Рекурсия и списки.

Лекция 3.

Специальности : *230105, 010501*

Список как частный вид структуры.

Определение. Под списком понимается упорядоченная последовательность элементов, которая может иметь произвольную длину.

Признак *упорядоченный* указывает на то, что порядок элементов в последовательности является существенным и список [1,2,3] не эквивалентен списку [3,2,1].

Элементами списка могут быть любые термы - константы, переменные, структуры, последние могут включать в себя другие списки.

В отличие от Лиспа, в Прологе списки - один из частных видов структур. Список - это либо пустой список, не содержащий не одного элемента, либо структура, имеющая два компонента : голову и хвост. Конец списка представляется как хвост, который является пустым списком.

Способы представления списков.

Их 3 : функторный, графический и скобочный.

При использовании функторной формы записи голова и хвост являются компонентами функтора, обозначаемого точкой “.”.

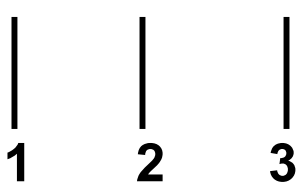
При использовании графической формы записи список представляется как специального вида дерево, “растущее” слева направо, причем ветви направлены вниз (“виноградная гроздь”).

При использовании скобочной формы записи последовательность элементов списка, разделенных запятыми, заключается в квадратные скобки.

<u>Пример :</u> список из трех чисел.

Функторная форма : **.(1,.(2,.(3,[ ])))**

Графическая форма :



Скобочная форма : **[1,2,3]**

Описание списков в Турбо-Прологе

Графическая форма записи списка в виде “виноградной лозы” удобна для записи списков на бумаге, но в Пролог-программах не используется. Функторная форма записи оказывается неудобной для записи сложных списков. В Прологе для записи списков используется скобочная форма записи.

Работа со списками основана на расщеплении на *голову* и *хвост* : [Head| Tail]. Голова есть первый аргумент функтора “.”, хвост - второй : . (Head, [Tail]]). Функтор “.” используется для конструирования списка. *Хвост* списка *есть список*, состоящий из всех элементов исходного списка, за исключением первого.

Примеры :

[1,2,3] - Head :1, Tail : [2,3] ; [[1], [2]] - Head : [1], Tail : [[2]].

Используя скобочную форму записи списков, можно создавать похожие на списки структуры, но не заканчивающиеся пустым списком (аналоги точечных пар Лиспа, в ряде реализаций Пролога (в том числе Visual Prolog и Turbo Prolog) не используются). Пример : [1 | 2]. Здесь Head : 1, Tail : 2.

Применение списков в программе.

Для использования списка в программе необходимо описать предикат списка. Пример : lexfun([“S0”,”Oper1”]).

Домен списка описывается в разделе domains, а работающий со списком предикат - в разделе predicates. Сам список задается в программе либо в разделе clauses, либо в разделе goal. Пример :

```
domains
      os_list=symbol*
predicates
       print_list(os_list)
goal
       print_list( [«DOS»,»Windows»,»Novell»,»Unix»,»Linux»] )
clauses
       print_list([ ]).
       print_list([ Head | Tail ]) :-
                    write(Head), nl, print_list(Tail).
```

Правила сопоставления списков.

Сопоставление списков - путем конкретизацией переменных.

Список 1	Список 2	Результат
[X,Y,Z]	[cat,dog,mouse]	X=cat Y=dog Z=mouse
[dog]	[X|Y]	X=dog Y=[]
[X,Z|Y]	[cat,dog,mouse]	X=cat Z=dog Y=[mouse]
[[b,c]|Z]	[[X,Y]|[w,b]]	X=b Y=c Z=[[w,b]]
[X,Y|Z,W]	Синтаксически некорректная конструкция списка	
[white,cat]	[cat,X]	Сопоставление невозможно
[white|Q]	[P|cat]	P=white Q=cat[1]

[1] сопоставление возможно не во всех реализациях Пролога

Рекурсия как основной метод программирования на Прологе.

Правило рекурсии содержит само себя в качестве компоненты.

В общем случае рекурсивное правило имеет следующий вид :

recursive_rule(<фактические параметры через запятую>):-

<предикаты и правила>,

recursive_rule(<фактич. параметры рекурсивного вызова>).

Для передачи значений между правилами используется стек. Всякий раз при рекурсивном вызове новые копии используемых значений помещаются в стек. В Турбо-Прологе имеются средства для автоматического освобождения использованной части стека.

Правила построения рекурсивных процедур.

Любое рекурсивное правило должно включать в себя как минимум по одной из следующих компонент :

- **Условие окончания рекурсии в виде нерекурсивного правила (факта).**
- **Изменяющийся аргумент рекурсивного вызова. При этом положение рекурсивного вызова в теле правила может быть любым.**

Факт или факты, обеспечивающие завершение рекурсии, должны в программе помещаться перед правилом, а не после него во избежание *левосторонней рекурсии*.

Левосторонняя рекурсия.

Возникает в случае, когда правило порождает подцель, эквивалентную исходной цели, которая явилась причиной использования этого правила. В процедуре с левой рекурсией рекурсивная подцель стоит слева от других подцелей.

Пример:

dog(X):-dog(Y), parent(Y,X).

dog(reks).

При попытке согласовать целевое утверждение dog(X) Пролог вначале пытается использовать правило и рекурсивно порождает подцель dog(Y). Попытка найти соответствие этой цели вновь приводит к выбору первого правила и так далее. Причина зацикливания заключается в отсутствии возможности использования механизма возврата. Для того, чтобы начался возврат, Пролог должен потерпеть неудачу при проверке первого утверждения, чего не происходит в приведенном примере.

Примеры использования рекурсии.

Пример 1. Вычисление факториала.

Количество аргументов : 1. Тип аргумента - целое число.

Условие выхода из рекурсии - факториал 0.

Вид правила рекурсии :

```
fact ( 0,1 ).
fact ( N, M ) : -
        N1 = N - 1,
        fact ( N1, M1 ),
        M = N * M1.
```

Рассмотрим работу нашего правила для вычисления 4!.

Работа правила вычисления факториала.

Вначале Пролог пытается выполнить подцель fact (4, M). Программа пытается сопоставить подцель с подправилом

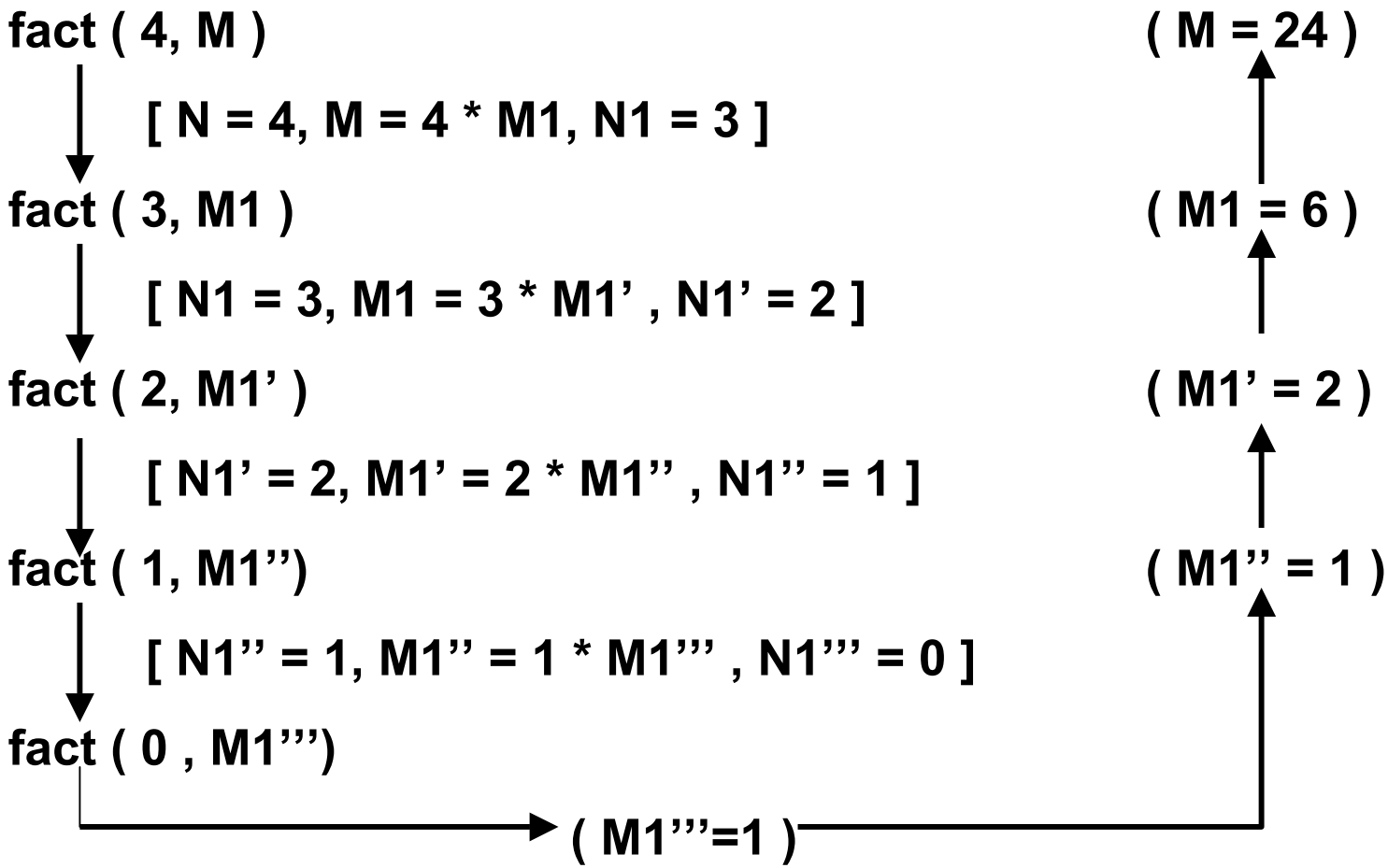
fact (0 , 1).

Сопоставление неудачно. Затем следует попытка сопоставления подцели с fact (N, M). На этот раз сопоставление завершается успешно с присвоением переменной N значения 4.

В этом правиле переменной N1 присваивается значение 3, то есть значение N - 1. Затем правило вызывает самого себя в виде fact (3, M). После этого вызова в теле правила идет вычисление значения переменной M с использованием свободной переменной M1, представляющей промежуточное значение факториала. Однако поскольку только что был вызван рекурсивный процесс, значение переменной M1 не может быть вычислено.

Этот циклический процесс сопоставления продолжается до тех пор, пока не будет получено fact (0, M1). Теперь это правило сопоставляется с fact (0 , 1) и M1 конкретизируется значением 1. При развертывании рекурсии программа запоминала значение M для последующего использования.

Этапы решения задачи “Вычисление факториала”



Пример 2 : Суммирование элементов списка.

Кол-во аргументов : 1, тип аргумента : список целых чисел.

Условие завершения рекурсии : пустой список.

Описание рекурсивного правила :

sum_lst ([ ] , 0).

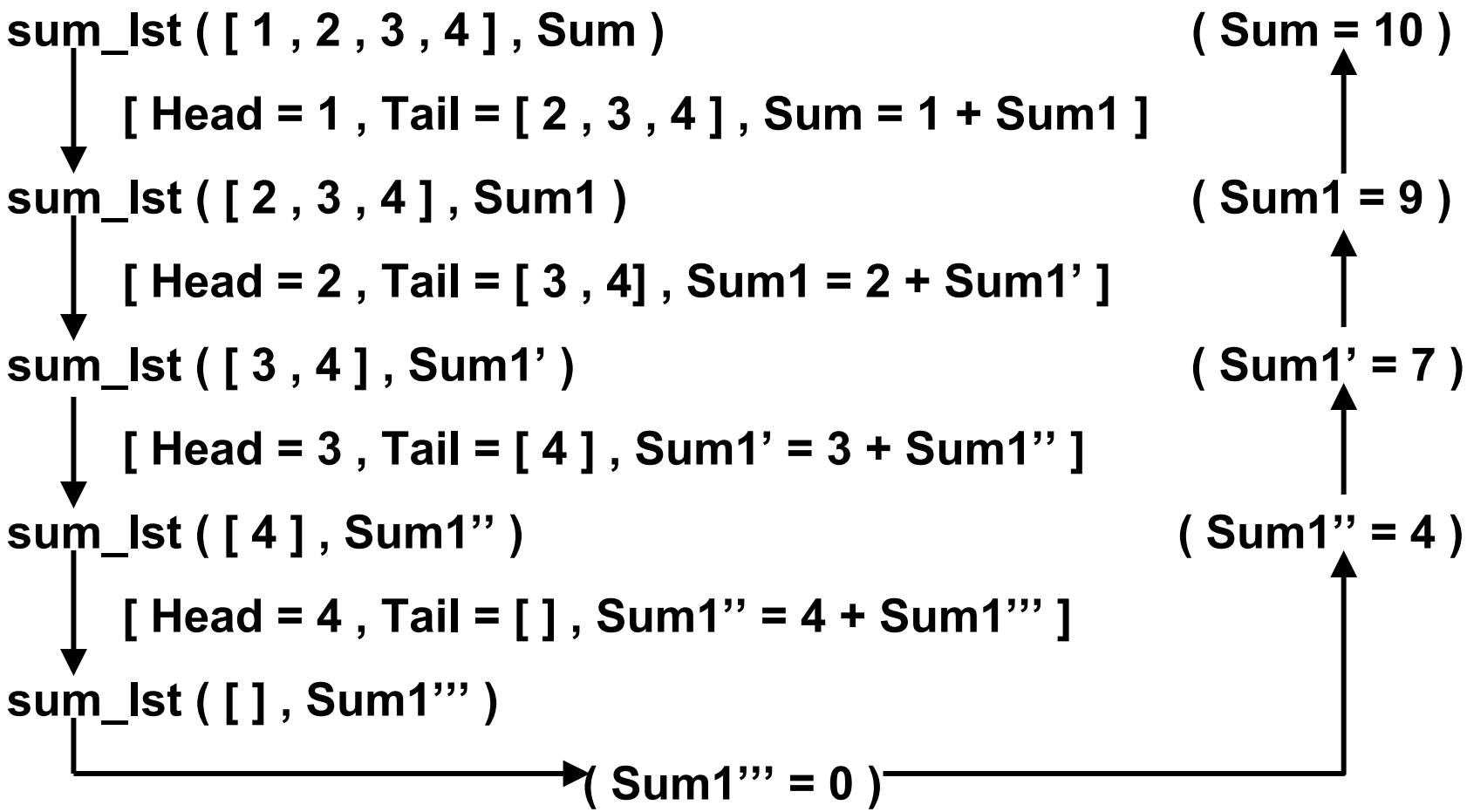
sum_lst ([Head | Tail] , Sum) : -

sum_lst (Tail , Sum1) ,

Sum = Head + Sum1 .

Рассмотрим работу правила на примере нахождения суммы элементов списка [1 , 2 , 3 , 4].

Этапы решения задачи “Нахождение суммы элементов списка”



Пример 3 : Печать элементов списка.

Постановка задачи. Есть список. Требуется напечатать все элементы списка в строчку.

Кол-во аргументов : 1.

Условие завершения рекурсии : пустой список.

Описание рекурсивного правила :

```
print_list ( [ ] ) .

print_list ( [ Head | Tail ] ) : -
                write ( Head , « «) , print_list ( Tail ).
```

Аналогично можно организовать печать элементов списка в столбик :

```
print_list_col ( [ ] ) .

print_list_col ( [ Head | Tail ] ) : - write ( Head ) , nl ,
                                      print_list_col ( Tail ) .
```

Объединение двух списков

Постановка задачи. Аргументы : 2 списка. Требуется построить список – результат присоединения одного списка к другому.

Описание рекурсивного правила :

```
domains
   t_elem=integer
   t_list=t_elem*
predicates
   append(t_list,t_list,t_list).
   append1(t_list,t_list,t_list).
clauses
   append([],L,L).
   append([Head_lst1|Tail_lst1],Lst2,[Head_lst1|Tail_res]):-
         append(Tail_lst1,Lst2,Tail_res).
```

Разностные списки

В общем случае разностью двух списков X и Y называется список X1 такой, что X есть результат объединения X1 и Y.

Разностным списком считается структура [X|Y]\Y, используемая для обозначения разности списков [X|Y] и Y. Пример. Для разности списков [1,2,3,4,5] и [4,5], [1,2,3,8] и [8] наиболее общий разностный список, представляющий последовательность 1,2,3, есть [1,2,3|Xs]\Xs. Здесь [1,2,3|Xs] – голова, а Xs – хвост разностного списка.

Разностные списки напрямую не могут быть объявлены в изучаемых реализациях Пролога. Один из вариантов их описания – использование составных объектов вида dl(Голова,Хвост). Рассмотрим пример использования неполных разностных списков для решения задачи объединения списков.

```
domains
   slist=string*
   dl=dl(slist,slist)
predicates
   append_dl(dl,dl,dl)
clauses
   append_dl(dl(Xs,Ys),dl(Ys,Zs),dl(Xs,Zs)).
```

Пример : вопросу append_dl(dl(["a","b","c"|Xs],Xs),dl(["1","2"],[]),Ys) будет соответствовать результат Xs=["1","2"],Ys=dl(["a","b","c","1","2"],[]).

Применение разностных списков в программах на Прологе повышает их быстродействие за счет большей эффективности процедуры объединения разностных списков по сравнению с обычными. Неполные разностные списки dl(Xs,Ys) и dl(Ys,Zs) могут быть соединены посредством правила append_dl за константное время. Соединение обычных списков посредством append происходит за линейное время, пропорциональное длине первого списка.

Сложность логических программ

Естественной мерой сложности логической программы является длина доказательств, порождаемых при выводе целей из значений логической программы.

Ключевое понятие – размер цели, определяется рекурсивно.

Размер терма – это число символов в текстовой записи терма. Константы и переменные записываются с помощью одного символа и имеют размер 1. Размер составного терма на единицу больше суммы размеров аргументов терма. Пример : список [a,b] имеет размер 5, цель append([a,b],[c,d],X) имеет размер 12.

Сложность длины вывода программы P равна L(n), если любая цель G, принадлежащая значению программы и имеющая размер n, может быть выведена из программы P с длиной вывода, не превосходящей L(n). Сложность длины вывода связана с обычными мерами сложности в теории алгоритмов. В случае последовательной реализации вычислительной модели эта сложность соответствует временной сложности. Применимость этой меры сложности к программам на Прологе, а не к логическим программам, зависит от использования алгоритма унификации без проверки на вхождение. Так, общая сложность вычисления *append* квадратично зависит от размера входных списков при использовании проверки на вхождение и линейно – без использования проверки.

Пусть R – доказательство. Глубиной R называется самое глубокое использование цели в некоторой резолюции. Размер цели в R – максимальный размер редуцируемых в R целей.

Логическая программа имеет сложность размера цели G(n), если любая цель A, принадлежащая значению программы и имеющая размер n, может быть выведена из программы P так, что размер цели в выводе не превысит G(n).

Логическая программа P имеет сложность глубины вывода D(n), если любая цель A, принадлежащая значению программы и имеющая размер n, может быть выведена из программы P с глубиной вывода, не превосходящей D(n).

Деревья как частный случай многодоменных структур.

Описание деревьев в Пролог-программе производится с использованием многодоменных структур как частного случая составных объектов. Домен дерева некоторого типа задается при помощи термового функтора :

<домен_дерева>=<функтор_дерева>(<тип_домена_узла>,<домен_дерева>, … ,<домен_дерева>).

При этом первый объет в структуре представляет информационное наполнение узла, последующие элементы структуры представляют дочерние поддеревья. Причем дочерине поддеревья могут быть заданы списком, например :

```
domains
        node=integer.
        child_list=tree*.
        tree=tree(node,child_list).
```

Бинарные деревья.

Бинарное дерево можно рекурсивно определить как многодоменную структуру, первый объект которой соответствует вершине, второй и третий - левому и правому поддеревьям.

Поскольку количество дочерних поддеревьев фиксировано, их указывают явным образом в виде объектов структуры.

Пример.

```
/* Бинарное дерево. */
domains
   node=integer.
   tree=tree(node,tree,tree);
        nil;empty.

predicates
   bintree(tree).
```

```
clauses
  bintree(tree(100,tree(90,
                        tree(70,nil,nil),
                        tree(95,nil,nil)),
               tree(200,tree(150,nil,
                             tree(175,nil,nil)),
                        tree(250,nil,nil)
                    )
                )
          ).
```

Основные действия над деревьями.

Основное применение бинарных деревьев - базы данных. Смысл использования бинарного дерева - хранить базу в отсортированном виде. При этом использование бинарного дерева является одним из вариантов построения хэш-функции как разбиения всего списка данных на группы в зависимости от значений ключевых полей. В частности, метод сбалансированных деревьев (с различием правой и левой ветви любой вершины не более чем на один уровень) позволяет искать, удалять, вставлять узел дерева за количество шагов ~$2LOG_2N$, где N - количество записей в БД. Программы работы с бинарными деревьями используют двойную рекурсию, по одной на каждую ветвь дерева.

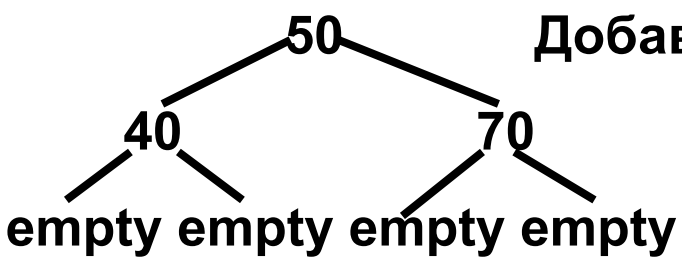
Основные действия над деревом.

- **Включение элемента в дерево листом.**
- **Поиск элемента в дереве.**
- **Удаление элемента из дерева.**
- **Включение корня в дерево.**

Включение элемента в дерево листом.

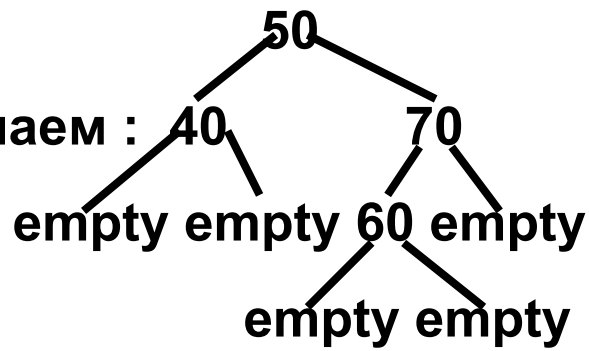
```
/* Включение листом :
     insert(Доб_элемент,Старое_дерево,Новое_дерево) */
/* Условие окончания рекурсии */
   insert(E,empty,tree(E,empty,empty)).
   insert(E,nil,tree(E,nil,nil)).
/* Добавляемый элемент больше корня */
   insert(E,tree(K,L,R),tree(K,L,R1)):-
         E>K,insert(E,R,R1).
/* Добавляемый элемент меньше корня */
   insert(E,tree(K,L,R),tree(K,L1,R)):-
         E<K,insert(E,L,L1).
/* Добавляемый элемент равен корню */
   insert(E,tree(E,L,R),tree(E,L,R)).
```

Пример.



Добавляем 60

Получаем :



Поиск элемента в дереве.

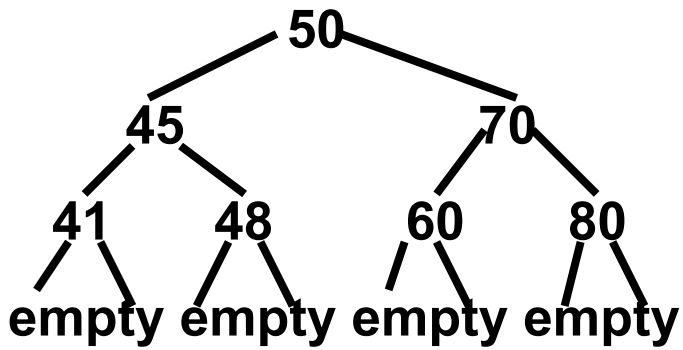
```
/* Поиск по дереву */
   find(Elem,tree(Elem,_,_)).
/* Элемент больше корня */
   find(E,tree(K,L,R)):-
        E>K,find(E,R).
/* Элемент меньше корня */
   find(E,tree(K,L,R)):-
        E<K,find(E,L).
```

Следует отметить, что описанное правило работает только для того бинарного дерева, для любого узла которого выполняется условие : вершина левого непустого поддерева должна содержать число строго меньшее, а вершина правого непустого поддерева - строго большее находящегося в рассматриваемом узле.

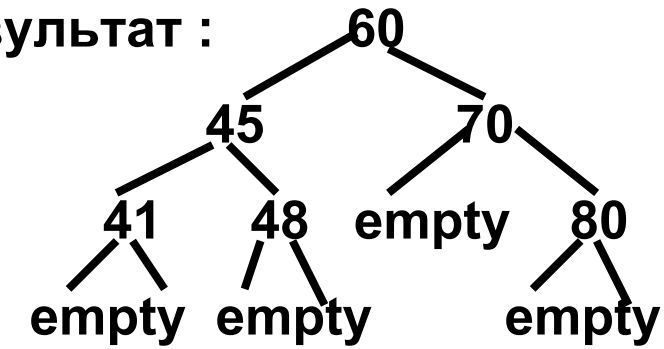
Удаление элемента из дерева.

```
/* Удаление элемента из дерева */
    delete(X,tree(X,empty,R),R).
    delete(X,tree(X,nil,R),R).
    delete(X,tree(X,L,empty),L).
    delete(X,tree(X,L,nil),L).
/* Поиск удаляемого элемента */
/* Удаляемый элемент больше корня */
    delete(X,tree(K,L,R),tree(K,L,R1)):-
          X>K,delete(X,R,R1).
/* Удаляемый элемент меньше корня */
    delete(X,tree(K,L,R),tree(K,L1,R)):-
          X<K,delete(X,L,L1).
/* Удаление вершины */
    delete(X,tree(X,L,R),tree(Y,L,R1)):-
          move_root(Y,R,R1).
/* Замена удаляемого корня самым левым элементом правого поддерева */
    move_root(X,tree(X,empty,R),R).
    move_root(X,tree(X,nil,R),R).
    move_root(X,tree(K,L,R),tree(K,L1,R)):-
          move_root(X,L,L1).
```

Пример : удаление корня



Результат :



Включение корня в дерево.

При включении корнем в зависимости от значения добавляемого элемента производится сортировка просеиванием части элементов исходного дерева относительно добавляемого элемента таким образом, чтобы левое поддерево содержало элементы строго меньшие, правое - строго большие корня.

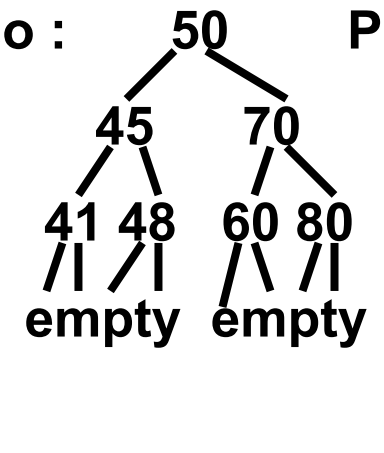
```
/* Включение нового корня */
/* Включение корня в пустое дерево */
   ins_t(K,empty,tree(K,empty,empty)).
   ins_t(K,nil,tree(K,nil,nil)).
/* Добавляемый элемент меньше
   старого корня */
   ins_t(K,tree(E,L,R),tree(K,L1,tree(E,L2,R))):-
        K<E,ins_t(K,L,tree(K,L1,L2)).
/* Добавляемый элемент больше
   старого корня */
   ins_t(K,tree(E,L,R),tree(K,tree(E,L,R1),R2)):-
        K>E,ins_t(K,R,tree(K,R1,R2)).
```

Примеры : включение корня.

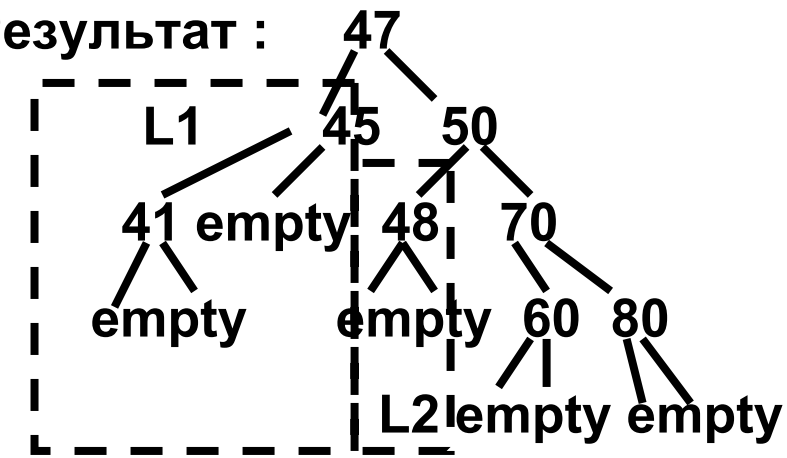
Пример 1. Включаемый элемент меньше старого корня.

Исходное дерево :

Добавляемый элемент : 47

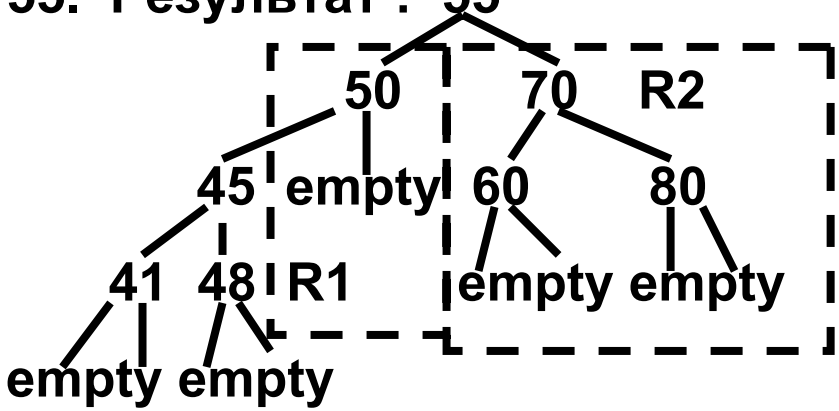


Результат :



Пример 2. Включаемый элемент больше старого корня.

Добавляемый элемент : 55. Результат :



Выводы.

- Использование многодоменных структур Пролога позволяет описывать помеченные деревья произвольной арности и с произвольной структурой информационного наполнения узлов и ветвей.
- Рекурсивная природа деревьев позволяет организовать параллельную рекурсивную обработку леса дочерних поддеревьев каждого узла.
- Введение альтернативных доменов дает возможность рассмотрения случаев пустого дерева для описания условий завершения рекурсии.

Литература.

Клоксин У., Меллиш К. Программирование на языке Пролог : Пер. с англ. - М.: Мир, 1987. С. 63-74

Маплас Дж. Реляционный язык Пролог и его применение : Пер. с англ. - М.: Наука, 1990. С. 90-95, 120-126

Ин Ц., Соломон Д. Использование Турбо-Пролога : Пер. с англ. - М.: Мир, 1993. С. 93-106, 139-152, 153-187

Стерлинг Л., Шапиро Э. Искусство программирования на языке Пролог : Пер. с англ. - М.: Мир, 1990. С. 54-57, 74-75, 190-196

Доорс Дж. и др. Пролог - язык программирования будущего : Пер. с англ. - М.: Финансы и статистика, 1990. С. 52-57