**ЛАБОРАТОРНАЯ РАБОТА №2**. ОБРАБОТКА СПИСКОВ.

**Цель**. Освоить принципы работы со списками.

Теоретические сведения.

Списки должны быть объявлены в разделе domains, например,

```
domains
  mylist=integer*
```

Здесь объявлен списочный тип mylist. Данный тип представляет собой список целых чисел. Запомните, что звездочка (*) в конце типа указывает на списочный тип. Списки – очень важный тип в Прологе.
Для работы со списками нужно использовать специальную символику. Обозначение [X | Y] позволяет оторвать от списка голову и занести ее в переменную X. Оставшаяся часть списка помещается в переменную Y. Например, если список есть

L=[1,3,7,8], то
L=[X | Y] даст X=1 и Y=[3,7,8]. Y будет списком.

Рассмотрим простейшую программу для ввода-вывода списка целых чисел:

```
domains
    li=integer*

predicates

  nondeterm work
  nondeterm writeList(li)
clauses

  work:-
     write("Input list:"), nl,
     readterm(li,L),
     nl,
     writeList(L),
     readchar(_).


   writeList([]).

writeList(LL):-
     LL=[X | Y],
     write(X,","),
    writeList(Y).

goal

  work.
```

Для ввода списка используем стандартную команду readterm(li,L), первый аргумент которой есть списочный тип, а второй списочная переменная. При вводе списка нужно вводить и квадратные скобки, как показано на рисунке 1, а в конце нажать клавишу ENTER

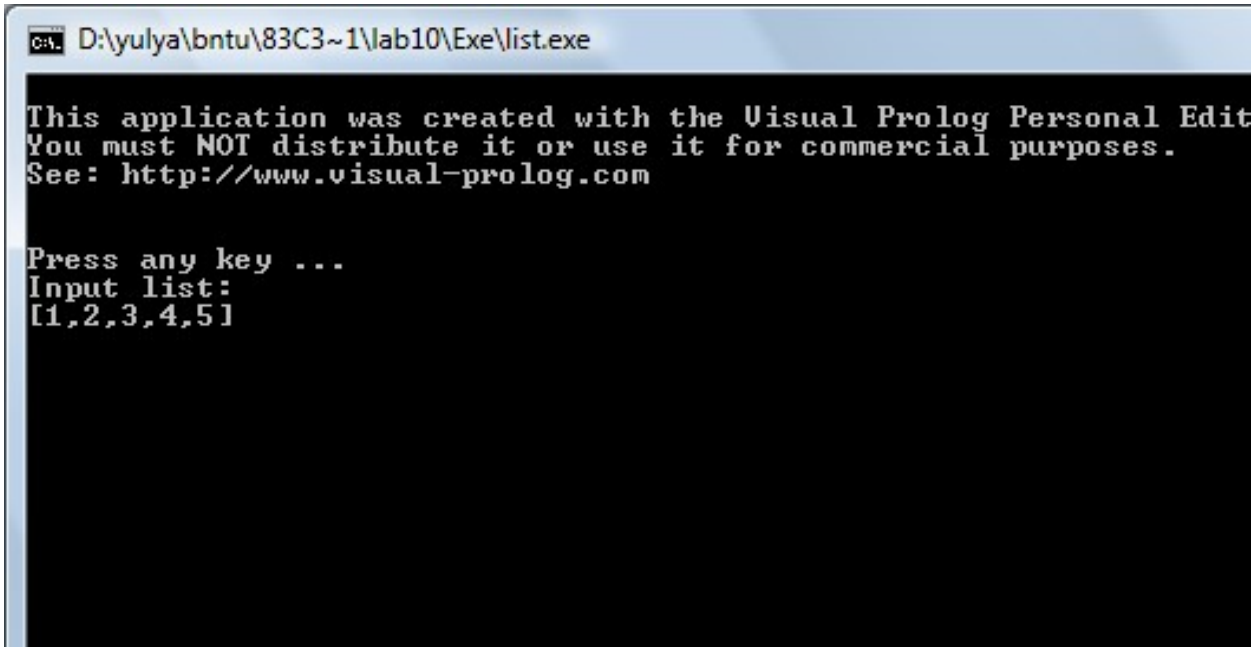


Рис. 1

При работе со списками всегда (или обычно) применяется рекурсия. Рекурсия – это раскрытие предиката через самого себя. Рассмотрим пример рекурсии в приведенной программе:

```
writeList(LL):-
    LL=[X | Y],
    write(X,","),
writeList(Y).
```

Здесь рекурсивно определяется предикат writeList(LL). Предполагается, что список LL не пустой. Тогда мы используем «распаковку» LL=[X | Y] и выводим на экран элемент-голову X:

. write(X,","),

А затем мы рекурсивно вызываем снова предикат writeList(Y), но с другими аргументами, именно - с оставшимися хвостовыми элементами Y.

Рассмотрим пример программы, проверяющей, что заданный элемент содержится в списке. Вот ее текст:

```
include "list.inc"

 domains
   li=integer*

predicates

 nondeterm work
 nondeterm member(integer,li)
clauses

 work:-
    write("Input list:"), nl,
```

```
    readterm(li,L),

    nl,
    write("Input element:"),
    readint(E),
    member(E,L),
    nl,
    write("Element prinadlezit spisku"),
    readchar(_).

  member(_,[]):-
  write("Element ne prinadlezit spisku"),
  readchar(_),
  fail.

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

goal

  work.
```

В этом примере проверку принадлежности к списку реализует предикат member. Вот его определение:

```
(1)   member(_,[]):-
          write("Element ne prinadlezit spisku"),
          readchar(_),
          fail.

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

(3)   member( X, [_ | Y]):-
        member( X,Y).
```

Здесь три правила. Каждое правило предназначено для выявления одной из возможных ситуаций. Первое правило (**1**) устанавливает, что когда список пустой (обозначается [ ]) , то элемент не принадлежит пустому списку. Команду fail надо вставлять, когда следует прекратить просмотр и завершить выполнение предиката ***неудачей***.
Второе правило (**2**) предназначено для выявления ситуации, когда элемент совпадает с головой списка. В этом случае значение хвоста списка не имеет значения. В Прологе в таком случае используют специальное обозначение для аргументов – знак подчеркивания.
Наконец, третье правило есть рекурсия. Это правило применяется, когда не подошли первые два правила, т.е., когда список не пустой и его голова не совпадает с элементом. Тогда рекурсивно вызываем предикат member , но с другими аргументами.

Наконец, рассмотрим последний пример. Требуется подсчитать сумму элементов списка. Программа такова.

```
  domains
    li=integer*

predicates

  nondeterm sum(li,integer,integer)

clauses
```

```
sum([ ],Z,R):- R=Z.

sum([X | Y],Z, R):-
  Z1=Z+ X,
   sum(Y,Z1, R).


goal

  sum([1,2,3,4,5,6,7,8],0,S), write("Summa=",S),readchar(_).
```

Здесь исходный список задается прямо в программе в цели. Второй аргумент равен 0 и представляет начальное значение суммы элементов списка. Третий аргумент должен получить значение суммы элементов списка. Мы последовательно отрываем голову и добавляем ее к текущей сумме. Это делается в рекурсивном правиле:

```
sum([X | Y],Z, R):-
   Z1=Z+ X,
    sum(Y,Z1, R).
```

Когда голов больше не останется, используется первое правило `sum([ ],Z,R):- R=Z.`

**ЗАДАНИЕ**.

1. Написать программу, проверяющую, что элемент не содержится в списке.
2. Написать программу, определяющую число элементов списка.
3. Написать программу, выдающую последний элемент списка.
4. Написать программу для отыскания минимального элемента списка.
5. Написать программу для определения разности двух списков.
6. Найти два наименьших элемента в списке целых чисел.
7. Найти элемент в списке по его номеру.
8. Упорядочить списко целых чисел по возрастаннию.