МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №2
по дисциплине «Логическое программирование»
Тема: Рекурсия и структура данных
Вариант 1

Студентка гр. 1304 _____ Чернякова В.А.
Студентка гр. 1304 _____ Ярусова Т.В.
Студент гр. 1304 _____ Байков Е.С.
Студент гр. 1304 _____ Мамин Р.А.
Преподаватель _____ Родионов С.В.

Санкт-Петербург
2025

Цель работы.

Изучение особенностей реализации рекурсии на языке Пролог, освоение принципов решения типовых логических программ.

Задачи.

Для достижения поставленной цели требуется решить следующие задачи:

1) Изучить теоретический материал.

2) Создать правила в соответствии с вариантом задания и общей формулировкой задачи.

3) Проверить выполнение программы.

4) Составить отчет о выполнении работы.

5) Представить на проверку файл отчета и файл текста программы на языке GNU Prolog, решающей поставленные задачи.

Задание.

Реализуйте выполнение задания с номером варианта, равным номеру бригады (для каждого варианта - по две задачи, одна – из Задания 1, вторая – из Задания 2).

Под заданиями приведены примеры для проверки решений. Рекомендуется во всех заданиях использовать рекурсивную обработку списка, с разделением его элементов на голову и хвост; можно определять/использовать вспомогательные предикаты.

[Задание 1, Списки]

Вставить число в упорядоченный список

```
?- list_insert(2, [1,2,3,4], X).
X = [1,2,2,3,4]
```

[Задание 2, Деревья]

Создайте предикат, проверяющий, что дерево является двоичным справочником.

```
?- is_ordered(tr(2,tr(7, nil, nil),tr(3,tr(4, nil, nil), tr(1, nil, nil)))).
No
```

Основные теоретические положения.

Рассмотрим несколько вариантов использования рекурсивного вызова на языке Пролог применительно к спискам.

Принадлежность списку. Сформулируем задачу проверки принадлежности данного терма списку.

Граничное условие:

Терм R содержится в списке [H|T], если R=H.

Рекурсивное условие:

Терм R содержится в списке [H|T], если R содержится в списке T.

Первый вариант записи определения на Прологе имеет вид:

содержится(R, L) :- L=[H | T], H=R.

содержится(R, L) :- L=[H | T], содержится (R, T).

Цель L=[H | T] в теле обоих утверждений служит для того, чтобы разделить список L на голову и хвост.

Можно улучшить программу, если учесть тот факт, что Пролог сначала сопоставляет с целью голову утверждения, а затем пытается согласовать его тело. Новая процедура, которую мы назовем "принадлежит", определяется таким образом:

принадлежит (R, [R | T]).

принадлежит (R, [H | T]) :- принадлежит (R, T).

На запрос

?- принадлежит(a, [a, Ь, c]).

будет получен ответ

да

на запрос

?- принадлежит(b, [a, b, c]).

- ответ

да

но на запрос

?- принадлежит(d, (a, b, c)).

Пролог дает ответ

нет

В большинстве реализации Пролога предикат «принадлежит» является встроенным.

Соединение двух списков. Задача присоединения списка Q к списку P, в результате чего получается список R, формулируется следующим образом:

Граничное условие:

Присоединение списка Q к [] дает Q.

Рекурсивное условие:

Присоединение списка Q к концу списка P выполняется так: Q присоединяется к хвосту P, а затем спереди добавляется голова P.

Определение можно непосредственно написать на Прологе:

соединить([],Q,Q).

соединить(P,Q,R) :- P=[HP | TP], соединить(TP, Q, TR), R=[HP | TR].

Однако, как и в предыдущем примере, воспользуемся тем, что Пролог сопоставляет с целью голову утверждения, прежде чем пытаться согласовать тело:

присоединить([] ,Q,Q).

присоединить(HP | TP], Q, [HP | TR]) :- присоединить (TP, Q, TR).

На запрос

?- присоединить [a, b, c], [d, e], L).

будет получен ответ

L = [a, b, c, d].

но на запрос

?- присоединить([a, b], [c, d], [e, f]).

ответом будет No

Часто процедура «присоединить» используется для получения списков, находящихся слева и справа от данного элемента:

присоединить (L [джим, p], [джек,.билл, джим, тим, джим, боб]) .

L = [джек, билл]

R = [тим, джим, боб]

другие решения (да/нет)? да

L=[джек, билл, джим, тим]

R=[боб]

другие решения (да/нет)? да

других решений нет

Индексирование списка. Задача получения N-го терма в списке определяется следующим образом:

Граничное условие:

Первый терм в списке [H | T] есть H.

Рекурсивное условие:

N-й терм в списке [H | T] является (N-1)-м термом в списке T.

Данному определению соответствует программа:

/* Граничное условие:

получить ([H | T], 1, H).

/* Рекурсивное условие:

получить([H | T], N, Y) :- M is N - 1, получить (T, M ,Y).

Порядок выполнения работы.

Задание 1. Списки

Вставить число в упорядоченный список.

В качестве упорядоченного списка рассмотрим такой, в котором все элементы расположены в порядке неубывания, как в примере к заданию.

Представление списка имеет следующий вид:

```
[Head | Tail]
```

Head – первый элемент списка (голова).

Tail – все остальные (хвост).

Обозначим элемент, который надо вставить в список за *X*.

На рисунке 1 изображена схема процесса добавления элемента в список.

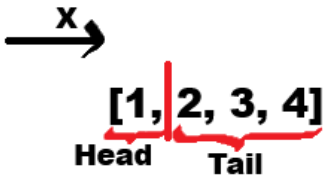

Рисунок 1. Схема добавления элемента в список

Рассмотрим различные ситуации, когда надо вставить элемент в упорядоченный список (неубывающий):

1) Список пустой.
    a. Если список пустой, то нет необходимости сравнивать добавляемый элемент X с другими, так как их нет. Значит, предикат, добавляющий элемент в пустой список, будет следующим:
    b. `list_insert(X, [], [X]).`
2) Место для добавляемого элемента найдено сразу.
    a. Если X =< Head, X будет добавлен перед текущем Head, чтобы сохранить упорядоченность списка.
    b. `list_insert(X, [Head | Tail], [X, Head | Tail]) :- X =< Head.`
3) Необходимо найти нужное место для вставки элемента в список.
    a. Если же X > Head, то необходимо дальше искать правильное место для вставки X в оставшуюся часть списка. В правило будет добавлен рекурсивный вызов до тех пор, пока X не получится добавить в нужное место, то есть данное условие – рекурсивное. А правило из пункта 2 – это остановка данного рекурсивного вызова (граничное условие).
    b. Голова правила должна давать возможность сохранять элементы списка, которые уже обработаны (Head), без изменений и выполнить вставку X в оставшуюся часть списка (Tail). А когда рекурсия найдёт место для X, обновлённый хвост (NewTail) должен передастся вверх для формирования правильного списка.
    c. `list_insert(X, [Head | Tail], [Head | NewTail]) :- X > Head, list_insert(X, Tail, NewTail).`

Итоговый код программы:

```
list_insert(X, [], [X]).
list_insert(X, [Head | Tail], [X, Head | Tail]) :- X =< Head.
list_insert(X, [Head | Tail], [Head | NewTail]) :-
X > Head, list_insert(X, Tail, NewTail).
```

Для наглядности разберем работу программы на примере:

№	Действие
1	list_insert(3, [1,2,3,4], Res)
2	`list_insert(X, [Head | Tail], [X, Head | Tail]) :- X =< Head` 3 =< 1 ? Условие не выполнено

2	`list_insert(X, [Head | Tail], [Head | NewTail]) :-`
	`X > Head,`
	`list_insert(X, Tail, NewTail).`
	3 > 1 ? Условие выполнено
	И
	list_insert(3,[2,3,4],NewTail)

3	list_insert(3,[2,3,4],NewTail)
3	`list_insert(X, [Head | Tail], [X, Head | Tail]) :- X =< Head` 3 =< 2 ? Условие не выполнено

3	`list_insert(X, [Head | Tail], [Head | NewTail]) :-`
	`X > Head,`
	`list_insert(X, Tail, NewTail).`
	3 > 2 ? Условие выполнено
	И
	list_insert(3,[3,4], NewTail)

4	list_insert(3,[3,4], NewTail)
4	`list_insert(X, [Head | Tail], [X, Head | Tail]) :- X =< Head` 3 =< 3 ? Условие выполнено Рекурсивно восстанавливаем значение NewTail на предыдущих вызовах
3	`list_insert(3,[3,4], NewTail)= list_insert(X, [Head | Tail], [X, Head | Tail])` На выходе: list_insert(3,[3,4],[3,3,4]) => `NewTail = [3,3,4]`

2	`list_insert(3,[2,3,4], NewTail) = list_insert(X, [Head | Tail], [Head | NewTail])` На выходе: list_insert(3,[2,3,4],[2,3,3,4]) => `NewTail = [2,3,3,4]`
1	`list_insert(3,[1,2,3,4], Res) = list_insert(X, [Head | Tail], [Head | NewTail])` На выходе: list_insert(3,[1,2,3,4],[1,2,3,3,4])

Итого Res = [1,2,3,3,4]

Задание 2. Деревья

Создайте предикат, проверяющий, что дерево является двоичным справочником.

Дерево обозначим предикатом *tr()* с тремя параметрами: 1 – корень, 2 – левое поддерево, 3 – правое поддерево.

Дадим определение двоичному справочнику.

Двоичный справочник — это особый вид бинарных деревьев, в котором

- Все значения, входящие в левое поддерево, меньше значения, находящегося в корне,
- Все значения, расположенные в вершинах правого поддерева, больше корневого значения,
- Левое и правое поддеревья, в свою очередь, также являются двоичными справочниками.

Из определения можно сделать следующие выводы:

- Для левого поддерева корень всего справочника – максимальное значение. Все остальные значения там меньше.
- Для правого поддерева корень всего справочника – минимальное значение. Все остальные значения там больше.

Рассмотрим некоторое поддерево двоичного дерева.

Запишем его следующим образом:

```
tr(Root, Left, Right)
```

- *Root* – корень поддерева.
- *Left* – левое поддерево.

- *Right* – правое поддерево.

Также введем следующие обозначения

- *Max* – значение корня самого дерева. Для левого поддерева все значения меньше *Max*.
- *Min* – также значение корня самого дерева, но для правого поддерева, где все значения больше *Min*.

В таком случае

- Для левого поддерева
  - `Root < Max`
- Для правого поддерева
  - `Root > Min`

Напишем правило:

```
is_ordered_check(tr(Root, Left, Right), Min, Max) :-
Root < Max,
Root > Min,
is_ordered_check(Left, Min, Root),
is_ordered_check(Right, Root, Max).
```

Был добавлен рекурсивный вызов для проверки всех поддеревьев. Для каждого следующего поддерева *Root* будет выступать в качестве минимального или максимального значения, в зависимости от того левое оно или правое.

Необходимо усовершенствовать правило, так как при первом вызове значения для *Min* и *Max* не определены.

Создадим дополнительное и основное правило в программе:

```
is_ordered(BTree) :- is_ordered_check(BTree, nil, nil).
```

nil – пустое поддерево.

Таким образом, правило, описанное ранее, изначально будет вызываться со значениями параметров *Max* и *Min* равными *nil*.

Так как числа нельзя сравнивать с данным значением, то в начале будем проверять, что *Min/Max = nil*, после чего через оператор И добавим встроенный предикат отсечения.

```
is_ordered_check(tr(Root, Left, Right), Min, Max) :-
(Max = nil,!; Root < Max),
```

```
(Min = nil,!; Root > Min),
is_ordered_check(Left, Min, Root),
is_ordered_check(Right, Root, Max).
```

Теперь *Max=nil* и *Min = nil* условия отсечения, и если условие истинно, то произойдет само отсечение, то есть альтернативный путь рассмотрен не будет, в данном случае сравнение *Root* с *Max = nil/Min = nil*, что не вызовет ошибки.

Необходимо также добавить правило, что пустое поддерево– является двоичным справочником.

```
is_ordered_check(nil, _, _) :- !.
```

Данное правило будет находиться в начале программы и заканчиваться отсечением, чтобы не рассматривать альтернативные варианты, которые могут привести к окончанию работы программы с ошибкой.

На данный момент программой также не обрабатывается случай, если *is_ordered_check()* будет вызвано, и первым параметром будет число, что возможно в случае обработки листа дерева.

В таком случае необходимо создать еще одно правило, которое уже не будет вызывать рекурсивно *is_ordered_check* для поддеревьев (их нет, так как рассматривается конкретный лист), и проверит, что первый переданный аргумент – число. Для этого будет использоваться встроенный предикат *integer()*, который проверяет, что значение – целое.

```
is_ordered_check(nil, _, _):-!.

is_ordered_check(Value, Max, Min):-
integer(Value),(Max = nil,!; Value < Max),(Min = nil,!; Value > Min).

is_ordered_check(tr(Root, Left, Right), Max, Min):-
(Max = nil,!; Root < Max),(Min = nil,!; Root > Min),
is_ordered_check(Left, Root, Min), is_ordered_check(Right, Max, Root).

is_ordered(BTree) :- is_ordered_check(BTree, nil, nil).
```

Для наглядности разберем работу программы на примере:

```
is_ordered(tr(2,tr(7, nil, nil),tr(3,tr(4, nil, nil), tr(1, nil, nil)))).
```

На рисунки 2 изображены вызовы во врем работы программы. Названия правил были сокращены для удобного отображения.

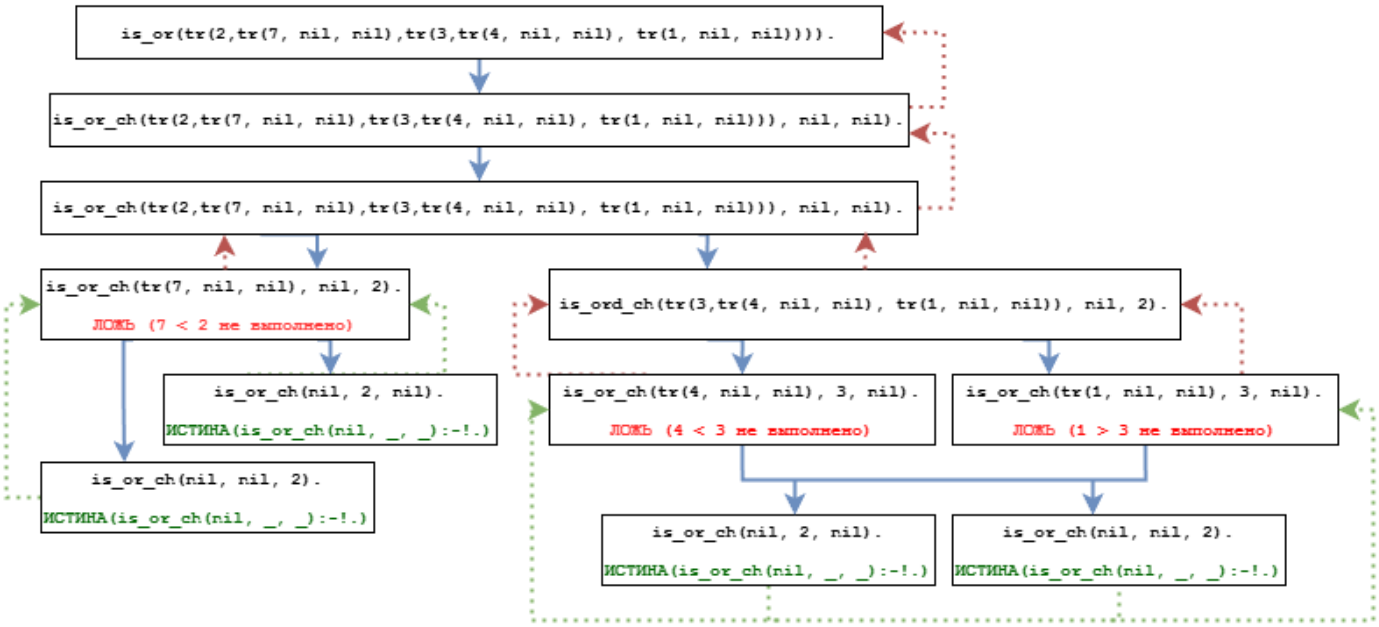

Рисунок 2. Вызов is_ordered(tr(2,tr(7, nil, nil),tr(3,tr(4, nil, nil), tr(1, nil, nil)))).

В результате на выходе программы получим no.

Полный текст программ с комментариями смотри в приложении А.

Примеры вызова правил.

На рисунках 3-6 примеры работы программы по добавлению элемента в упорядоченный список:

```
| ?- list_insert(2, [1,2,3,4], X).

X = [1,2,2,3,4] ?
```

Рисунок 3. Вызов list_insert(2, [1,2,3,4], X).

```
| ?- list_insert(-12, [15,16,80,99], X).

X = [-12,15,16,80,99] ?
```

Рисунок 4. Вызов list_insert(-12, [15,16,80,99], X).

```
| ?- list_insert(123, [], X).

X = [123] ?
```

Рисунок 5. Вызов list_insert(123, [], X).

```
| ?- list_insert(2345, [1,2,3,4], X).

X = [1,2,3,4,2345] ?
```

Рисунок 6. Вызов list_insert(2345, [1,2,3,4], X).

На рисунках 7-10 примеры работы программы по определению, является ли дерево двоичным справочником:

```
| ?- is_ordered(tr(2,tr(7, nil, nil),tr(3,tr(4, nil, nil), tr(1, nil, nil)))).
no
```

Рисунок 7. Вызов is_ordered(tr(2,tr(7, nil, nil),tr(3,tr(4, nil, nil), tr(1, nil, nil)))).

```
| ?- is_ordered(tr(2,tr(1, nil, nil),tr(8,tr(4, nil, nil), tr(12, nil, nil)))).
yes
```

Рисунок 8. Вызов is_ordered(tr(2,tr(1, nil, nil),tr(8,tr(4, nil, nil), tr(12, nil, nil)))).

```
| ?- is_ordered(tr(15,tr(7, 4, 10),tr(23,tr(21, 16, 22), tr(25, 24, 33)))).
yes
```

Рисунок 9. Вызов is_ordered(tr(15,tr(7, 4, 10),tr(23,tr(21, 16, 22), tr(25, 24, 33)))).

```
| ?- is_ordered(tr(14,22,7)).
no
```

Рисунок 10. Вызов is_ordered(tr(14,7,22)).

Выводы.

В ходе выполнения лабораторной работы были описаны правила на языке GNU Prolog. Данные правила позволяют решать следующие задачи: вставка элемента в упорядоченный список, определение, является ли дерево двоичным справочником. Приведены примеры работы программы для случаев, когда дерево является справочником и не является им. Добавление элемента проверено на таких примерах как пустой список, вставки в начало, конец, а также в середину списка.

Зоны ответственности членов бригады:

- Чернякова В.А. – составление отчета.
- Ярусова Т.В. – написание программы;
- Байков Е.С. – написание программы;
- Мамин Р.А. – тестирование программы.

Каждый участник бригады проконтролировал действия других участников и разобрался в проделанной ими работе.

В ходе выполнения лабораторной работы возникли следующие трудности:

- Изначально в задании с двоичным справочником все элементы поддерева сравнивались с корнем на текущем уровне. Это работало не совсем корректно. Например, корень дерева 5. У него есть левое поддерево с корнем 3, у которого также есть поддеревья (например, левый лист 2 и правый лист 20). По определению в правом поддереве значения должны быть больше корня, 20 > 3, правило вернет истину. Но при этом 20 в левом поддереве относительно всего корня дерева 5 => дерево будет ложно определенно двоичным справочником. Поэтому было принято решение на каждом шаге передавать значение корня всего дерева для сравнения, для левых поддеревьев это максимальное значение *Max* (все остальные меньше), для правых – минимальное *Min* (все остальные больше).
- Для переменных *Max* и *Min* нужно было придумать начальную инициализацию, так как в правило по заданию передается только дерево. Было придумано следующее решение: после вызова основного правила *is_ordered* вызывать дополнительно *is_ordered_check*, куда передавать в качестве значений *Min* и *Max* – *nil*, пустое поддерево. Так как условия в правиле изначально выглядели так: *Root* > *Min* и *Root* < *Max*, то при значениях *nil* в ходе работы были получены ошибки, так как нельзя сравнить число и *nil* соответственно. Для этого перед условием сравнения было добавлено условие, определяющие, что *Min/Max* на данный момент принимают значение *nil*. В случае истины надо было сделать так, чтоб дальнейшее сравнение с *Root* не производилось. В материалах по Прологу было найдено отсечение *!*, которое было добавлен после *Min/Max = nil,!*. Это позволило отсечь ту часть, где происходит сравнение с Root и все заработало корректно.
- Также в начале не был предусмотрен вариант, если дерево будет задано не так *tr(2,tr(1,nil,nil),tr(4,nil,nil))*, а *tr(2,1,4),* то есть когда лист задан не через *tr(),* а просто как число. Программа такие случаи обрабатывала некорректно. Решением стало добавление еще одного *is_ordered_check*, но

принимающего на вход *Value, Min, Max* и проверяющего, что *Value* именно число. В таком случае происходило только сравнение с *Min* и *Max*, вызовов для поддеревьев не было, так как правило создано исключительно для проверки листов. А чтобы проверить, что *Value* – число, был использован встроенный предикат *integer(),* информация о нем была найдена в материалах по Прологу.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: parent.pl

```
% Вариант 1.
% Бригада 1 группы 1304 - Чернякова, Ярусова, Байков, Мамин.

% Задание 1. Списки: Вставить число в упорядоченный список.

% Добавление в пустой список
list_insert(X, [], [X]).
% Добавление элемента в упорядоченный список в нужное место
list_insert(X, [Head | Tail], [X, Head | Tail]):-X =< Head.
% Рекурсивное условие
list_insert(X, [Head | Tail], [Head | NewTail]):-X > Head, list_insert(X, Tail, NewTail).

% Задание 2. Деревья: Создайте предикат, проверяющий, что дерево является двоичным справочником.

% Пустое поддерево - двоичный справочник
is_ordered_check(nil, _, _):-!.

% Проверка на соответствие определению двоичного справочника
% Лист - просто число, не tr(VALUE, nil, nil)
% is_ordered_check(Value, Min, Max):-
% integer(Value),(Max = nil,!; Value < Max),(Min = nil,!; Value > Min).

% Проверка на соответствие определению двоичного справочника
% Рекурсивные вызовы для проверки всех правых и левых поддеревьев
is_ordered_check(tr(Root, Left, Right), Min, Max):-
(Max = nil,!; Root < Max),(Min = nil,!; Root > Min),
is_ordered_check(Left, Min, Root), is_ordered_check(Right, Root, Max).

% Начальный вызов программы
```

```
% nil - в качестве начального значения для Min и Max
is_ordered(BTree):-is_ordered_check(BTree, nil, nil).
```