Тема 3.2 Работа со списками и бинарными деревьями

Список – последовательность логических термов, перечисленных через запятую и заключенных в квадратные скобки.

Пустой список – открывающая и закрывающая квадратные скобки без элементов внутри.

Примеры списков: [1, house, f(4)], [], [a, b, c].

Список делится на две части: **голову** и **хвост**.

Голова – первый(ые) элемент(ы) списка, хвост – остаток списка.

Для деления на голову и хвост используется вертикальная черта.

Пример деления:

Список	Голова	Хвост
`[a | b, c]`	`a`	`[b, c]`
`[a, b | c]`	`a, b`	`[c]`
`[a, b, c |]`	`a, b, c`	`[]`

`[a, b, c]` ⇔ `[a | [b, c]]` ⇔ `[a, b | [c]]` ⇔ `[a, b, c |]`

Пустой список нельзя разделить на голову и хвост!

Хвост – всегда список!

Список – структура языка Пролог, которая может быть записана следующим образом:

.(Голова, Хвост)

[a, b, c] ⇔ .(a, .(b, .(c, [])))

[a] ⇔ .(a, [])

Но это академические знания, т.к. GNU Prolog их не понимает.

Рассмотрим полезный пример, который он понимает – «Вхождение элемента в список».

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

Программа может быть прочитана следующим образом: «Элемент содержится в списке, если он находится в голове списка или в его хвосте».

Что ответит Пролог на следующие запросы? Выполните трассировку:

```
?- member(a, [b, a, c]).
?- member(a, [b, a, a]).
?- member(a, [b, c, X]).
?- member(X, [a, b, c]).
```

Программа «объединения двух списков» conc(List1, List2, ResultList):

```
?- conc([a, b], [c, d], [a, b, c, d]).
yes
```

Решение:

```
1. conc([], L, L).
2. conc([Head | Tail], L, [Head | NewTail]) :- conc(Tail, L,
NewTail).
```

Обратите внимание, что вторая строчка программы может быть переписана следующим образом:

```
2. conc([Head | Tail], L, ResultList) :- conc(Tail, L, NewTail), ResultList = [Head | NewTail].
```

Выполните трассировку программы для следующих запросов:

```
?- conc([a, b], [c], Res).
?- conc(X, [c], [a, b, c]).
?- conc(X, Y, [a, b, c]).
```

Как можно записать программу **member** с использованием **conc** (в одну строчку)? Какой вариант будет быстрее работать?

Как можно написать программу добавления элемента в список: **add(Item, SourceList, DestList)**?

Напишите программу удаления элемента из списка: **del(Item, SourceList, DestList)**.

Проверьте функцию del на запрос

```
?- del(X, [red, green, blue], L).
```

Напишите программу удаления всех вхождений элемента в список: **delAll(Item, SourceList, DestList)**.

Необходимо написать программу генерации перестановок, которая при возврате и повторном передоказательстве должна генерировать все возможные перестановки элементов списка. Решение:

```
permutation([], []).
permutation(L, [X | P]) :- del(X, L, L1), permutation(L1, P).
```

Как это работает. База индукции: перестановка пустого списка – это пустой список. Индукционный переход: из списка L удаляется элемент (функция del начинает с удаления первого элемента и затем начинает удалять их последовательно один за другим), который затем помещается в голову перемешанного списка. Перемешивание начинается с конца списка.

```
?- permutation([red, green, blue] , P).
```

Задача: есть список чисел. Необходимо получить этот список в упорядоченном по возрастанию виде. Решение:

```
ordered([]).
ordered([_]).
ordered([X, Y | T]) :- X < Y, ordered([Y|T]).
```

Пустой список и список из одного элемента – упорядочены. Если в голове списка есть два элемента, то список упорядочен, если первый элемент меньше второго, а список из второго и хвоста так же упорядочен.

Теперь задача сортировки решается в одну строчку:

```
sort(SourceList, DestList) :- permutation(SourceList, DestList), ordered(DestList).
```

У метода сортировки два параметра: первый – исходный список, второй – упорядоченный. Сортировка считается завершенной, если после перестановки элементов в исходном списке получился упорядоченный по возрастанию список.

Усовершенствуем программу «обезьяна и банан», добавив вычисление пути, по которому обезьяна добралась до банана.

```
1. start(1, 1).
2. stop(4, 4).
3. go(Path):- start(X, Y), move(X, Y, [], Path).
4. move(X, Y, P, [m(X,Y)| P]) :- stop(X, Y).
5. move(X, Y, From, To) :- X < 5, X1 is X + 1, move(X1, Y,
[m(X,Y) | From], To).
6. move(X, Y, From, To) :- Y < 5, Y1 is Y + 1, move(X, Y1,
[m(X,Y) | From], To).
```

Для проверки корректности программы выполним к ней запрос:

```
?- go(P).
P = [m(4,4),m(4,3),m(4,2),m(4,1),m(3,1),m(2,1),m(1,1)] ? ;
P = [m(4,4),m(4,3),m(4,2),m(3,2),m(3,1),m(2,1),m(1,1)] ? ;
P = [m(4,4),m(4,3),m(3,3),m(3,2),m(3,1),m(2,1),m(1,1)] ?
yes
```

В данном примере программа передоказывалась дважды после первого успешного доказательства.

Почему это работает: в предикат move добавлены два параметра. Первый добавленный параметр инициализируется пустым списком и накапливает каждый вновь выполненный шаг, второй добавленный параметр – рекурсивно возвращает результат после того, как мы добрались до банана. Функтор m(X, Y) просто хранит координаты точки, в которую мы пришли.

Обратите внимание, что путь выдаётся в обратном порядке. Как можно сделать путь в правильном порядке?

```
1. start(1, 1).
2. stop(4, 4).
3. go(Path):- start(X, Y), move(X, Y, [], Path).
4. move(X, Y, P, [m(X,Y)|P]) :- stop(X, Y).
5. move(X, Y, From, [m(X,Y) | To]) :- X < 5, X1 is X + 1,
move(X1, Y, From, To).
6. move(X, Y, From, [m(X,Y) | To]) :- Y < 5, Y1 is Y + 1,
move(X, Y1, From, To).
```

И тогда результат запроса будет другой:

```
?- go(P).
P = [m(1,1),m(2,1),m(3,1),m(4,1),m(4,2),m(4,3),m(4,4)]
yes
```

Обратите внимание, чем отличаются эти программы.

В первом случае мы сохраняем используемые координаты в процессе движения вглубь рекурсии, во втором случае мы делаем то же самое, но при возврате из рекурсии.

На всякий случай, для тех кто всё ещё не понял. Переписываю строчку 4 последнего примера:

```
4. move(X, Y, P, PReturn) :- stop(X, Y), PReturn = [m(X,Y)|P].
```

Обе строчки идентичны. Просто в данном случае я ввел дополнительную переменную PReturn, появления которой в данном случае можно избежать.

Бинарное дерево – дерево, имеющее максимум две ветки в каждом узле.

Варианты задания бинарных деревьев на Прологе:

1. `a(b(d), c(e, f)).`

2. Вводим понятие `nil` – пустое поддерево. Обозначим дерево предикатом `btree` с тремя параметрами: первый – корень, второй – левое поддерево, третий – правое поддерево.

```
btree(a, btree(b, btree(d, nil, nil), nil), btree(c, btree(e, nil, nil), btree(f, nil, nil)))
```

Второй вариант записи более громоздкий, но позволяет удобно писать программы.

Рассмотрим программу поиска элемента в дереве – аналог программы member, но уже для деревьев.

Назовем её **in(Item, Tree)**.

В качестве первого параметра – искомый элемент, в качестве второго параметра – дерево вида btree(…)

```
1. in(Item, btree(Item, _, _)).
2. in(Item, btree(_, Left, _)) :- in(Item, Left).
3. in(Item, btree(_,_, Right)) :- in(Item, Right).
```

Первое правило – база индукции – гласит: искомый элемент находится в корне дерева. Второе правило гласит: искомый элемент находится в левом поддереве. Третье правило – искомый элемент находится в правом поддереве.

Напишите трассировку для запроса:

```
?- in(b, btree(a, btree(b, nil, nil)), nil).
```

Представленная программа осуществляет поиск в глубину. Предположим, что у нас не просто бинарное дерево, а бинарный словарь: в узлах дерева находятся числа, причем все числа каждого левого поддерева меньше либо равны корня, а все узлы каждого правого – строго больше.

Необходимо усовершенствовать программу поиска вхождения элемента в дерево. Кстати, именно такие деревья строит СУБД Oracle при индексировании.

```
1. inS(Item, btree(Item, _, _)).
2. inS(Item, btree(Root, Left, _)) :- Item =< Root, in(Item, Left).
3. inS(Item, btree(Root,_, Right)) :- Item > Root, in(Item, Right).
```

Бинарное дерево можно представить в виде списка: **btree2list(Tree, List)**.

```
1. btree2list(Tree, List):- btree2list(Tree, [], List).
2. btree2list(nil, List, List).
3. btree2list(btree(Root, Left, Right), List, [Root | RList]) :- btree2list(Left, List, List1), btree2list(Right, List1, RList).
```

Первое правило содержит вызов того же предиката, но с тремя параметрами, у которого инициализируется один из параметров пустым списком. Второе – база индукции. Третье – сначала собирает результаты из одного списка, затем из другого, после чего добавляет к ним корень дерева.

Рассмотрим запрос и ответ Пролога в режиме трассировки:

```
?- trace.
?- btree2list(btree(a, nil, nil), R).
      1    1  Call: btree2list(btree(a,nil,nil),_20) ?
      2    2  Call: btree2list(btree(a,nil,nil),[],_20) ?
      3    3  Call: btree2list(nil,[],_117) ?
      3    3  Exit: btree2list(nil,[],[]) ?
      4    3  Call: btree2list(nil,[],_77) ?
      4    3  Exit: btree2list(nil,[],[]) ?
      2    2  Exit: btree2list(btree(a,nil,nil),[],[a]) ?
      1    1  Exit: btree2list(btree(a,nil,nil),[a]) ?
R = [a]
yes
```

После каждого знака вопроса нажималась клавиша «Enter».