Списки в Prolog. Синтаксис

При использовании статически-типизированного диалекта необходимо объявить тип списка (например в Turbo Prolog это делается в разделе domains):

[?](http://pro-prof.com/archives/845)

|  |  |
| --- | --- |
| объявление типа данных списка | |
| 1  2 | domains    ilist = integer\* |

В дальнейшем, при объявлении функций обработки списков (раздел predicates) необходимо использовать объявленный тип данных. Функция обработки списков вещественных чисел не будет обрабатывать список целых – это не очень удобно, однако позволяет выявлять несоответствие типов на этапе трансляции и генерировать более оптимальный код компилятору.

При использовании таких диалектов как SWI Prolog, предварительное объявление типов не требуется, кроме того, список может содержать данные разных типов.

[?](http://pro-prof.com/archives/845)

|  |  |
| --- | --- |
| синтаксис обработки списков в Prolog | |
| 01  02  03  04  05  06  07  08  09  10  11  12  13 | list\_syntax:-    % объявление списка из трех чисел с именем ListA:    ListA = [7, 5, 3],    % разделение списка на первый элемент (переменная с именем Head)    % и остальную часть списка (Tail)    % в результате Head = 7, Tail = [5, 3]:    ListA = [Head|Tail],    % формирование списка ListB из нового элемента, Head и tail:    % в результате ListB = [7, 9, 5, 3]    ListB = [Head, 9|Tail],    % сравнение списка ListA с пустым списком    % (завершится неудачей, т.к. ListA не пуст)    ListA = []. |

Приведенный выше пример показывает рекурсивную природу списков – после отделения головы (первого элемента) остается хвост, также являющийся списком. Операция разделения на голову и хвост может использоваться для формирования нового списка. В языке имеется обозначение для пустого списка и оно постоянно используется в рекурсивных функциях для фиксации окончания обработки.

Рекурсивная обработка списков. Примеры

**sum\_list(List, Sum)** – вычисление суммы элементов списка

Сумма элементов пустого списка равна нулю. Если список не пуст – разделим его на первый элемент и хвост. Обработаем рекурсивно хвост, в результате получим сумму элементов части списка без учета первого элемента. Добавим первый элемент чтобы получить окончательный результат.

[?](http://pro-prof.com/archives/845)

|  |  |
| --- | --- |
| 1  2  3  4 | sum\_list([], 0):-!.  sum\_list([Head|Tail], Sum):-    sum\_list(Tail, TailSum),    Sum is TailSum + Head. |

**nth0(Index, List, Elem)** – функция получения элемента списка с заданным индексом. Индексация начинается с нуля. Если нужного элемента нет – функция завершается неудачей.

[?](http://pro-prof.com/archives/845)

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | nth0(0, [Elem|\_Tail], Elem):-!.  nth0(Index, \_List, \_Elem):-    Index < 0, !, fail.  nth0(Index, [\_Head|Tail], Elem):-    NextIndex is Index - 1,    nth0(NextIndex, Tail, Elem). |

Функция завершает свою и успешно возвращает первый элемент списка если Index равен нулю и от списка удалось отделить первый элемент. Функция завершается неудачей если Idex оказался меньше нуля или остался больше нуля при пустом списке. Если же список не пуст (от него успешно отделяется первый элемент) и индекс больше ноля – рекурсивно обрабатывается хвост списка и уменьшенный на единицу Index (ведь список уменьшился на один элемент).

**member(Elem, List)**– выполняет поиск значения в списке. Завершается удачей если элемент найден.

Если значение первого элемента списка совпадает со значением искомого элемента – правило сразу завершается удачей. В противном случае первый элемент отбрасывается и поиск продолжается в хвосте. Как только разделение списка на голову и хвост станет невозможно – правило завершится неудачей.

[?](http://pro-prof.com/archives/845)

|  |  |
| --- | --- |
| 1  2  3 | member(Elem, [Elem|\_Tail]).  member(Elem, [\_Head|Tail]):-    member(Elem, Tail). |

Важно то, что предикат может вернуть несколько решений (предикат является недетерминированным – *nondeterm*), поэтому первое правило не содержит отсечения. Такое поведение особенно важно если *функция member*используется для перебора всех элементов списка (достаточно передать вместо Elem анонимную переменную).

**min\_list(List,MinElem)**– вычисление наименьшего элемента списка

Решение очевидно, если список состоит из одного элемента. Если элементов больше, то список разделяется на голову и хвост. Рекурсивно вычисляется наименьший элемент хвоста и сравнивается с первым элементом для определения результата для всего списка.

[?](http://pro-prof.com/archives/845)

|  |  |
| --- | --- |
| 1  2  3  4  5 | min\_list([MinElem], MinElem):-!.  min\_list([Head|Tail], MinElem):-    min\_list(Tail, TailMinElem),    TailMinElem < Head, !, MinElem = TailMinElem;    MinElem = Head. |

Если на вход будет подан пустой список, то разделение на голову и хвост провалится и правило завершится неудачей.

**reverse(List, ReverseList)** – функция переворота списка

Чтобы перевернуть список используется *метод накапливающего параметра*, при этом создается вспомогательная функция, которая помимо двух списков принимает буфер. Изначально буфер пуст, но по мере обработки, в его начало добавляются элементы из исходного списка – поэтому первый добавленный элемент окажется последним, а последний – первым. Как только все элементы окажутся обработаны – накопленные в буфере значения будут переписаны в список-результат.

[?](http://pro-prof.com/archives/845)

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | reverse(List, ReverseList):-    reverse(List, [], ReverseList). % вызов вспомогательной функции с пустым буфером    reverse([], Buffer, Buffer):-!.  reverse([Head|Tail], Buffer, ReverseList):-    reverse(Tail, [Head|Buffer], ReverseList). |

Метод накапливающего параметра используется достаточно часто, т.к. при его использовании рекурсивный вызов оказывается последней операцией, выполняемой функцией. Компиляторы заменяют такой вызов циклом, что позволяет обрабатывать огромные списки и не бояться переполнения стека.

**sublist(Sub, List)** – завершается удачей если все элементы списка Sub встречаются в списке List в точно таком же порядке.

Напишем вспомогательную функцию sub\_start для проверки случая, когда список List начинается со списка Sub. Очевидно, проверка вспомогательного правила даст условие выхода из рекурсии в случае успешного завершения работы. Если же проверка не увенчалась успехом – отделим от List первый элемент, а остальные попробуем обработать рекурсивно.

[?](http://pro-prof.com/archives/845)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | sub\_start([], \_List):-!.  sub\_start([Head|TailSub], [Head|TailList]):-    sub\_start(TailSub, TailList).    sublist(Sub, List):-    sub\_start(Sub, List), !.  sublist(Sub, [\_Head|Tail]):-    sublist(Sub, Tail). |

Функция sub\_start завершается успехом если список Sub оказался пуст, т.к. пустой список по определению является часть любого списка.

**delete(InputList, Elem, ResultList)**– функция удаления всех элементов с заданным значением из списка

Если на вход подан пустой список, то результатом должен также являться пустой список. Если первый элемент списка совпал с удаляемым значением – функция должна вернуть результат рекурсивной обработки хвоста, а в противном случае – добавить к полученному результату первый элемент.

[?](http://pro-prof.com/archives/845)

|  |  |
| --- | --- |
| 1  2  3  4  5 | delete([], \_Elem, []):-!.  delete([Elem|Tail], Elem, ResultTail):-    delete(Tail, Elem, ResultTail), !.  delete([Head|Tail], Elem, [Head|ResultTail]):-    delete(Tail, Elem, ResultTail). |

**append(List1, List2, List1AndList2)**– функция объединения двух списков

Постепенно выбирая элементы первого списка сводим ситуацию к тому, что список станет пуст – в этом случае результатом работы функции должен являться второй список. На рекурсивном подъеме добавим к полученному результату элементы первого списка.

[?](http://pro-prof.com/archives/845)

|  |  |
| --- | --- |
| 1  2  3 | append([], List2, List2):-!.  append([Head|Tail], List2, [Head|TailResult]):-    append(Tail, List2, TailResult). |

**unique(List)** – проверка того, что ни один элемент списка не повторяется дважды

В списке нет повторяющихся элементов, если первый элемент списка не встречается в хвосте, а также, в хвосте нет повторяющихся элементов. Условием выхода из рекурсии может быть пустота исходного списка – в нем гарантированно нет повторяющихся элементов.

[?](http://pro-prof.com/archives/845)

|  |  |
| --- | --- |
| 1  2  3  4 | unique([]):-!.  unique([Head|Tail]):-    member(Head, Tail), !, fail;    unique(Tail). |

**rangConcat(List1, List2, List1AndList2)**– объединение двух отсортированных списков так, чтобы в результате получился отсортированный список.

Очевидно, что если один из списков пуст – результатом является другой список. На каждом шаге алгоритма оба входных списка разделяются на голову и хвост. Головы сравниваются и рекурсивно обрабатываются списки без наименьшей головы, на рекурсивном подъеме она добавляется к результату.

[?](http://pro-prof.com/archives/845)

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | rangConcat([], List2, List2):-!.  rangConcat(List1, [], List1):-!.  rangConcat([Head1|Tail1], [Head2|Tail2], [Head1|TailResult]):-    Head1 < Head2, !, rangConcat(Tail1, [Head2|Tail2], TailResult).  rangConcat(List1, [Head2|Tail2], [Head2|TailResult]):-    rangConcat(List1, Tail2, TailResult). |

**length(List, Length)** – вычисляет длину списка

Используем метод накапливающего параметра чтобы рекурсия функции была хвостовой, т.к. длина списков на языке Prolog вычисляется очень часто поэтому эффективность функции является критичной.

Функция length вызывает вспомогательную, при этом задает начальное значение длины равным нулю.

[?](http://pro-prof.com/archives/845)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | length(List, Length):-    length(List, 0, Length).    length([], Length, Length):-!.  length([\_Head|Tail], Buffer, Length):-    NewBuffer is Buffer + 1,    length(Tail, NewBuffer, Length). |

Вспомогательная функция возвращает накопленное в буфере значение в качестве длины если на вход подан пустой список. В противном случае от исходного списка отделяется один элемент, вычисляется новое значение буфера. Хвост списка и буфер передаются для рекурсивной обработки.