Антик М.И. Программирование в ПРОЛОГе. 2018

Оглавление

ПРОГРАММИРОВАНИЕ В ПРОЛОГЕ (ПРИМЕРЫ)

1. Элементы грамматики языка Visual Prolog

1.1. Литературный источник: Адаменко А.Н., Кучуков А.М. Логическое программирование и Visual Prolog. 2003 – 992с.

1.2. Структура программы. Программа состоит из предложений, которые могут быть фактами, правилами или запросами. Как правило, программа состоит из четырех секций.

DOMAINS – секция описания доменов (типов). Секция применяется, если в программе используются нестандартные домены.

PREDICATES – секция описания предикатов.

CLAUSES – секция предложений (фраз). Именно в этой секции записываются факты и правила вывода.

GOAL – секция цели. В этой секции записывается запрос.

Системные (стандартные) домены

Числовые домены

integer – целые числа со знаком. В том числе 16-ричные числа - первые два символа 0x (-0x1F = -31).

unsigned – целые без знака. В том числе 16-ричные числа.

ushort – короткие целые без знака.

real – действительные числа. В одном из двух форматов: с фиксированной точкой – 345.678 или с плавающей точкой – 3.456E+2; если нет дробной части, то и точка не нужна.

Символьные домены

char – один любой символ в апострофах.

symbol - имена, начинающиеся с символа нижнего регистра и содержащие только символы, цифры, и символы подчеркивания.

string – в двойных кавычках могут содержать любую комбинацию символов, кроме #0, который отмечает конец строки.

Домены *symbol*, *string* могут содержать специальные символы:

\n, \r − символы новой строки;

\t − символ табуляции;

\кодASCII, например: \27 − символ клавиши Esc;

\\ − символ \.

В секции DOMAINS объявляются домены списков, сложных структур (…, деревьев,….), синонимы стандартных доменов. Например:

```
people, object = symbol /*синонимы стандартного домена, интерпретируются
                          синонимы как различные домены, что позволяет
                          транслятору контролировать правильность их
                          различного использования*/
list = element*          %  списковый домен
binary_tree = b_t(binary_tree, root, binary_tree); nil  % домен бинарного дерева
```

В секции PREDICATES объявляются все предикаты, использованные в программе (кроме предопределённых): объявляется имя предиката и домены его аргументов. Например:

```
insert_tree(element, binary_tree)
```

Имена определяемых предикатов, функторов термов, специальных доменов – без пробелов последовательность любых букв (латиница, кириллица, строчные, прописные), цифр, подчёркивания; первый символ не должен быть цифрой. Нельзя использовать зарезервированные (системные, предопределённые) имена.

Предопределённые предикаты

- внелогические: write(…), writef(…), nl, !, exit – эти предикаты всегда выполняются, т.е. считаются истинными:

  writef(…) – форматированный вывод,
  nl – новая строка,
  ! – запрещает откат (**cut**, отсечение),
  exit – прекращает выполнение программы;

  Предикаты ввода данных:

  readreal(X) – для домена *real*,
  readint(X) – для целочисленных доменов, предикат ввода числа не выполнятся (значение ложь), если вводится не число;
  readln(X) – для доменов *symbol* и *string*, неудача при вводе символа с клавиши Esc;
  readchar(X) – для домена *char*, неудача при вводе символа с клавиши Esc, readchar не «печатает» вводимый символ, в отличие от остальных предикатов ввода;
  readterm(domen, Term) – читает строку и превращает её в домен указанного типа, строка должна выглядеть точно также как печатает write

указанный домен, иначе − программная ошибка. Окончание ввода клавишей Esc.

- логические: =, >, <, <>, >=, <=, not(φ (…)), fail (ЛОЖЬ);

В секции CLAUSES помещаются все факты и правила, составляющие программу. Все предложения для каждого конкретного предиката, которые называются *процедурой,* должны располагаться компактно.

GOAL – цель. Секция обязательна. Здесь формулируется цель Visual Prolog программы, запуск исполняемого файла.

Имя переменной – начинается с любой прописной (заглавной) буквы или подчёркивания, далее без пробелов последовательность любых букв, цифр, подчёркивания. Нельзя использовать зарезервированные имена.

Имя анонимной переменной – один символ: подчёркивание.

В Visual Prolog объявляются типы аргументов предикатов, а не типы переменных. Нельзя переменную заменять выражением (например, арифметическим, в том числе функцией) или предикатом.

Не правильно − summa(X, Y, X+Y).

Правильно − summa(X, Y, Z) **:–** Z=X+Y.

Или в теле фразы − …**:–**…, Z=A+B, summa(X, Y, Z),…

Область действия переменной − это одна единственная фраза программы, с тем чтобы указать на один и тот же объект в различных позициях этой фразы.

2. Простые примеры

2.1. Программа перечисления всех 3-х разрядных двоичных кодов

(декларативный вариант):

```
PREDICATES
 bit(ushort)
 digit(ushort, ushort, ushort)
CLAUSES
 bit(0). % возможные значения
 bit(1). % двоичной переменной
 digit(A, B, C) :– bit(A), bit(B), bit(C). % двоичное представление кода
GOAL   digit(A, B, C).
```

Подробное пошаговое выполнение (протокол) этой программы см. в приложении 1.

2.2. Решение задачи исчисления высказываний (см. 1.4.6):

Браун (B), Джонс (J) и Смит(S) обвиняются в преступлении. На допросе они дали показания:

B: fB: J&¬S (Джонс виновен, а Смит не виновен)

J: fJ: B⇒S (если виновен Браун, то виновен и Смит)

S: fS: ¬S&(B∨J) (Я не виновен, но кто-то из них виновен)

Программа на ПРОЛОГе описывает процесс вычисления конъюнкции этих высказывания и определения набора значений переменных (B, J, S), при которых эта конъюнкция истина.

```
DOMAINS
 u=ushort
PREDICATES
 ss(u,u,u) % конъюнкция всех показаний
 bit(u) % возможные логические значения
b(u) % Браун
j(u) % Джонс
s(u) % Смит
 fB(u,u,u) % показания Брауна
 fJ(u,u,u) % показания Джонса
 nn(u,u) % отрицание
 con(u,u,u) % конъюнкция
 imp(u,u,u) % импликация
%dis(u,u,u) % дизъюнкция
CLAUSES
 bit(0). bit(1).
 j(X) :– bit(X). b(X) :– bit(X). s(X) :– bit(X).
 nn(Y,X) :– Y=1−X. %отрицание
 con(Z,X,Y) :– Z=X*Y. %конъюнкция
 imp(Z,X,Y) :– Z=1−X*(1−Y). %импликация
%dis(Z,X,Y) :– Z=X+Y−X*Y. %дизъюнкция
 fB(W,A,B) :– j(A),s(B),nn(X,B),con(W,A,X).
 fJ(W,A,B) :– b(A),s(B),imp(W,A,B).
 ss(B,J,S) :– fB(W1,J,S),W1=1,
              fJ(W2,B,S),W2=1.
GOAL ss(B,J,S).
```

3. Рекурсивные вычисления

Рекурсивные вычисления – это один из основных механизмов вычислений в ПРОЛОГе. Механизм вызова процедуры (функции, метода в ООП), в общем случае, требует передачи данных (параметров, аргументов), одни из которых являются входными, другие выходными и наконец возврату к процессу, вызвавшему процедуру. В процедуре с рекурсией вычисления выполняются через обращения к себе самой с другими значениями аргументов. Такое обращение может выполняться через вызовы промежуточных процедур. Возможен вырожденный вариант – само вызов процедуры без аргументов (простое повторение): run:– …, run. В Visual

Prolog нет инструкций для циклов таких, например, как for и while. Эффект итераций реализуется применением рекурсивной процедуры, в которой одним из аргументов является переменная, управляющая итерацией.

Рекурсивная процедура должна включать:

(1) Нерекурсивную(ые) фразу(ы), определяющую(ие) условие прекращения рекурсии, это называют *базисом рекурсии*, чаще всего это факт(ы), но не обязательно.

(2) Фразу (одну или более), в которой(ых) в заголовке выделяются переменные, значения которых должны быть определены. В предикатах тела фразы определяются новые значения аргументов, затем следует рекурсивная подцель, в которой эти новые значения аргументов используются.

(3) Реализуются рекурсивные вызовы с использованием стека. В стек помещаются предикаты, расположенные после рекурсивной подцели, а также не конкретизированные переменные из заголовка фразы, с тем чтобы быть вычисленными, начиная с базиса.

3.1. Представление натурального числа N в системе счисления по основанию S.

```
translate(N, S) :– N>0,
                  Nt=N div S,  % целочисленное деление
                  R=N mod S, % остаток от деления
                  translate(Nt, S),
                  write(R,".").
translate(0,_) :–  write("reply >> ").
GOAL translate(22, 2).          % Будет напечатано:  reply >> 1.0.1.1.0.
```

Почему печатается в таком порядке? Вычисленные значения R не могут быть напечатаны пока не доказана цель translate(Nt,S), поэтому вычисленные значения R размещаются в стеке (stack). Доказательство цели translate(Nt, S) завершается фактом translate(0,_):–write("reply >> "). Поэтому первая печать – это текст: reply >>, затем из стека извлекаются (печатаются) вычисленные значения R предикатом write(R,"."), с тем чтобы завершить доказательство цели, стоящей в заголовке фразы: translate(N, S). Значения извлекаются из стека, а значит – в порядке обратном вычисленным.

3.2. Факториал (1). 1!=1, n!=n*(n-1)!

```
      PREDICATES
1.      factorial(long, unsigned)
      CLAUSES
2.    factorial(1,1).              % Факториал от 1 равен 1. Базис рекурсии
3.    factorial(FN, N) :– N>1,     % Чтобы вычислить факториал числа N,
4.          NewN=N-1,              % надо вычислить
```

```
5.        factorial(Ft,NewN),    % факториал числа N-1
6.        FN = N*Ft.             % и умножить его на N.
      GOAL
7.    X=7, factorial(FX, X).
```

При выполнении рекурсивных процедур используется стек отложенных вычислений, что требует дополнительного времени и памяти. При рекурсивном вычислении факториала по варианту (1) вначале запоминаются все числа n,n-1,…,1, а затем вычисляется их произведение. См. ПРИЛОЖЕНИЕ 1. Такой вариант вычисления называют ещё *нисходящим* (к базису).

Можно сконструировать рекурсивную процедуру так, что отложенных вычислений не будет. Будем называть такую процедуру рекурсией без хвоста (другой, более распространенный вариант - хвостовая рекурсия). Такая рекурсия эквивалентна итерационной процедуре, в которой нет отложенных вычислений. Такой вариант вычисления называют ещё *восходящим* (от базиса).

В операторном варианте:

```
definition factorial(FactN, N)
do  I:=1; P:=1;
    while I<N
      do   I:=I+1;
           P:=P*I;
      od;
    FactN:=P
od
```

3.3. Факториал (2) – рекурсией без хвоста

```
factorial (FactN, N) :– factorialt(FactN, N, 1, 1).
factorialt(FactN, N, I, P) :– N>I, NewI=I+1,
                              NewP=P*NewI,
                              factorialt(FactN, N, NewI, NewP).
factorialt(FactN, N, N, FactN).
```

Базис рекурсии:

factorial(FactN, N, N, FactN) – конец рекурсии при равенстве аргументов.

factorial(FactN, N, N, FactN) – промежуточную величину (последний аргумент результата) приравняли выходному аргументу. Назвали одинаково разные аргументы, тем самым сделали их равными.

При вычислении факториала по варианту (2) к моменту рекурсивного вызова текущее значение факториала уже вычислено (нет отложенных вычислений), точно также как в итерационной процедуре вычисления факториала.

Ещё пример рекурсии без хвоста.

3.4. Наибольший Общий Делитель (НОД)

Если A>B, то НОД(A,B) = НОД(A−B,B)=…= НОД(X,X).
Для вычисления НОД нужно определить три альтернативы.

```
nod(X, Y) :– X>Y, NX=X−Y, nod(NX, Y).
nod(X, Y) :– X<Y, NY=Y−X, nod(X, NY).
nod(A, A) :– write("НОД=",A).
GOAL nod(165,75).
```

Напечатает: НОД=15 yes

Рекурсия может быть переделана в итерацию, но часто это сделать не просто, иногда невозможно. Необходимое условие рекурсии без хвоста – это отсутствие в теле фразы предикатов после рекурсивного вызова. Но это недостаточное условие. Программируя в ПРОЛОГЕ, вовсе не обязательно стремится создавать «бесхвостые» рекурсии.

Бывают «совсем плохие» рекурсии (порядка более высокого чем первый), когда стоит позаботиться об оптимизации кода.

3.5. Числа Фибоначчи ($F_n=F_{n-1}+F_{n-2}$). Буквальная реализация рекурсивного соотношения в виде программы:

```
f(0,0).
f(1,1).
f(F, N) :–  N1=N–1, f(F1, N1),
            N2=N–2, f(F2, N2),
            F=F1+F2.
```

Такой рекурсивный вариант требует вычисления одной и той же величины дважды, и каждый раз столь же не эффективно. Буквальная реализация такого прямолинейного варианта вычислений в виде рекурсии без хвоста так же потребует повторных вычислений. (Программы со структурой цикла также не лишены это недостатка.) Хотелось бы, чтобы программа содержала только одно рекурсивное обращение и не вычисляла одно и то же значение повторно. Это можно сделать, если запоминать промежуточный результат. Последний аргумент предиката f1 выполняет эту функцию.

```
f1(1,1,0).
f1(F, N, P) :– N>1, N1=N–1,
               f1(NF, N1, NP),
               F=NF+NP,
               P=NF.
f(F, N) :– f1(F, N, F).
```

Вычисления можно реализовать в виде рекурсии без хвоста:

```
f1(F, N, I, P, S) :– I<N,
                     NewI=I+1,
                     NewS=P+S,
                     f1(F, N, NewI, S, NewS). % P и S поменялись местами
f1(F, N, N, F,_).
f(F, N) :– f1(F, N, 0,0,1).
```

4. Списки

Терм списка − это последовательность элементов списка в квадратных скобках. Все элементы списка принадлежат одному и тому же домену, например: [1,3,2], этот список – константа. Элементами списка могут быть переменные, например: [X,Y,Z]. Список может быть пустым: [ ]. Пустой список – константа. Список − это структура последовательного доступа. Доступ к элементам списка осуществляется последовательно с первых (левых) элементов списка.

Список можно разделить на «голову» и «хвост». Операция деления списка на голову и хвост обозначается при помощи вертикальной черты (|): [Head1,Head2 | Tail].

- Head1,Head2 − это переменные для обозначения двух элементов головы списка.
- Переменная Tail обозначает хвост списка, который является списком.

Пустой список нельзя разделить на голову и хвост.

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

```
list = elem*  % где elem − домен элемента списка.
```

Все элементы списка принадлежат одному и тому же домену, но домен может иметь альтернативное описание в виде элементов разного типа, каждый из типов должен быть именован (иметь функтор).

```
vd = i(integer); c(char); s(string); date(integer, symbol, integer)
list = vd*
```

Пример, составного списка:

Задан: L=[i(2), c('c'), s("This"), date(1,september,2014)]

Такой список называют *составным списком*.

Напечатан предикатом write(L):

[i(2), c('c'), s("This"), s("list"), date(1,”september”,2014)]

Возможны *вложенные списки*:

- Фиксированной вложенностью.

```
i = integer
```

```
list = i*
lilist = list*
```

Пример: [[1,2,3],[1,2,4],[1,3,4],[2,3,4]]

- Произвольной вложенности, используя домен с альтернативами.

```
lilist = li(list); e(elem)
list = lilist*
elem= integer
```

Пример, списка : [e(11),li([e(22),li([e(33),e(43)]),e(52)]),e(61),li([e(72),e(82)])],
Вторая цифра − это уровень списка.

- Разумеется, составные произвольной вложенности.

4.1. Программа перечисления всех 3-х разрядных двоичных кодов

(вариант со списком)

```
DOMAINS
 u=ushort
 list=u*
PREDICATES
   bit(u)
   digit(list)
CLAUSES
    bit(0). bit(1).
    digit([A, B, C]) :— bit(A), bit(B), bit(C).
GOAL    digit(L).
```

4.2. Длина списка (декларативный вариант с хвостом)

```
   DOMAINS
1.   list = elem*
2.   elem=  % домен элемента списка
   PREDICATES
3.   length(list, integer)
   CLAUSES
4.   length([ ], 0).                          % длина пустого списка 0
5.   length([_|T], D):— length(T, Dt),       % длина списка на единицу
6.                          D = Dt+1.         % больше чем длина хвоста
7. GOAL    LIST=[1,2,3], length(LIST, D).
```

4.3. Элемент списка

```
   DOMAINS
1.   namelist = name*
2.   name = symbol
   PREDICATES
3.   member(name, namelist)
   CLAUSES
4.   member(X, [X|_]).
5.   member(X, [_|Tail]) :— member(X, Tail).
```

6. GOAL member(a, [aa,b,cc]).

Ответ: No Solution

Декларативная трактовка: X либо в голове списка, либо в хвосте списка.

Процедурная трактовка: для проверки, является ли X элементом списка L, нужно

(1) проверить, не совпадает ли голова списка с X,

(2) проверить, не принадлежит ли X хвосту списка Tail.

предикат member(X, [X|_]) заканчивает рекурсию при совпадении значения первого аргумента с головой списка, но, если при исчерпании списка такого элемента нет, то предикат member не будет доказан (ответ: **no**).

Если GOAL member(X, [aa,b,cc]), то в силу откатов решение:

X=aa, X=b, X=cc.

4.4. Добавление элемента в голову списка

```
insert(X, L, [X|L]). % X− элемент, L− список
```

4.5. Добавление без повторений элементов

```
insert(X, L, L) :— member(X, L). %список не меняется
insert(X, L, [X|L]).
```

4.6. Добавление с терминала в голову списка

```
1. update_stack(L, NewL) :— write(">> "), readint(C),
2.                          update_stack([C|L], NewL).
3. update_stack(L, L).% список будет создан после неудачи readint
4. GOAL update_stack (L, Lf).
```

Элементы к списку добавляются по принципу стека: последний из введённых элементов будет первым в списке.

4.7. Создание нового списка - очереди

```
create_queue([H|T]) :– write(">> "), readint(H),
                       create_queue(T).
create_queue([ ]).
GOAL create_queue(L).
```

В заголовке фразы выделяется переменная (H), с которой что-то будет происходить в теле фразы. После того как все предикаты тела будут доказаны, эта уже конкретизированная переменная должна стоять в голове списка. См. приложение 1.

4.8. Реверс списка

```
reverse(L, Lr) :— app_rev(L, [ ], Lr).    %второй аргумент –это накопитель
                  app_rev([ ], L, L).
app_rev([H|T], Acc, Lr) :— app_rev(T, [H|Acc], Lr).
```

Элементы исходного списка помещаем по очереди в накопитель; также как при «стековом» создании списка, поэтому элементы расположатся в обратном

порядке – последний из списка L будет первым в списке Acc; после исчерпания исходного списка, список Acc становится списком Lr.

4.9. Палиндром

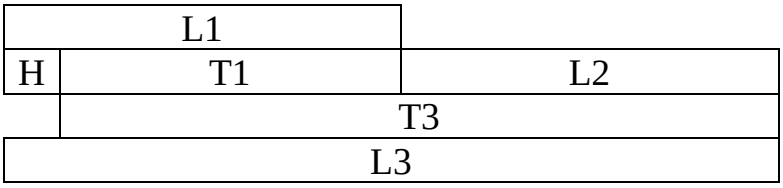
```
palindrom(L) :— reverse(L, L). % если есть процедура reverse
```

4.10. Слияние списков

Доказать истинность предиката append(L1, L2, L3), у которого список L3 — катенация списков L1 и L2.

```
  append([ ], L, L).
  append([H|T1], L2, [H|T3]) :— append (T1, L2, T3).
GOAL   append ([a,b,c], [d,e], L).
```

Декларативный смысл этой процедуры в описании совместной структуры списков. Если список L1 пуст, то L2 и L3 один и тот же список. Голова списка L3 должна быть такой же, как голова списка L1 (append([H|T1], L2, [H|T3])). Тоже самое для хвостов этих списков (…**:—** append (T1, L2, T3)).



Это не рекурсия без хвоста, поскольку переменные в заголовке фразы требуют конкретизации. См. приложение 1.

Аргументы в ПРОЛОГЕ из входных могут стать выходными.

Эту процедуру append можно применить, как бы в обратном направлении, для разбиения заданного списка на две части.

```
GOAL   append(X, Y, [1,2,3,5,6])
```

Выдаст все варианты разбиения на два списка:

X=[ ], Y=[1,2,3,5,6],
X=[1], Y=[2,3,5,6],
……..,
X=[1,2,3,5,6], Y=[ ]

Эту процедуру можно применить для поиска в списке комбинации элементов, удовлетворяющей некоторому условию. Например, можно найти элементы, предшествующие данному, и все элементы, следующие за ним.

GOAL X=3, append(TO, [X | AFTER], [1,2,3,5,6]).

Solution:

Можно найти элемент, непосредственно предшествующий данному, и элемент, непосредственно следующий за ним.

GOAL X=3, append(_, [TO, X, AFTER |_], [1,2,3,5,6]).

Solution:

Можно удалить из списка все элементы, которые следуют за данным вместе с заданным.

GOAL L3=[1,2,3,5,6], X=3, append(L1, [X |_], L3).

Каким должен быть список L1, чтобы получился список L3 ?

Solution:

***4.11.* «Принадлежать списку» через слияние списков**

```
member(X, L) :— append(_, [X|_], L).
```

Разумеется должна быть процедура append.

4.12. Добавить последним

Можно конечно использовать процедуру слияния списков: (1)

```
add_end(X, L, Lf) :— append(L, [X], Lf).
append([H|T1], L2, [H|Tf]) :— append(T1, L2, Tf).
append([ ], L, L).
GOAL   add_end(e, [d,c,a], L).
```

Или по аналогии с процедурой слияния списков: (2)

```
add_end(X, [H|T], [H|Tf]) :— add_end(X, T, Tf).
add_end(X, [ ], [X]).
GOAL    add_end(e, [d,c,a], L).
```

4.13. Добавление к списку как к очереди (см. создание списка очереди 4.7)

```
update_queue(L, NewL) :— write(">>"), readint(C),
                         add_end(C, L, TempL),
                         update_queue(TempL, NewL).
update_queue(L, L).
```

4.14. Удаление одного заданного элемента из списка

```
delete(X, [X|T], T) :— !.
delete(X, [H|T], [H|Tf]) :— delete(X, T, Tf).
GOAL  delete(X, [d,a,c,a,t,a], L).
```

Удалит только первый элемент списка «d». Если X конкретизировано, например, X=a, то удалит первый из элементов «a». Без отсечения (!) будет по очереди удалять по одному элементу (в силу откатов).

4.15. Процедуру delete можно использовать в обратном направлении для того, чтобы добавлять элемент в список, вставляя его в произвольные места списка.

```
delete(X, [X|T], T).
delete(X, [H|T], [H|Tf]) :– delete(X, T, Tf).
insert(X, LLittle, LBig) :– delete(X, LBig, LLittle).
GOAL insert("W", [d,a,c,a,t,a], L).
```

Предикат insert(X, LL, LB) истинен, если истинен предикат delete(X,LB,LL).

Элемент X размещается в списке LL, который становится списком LB, если элемент X можно удалить из списка LB.

4.16. Процедуру delete (удалить) можно использовать для определения принадлежности элемента списку. Элемент X принадлежит списку, если X из этого списка можно удалить.

```
delete(X, [X|T], T).
delete(X, [H|T], [H|Tf]) :– delete(X, T, Tf).
member(X,L) :– delete(X,L,_).
GOAL member(X, [d,a,c,a,t,a]).
```

Если в программе нужны все три действия (определять принадлежность, вставлять и удалять), то можно использовать только одну общую процедуру delete.

4.17. Удаление из списка всех экземпляров заданного элемента

```
1. delete_all(_, [ ], [ ]).
2. delete_all(X, [X|T], L):– delete_all(X,T,L).
3. delete_all(X, [H|T], [H|Tf]):–  X<>H,
4.                                 delete_all(X, T, Tf).
```

4.18. Удаление последнего элемента списка

```
del_last([ _ ], [ ]).  % вот собственно удаление последнего
del_last([H|T1], [H|T2]):– del_last(T1, T2).
```

4.19. Удаление дубликатов в списке *(превращение списка в множество)*

```
1. no_doubl(L, Lf) :– delrepeate(L, [ ], Lf).
   % второй аргумент это накопитель (см. «реверс списка (1)»)
2. delrepeate([ ], L, L).
3. delrepeate([H|T], Acc, Lf):– member(H, T), !,
4.                              delrepeate(T, Acc, Lf).
5. delrepeate([H|T], Acc, Lf) :– delrepeate(T, [H|Acc], Lf).
```

Как написать без (!)?

Итоговый список (без повторений элементов) будет перевёрнут относительно исходного.

4.20. Быть подсписком (1) декларативный вариант

```
sublist(SL, L) :– append(Lh, _, L),
                  append( _, SL, Lh).
append([H|T1], L2, [H|Tf]) :– append(T1, L2, Tf).
append([ ], L, L).
```

4.21. Быть подсписком (2) процедурный вариант

```
1. sublist(L1, [ _ |T2]) :– sublist(L1, T2).
2. sublist([H|T1], [H|T2]) :– front(T1,T2).
3. front([ ], _).  % пустой список – всегда подсписок (несобственный)
4. front([H|T1], [H|T2]) :– front(T1, T2).
```

/* начиная с некоторого H и до исчерпания списка 1, все элементы должны совпасть */

4.22. Быть подмножеством

```
subset([ ], _).
subset([H1|T1], L2) :— member(H1, L2),
                       subset(T1, L2).
GOAL  subset([c,a,d], [a,b,c,d,e]).
```

Запрос к этой процедуре не должен содержать переменных. Поэтому её нельзя использовать для порождения всех подмножеств. L2 — должен быть конкретизирован потому, что обращаемся к процедуре member. L1 (первый аргумент) — должен быть конкретизирован потому, что он обеспечивает управление рекурсией (в списке допустимы повторения элементов (это не множество) поэтому первый же элемент при запросе subset(X, [a,b,c,d,e]) породит бесконечное число решений).

4.23. Все подмножества (см. «Быть подсписком (2)» − 4.21)

```
subset_all([ ], [ ]).
subset_all([H|Tx], [H|T]) :— subset_all(Tx, T).
subset_all(Lx, [ _ |T]) :— subset_all(Lx, T).
GOAL   subset_all(X, [a,b,c]).
```

Нельзя использовать для проверки отношения «быть подмножеством».

GOAL subset_all([c,a], [a,b,c]). Ответ: no.

Можно проверять отношение «быть подпоследовательностью».

GOAL subset_all([a,c], [a,b,c]). Ответ: yes.

4.24. Все подмножества фиксированной мощности (сочетания)

```
DOMAINS
  list = i*
  i = integer
PREDICATES
  subset_f (list,list,i)
CLAUSES
  subset_f ([ ],[ ],0).
  subset_f ([H|Tx],[H|T],N) :— N>0,N1=N-1,
                               subset_f (Tx,T,N1).
  subset_f (Lx,[_|T],N) :— subset_f (Lx,T,N).
GOAL   readterm(list,L),     % L=[1,2,3,4,5]
       readint(K),           % K=3
       subset_f (L,T,K).
```

4.25. Объединение множеств

```
unionset([ ], L, L).
unionset([H|T1], L2, Lf) :— member(H, L2),!,
```

```
                                    unionset(T1, L2, Lf).
unionset([H|T1], L2, [H|Tf]) :— unionset(T1, L2, Tf).
```

4.26. Пересечение множеств

```
intersect([ ], _, [ ]).
intersect([H|T1], L2, [H|Tf]) :— member(H, L2),!,
                                 intersect(T1, L2, Tf).
intersect([_|T1], L2, Lf) :— intersect(T1, L2, Lf).
```

Факт intersect(_, [ ], [ ]) не нужен.

4.27. Разность множеств

```
difset([ ], _, [ ]).
difset([H|T1], L2, Lf) :— member(H, L2),
                          difset(T1, L2, Lf).
difset([H|T1], L2, [H|Tf]) :— not(member(H, L2)),
                            difset(T1, L2, Tf).
```

5. Сортировка списков

5.1. Метод обменов (парными перестановками, пузырька, взбалтыванием)

Для того, чтобы исходный список L преобразовать в упорядоченный список Ls, sort_bubl(L, Ls) необходимо:

- найти в списке L два смежных элемента X и Y, таких, что X>Y и поменять (permute) их местами, получив тем самым новый список NewL; затем отсортировать этот новый список;
- список L отсортирован тогда, когда в нём не будет изменений.

```
1. sort_bubl(L, Ls) :— permute(L, NewL),!,
2.                      sort_bubl(NewL, Ls).
3. sort_bubl(L, L).
4. permute([X,Y|T], [Y,X|T]) :— X>Y.
5. permute([H|T], [H|Ts]) :— permute(T, Ts).
```

5.2. Сортировка со вставками

Для того, чтобы исходный список L преобразовать в упорядоченный список Ls, sort_in(L,Ls) необходимо:

- удалить из списка L голову H, упорядочить хвост T этого списка, получив тем самым упорядоченный хвост Ts;
- затем вставить (in_s) удалённый элемент H в отсортированный хвост Ts так, чтобы получившийся список Ls остался упорядоченным;
- пустой список уже отсортирован. Рекурсивная структура процедуры приведёт к тому, что вставки начнутся с пустого списка.

```
1. sort_in([H|T], Ls) :— sort_in(T, Ts),
2.                       in_s(H, Ts, Ls).
3. sort_in([ ], [ ]).
```

```
4.  in_s(X, [H|T1], [H|T2]) :— X>H, !,
5.                             in_s(X, T1, T2).
6.  in_s(X, L, [X|L]).
```

5.3. Быстрая сортировка

Для того, чтобы непустой список L упорядочить и получить список Ls, используя процедуру sort_quick(L,Ls) необходимо:

- удалить из списка L какой-нибудь элемент (проще всего голову H) и разбить (partition) оставшуюся часть на два списка, список Llit, с элементами меньшими или равными элементу H, и список Lbig, с остальными элементами (большими, чем H);
- отсортировать эти списки и получить списки LLS и LBS
- сформировать окончательный список Ls, соединив списки LLS и [H|LBS]

```
1.  sort_quick([ ], [ ]).
2.  sort_quick([H|T], Ls) :— partition(H, T, Llit, Lbig),
3.                           sort_quick(Llit, LLS),
4.                           sort_quick(Lbig, LBS),
5.                           append(LLS, [H|LBS], Ls).
6.  partition(_,[ ], [ ], [ ]).
7.  partition(X, [Y|T], [Y|LL], LB):— X>Y, !,
8.                                    partition(X, T, LL, LB).
9.  partition(X, [Y|T], LL, [Y|LB]):— partition(X, T, LL, LB).
```

5.4. Сортировка слиянием

sort_merge(L,Ls). Чтобы отсортировать список, необходимо:

- разбить (divide) список на два списка L1 и L2 примерно одинаковой длины;
- отсортировать эти списки, получив списки L1s и L2s;
- слить (merge) эти списки в один Ls, не нарушая упорядоченности.

```
1.  sort_merge([ ], [ ]).
2.  sort_merge([X], [X]).  % если список нечётной длины
3.  sort_merge(L, Ls) :—  divide(L, L1, L2),
4.                        sort_merge(L1, L1s),
5.                        sort_merge(L2, L2s),
6.                        merge(L1s, L2s, Ls).
7.  divide([ ], [ ], [ ]).
8.  divide([X], [X], [ ]).  % если список нечётной длины
9.  divide([X,Y|T], [X|T1], [Y|T2]) :— divide(T, T1, T2).

10. merge([ ], L, L).
11. merge(L, [ ], L).
12. merge([X|T1], [Y|T2], [X|Ts]) :— X<Y, merge(T1, [Y|T2], Ts).
13. merge([X|T1], [Y|T2], [Y|Ts]) :— X>Y, merge([X|T1], T2, Ts).
```

```
14. merge([X|T1], [Y|T2], [X,X|Ts]) :— X=Y, merge(T1, T2, Ts).
```

6. ВЛОЖЕННЫЕ СПИСКИ

6.1. Слияние

Процедура слияния append одна и та же для списков любой одинаковой вложенности.

```
DOMAINS
  list1 = elem*
  list2 = list1*
  elem=integer
PREDICATES
  append(list2,list2,list2)
CLAUSES
  append([ ], L, L).
  append([X|L1], L2, [X|L3]) :— append(L1, L2, L3).
GOAL
  append([[0],[1,2],[3]],[[4],[5]],Ls).
```

6.2. Линеаризация списка произвольной вложенности

```
    DOMAINS
1.    lilist = l(list); e(elem)
2.    list = lilist*
3.    elem = integer
4.  PREDICATES
5.    a_list(list, list)
6.    acc(list, list, list)
7.    append(list, list, list)
    CLAUSES
8.    a_list(L, Lf) :— acc(L, [ ], Lf) .
9.    acc([e(X)|T], Acc, Lf):— acc(T, [e(X)|Acc], Lf).
10.   acc([l(X)|T], Acc, Lf):— append(X, T, L),
11.                              acc(L, Acc, Lf).
12.   acc([ ], L, L).
13.   append([H|T], L, [H|Tf]):— append(T, L, Tf).
14.   append([ ], L, L).
    GOAL
15.   a_list([e(11),l([e(22),l([e(33),e(43)]),e(52)]),e(61),l([e(72),e(82)])], L), nl.
```

Результат: L=[e(82),e(72),e(61),e(52),e(43),e(33),e(22),e(11)] yes

Правда, список получиться в обратном порядке, но это легко исправить.

6.3. Транспонирование матрицы

Матрица представлена двухуровневым списком.

```
DOMAINS
```

```
 li1 = r*
 li2 = li1*
 r=real
PREDICATES
 transpose(li2,li2)
 t0(li2,li2,li2)
 t1(li1,li2,li2)
 tR(li2,li2)
 tN(li2,li2)
 tN1(li1,li2)
 app_rev(li1,li1,li1)
 reverse(li1,li1)
CLAUSES
 transpose(L,Lf) :— % вызов процедуры транспонирования
                   tN(L,Ln), /* вычисление Ln=[[ ],…,[ ]], где количество
                                пустых подсписков равно количеству столбцов
                                исходной матрицы*/
                   t0(L,Lt,Ln), % транспонирование
                   tR(Lt,Lf),    % реверс подсписков строк
                   write(L),nl,write(Lf),
                   exit.
 tN([H|_],Ln) :— tN1(H,Ln).
 tN1([_|T],[[ ]|L]) :— tN1(T,L).
 tN1([ ],[ ]).
 t0([R|T],Lf,Ln) :— t1(R,Lt,Ln),
                    t0(T,Lf,Lt).
 t0([ ],L,L).
 t1([E|T1],[[E|H]|Tf],[H|T2]) :— /* E — очередной элемент исходной строки
                                     H — очередная новая строка
                                     [E|H] — модификация новой строки */
                                  t1(T1,Tf,T2).
 t1([ ],[ ],[ ]).
 tR([H|T],[HR|Tf]) :—  reverse(H,HR),
                       tR(T,Tf).
 tR([ ],[ ]).
  reverse(L1,L2) :— app_rev(L1,[ ],L2).
  app_rev([ ],L,L).
  app_rev([H|T1],L2,L3) :— app_rev(T1,[H|L2],L3).
GOAL  L=[[11,12,13],[21,22,23]],
          transpose(L,Lf).
```

Результат: Lf=[[11,21],[12,22],[13,23]] yes

6.4. Предикат findall

Предикат второго порядка, поскольку второй аргумент - это предикат. Встроенный предикат findall(Var, myPredicate(_, _, Var, _), List) вызывает процедуру (myPredicate), указанную в качестве второго аргумента, и собирает все решения для одной из переменных этого предиката (Var) в список (List). Предикат findall всегда истинен, т.е. список List=[ ], если у предиката myPredicate – нет решений.

Пример применения предиката findall в программе «***получения всех перестановок элементов списка***».

```
   DOMAINS
1.    i=integer
2.    list = i*
3.    lilist = list*
   PREDICATES
4.   delete(i, list, list)
5.   permute(list, list)
6.   permutation(list, lilist)
   CLAUSES
7.   permute([ ], [ ]).
8.   permute(L, [X|P]) :— delete(X, L, Lt),
9.                        permute(Lt, P).
10.  delete(X, [X|T], T).
11.  delete(X, [H|T], [H|Tf]) :— delete(X, T, Tf).

12.  permutation(L, LS) :— findall(Lt, permute(L, Lt), LS).
13. GOAL   permutation([1,2,3], LS).
```

Все перестановки N объектов могут быть получены, если брать по очереди каждый объект и помещать его перед всеми перестановками (N-1) оставшихся объектов. Через переменную X по очереди удаляются элементы из исходного списка, которые затем ставятся в голове переставленного списка.

7. ГРАФЫ

7.1. Путь в ациклическом графе

7.1.1.

```
DOMAINS
s=symbol
list=s*
PREDICATES
 d(s,s)
 path(s,s,list)
CLAUSES
```

```
 d(s,a).d(s,b).d(s,c).d(d,f).d(e,f).d(g,f).
 d(a,d).d(a,e).d(d,b).d(c,e).d(b,g).d(g,c).
 path(A,Z,[A|Trace]):-d(A,X),              % все решения в силу откатов
                       path(X,Z,Trace).   %печать в прямом порядке
 path(X,X,[X|[ ]]).   % чтобы последняя вершина попала в список
GOAL path(s,f,T).
```

7.1.2.

```
DOMAINS
s=symbol
list=s*
PREDICATES
 d(s,s)
 path(s,s,list)
 put(s,s,list,list)
CLAUSES
 d(s,a).d(s,b).d(s,c).d(d,f).d(e,f).d(g,f).
 d(a,d).d(a,e).d(d,b).d(c,e).d(b,g).d(g,c).
 path(A,Z,T) :– put(A,Z,T,[A]).
 put(A,Z,T,Trace) :– d(A,X),
                      put(X,Z,T,[X|Trace]).
 put(X,X,T,T).
GOAL path(s,f,T). % все решения в силу откатов, печать в обратном порядке
```

7.1.3.

```
DOMAINS
 s=symbol
 list=s*
PREDICATES
 d(s,s)
 put(s,s,list)
 put0(s,s)
CLAUSES
 d(s,a).d(s,b).d(s,c).d(d,f).d(e,f).d(g,f).
 d(a,d).d(a,e).d(d,b).d(c,e).d(b,g).d(g,c).

 put0(A,Z) :– put(A,Z,[A]).
 put(A,Z,Trace) :– d(A,X),
                     put(X,Z,[X|Trace]).
 put(X,X,T):– write(T),nl,fail. %искусственный откат печать в обратном порядке
GOAL   put0(s,f).
```

7.1.3′. С вводом структуры графа с терминала.

```
DOMAINS
 i=integer
 list=i*
```

```
FACTS-graf      % раздел базы данных
 d(i,i)
PREDICATES
 put(i,i,list)
 run
CLAUSES
 put(A,Z,Trace) :– d(A,X),
                   put(X,Z,[X|Trace]).
 put(X,X,T) :– write(T),nl,fail.
 run:– readterm(graf,Term),  % ввод дуг графа (окончание клавишей Esc)
       assert(Term,graf),run.  % в базу данных
 run:– write(">>"),readint(X),
       write("<<"),readint(Y),
       put(X,Y,[X]).
GOAL write("GRAF:"),nl,run.
```

7.2. Путь в неориентированном графе

```
rebro(X,Y) :– d(Y,X);d(X,Y). % ребро как симметричное отношение
put0(A,Z) :– put(Z,A,[Z]).      % ищем с финиша, чтобы печатать со старта
put(A,Z,Trace) :– rebro(A,X),
                  not(member(X,Trace)), % не должно быть циклов
                  put(X,Z,[X|Trace]).
 put(X,X,T) :– write(T),nl,fail.

 member(Name,[Name|_]).и
 member(Name,[_|Tail]) :– member(Name,Tail).
GOAL put0(s,f).
```

8. БИНАРНЫЕ ДЕРЕВЬЯ

8.1. Домен бинарного дерева *определяется рекурсивно:*

```
bi_tree = tree(bi_t, root, bi_t); nil
```

tree ─ функтор терма «дерево»,
root ─ домен корневого элемента (каждой вершины) дерева,
nil ─ атом пустого дерева.

Такой терм в качестве аргумента предиката может выглядеть следующим образом: tree(nil, E, Right) ─ дерево с пустым левым поддеревом.

8.2. *Печать структуры дерева*. Печать структуры дерева изображённого на рис.8.1 предикатом write(Tree).

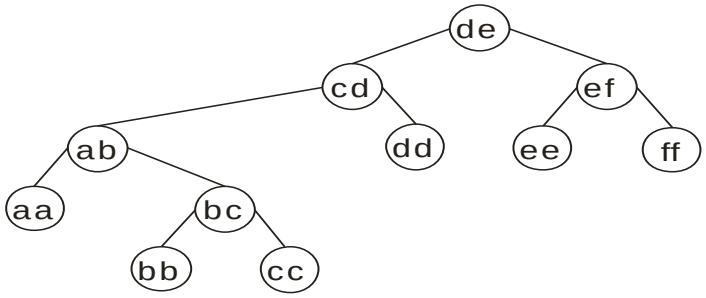


Рис. 8.1.

tree(tree(tree(tree(nil,"aa",nil),"ab",tree(tree(nil,"bb",nil),"bc",tree(nil,"cc",nil))), "cd",tree(nil,"dd",nil)),"**de**",tree(tree(nil,"ee",nil),"ef",tree(nil,"ff",nil)))

8.3. Изображение дерева (растущим в право)

```
   PREDICATES
1.   map_tree (bi_t)
2.   map(bi_t, integer).    % второй аргумент - расстояние на экране
3.   tab(integer)
   CLAUSES
4.   map_tree(T) :— map(T, 0).
5.   map(nil,_).
6.   map(tree(Left,E,Right), D) :—
7.                              NewD=D+1,        % D − количество табуляций
8.                              map(Right, NewD),
9.                              tab(D), write(E), nl,
10.                             map(Left, NewD).
11.  tab(0).
12.  tab(D):— write("\t"),
13.           NewD=D-1,
14.           tab(NewD).
```

8.4. Изображение дерева (растущим вниз)

```
1.  write_tree(Tree) :— height(Tree, Height), % вычисляется высота дерева
2.                      m_t(Tree, 0, Height). % аргумент-2 − уровень дерева
3.  m_t(_, Level, Height) :— Level>Height, !.
4.  m_t(Tree, Level, Height) :— out_t(Tree, Level, 0), nl,
5.                      NewLevel=Level+1,
6.                      m_t(Tree, NewLevel, Height).
7.  out_t(nil,_,_).
8.  out_t(tree(Left,E,Right), Level, Depth) :— NewDepth=Depth+1,
9.          out_t(Left, Level, NewDepth),
10.         w_t(E, Level, Depth),
11.         out_t(Right, Level, NewDepth).
12. w_t(E, Level, Level) :— !, writef("%2", E). /*форматированный вывод
    элемента дерева не более чем двухразрядное целое со знаком (или два
    символа)*/
```

```
13. w_t(_,_,_):— writef("%5"," "). % форматированный вывод пробелов
```

8.5. Бинарный справочник — это бинарное дерево, упорядоченное определённым образом (слева направо). Непустое дерево tree_b(Left, E, Right) упорядочено слева направо, если:

- все вершины левого поддерева Left меньше E;
- все вершины правого поддерева Right больше E;
- оба поддерева упорядочены.

(Иначе, корень любого поддерева больше корня своего левого поддерева, и меньше корня своего правого поддерева.)

Преимущество упорядочивания состоит в том, что для поиска некоторого объекта в бинарном справочнике всегда достаточно просмотреть не более одного поддерева. Экономия при поиске объекта E достигается за счёт того, что, сравнив E с корнем, можно сразу же отбросить одно из поддеревьев**.**

Создание бинарного справочника

```
1.  create_tree(Tree, NewTree):— readchar(C), C<>'\27',!,
2.                               insert(C, Tree, TempTree), nl,
3.                               map(TempTree),
4.                               create_tree(TempTree, NewTree).
5.  create_tree(Tree, Tree).
    % Новый элемент вставляется как лист, конец рекурсии
6.  insert(New, nil, tree(nil, New, nil)).
    % Если элемент уже существует, то дерево не меняется, конец рекурсии
7.  insert(E, tree(Left, E, Right),tree(Left, E, Right)).
8.  insert(New, tree(Left, E, Right), tree(NewLeft, E, Right)):— New<E,
9.                                                  insert(New, Left, NewLeft).
10. insert(New, tree(Left, E, Right), tree(Left, E, NewRight)):— E<New,
11.                                                 insert(New, Right, NewRight).
```

Если ввести такую последовательность (в домене symbol), то получим выше нарисованное дерево (рис.8.1). de,ef,ff,ee,cd,dd,ab,bc,cc,bb,aa.

8.6. Обходы дерева

Одна из полезных процедур — это процедура выстраивания элементов дерева в список. Назовём такую процедуру. Процедура производит обход всех вершин дерева и строит список из этих вершин. Часто порядок обхода вершин и помещения их в список важен для решения определённых задач.

8.6.1. Разглаживание (левосторонний обход дерева)

Процедура flatten осуществляет обход таким образом, чтобы любая вершина попадала в список после вершин левого поддерева и до вершин правого поддерева (левосторонний обход). Ели разглаживается бинарный справочник, то список будет упорядоченным.

```
flatten(nil, [ ]).
flatten(tree(Left, E, Right), L):— flatten(Left, L1),
                                   flatten(Right, L2),
                                   append(L1, [E|L2], L).
```

Результат левостороннего обхода дерева, изображённого на рис.8.1, – проекция вершин на горизонтальную линию: [aa,ab,bb,bc,cc,cd,dd,de,ee,ef,ff]

Процедура создания бинарного справочника из элементов списка, а затем разглаживания созданного дерева – эффективная процедура сортировки. При реализации такой сортировки будут удалены повторяющиеся значения.

8.6.2. Обход дерева сверху вниз

```
byp_td(nil, [ ]).
byp_td(tree(Left, E, Right), L):— byp_td(Left, L1),
                                  append([E|L1], L2, L),
                                  byp_td(Right, L2).
```

8.6.3. Обход дерева снизу вверх

```
byp_dt(nil, [ ]).
byp_dt(tree(Left, E, Right), L):— byp_dt(Left, L1),
                                  byp_dt(Right, L2).
                                  append(L2, [E], Lt),
                                  append(L1, Lt, L).
```

8.7. Удаление элемента бинарного справочника

```
% Рекурсивный поиск вершины, которую нужно удалить
del_tree_mem(_, nil, nil).
del_tree_mem(X, tree(Left,E,Right), tree(NewLeft,E,Right)) :— E>X,
                                    del_tree_mem(X, Left, NewLeft).
del_tree_mem(X, tree(Left,E,Right), tree(Left,E,NewRight)) :— X>E,
                                    del_tree_mem(X, Right, NewRight).
/* Если у удаляемой вершины одно из деревьев пусто, то вместо
удаляемой вершины присоединяется другое (может быть пустое)
дерево. */
del_tree_mem(X, tree(Left,X,nil), Left).
del_tree_mem(X, tree(nil,X,Right), Right).
/* Если у удаляемой вершины оба дерева не пусты, то нужно найти ей
замену (например) в её правом дереве, вызвав процедуру min_mig */
del_tree_mem(X, tree(Left,X,Right), tree(Left,E,NewRight)) :—
                                    min_mig( E, Right, NewRight).
/*min_mig( X,Tree,NewTree), X – перемещаемая (migration) вершина из
дерева Tree. После удаления этой вершины дерево станет NewTree.
Рекурсивно ищем в правом дереве удаляемой вершины самую левую
вершину, т.е. ищем её в левых поддеревьях */
min_mig(X, tree(Left,E,Right), tree(NewLeft,E,Right)) :—
```

```
11.                                    min_mig( X, Left, NewLeft).
   /*Правое (может быть пустое) дерево перемещаемой вершины
   подставляется вместо этой вершины. Конец рекурсии.*/
12. min_mig(X, tree(nil,X,Right), Right).
```

8.8. Добавить в корень

Добавить X на место корня дерева. При этом дерево рекурсивно делится так, чтобы оно оставалось упорядоченным. insert_root(X, T, Tnew)

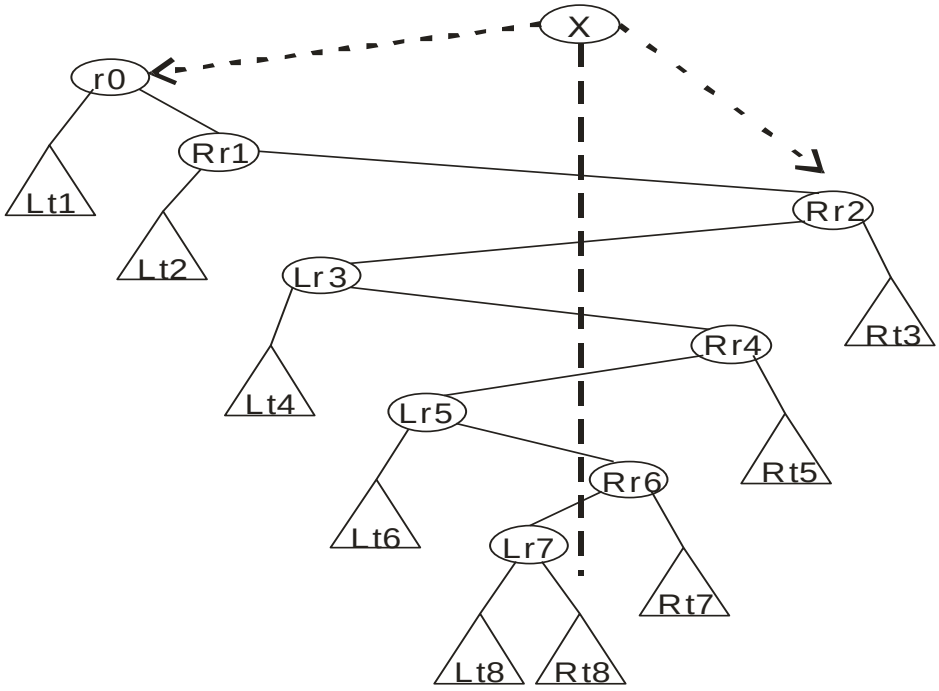


Рис. 8.2

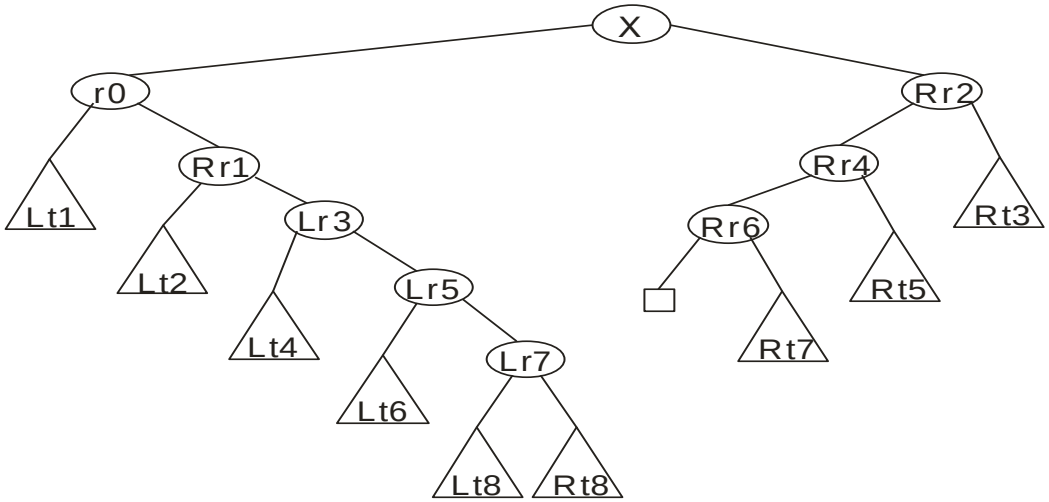


Рис. 8.3

```
1. insert_root(X, nil, tree(nil,X, nil)).
2. insert_root(E, tree(Left,E,Right), tree(Left,E,Right)).
3. insert_root(X, tree(Ly,Y,Ry), tree(Lx,X,tree(NLy,Y,Ry))) :— X<Y,
4.                                    insert_root(X, Ly, tree(Lx,X,NLy)).
5. insert_root(X, tree(Ly,Y,Ry), tree(tree(Ly,Y,NRy), X, Rx)) :— Y<X,
6.                                    insert_root(X,        Ry,
                                      tree(NRy,X,Rx)).
```

```
   /*Процедуру insert_root(X, Ty,Tx) можно использовать в «обратном
   направлении» - для удаления элемента X из дерева Tx. Процедура
   del_root(X, Ty,Tx), после того как находит элемент X, «вызывает»
   процедуру insert_root(X,Ty,Tx), которая вычисляет дерево Ty, такое,
   что при добавлении к нему элемента X получается дерево Tx.*/
7. del_root(X, T, Tnew) :— insert_root(X, T, Tnew).
8. del_root(X, tree(L,E,R), tree(NL,E,R)) :— X<E, del_root(X, L, NL).
9. del_root(X, tree(L,E,R), tree(L,E,NR)) :— E<X, del_root(X, R, NR).
```

При удалении вершины с пометкой X дерево перестраивается согласно рис.8.4, оставаясь бинарным справочником.

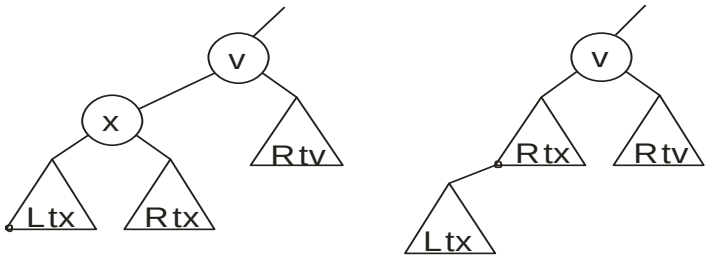


Рис. 8.4

8.9. СбалансированныеАВЛ (AVL) деревья

```
   DOMAINS
1.          t = bt(av, root, av); nil
2.          av = avl(t, integer)
            root = …
   PREDICATES
3. .        insertAVL(root, av, av)
4.          unionAVL(av, root, av, root, av, av)
5.          max_1(integer, integer, integer)
   CLAUSES
6. insertAVL(W, avl(nil,0), avl(bt(avl(nil,0),W,avl(nil,0)), 1)).
7. insertAVL(E, avl(bt(Left,E,Right), H), avl(bt(Left,E,Right), H)).
8. insertAVL(W, avl(bt(Left,E, Right),_), NewAVL):— W<E,
9.                                          insertAVL(W, Left, avl(bt(LT,X,RT),_)),
10.                                         unionAVL(LT, X, RT, E, Right, NewAVL).
11. insertAVL(W, avl(bt(Left,E,Right),_), NewAVL):— W>E,
12.                                         insertAVL(W, Right, avl(bt(LT,X,RT),_)),
13.                                         unionAVL(Left, E, LT, X, RT, NewAVL).
14. unionAVL(avl(T1,H1), A, avl(bt( L2,C,R2), H2),B,avl(T3,H3),  % старое дерево
15.          avl(bt(avl(bt(avl(T1,H1),A,L2),HA), C, avl(bt(R2,B,avl(T3,H3)), HB)), HC)):—
16.                                         H2>H1,H2>H3,
17.                                         HA=H1+1,HB=H3+1,HC=H2+1.
18. unionAVL(avl(T1,H1), A, avl(T2,H2), B, avl(T3,H3),   % старое дерево
19.                  avl(bt(avl(T1,H1),A,avl(bt(avl(T2,H2),B,avl(T3,H3)),HB)),HA)):—
20.                                         H1>=H2,H1>=H3,
21.                                         max_1(H2,H3,HB),max_1(H1,HB,HA).
22. unionAVL(avl(T1,H1),A,avl(T2,H2),B,avl(T3,H3),  % старое дерево
23.                  avl(bt(avl(bt(avl(T1,H1),A,avl(T2,H2)),HA),B,avl(T3,H3)),HB)):—
24.                                         H3>=H2,H3>=H1,
25.                                         max_1(H1,H2,HA),max_1(HA,H3,HB).
26.  max_1(X,Y,Z):— X>Y,!,Z=X+1.
```

```
27. max_1(X,Y,Z):- X<=Y,!,Z=Y+1.
```

Приложение 1. Протоколы выполнения

Факториал *«с хвостом»*

```
PREDICATES
  factorial(unsigned, long)
CLAUSES
  factorial(1,1).              %  Факториал от 1 равен 1.
  factorial(N, FN):-           %  Чтобы вычислить факториал числа N, надо
                                  вычислить
        NewN=N-1,              %  факториал числа N-1 и умножить его на N.

  factorial(NewN,Ft),
        FN = N*Ft.
GOAL
  X=3, factorial(X, FX).
```

Обозначим factorial(X, FX) как φ(X, FX).

Список целей	Унифицируемая фраза
X=3 φ(3, FX)	φ(X, FX):– Y=X-1, φ(Y, F1), FX=X*F1.
Y=2 φ(2, F1)	φ(X, F1):– Y=X-1, φ(Y, F2), F1=X*F2.
X=2, FX=3*F1	
Y=1 φ(1, F2)	φ(1, 1)
F2=1 F1=2*F2 FX=3*F1	
F1=2 FX=3*F1	
FX=6	Печать: FX=6, 1 Solution

Слияние списков.

```
        append([H|T], S, [H|Lx]) :— append (T, S, Lx).
        append([ ], S, S).
GOAL    append ([a,b,c],[d,e],L).
```

Обозначим append(L1,L2,L3) как α(L1,L2,L3)

Список целей		**Унифицируемая фраза**
	α([a,b,c],[d,e],L).	α([H|T], S, [H|L_x]):— α(T, S, L_x).
H=a, T=[b,c], S=[d,e],	L=[a|L_1]	
	α([b,c],[d,e],L_1).	α([H|T], S, [H|L_x]) :— α(T, S, L_x).
H=b, T=[c], S=[d,e],	L_1=[b|L_2] L=[a|L_1]	
	α([c],[d,e],L_2).	α([H|T], S, [H|L_x]) :— α(T, S, L_x).
H=c, T=[ ], S=[d,e],	L_2=[c|L_3] L_1=[b|L_2] L=[a|L_1]	
	α([],[d,e],L_3).	α([ ], S, S).
	L_3=[d,e] L_2=[c|L_3] L_1=[b|L_2] L=[a|L_1]	
	L_2=[c,d,e] L_1=[b|L_2] L=[a|L_1]	
	L_1=[b,c,d,e] L=[a|L_1]	
	L=[a,b,c,d,e]	Печать: L=[a,b,c,d,e]

Создание нового списка - очереди

```
create_queue([H|T]) :– write("> "), readint(H),
                       create_queue(T).
create_queue([ ]).
GOAL create_queue(L).
```

Обозначим create_queue(L) как β(L).

	Список целей	**Унифицируемая фраза**
1	β(L)	β([H|T]) :– read(H), β(T).
	read(H) β (T) L=[H|T]	
2	β (T) H=1 L=[1|T]	β([H_1|T_1]) :– read(H_1), β (T_1). % H_1=1
3	read(H_1) β (T_1) T=[H_1|T_1] L=[1|T]	
4	β (T_1) H_1=2 T=[2|T_1] L=[1|T]	β([H_2|T_2]) :– read(H_2), β (T_2). % H_2=2
5	read(H_2) β (T_2) T=[2|T_1] L=[1|T]	отказ
6	β (T_1) T=[2|T_1] L=[1|T]	β ([ ]).
7	T_1=[ ] T=[2|T_1] L=[1|T]	
8	L=[1,2]	L=[1,2]

Приложение 2. Предикаты fail и cut

При программировании могут потребоваться средства управления механизмом отката пролог-интерпретатора. В Прологе для этого используются два стандартных предиката без аргументов - предикаты **fail** и **cut** (**отсечение**), обозначаемый как «**!**».

Предикат **fail** тождественен логическому значению «ЛОЖЬ» поэтому не может быть доказан. Этот предикат (как и любой предикат, значение которого «ЛОЖЬ») инициирует откат. Предикат fail, разумеется, должен стоять последним в теле фразы.

Предикат **!** (**cut**, отсечение), который всегда истинен, предотвращает откат, в этом смысле его действие противоположно действию fail. Предикат **!**, прежде всего, применяют, когда надо ограничить вычисления первым же найденным решением (конкретизацией переменных).

Другое распространённое применение предиката **cut** это замена отрицания. Такая замена несколько компактнее, чем прямое использование отрицания (см. **отрицание**).

Отрицание (not). При программировании может оказаться полезным стандартный пролог-оператор ***отрицание*** not(φ(X)), реализующий отрицание цели φ(X), с некоторыми ограничениями по сравнению со стандартным логическим отрицанием. Точнее: достижение цели φ(X) трактуется как поиск примера для ∃Xφ(X); аналогичная трактовка∃Xnot(φ(X)) при бесконечных доменах не реализуема. Поэтому доказывается not(φ(a)) с конкретизированными переменными. Т.е. not(φ(a)) истинен тогда, когда аргумент этого оператора – цель φ(a) не может быть доказана, т.е. возникает неуспех для цели φ(a), или первый же успех цели φ(a) для цели not(φ(a)) означает неудачу. Цель not(φ(X)) при её выполнении не попадает в стек откатов, поскольку у неё нет альтернатив. Аргументом оператора not может быть только один предикат (не конъюнкция и не дизъюнкция предикатов).

Замена отрицания отсечением:

α**:—**β,δ.	α**:—**β, **!**,δ.
α**:—**not(β),γ.	α**:—**γ.

Добавление элементов в список без повторений

(1)

```
insert(X, L, L):—. member(X, L).
insert(X, L, [X|L]):— not(member(X, L)).
```

(2)

```
insert(X, L, L) :—. member(X, L),!.
insert(X, L, [X|L]).
```

Предикат **!** употребляется также для сокращения дерева доказательства цели путем отсечения некоторых его ветвей, поэтому он и называется отсечением. Этот предикат заставляет интерпретатор «забыть» все откаты, установленные с момента унификации заголовка до момента выполнения **cut**. Тем самым запрещается поиск альтернативных решений для целей, начиная с заголовка и до предиката **cut**.

Область действия оператора cut рассмотрим на следующем примере общего вида (греческие буквы означают предикаты).

α**:—**β,γ,δ,**!**, ε,ζ,η.
α**:—**θ.
λ**:—**μ,α, ν.
? λ.

После того как редуцирована цель α, перебор (откат) возможен в списке целей α, β, γ, δ. Как только будет достигнута точка отсечения (!) поиск альтернатив для целей β, γ, δ прекращается. Для цели α будет доступна только текущая фраза (например, фраза α**:—**θ недоступна). Перебор возможен в списке целей ε,ζ,η. Но поскольку α вызывалась из фразы с заголовком λ, в которой отсечений нет, то перебор в списке целей μ,α,ν разрешён, т.е. перебор для цели α может быть возобновлён с ограничениями, указанными выше.

Поскольку предикат **cut** внелогический, то он может нарушить декларативный смысл программы. Например:

α**:—**β,γ.	
α**:—**δ.	α⇐ (β&γ) ∨δ

Если изменить порядок фраз, то логический смысл останется тем же.
В случае же:

α**:—**β,**!,** γ.	
α**:—**δ.	α⇐ (β&γ) $\overline{\vee}$(β&δ)

при изменении порядка фраз логический смысл меняется

α**:—**δ.	
α**:—**β,**!,** γ.	α⇐δ∨(β&γ)

Приложение 3. Задачи

№	баллы	
1	4	НОК двух целых положительных (через сложение).
2	4	Создать список как интервал целых чисел от n1 до n2 включительно.
3	4	С шагом Step и модулем Mod (Step и Mod - взаимно простые числа больше 0) создать список длиной Mod состояний счётчика начиная со значения 0≤Begin<Mod .
4	6	Создать список простых множителей заданного целого числа (факторизация).
5	4	Создать список цифр в системе счисления по основанию M, представляющих десятичное целое число.
6	4	Число, представленное списком двоичных разрядов напечатать в виде десятичного числа.
7	4	Десятичное число, представленное списком десятичных цифр, представить десятичным числом.
8	4	Заменить в исходном списке подряд идущие одинаковые элементы одним.
9	6	Представить положительное целое в виде списка чисел Фибоначчи, сумма которых равна исходному.
10.x	4	Удалить элементы списка, находящиеся на чётных/нечётных позициях. (10.1, 10.2)
11	2	Прибавить к каждому элементу списка целых чисел единицу.
12.x	2	Вычислить сумму элементов списка целых чисел, находящихся на чётных/нечётных позициях. (12.1, 12.2)
13	2	Изменить знак на противоположный у элементов списка целых чисел.
14	2	Вычислить две суммы элементов списка целых чисел, находящихся на чётных и нечётных позициях.
15	2	Вычислить две суммы элементов списка целых чисел положительных и отрицательных.
16	2	Все элементы списка сделать положительными.
17.x	2	Заменить в списке один (все) элемент(ы) с одним заданным значением на другое заданное. (17.1, 17.2)
18	2	Удалить из списка все отрицательные элементы.
19	2	Создать список из N начальных чисел ряда Фибоначчи.
20	2	Разделить исходный список на два списка: список элементов с чётными позициями и список элементов с нечётными позициями.
21	2	Разделить исходный список целых чисел на два списка: список положительных и список отрицательных чисел.
22	2	По исходному списку целых чисел построить списки позиций положительных и отрицательных элементов.
23	2	Выдать значение последнего элемента в списке.

24	2	Заменить последний элемент списка константой.
25	2	Выдать значение предпоследнего элемента в списке.
26	2	Заменить в исходном списке N-ый элемент заданной константой.
27	2	Вставить заданный элемент на N-ую позицию исходного списка.
28	2	Удалить N-ый элемент списка.
29	2	Подсчитать количество вхождений заданного элемента в список.
30	2	По заданному порядковому номеру элемента в списке выдать его значение.
31	4	По списку элементов сформировать новый список элементов согласно заданному списка порядковых номеров.
32.x	4	Объединить два списка одинаковой длины в третий так, чтобы вначале списка были элементы с н/ч позиций первого списка, а в конце с н/ч - из второго. н - нечётные, ч - чётные. 1(нн), 2(нч), 3(чн), 4(чч).
33	4	Вычислить число элементов списка после элемента с заданным значением.
34	4	Создать список позиций заданного элемента в исходном списке.
35	4	Объединить два списка одинаковой длины в третий так, чтобы элементы на нечётных позициях были из первого списка, а на чётных - из второго.
36	4	Подсписок из K первых элементов исходного списка.
37	4	Подсписок, в котором нет K первых элементов исходного списка.
38	4	Разделить исходный список на два списка: в первый список должно попасть N элементов из начала списка, во второй – оставшиеся элементы.
39	4	Дописать заданную константу N раз в конец списка.
40	4	Подсписок из N последних элементов исходного списка, в том же порядке.
41	4	Удалить из списка N последних элементов.
42	4	Каждый из N последних элементов списка заменить заданной константой.
43	4	Вместо N последних элементов списка оставить одну заданную константу.
44	4	Выдать значение максимального элемента списка целых чисел и его порядковый номер в списке.
45	4	Проверить эквивалентность двух множеств.
46	4	Удвоив длину списка создать палиндром.
47	6	Получить список элементов, которые встречаются в исходном списке более одного раза.

48	6	Получить список элементов, которые встречаются в исходном списке ровно один раз.
49	6	Определить номер позиции в списке, с которого начинается заданный подсписок.
50	6	Подсписок длины K, начиная с позиции N исходного списка.
51	6	Заменить K элементов списка, начиная с N-ой позиции, на заданную константу.
52	6	Вставить в исходный список, начиная с N-ой позиции, K раз заданную константу.
53	6	Вставить в первый список другой список, начиная с N-ой позиции.
54	6	Удалить из исходного списка K элементов, начиная с N-ой позиции.
55	6	Удалить из исходного списка элементы, начиная с позиции n1 по n2.
56	8	Для списка целых чисел напечатать горизонтальную диаграмму.
57	10	Для списка целых чисел напечатать вертикальную диаграмму.
58	8	Напечатать n-разрядную последовательность кодов Грея
59	8	Создать обратную польскую запись для арифметической формулы.
60	8	Создать обратную польскую запись для логической формулы.
61	6	Указать характер упорядоченности списка.
62	6	Вычислить значение арифметической формулы, используя обратную польскую запись.
63	6	Вычислить значение логической формулы, используя обратную польскую запись.
64	6	Разделить исходный список целых чисел на три списка с минимально отличающимися суммами, используя жадный алгоритм.
65	6	Кратчайший путь в ненагруженном графе. (если перебором путей, то - 2 балла)
66	8	Кратчайший путь в нагруженном графе. (если перебором путей, то - 2 балла)
67	8	Путь с максимальной пропускной способностью. (если перебором путей, то - 2 балла)
68	8	По исходному списку создать описание мультимножества в виде двух списков, первый – элементы множества, второй – их количество в исходном списке.
69	8	Выполнить топографическую сортировку ациклического графа.
70	8	Минимальный остов в нагруженном графе.
71	8	По исходному списку целых чисел создать список пар, в которых первая компонента – элемент исходного списка, вторая – число его подряд идущих повторений.

72	8	По исходному списку целых чисел создать описание мультимножества в виде списка пар, в которых первая компонента – элемент множества, вторая – его количество в исходном списке.
73	10	Критический путь. (если перебором путей, то - 2 балла)
74	10	Перечислить разбиения множества из N элементов
75.x		Реализовать циклический сдвиг списка влево/вправо. 75.1,2

№	баллы	
80	2	Вычислить высоту бинарного дерева.
81	2	Вычислить количество вершин в бинарном дереве.
82	2	Вычислить количество листьев в бинарном дереве.
83	2	Указать уровень, на котором находится заданный элемент.
84	4	Напечатать путь от корня к указанной вершине в бинарном справочнике
85	4	Найти максимальный/минимальный элемент в бинарном справочнике.
86	6	Найти максимальный/минимальный элемент в AVL дереве
87	6	Перечислить вершины в бинарном дереве, находящиеся на заданном уровне.
88	6	Бинарное дерево с указанием высот поддеревьев.
89	6	Бинарное дерево с указанием количества вершин в поддеревьях.
90	8	Бинарное дерево с указанием высот поддеревьев и функцией удаления.
91	6	Бинарное дерево с указанием количества вершин в поддеревьях и функцией удаления.
92	8	AVL дерево с функцией удаления.
93	10	Построить дерево Хаффмена.