

**Министерство науки и высшего образования Российской Федерации Федеральное государственное бюджетное образовательное учреждение высшего образования**

**«Московский государственный технический университет имени Н.Э. Баумана**

**(национальный исследовательский университет)» (МГТУ им. Н.Э. Баумана)**

**ОТЧЕТ**

По лабораторной работе №3

По курсу «Конструирование компиляторов»

Тема: «Синтаксический разбор с использованием метода рекурсивного спуска»

Вариант 2

Студент:

Апальков Ф.С.

Группа:

ИУ7-23М

Преподаватель:

Ступников А.А.

Москва 2022

**Теоретическая часть**

Одним из наиболее простых и потому одним из наиболее популярных методов нисходящего синтаксического анализа является метод рекурсивного спуска (recursive descent method).

Метод основан на «зашивании» правил грамматики непосредственно в управляющие конструкции распознавателя. Синтаксические анализаторы, работающие по методу рекурсивного спуска без возврата, могут быть построены для класса грамматик, называющегося LL(1). Первая буква L в названии связана с тем, что входная цепочка читается слева направо, вторая буква L означает, что строится левый вывод входной цепочки, 1 означает, что на каждом шаге для принятия решения используется один символ непрочитанной части входной цепочки. Для строгого определения LL(1) грамматики потребуются две функции - FIRST и FOLLOW. Определение. Пусть α ∈ (Ν ∪ Σ) + , x ∈ Σ + , β ∈ (Ν ∪ Σ) \* . Для КС-грамматики G = (Ν, Σ, P, S) определена функция

FIRSTk(α)={x | α =>\* xβ и |x| = k или α =>\* x и |x| < k } ∪ { ε if α =>\* ε } Иначе говоря, множество FIRSTk(α) состоит из всех терминальных префиксов длины k (или меньше, если из α выводится терминальная цепочка длины, меньшей k) терминальных цепочек, выводимых из α. По определению полагают, что FIRSTk (ε ) = { ε }. Определение. Пусть α, γ ∈ (Ν ∪ Σ) \* , β ∈ (Ν ∪ Σ) + , x ∈ Σ + .

Для КС-грамматики G = (Ν, Σ, P, S) определена функция FOLLOWk(β) = { x | S =>\* αβγ and x ∈ FIRSTk (γ) } ∪ { ε if S =>\* αβ } Иначе говоря, множество FOLLOWk(β) состоит из всех цепочек длины k (или меньше) терминальных цепочек, которые могут встречаться непосредственно справа от β в каких-нибудь цепочках, выводимых из аксиомы, причем если αβ выводимая цепочка, то ε тоже принадлежит FOLLOWk(β). Для грамматики LL(1) k=1, и имеет смысл говорить только о функциях FIRST1(α) и FOLLOW1(β), а вместо фразы «терминальных цепочек» говорить «терминальных символов».

Теорема. КС-грамматика G = (Ν, Σ, P, S) является LL(1)-грамматикой тогда и только тогда, когда для каждых двух различных правил A → β и A → γ выполняется условие FIRST1 (β FOLLOW1 (A)) ∩ FIRST1 (γ FOLLOW1 (A)) = ∅ при всех A ∈ Ν.

**Практическая часть**

В методе рекурсивного спуска полностью сохраняются идеи нисходящего разбора, принятые в LL(1)- грамматиках:

• происходит последовательный просмотр входной строки слева-направо;

• очередной символ входной строки является основанием для выбора одной из правых частей правил группы при замене текущего нетерминала;

• терминальные символы входной строки и правой части правила «взаимно уничтожаются»;

• обнаружение нетерминала в правой части рекурсивно повторяет этот же процесс. В методе рекурсивного спуска эти идеи претерпевают следующие изменения:

• каждому нетерминалу соответствует отдельная процедура (функция), распознающая (выбирающая и «закрывающая») одну из правых частей правила, имеющего в левой части этот нетерминал (т.е. для каждой группы правил пишется свой распознаватель);

• во входной строке имеется указатель (индекс) на текущий «закрываемый символ». Этот символ и является основанием для выбора необходимой правой части правила. Сам выбор «зашит» в распознавателе в виде конструкций if или switch. Правила выбора базируются на построении множеств выбирающих символах, как это принято в LL(1)-грамматике;

• просмотр выбранной части реализован в тексте процедуры-распознавателя путем сравнения ожидаемого символа правой части и текущего символа входной строки;

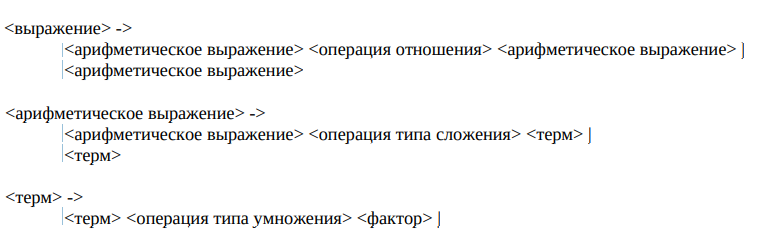
• если в правой части ожидается терминальный символ, и он совпадает с очередным символом входной строки, то символ во входной строке пропускается, а распознаватель переходит к следующему символу правой части;

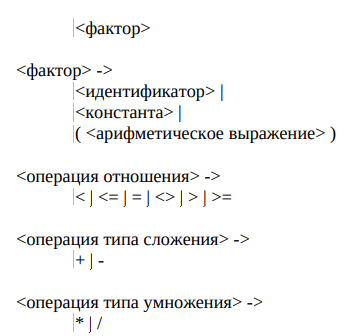
• несовпадение терминального символа правой части и очередного символа входной строки свидетельствует о синтаксической ошибке;

• если в правой части встречается нетерминальный символ, то для него необходимо вызвать аналогичную распознающую процедуру (функцию). Использование рекурсивного спуска позволяет достаточно быстро и наглядно писать программу распознавателя на основе имеющейся грамматики. Главное, чтобы последняя соответствовала требуемому виду. Естественно, возникает вопрос: если грамматика не является LL(1), то существует ли эквивалентная КСграмматика, для которой метод рекурсивного спуска применим? К сожалению, нет алгоритма, отвечающего на поставленный вопрос, т.е. это алгоритмически неразрешимая проблема. Чтобы применить метод рекурсивного спуска, необходимо преобразовать грамматику к виду, в котором множества FIRST не пересекаются. Этот процесс может оказаться сложным.

Поэтому на практике часто используется прием, называемый рекурсивным спуском с возвратами. Для этого лексический анализатор представляется в виде объекта, у которого помимо традиционных методов scan, next и т. п., есть также копирующий конструктор. Затем во всех ситуациях, где может возникнуть неоднозначность, перед началом разбора запоминается текущее состояние лексического анализатора (т.е. заводится копия лексического анализатора) и делается попытка продолжить разбор текста, считая, что рассматривается первая из возможных в данной ситуации конструкций.

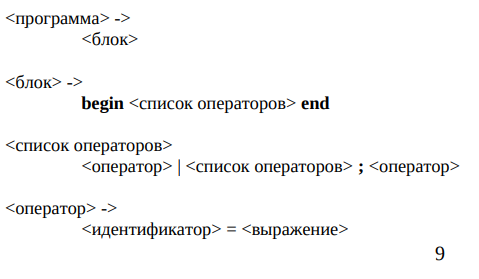
Если этот вариант разбора заканчивается неудачей, то восстанавливается состояние лексического анализатора и делается попытка заново разобрать тот же самый фрагмент с помощью следующего варианта грамматики и т. д. Если все варианты разбора заканчиваются неудачно, то вызывается функция обработки и нейтрализации ошибки. Такой метод разбора потенциально медленнее, чем рекурсивный спуск без возвратов, но в этом случае удается сохранить грамматику в ее оригинальном виде и сэкономить усилия программиста

**Вариант 2.** 



**Задание на лабораторную работу**

Дополнить грамматику блоком, состоящим из последовательности операторов присваивания. Для реализации предлагаются два варианта расширенной грамматики. Вариант в стиле Алгол-Паскаль



**Текст программы**

**Main.py**

from grammairs import \*

# запишем грамматику G2 и присваивание в стиле Паскаля во внутреннее представление

g = Grammair()

p = [

    [['выражение'], ['арифметическоеВыражение', 'операцияОтношения','арифметическоеВыражение']],

    [['выражение'], ['арифметическоеВыражение']],

    [['арифметическоеВыражение'], ['арифметическоеВыражение', 'операцияТипаCложения', 'терм']],

    [['арифметическоеВыражение'], ['терм']],

    [['терм'], ['терм', 'операцияТипаУмножения', 'фактор']],

    [['терм'], ['фактор']],

    [['фактор'], ['var']],

    [['фактор'], ['const']],

    [['фактор'], ['(', 'арифметическоеВыражение', ')']],

    [['операцияОтношения'], ['<']],

    [['операцияОтношения'], ['<=']],

    [['операцияОтношения'], ['==']],

    [['операцияОтношения'], ['<>']],

    [['операцияОтношения'], ['>']],

    [['операцияОтношения'], ['>=']],

    [['операцияТипаCложения'], ['+']],

    [['операцияТипаCложения'], ['-']],

    [['операцияТипаУмножения'], ['\*']],

    [['операцияТипаУмножения'], ['/']],

    [['программа'], ['блок']],

    [['блок'], ['begin', 'списокОператоров', 'end']],

    [['списокОператоров'], ['оператор']],

    [['списокОператоров'], ['списокОператоров', ';', 'оператор']],

    [['оператор'], ['идентификатор', '=', 'выражение']],

    [['оператор'], ['выражение']],

    [['идентификатор'], ['var']],

    [['идентификатор'], ['const']]

]

n = [

    'выражение',

    'арифметическоеВыражение',

    'операцияОтношения',

    'операцияТипаCложения',

    'терм',

    'операцияТипаУмножения',

    'фактор',

    'программа',

    'блок',

    'списокОператоров',

    'оператор',

    'идентификатор',

]

t = [

    'var',

    'const',

    '(', ')',

    '<', '<=', '==', '<>', '>', '>=', '+', '-', '\*', '/',

    'begin',

    'end',

    '=',

    ';'

]

s = 'программа'

fillGrammair(g, t, n, s, p)

print(g)

# удалить левую рекурсию

deleteAllLeftRecursion(g)

print('-------------------------------------------------------')

print('-------------------------------------------------------')

print(g)

# попробуем разобрать примеры

# var + var

# const = var \* ( var + var )

from ATLcreate import \*

tokens1 = 'begin const = var ; var = var + var ; var = const <= var end'.split(' ')

MyTree1 = LLRecursion(g, tokens1, debug=False)

print('Дерево по токенам', tokens1)

if MyTree1:

    MyTree1.printTree()

else:

    print('невозможно построить')

**ATLcreate.py**

from grammairs import \*

import pptree

class ATLTree:

    def \_\_init\_\_(self) -> None:

        self.Root = None

    def printTree(self, flag = False, Current = None, Horizontal = True):

        Verticals = None

        PPVerticals = None

        if flag: #Dubug print tree

            Verticals = [self.Root]

            if self.Root == Current:

                PPVerticals = [pptree.Node(f'>{self.Root.Letter}<:{self.Root.CurInd}')]

            else:

                PPVerticals = [pptree.Node(self.Root.Letter)]

            index = 0

            while True:

                bufChildrens = Verticals[index].Childrens.copy()

                for children in bufChildrens[-1::-1]:

                    Verticals.append(children)

                    if children == Current:

                        PPVerticals.append(pptree.Node(f'>{children.Letter}<:{children.CurInd}', PPVerticals[index]))

                    else:

                        PPVerticals.append(pptree.Node(f'{children.Letter}:{children.CurInd}', PPVerticals[index]))

                index += 1

                if index == len(Verticals): break

        else:

            Verticals = [self.Root]

            PPVerticals = [pptree.Node(self.Root.Letter)]

            index = 0

            while True:

                bufChildrens = Verticals[index].Childrens.copy()

                for children in bufChildrens[-1::-1]:

                    Verticals.append(children)

                    PPVerticals.append(pptree.Node(children.Letter, PPVerticals[index]))

                index += 1

                if index == len(Verticals): break

        pptree.print\_tree(PPVerticals[0], horizontal = Horizontal)

class ATLNode:

    def \_\_init\_\_(self, stype, sletter) -> None:

        self.TypeElement = stype # 'N' - nonterminal \ 'T' - terminal

        self.Childrens = []

        self.Parent = None

        self.Letter = sletter

        self.Status = 'process' # ready

        self.Rules = []

        self.CurInd = None

    def AddChildrens(self, added):

        for elem in added:

            self.Childrens.append(elem)

    def clearChildrens(self):

        self.Childrens = []

    def changeStatus(self, sstatus):

        self.Status = sstatus

def Rstr(BufferRoot):

    """

    Обход вглубину

    """

    ret = []

    count = 0

    for element in BufferRoot.Childrens:

        if count == 1:

            ret.append(BufferRoot)

        for i in Rstr(element):

            ret.append(i)

        count += 1

    if not BufferRoot in ret: ret.append(BufferRoot)

    return ret

def Shstr(BufferRoot):

    """

    Обход в обратную ширину

    """

    allNodes = Rstr(BufferRoot)

    ret = []

    while True:

        if len(allNodes) == 0: break

        for ind in range(len(allNodes)):

            noda = allNodes[ind]

            if len(noda.Childrens) == 0 or all([(i in ret) for i in noda.Childrens]):

                ret.append(noda)

                allNodes.pop(ind)

                break

    return ret

def Rs(root: ATLNode, g:Grammair):

    """

    Вернет все терминалы в дереве слева направо

    """

    buf = Rstr(root)

    ret = []

    for i in buf:

        if i.Letter in g.Terminals:

            ret.append(i.Letter)

    return ret

def nextLeft(root, CurrentLeft, CGMAP):

    buf = Rstr(root)

    Nbuf = []

    for i in buf:

        if i.TypeElement == 'N':

            Nbuf.append(i)

    for k in Nbuf:

        if k.Status == 'process':

            if k != CurrentLeft:

                killChildrens(k)

                FillRules(k, CGMAP)

            return k

    return CurrentLeft

def findPRC(root):

    buf = Rstr(root)

    prc = 0

    for i in buf:

        if i.Status == 'process' and i.TypeElement == 'N':

            prc += 1

    return prc

def FillRules(CLeft: ATLNode, CGMAP):

    CLeft.Rules = CGMAP[CLeft.Letter].copy()

def parceLeft(CLeft: ATLNode, g:Grammair, CGMAP):

    letters = CLeft.Rules.pop()

    newNodes = []

    newNode = None

    for lettr in letters:

        if lettr in g.NonTerminals:

            newNode = ATLNode('N', lettr)

            newNode.Rules = CGMAP[lettr].copy()

        else:

            newNode = ATLNode('T', lettr)

        newNodes.append(newNode)

    CLeft.Childrens = newNodes.copy()

    CLeft.Status = 'ready'

def killChildrens(CLeft: ATLNode):

    CLeft.Childrens = []

    CLeft.Status = 'process'

def FindFather(CLeft, CRoot):

    #среди всех елементов выбрать тот у которого ребенок CLeft

    for element in Rstr(CRoot):

        if CLeft in element.Childrens:

            return element

def goPrevious(CLeft: ATLNode, CLeftStack, CGMAP, CRoot):

    counter = 0

    #если в списке разбираемых нод есть еще кто-то

    #обновить списокдоступных правил для текущей ноды

    #убить ее дете

    #перейти в предыдущую разбираемую ноду

    #если у нее еще есть правила для разбора, убить ее детей и удалить верхнее правило, вернуть ее из функции

    #если правил нет, то убить ее детей, обновить список доступных правил и отправиться дальше назад

    #если позадди никого нет,то это конец разбора, вернуть none

    Papa = FindFather(CLeft, CRoot)

    # if Papa == None:

    #     return None

    killChildrens(CLeft)

    FillRules(CLeft, CGMAP)

    if CLeft == CLeftStack[-1]: CLeftStack.pop()

    # if Papa != CLeftStack[-1]:

    #     FillRules(Papa, CGMAP)

    while True:

        if len(CLeftStack) > 0:

            counter += 1

            CLeft = CLeftStack.pop()

            1+1

            if len(CLeft.Rules) > 0:

                killChildrens(CLeft)

                if not CLeft in Rstr(Papa) or CLeft == Papa:

                    Papa.Childrens = []

                    Papa.Status = 'process'

                return CLeft

        else:

            return None

    #откатиться до разбираемого элемента с номером ind

def countCoincidence(ctokens, Rstokens):

    myTokens = Rstokens.copy()

    count = 0

    if len(myTokens) == 0:

        return count

    now = myTokens.pop(0)

    # for lettr in ctokens:

    #     if lettr == now:

    #         count += 1

    #         if len(myTokens) == 0:

    #             return count

    #         else:

    #             now = myTokens.pop(0)

    for lettr in ctokens:

        if now == lettr:

            count += 1

            if len(myTokens) == 0:

                return count

            else:

                now = myTokens.pop(0)

        else:

            return count

    return count

def LLRecursion(g:Grammair, tokens, debug = False) -> ATLTree:

    ind = 1

    GMAP = g.convertToDict()

    retTree = ATLTree()

    ROOT = ATLNode('N', g.Start)

    FillRules(ROOT, GMAP)

    ROOT.CurInd = ind

    retTree.Root = ROOT

    Left = ROOT

    LeftStack = []

    processFlag = False

    while True:

        RScount = countCoincidence(tokens, Rs(ROOT, g))

        if debug:

            print()

            retTree.printTree(flag=True, Current=Left)

            print()

            print('RScount', RScount, 'ind', ind)

            # print('RScount', RScount, 'ind', ind)

            print(Rs(ROOT, g))

            print(tokens)

            # lols = ''

            # lollen = 0

            # for loli in Rstr(ROOT):

            #     lols += f'<{loli.Letter}>'

            #     lollen += 1

            # print(lols, lollen)

        bufRSlen = len(Rs(ROOT, g))

        # if bufRSlen > ind:

        #     ind += 1

        #     continue

        if bufRSlen > len(tokens):

            #длина разбора больше

            # Удалить детей и верхнее правило или Идти назад по разбираемым верщинам в поиске той

            # у которой еще есть правила для разбора

            # У нее убить детей , дропнуть последнее разобранное правило

            # И сделать ее неразобранной

            if len(Left.Rules) > 0:

                #Если есть еще правило

                killChildrens(Left)

            else:

                #сделать Left неразобранным, обновить его

                # и искать предыдущий left который можно еще как-то разобрать

                # убить его детей и дропнуть верхнее правило

                killChildrens(Left)

                FillRules(Left, GMAP)

                Left = goPrevious(Left, LeftStack, GMAP, ROOT)

                if Left == None: return None

                ind = Left.CurInd

        #разбор несошелся

        if ( RScount < ind and RScount != len(tokens) ) or processFlag:

            processFlag = False

            #если можно разобрать текущю вершину

            if Left.Status == 'process':

                #раскрыть Left по верхнему правилу из ее доступных (там точно есть правила, ведь мы ее только добавили)

                parceLeft(Left, g, GMAP)

                Left.Status = 'ready'

                Left.CurInd = ind

                continue

            #если текущая вершина уже разобрана

            else:

                #Левый ребенок не разобран?

                if Left.Childrens[0].Status == 'process' and Left.Childrens[0].TypeElement == 'N':

                    #Заносим текущую вершину в стек разбора

                    #и делаем левого ребенка текущим Left

                    Left.Status = 'ready'

                    LeftStack.append(Left)

                    Left = Left.Childrens[0]

                    #смотрим дальше

                    # if ind != 1:

                    #     ind -= 1

                    continue

                #нет левого ребенка для разбора

                else:

                    #попытаемся разобрать как-то по-другому, или переходить к родителям

                    #пока не сможем разобрать как-то по-другому их

                    #если разбирать как-то по-другому некого, то это - ошибка

                    if len(Left.Rules)>0:

                        #если еще есть правила, убить детей и дропнуть верхнее правило

                        killChildrens(Left)

                        #Left.Rules.pop()

                        1+1

                    else:

                        # парвил нет, переходим на предыдущее правило которое можно исправить

                        Left = goPrevious(Left, LeftStack, GMAP, ROOT)

                        if Left == None: return None

                        ind = Left.CurInd

                        1+1

                    if Left == None: # если перейти неудалось то все

                        return None

                    #идем дальше

                    # if ind != 1:

                    #     ind -= 1

                    continue

        else:

            #разбор сходится до ind символа

            #Left меняется на следующий неразобраный левый нетерменал, увеличиваем совпадающую строку

            ind += 1

            #если проверена всетокены

            if ind == len(tokens) + 1:

                #если больше нет неразобранных нетерменалов

                #то все

                prc = findPRC(ROOT)

                if prc == 0:

                    return retTree

                #Если есть неразобранный нетерминал, то падаем в обратно в разбор

                else:

                    processFlag = True

            #продолжить разбор

            if not Left in LeftStack:

                LeftStack.append(Left)

            if RScount >= ind - 1:

                Left = nextLeft(ROOT, Left, GMAP)

            continue

**Тесты**

**Terminals:**

**var const ( ) < <= == <> > >=**

**+ - \* / begin end = ;**

**Nonterminals:**

**выражение арифметическоеВыражение операцияОтношения операцияТипаCложения терм операцияТипаУмножения фактор программа блок списокОператоров**

**оператор идентификатор арифметическоеВыражение` терм` списокОператоров`**

**Start: программа**

**Rules:**

**выражение -> <арифметическоеВыражение><операцияОтношения><арифметическоеВыражение>**

**выражение -> <арифметическоеВыражение>**

**фактор -> <var>**

**фактор -> <const>**

**фактор -> <(><арифметическоеВыражение><)>**

**операцияОтношения -> <<>**

**операцияОтношения -> <<=>**

**операцияОтношения -> <==>**

**операцияОтношения -> <<>>**

**операцияОтношения -> <>>**

**операцияОтношения -> <>=>**

**операцияТипаCложения -> <+>**

**операцияТипаCложения -> <->**

**операцияТипаУмножения -> <\*>**

**операцияТипаУмножения -> </>**

**программа -> <блок>**

**блок -> <begin><списокОператоров><end>**

**оператор -> <идентификатор><=><выражение>**

**оператор -> <выражение>**

**идентификатор -> <var>**

**идентификатор -> <const>**

**арифметическоеВыражение` -> <операцияТипаCложения><терм><арифметическоеВыражение`>**

**арифметическоеВыражение` -> <операцияТипаCложения><терм>**

**арифметическоеВыражение -> <терм><арифметическоеВыражение`>**

**арифметическоеВыражение -> <терм>**

**терм` -> <операцияТипаУмножения><фактор><терм`>**

**терм` -> <операцияТипаУмножения><фактор>**

**терм -> <фактор><терм`>**

**терм -> <фактор>**

**списокОператоров -> <оператор><списокОператоров`>**

**списокОператоров -> <оператор>**

**списокОператоров` -> <;><оператор><списокОператоров`>**

**списокОператоров` -> <;><оператор>**

**Дерево по токенам ['begin', 'const', '=', 'var', ';', 'var', '=', 'var', '+', 'var', ';', 'var', '=', 'const', '<=', 'var', 'end']**

**программа┐**

**│ ┌end**

**│ ├begin**

**└блок┤**

**│ ┌=**

**│ ├идентификатор┐**

**│ │ └const**

**│ ┌оператор┤**

**│ │ └выражение┐**

**│ │ └арифметическоеВыражение┐**

**│ │ └терм┐**

**│ │ └фактор┐**

**│ │ └var**

**└списокОператоров┤**

**│ ┌;**

**│ │ ┌=**

**│ │ ├идентификатор┐**

**│ │ │ └var**

**│ ├оператор┤**

**│ │ └выражение┐**

**│ │ │ ┌терм┐**

**│ │ │ │ └фактор┐**

**│ │ │ │ └var**

**│ │ └арифметическоеВыражение┤**

**│ │ │ ┌операцияТипаCложения┐**

**│ │ │ │ └+**

**│ │ └арифметическоеВыражение`┤**

**│ │ └терм┐**

**│ │ └фактор┐**

**│ │ └var**

**└списокОператоров`┤**

**│ ┌;**

**└списокОператоров`┤**

**│ ┌=**

**│ ├идентификатор┐**

**│ │ └var**

**└оператор┤**

**│ ┌операцияОтношения┐**

**│ │ └<=**

**└выражение┤**

**├арифметическоеВыражение┐**

**│ └терм┐**

**│ └фактор┐**

**│ └const**

**└арифметическоеВыражение┐**

**└терм┐**

**└фактор┐**

**└var**

**Вывод**

Дерево разбора строится достаточно быстро для строки токенов не более 100 символов. Для ускорения процесса создания дерева можно попробовать разбить строку токенов на подстроки и разбирать их от их корня, а не от общего. Данная программа рассчитана на полный перебор всех возможных вариантов построения дерева. В качестве левого правила выбирается сначала нижнее правило из списка, а далее если оно не подошло выбираются следующие. Работа может быть использована для создания ATL дерева, если грамматика не будет иметь левой рекурсии. Левую рекурсию можно устранить, используя программу из второй лабораторной работы.