсии:

```
<имя правила рекурсии> :-
 <список предикатов>, (1)
 <предикат условия выхода>, (2)
<список предикатов>, (3)
<имя правила рекурсии>, (4)
<список предикатов>. (5)
```

Хотя структура этого правила сложнее чем структура простого правила рекурсии, рассмотренного в предыдущем разделе, однако принципы, применяемые к первому из них применимы и ко второму. Данное правило рекурсии имеет пять компонент. Первая – это группа предикатов. Успех или неудача любого из них на рекурсию не влияет. Следующая компонента - предикат условия выхода. Успех или неудача этого предиката либо позволяет продолжить рекурсию, либо вызывает ее остановку. Третья компонента - список других предикатов. Аналогично, успех или неудача этих предикатов на рекурсию не оказывает влияния. Четвертая группа – само рекурсивное правило. Успех этого правила вызывает рекурсию. Пятая группа - список предикатов, успех или неудача которых не влияет на рекурсию. Пятая группа также получает значения (если они имеются), помещенные в стек во время выполнения рекурсии. 19

Правила, построенные указанным образом, являются обобщенными вилами рекурсиии (ОПР), а метод называется ОПР-методом. Например, вы хотите написать правило

генерации всех целых чисел, начиная с 1 и кончая 7. Пусть имя правила будет write_number(Number).

Для этого примера первая компонента структуры общего правила рекурсии не используется. Второй компонентой, т.е. предикатом выхода, является Number < 8. Когда значение Number равно 8, правило будет успешным и программа завершится. Третья компонента правила оперирует с числами. В этой части правила число вы-дается на экран и затем увеличивается на 1. Для увеличенного числа будет использоваться новая переменная Next_Number. Четветая компонента - вызов самого правила рекурсии write_number(Next_number). Пятая компонента, представленная в общем случае, здесь не используется. Программа генерации ряда чисел (Пример 3) использует следующее правило рекурсии:

Пример 3.

```
write_number(8).
write_number(Number) :-
 Number < 8, stdIO::write(Number),
 stdIO::nl,
 Next_Number = Number + 1,
 write_number(Next_number).
```

Программа начинается с попытки вычислить подцель write_number(1). Сначала программа сопоставляет подцель с первым правилом write_number(8). Так как 1 не равно 8, то сопоставление неуспешно. Программа вновь пытается сопоставить подцель, но уже с головой правила write_number(Number). На этот раз сопоставление успешно вследствие того, что переменной Number присвоено значение 1. Программа сравнивает это значение с 8; это условие выхода. Так как 1 меньше 8, то подправило успешно. Следующий предикат выдает значение, присвоенное Number. Переменная Next_Number получает значение 2, а значение Number увели-чивается 1. Важное свойство правила рекурсии состоит в его расширяемости. На-пример, оно может быть расширено для подсчета суммы ряда целых чисел.

Пример 4. – Сумма ряда. Вычислить S(7) = 1 + 2 + 3 + 4 + 5 + 6 + 7 = 28 или S(7) = 7 + 6 + 5 + 4 + 3 + 2 + 1 = 28 Правило рекурсии программы выполняет вычисления по обычной схеме сложения:

```
1 Начальное значение
+ 2 Следующее значение

___
3 Частичная сумма
+ 3
Следующее значение

___
6 Частичная сумма
...
```

Правило рекурсии имеет вид:

```
sum_series(1,1).  /* сумма ряда */
sum_series(Number,Sum) :-
 Number > 0,
 Next_number = Number - 1,
 sum_series(Next_number, Partial_Sum),
 Sum = Number + Partial_Sum.
```

Данное правило имеет четыре компоненты и одно дополнительное нерекур-сивное правило. Заметим, что последняя компонента правила рекурсии - это правило Sum с Partial_Sum (частичная сумма) в качестве переменной. Это правило не может быть выполнено до тех пор, пока Partial_Sum не получит некоторого значения.

При описании рекурсивных правил следует соблюдать осторожность во избежание зацикливания рекурсии. Для этого любое рекурсивное определение отношения должно включать, по крайней мере, два правила:

- тривиальные, или «граничные» случаи;
- «общие» случаи, в которых решение получается из решений для (более простых) вариантов самой исходной задачи.

Рассмотрим еще несколько примеров:

Пример 5. Вычисление факториала

```
factorial(X, _) :– X<0, ! ,fail.                    % если X<0 то стоп, ошибка.
Factorial (0, 1) :– !.                              % 0!=1
factorial(N, Fact) :–
N1=N–1, factorial(N1, Fact1), Fact=N*Fact1.         % общий случай
```

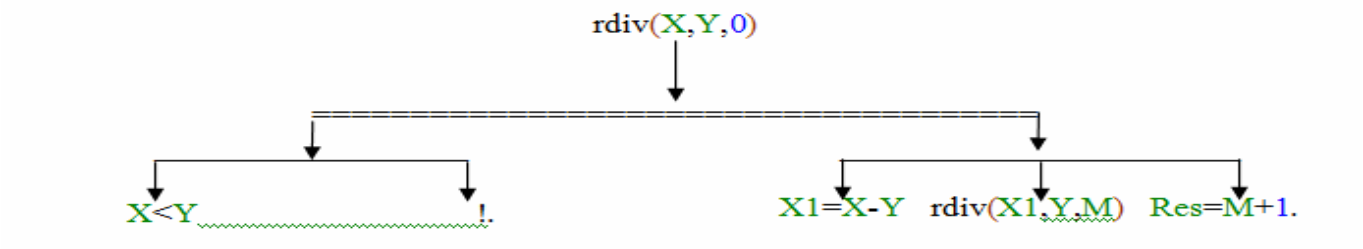
Пример 6.Определение чисел Фибоначчи:

```
 f(1,1) :– !.                % Первое число есть 1
f(2,1) :– !.                 % Второе число есть 2
f(I,R) :– I>2, I1=I–1, I2=I–2, f(I1,M), f(I2,N), R=N+M.      % общий случай
```

Пример выполнения лабораторной работы

Постановка задачи: Разработать рекурсивную программу вычисления частного двух выражений.

Представление программы в виде дерева



Текст программы

```
implement main
    open core
constants
    className = "main".
    classVersion = "3.4".

class predicates

    rdiv : (integer Dividend, integer Divider, integer Result)procedure(i,i,o).
clauses
    classInfo(className, classVersion).

    rdiv(X,Y,0):-X<Y,!.
    rdiv(X,Y,Res):-X1=X-Y,rdiv(X1,Y,M),Res=M+1.
    clauses
(X),
        succeed(). % place your own code here
    run():-
        console::init(),stdio::write("Напишите делимое и делитель"),
        rdiv(stdio::read(),stdio::read(),X),stdio::write
end implement main
```

```
goal
mainExe::run(main::run).
```

Результат выполнения программы

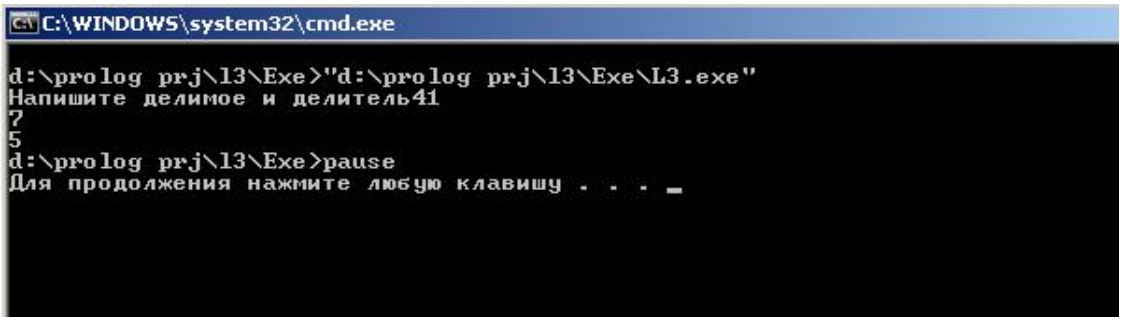


Рисунок 1 – Результат выполнения программы

Индивидуальное задание и методика выполнения работы

В ходе выполнения работы требуется:

1. Выполнить задание согласно индивидуальному варианту (Приложение В). Для выполнения работы воспользоваться средой Visual Prolog, создав консольный проект.

Содержание отчета о выполнении индивидуального задания

1. Постановка задачи.
2. Представление структур данных и структуры программ в виде деревьев И/ИЛИ в соответствии с заданным вариантом.
3. Примеры обработки запросов. Представление доказательства цели в виде дерева И/ИЛИ.
4. Анализ результатов работы программ в режиме трассировки.
5. Выводы по работе.

Контрольные вопросы

1. Какие основные ключевые слова используются в Пролог-программах? Каково их назначение?
2. Из каких основных секций состоит Пролог-программа?
3. Что такое «составные домены» Пролога? Каково их назначение, как они объявляются в Прологе?
4. В чем состоит метод обобщенного правила рекурсии? Какие элементы должны присутствовать обязательно?

ЛАБОРАТОРНАЯ РАБОТА №5. ОБРАБОТКА СПИСКОВ

Цель работы: Ознакомиться с реализацией структуры данных типа список в языке Пролог и методами их обработки.

Теоретические сведения.

1. Основные положения

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

Пролог позволяет выполнять со списком целый ряд операций, например:

- доступ к объектам списка,
- проверка элемента на принадлежность к списку,
- разделение списка на два подсписка,
- слияние двух списков,
- сортировку элементов списка в порядке возрастания или убывания и другие.

Списки бывают полезны при создании баз знаний (баз данных), экспертных систем, словарей; перечень областей применения можно продолжать еще долго. В настоящей работе рассматриваются структура, организация и представление списков, демонстрируются некоторые из методов, применяемых при программировании на Прологе.

Список является набором объектов одного и того же доменного типа. Объектами списка могут быть целые числа, действительные числа, символы, символьные строки и структуры. Порядок расположения элементов является отличительной чертой списка; те же самые элементы, упорядоченные иным способом, представляют уже совсем другой список. Порядок играет важную роль в процессе сопоставления.

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

Совокупность элементов списка заключается в квадратные скобки ([]), а друг от друга элементы отделяются запятыми.

Примерами списков могут служить:

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

[3.2,4.6,1.1,2.64,100.2]

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

Элементами первого списка являются целые числа. Элементами второго - действительные числа, третьего - символьные строки, т. е. конкретные значения символов. Любой печатный символ кода ASCII пригоден для списков этого типа. (Более подробно о коде ASCII будет написано в гл. 6).

2. Атрибуты списка

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

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

["JOHN WALKER",3.50,45.50]

некорректен, ввиду того, что составлен из элементов разных типов. Списки структур являются исключением из правила.

Количество элементов в списке называется его длиной.

Длина списка ["MADONNA","AND","CHILD"] равна 3.Длина списка

[4.50,3.50,6.25,2.9,100.15] равна 5. Список может содержать всего один элемент и даже не содержать элементов вовсе:

["summer"]

[ ]

Список, не содержащий элементов, называется пустым списком.

Непустой список можно рассматривать как состоящий из двух частей:

(1) первый элемент списка - его голова, и

(2) остальная часть списка - хвост.

Голова является элементом списка, хвост есть список сам по себе. Голова - это отдельное неделимое значение. Наоборот, хвост представляет из себя список, составленный из того, что осталось от исходного списка в результате "отсеченния головы". Этот новый список зачастую можно делить и дальше. Если список состоит из одного элемента, то его можно разделить на голову, которой будет этот самый единственный элемент, и хвост, являющийся пустым списком.

В списке

[4.50,3.50,6.25,2.9,100.15]

например, головой является значение 4.50, а хвостом - список

[3.50,6.25,2.9,100.15]

Этот список в свою очередь имеет и голову, и хвост. Голова - это значение 3.50, хвост - список

[6.25,2.9,100.15]

В табл. 1. показаны головы и хвосты нескольких списков.

Таблица 1. Головы и хвосты различных списков

Список	Голова	Хвост
[1,2,3,4,5]	1	[2,3,4,5]
[6.9,4.3,8.4,1.2]	6.9	[4.3,8.4,1.2]
[cat,dog,horse]	cat	[dog,horse]
['S','K','Y']	'S'	['K','Y']
["PIG"]	"PIG"	[]
[]	не определена	неопределен

Отличительной особенностью описания списка является наличие звездочки (*) после имени домена элементов. Так запись

bird_name *

указывает на то, что это домен списка, элементами которого являются bird_name, т. е. запись bird_name* следует понимать как список, состоящий из элементов домена bird_name.

Описание в разделе domains, следовательно, может выглядеть либо как

bird_list = bird_name *

bird_name = symbol

либо как

bird_list = symbol *

Домен bird_list является доменом списка элементов типа symbol (списка птиц).

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

birds(bird_list)

Как видим, описание предиката списка ни в чем не отличается от описания обычного предиката.

Сам список присутствует в разделе утверждений clauses:

```
birds(["sparrow","robin","mockingbird","thunderbird",
       "bald eagle"]).
```

Чуть позднее будет описано, как использовать списки в разделе программы goal.

Законченный пример использования списков приведен в программе "Списки" (Пример.1), в которую дополнительно включены список из 7 целых чисел (домен number_list) и предикат score.

Пример. 1

```
        /* Работа со списками.              */
  domains
    bird_list = bird_name *
                bird_name = symbol
    number_list = number *
                  number = integer
  predicates
    birds(bird_list)
    score(number_list)
  clauses
    birds(["sparrow",
           "robin",
           "mockingbird",
           "thunderbird",
           "bald eagle"]).
    score([56,87,63,89,91,62,85]).     /*          конец программы          */
```

Эта программа создавалась в расчете на следующие внешние запросы:

```
birds(All).
birds([_,_,_,B,_]).
birds([B1,B2,_,_,_]).
score(All).
score([F,S,T,_,_,_,_]).
```

Читателю предлагается самостоятельно обработать приведенные запросы.

Пример 2. Деление списков

При работе со списками достаточно часто требуется разделить список на несколько частей. Это бывает необходимо, когда для целей текущей обработки нужна лишь определенная часть исходного списка, а оставшуюся часть следует обработать позднее. Сейчас Вы увидите, что деление списков на части является достаточно простой операцией.

Для пояснения сказанного рассмотрим предикат split, аргументами которого являются элемент данных и три списка:

split(Middle,L,L1,L2).

Элемент Middle здесь является компаратором, L - это исходный список, а L1 и L2 - подсписки, получающиеся в результате деления списка L. Если элемент исходного списка меньше или равен Middle, то он помещается в список L1; если больше, то в список L2.

Предположим, что вначале значением переменной Middle является число 40, переменной L присвоен список [30,50,20,25,65,95], а переменные L1 и L2 не инициализированы.

split(40,[30,50,20,25,65,95],L1,L2).

Правило для разделения списка должно быть написано таким образом, чтобы элементы исходного списка, меньшие либо равные 40, помещались в список L1, а большие 40 - в список L2. Правило устроено следующим образом: очередной элемент извлекается из списка при помощи метода разделения списка на голову и хвост, а потом сравнивается с компаратором Middle. Если значение этого элемента меньше или равно значению компаратора, то элемент помещается в список L1, в противном случае - в список L2. На рис. 5.7 приведены состояния объектов Middle, L, L1, L2 на различных этапах работы правила.

В результате применения правила к списку [30,50,20,25,65,95] значениями списков L1 и L2 станут соответственно [30,20,25] и [50,65,95].

Само правило для разделения списка записывается в Прологе следующим образом:

```
split(_,[],[],[]).
split(Middle,[Head|Tail],[Head|L1],L2) :-
        Head <= Middle,
        split(Middle,Tail,L1,L2).
split(Middle,[Head|Tail],L1,[Head|L2]) :-
        split(Middle,Tail,L1,L2),
        Head > Middle.
```

Отметим, что метод деления списка на голову и хвост используется в данном правиле как для разделения исходного списка, так и для формирования выходных списков.

Попробуйте ввести такое целевое утверждение:

split(40,[30,50,20,25,65,95],L1,L2).

Результат работы программы с данной целью Вы можете получить на компьютере.

Пример выполнения лабораторной работы

Выполнение контрольных примеров

Программа 1:

```
implement main
    open core

constants
    className = "main".
    classVersion = "".

domains
```

```
    bird_list = bird_name*.
    bird_name = string.
    number_list = integer*.
    nnumber = integer.

class predicates
        birds : (bird_list) determ anyflow.
        score : (number_list) determ .

clauses
  classInfo(className, classVersion).

        birds(["sparrow",
          "robin",
          "mockingbird",
          "thunderbird",
          "bald eagle"]).
        score([56,87,63,89,91,62,85]).

   run():-console::init(),
   birds(B),stdio::write(B),!;B=[],stdio::write(B).
end implement main

goal
   mainExe::run(main::run).
```

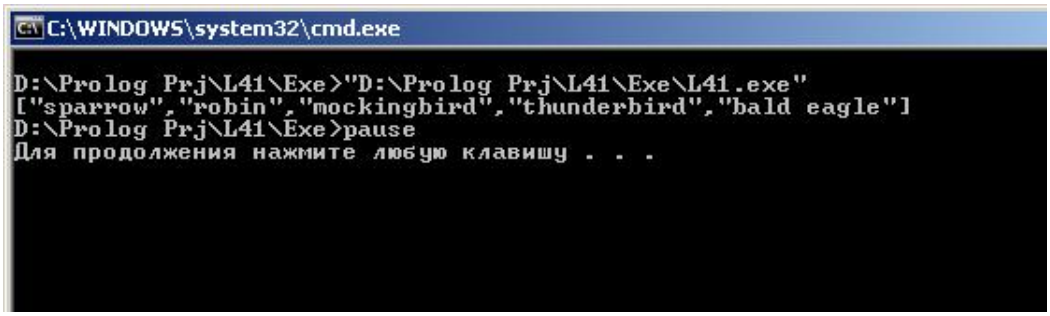


Рисунок 1 – Результат выполнения контрольного примера №1

Программа 2.

```
implement main
   open core

constants
   className = "main".
   classVersion = "".

domains
        middle = integer.
        list = integer*.

class predicates
       split : (middle,list,list,list) nondeterm anyflow.

clauses
classInfo(className, classVersion).
```

```
        split(_,[],[],[]).
        split(Middle,[Head|Tail],[Head|L1],L2) :-
                              Head <= Middle,
                             split(Middle,Tail,L1,L2).
         split(Middle,[Head|Tail],L1,[Head|L2]) :-
                             split(Middle,Tail,L1,L2),
                             Head > Middle.

run():-console::init(),
split(40,[30,50,20,25,65,95],L1,L2),stdio::write(L1,"\n",L2),!;succeed().
end implement main

goal
   mainExe::run(main::run).
```

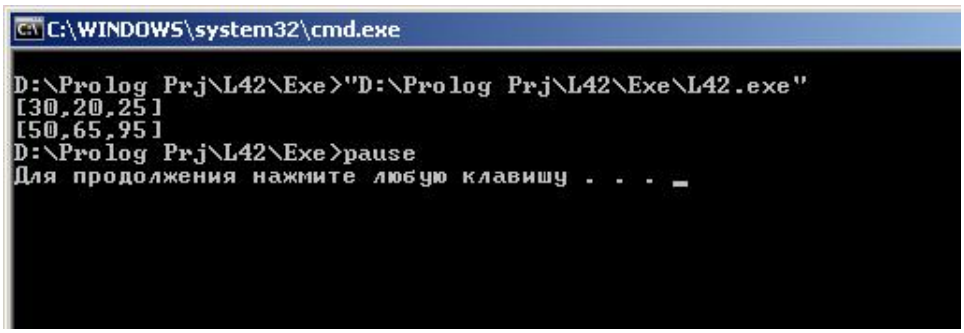
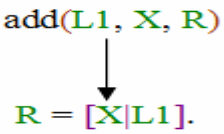


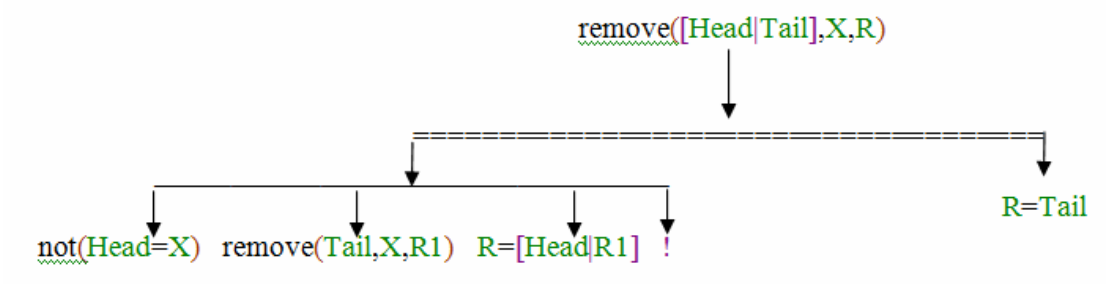
Рисунок 2 – Результат выполнения контрольного примера №2.

Набор функций для обработки списков.
Представление программ в виде дерева

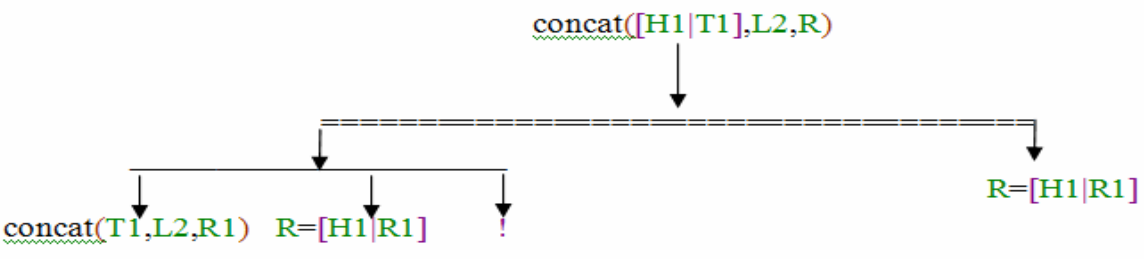
Функция добавления элемента в начало списка



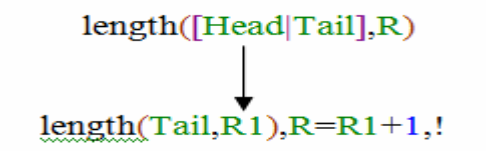
Функция удаления элемента из списка



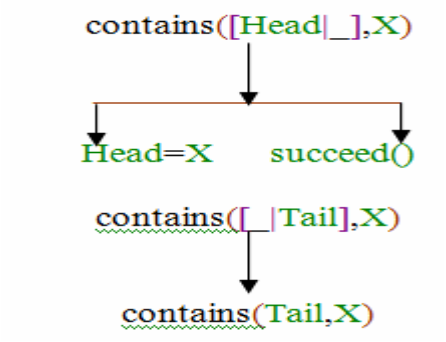
Функция конкатенации списков



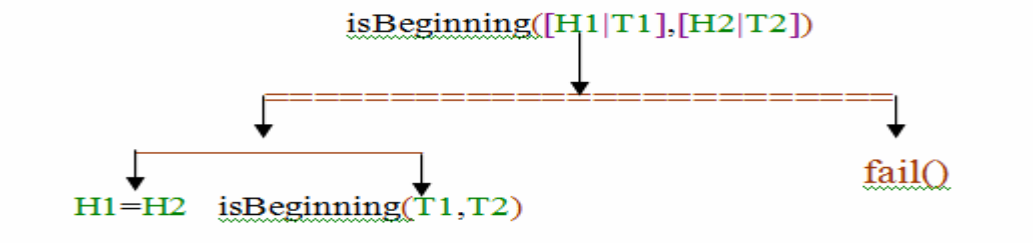
Функция поиска длины списка



Функция определения принадлежности элемента X списку



Функция «является ли началом»



Текст программы

```
implement main
    open core
constants
    className = "main".
    classVersion = "".
domains
     list = integer*.
class predicates
    add : (list, integer, list) procedure(i,i,o).
    remove : (list, integer, list) procedure(i,i,o).
    concat : (list,list,list) procedure(i,i,o).
    length :(list,integer) procedure(i,o).
    contains :(list, integer) determ.
    isBeginning :(list,list) determ.
clauses
    classInfo(className, classVersion).

   add(L1, X, R):- R = [X|L1].

   remove([],_,[]):-!
   remove([Head|Tail],X,R):-not(Head=X),remove(Tail,X,R1),R=[Head|R1],!.
   remove([_|Tail],_,R):-R=Tail.

   concat(_,_,[]):-!
```

```
    concat([H1|[]],L2,R):-R = [H1|L2],!.
  concat([H1|T1],L2,R):- concat(T1,L2,R1), R=[H1|R1],!.

  length([],0):-!
 length([Head|Tail],R):-length(Tail,R1),R=R1+1,!.

  contains([],_):-fail().
  contains([Head|_],X):-Head=X, succeed(),!.
  contains([_|Tail],X):-contains(Tail,X).

  isBeginning([H1|T1],[H2|T2]):-
H1=H2,isBeginning(T1,T2);fail(). % второй является началом первого списка
  isBeginning([H1|_],[]):- succeed().

clauses
   run():-
      console::init(),
      stdio::write("% add check\n"),
      add([5,10,13,11],12,R),stdio::write(R), stdio::nl(),
      add([],12,S),stdio::write(S), stdio::nl(),
      stdio::write("% remove check\n"),
      remove([5,10,13,11],14,A),stdio::write(A), stdio::nl(),
      remove([5,10,13,11],13,B),stdio::write(B), stdio::nl(),
      remove([5,10,13,11],5,C),stdio::write(C), stdio::nl(),
      stdio::write("% concat check\n"),
      concat([10,14],[1,3,11],E),stdio::write(E), stdio::nl(),
      concat([10,14],[],F),stdio::write(F), stdio::nl(),
      stdio::write("% length check\n"),
      length([10,11,12,13,14,15,16],L),stdio::write(L), stdio::nl(),
      length([],L2),stdio::write(L2), stdio::nl(),
      length([11,12,13,14,15,16],L1),stdio::write(L1), stdio::nl(),
      stdio::write("% contains check\n"),(
      contains([10,11,12,13,14,15],16),stdio::write("Contains\n"),!;stdio::write("Not contains\n"),
      succeed()),(
      stdio::write("% isBeginning check\n"),
      isBeginning([1,3,5,7,9,12],[1,3,5,9]),stdio::write("is beginning\n"),!;stdio::write("not a beginnin
g\n")   , succeed()),  succeed().
end implement main
goal
   mainExe::run(main::run).
```

Результат выполнения программы

Исходные списки выглядят следующим образом:
- для функции contains: [10,11,12,13,14,15], ищется элемент 16
- для функции isBeginning: [1,3,5,7,9,12],[1,3,5,9]

Индивидуальное задание и методика выполнения работы

В ходе выполнения работы требуется:

1. Выполнить контрольные примеры. Реализовать набор функций для обработки списков:

- Добавление элемента X к списку List. Выдать новый список.
- Удаление элемента X из списка List. Выдать список без элемента X.

- Конкатенация списков.
- Определение длины списка.
- Определение принадлежности элемента X списку List.

2. Создать программу на языке Пролог, которая решает задачи согласно варианту (исходные данные выбираются из приложения Г по формуле **I=N(MOD 25)+1, J=(I+12)(MOD 25)+1.)**

3. Составить отчет о проделанной работе.

Содержание отчета о выполнении индивидуального задания

1. Постановка задачи.
2. Структура программы в виде дерева И/ИЛИ.
3. Описание структур данных.
4. Анализ результатов работы программ в режиме трассировки.
5. Схема доказательства цели в виде дерева И/ИЛИ.
6. Выводы по работе.

Контрольные вопросы

1. Управление процессом выполнения программы. Предикаты: fail, cut, not.
2. Правила использования предиката **not** в ПРОЛОГе.
3. Вопросы эффективности разработки программ в ПРОЛОГе.
4. Понятие составных обьектов. Функторы в ПРОЛОГе.
5. Понятие хвостовой рекурсии. Оптимизация хвостовой рекурсии.
6. Механизм логического вывода в ПРОЛОГе.
7. Обработка списков. Основные процедуры в ПРОЛОГе.

ЛАБОРАТОРНАЯ РАБОТА №6. РАБОТА С ДИНАМИЧЕСКИМИ БАЗАМИ ДАННЫХ

Цель работы: изучить основные приемы реализации динамических баз данных в языке Пролог

Теоретические сведения.

Реляционная модель данных предполагает, что база данных – это есть описание некоторого множества отношений. Пролог–программу можно рассматривать как такую базу данных, здесь отношения между данными представлены в виде фактов и правил. В Прологе есть специальные средства для модифицирования этой базы данных, то есть добавлять и удалять новые отношения из файла. Для этих целей служат встроенные предикаты.

Чтобы понять, как в Прологе реализуется обращение к БД, рассмотрим запрос на примере БД игроков футбольной команды:

```
dplayer("Bernie Kosar",Team,Pos).
```

В этом утверждении Team и Pos есть переменные, значения которых нужно найти. Когда этот запрос (цель) испытывается, процедуры Пролога просматривают утверждения БД игроков (см.ниже) на предмет сопоставления с утверждением, содержащим Bernie Kosar. Так как в базе данных такое утверждение присутствует, то переменной Team присваивается значение Cleveland Browns, а переменной Pos - QB.

Если трактовать dplayer как предикат БД Пролог, то отсюда следует с необходимостью такое его описание

```
database
    dplayer(name,team,position)
```

Раздел **database** в Прологе предназначен для описания предикатов базы данных, таких как dplayer. Все различные утверждения этого предиката составляют динамическую базу данных Пролога. База данных называется динамической, так во время работы программы из нее можно удалять любые содержащиеся в ней утверждения, а также добавлять новые. В этом состоит ее отличие от "статических" баз данных, где утверждения являются частью кода программы и не могут быть изменены во время счета. Другая важная особенность динамической базы данных состоит в том, что такая база может быть записана на диск, а также считана с диска в оперативную память.

Иногда бывает предпочтительно иметь часть информации базы данных в виде утверждений статической БД; эти данные заносятся в динамическую БД сразу после активизации программы. Для этой цели используются предикаты **asserta и assertz,** которые будут рассмотрены ниже. В общем, предикаты статической БД имеют другое имя, но ту же самую форму представления данных, что и предикаты динамической. Предикат статической БД, соответствующий предикату dplayer динамической базы данных, есть

```
predicates
    player(name,team,position)
clauses
    player("Dan Marino","Miami Dolphins","QB").
    player("Richart Dent","Chicago Bears","DE").
    player("Bernie Kosar","Cleveland Browns","QB").
    player("Doug Cosbie","Dallas Cowboy","TE").
    player("Mark Malone","Pittsburgh Steelers","QB").
```

Заметим, что все отличие предиката dplayer по сравнению с player заключается лишь в одной лишней букве терма. Добавление латинской буквы d - обычный способ различать предикаты динамической и статической баз данных. Правилом для занесения в динамическую БД информации из утверждений предиката player служит

```
assert_database :-
  player(Name,Team,Number),
  assertz( dplayer(Name,Team,Number) ),
  fail.
assert_database :- !.
```

В этом правиле применяется уже знакомый вам метод отката после неудачи, который позволяет перебрать все утверждения предиката player.

Таким образом, можно указать следующее соответствие понятий:

база данных Пролога – реляционная база данных,

предикат БД – отношение,

объект – атрибут,

отдельное утверждение – элемент отношения,

количество утверждений – мощность.

Пример использования динамической базы данных в Прологе.

При создании динамической базы данных Пролога будут очень полезны многие уже известные вам вещи. Так, здесь применим почти весь материал, касающийся предикатов и утверждений Пролога. Можно создавать правила для работы с информацией БД, можно также использовать введенные предикаты, осуществляющие обращение к файлам, файлам БД, записанным на диск.

В Прологе имеются и специальные встроенные предикаты для работы с динамической базой данных. Таковыми являются **asserta, assertz, retract, save, consult, и findall.**

Предикаты **asserta, assertz и retract** позволяют занести факт в заданное место динамической БД и удалить из нее уже имеющийся факт.

Предикат **asserta** заносит новый факт в базу данных, располагающуюся в оперативной памяти компьютера (резидентная БД). Новый факт помещается перед всеми уже внесенными утверждениями данного предиката. Этот предикат имеет такой синтаксис:

asserta(Clause).

Таким образом, чтобы поместить в БД утверждение

dplayer("Bernie Kosar","Cleveland Browns","QB").

перед уже имеющимся там утверждением

dplayer("Doug Cosbie","Dallas Cowboy","TE"),

стоящим в настоящий момент в базе данных на первом месте, необходимо следующее предикатное выражение:

asserta(dplayer("Bernie Kosar","Cleveland Browns","QB")).

Теперь БД содержит два утверждения, причем утверждение со сведениями о Kosar предшествует утверждению со сведениями о Cosbie:

dplayer("Bernie Kosar","Cleveland Browns","QB").

dplayer("Doug Cosbie","Dallas Cowboy","TE").

Крайне важно отдавать себе отчет в том, что в динамической базе данных могут содержаться только факты (не правила).

Предикат **assertz** так же, как и **asserta**, заносит новые утверждения в базу данных. Однако он помещает новое утверждение за всеми уже имеющимися в базе утверждениями того же предиката. Синтаксис предиката столь же прост:

assertz(Clause).

Для добаления к двум уже имеющимся в БД утверждениям третьего

dplayer("Mark Malone","Pittsburgh Steelers","QB")

потребуется следующее предикатное выражение:

assertz(dplayer("Mark Malone","Pittsburgh Steelers","QB")).

после чего БД будет содержать третьего утверждения

dplayer("Bernie Kosar","Cleveland Browns","QB").

dplayer("Doug Cosbie","Dallas Cowboy","TE").

dplayer("Mark Malone","Pittsburgh Steelers","QB").

Третье, новое, только что занесенное утверждение, следует за двумя старыми.

Предикат **retract** удаляет утверждение из динамической БД (еще раз напомним, что динамическая БД содержит факты, но не правила.) Его синтаксис таков:

retract(Existing_clause).

Предположим, что вы хотите удалить из базы данных второе утверждение. Для этого необходимо написать выражение

retract(dplayer("Doug Cosbie","Dallas Cowboy","TE")).

и БД будет состоять уже только из двух утверждений:

dplayer("Bernie Kosar","Cleveland Browns","QB").

dplayer("Mark Malone","Pittsburgh Steelers","QB").

Так же, как **asserta и assertz, retract** применим только в отношении фактов.

Предикаты **save и consult** применяются для записи динамической БД в файл на диск и для загрузки содержимого файла в динамическую БД.

Предикат **save** сохраняет находящуюся в оперативной памяти базу данных в текстовом файле. Синтаксис этого предиката

save(DOS_file_name),

где DOS_file_name есть произвольное допустимое в MS DOS или PC DOS имя файла.

Для того, чтобы сохранить содержимое футбольной БД в файле с именем football.dba, требуется предикат

save("football.dba").

В результате все утверждения находящейся в оперативной памяти динамической БД будут записаны в файл football.dba.

Файл БД может быть считан в память (загружен) при помощи предиката **consult**, систаксис которого таков:

consult(DOS_file_name).

Для загрузки файла футбольной БД требуется выражение

consult("football.dba").

Предикат **consult** неуспешен, если файл с указанным именем отсутствует на диске, или если этот файл содержит ошибки, как, например, в случае несоответствия синтаксиса предиката из файла описаниям из раздела программы **database**, или если содержимое файла невозможно разместить в памяти ввиду отсутствия места.

Предикат **findall** позволяет собрать все имеющиеся в базе данные в список, который может быть полезен при дальнейшей работе. Так, findall можно использовать для получения списка имен всех игроков.

После того, как успешным будет предикат

findall(Name,dplayer(Name,_,_),Name_list)

переменная Name_list содержит список имен всех игроков.

Пример программы добавления, вывода и удаления из базы данных фактов и значений.

```
Add :– write('Введите имя мужчины: '),
      read_string(S), assertz(man(S)).

Out :– write('Список имен мужчин:\n'), man(S),write(S),nl.

Del :– write('Введите имя мужчины: '),
      read_string(S), retractall(man(S)).
```

Main :– consult(‘db.pro’),add,out,del,out.

Файл db.pro первоначально содержит следующие факты:

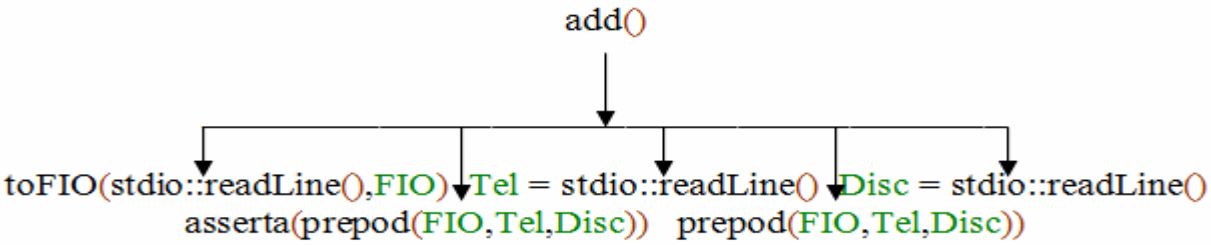
man(oleg).
woman(zina).
man(alexandr).
man(petr).

Пример выполнения лабораторной работы

Допустим, в базе данных необходимо хранить следующие данные: ФИО преподавателя, его контактный номер телефона и название преподаваемой дисциплины. Базе данных можно задавать любые вопросы (запросы), механизмы пролога обеспечат обработку любых вопросов. Добавим операции добавления и удаления факта, а так же вывода состояния базы данных в терминал.

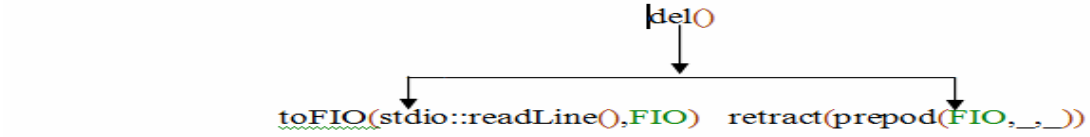
Фрагмент программы добавления факта в базу данных:

```
add():- stdio::write("Добавляем новую запись\n"),
   stdio::write("Введите фамилию преподавателя: "), toFIO(stdio::readLine(),FIO),
   stdio::write("Введите номер телефона: "), Tel = stdio::readLine(),
   stdio::write("Введите преподаваемую дисциплину: "), Disc = stdio::readLine(),
    asserta(prepod(FIO,Tel,Disc)),
    stdio::write("Добавлен преподаватель: ",prepod(FIO,Tel,Disc)), stdio::nl(),
succeed(),!;succeed().
```



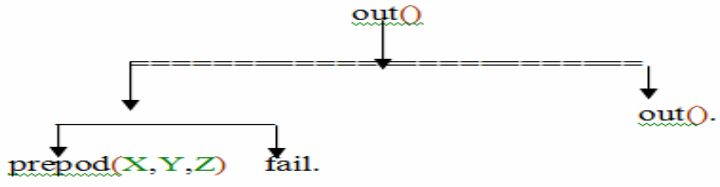
Операция удаления факта из базы данных:

```
del():- stdio::write("Удаляем запись\n"),
          stdio::write("Введите фамилию преподавателя: "),
          toFIO(stdio::readLine(),FIO),
          retract(prepod(FIO,_,_)),
          stdio::write("Удален преподаватель: ",prepod(FIO,Tel,Disc)), stdio::nl(),
  succeed(),!;succeed().
```



Операция вывода базы данных:

```
out():- prepod(X,Y,Z), stdio::write(X,"\n",Y,"\n",Z,"\n\n"),fail.
out().
```



Текст программы

```
        implement main
    open core

constants
    className = "main".
    classVersion = "".
domains
    fio = fio(string,string,string).
 %   prepod = prepod(fio,string,string).
class facts
    prepod : (fio,string,string).
class predicates
  %  prepod : (fio FIO, string Tel, string Discp) determ.
    add : () procedure.
    out : ()procedure.
    del : ()procedure.
    toFIO : (string,fio) determ (i,o).
    fio : (string*, fio)determ (i,o).
clauses
    classInfo(className, classVersion).
       toFIO(In, FIO):- L=string::split(In," "), fio(L,FIO).
    fio([X|[Y|[Z|[]]]], FIO):- FIO=fio(X,Y,Z).

    prepod(fio("Ярков","Игнат","Федорович"),"0-100-500","Компьютерная  логика").
    prepod(fio("Селезнёв","Михаил","Вадимович"),"1-311-404","Компьютерная электроника").
    prepod(fio("Брюлов","Дмитрий","Алексеевич"),"3-333-13","Основы  проектирования ЭВМ").

    add():- stdio::write("Добавляем новую запись\n"),
            stdio::write("Введите фамилию преподавателя: "), toFIO(stdio::readLine(),FIO),
            stdio::write("Введите номер телефона: "), Tel = stdio::readLine(),
            stdio::write("Введите преподаваемую дисциплину: "), Disc = stdio::readLine(),
             asserta(prepod(FIO,Tel,Disc)),
             stdio::write("Добавлен преподаватель: ",prepod(FIO,Tel,Disc)), stdio::nl(),
      succeed(),!;succeed().

    out():- prepod(X,Y,Z), stdio::write(X,"\n",Y,"\n",Z,"\n\n"),fail.
    out().

    del():- stdio::write("Удаляем запись\n"),
            stdio::write("Введите фамилию преподавателя: "), toFIO(stdio::readLine(),FIO),
        %     stdio::write("Введите номер телефона: "), Tel = stdio::readLine(),
        %     stdio::write("Введите преподаваемую дисциплину: "), Disc = stdio::readLine(),
            retract  (prepod(FIO,_,_)),
            stdio::write("Удален преподаватель: ",prepod(FIO,Tel,Disc)), stdio::nl(),
    succeed(),!;succeed().

    run():-console::init(),       add(),out(),       del(),out(),       succeed().
end implement main
goal
    mainExe::run(main::run).
```

Индивидуальное задание и методика выполнения работы

В ходе выполнения работы требуется:

Составить программу, реализующую следующие операции с базой данных:

1. Добавление записей
2. Удаление конкретной записи по заданному значению.
3. Поиск заданной записи.
4. Вывод данных из базы данных.

Выполнить вариант для заданной предметной области: I=N(MOD 10)+1.

1.Отдел кадров	6.Авиакасса.
2.Библиотека	7.Магазин.
3.Отдел договоров.	8.Склад.
4.Деканат.	9.Автотранспортное предприятие.
5.Кафедра.	10. Фирма по продаже компьютеров.

Содержание отчета о выполнении индивидуального задания

1. Постановка задачи.
2. Представление структур данных и структуры программ в виде деревьев И/ИЛИ в соответствии с заданным вариантом.
3. Примеры обработки запросов. Представление доказательства цели в виде дерева И/ИЛИ.
4. Анализ результатов работы программ в режиме трассировки.
5. Выводы по работе.

Контрольные вопросы

1 . Назовите способы представления баз данных в ПРОЛОГ.

2 . Какие есть типы отношений между термами в ПРОЛОГ.

3 . В чем отличие структуры программы для работы с базой данных.

4 . Назовите основные предикаты при работе с базами данных.

5 .Какие типовые операции над базами данных Вам известны.

ЛАБОРАТОРНАЯ РАБОТА №7. ТИПОВЫЕ ОПЕРАЦИИ С БИНАРНЫМИ ДЕРЕВЬЯМИ

Цель работы: научиться описывать рекурсивные типы данных, выработать навыки представления структур в виде дерева и реализации типовых процедур с деревьями.

Теоретические сведения.

Типы данных являются рекурсивными, если они допускают, чтобы в его структуре данных была рекурсия.

Примером рекурсивного типа данных является бинарное дерево. Бинарное дерево либо пусто, либо состоит из следующих трех частей:

- корень;
- левое поддерево;
- правое поддерево.

Где левое и правое поддеревья сами являются бинарными деревьями. Таким образом, рекурсивность бинарного дерева заложена уже в его определении.

Бинарное дерево упорядочено, если в нем все вершины левого поддерева меньше корня, все вершины правого поддерева больше корня, и оба поддерева также упорядочены. Такое дерево называется бинарным справочником. Преимущество упорядочивания состоит в том, что для поиска некоторого объекта в бинарном справочнике достаточно просмотреть не более одного поддерева.

Стоит заметить, что поиск в двоичном справочнике наиболее эффективен для сбалансированного дерева, то есть дерева, в котором два поддерева содержат примерно равное количество элементов. Если же дерево полностью разбалансировано, то поиск в нем так же неэффективен, как и поиск в списке. Логические программы, работающие с бинарными деревьями, подобны программам, работающим со списками. Как и в случаях натуральных чисел и списков, мы начнем с определения типа бинарного дерева. Оно дается программой 5.1. Отметим, что в программе имеется двойная рекурсия, 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 и может быть замечен в остальных программах данного раздела.

Давайте напишем некоторые программы обработки деревьев. Наш первый пример состоит в проверке, появляется ли некоторый элемент в дереве. Реляционная схема – **tree_member(Element,Tree).** Отношение выполнено, если элемент Element является одной из вершин дерева Tree. В программе 5.2 приведено определение. Декларативное понимание программы: "X - элемент дерева, если X находится в вершине (ввиду факта) или X - элемент левого или правого поддерева (ввиду двух рекурсивных правил)".

Две ветви бинарного дерева различны, но во многих приложениях это различие несущественно. Следовательно, возникает полезное понятие изоморфизма, которое определяет, являются ли два неупорядоченных дерева по существу одинаковыми. Два бинарных дерева T1 и T2 изоморфны, если T2 может быть получено из T1 изменением порядка ветвей в поддеревьях.

Изоморфизм является отношением эквивалентности с простым рекурсивным определением. Два пустых дерева изоморфны. В случае непустых деревьев, два дерева изоморфны, если элементы в вершинах дерева 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.

Программа 5.1. Определение бинарного дерева

```
% binary_tree(Tree) Tree - бинарное дерево.

binary_tree(empty).
binary_tree(tree(Element,Left,Right)):–
binary_tree(Left), binary_tree(Right).
```

Программа 5.2. Проверка принадлежности дереву

```
% tree_member(Element,Tree) Element является элементом бинарного дерева Tree.
tree_member(X,tree(X,Left,Right)).
tree_member(X,tree(Y,Left,Right));– tree_member(X,Left).
tree_member(X,tree(Y,Left,Right));– tree_member(X,Right).
```

Программа 5.3. Определение изоморфизма деревьев

```
% isotree(Tree1,Tree2) бинарные деревья Tree1 и Tree2 изоморфны.
isotree(empty, empty).
isotree(tree(X,Left1,Right1),tree(X,Left2,Right2)):–
    isotree(Left1,Left2), isotree(Right1,Right2).
    isotree(tree(X,Left1,Right1),tree(X,Left2,Right2)):–
    isotree(Left1,Right2), isotree(Right1,Left2).
```

Программа 5.3 определяет предикат isotree(Tree1, Tree2), который истинен, если дерево Tree1 изоморфно дереву Tree2. Аргументы входят в предикат симметрично.

Программа 5.4. Подстановка терма в дерево

```
%substitute(X,Y,TreeX,TreeY) бинарное дерево TreeY - результат замены всех
                                вхождений X в бинарном дереве TreeX на Y.
substitute(X,Y,empty,empty).
substitute(X,Y,tree(X,Left,Right),tree(Y,Left1,Right1)):–
  substitute(X,Y,Left,Left1),
  substitute(X,Y,Right,Right1).
substitute(X,Y,tree(Z,Left,Right),tree(Z,Left1,Right1)):–
  X=Z,
  substitute(X,Y,Left,Left1),
  substitute(X,Y,Right,Right1).
```

Программы, относящиеся к бинарным деревьям, используют двойную рекурсию, по одной на каждую ветвь дерева. Двойная рекурсия может проявляться двумя способами. В программе могут рассматриваться два отдельных случая, как в программе 5.2 для отношения tree_member. В отличие от этого, программа, проверяющая принадлежность элемента списку, содержит лишь один рекурсивный случай. При другом способе, в теле рекурсивного правила содержатся два рекурсивных вызова, как в каждом рекурсивном правиле для isotree в программе 5.3.

Ранее, одно из заданий состояло в том, чтобы написать программу подстановки элементов в списки. Аналогичная программа может быть написана для подстановки элементов в бинарные деревья. Предикат **substitute(X, Y, OldTree, NewTree)** выполнен, если дерево **NewTree** получается из дерева **OldTree** заменой всех вхождений элемента X на Y. Аксиоматизация отношения substitute/4 приведена в программе 5.4.

Во многих приложениях, использующих деревья, требуется доступ к элементам, указанным в вершинах. Основной является идея обхода дерева в предписанном порядке. Имеется три возможности линейного упорядочения при обходе:

– сверху-вниз, когда сначала идет значение в вершине, далее вершины левого поддерева, затем вершины правого поддерева;

– слева-направо, когда сначала приводятся вершины левого поддерева, затем вершина дерева и далее вершины правого поддерева;

– снизу-вверх, когда значение в вершине приводится после вершин левого и правого поддерева.

Определение каждого из этих трех обходов приведено в программе 5.5.

Программа 5.5. Обходы бинарного дерева

```
%pre_order(Tree,Pre) Pre - обход бинарного дерева Tree  сверху вниз.
pre_order(tree(X,L,R),Xs):–
   pre_order(L,Ls), pre_order(R,Rs), append([X|Ls],Rs,Xs).
pre_order(empty,[]).
% in_order(Tree,In) In - обход бинарного дерева Tree слева направо.
in_order(tree(X,L,R),Xs):–
   in_order(L,Ls), in_order(R,Rs), append(Ls,[X|Rs],Xs).
in_order(empty,[]).
% post_order(Tree,Post) Post - обход бинарного дерева Tree снизу вверх.
post_order(tree(X,L,R),Xs):–
   post_order(L,Ls),
   post_order(R,Rs),
   append(Rs,[X],Rs1),
   append(Ls,Rs1,Xs). post_order(empty,[]).
```

Рекурсивная структура определений одинакова, единственное отличие состоит в порядке элементов, полученных применением целей вида append.

Опишем предикаты для создания и модификации бинарного дерева. Бинарное дерево задается с помощью функтора

tree(K, LeftT, RightT), где К – элемент, находящийся в вершине;

LeftT и RightT – левое и правое поддерево соответственно.

create_tree(A, tree(A, empty, empty)). % создание дерева

insert_left(X, tree(A, _, B), tree(A, X, B)). % включение элемента данных А, как левого поддерева В

insert_right(X, tree(A, B, _), tree(A, B, X)). % включение элемента данных А, как правого поддерева В

append – это процедура **append(LeftList, RightList, ListRes),** где ListRes является результатом слияния списков LeftList, RightList.

Фрагмент программы печатает все элементы, проходя его «в глубину».

```
show_tree(nil).
show_tree(tree(X,Left,Right)):-write(X),show_tree(Left), show_tree(Right).
```

Пример выполнения лабораторной работы

Текст программы 1:

```
binary_tree(empty).
binary_tree(tree(Element,Left,Right)):- binary_tree(Left), binary_tree(Right).

goal :- binary_tree(tree(10,tree(50,empty,empty),tree(100,tree(70,empty,empty),empty))).
```

Трассировка программы 1:

```
Trace: >> CALL:   goal()
Trace: >> CALL:   binary_tree(tree(10,tree(50,"empty","empty"),tree(100,tree(70,"empty","empty"),
"empty")))
Trace: >> CALL:   binary_tree(tree(50,"empty","empty"))
Trace: >> CALL:   binary_tree("empty")
Trace: >> RETURN: binary_tree("empty")
Trace: >> CALL:   binary_tree("empty")
Trace: >> RETURN: binary_tree("empty")
Trace: >> RETURN: binary_tree(tree(50,"empty","empty"))
Trace: >> CALL:   binary_tree(tree(100,tree(70,"empty","empty"),"empty"))
Trace: >> CALL:   binary_tree(tree(70,"empty","empty"))
Trace: >> CALL:   binary_tree("empty")
Trace: >> RETURN: binary_tree("empty")
Trace: >> CALL:   binary_tree("empty")
Trace: >> RETURN: binary_tree("empty")
Trace: >> RETURN: binary_tree(tree(70,"empty","empty"))
Trace: >> CALL:   binary_tree("empty")
Trace: >> RETURN: binary_tree("empty")
Trace: >> RETURN: binary_tree(tree(100,tree(70,"empty","empty"),"empty"))
Trace: >> RETURN: binary_tree(tree(10,tree(50,"empty","empty"),tree(100,tree(70,"empty","empty
"),"empty")))
Trace: >> RETURN: goal()
True
1 Solution
```

Текст программы 2:

```
tree_member(X,tree(Y,Left,Right)):- X=Y,!.
tree_member(X,tree(Y,Left,Right)):- tree_member(X,Left).
tree_member(X,tree(Y,Left,Right)):- tree_member(X,Right).

goal :- tree_member(50,tree(10,tree(50,empty,empty),tree(100,tree(70,empty,empty),empty))).
```

Трассировка программы 2:

```
goal
Trace: >> CALL:   goal()
Trace: >> CALL:   tree_member(50,tree(10,tree(50,"empty","empty"),tree(100,tree(70,"empty","emp
ty"),"empty")))
Trace: >> CALL:   50 = 10
```

```
   Trace: >> FAIL:   50 = 10
Trace: >> CALL:   tree_member(50,tree(50,"empty","empty"))
Trace: >> CALL:   50 = 50
Trace: >> RETURN: 50 = 50
Trace: >> RETURN: tree_member(50,tree(50,"empty","empty"))
Trace: >> RETURN: tree_member(50,tree(10,tree(50,"empty","empty"),tree(100,tree(70,"empty","e
mpty"),"empty")))
Trace: >> RETURN: goal()
True
1 Solution
```

Текст программы 3:

```
isotree(empty, empty).
isotree(tree(X,Left1,Right1),tree(X,Left2,Right2)):- isotree(Left1,Left2), isotree(Right1,Right2).
isotree(tree(X,Left1,Right1),tree(X,Left2,Right2)):- isotree(Left1,Right2), isotree(Right1,Left2).

goal :- isotree(tree(10,tree(100,empty,empty),empty),tree(10,empty,tree(100,empty,empty))).
```

Текст программы 4:

```
isotree(empty, empty).
isotree(tree(X,Left1,Right1),tree(X,Left2,Right2)):- isotree(Left1,Left2), isotree(Right1,Right2).
isotree(tree(X,Left1,Right1),tree(X,Left2,Right2)):- isotree(Left1,Right2), isotree(Right1,Left2).

goal :- isotree(tree(10,tree(101,empty,empty),empty),tree(10,empty,tree(100,empty,empty))).
```

Текст программы 5:

```
substitute(X,Y,empty,empty).
substitute(X,Y,tree(X,Left,Right),tree(Z,Left1,Right1)):-Z=Y,
                                        substitute(X,Y,Left,Left1),
                                        substitute(X,Y,Right,Right1).
substitute(X,Y,tree(Z,Left,Right),tree(C,Left1,Right1)):-
                                        substitute(X,Y,Left,Left1),
                                        substitute(X,Y,Right,Right1).

goal :- substitute(10,100,tree(12,tree(10,empty,empty),empty),X),write(X),nl().
```

Трассировка программы 5:

```
Trace is On
goal
Trace: >> CALL:   goal()
Trace: >> CALL:   substitute(10,100,tree(12,tree(10,"empty","empty"),"empty"),X$49)
Trace: >> CALL:   substitute(10,100,tree(10,"empty","empty"),LEFT1$50)
Trace: >> CALL:   Z$51 = 100
Trace: >> RETURN: 100 = 100
Trace: >> CALL:   substitute(10,100,"empty",LEFT1$52)
Trace: >> RETURN: substitute(10,100,"empty","empty")
Trace: >> CALL:   substitute(10,100,"empty",RIGHT1$53)
```

```
Trace: >> RETURN: substitute(10,100,"empty","empty")
Trace: >> RETURN: substitute(10,100,tree(10,"empty","empty"),tree(100,"empty","empty"))
Trace: >> CALL:   substitute(10,100,"empty",RIGHT1$54)
Trace: >> RETURN: substitute(10,100,"empty","empty")
Trace: >> RETURN: substitute(10,100,tree(12,tree(10,"empty","empty"),"empty"),tree(C$55,tree(10
0,"empty","empty"),"empty"))
Trace: >> CALL:   write(tree(C$55,tree(100,"empty","empty"),"empty"))
tree(C$55,tree(100,"empty","empty"),"empty")Trace: >> RETURN: write(tree(C$55,tree(100,"empt
y","empty"),"empty"))
Trace: >> CALL:   nl()

Trace: >> RETURN: nl()
Trace: >> RETURN: goal()
True
1 Solution
Trace is OFF
goal
tree(C$56,tree(100,"empty","empty"),"empty")
True
1 Solution
```

Программа индивидуального варианта:

```
%pre_order(Tree,Pre) Pre - обход бинарного дерева Tree  сверху вниз.
pre_order(tree(X,L,R),Xs):-
pre_order(L,Ls), pre_order(R,Rs), append([X|Ls],Rs,Xs).
pre_order(empty,[]).
% in_order(Tree,In) In - обход бинарного дерева Tree слева направо.
in_order(tree(X,L,R),Xs):-
in_order(L,Ls), in_order(R,Rs), append(Ls,[X|Rs],Xs).
in_order(empty,[]).
% post_order(Tree,Post) Post - обход бинарного дерева Tree снизу вверх.
post_order(tree(X,L,R),Xs):-
post_order(L,Ls),
post_order(R,Rs),
append(Rs,[X|[]],Rs1),
append(Ls,Rs1,Xs).
post_order(empty,[]).

append([],L2,R):- R=L2,!.
append([H1|[]],L2,R):-R = [H1|L2],!.
append([H1|T1],L2,R):-append(T1,L2,R1), R=[H1|R1],!.
append(_,_,[]).

goal :- post_order(tree(12,tree(10,empty,empty),tree(13,empty,empty)),X),write(X),nl().
```

Результаты выполнения программы

Результат обхода дерева «сверху-вниз»: [10,12,13]
Результат обхода дерева «слева-направо»: [12,10,13]
Результат обхода дерева «справа-налево»: [10,13,12]

```
    subtree(empty,empty).
subtree(tree(X,L,R), tree(Y,LY,RY)):- subtree(L,tree(Y,LY,RY));subtree(R,tree(Y,LY,RY)).
subtree(tree(X,L1,R1),tree(X,L2,R2)):- subtree(L1,L2),subtree(R1,R2).

goal :- subtree(tree(12,tree(13,empty,empty),empty),tree(13,empty,empty)).
```

```
goal
Trace: >> CALL:   goal()
Trace: >> CALL:   subtree(tree(12,tree(13,"empty","empty"),"empty"),tree(13,"empty","empty"))
Trace: >> CALL:   subtree(tree(13,"empty","empty"),tree(13,"empty","empty"))
Trace: >> CALL:   subtree("empty",tree(13,"empty","empty"))
Trace: >> FAIL:   subtree("empty",tree(13,"empty","empty"))
Trace: >> CALL:   subtree("empty",tree(13,"empty","empty"))
Trace: >> FAIL:   subtree("empty",tree(13,"empty","empty"))
Trace: >> CALL:   subtree("empty","empty")
Trace: >> RETURN: subtree("empty","empty")
Trace: >> CALL:   subtree("empty","empty")
Trace: >> RETURN: subtree("empty","empty")
Trace: >> RETURN: subtree(tree(13,"empty","empty"),tree(13,"empty","empty"))
Trace: >> RETURN: subtree(tree(12,tree(13,"empty","empty"),"empty"),tree(13,"empty","empty"))
Trace: >> RETURN: goal()
True
1 Solution
```

Индивидуальное задание и методика выполнения работы

1. Исследовать на конкретных тестовых данных примеры программ 5.1–5.5.
2. Решить задачи согласно варианту (приложение Д): **I=N(MOD 7)+1.**
3. Подобрать тестовые данные и оформить отчет.

Содержание отчета о выполнении индивидуального задания

1. Постановка задачи.
2. Представление структур данных и структуры программ в виде деревьев И/ИЛИ в соответствии с заданным вариантом.
3. Примеры обработки запросов. Представление доказательства цели в виде дерева И/ИЛИ.
4. Анализ результатов работы программ в режиме трассировки.
5. Выводы по работе.

Контрольные вопросы

1. Дайте определение бинарного дерева.
2. В чем отличия между упорядоченными и сбалансированными деревьями.
3. Что такое двоично-троичные деревья, AVL-деревья.
4. Типовые операции над деревьями.
5. Как представляются графы в ПРОЛОГ, деревья И/ИЛИ.
6. Чем отличается нисходящий от восходящего вывода в ПРОЛОГ.

7 .Назовите основные стратегии решения задач в ПРОЛОГ.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Братко И. Программирование на языке Пролог для искусственного интеллекта. – М.: Мир, 1990.
2. Стерлинг Л., Шапиро Э. Искусство программирования на языке Пролог. – М.: Мир, 1990.
3. Метакидес Г., Нероуд А. Принципы логики и логического программирования. – М.: Факториал, 1998
4. Хоггер К. Введение в логическое программирование. – М.: Мир, 1988.
5. Брюховецкий А.А. Методические указания к лабораторным занятиям для студентов направления «Компьютерная инженерия - 0915» по курсу «Логическое программирование» на тему «Функциональное программирование». / А.А.Брюховецкий. – СевНТУ, 2002.-14с.
6. Малпас Дж. Реляционный язык Пролог и его применение. /Малпас Дж. ; Пер. с англ.; — М.: Наука, 1990. –574с.
7. Ин Ц., Соломон Д. Использование Пролога. – М., 1990.
8. Чень Ч., Ли Р. Математическая логика и автоматическое доказательство теорем. – М.: Наука, 1983.

ПРИЛОЖЕНИЕ А

1. Определить предикат «двоюродный дедушка».
2. Определить предикат «двоюродная бабушка».
3. Определить предикат «двоюродный брат».
4. Определить предикат «двоюродная сестра».
5. Определить предикат «племянник».
6. Определить предикат «племянница».
7. Определить предикат «потомки мужского пола».
8. Определить предикат «потомки женского пола».
9. Определить предикат «предки мужского пола».
10. Определить предикат «предки женского пола».
11. Определить предикат «жена брата».
12. Определить предикат «муж сестры».
13. Определить предикат «муж маминой сестры».
14. Определить предикат «жена папиного брата».
15. Определить предикат «троюродный брат».
16. Определить предикат «троюродная сестра».
17. Определить предикат «муж дочери».
18. Определить предикат «жена сына».
19. Определить предикат «свекровь».
20. Определить предикат «теща».
21. Определить предикат «тесть».
22. Определить предикат «свекор».
23. Определить предикат «жена маминого брата».
24. Определить предикат «муж папиной сестры».

25. Определить предикат «правнук».
26. Определить предикат «правнучка».
27. Определить предикат «внучатый племянник».
28. Определить предикат «внучатая племянница».
29. Определить предикат «прадед».
30. Определить предикат «правнучка».

ПРИЛОЖЕНИЕ Б

1. Заданы три числа *a, b* и *c*. Определить, сколько из них положительных.
2. Заданы четыре числа *a, b, c* и *d*. Определить количество нулевых значений.
3. Задана прямая $y=kx+b$. Для произвольной точки (x, y) определить, как она расположена относительно прямой: а) выше; б) на прямой; в) ниже прямой.
4. Заданы четыре числа *a, b, c* и *d*. Определить максимальное из чисел.
5. Заданы три числа *a, b* и *c*. Определить, являются ли они упорядоченными: а) по возрастанию; б) равны ($a=b=c$); в) по убыванию; г) не упорядочены.
6. Заданы три числа *a, b* и *c*. Определить сумму положительных чисел.
7. Заданы три числа *a, b* и *c*. Определить, сколько среди них отрицательных.
8. Заданы три числа *a, b* и *c*. Определить, сколько из них равны максимальному элементу.
9. Заданы три числа *a, b* и *c*. Определить, сколько из них равны минимальному элементу.
10. Для произвольных *a, b* и *c* найти решение уравнения $ax^*x+bx+c=0$.
11. Для произвольных *a, b* и *c* определить количество действительных корней уравнения $ax^*x+bx+c=0$.
12. Заданы три точки $(x1, y1)$, $(x2, y2)$, $(x3, y3)$. Определить их взаимное расположение: а) все точки совпадают; б) две точки совпадают, а третья отличается; в) все точки различны.
13. Заданы три точки $(x1, y1)$, $(x2, y2)$, $(x3, y3)$. Определить их взаимное расположение: а) точки лежат на одной прямой; б) точки образуют невырожденный треугольник.
14. Заданы система координат и произвольная точка (x, y). Определить, где лежит точка: а) на пересечении осей координат; б) на оси X; в) на оси Y; г) вне осей координат.
15. Заданы система координат и произвольная точка (x, y). Определить, где лежит точка: а) на осях координат; б) в I квадранте; в) во II квадранте; г) в III квадранте; д) в IV квадранте.
16. Заданы три числа *a, b* и *c*. Определить минимальное значение среди положительных.
17. Заданы три числа *a, b* и *c*. Определить максимальное значение среди отрицательных.
18. Заданы три числа *a, b* и *c*. Определить, какой случай имеет место: а) все числа ненулевые и одного знака; б) среди чисел есть нулевые; в) все числа ненулевые и разных знаков.
19. Заданы система координат в пространстве и произвольная точка (x, y, z). Определить, как расположена точка относительно системы координат: а) на оси X; б) на оси Y; в) на оси Z; г) в начале системы координат; д) вне осей координат.
20. Заданы два круга. Один с центром в $(x1, y1)$ и радиусом $r1$, и другой, соответственно $(x2, y2)$ и $r2$. Определить взаимное расположение кругов: а) один внутри другого; б) пересекаются; в) не пересекаются.
21. Заданы прямая $ax+by+c=0$ и две точки $(x1, y1)$, $(x2, y2)$. Определить их взаимное расположение: а) точки по одну сторону прямой; б) точки по разные стороны от прямой; в) одна точка на прямой, а другая нет; г) обе точки на прямой.
22. Задана окружность с центром в точке (x, y) и радиусом r. Определить взаимное расположение окружности и осей координат: а) окружность пересекает обе оси; б) окружность пересекает лишь ось X; в) окружность пересекает лишь ось Y; г) окружность не пересекает оси координат.
23. Задан отрезок своими концами $(x1, y1)$ и $(x2, y2)$. Определить взаимное

положение отрезка и осей координат: а) отрезок параллелен оси X; б) отрезок параллелен оси Y; в) отрезок вырожден; г) отрезок не параллелен осям координат.

24. Заданы прямая *ax+by+c=0* и отрезок своими концами (*x1, y1*) и (*x2, y2*). Определить взаимное расположение отрезка и прямой: а) отрезок пересекает прямую; б) отрезок лежит на прямой; в) отрезок по одну сторону от прямой.

25. Задан отрезок своими концами (*x1, y1*) и (*x2, y2*). Определить взаимное положение отрезка и осей координат: а) отрезок лежит на какой-то оси координат; б) отрезок параллелен какой-то оси; в) отрезок не параллелен осям координат.

26. Заданы круг с центром в (*x0*, *y0*) и радиусом *r0* и треугольник своими вершинами (*x1*, *y1*), (*x2*, *y2*) и (*x3*, *y3*). Определить взаимное расположение треугольника и круга: а) треугольник внутри круга; б) треугольник пересекает круг; в) треугольник вне круга.

27. Задан треугольник своими сторонами *a, b* и *c*. Определить, образуют ли они: а) равносторонний треугольник; б) равнобедренный треугольник; в) не образуют треугольника.

28. Задан треугольник своими сторонами *a, b* и *c*. Определить, образуют ли они: а) прямоугольный треугольник; б) тупоугольный треугольник; в) остроугольный треугольник; г) не образуют треугольник.

29. Заданы прямые *a1*x+b1*x+c=0* и *a2*x+b2*x+c=0*. Определить их взаимное расположение: а) пересекаются; б) параллельны; в) совпадают.

30. Задан треугольник своими вершинами (*x1*, *y1*), (*x2*, *y2*) и (*x3*, *y3*) и точка (*x0*, *y0*). Определить их взаимное расположение: а) точка внутри треугольника; б) точка совпадает с одной из вершин треугольника; в) точка лежит на одной из сторон треугольника; г) точка вне треугольника.

ПРИЛОЖЕНИЕ В

1. Возведение в степень как повторяющееся умножение.
2. Возведение в степень как повторяющееся сложение.
3. Рекурсивное определение остатка от деления (mod).
4. Рекурсивное определение деления нацело (div).
5. Функция Аккермана

Ak(0,N) = N + 1 Ak(M,0) = Ak(M-1,1) Ak(M,N) = Ak(M-1,Ak(M,N-1))

7. Умножение через сложение единицы.
6. Деление через вычитание единицы.
7. Вычислить: Y = k*(k+1)*(k+2)*….*(k+n) .
8. Вычислить: W = (3+ *i)*+(6+i)+(9+i)+….+(m+i).
9. Определить, является ли заданное число числом Фибоначчи.
10. Нахождение чисел Фибоначчи, не превышающих заданное число.
11. Вычисление *n-го* члена арифметической прогрессии, у которой

вый член равен 1, а шаг = 2.

12. Вычисление *n-го* члена геометрической прогрессии, у которой первый член равен 2, а знаменатель равен 4
13. Сумма четных натуральных чисел от 1 до *n*
14. Произведение нечетных натуральных чисел от 1 до *n*..
15. Сумма всех двузначных чисел, кратных трем.
16. Сумма всех трехзначных чисел, не делящихся ни на 5, ни на 7.
17. Вычислить сумму *n* первых членов ряда:
18. 1 + 1/2 + 1/3 + ...

Вычислить сумму n первых членов ряда:

19. 4 - 4/3 + 4/5 - 4/7 + 4/9 - ... + (-1)^(*n*-1) × 4 / *(2*× *n* - 1) + ...

20. Построить рекурсивную функцию для вычисления *n-го* члена последовательности, в которой каждый следующий член равен сумме *n-2* - го и *n-3* -го. Первые 3 члена равны соответственно 1, 2, 3.

1 2 3 3 5 6 8 11

21. Построить рекурсивную функция для вычисления *n-го* члена последовательности, в которой каждый четный член равен сумме двух предыдущих четных, а нечетный равен сумме двух предыдущих нечетных. Первые четыре члена равны соответственно 1, 2, 3, 4.

1 2 3 4 4 6 7 10 11 16 18 ...

22. Построить рекурсивную функцию для вычисления *n*-го члена последовательности, в которой каждый следующий четный член равен произведению двух предыдущих, а каждый следующий нечетный член равен сумме двух предыдущих, а первые 2 члена равны соответственно 1и 2.

1 2 3 6 9 54 63 ...

23. Построить рекурсивную функцию для вычисления *n*-го члена последовательности, в которой каждый следующий член равен произведению двух предыдущих, а первые 2 члена равны соответственно 1 и 2.

1 2 2 4 8 32 ...

24. Построить рекурсивную функцию для вычисления *n*-го члена последовательности, в которой первый член равен 0, второй 1, третий 2, акаждый следующий равен сумме трех предыдущих.

1 2 3 6 11 ....

ПРИЛОЖЕНИЕ Г

1. Два списка U и V имеют одинаковое число элементов. Составить программу, выполняющую правило объединить**(U,V,List),** которое последовательно помещает соответствующие элементы U и V в List. ПРИМЕР: [0,1,2],[3,4,5] => [0,3,1,4,2,5]
2. Задан список L ,содержащий четное число элементов. Построить списки M и N, содержащие элементы, находящиеся на четных и нечетных позициях соответственно.
3. Определить принадлежность элемента X списку L.
4. Определить отношение перевод**(**Список**1,**Список**2)** для перевода списка чисел от 0 до 9 в список соответствующих слов нуль, один и т.д.
5. Определить, что список M является началом списка L.
6. Определите два предиката четнаядлина**(**Список**)** и нечетная длина**(**Список**)** таким образом, чтобы они были истинными, если их аргументом является список четной или нечетной длины соответственно. Например, список [a,b,c,d] имеет четную длину.
7. Определить предикат суммасписка**(**Список**,**Сумма**)** так, чтобы Сумма равнялась сумме чисел, входящих в Список**.**
8. Задать правило для объединения списков: U \/ V -> L.
9. Определить предикат упорядоченный**(**Список**)** так, чтобы он принимал значение истина, если Список упорядочен. Например, список [1,3,7,8,19] – упорядочен.
10. Определить отношение **n**_элемент**(N,**Список**,**X**),** которое выполняется, если X является **N**-ым элементом списка Список.
11. Удалить первый по порядку элемент x, принадлежащий L.

12. Выбрать последний элемент списка L.

13. Подсчитать количество положительных чисел в списке.

14. Удалить повторяющиеся элементы в списке.

15. Вычесть 1-цу от каждого элемента списка.

16. Найти сумму четных элементов списка.

17. Определить длину списка.

18. Найти сумму нечетных элементов списка.

19. Подсчитать число элементов списка больших 10.

20. Выделить из списка все элементы, которые делятся на 2 и на 3;

ПРИЛОЖЕНИЕ Д

1. Определите предикаты

   Двдерево **(** Объект**)** справочник **(** Объект**)**

распознающие, является ли Объект двоичным деревом или двоичным справочником соответственно. Используйте обозначения, введенные в данном разделе.

2. Определите процедуру

   глубина**(** ДвДерево**,** Глубина**)**

вычисляющую глубину двоичного дерева в предположении, что глубина пустого дерева равна 0, а глубина одноэлементного дерева равна 1.

3. Определите отношение

   линеаризация**(** Дерево**,** Список**)**

соответствующее "выстраиванию" всех вершин дерева в список.

4. Определите отношение

   максэлемент**(** Д**,** Элемент**)**

таким образом, чтобы переменная Элемент приняла значение наибольшего из элементов, хранящихся в дереве Д.

5. Составьте программу subtree(S,T), определяющую, является ли S поддеревом T.

6. Определите отношение sum_tree(TreeOfIntegers,Sum), выполненное, если число Sum равно сумме целых чисел, являющихся вершинами дерева TreeOfIntegers.

7. Определите отношение ordered(Tree), выполненное, если дерево Tree является упорядоченным деревом целых чисел, то есть число стоящее в любой вершине дерева больше любого элемента в левом поддереве и меньше любого элемента в правом поддереве.

(Указание: Определите два вспомогательных отношения **ordered_left(X,Tree)** и **ordered_right(X,T),** выполненных если X меньше (больше) корня дерева Tree и дерево Tree упорядочено).

Заказ № ___ от «___» _____ 2015 г. Тираж ___ экз.
Изд-во СевГУ