

**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 №2

По курсу «Конструирование компиляторов»

Тема: «Преобразования грамматик»

Вариант 2

Студент:

Апальков Ф.С.

Группа:

ИУ7-23М

Преподаватель:

Ступников А.А.

Москва 2022

**Цель и задачи работы**

Цель работы: приобретение практических навыков реализации наиболее важных (но не всех) видов преобразований грамматик, чтобы удовлетворить требованиям алгоритмов синтаксического разбора.

Задачи работы:

1) Принять к сведению соглашения об обозначениях, принятые в литературе по теории формальных языков и грамматик и кратко описанные в приложении.

2) Познакомиться с основными понятиями и определениями теории формальных языков и грамматик.

3) Детально разобраться в алгоритме устранения левой рекурсии.

4) Разработать, тестировать и отладить программу устранения левой рекурсии.

5) Разработать, тестировать и отладить программу преобразования грамматики в соответствии с предложенным вариантом

**Теоретическая часть**

Данную грамматику часто требуется модифицировать так, чтобы порождаемый ею язык приобрел нужную структуру. Общего алгоритмического метода, который придавал бы данному языку произвольную структуру, не существует.

Но с помощью ряда преобразований можно видоизменить грамматику, не испортив порождаемого грамматикой языка. В данной лабораторной работе рассматривается несколько преобразований такого рода. Для быстрого погружения в данную предметную область рекомендуется самостоятельно проработать главу 3 (параграфы 3.1, 3.2, 3.3) учебного пособия [3].

**Варианты заданий на лабораторную работу**

Общий вариант для всех:

Устранение левой рекурсии. Определение. Не терминал A КС-грамматики G = (Ν, Σ, P, S) называется рекурсивным, если A =>+ αAβ для некоторых α и β. Если α = ε, то A называется леворекурсивным. Аналогично, если β = ε, то А называется праворекурсивным. Грамматика, имеющая хотя бы один леворекурсивный нетерминал, называется леворекурсивной. Аналогично определяется праворекурсивная грамматика.

Грамматика, в которой все нетерминалы, кроме, быть может, начального символа, рекурсивные, называется рекурсивной. Некоторые из алгоритмов разбора не могут работать с леворекурсивными грамматиками. Можно показать, что каждый КС-язык определяется хотя бы одной не леворекурсивной грамматикой.

Постройте программу, которая в качестве входа принимает приведенную КС-грамматику G = (Ν, Σ, P, S) и преобразует ее в эквивалентную КС-грамматику G' без левой рекурсии. Указания. 1) Проработать самостоятельно п. 4.3.3. и п. 4.3.4. [2]. 2) Воспользоваться алгоритмом 2.13. При тестировании воспользоваться примером 2.27. [1]. 3) Воспользоваться алгоритмами 4.8 и 4.10. При тестировании воспользоваться примерами 4.7., 4.9. и 4.11. [2]. 4) Устранять надо не только непосредственную (immediate), но и косвенную (indirect) рекурсию. Этот вопрос подробно затронут в [4]. 5) После устранения левой рекурсии можно применить левую факторизацию.

Вариант 2.

Устранение бесполезных символов. Определение. Назовем символ X ∈ Ν ∪ Σ бесполезным в КС-грамматике G = (Ν, Σ, P, S), если в ней нет вывода вида S =>\* wXy =>\* wxy, где w, х, у принадлежат Σ\*. Чтобы установить, бесполезен ли нетерминал А, надо построить сначала алгоритм, выясняющий, может ли нетерминал порождать какие-нибудь терминальные цепочки, т. е. решающий проблему пустоты множества {w | А =>\* w, w ∈ Σ\*}.

Постройте программу, которая в качестве входа принимает произвольную КС-грамматику G = (Ν, Σ, P, S) и преобразует ее в эквивалентную КС-грамматику G' = (Ν', Σ', P', S'), не содержащую бесполезных символов. Указания. Воспользоваться алгоритмом 2.9. [1]. При тестировании воспользоваться примером 2.22. и упражнением 2.4.6. [1]

**Текст программы**

**Grammairs.py**

class Grammair:

    def \_\_init\_\_(self) -> None:

        self.Terminals = []

        self.NonTerminals = []

        self.Rules = []

        self.Start = None

        self.Epsilon = 'eps'

    def setTerminal(self, letter: str) -> None:

        self.Terminals.append(letter)

    def setNonTerminal(self, letter: str) -> None:

        self.NonTerminals.append(letter)

    def setStart(self, letter: str) -> None:

        self.Start = letter

    def setRule(self, LeftLetters: list, RightLetters: list) -> None:

        self.Rules.append([LeftLetters, RightLetters])

    def \_\_str\_\_(self) -> str:

        buf = ''

        buf += 'Terminals:\n'

        c = 0

        for letter in self.Terminals:

            buf += ' ' + letter

            c += 1

            if c == 10:

                buf += '\n'

                c = 0

        buf += '\nNonterminals:\n'

        c = 0

        for letter in self.NonTerminals:

            buf += ' ' + letter

            c += 1

            if c == 10:

                buf += '\n'

                c = 0

        buf += f'\nStart: {self.Start}\n'

        buf += 'Rules:\n'

        for rule in self.Rules:

            left = ''

            right = ''

            for letter in rule[0]:

                left += letter

            for letter in rule[1]:

                right += f'<{letter}>'

            strbuf = f' {left} -> {right}'

            buf += strbuf + '\n'

        return buf

    def convertToDict(self):

        retDict = {}

        for rule in self.Rules:

            leftSide = getLeft(rule)[0]

            rightSide = getRight(rule)

            if retDict.get(leftSide):

                retDict[leftSide].append(rightSide)

            else:

                retDict[leftSide] = [rightSide]

        return retDict

def findEpsGenerating(g: Grammair) -> list:

    # 1. Найти все ε-правила. Составить множество, состоящее из нетерминалов, входящих в левые части таких правил.

    # Правила вида A→ε называются ε-правилами (англ. ε-rule).

    epsGenerating = []

    for rule in g.Rules:

        if getRight(rule) == [g.Epsilon]:

            for letter in getLeft(rule):

                if letter in g.NonTerminals:

                    epsGenerating.append(letter)

    # 3. Если на шаге 2 множество изменилось, то повторить шаг 2.

    change = True

    while change:

        change = False

        # 2. Перебираем правила грамматики Γ. Если найдено правило A→C1C2…Ck, для которого верно,

        # что каждый Ci принадлежит множеству, то добавить A в множество.

        for rule in g.Rules:

            flag = True

            for letter in getRight(rule):

                if not letter in epsGenerating:

                    flag = False

            if flag:

                l = getLeft(rule)[0]

                if not l in epsGenerating:

                    change = True

                    epsGenerating.append(l)

    return epsGenerating

#стремная магия

from itertools import chain, combinations

def powerset(iterable):

    # powerset([1,2,3]) --> () (1,) (2,) (3,) (1,2) (1,3) (2,3) (1,2,3)

    s = list(iterable)

    return chain.from\_iterable(combinations(s, r) for r in range(len(s)+1))

def magiclol(input\_list):

    return list(map(list, powerset(input\_list)))[1:]

# начало магии

def magicConvert(inp\_list):

    k = magiclol(inp\_list)

    buf = []

    buf.append([None for k in range(len(inp\_list))])

    for element in k:

        microbuf = []

        for letter in inp\_list:

            if letter in element:

                microbuf.append(letter)

            else:

                microbuf.append(None)

        buf.append(microbuf)

    return buf

#конец магии

def deleteEpsGenerating(g: Grammair):

    flagStartSymbToEps = False

    for rule in g.Rules:

        if getLeft(rule) == [g.Start] and getRight(rule) == [g.Epsilon]:

            flagStartSymbToEps = True

    #1. Добавить все правила из P в P′.

    newRules = g.Rules.copy()

    #2. Найти все ε-порождаюшие нетерминалы.

    epsGenerating = findEpsGenerating(g)

    #3. Для каждого правила вида A→α0B1α1B2α2…Bkαk  (где αi — последовательности из терминалов и нетерминалов,

    # Bj — ε-порождающие нетерминалы) добавить в P′ все возможные варианты правил, в которых либо присутствует,

    # либо удалён каждый из нетерминалов Bj(1⩽j⩽k).

    for rule in g.Rules:

        epsGen = []

        pattern = []

        leftPart = getLeft(rule)

        for letter in getRight(rule):

            if not letter in epsGenerating:

                pattern.append([letter])

            else:

                pattern.append([])

                epsGen.append(letter)

        magicList = magicConvert(epsGen)

        for element in magicList:

            patternedBuf = []

            index = 0

            for inpattern in pattern:

                if inpattern == []:

                    if element[index] != None:

                        patternedBuf.append([element[index]])

                    else:

                        patternedBuf.append([])

                else:

                    patternedBuf.append(inpattern)

                index += 1

            rightPart = []

            for ii in patternedBuf:

                if ii != []:

                    rightPart.append(ii[0])

            bufnewrule = [leftPart, rightPart]

            if not bufnewrule in newRules and rightPart != []:

                newRules.append(bufnewrule)

    # 4. Удалить все ε-правила из P′.

    new\_newRules = []

    for rule in newRules:

        if getRight(rule) != [g.Epsilon]:

            new\_newRules.append(rule)

    #5. Если в исходной грамматике Γ выводилось ε, то необходимо добавить новый нетерминал S′,

    # сделать его стартовым, добавить правило S′→S|ε.

    if flagStartSymbToEps:

        newStartLetter = g.Start + '`'

        newStartRule1 = [[newStartLetter], [g.Start]]

        newStartRule2 = [[newStartLetter], [g.Epsilon]]

        new\_newRules.append(newStartRule1)

        new\_newRules.append(newStartRule2)

        g.Start = newStartLetter

        g.NonTerminals.append(newStartLetter)

    # обновить правила в грамматике

    g.Rules = new\_newRules

def findNonGenerating(g:Grammair):

    # Шаг 0. Множество порождающих нетерминалов пустое.

    generating = []

    # Шаг 1. Находим правила, не содержащие нетерминалов в правых частях и добавляем нетерминалы,

    # встречающихся в левых частях таких правил, в множество.

    for rule in g.Rules:

        flag = True

        for letter in getRight(rule):

            if letter in g.NonTerminals:

                flag = False

        if flag:

            for letter in getLeft(rule):

                generating.append(letter)

    # Шаг 2. Если найдено такое правило, что все нетерминалы, стоящие в его правой части, уже входят в множество,

    # то добавим в множество нетерминалы, стоящие в его левой части.

    change = True

    while change:

        change = False

        for rule in g.Rules:

            flag = True

            for letter in getRight(rule):

                if letter in g.NonTerminals:

                    if not letter in generating:

                        flag = False

            if flag:

                for letter in getLeft(rule):

                    if letter in g.NonTerminals:

                        if not letter in generating:

                            generating.append(letter)

                            change = True

    # Шаг 3. Повторим предыдущий шаг, если множество порождающих нетерминалов изменилось.

    #В результате получаем множество всех порождающих нетерминалов грамматики, а все нетерминалы,

    # не попавшие в него, являются непорождающими.

    nonGener = []

    for letter in g.NonTerminals:

        if not letter in generating:

            nonGener.append(letter)

    return nonGener

def findUnreachable(g: Grammair):

    #Шаг 0. Множество достижимых нетерминалов состоит из единственного элемента: {S}.

    reachable = [g.Start]

    #Шаг 1. Если найдено правило, в левой части которого стоит нетерминал, содержащийся в множестве,

    # добавим в множество все нетерминалы из правой части.

    change = True

    while change:

        change = False

        for rule in g.Rules:

            flag = False

            for letter in getLeft(rule):

                if letter in reachable:

                    flag = True

            if flag:

                for letter in getRight(rule):

                    if letter in g.NonTerminals:

                        if not letter in reachable:

                            reachable.append(letter)

                            change = True

    # Шаг 2. Повторим предыдущий шаг, если множество порождающих нетерминалов изменилось.

    # Получаем множество всех достижимых нетерминалов, а нетерминалы, не попавшие в него, являются недостижимыми.

    unreachable = []

    for letter in g.NonTerminals:

        if not letter in reachable:

            unreachable.append(letter)

    return unreachable

def deleteUseless(g:Grammair):

    # Удалить из грамматики правила, содержащие непорождающие нетерминалы.

    nonGen = findNonGenerating(g)

    newRules = []

    newNonTerms = []

    for rule in g.Rules:

        flag = False

        for letter in getRight(rule):

            if letter in nonGen:

                flag = True

        for letter in getLeft(rule):

            if letter in nonGen:

                flag = True

        if not flag:

            if not rule in newRules:

                newRules.append(rule)

    for letter in g.NonTerminals:

        if not letter in nonGen:

            if not letter in newNonTerms:

                newNonTerms.append(letter)

    g.NonTerminals = newNonTerms

    g.Rules = newRules

    #Удалить из грамматики правила, содержащие недостижимые нетерминалы.

    nonGen = findUnreachable(g)

    newRules = []

    newNonTerms = []

    for rule in g.Rules:

        flag = False

        for letter in getRight(rule):

            if letter in nonGen:

                flag = True

        for letter in getLeft(rule):

            if letter in nonGen:

                flag = True

        if not flag:

            if not rule in newRules:

                newRules.append(rule)

    for letter in g.NonTerminals:

        if not letter in nonGen:

            if not letter in newNonTerms:

                newNonTerms.append(letter)

    g.NonTerminals = newNonTerms

    g.Rules = newRules

def deleteLeftRecursion(g: Grammair, Ai):

    """

        Удаляет непосредственную левую рекурсию из грамматики g для символа Ai

    """

    A = g.NonTerminals.copy()

    index = A.index(Ai)

    B = A[index]

    A = [B]

    toDeleteRules = []

    toAddRules = []

    toAddNonTerminals = []

    RULES = g.Rules.copy()

    for i in range(len(A)):

        # устранить непосредственную левую рекурсию

        flagToRecursionForward = False

        bufToAppendRules = []

        bufToDeleteRules = []

        newShtrish = A[i] + '`'

        for rule in RULES:

            if getLeft(rule)[0] == A[i]:

                if getRight(rule)[0] == A[i]:

                    flagToRecursionForward = True #есть хотябы одно леворекурсивное правило

                    # Собрать правила из A[i] -> A[i]alpha[1]|...|A[i]alpha[n]

                    # Такого вида: 1) A[i]` -> alpha[1]A[i]`|...|alpha[n]A[i]`

                    #              2) A[i]` -> alpha[1]|...|alpha[i]

                    if not rule in bufToDeleteRules:

                        bufToDeleteRules.append(rule)

                    newRuleBuf1 = [[newShtrish]]

                    newRuleBuf2 = [[newShtrish]]

                    alphaBuf = getRight(rule)[1:len(getRight(rule))]

                    rbuf = []

                    for lettr in alphaBuf: rbuf.append(lettr)

                    if rbuf != [g.Epsilon]:

                        rbuf.append(newShtrish)

                        newRuleBuf1.append(rbuf)

                        newRuleBuf2.append(alphaBuf)

                        if not newRuleBuf1 in bufToAppendRules:

                            bufToAppendRules.append(newRuleBuf1)

                        if not newRuleBuf2 in bufToAppendRules:

                            bufToAppendRules.append(newRuleBuf2)

                else:

                    # собрать правила из A[i] -> betta[1]|...|betta[m]

                    # Вида 1) A[i] -> betta[1]A[i]`|...|betta[m]A[i]`

                    #      2) A[i] -> betta[1]|...|betta[m]

                    if not rule in bufToDeleteRules:

                        bufToDeleteRules.append(rule)

                    newRuleBuf1 = [[A[i]]]

                    newRuleBuf2 = [[A[i]]]

                    bettaBuf = getRight(rule)

                    rbuf = []

                    for lettr in bettaBuf: rbuf.append(lettr)

                    rbuf.append(newShtrish)

                    newRuleBuf1.append(rbuf)

                    newRuleBuf2.append(bettaBuf)

                    if not newRuleBuf1 in bufToAppendRules:

                            bufToAppendRules.append(newRuleBuf1)

                    if not newRuleBuf2 in bufToAppendRules:

                        bufToAppendRules.append(newRuleBuf2)

        # добавим правила если была хоть одна леворекурсивная продукция

        if flagToRecursionForward:

            #добавим новый нетерминал

            toAddNonTerminals.append(newShtrish)

            for microRule in bufToAppendRules:

                if not microRule in toAddRules:

                    toAddRules.append(microRule)

            for microRule in bufToDeleteRules:

                if not microRule in toDeleteRules:

                    toDeleteRules.append(microRule)

    newRules = []

    for rule in g.Rules:

        if not rule in toDeleteRules:

            newRules.append(rule)

    for rule in toAddRules:

        if not rule in newRules:

            newRules.append(rule)

    g.Rules = newRules

    for newNonTermMicro in toAddNonTerminals:

        g.NonTerminals.append(newNonTermMicro)

def deleteAllLeftRecursion(g: Grammair):

    """

        Устраняет произвольную левую рекурсию из грамматики g

    """

    #избавиться от e переходов

    deleteEpsGenerating(g)

    A = g.NonTerminals.copy()

    for i in range(len(A)):

        toDeleteRules = []

        toAddRules = []

        for j in range(i+1):

            for rule in g.Rules:

                # Для каждого правила вида A[i] -> A[j]gamma

                # Удалить такое правило

                if getLeft(rule)[0] == A[i] and getRight(rule)[0] == A[j]:

                    if not rule in toDeleteRules:

                        toDeleteRules.append(rule)

                bufGamma = []

                r = getRight(rule)[1: len(getRight(rule))]

                for lettr in r:

                    bufGamma.append(lettr)

                # Для каждого праивла вида Q -> x[i] из правил вида A[j] -> delta[i]

                # Добавть правило вида A[i] -> x[i]gamma

                for microRule in g.Rules:

                    if getLeft(microRule)[0] == A[j]:

                        bufDelta = getRight(microRule)

                        bufToNewRule = bufDelta.copy()

                        for lettr in bufGamma:

                            bufToNewRule.append(lettr)

                        if not [[A[i]], bufToNewRule] in toAddRules:

                            toAddRules.append([[A[i]], bufToNewRule])

            # Добавить и удалить полученные правила

                new\_Rules = []

                for microRule in g.Rules:

                    if not microRule in toDeleteRules:

                        new\_Rules.append(microRule)

                for microRule in toAddRules:

                    if not microRule in new\_Rules:

                        new\_Rules.append(microRule)

            # Устранить непосредственную левую рекурсию для символа A[i]

            deleteLeftRecursion(g, A[i])

def fillGrammair(G: Grammair, Terms: list, NonTerms: list, Start: str, Rules: list) -> None:

    for letter in Terms:

        G.setTerminal(letter)

    for letter in NonTerms:

        G.setNonTerminal(letter)

    G.setStart(Start)

    for rule in Rules:

        G.setRule(rule[0], rule[1])

def getLeft(rule: list) -> list:

    return rule[0]

def getRight(rule: list) -> list:

    return rule[1]

# g = Grammair()

# t = 'a,+,\*,(,)'.split(',')

# n = 'E,T,F'.split(',')

# s = 'E'

# r = [

#     [['E'], ['E', '+', 'T']],

#     [['E'], ['T']],

#     [['T'], ['T', '\*', 'F']],

#     [['T'], ['F']],

#     [['F'], ['a']],

#     [['F'], ['(', 'E', ')']],

# ]

# fillGrammair(g, t, n, s, r)

# print(g)

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

# deleteAllLeftRecursion(g)

# print(g)

**Main.py**

# Получить грамматику G0

from myParser import Parser

p = Parser()

Bufer = p.INView('G0.xml')

# Получить внутренне представление грамматики G0

from grammairs import Grammair, deleteUseless, deleteAllLeftRecursion, fillGrammair

g = Bufer['Grammair']

# Печать Грамматики до манипуляций

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

print(g)

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

# Удаление бесполезных символов

print('Удалить бесполезные символы')

deleteUseless(g)

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

print(g)

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

# Удаление левой рекурсии

print('Удалить произвольную левую рекурсию')

deleteAllLeftRecursion(g)

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

print(g)

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

print('=======================================================================')

print('=======================================================================')

print('Еще примеры')

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

print('Удаление бесполезных символов')

pg = Grammair()

t = 's,a,f'.split(',')

n = 'A,B,S,E,F'.split(',')

s = 'S'

r = [

    [['S'], ['A', 'S']],

    [['S'], ['s']],

    [['E'], ['E', 'F']],

    [['E'], ['F', 'F']],

    [['A'], ['a']],

    [['F'], ['f']],

]

fillGrammair(pg, t, n, s, r)

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

print(pg)

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

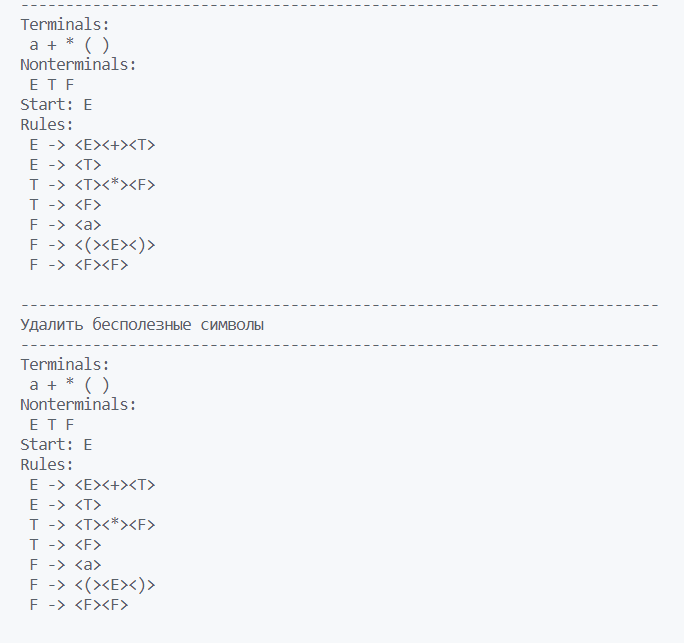
deleteUseless(pg)

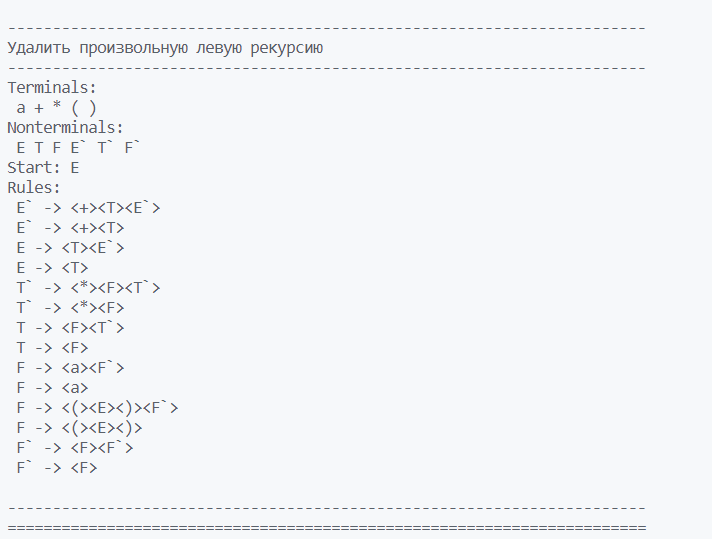
print('-----------------------------------------------------------------------')

print(pg)

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

**Тесты**





**Вывод**

Устранение левой рекурсии и бесполезных символов происходит быстро для грамматик с количеством правил меньше 100, далее процесс начинает занимать огромное количество времени. Это можно исправить, пользуясь не списками, а хэш таблицами. Так же для ускорения процесса, можно отдельно избавляться от рекурсии только для той части грамматики, где это непосредственно необходимо, избегая правил вида терминал -> символ. Их можно после основного процесса сделать отдельно.

В результате работы получены навыки создания грамматик и их преобразования.