3.3.2. Списки

Список – последовательный набор любого количества связанных объектов, как правило, одного и того же доменного типа.

Каждая составляющая списка называется *элементом*.

Длина списка равна количеству элементов в списке.

Для того чтобы оформить списочную структуру, надо отделить элементы списка *запятыми* и заключить их в *квадратные скобки*:

[1, 2, 3]
[smalltalk, ada, eifell, java]
["лекция", "лабораторная работа", "зачёт", "экзамен", "каникулы"]
[каникулы]
[day (10, 2, 2009), day (23, 5, 2009)]
[[1, 2, 3], [4, 5]]
[ ]

Элементами списка могут быть простые объекты и структуры, а также другие списки. Обязательное и единственное условие – одинаковый доменный тип элементов.

Список может содержать только один элемент и даже совсем не содержать элементов *(пустые квадратные скобки)*.

Не содержащий элементов список называется пустым (нулевым) списком.

При построении списков, имеющих рекурсивную природу, необходимо иметь некоторую константу, чтобы рекурсия не была бесконечной. Такой константой является пустой список, который обозначается, как **[ ]**.

Следующие два атрибута списка наглядно показывают его рекурсивную природу. Непустой список состоит из двух частей: голова списка и хвост.

Голова *– отдельное, неделимое значение, то есть* ***всегда элемент.***

Хвост *– это* ***всегда список****, составленный из того, что осталось от исходного списка в результате отделения головы.*

Новый список зачастую можно делить и дальше. Если список состоит из одного элемента, то его можно разделить на голову, которой является этот единственный элемент, и хвост, являющийся пустым списком. *Пустой список нельзя разделить на голову и хвост*. Следовательно, если выбирать первый элемент списка достаточное число раз, то обязательно получится пустой список.

[a, b, c, d]
голова → a хвост → [b, c, d]
голова → b хвост → [c, d]
голова → c хвост → [d]
голова → d хвост → [ ]

Операция деления списка на голову и хвост обозначается с помощью *вертикальной черты*:

[Head | Tail]

, где **Head** – переменная для обозначения головы списка,
Tail – переменная для обозначения хвоста списка.

На местах **Head** и **Tail** могут быть любые пригодные допустимые имена переменных *Пролога*.

Пример использования разделителя *вертикальная черта*:

[a, b, c] ≡ [a | [b, c]] ≡ [a | [b | [c | [ ]]]]

Допускается использование обоих разделителей (*вертикальная черта* и *запятая*) вместе:

[a, b, c, d] ≡ [a, b | [c, d]]

Список объявляется в секции **domains**. Для объявления списка в программе используется символ ***** *(звёздочка)*, который означает список чего-нибудь, и ставится после типа элементов списка.

Таким образом, **integer*** объявляет список целых. Именно обозначение *****, а не название говорит о том, что имеется в виду список.

Для объявления списка, составленного из элементов разного типа (например, действительных, целых и символьных элементов), надо определить один тип, включающий все три типа с функторами, которые покажут, к какому типу относится тот или иной элемент.

Пример 3-10. Объявление списка однотипных элементов.

```
domains    list = elements*
           elements = string
predicates db (list)
clauses
        db (["FoxPro", "MS Access", "Paradox", "Oracle", "dbVista"]).
goal    db (X).
```

Решение:

X = ["dBase", "FoxPro", "MS Access", "Paradox", "Oracle", "dbVista"]

Пример 3-11. Объявление списка разнотипных элементов.

```
domains    elementlist = elements*
           elements = r (real);
           i (integer);
           s (string)
predicates list (elementlist)
clauses    list ([i (5), i (17), r (7.5), s ("bye")]).
goal       list (List).
```

Решение:

List = [i (5), i (17), r (7.5), s ("bye")]

3.3.3. Процедуры обработки списка

Процедуры работы со списками сводятся к поэлементной обработке путём разбиения исходного списка на голову и хвост.

Следует помнить, что объём памяти, необходимой для решения рекурсивной процедуры, зависит от числа выполняемых рекурсивных обращений.

Пример 3-12. Вывод элементов списка.

```
domains    list = integer*
predicates wList (list)
goal       wList ([1, 2, 3]), exit.
clauses    wList ([ ]).              % 1
           wList ([H | T]) :-        % 2
                 write (H),
                 write (' '),
                 wList (T).
```

Процесс согласования цели начинается с решения правила %2 процедуры **wList**, так как с правилом %1 цель не согласуется (список непустой).

Согласно заголовку правила %2, список **[1, 2, 3]** делится на голову **1** и хвост **[2, 3]**.

Выводится голова списка и пробел.

Происходит рекурсивный вызов с аргументом, конкретизированным хвостом **[2, 3]**.

Когда переменная **T** конкретизируется значением пустого списка, согласуется правило %1.

Возврат из рекурсии происходит согласно заголовку правила %2, в результате чего список приобретает первоначальное состояние.

Пример 3-13. Вычисление длины без использования итерации.

Декларативный смысл: размер пустого списка равен нулю, а обычного (непустого) – размеру его хвоста плюс единица.

```
domains    list = symbol*
predicates listSize (list, integer)
goal       listSize ([a, b, c], X), write (X), exit.
clauses    listSize ([ ], 0).              % 1
           listSize ([_ | T], L) :-        % 2
                 listSize (T, TL),
                 L = 1 + TL.
```

Процесс согласования цели начинается с решения правила %2, так как список непустой.

Правило %1процедуры **listSize** утверждает то, что размер пустого списка равен нулю.

Правило %2 пытается определить размер хвоста **listSize (T, TL)**.

Определение числа элементов начнётся только после конкретизации переменной **TL** нулевым значением в правиле %1 при согласовании цели **listSize ([ ], TL)** из правила %2.

Согласно заголовку правила %2 к пустому списку присоединяются отделённые ранее головы (восстанавливая список), и переменная **L** конкретизируется значением числа элементов *на выходе* из рекурсии.

Пример 3-14. Вычисление длины с использованием итерации.

```
domains     list = integer*
predicates  lenList (list, integer, integer)
clauses     lenList ([ ], Result, Result).                % 1
            lenList ([_ | T], Result, Counter) :-         % 2
                  Counter1 = Counter + 1,
                  lenList (T, Result, Counter1).
goal        lenList ([1, 2, 3], X, 0), write (X), exit.
```

Правило %1 процедуры **lenList** утверждает, что размер списка инициализируется значением счётчика количества отделений головы в тот момент, когда исходный список станет пустым.

Правило %2 рекурсивно подсчитывает число отделений головы от хвоста в переменной **Counter1**.

После согласования правила %1 происходит восстановление исходного списка согласно заголовку правила %2.

При вызове процедуры необходимо задать *нулевое значение* счётчика: **lenList ([1, 2, 3], X, *0*)**.

Пример 3-15. Модификация списка.

Декларативный смысл: каждый элемент выходного списка равен увеличенному на единицу значению элемента входного списка.

```
domains     list = integer*
predicates  modList (list, list)
clauses     modList ([ ], [ ]).                           % 1
            modList ([H | T], [Hnew | Tnew]) :-           % 2
                  Hnew = H + 1,
                  modList (T, Tnew).
goal        modList ([1, 2, 3], ModList),
            write (ModList), exit.
```

Правило %1процедуры **modList** утверждает, что выходной список становится пустым тогда, когда входной список пустой.

Правило %2 определяет голову выходного списка **Hnew = H + 1**, и исследует хвост входного списка **modList (T, Tnew)** до тех пор, пока он не станет пустым.

После согласования правила %1 происходит восстановление обоих списков согласно заголовку правила %2.

Пример 3-16. Фильтрация списка.

Декларативный смысл: выходной список содержит только положительные элементы входного списка.

```
domains    list = integer*
predicates
           nondeterm disNeg (list, list)
clauses
           disNeg ([ ], [ ]).                    % 1
           disNeg ([H | T], NewList) :-          % 2
                 H <= 0,
                 disNeg (T, NewList).
           disNeg ([H | T], [H | NewList]) :-    % 3
                 disNeg (T, NewList).
goal       disNeg ([2, -45, -3, 468], X ), write (X),
           exit.
```

Предложенная версия программы корректна только при наличии предиката **exit** в целевом утверждении, который обеспечивает единственное решение для процедуры **disNeg**.

Предикат **disNeg** имеет два аргумента: входной список и выходной (для сохранения результата преобразования).

Правило % 1 обеспечивает завершение рекурсии: когда входной список становится пустым, выходной конкретизируется пустым списком.

Правило % 2 проверяет, является ли голова списка отрицательным элементом. Если это так, то выходной список остаётся без изменения и рекурсивно обрабатывается хвост входного списка.

Правило %3 согласуется тогда, когда не согласуется правило %1 и правило %2. По заголовку правила отделённая от хвоста голова выходного списка конкретизируется положительным значением головы входного списка. Тело правило содержит рекурсивный вызов с конкретизированным хвостом первого списка и с неопределённым хвостом второго списка.

Возврат из рекурсии происходит в соответствии с заголовками правил, которые согласовывались в процессе поиска решения.

Пример 3-17. Дублирование элементов списка.

Декларативный смысл: в выходной список каждый элемент входного списка помещается дважды.

```
domains     list = char*
predicates  double ( list, list )
clauses     double ([ ], [ ]).                    % 1
            double ([H | T], [H, H | DT]) :-      % 2
                  double (T, DT).
goal        double ( ['a', 'b', 'c', 'd'], DList ).
```

Решение: **DList = ['a', 'a', 'b', 'b', 'c', 'c', 'd', 'd']**

На выходе из рекурсии согласно заголовку правила %2 голова входного списка будет присоединяться к выходному списку два раза, благодаря разделителям *запятая* и *вертикальная черта* **[H, H | DT]**.

Пример 3-18. Принадлежность элемента списку.

Декларативный смысл: элемент принадлежит списку, если элемент является первым в списке или принадлежит хвосту списка.

```
domains     list = name*      name = symbol
predicates  nondeterm member (name, list)
clauses     member (Name, [Name | _]).            % 1
            member (Name, [_ | Tail ]) :-         % 2
                  member (Name, Tail).
```

Правило %1 разделяет входной список на голову и анонимный хвост с одновременным согласованием искомого элемента с головой. При их идентичности правило возвращает истину.

Правило %2 доказывается в случае неудачного согласования правила %1. Оно разделяет список на анонимную голову, так как она уже не имеет значения, и хвост, который передаётся для нового согласования.

Версия программы, показанная в примере, представляет *недетерминированный* предикат **member**.

Ниже рассматриваются результаты работы программы для разных целей:

1. Список состоит из уникальных элементов, и искомое значение присутствует в списке.

   ```
   goal      member (ann, [tom, bill, ann, susan]).
   yes
   ```

2. Список состоит из уникальных элементов, и искомого значения нет в списке.

   ```
   goal      member ( eve, [tom, bill, ann, susan]).
   no
   ```

3. Список содержит элементы, значения которых повторяются, и искомое значение имеется в списке.

```
goal    X = tom, member (X, [tom, bill, ann, tom]).
X = tom, X = tom
```

4. Требуется узнать, из каких элементов состоит список.

```
goal        member (X, [tom, bill, ann, susan]).
X = tom, X = bill, X = ann, X = susan
```

Множественные решения, полученные для целей в пунктах 3 и 4, объясняются автоматическим перебором *Пролог*-системы, который пытается найти **все** возможные решения. *Детерминированное* решение предиката для этих целей будет рассмотрено в примере 5-2.

Пример 3-19. Ввод элементов списка с подтверждением.

```
domains     list = integer*
predicates  nondeterm rList (char, list)
clauses     rList ('n', [ ]).                          % 1
            rList (_, [H | T] ) :-                     % 2
                  nl, write ("Введите элемент списка: "),
                  readint (H),
                  write ("Продолжать ввод? (_/n)"),
                  readchar (R),
                  rList (R, T).
goal        rList ('y', S),
            write ("Введён список:"),
            nl, write (S), exit.
```

Согласование цели **rList ('y', S)** начинается с правила %2, так как **'y' <> 'n'**. Введённые значения **H** последовательно помещаются в стек до тех пор, пока переменная **R** не конкретизируется значением **n**.

Становится возможным согласование правила %1, по которому список становится пустым.

Начинается возврат из рекурсии. Элементы извлекаются из стека и присоединяются как головы согласно заголовку правила %2.

Вариант детерминированной процедуры ввода списка представлен в [29].

Пример 3-20. Объединение списков.

Декларативный смысл: Если первый список пустой, то результирующий список станет равным второму списку. Если первый список не пустой, то можно объединить первый и второй список, сделав голову первого списка головой результирующего списка. Хвост результирующего списка состоит из объединения остатка первого списка и всего второго списка.

Программа работает согласно формуле **List3 = List1 + List2**.

```
domains     list = integer*
predicates  nondeterm append ( list, list, list )
clauses     append ([ ], List, List).                    % 1
            append ([H | List1], List2, [H | List3]) :–  % 2
                  append (List1, List2, List3).
```

Далее рассматриваются варианты использования предиката для разных потоков данных:

1. Объединение **List1** и **List2** в **List3**: *(input, input, output)*.

   goal append ([1, 2, 3], [4, 5], L).
   L = [1, 2, 3, 4, 5]

   Список **List3** пока не определён: **append ([1, 2, 3], [4, 5], _)**.

   Цепочка рекурсий согласно правилу %2 раскручивается до тех пор, пока не обнуляется **List1**, элементы списка при этом последовательно пересылаются в стек.

   Когда **List1** становится пустым, становится возможным применение правила %1, по которому **List3** инициализируется списком **List2**:

   append ([ ], [4, 5], _)
   append ([ ], [4, 5], [4, 5])

   Правило %1 полностью удовлетворено.

   Начинается сворачивание рекурсивных вызовов правила %2: извлекаемые из стека элементы помещаются один за другим в качестве головы одновременно к **List1** и **List3**:

   append ([ ], [4, 5], [4, 5])
   append ([3], [4, 5], [3, 4, 5])
   append ([2, 3], [4, 5], [2, 3, 4, 5])
   append ([1, 2, 3], [4, 5], [1, 2, 3, 4, 5])

2. Определение **List1** как разности **List3** и **List2**: *(output, input, input)*.

   goal append (L, [4, 5], [1, 2, 3, 4, 5]).
   L = [1, 2, 3]

   Цепочка рекурсий согласно правилу %2 раскручивается до тех пор, пока список **List2** не станет равным списку **List3**, элементы списка **List3** при этом последовательно пересылаются в стек.

   append (_, [4, 5], [2, 3, 4, 5]) *1 → в стек*
   append (_, [4, 5], [3, 4, 5]) *2 → в стек*
   append (_, [4, 5], [4, 5]) *3 → в стек*

   Правило %1 удовлетворено, потому что списки **List2** и **List3** равны.

   Сворачивание рекурсии:

   append ([3], [4, 5], [3, 4, 5])

append ([2, 3], [4, 5], [2, 3, 4, 5])
append ([1, 2, 3], [4, 5], [1, 2, 3, 4, 5])

3. Определение **List2** как разности **List3** и **List1**: *(input, output, input)*.

goal append ([1, 2], L, [1, 2, 3, 4, 5]).
L = [3, 4, 5]

Цепочка рекурсий согласно правилу %2 раскручивается до тех пор, пока не обнуляется **List1**, элементы при этом последовательно пересылаются в стек.

append ([2], _, [2, 3, 4, 5]) *1 → в стек*
append ([ ], _, [3, 4, 5]) *2 → в стек*

Правило %1 удовлетворено, так как **List1** теперь пустой список.

Согласно правилу %1 **List2** инициализируется списком **List3**:

append ([ ], [3, 4, 5], [3, 4, 5])

Сворачивание рекурсии:

append ([2], [3, 4, 5], [2, 3, 4, 5])
append ([1, 2], [3, 4, 5], [1, 2, 3, 4, 5])

4. Определение возможных сочетаний **List1** и **List2** для получения **List3** *(output, output, input)*.

goal append (X, Y, [1, 2, 3, 4, 5]).
X = [ ] Y = [1, 2, 3, 4, 5]
X = [1] Y = [2, 3, 4, 5]
X = [1, 2] Y = [3, 4, 5]
X = [1, 2, 3] Y = [4, 5]
X = [1, 2, 3, 4] Y = [5]
X = [1, 2, 3, 4, 5] Y = [ ]
6 Solutions

Первое решение доказывается правилом %1. Все остальные решения получаются отделением головы списка **List3** по правилу %2, согласованием правила %1 и сворачиванием рекурсии. В результате автоматического перебора *Пролог*-система найдёт шесть решений.

5. Проверка того, что **List3** является объединением списков **List1** и **List2** *(input, input, input)*.

goal append ([1, 2, 3], [3, 4, 5], [1, 2, 3, 4, 5]).
no

Цепочка рекурсий согласно правилу %2 раскручивается до тех пор, пока не обнуляется **List1** и пока головы списков **List1** и **List3** совпадают.

По правилу %2 остаток **List3** должен быть равен списку **List2**. Если это истинно, будет получен ответ **yes**. Для приведённого целевого утверждения **[3, 4, 5] <> [4, 5]**, поэтому ответ **no**.

После согласования правила %1 рекурсия сворачивается, и списки приобретают исходное состояние.

Пример 3-21. Реверс списка.

Декларативный смысл: В выходной список элементы входного списка помещаются в обратном порядке.

```
domains    list = integer*
predicates reverse (list, list)                 % реверс списка
           append (list, list, list) - (i, i, o) % объединение списков
clauses
           % объединение списков
           append ([ ], List, List).                          % 1-a
           append ([H|List1], List2, [H|List3]) :-            % 2-a
                 append (List1, List2, List3).

           % реверс списка
           reverse ([ ], [ ]).                                % 1-r
           reverse ([H|Tail], ReverseList) :-                 % 2-r
                 reverse (Tail, ReverseTail),
                 append (ReverseTail, [H], ReverseList).
goal       reverse ([1, 2, 3], List).
```

Решение: **List = [3, 2, 1]**

Предикат **append** объявлен только для потока данных *(input, input, output)*, поэтому является детерминированным.

Правило %1-r процедуры **reverse** утверждает, что выходной список пустой, если входной тоже пустой.

По правилу %2-r рекурсивно происходит реверс хвоста списка, при этом головы входного списка последовательно помещаются в стек:

reverse ([2, 3], _) *1 → в стек*
reverse ([3], _) *2 → в стек*
reverse ([ ], _) *3 → в стек*

Согласуется правило %1-r и список **ReverseTail** становится пустым **[ ]**.

Становится возможным согласование цели **append (ReverseTail, [H], ReverseList)**. Результатом согласования будет объединение конкретизированного хвоста **ReverseTail** со списком, который состоит из одного элемента **[H]**, расположенного в стеке.

Согласование продолжается до тех пор, пока стек не станет пустым:

%2-r **append ([ ], [3], _)**

%1-a **append ([ ], [3], [3])** **ReverseList = [3]**
%2-a **append ([3], [2], _)**
%1-a **append ([ ], [2], [2])**

Восстановление списков согласно заголовку правила %2-а:

append ([3], [2], [3, 2]) **ReverseList = [3,2]**

%2-a **append ([3, 2], [1], _)**
%2-a **append ([2], [1], _)**
%1-a **append ([ ], [1], [1])**

Восстановление списков согласно заголовку правила %2-а:

append ([2], [1], [2,1])
append ([3, 2], [1], [3, 2, 1]) **ReverseList = [3, 2, 1]**