МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

МИНИСТЕРСТВО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Учреждение образования Белорусско – Российский университет

Кафедра «АСУ»

Лабораторная работа №6

на тему «Списки и рекурсия »

Могилев 2009

Оглавление

1 Цель лабораторной работы

Изучение процесса повторения: циклы и рекурсивные процедуры

1.1 Содержание отчета

- Тема и цель работы.
- Листинг программы.

2 Списки

Обработка списков, т. е. объектов, которые содержат произвольное число элементов — мощное средство Пролога. В этой главе объясняется, что такое списки и как их объявлять. Затем приводится несколько примеров, в которых показано, как можно использовать обработку списков в задачах. Далее определяются два известных предиката Пролога — `member` (член) и `append` (объединение) при рассмотрении процедурных и рекурсивных аспектов обработки списков.

После этого определяется стандартный предикат Visual Prolog — `findall`, который дает возможность находить и собирать все решения для одной цели. Завершается эта глава рассмотрением составных списков, т. е. комбинаций элементов различных типов, и примером грамматического разбора списков.

В Прологе *список* — это объект, который содержит конечное число других объектов. Списки можно грубо сравнить с массивами в других языках, но, в отличие от массивов, для списков нет необходимости заранее объявлять их размер.

Конечно, есть другие способы объединить несколько объектов в один. Если число объектов заранее известно, то вы можете сделать их аргументами одной составной структуры данных. Если число объектов не определено, то можно использовать рекурсивную составную структуру данных, такую как дерево. Но работать со списками обычно легче, т. к. Visual Prolog обеспечивает для них более четкую запись.

Список, содержащий числа 1, 2 и 3, записывается так:

[1, 2,3]

Каждая составляющая списка называется *элементом.* Чтобы оформить списочную структуру данных, надо отделить элементы списка запятыми и заключить их в квадратные скобки. Вот несколько примеров:

```
(dog,cat,canary)
["valerie ann","Jennifer caitlin","benjamin thomas"]
```

2.1 Объявление списков

Чтобы объявить домен для списка целых, надо использовать декларацию домена, такую как:

```
domains
integerlist = integer*
```

Символ (*) означает "список чего-либо"; таким образом, `integer*` означает "список целых".

Обратите внимание, что у слова "список" нет специального значения в Visual Prolog. С тем же успехом можно назвать список "Занзибаром". Именно обозначение * (а не название), говорит компилятору, что это список.

Элементы списка могут быть любыми, включая другие списки. Однако все его элементы должны принадлежать одному домену. Декларация домена для элементов должна быть следующего вида:

```
domains
elementlist = elements*
elements = ....
```

Здесь `elements` имеют единый тип (например: `integer`, `real` или `symbol`) или являются набором отличных друг от друга элементов, отмеченных разными функторами. В Visual Prolog нельзя смешивать стандартные типы в списке. Например, следующая декларация неправильно определяет список, составленный из элементов, являющихся целыми и действительными числами или идентификаторами:

```
elementlist = elements*
elements =integer;real;symbol /*Неверно*/
```

Чтобы объявить список, составленный из целых, действительных и идентификаторов, надо определить один тип, включающий все три типа с функторами, которые покажут, к какому типу относится тот или иной элемент. Например:

```
elementlist = elements*
elements = i(integer);r(real);s(symbol) %функторы здесь i,r и s
```

2.2 Головы и хвосты

Список является рекурсивным составным объектом. Он состоит из двух частей -головы, которая является первым элементом, и хвоста, который является списком, включающим все последующие элементы. *Хвост списка — всегда список, голова списка — всегда элемент.* Например:

голова [a, b, c] есть a

хвост [a, b, c] есть [b,c]

Если список одноэлементный, ответ таков:

голова[c] есть c

хвост [c] есть [ ]

Если выбирать первый элемент списка достаточное количество раз, вы обязательно дойдете до пустого списка [].

Пустой список нельзя разделить на голову и хвост.

В концептуальном плане это значит, что список имеет структуру дерева, как и другие составные объекты. Структура дерева [a, b,c,d] представлена на рис. 1.

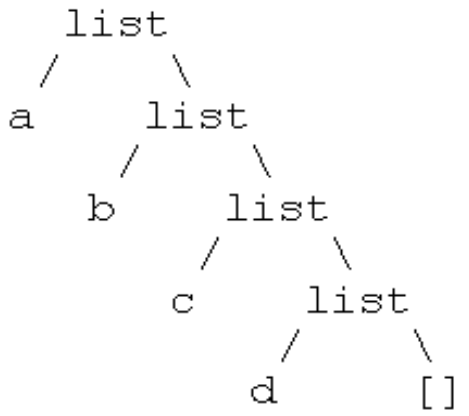

Рис. 1 – Структура дерева.

Одноэлементный список, как, например [a], не то же самое, что элемент, который в него входит, потому что [a] на самом деле — это составная структура данных, как показано на рис. 2.

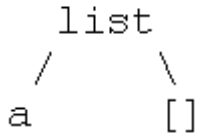

Рис.2 – Составная структура данных.

2.3 Работа со списками

В Прологе есть способ явно отделить голову от хвоста. Вместо разделения элементов запятыми, это можно сделать вертикальной чертой "|". Например:

[a, b, c] эквивалентно [a | [b, c]] и, продолжая процесс,

[a | [b, c]] эквивалентно [a| [b | [c]]], что эквивалентно [a | [b | [c | []]]]

Можно использовать оба вида разделителей в одном и том же списке при условии, что вертикальная черта есть последний разделитель. При желании можно набрать

[a, b, c, d] как [a, b | [c, d]]. В таблице 1 приведены другие примеры.

Таблица 1. Головы и хвосты списков

Список	Голова	Хвост
['a', 'b', 'c']	'a'	['b', 'c']
['a']	'a'	[] % пустой список
[]	Не определена	Не определен
[[1, 2, 3], [2, 3, 4], []]	[1, 2, 3]	[[2, 3, 4], []]

В таблице 2 приведены несколько примеров на присвоение в списках.

Таблица 2. Присвоение в списках

Список 1	Список 2	Присвоение переменным
[X, Y, Z]	[эгберт, ест, мороженое]	X=эгберг, Y =ест, E=мороженое
[7]	[X | Y]	X=7, Y=[]
[1, 2, 3, 4]	[X, Y | Z]	X=l, Y=2, Z=[3,4]
[1, 2]	[3 | X]	fail % неудача

2.4 Использование списков

Список является рекурсивной составной структурой данных, поэтому нужны алгоритмы для его обработки. Главный способ обработки списка — это просмотр и обработка каждого его элемента, пока не будет достигнут конец.

Алгоритму этого типа обычно нужны два предложения. Первое из них говорит, что делать с обычным списком (списком, который можно разделить на голову и хвост), второе — что делать с пустым списком.

2.4.1 Печать списков

Если нужно напечатать элементы списка, это делается так, как показано в листинге **программы ch07e01.pro.**

2.4.1.1 Листинг программы ch07e01.pro

```
domains
list = integer*       % Или любой тип, какой вы хотите
predicates
write_a_list(list)
clauses
write_a_list([]).  %Если список пустой — ничего не делать
write_a_list([H|T]):-  % Присвоить H - голова, T- хвост, затем...
write(H), nl,
write_a_list(T).
goal
write_a_list([1, 2, 3]).
```

Вот два целевых утверждения `write_a_list`, описанные на обычном языке:

Печатать пустой список — значит ничего не делать. Иначе, печатать

список – означает печатать его голову (которая является одним элементом) , затем печатать его хвост (список).

При первом просмотре целевое утверждение таково:

```
write_a_list([1, 2, 3]).
```

Оно удовлетворяет второму предложению при `H = 1` и `T = [2, 3]`. Компьютер напечатает 1 и вызовет рекурсивно `write_a_list`:

```
write_a_list([2, 3]).       % Это write_a_list (T)
```

Этот рекурсивный вызов удовлетворяет второму предложению. На этот раз `H = 2` и `T = [3]`, так что компьютер печатает 2 и снова рекурсивно вызывает `write_a_iist` с целевым утверждением

```
write_a_list ( [3] ) .
```

Список [3] имеет всего один элемент, у него есть голова 3 и хвост []. Следовательно, целевое утверждение снова подходит под второе предложение с `H = 3` и `T = []`. Программа печатает 3 и делает рекурсивный вызов:

```
write_a_list ( [] ) .
```

Теперь видно, что этому целевому утверждению подходит первое предложение. Второе предложение не подходит, т. к. [] нельзя разделить на голову и хвост. Так что, если бы не было первого предложения, целевое утверждение было бы невыполнимым. Но первое предложение подходит, и целевое утверждение выполняется без дальнейших действий.

2.4.2 Подсчет элементов списка

Рассмотрим, как можно определить число элементов в списке. Что такое длина списка? Вот простое логическое определение:

Длина [ ] – 0.

Длина любого другого списка — 1 плюс длина его хвоста.

Для применения этого в Прологе нужны два предложения (листинг **программы ch07e02.pro**).

2.4.2.1 Листинг программы ch07e02.pro

```
domains
list = integer*      % или любой другой тип
predicates
length_of (list, integer)
clauses
length_of([], 0). length_of ([ _ | T],) :-
length_of (T, TailLength),
L = TailLength + 1.
```

Посмотрим сначала на второе предложение. Действительно, `[_|T]` можно сопоставить любому непустому списку, с присвоением т хвоста списка. Значение головы не важно, главное, что оно есть, и компьютер может посчитать его за один элемент.

Таким образом, целевое утверждение `length_of([1,2,3,L)`. подходит

второму предложению при `T = [2, 3]`. Следующим шагом будет подсчет длины `T`. Когда это будет сделано (не важно как), `TailLength` будет иметь значение 2, и компьютер добавит к нему 1 и затем присвоит `L` значение 3.

Итак, как компьютер выполнит промежуточный шаг? Это шаг, в котором определяется длина [2, 3] при выполнении целевого утверждения

```
length_of([2, 3],TailLength).
```

Другими словами, `length_of` вызывает сама себя *рекурсивно.* Это целевое утверждение подходит второму предложению с присвоением:

- [3] из целевого утверждения присваивается `T` в предложении;
- `TailLength` из целевого утверждения присваивается `L` в предложении.

Напомним, что `TailLength` в целевом утверждении не совпадает с `TailLength` в предложении, потому что каждый рекурсивный вызов в правиле имеет свой собственный набор переменных.

Итак, целевое утверждение состоит в том, чтобы найти длину [3], т. е. 1, а затем добавить 1 к длине [2, 3], т. е. к 2, и т. д.

Таким образом, `length_of` вызывает сама себя рекурсивно, чтобы получить длину списка [3]. Хвост [3] - [], так что T будет присвоен [], а целевое утверждение будет состоять в том, чтобы найти длину [] и, добавив к ней 1, получить длину [3].

На сей раз все просто. Целевое утверждение `length_of([],TailLength)` удовлетворяет *первому* предложению, которое присвоит о переменной `TailLength`. Visual Prolog добавит к нему 1 и получит длину [3], затем вернется к вызывающему предложению. Оно в свою очередь снова добавит 1, получит длину [2, 3] и вернется в вызывающее его предложение. Это начальное предложение снова добавит 1 и получит длину [1, 2, 3].

Посмотрим иллюстрацию всех вызовов.

```
length_of  ([1,2,3],L1).
length_of  ([2,3],L2).
length_of  ([3],L3).
length_of  ([ ],0).
L3 = 0+1 = 1.
L2 = L3+1 = 2.
L1 = L2+1 = 3.
```

2.4.2.2 Упражнение

1.Запишите предикат с названием `sum_of`, который работает так же, как `length_of`, за исключением того, что он работает со списком чисел и суммирует их. Например, целевое утверждение:

```
sum_of( [1,2,3,4,S).
```

должно присваивать s значение 10.

2.Что будет, если вы попробуете выполнить целевое утверждение:

```
sum_of(List,10) .
```

Это целевое утверждение требует: "Создай мне список, к элементам которого надо добавить 10". Можно ли это сделать в Visual Prolog? Если нет, почему? (Подсказка: в Visual Prolog нельзя выполнять арифметические операции с несвязанными переменными.)

2.5 Хвостовая рекурсия

`length_of` не является и не может быть хвостовой рекурсией потому, что рекурсивный вызов не является последним шагом в своем предложении.

Проблема использования length_of заключается в том, что нельзя подсчитать длину списка, пока не подсчитана длина хвоста. Но есть обходной путь. Для определения длины списка вам потребуется предикат с тремя аргументами:

- первый — это сам список, который компьютер уменьшает при каждом вызове, пока список не опустеет так же, как и раньше;
- второй — свободный параметр, который будет хранить промежуточный результат (длину);
- третий — счетчик, который начинается с нуля и увеличивается на 1 при каждом вызове.

Когда список станет пустым, унифицируем счетчик со свободным результатом.

Рассмотрим пример (листинг **программы ch07e03.pro**).

2.5.1 Листинг программы ch07e03.pro

```
domains
list = integer*              % или любой другой тип
predicates
length_of(list,integer, integer)
clauses
length_of([], Result, Result). length_of([_|T],Result,Counter) :-
NewCounter = Counter + 1,
length__of (T, Result, NewCounter).
goal
length_of([1,2,3,L,0), %  начать  со счетчика = 0
write("L=",L),nl.
```

Данная версия предиката length_of более сложная и менее логичная, чем предыдущая. Она продемонстрирована лишь для доказательства того, что можно найти хвостовые рекурсивные алгоритмы для целевых утверждений, которые, возможно, требуют другого типа рекурсии.

2.5.2 Упражнения

1.Попробуйте обе версии length_of для очень больших списков (например, 200 или 500 элементов). Что произойдет? Как соотносятся по скорости обе версии на длинных списках?

2.Перепишите sum_of по подобию новой версии length_of.

Иногда необходимо преобразовать один список в другой. Это делается со списком поэлементно, заменяя каждый элемент вычисленным значением. Пример: **программа ch07e04.pro** добавит 1 к каждому элементу числового списка.

2.5.2.1 Листинг программы ch07e04.pro

```
domains
list = integer*
predicates
addl(list,list)
clauses
addl([]/[]). % граничное условие
addl([Head]Tail],[Headl | Tail1]) :- % отделить голову списка
```

```
Head1= Head+1, % добавить 1 к 1-му элементу
addl(Tail,Tail1). % вызвать элемент из остатка списка
goal
addl ([1,2,3,4], NewList).
```

Переведя это на естественный язык, получим:

Чтобы добавить 1 ко всем элементам пустого списка, надо создать другой пустой список. Чтобы добавить 1 ко всем элементам любого непустого списка, надо добавить 1 к голове и сделать полученный элемент головой

результирующего списка, затем добавить 1 к каждому элементу хвоста списка и сделать это хвостом результата.

Введите программу и запустите `Test Goal` с целевым утверждением

`addl ( [1,2,3,4] , NewList). Test Goal` выдаст результат:

```
NewList = [2, 3, 4, 5]
1 Solution
```

Предикат `addl` выполняет следующие операции:

1. Разделяет список на `Head` и `Tail`.
2. Добавляет 1 к `Head` и результат присваивает `Headl`.
3. Рекурсивно добавляет 1 ко всем элементам `Tail`, присваивает результат `Taiil`.
4. Объединяет `Headl` и `Taill` и присваивает результат новому списку.

Эта процедура не является хвостовой рекурсией, потому что рекурсивный вызов — это не последний шаг.

Но, что важно — Visual Prolog делает это не так; в нем `addl` является хвостовой рекурсией, потому что шаги на самом деле следующие:

1. Связать голову и хвост исходного списка с `Head` и `Tail`.
2. Связать голову и хвост результата с `Headl` и `Taill` (`Headl` и `Taill` пока не определены).
3. Добавить 1 к `Head` и присвоить результат `Headl`.
4. Рекурсивно добавить 1 ко всем элементам Tail, присваивая результат `Taill`.

Когда все будет завершено, `Headl` и `Taill` уже являются головой и хвостом результата, и нет отдельной операции для их объединения. Таким образом, рекурсивный вызов является последним шагом.

Конечно, не всегда нужно заменять каждый элемент. Далее следует пример **программы ch07e05.pro** , которая просматривает список из чисел и делает из него копию, отбрасывая отрицательные числа.

2.5.2.2 Листинг программы ch07e05.pro

```
domains
list = integer*
predicates
discard_negatives(list, list)
clauses
discard_negatives([], []).
discard_negatives([H | T], ProcessedTail):-
H < 0,      % если H отрицательно, то пропустить
!,
```

```
discard_negatives(T, ProcessedTail).
discard_negatives([H | T],[H I ProcessedTail]):-
discard_negatives(T, ProcessedTail).
```

К примеру, целевое утверждение

`discard_negatives([2, -45, 3, 468], X)` получит `X = [2, 3, 468].`

Далее приведем предикат, который копирует элементы списка, заставляя каждый элемент появляться дважды:

```
doubletalk ([], []) .
doubletalk ([H | T],[H, H | DoubledTail]) :-
doubletalk(T,DoubledTail).
```

2.6 Принадлежность к списку

Предположим, что есть список имен John, Leonard, Eric и Frank, и необходимо, используя Visual Prolog, проверить, имеется ли заданное имя в этом списке. Другими словами, нужно выяснить отношение "принадлежность" между двумя аргументами — именем и списком имен. Это выражается предикатом

```
member(name,namelist). % "name" принадлежит "namelist"
```

В **программе ch07e06.pro** первое предложение проверяет голову списка. Если голова списка совпадает с именем, которое ищется, то можно заключить, что `Name` (имя) принадлежит списку. Так как хвост списка не представляет интереса, он обозначается анонимной переменной. По первому предложению целевое утверждение

```
member(john,john,leonard,eric,frank)
```

будет выполнено.

2.6.1 Листинг программы ch07e06.pro

```
domains
namelist = name*
name = symbol
predicates
member(name,namelist)
clauses
member(Name,[Name | _ ]).
member(Name, [ _ | Tail]):-
member(Name, Tail).
```

Если голова списка не совпадает с Name, то нужно проверить, можно ли Name найти в хвосте списка. На обычном языке:

Name принадлежит списку, если Name есть первый элемент списка, или Name принадлежит списку, если Name принадлежит хвосту.

Второе предложение `member`, выражающее отношение принадлежности, в Visual Prolog выглядит так:

```
member (Name, [_|Tail]) :-
member(Name,Tail).
```

2.6.2 Упражнения

1.Загрузите **программу ch07e06.pro** и попробуйте с помощью `Test Goal` выполнить следующее целевое утверждение:

```
member(susan,[ian,susan,john]).
```

2.Добавьте утверждения в разделы `domain` и `predicate` таким образом, чтобы вы могли использовать `member` для установки принадлежности числа числовому списку. Попробуйте несколько целевых утверждений, включая

```
member (X, [1,2, 3, 4]).
```

для проверки вашей новой программы.

3.Имеет ли значение порядок написания двух предложений для предиката `member`? Посмотрите, как ведет себя программа, если поменять местами два предложения. Обнаружится ли различие при выполнении целевого утверждения

```
member (X,[1,2,3,4,5])
```

для обоих случаев?

2.7 Объединение списков

В том виде, как предикат `member` дан в **программе ch07e06.pro**, он работает двумя способами. Рассмотрим его предложения еще раз:

```
member (Name,[Name | _ ]).
member (Name,[ _ | Tail]) :-
member(Name,Tail).
```

На эти предложения можно смотреть с двух различных точек зрения: декларативной и процедурной.

- С декларативной точки зрения предложения сообщают:

Name принадлежит списку, если голова совпадает с Name;

если нет, то Name принадлежит списку, если оно принадлежит его хвосту.

- С процедурной точки зрения эти два предложения можно трактовать так:

Чтобы найти элемент списка, надо найти его голову; иначе надо найти элемент в хвосте.

Эти две точки зрения соотносятся с целевым утверждением:

```
member(2,[1,2,3,4]).
member(X,[1,2,3,4]).
```

В результате, первое целевое утверждение "просит" Visual Prolog выяснить, *верно ли утверждение,* второе, — *найти всех членов списка* [1,2,3,4]. Предикат `member` одинаков в обоих случаях, но его поведение может быть рассмотрено с разных точек зрения.

2.8 Рекурсия с процедурной точки зрения

Особенность Пролога состоит в том, что часто, когда вы задаете предложения для предиката с одной точки зрения, они будут выполнены с другой. Чтобы увидеть эту двойственность, создадим в следующем примере предикат для присоединения одного списка к другому. Определим предикат append с тремя аргументами:

```
append(List1, List2, List3)
```

Он объединяет List1 и List2 и создает List3. Еще раз воспользуемся рекурсией (на этот раз с процедурной точки зрения).

Если List1 пустой, то результатом объединения List1 и List2 останется List2. На Прологе это будет выглядеть так:

```
append([],List2,List2).
```

Если List1 не пустой, то можно объединить List1 и List2 для формирования Lists, сделав голову List1 головой Lists. (В следующем утверждении переменная H используется как голова для List1 и для Lists.) Хвост List3 — это L3, он состоит из объединения остатка List1 (т. е. L1) и всего List2. То есть:

```
append ([H | L1],List2,[H|L3]) :-
append(LI,List2,L3).
```

Предикат append выполняется следующим образом: пока List1 не пустой, рекурсивное предложение передает по одному элементу в Lists. Когда List1 станет пустым, первое предложение унифицирует List2 с полученным Lists.

2.8.1 Упражнение

Предикат append определен в **программе ch07e07.pro**. Загрузите **программу ch07e07.pro**.

2.8.1.1 Листинг программы ch07e07.pro

```
domains
integerlist = integer*
predicates
append(integerlist, integerlist, integerlist)
clauses
append![], List, List).
append![H | L1], List2, [H | L3]) :-
append(LI, List2, L3).
```

Запустите следующее целевое утверждение:

```
append ([1,2,3],[5,6],L) .
```

А теперь попробуйте это:

```
append ([1,2],[3],L) ,append(L,L,LL) .
```

2.9 Количество вариантов использования предикатов

Рассматривая append с декларативной точки зрения, вы определили отношение между тремя списками. Однако это отношение сохранится, даже если List1 и List3 известны, а List2 — нет. Оно также справедливо, если известен только Lists. Например, чтобы определить, какие из двух списков можно объединить для получения известного списка, надо составить целевое утверждение такого вида:

```
append(Ll,L2,[1,2,4]) .
```

По этому целевому утверждению Visual Prolog найдет следующие решения:

```
Ll =[], L2=[1, 2, 4]
Ll=[1], L2=[2, 4]
Ll=[1, 2], L2=[4]
Ll=[1, 2, 4], L2=[]
4 Solutions
```

Можно также применить append, чтобы определить, какой список можно подсоединить к [3, 4] для получения списка [1, 2, 3, 4].

Запустите целевое утверждение

```
append (L1,[3,4],[1,2,3,4]).
```

Visual Prolog найдет решение:

```
L1=[1, 2].
```

Предикат append определил отношение между *входным набором* и *выходным набором* таким образом, что отношение применимо в обоих направлениях.

Состояние аргументов при вызове предиката называется *потоком параметров.* Аргумент, который присваивается или назначается в момент вызова, называется *входным аргументом* и обозначается буквой i; а свободный аргумент — это *выходной аргумент,* обозначается буквой o.

У предиката append есть возможность работать с разными потоками параметров, в зависимости от того, какие исходные данные вы ему дадите. Однако не для всех предикатов имеется возможность быть вызванными с различными потоками параметров. Если предложение Пролога может быть использовано с различными потоками параметров, оно называется *обратимым предложением.* Когда вы записываете собственные предложения в Visual Prolog, помните, что обратимые предложения имеют дополнительные преимущества, и их создание добавляет "мощности" предикатам.

2.9.1 Упражнение

Измените предложения, определяющие member в **программе ch07e06.pro**, и напишите предложения для предиката even_member, который будет *успешным,* если вы дадите целевое утверждение

```
even_member(2,[1, 2, 3, 4, 5, 6]).
```

Предикат должен также вернуть следующий результат:

```
X=2 X=4 X=6
3 Solutions,
```

если вы дадите ему:

```
even_member(X,[1,2,3,4,5,6]).
```

2.10 Поиск всех решений для цели сразу

Преимущество рекурсии состоит в том, что, в отличие от поиска с возвратом, она передает информацию (через параметры) от одного рекурсивного вызова к следующему. Поэтому рекурсивная процедура может хранить память о промежуточных результатах или счетчиках по мере того, как она выполняется.

Но есть одна вещь, которую поиск с возвратом *может* делать, а рекурсия — *нет.* Это поиск всех альтернативных решений в целевом утверждении. Может оказаться, что нужны все решения для целевого утверждения, и они необходимы все сразу, как часть единой сложной составной структуры данных. Встроенный предикат findall использует целевые утверждения в качестве одного из своих аргументов и собирает все решения для этого целевого утверждения в единый список. У предиката findall три аргумента:

- VarName (имя переменной) — определяет параметр, который необходимо собрать в список;

- `myPredicate` (мой предикат) — определяет предикат, из которого надо собрать значения;
- `ListParam` (список параметров) — содержит список значений, собранных методом поиска с возвратом. Заметьте, что должен быть определенный пользователем тип, которому принадлежат значения `ListParam`.

Программа ch07e08.pro использует `findall` для печати среднего возраста группы людей.

2.10.1 Листинг программы ch07e08.pro

```
domains
name, address = string
age = integer
list = age*
predicates
     person (name,address,age)
     sumlist (list,age,integer)
clauses
sumlist ([] ,0,0) .
sumlist([H | T],Sum,N) :-
sumlist(T,S1,N1),
Sum=H+S1,
N=1+N1.
person ("Sherlock Holmes", "22B Baker Street", 42). person("Pete Spiers",
"Apt. 22, 21st Street", 36).
person("Mary Darrow", "Suite 2, Omega Home", 51).
goal
findall(Age,person(_, _, Age),L),
sumlist(L,Sum,N),
Ave = Sum/N,
write("Average=", Ave),nl.
```

Предложение `findall` в этой программе создает список `L`, в котором собраны все возрасты, полученные из предиката `person`. Если бы вы захотели собрать список из всех людей, которым 42 года, то вам следовало бы выполнить следующее подцелевое утверждение:

```
findall(Who,person(Who,_,42),List)
```

Но эта подцель требует от программы, чтобы та содержала объявление домена для результирующего списка:

```
slist = string*
```

3 Составные списки

Список целых может быть объявлен просто:

```
integerlist = integer*
```

Это же справедливо и для списка действительных чисел, списка идентификаторов или списка строк.

Часто бывает важно иметь внутри одного списка комбинацию элементов, принадлежащих разным типам:

```
[2,3,5.12,["food","goo"],"new"]    % Некорректно в Visual  Prolog
```

Составные списки — это списки, в которых используется более чем один тип элементов. Для работы со списками из разнотипных элементов нужны специальные декларации, потому что Visual Prolog требует, чтобы *все элементы списка* принадлежали одному типу. Для создания списка, который мог бы хранить различные типы элементов, в Прологе необходимо использовать функторы, потому что домен *может содержать более одного типа данных* в качестве аргументов для функторов.

Пример объявления доменов для списка, который может содержать символы, целые, строки или списки:

```
Domains    % функторы l, i, c и s
Hist = l(list);i(integer);c(char);s(string)
list = llist*
```

Список

```
[2,9,["food","goo"],"new"]          % Некорректно в Visual Prolog
```

должен быть представлен в Visual Prolog как:

```
[i(2),i(9),l( [s ("food") , s("goo")]),s("new")] % Корректно.
```

В **программе ch07e09.pro** приведен пример , показывающий объединение списков и использование объявления доменов в типичном случае работы со списками.

3.1 Листинг программы ch07e09.pro

```
domains
llist = l(list); i (integer); c (char); s (string)
list = llist*
predicates
    append(list, list, list)
clauses
    append([] , L, L) .
append([X | L1], L2, [X | L3]):-
append(L1, L2, L3).
goal
append([s(likes),l([s(bill),s(mary)])],[s(bill),s(sue)],Ans),
write("First list:",Ans,"\n\n"),
append([1( [s("This") , s ("is") , s ("a") ,s("list") ] ) , s(bee) ], [c('c')],
Ans2),
write("Second list:",Ans2,'\n','\n').
```

3.2 Упражнения

1. Запишите предикат `oddlist`, который использует два аргумента. Первый аргумент — список целых, а второй — это список нечетных чисел, найденных в первом списке.

2. Запишите предикат `real_average`, который вычисляет среднее значение всех элементов списка действительных чисел.

3. Запишите предикат `flatten`, в котором первый аргумент — это составной список, а второй аргумент — список, из которого удалены все подсписки. Он выравнивает список из нескольких списков в один. Например, вызов

```
flatten([s(ed), i(3), l([r(3.9), 1([s(sally)])])], r(4.21),X)
```

даст результат

```
X=[s(ed), i(3),r(3.9),s(sally),r(4.21)]
1 Solution
```

который получается из первоначального списка после выравнивания.

4 Задания к лабораторной работе

4.1 Контрольные вопросы

1. Дайте определение понятию список.
2. Как называются составляющие списка?
3. Какой символ указывает на то, что создан список.
4. Из каких частей состоит список?
5. Какой существует способ в Visual Prolog для отделения головы от хвоста?
6. Чему равна длина любого списка?
7. Как называется состояние аргументов при вызове предиката?
8. Дайте определения понятиям: входной и выходной аргумент.
9. Какое предложение называется обратимым?
10. Что может делать поиск с возвратом в отличии от рекурсии и какой встроенный предикат служит для решения данной проблемы?
11. Дайте определение понятию составной список.
12. Для чего предназначены составные функторы?

4.2 Практические задания

```
DOMAINS
  list = integer*
PREDICATES
  write_a_list(list)
CLAUSES
  write_a_list([]).
  write_a_list([H|T]):-
    write(H),nl,
    write_a_list(T).
GOAL
  write_a_list([9, 3, 6, 7, 1, 2, 5]).
```

На основе листинга представленного выше выполнить следующие задания:

Варианты заданий:

1. Вывести максимальный элемент списка
2. Вывести минимальный элемент списка
3. Вывести среднее арифметическое элементов списка