Решение логических задач с использованием рекурсивных типов данных – списков и деревьев

Рекурсивные программы

Основной способ решения сложных задач заключается в том, что исходная задача разбивается на части или подзадачи. Для каждой подзадачи находится алгоритм решения, а затем эти алгоритмы объединяются, чтобы получить решение исходной задачи. Если подзадачи оказываются достаточно сложными, их в свою очередь разбивают на части и т.д.

Такой способ решения представляет особый интерес, если подзадача является уменьшенной копией основной задачи. В этом случае к копии может быть применен тот же способ выделения подзадачи, что и в исходной задаче.

Применяя последовательно один и тот же способ выделения подзадачи к каждой новой копии меньшего размера, получаем в общем случае бесконечную последовательность копий.

Описанный процесс образования копий называют **рекурсией**. Для того, чтобы остановить процесс образования копий необходимо указать копию, не допускающую дальнейшего выделения подзадач. Такую копию называют *граничным условием*.

Пример. Рассмотрим построение рекурсивной процедуры для вычисления факториала n!. Согласно определению значение n! находится путем перемножения натуральных чисел от 1 до n . Процесс последовательного вычисления факториала можно представить в виде рекурсии, выделяя в качестве подзадачи вычисление факториала меньшего размера:

n ! = (n - 1) ! * n ,

при граничном условии 0! = 1. Этому описанию можно поставить в соответствие рекурсивную процедуру. Для этого введем отношение factorial(N,F), где F равно факториалу числа N . Программа, задающая это отношение состоит из двух предложений и имеет вид:

factorial(0,1).

factorial(N,F) :– Y = N–1, factorial(Y,Z), F = N*Z .

Выполнение рекурсивной программы разбивается на два этапа: разворачивание и сворачивание рекурсии. Каждый шаг этапа разворачивания рекурсии начинается повторным вызовом той же самой функции. При этом на каждом шаге этапа разворачивания могут остаться незавершенные вычисления, для выполнения которых необходимо предусмотреть сохранение промежуточных результатов. Этап сворачивания начинается после выполнения граничного условия. Во время этого этапа заканчиваются все незавершенные вычисления на каждом шаге в обратном порядке.

Если к программе вычисления факториала обратиться с вопросом

goal factorial(3, F).

то процесс выполнения рекурсивной программы можно представить в следующем виде.

1) Разворачивание рекурсии начинается с попытки унификации цели, определяемой вопросом, и первого правила – факта. Первые аргументы цели и правила представляют собой разные константы, поэтому унификация невозможна. Анализ второго правила начинается с переименования переменных. В результате имеем:

factorial(N1,F1) :– Y1= N1–1, factorial(Y1,Z1), F1 = N1*Z1.

Заголовок этого правила и цели унифицируется с помощью подстановки Q1 = {N1 = 3, F = F1}.

В результате выполнения первого предиката тела правила переменная Y1 становится равной 2. Третий предикат остается невыполненным.

2) Новая цель factorial(2, Z1) не унифицируется с первым правилом программы, поэтому переходим ко второму правилу. Выполняя переименование переменных, получаем:

factorial(N2,F2) :– Y2 = N2–1,

factorial(Y2,Z2), F2 = N2*Z2.

Заголовок этого правила унифицируется с целью с помощью подстановки Q2 = {N2=2, F2=Z1}.

После выполнения первого предиката переменная Y2 становится равной 1, а второй предикат порождает новый шаг разворачивания. третий конъюктивный член правила остается невыполненным.

3) Новая цель factorial(1, Z2) не унифицируется с первым правилом программы, поэтому выполняя переименование переменных второго правила, имеем:

factorial(N3,F3) :– Y3 = N3–1,

factorial(Y3,Z3), F3 = N3*Z3.

Заголовок правила унифицируется с целью с помощью подстановки Q3= {N3 = 1, F3 = Z2}. В результате выполнения первого предиката правила переменная Y3 становится равной 0. Второй предикат правила порождает новый шаг разворачивания, а третий член правила остается невыполненным.

4) Новая цель factorial(0,Z3) унифицируется с первым правилом программы с помощью подстановки Q4 = {Z3 = 1}. Первое правило является фактом, поэтому его успешный вывод соответствует выполнению граничного условия и определяет начало этапа сворачивания рекурсии.

5) Заканчивается вычисление третьего шага разворачивания. При этом используется переменная Z3 , которая была конкретизирована на четвертом шаге. В результате выполнения конкретизируется переменная F3 = N3*Z3 = 1*1 = 1 .

6) Заканчивается вычисление второго шага разворачивания. При этом используется переменная Z2=F3=1, которая была конкретизирована на третьем шаге. В результате выполнения находим, что F2 = N2*Z2 = 2*1 = 2 .

7) Выполняя последнее незавершенное вычисление первого шага, находим значение переменной F1 , определяющей окончательный результат F1=6.

Рекурсия является одним из основных методов программирования на Прологе, поскольку в Прологе отсутствуют конструкции, позволяющие описывать циклические вычисления.

Отсечение

В случае, когда одна из целей в резольвенте заканчивается неудачей, включается механизм возврата Пролога. Существует возможность управлять работой механизма возврата с помощью специальной цели – отсечения, обозначаемой символом "!". Доказательство этой цели всегда заканчивается успешно. После этого отключается механизм возврата и все цели, расположенные в тексте правила до цели-отсечения (включая и головную цель правила) не доказываются повторно.

Например, при использовании правил

r :– a, b, !, c.
r :– d.

в случае неудачи при доказательстве цели "c" механизм возврата отключается и цели a, b и головная цель r из второго правила повторно не доказываются. Тем самым определяется достаточность их первоначального доказательства.

Отсечение позволяет исключать из процесса вычисления те пути, которые не приводят к новым решениям, при этом сокращается время выполнения запроса. Часто отсечение используется в программах, содержащих взаимоисключающие правила.

Отсечение может эффективно применяться также для устранения бесконечных циклов в процессе вычисления. Например, если в программе для вычисления факториала

factorial(0, 1):- !.
factorial(N,F):- Y = N-1, factorial 0(Y,Z), F = N*Z.

устранить отсечение, то в случае запроса

goal factorial(0,Z),

второе правило также успешно унифицируется с запросом и создается новая цель

goal factorial(-1, Z1),

что, в свою очередь, приводит к цели

goal factorial(-2, Z2)

и т.д. до бесконечности.

Отсечение позволяет устранить переход ко второму (альтернативному) правилу при удачном вычислении для первого.

Использование отсечения может привести и к нежелательным последствиям, например, к изменению декларативного значения программы. Декларативная интерпретация программы изменяется и не соответствует ее процедурной интерпретации, если программа с отсечениями может приводить к другим результатам, нежели такая же программа, но без отсечений. Такие отсечения носят название красных отсечений.

Если в программе отсечения используются так, что их исключение приводит к программе, не изменяющей свою работу и, следовательно, не изменяющей декларативную интерпретацию, то такие отсечения называются зелеными.

Другим нежелательным эффектом применения отсечения является устранение необходимых альтернатив в процессе вычисления.

Вообще говоря, использование красных отсечений в логических программах нельзя отнести к хорошему стилю программирования, так как оно затрудняет анализ и понимание программ и поощряет нежелание тщательно продумывать описание логики отношений в программе. Вместо этого у программиста появляется возможность описать более широкое и простое отношение, чем требуется, и расставить затем в программе цели-отсечения, устраняя тем самым нежелательные пути вычислений. Но такое программирование значительно увеличивает вероятность ошибок и побочных эффектов от красных отсечений. Использование зеленых отсечений позволяет получить эффективную программу и при этом сохранить ее ясность. Однако необходим тщательный анализ программы, чтобы красное отсечение не было принято за зеленое.

Списки

Основной формой представления структурированных данных в Прологе является список.

Списком называют упорядоченную последовательность элементов, которая может иметь произвольную длину.

Элементы последовательности должны быть отделены друг от друга запятыми, а вся последовательность должна быть заключена в квадратные скобки. В качестве элементов списка могут быть атомы, переменные, составные

термы и другие списки. Например, списками являются следующие конструкции:

[a, b, c, d]
[abc]
[X, [t, f], dol]

Над списками определена одна операция, определяющая “голову” и “хвост” списка. Эта операция обозначается вертикальной чертой “|”. Головой списка считают его первый элемент, а хвостом списка называют список, полученный из исходного после отбрасывания первого элемента. Пустой список обозначается двумя квадратными скобками []. Следующие примеры показывают, как производится разделение списка на голову и хвост.

Список	**Голова**	**Хвост**
[a, b, c, d]	a	[b, c, d]
[abc]	abc	[]
[[X, Y], f, [h, g]]	[X, Y]	[f, [h, g]]
[X+Y, Z+W]	X+Y	[Z+W]
[аист, на, крыше]	Аист	[на, крыше]
[]	–	–

Для того, чтобы разделить, например, список [a, b, c, d] на голову и хвост, можно использовать следующий вопрос:

GOAL [X | Y] = [a, b, c, d].

Выполнение этого вопроса дает X=a, Y=[b, c, d]. Присоединение элемента a к списку [b, c, d] может быть выполнено с использованием вопроса:

GOAL L = [a | [b, c, d]]

В результате получаем L = [a, b, c, d].

Оператор “=”, использованный в вопросах, проверяет возможность отождествления двух термов.

Обработка списковых структур осуществляется на Прологе, как правило, с использованием рекурсии. Чтобы продемонстрировать возможности работы со списками, приведем несколько примеров.

Пример 1. Требуется написать программу на Прологе, проверяющую совпадение двух слов, заданных в виде списков.

Например, [т, р, о, н] и [т, р, е, с, т].

Для построения программы можно воспользоваться следующими рассуждениями. Если первые буквы слов совпадают, то операцию выделения и сравнения первых букв нужно повторить для оставшихся частей списков. Выделение и сравнение нужно прекратить, если оба списка оказались пустыми. В этом случае ответ положительный. Если же какая-либо пара букв не совпадает, то ответ отрицательный. В соответствии с этими рассуждениями программу можно записать так:

```
%проверка совпадения элементов двух списков
domains
list = symbol*

predicates
overlap(list, list)

clauses
overlap([], []).
overlap([X|L], [X|M]) :– overlap(L, M).
```

Пример 2. Написать программу, проверяющую принадлежность элемента списку. Для решения задачи введем отношение со схемой enter(element, list), которое должно давать положительный ответ, если “element” принадлежит списку “list”. Простейший случай, определяющий граничные условия, заключается в том, что заданный элемент совпадает с первым элементом списка. Если это не так, то необходимо повторить проверку для оставшейся части списка. Перечисленные случаи могут быть описаны с помощью следующей программы:

```
% проверка принадлежности элемента списку
domains
list = symbol*

predicates
enter(symbol, list)


clauses
enter(X, [X|L).
enter(X, [Y|L] :– enter(X,L).
```

Программа завершает работу либо при совпадении элемента X с одним из элементов списка, либо когда список становится пустым.

Пример 3. Написать программу, осуществляющую объединение двух списков путем присоединения к первому списку второго. Например, объединение списка [a, b, c] со списком [d, e, f] дает в результате список [a, b, c, d, e, f].

Исходя из условия задачи, выберем схему основного отношения в виде: append(list1, list2, result_list), в котором список result_list является итогом объединения списков list1 и list2.

Очевидным решением задачи представляется случай, когда первый список пустой. При этом результатом является второй список. Если же первый список не пуст, то можно выделить его первый элемент и присоединить его ко второму списку, а затем повторить эти действия для оставшейся части первого списка. Перечисленные ситуации могут быть представлены в виде программы:

```
%объединение двух списков в один
domains
list = symbol*

predicates
append(list, list, list)

clauses
append([], L, L).
append([X|M], L, [X|N]) :– append(M, L, N).
```

Эта программа на вопрос

```
goal
append([a, b, c,], [d, e, f], N)
```

дает ответ

T=[a, b, c, d, e, f].

Пример 4. Необходимо преобразовать список, элементами которого являются целые числа, инвертируя знак элементов списка, то есть положительные числа преобразовать в отрицательные, отрицательные в положительные, для нулевых значений никаких действий не предпринимать.

Придется рассмотреть два случая - для непустого и пустого списков. Преобразование пустого списка дать в результате также пустой список. Если же список не пуст, то следует рекурсивно выполнять отделение головы списка, ее обработку и рассматривать полученный результат как голову списка-результата.

```
domains
intlist=integer*

predicates
inverting (intlist, intlist)
processing (integer, integer)

clauses
%обработка пустого списка дает, в результате, тоже пустой список
inverting ([ ], [ ]):- !.
%если список непустой, отделить голову, обработать ее,
```

```
%и добавить в качестве головы списка-результата
inverting ([H | T], [Inv_H | Inv_T]):-
        processing (H, Inv_H), inverting (T, Inv_T).
%предикат processing выполняет действия по обработке элемента списка в
%зависимости от его знака, предикат имеет два предложения,
%так как нужно рассмотреть два варианта: ненулевое и нулевое значения
processing (0, 0):- !.
processing (H, Inv_H):- Inv_H=-H.

goal
inverting ([-2, -1, 0, 1, 2], Inv_List), write("Inv_List=", Inv_List).
```

Результат работы программы:

```
Inv_List=[2, 1, 0, -1, -2]
```

Пример 5. сортировка списка методом "пузырька".

Программа работает следующим образом: от списка отделяются первый и второй элементы и сравниваются между собой. В том случае, если операция сравнения успешна (условие сравнения зависит от способа сортировки - по возрастанию или по убыванию), выполняется полезная перестановка. Если операция сравнения неуспешна, то от списка отделяется первый элемент и работа продолжается с хвостом списка.

В случае, если весь список будет проверен и ни одной перестановки сделано не будет, выполнение программы завершится, так как список будет отсортирован.

```
domains
integerlist=integer*

predicates
bubble (integerlist, integerlist)
exchange (integerlist, integerlist)

clauses
bubble (OrdList, OrdList).
bubble (List, OrdList):- exchange(List, TempList),
                         !, bubble (TempList, OrdList).

exchange ([H1,H2|T], [H2,H1|T]):- H1>H2.
exchange ([H|T], [H|T2]):- exchange (T, T2).

goal
bubble ([3,2,1], OrdList), write("List=", OrdList).
```

Пример 6. Вычисление длины списка.

len(L,N) возвращает длину N списка L.

```
domains
integerlist=integer*

predicates
len(integerlist)

clauses
len([],0).
len( [ _ | Tail ],N) :– len(Tail,N1), N is N1 + 1.
```

Пример 7. Вывод элементов списка “в столбик”.

Для пустого списка ничего не делать.
Для непустого списка отделить голову и вывести ее, перевести строку.
Продолжить вывод для хвоста списка.

```
domains
integerlist=integer*

predicates
printlist (integerlist)

clauses
printlist ([]):- !.
printlist ([H|T]):- write (H), nl, printlist (T).
```

Отрицание

Некоторые недостатки, связанные с отсечениями, можно преодолеть, если использовать средство управления перебором вариантов более высокого уровня, отношение not – отрицание произвольной цели. В Прологе отрицание определяется как безуспешное выполнение и реализуется на основе отсечения и системного предиката **fail**. Выполнение предиката **fail** всегда приводит к неуспеху. Отношение not используется обычно как встроенный префиксный оператор. Запишем его как предикат:

```
not(X) :– X, !, fail.
not(X).
```

В этом определении X используется как метапеременная для обозначения цели. Если X закончилось успешно, то мы проходим через оператор отсечения, сталкиваемся с предикатом fail и заканчиваем вычисление not(X) как безуспешное; возврата влево и выбора следующего правила not не происходит по причине выполненного отсечения. Если в другом случае выполнения not(X)

вычисление X закончилось безуспешно, то производится переход ко второму правилу – факту, который завершается успехом. Таким образом, not(X) выполняется успешно, если X – неуспешно, и, наоборот, not(X) заканчивается неуспехом, если X завершилось успешно.

Например, если определить отношение not_factorial(N,F) между двумя числами, выполняемое, если F не является факториалом числа N:

not_factorial(N,F) :– not(factorial(N,F)).

то вопрос

goal
not(factorial(4,23)).

приведет к ответу «yes», а вопрос

goal
not(factorial(4,24)).

– к ответу «no».

Пример. Определим, что отношение **not_enter** для двух списков выполняется, если списки не содержат ни одного общего элемента:

not_enter(La, Lb) :– not(enter(X, La), enter(X, Lb)).

В приведенном правиле возможный общий элемент обозначен через переменную X.

С помощью отрицания можно расширить функциональные возможности программы путем реализации, например, предиката **if_then_else** с обычным для процедурных языков программирования смыслом:

if_then_else(P, S, Q) :– P, !, S.

if_then_else(P, S, Q) :– not(P), !, Q.

Заметим, что во втором правиле можно избежать повторного вычисления P, если заменить используемые зеленые отсечения на красное, применение которого в этом случае принято считать оправданным:

if_then_else(P, S, Q) :– P, !, S.

if_then_else(P, S, Q) :– Q.

Отрицание как безуспешное выполнение не полностью соответствует отрицанию в математической логике. Отрицание в Прологе является отрицанием в ограниченном смысле, основывается на предположении о замкнутости мира и интерпретируется как то, что невозможно вывести указанное отношение (аргумент not) из программы. Если отношение не выведено, то это свидетельствует либо о его ложности, либо о нехватке информации для вывода, который при наличии информации мог бы быть, возможно, успешным. Отсутствие достаточной информации связано с тем, что на практике предположение о замкнутости мира в программе обычно не

выполняется. Например, если некоторая программа включает бинарное отношение is_located(high_school, city), которое определяет, что первый аргумент – учебное заведение – находится в городе – втором аргументе, то ответом на вопрос

goal not(is_located(leti, st_Petersburg))

может быть значение «yes», что не соответствует действительности. Такой ответ может быть вызван либо ошибкой в правилах программы, либо тем, что информация об атоме leti просто отсутствует.

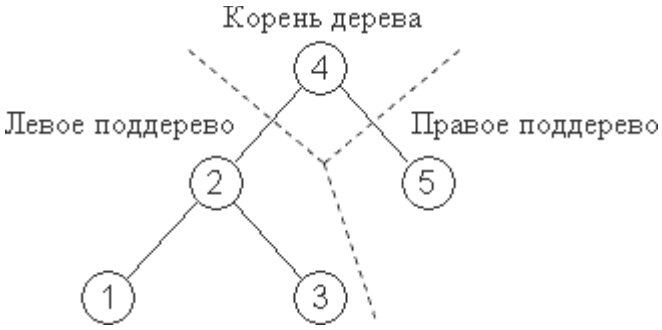
Сложности, связанные с использованием отрицания в Прологе, покажем еще на одном примере. Вопрос

goal not(A=13), A=12 приводит к безуспешному выполнению несмотря на то, что декларативна интерпретация этого вопроса соответствует ответу A=12. Оказывается, что подобные трудности исключены только в том случае, если аргумент оператора not при вычислении не содержит неконкретизированных переменных, а включает лишь константы и конкретизированные переменные. Ответственность за использование not с переменными в аргументе ложится на программиста. Так, ответ на поставленный вопрос оказывается верным, если в нем переставить цели:

goal A=12, not(A=13).

Таким образом, отрицание в Прологе нужно использовать после тщательного анализа возможных последствий.

Деревья, также как и списки, являются рекурсивным типом данных. Дерево - это структура данных, которая может быть разделена на корень дерева, левое и правое поддеревья. Так как левое и правое поддеревья в свою очередь являются деревьями, структура рекурсивна. Кроме того, дерево является еще и составным объектом данных.



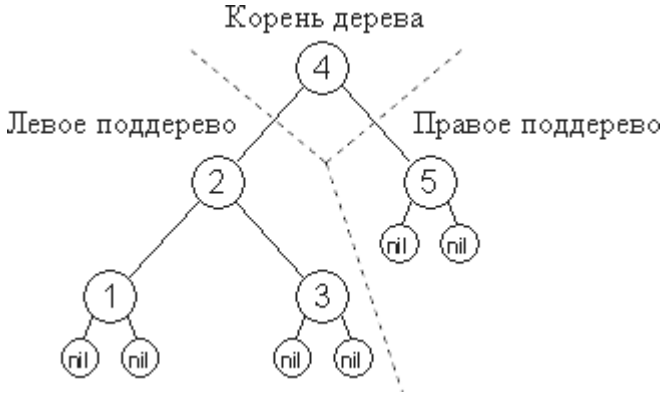
Дерево, которое имеет только два поддерева, называется двоичным или бинарным. В том случае, если для каждого корня дерева выполняется условие, при котором значение, находящееся в корне дерева меньше значения, находящегося в корне левого поддерева и больше значения, находящегося в корне правого поддерева, двоичное дерево называется упорядоченным. Дерево, приведенное на рисунке выше, является упорядоченным.

В Prolog'e можно определять рекурсивные типы данных. Например, можно определить дерево следующим образом:

DOMAINS
treetype=tree(integer, treetype, treetype); nil()

Такое определение говорит о том, что дерево является составным объектом, состоящим из трех составных частей: корня, принадлежащего домену integer и двух поддеревьев, принадлежащих домену treetype, так как именно этот домен и описывает структуру данных типа дерево. Так как дерево является составным объектом, его составные части объединяет функтор tree. Кроме того, дерево может находиться в двух состояниях: быть непустым (иметь хотя бы один корень) или пустым (не иметь ни одного корня). Пустое дерево описывается функтором nil без параметров. Если у функтора нет параметров, пустые скобки можно не указывать и записывать только имя функтора. Имена функторов tree и nil, домена treetype выбраны произвольно.

Узлы дерева, которые содержат значения 1, 3 и 5, не имеют левых и правых поддеревьев. Такие узлы можно назвать листьевыми узлами или листьевыми вершинами.



Такое определение позволяет записать следующую структуру данных:

tree (4,
tree (2,
tree (1, nil, nil),
tree (3, nil, nil)),
tree (5, nil, nil))

Одной из наиболее частых операций с деревом является обход узлов дерева и выполнение некоторых действий с ними. Например, вывод значений всех корней дерева. Способ решения этой задачи можно описать следующим образом:

1. если дерево пустое, корня в дереве нет, нет и значения корня для вывода, не выполнять никаких действий;

2. если дерево непустое, то разделить дерево на корень, левое и правое поддеревья, выполнить вывод значения, находящегося в корне дерева, затем обработать левое и правое поддеревья.

Каждое из условий в описании задачи соответствует предложению в программе Prolog'a.

```
DOMAINS
treetype=tree (integer, treetype, treetype); nil ()

PREDICATES
print_tree (treetype)

CLAUSES
%дерево пусто, поэтому никакие действия не выполняются
print_tree (nil):- !.
%дерево непусто, поэтому дерево разбивается на три составные части,
%сначала выводится корень, затем обрабатываются левое и правое
%поддеревья
print_tree (tree(Root, Left, Right)):- write(Root), nl,
                                       print_tree (Left),
                                       print_tree (Right).

GOAL
print_tree (tree (4, tree (2, tree (1, nil, nil),
     tree (3, nil, nil)), tree (5, nil, nil))).
```

Результат работы программы:

4

2

1

3

5

Следует отметить, что в большинстве случает рекурсия, используемая при работе с деревьями, хвостовой не является, так приходится обрабатывать левое и правое поддеревья, что дает две рекурсивные цели в одном предложении и, соответственно, не выполняется первое правило хвостовой рекурсии - рекурсивный вызов должен быть последней целью в хвостовой части правила вывода.

Действия, которые выполняет программа, последовательно обрабатывая корни дерева, называются обходом дерева. В зависимости от последовательности обработки корней дерева различают:

1. обход дерева сверху вниз (сначала обрабатывается корень, затем поддеревья);
2. обход дерева снизу вверх (сначала обрабатываются поддеревья, затем корень);
3. обход дерева слева направо (сначала обрабатывается левое поддерево, затем корень, затем правое поддерево);
4. обход дерева справа налево (сначала обрабатывается правое поддерево, затем корень, затем левое поддерево).

В том случае, если дерево упорядочено, то при обходе дерева слева направо будет получена возрастающая последовательность значений, находящихся в корнях деревьев, 1, 2, 3, 4, 5, при обходе справа налево будет получена убывающая последовательность значений 5, 4, 3, 2, 1.

Древовидная структура данных может быть создана программным путем. Рассмотрим создание упорядоченного дерева, начиная с пустого дерева. Новые узлы к дереву будут добавляться с сохранением упорядоченности дерева. Возможны три варианта:

1. добавление нового узла к пустому дереву;
2. добавление нового узла к непустому дереву, значение в новом узле меньше значения в корне дерева, новый узел тогда будет добавлен к левому поддереву;
3. добавление нового узла к непустому дереву, значение в новом узле больше значения в корне дерева, новый узел будет добавлен к правому поддереву.

Первый случай самый простой. Если к пустому дереву добавить новый узел, то получится дерево, состоящее из одного корня и двух пустых поддеревьев.

insert (New, nil, tree (New, nil, nil)).

Второй случай немного сложнее, так как придется определять место вставки нового узла, то есть сравнивать значение, которое нужно добавить к дереву, с корнем текущего дерева и если вставляемое значение окажется меньше, то добавлять новый узел в левое поддерево, получая таким образом новое левое поддерево. Правое поддерево остается неизменным.

insert (New, tree (Root, Left, Right), tree (Root, NewLeft, Right)):-
New<Root, !, insert (New, Left, NewLeft).

В третьем случае уже не требуется сравнивать значение нового узла с корнем текущего дерева, так как третье предложение будет выполняться только в случае, если для доказательства не будут использованы первые два. Новый

узел добавляется в правое поддерево, получая таким образом новое правое поддерево. В этом случае левое поддерево остается неизменным.

```
insert (New, tree (Root, Left, Right), tree (Root, Left, NewRight)):-
    insert (New, Right, NewRight).
```

Используем приведенные предложения в программе, которая строит дерево с сохранением упорядоченности, новые значения, добавляемые в дерево, вводятся до тех пор, пока не будет введен любой символ, не являющийся целым числом.

```
DOMAINS
treetype=tree (integer, treetype, treetype); nil ()

PREDICATES
create (treetype, treetype)
insert (integer, treetype, treetype)

CLAUSES
create (Tree, NewTree):- readint (New), !, insert (New, Tree, TmpTree),
                create (TmpTree, NewTree).
create (Tree, Tree).

insert (New, nil, tree (New, nil, nil)):- !.
insert (New, tree (Root, Left, Right), tree (Root, NewLeft, Right)):- New<Root, !,
    insert (New, Left, NewLeft).
insert (New, tree (Root, Left, Right), tree (Root, Left, NewRight)):-
    insert (New, Right, NewRight).

GOAL
create (nil, Tree), write (Tree).
```

Приведем еще один пример программы, работающей с деревом. В программе будет выполняться обход дерева сверху вниз, и вестись подсчет узлов дерева.

Снова нужно рассмотреть два случая: для пустого и непустого поддеревьев. Если дерево пустое, то узлов в нем 0. Если дерево непустое, то подсчитать число узлов в дереве в целом можно так: подсчитать число узлов левого поддерева, подсчитать число узлов правого поддерева, сложить эти значения и полученный результат увеличить на 1 (учесть корень дерева).

```
DOMAINS
treetype=tree (integer, treetype, treetype); nil ()
PREDICATES
count (treetype, integer)
```

```
CLAUSES
%дерево пусто, поэтому число его узлов равно 0
count (nil, 0):- !.
%дерево непусто, поэтому дерево разбивается на три составные части,
%подсчитывается число узлов каждого из поддеревьев, а затем находится
%общий результат
%обратите внимание, что для корня дерева использована анонимная
%переменная, так как в данной программе значение корня дерева не важно
count (tree (_, Left, Right), N):- count (Left, LN), count (Right, RN), N=LN+RN+1.

GOAL
count (tree (4, tree (2, tree (1, nil, nil), tree (3, nil, nil)),
      tree (5, nil, nil)), N), write ("Число узлов дерева N=", N).
```

И еще один пример программы, работающей с деревом. В программе будут выполняться те же самые действия, что и предыдущей, но работа с деревом организована немного по-другому. Если в предыдущем примере сначала было найдено число узлов левого и правого поддеревьев, а затем общее значение количества узлов для всего дерева, то в следующей программе параллельно с обходом дерева будет инкрементироваться счетчик узлов дерева. Обнуление счетчика будет сделано в секции GOAL.

Попробуйте самостоятельно определить, где задается начальное значение счетчика, равное 0, в предыдущем варианте программы. (Совет: попробуйте запустить программу на выполнение в режиме трассировки.)

```
DOMAINS
treetype=tree (integer, treetype, treetype); nil ()

PREDICATES
count2 (treetype, integer, integer)

CLAUSES
count2 (nil, Tmp, Tmp):- !.
count2 (tree (_, Left, Right), Tmp, N):- NewTmp=Tmp+1,
                                         count2 (Left, NewTmp, NewTmp2),
                                         count2 (Right, NewTmp2, N).

GOAL
count2 (tree (4, tree (2, tree (1, nil, nil), tree (3, nil, nil)),
      tree (5, nil, nil)), 0, N), write ("Число узлов дерева N=", N).
```