



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)

ФАКУЛЬТЕТ «Информатика и системы управления (ИУ)»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии (ИУ7)»

ОТЧЕТ

по лабораторной работе № 2
по курсу «Конструирование компиляторов»
на тему: «Преобразования грамматик»
Вариант № 6

Студент ИУ7-22М
(Группа)

(Подпись, дата)

И. А. Малышев
(И. О. Фамилия)

Преподаватель

(Подпись, дата)

А. А. Ступников
(И. О. Фамилия)

2024 г.

1 Выполнение лабораторной работы

1.1 Общий вариант для всех: Устранение левой рекурсии.

Определение. Нетерминал A КС-грамматики $G = (N, \Sigma, P, S)$ называется рекурсивным, если $A \Rightarrow +\alpha A\beta$ для некоторых α и β . Если $\alpha = \epsilon$, то A называется леворекурсивным. Аналогично, если $\beta = \epsilon$, то A 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 $G = (N, \Sigma, P, S)$ и преобразует ее в эквивалентную КС-грамматику G' без левой рекурсии.

1.2 Преобразование к нормальной форме Хомского.

Определение. КС-грамматика $G = (N, \Sigma, P, S)$ называется грамматикой в нормальной форме Хомского (или в бинарной нормальной форме), если каждое правило из P имеет один из следующих видов:

1. $A \rightarrow BC$, где A, B и C принадлежат N ,
2. $A \rightarrow a$, где $a \in \Sigma$,
3. $S \rightarrow \epsilon$, если $\epsilon \in L(G)$, причем S не встречается в правых частях правил.

Можно показать, что каждый КС-язык порождается грамматикой в нормальной форме Хомского. Этот результат полезен в случаях, когда требуется простая форма представления КС-языка.

Постройте программу, которая в качестве входа принимает приведенную КС-грамматику $G = (N, \Sigma, P, S)$ и преобразует ее в эквивалентную КС-грамматику G' в нормальной форме Хомского.

1.3 Результаты работы программы

Результаты работы программы по преобразованию грамматик приведены на рисунках 1.1–1.14.

1.3.1 Устранение левой рекурсии

```
G = ({E, T, F}, {+, *, (, ), a}, P, E), где P состоит из правил:  
E → E + T | T  
T → T * F | F  
F → a | ( E )
```

Рисунок 1.1 – Исходная грамматика для удаления левой рекурсии (пример 1)

```
G = ({E, E', T, T', F}, {+, *, (, ), a}, P, E), где P состоит из правил:  
E → T E'  
E' → + T E' | ε  
T → F T'  
T' → * F T' | ε  
F → a | ( E )
```

Рисунок 1.2 – Грамматика после удаления левой рекурсии (пример 1)

```
G = ({S, A}, {a, b, c, d}, P, A), где P состоит из правил:  
S → A a | b  
A → A c | S d | ε
```

Рисунок 1.3 – Исходная грамматика для удаления левой рекурсии (пример 2)

```
G = ({S, A, A'}, {a, b, c, d}, P, A), где P состоит из правил:  
S → A a | b  
A → b d A' | A'  
A' → c A' | a d A' | ε
```

Рисунок 1.4 – Грамматика после удаления левой рекурсии (пример 2)

```

G = ({E, T, F}, {+, -, *, /, (, ), id}, P, E), где P состоит из правил:
E -> E + T | E - T | T
T -> T * F | T / F | F
F -> ( E ) id

```

Рисунок 1.5 – Исходная грамматика для удаления левой рекурсии (пример 3)

```

G = ({E, E', T, T', F}, {+, -, *, /, (, ), id}, P, E), где P состоит из правил:
E -> T E'
E' -> + T E' | - T E' | ε
T -> F T'
T' -> * F T' | / F T' | ε
F -> ( E ) id

```

Рисунок 1.6 – Грамматика после удаления левой рекурсии (пример 3)

1.3.2 Устранение левой факторизации

```

G = ({S, E}, {i, t, e, a, b}, P, S), где P состоит из правил:
S -> i E t S | i E t S e S | a
E -> b

```

Рисунок 1.7 – Исходная грамматика для удаления левой факторизации

```

G = ({S, S', E}, {i, t, e, a, b}, P, S), где P состоит из правил:
S -> i E t S S' | a
S' -> ε | e S
E -> b

```

Рисунок 1.8 – Грамматика после удаления левой факторизации

1.3.3 Преобразование КС-грамматики к нормальной форме Хомского

$G = (\{S, A, B\}, \{a, b\}, P, S)$, где P состоит из правил:

$S \rightarrow a A B \mid B A$
 $A \rightarrow B B B \mid a$
 $B \rightarrow A S \mid b$

Рисунок 1.9 – Исходная грамматика перед преобразованием к нормальной форме Хомского (пример 1)

$G = (\{S, A, B, a', \langle AB \rangle, \langle BB \rangle\}, \{a, b\}, P, S)$, где P состоит из правил:

$S \rightarrow a' \langle AB \rangle \mid B A$
 $A \rightarrow B \langle BB \rangle \mid a$
 $B \rightarrow A S \mid b$
 $a' \rightarrow a$
 $\langle AB \rangle \rightarrow A B$
 $\langle BB \rangle \rightarrow B B$

Рисунок 1.10 – Грамматика после преобразования к нормальной форме Хомского (пример 1)

$G = (\{S\}, \{0, 1\}, P, S)$, где P состоит из правил:

$S \rightarrow 0 S 1 \mid 0 1$

Рисунок 1.11 – Исходная грамматика перед преобразованием к нормальной форме Хомского (пример 2)

$G = (\{S, 0', \langle S1 \rangle, 1'\}, \{0, 1\}, P, S)$, где P состоит из правил:

$S \rightarrow 0' \langle S1 \rangle \mid 0' 1'$
 $0' \rightarrow 0$
 $\langle S1 \rangle \rightarrow S 1'$
 $1' \rightarrow 1$

Рисунок 1.12 – Грамматика после преобразования к нормальной форме Хомского (пример 2)

$G = (\{S, A, B\}, \{a, b\}, P, S)$, где P состоит из правил:

$S \rightarrow a B \mid b A$
 $A \rightarrow a S \mid b A A \mid a$
 $B \rightarrow b S \mid a B B \mid b$

Рисунок 1.13 – Исходная грамматика перед преобразованием к нормальной форме Хомского (пример 3)

$G = (\{S, A, B, a', b', \langle AA \rangle, \langle BB \rangle\}, \{a, b\}, P, S)$, где P состоит из правил:

$S \rightarrow a' B \mid b' A$
 $A \rightarrow a' S \mid b' \langle AA \rangle \mid a$
 $B \rightarrow b' S \mid a' \langle BB \rangle \mid b$
 $a' \rightarrow a$
 $b' \rightarrow b$
 $\langle AA \rangle \rightarrow A A$
 $\langle BB \rangle \rightarrow B B$

Рисунок 1.14 – Грамматика после преобразования к нормальной форме Хомского (пример 3)

2 Контрольные вопросы

1. Как может быть определён формальный язык?
 - (a) Простым перечислением слов, входящих в данный язык.
 - (b) Словами, порождёнными некоторой формальной грамматикой.
 - (c) Словами, порождёнными регулярным выражением.
 - (d) Словами, распознаваемыми некоторым конечным автоматом.
2. Какими характеристиками определяется грамматика?
 - (a) Σ — множество терминальных символов.
 - (b) N — множество нетерминальных символов.
 - (c) P — множество правил (слева — непустая последовательность терминалов/нетерминалов, содержащая хотя бы один нетерминал, справа — любая последовательность терминалов/нетерминалов).
 - (d) S — начальный символ из множества нетерминалов.
3. Дайте описания грамматик по иерархии Хомского.
 - (a) Регулярные.
 - (b) Контекстно-свободные.
 - (c) Контекстно-зависимые.
 - (d) Неограниченные.
4. Какие абстрактные устройства используются для разбора грамматик?
 - (a) Распознающие грамматики — устройства (алгоритмы), которым на вход подается цепочка языка, а на выходе устройство печатает «Да», если цепочка принадлежит языку, и «Нет» — иначе.
5. Оцените временную и емкостную сложность предложенного вам алгоритма.
 - (a) Алгоритм удаления левой рекурсии
 - $O(N^2)$ — временная сложность;
 - $O(N)$ — ёмкостная сложность.

3 Текст программы

В листингах 3.1 и 3.2 представлен код программы.

Листинг 3.1 – Основной модуль программы

```
import os
from grammar import Grammar, readGrammarFromFile

MENU = f"""
\tМеню\n
1.  Исходная грамматика;
2.  Грамматика после устранения левой рекурсии;
3.  Грамматика после устранения левой факторизации;
4.  Грамматика после устранения левой рекурсии и левой
    факторизации;
5.  Преобразование КС-грамматики к нормальной форме Хомского.

0.  Выход.\n
Выбор: """

SIZE_MENU = 5
OUTPUT_FILE_NAME = "../data/result.txt"

def inputOption(minOptions: int, maxOptions: int, msg: str):
    try:
        option = int(input(msg))
    except:
        option = -1
    else:
        if option < minOptions or option > maxOptions:
            option = -1

    if option == -1:
        print(f"\nОжидался ввод целого числа от {minOptions} до
            {maxOptions}")

    return option
```



```

def chooseInputFile() -> str:
    with open("temp.txt", "w") as f:
        files = os.listdir('../data')
        for item in files:
            f.write("%s\n" % item)

    with open("temp.txt") as f:
        fileNames = [line[:-1] for line in f.readlines()]

    os.remove('temp.txt')

    msg = f"\n\tВходные файлы:\n\n"
    for i in range(len(fileNames)):
        msg += f"    {i + 1}.  {fileNames[i]};\n"
    msg += f"\n    Выбор: "

    option = -1
    while option == -1:
        option = inputOption(
            minOptions=1,
            maxOptions=len(fileNames),
            msg=msg,
        )

    return f"../data/{fileNames[option - 1]}"

def main():
    inputFile = chooseInputFile()
    option = -1
    while option != 0:
        option = inputOption(
            minOptions=0,
            maxOptions=SIZE_MENU,
            msg=MENU,
        )
    match option:
        case 1:
            grammar: Grammar = reedGrammarFromFile(inputFile)
            grammar.printGrammar()
        case 2:

```

```

        grammar: Grammar = reedGrammarFromFile(inputFile)
        grammar.removeLeftRecursion()
        grammar.printGrammar()
        grammar.createFileFromGrammar(OUTPUT_FILE_NAME)
    case 3:
        grammar: Grammar = reedGrammarFromFile(inputFile)
        grammar.removeLeftFactorization()
        grammar.printGrammar()
        grammar.createFileFromGrammar(OUTPUT_FILE_NAME)
    case 4:
        grammar: Grammar = reedGrammarFromFile(inputFile)
        grammar.removeLeftRecursion()
        grammar.removeLeftFactorization()
        grammar.printGrammar()
        grammar.createFileFromGrammar(OUTPUT_FILE_NAME)
    case 5:
        grammar: Grammar = reedGrammarFromFile(inputFile)
        grammar.convertToChomskyForm()
        grammar.printGrammar()
        grammar.createFileFromGrammar(OUTPUT_FILE_NAME)

if __name__ == '__main__':
    main()

```

Листинг 3.2 – Модуль для преобразования грамматик

```

from functools import reduce
from copy import deepcopy

class Grammar:
    notTerminals: list[str]
    terminals: list[str]
    rules: dict[str, list[list[str]]]
    start: str

    def __init__(
        self,
        notTerminals: list[str],
        terminals: list[str],
        rules: dict[str, list[list[str]]],
        start: str
    ):

```

```

) -> None:
    self.notTerminals = notTerminals
    self.terminals = terminals
    self.rules = rules
    self.start = start

def printGrammar(self) -> None:
    notTerminals =
        Grammar.__joinListWithSymbol(self.notTerminals, ", ")
    terminals = Grammar.__joinListWithSymbol(self.terminals,
        ", ")

    print(f"\nG = ({{{notTerminals}}}, {{{terminals}}}, P,
        {self.start}), где P состоит из правил:\n")
    for notTerminal in self.notTerminals:
        rightRules = self.rules[notTerminal]
        self.__printProduct(notTerminal, rightRules)

def removeLeftRecursion(self) -> None:
    i = 0
    while i < len(self.notTerminals):
        copyRightRules =
            self.rules[self.notTerminals[i]].copy()
        for j in range(i):
            self.__replaceProducts(
                notTerminal=self.notTerminals[i],
                replaceableNotTerminal=self.notTerminals[j],
            )
        if
            self.__removeDirectLeftRecursion(self.notTerminals[i]):
            i += 2
        else:
            self.rules[self.notTerminals[i]] = copyRightRules
            i += 1

def removeLeftFactorization(self) -> None:
    i = 0
    while i < len(self.notTerminals):
        maxPrefix = ""
        rightRules = self.rules[self.notTerminals[i]]
        for j in range(len(rightRules)):

```

```

prefix = ""
for symbol in rightRules[j]:
    indexList = self.__findPrefixMatches(
        rightRules=rightRules,
        prefix=prefix + symbol,
    )
    if len(indexList) > 1:
        prefix += symbol
    else:
        break

if len(prefix) > len(maxPrefix):
    maxPrefix = prefix

if maxPrefix:
    print(f"\nСамый длинный префикс для
        {self.notTerminals[i]}: {maxPrefix}")
    self.__removeDirectLeftFactorization(self.notTerminals[i],
        maxPrefix)
else:
    i += 1

def convertToChomskyForm(self) -> None:
    for notTerminal in self.notTerminals.copy():
        for i in range(len(self.rules[notTerminal])):
            rightRule = self.rules[notTerminal][i]
            if len(rightRule) == 2 and \
                rightRule[0] in self.notTerminals and \
                rightRule[1] in self.notTerminals or \
                len(rightRule) == 1 and rightRule[0] in
                self.terminals or \
                notTerminal == self.start and rightRule[0]
                == "":
                continue

            elif len(rightRule) == 2 and \
                (rightRule[0] in self.terminals or
                 rightRule[1] in self.terminals):
                if rightRule[0] in self.terminals:
                    firstElem = f"{rightRule[0]}"
                    if not firstElem in self.notTerminals:

```

```

        self.notTerminals.append(firstElem)
        self.rules[firstElem] =
            [[rightRule[0]]]
    else:
        firstElem = rightRule[0]

    if rightRule[1] in self.terminals:
        secondElem = f"{rightRule[1]}"
        if not secondElem in self.notTerminals:
            self.notTerminals.append(secondElem)
            self.rules[secondElem] =
                [[rightRule[1]]]
    else:
        secondElem = rightRule[1]

    self.rules[notTerminal][i] = [firstElem,
        secondElem]

elif len(rightRule) > 2:
    if rightRule[0] in self.notTerminals:
        self.rules[notTerminal][i] =
            [rightRule[0],
            f"<{"".join(rightRule[1:])}>"]
    else:
        self.rules[notTerminal][i] =
            [f"{rightRule[0]}"',
            f"<{"".join(rightRule[1:])}>"]
        if not f"{rightRule[0]}"' in
            self.notTerminals:
            self.notTerminals.append(f"{rightRule[0]}")
            self.rules[f"{rightRule[0]}"'] =
                [[rightRule[0]]]

    rightRule = rightRule[1:]
    newNotTerminal = f"<{"".join(rightRule)}>"
    while len(rightRule) > 2:
        if not newNotTerminal in
            self.notTerminals:
            self.notTerminals.append(newNotTerminal)

        if rightRule[0] in self.notTerminals:

```

```

        self.rules[newNotTerminal] =
            [[rightRule[0],
              f"<{"".join(rightRule[1:])}>"]]
    else:
        self.rules[newNotTerminal] =
            [[f"{rightRule[0]}'",
              f"<{"".join(rightRule[1:])}>"]]
        if not f"{rightRule[0]}'" in
            self.notTerminals:
            self.notTerminals.append(f"{right
            self.rules[f"{rightRule[0]}'"
                = [[rightRule[0]]

    rightRule = rightRule[1:]
    newNotTerminal =
        f"<{"".join(rightRule)}>"

    newNotTerminal = f"<{"".join(rightRule)}>"
    if not newNotTerminal in self.notTerminals:
        if rightRule[0] in self.terminals:
            firstElem = f"{rightRule[0]}'"
            if not firstElem in
                self.notTerminals:
                self.notTerminals.append(firstElem)
                self.rules[firstElem] =
                    [[rightRule[0]]]
        else:
            firstElem = rightRule[0]

    if rightRule[1] in self.terminals:
        secondElem = f"{rightRule[1]}'"
        if not secondElem in
            self.notTerminals:
            self.notTerminals.append(secondElem)
            self.rules[secondElem] =
                [[rightRule[1]]]
    else:
        secondElem = rightRule[1]

    self.notTerminals.append(newNotTerminal)
    self.rules[newNotTerminal] =

```

```

[[firstElem, secondElem]]

def createFileFromGrammar(self, fileName: str) -> None:
    with open(fileName, "w", encoding="utf-8") as f:
        for i in range(len(self.notTerminals)):
            if i:
                f.write(" ")
            f.write(f"{self.notTerminals[i]}")
        f.write("\n")

        for i in range(len(self.terminals)):
            if i:
                f.write(" ")
            f.write(f"{self.terminals[i]}")
        f.write("\n")

        for notTerminal in self.notTerminals:
            for rightRule in self.rules[notTerminal]:
                f.write(f"{notTerminal} ->")
                for symbol in rightRule:
                    f.write(f" {symbol}")
                f.write("\n")

        f.write(f"{self.start}\n")

def __removeDirectLeftFactorization(self, notTerminal: str,
    maxPrefix: str) -> None:
    indexList = self.__findPrefixMatches(
        rightRules=self.rules[notTerminal],
        prefix=maxPrefix,
    )
    newRightRules = []
    lenMaxPrefix = len(maxPrefix)
    for i in indexList:
        if len(self.rules[notTerminal][i]) > lenMaxPrefix:
            newRightRules.append(self.rules[notTerminal][i][lenMa
        else:
            newRightRules.append([""])

    rightRules = []
    for i in range(len(self.rules[notTerminal])):

```

```

        if not i in indexList:
            rightRules.append(self.rules[notTerminal][i])

newNotTerminal = self.__findNewNotTerminal(notTerminal)
self.rules[newNotTerminal] = newRightRules
self.rules[notTerminal] = \
    [list(maxPrefix) + [newNotTerminal]] + rightRules

indexNotTerminal = self.notTerminals.index(notTerminal)
self.notTerminals = \
    self.notTerminals[:indexNotTerminal + 1] +
    [newNotTerminal] + \
    self.notTerminals[indexNotTerminal + 1:]

def __findNewNotTerminal(self, notTerminal: str):
    try:
        baseNotTerminal =
            notTerminal[:notTerminal.index("'")]
    except ValueError:
        baseNotTerminal = notTerminal

    quotationMarkCount = 0
    for item in self.notTerminals:
        if item.find(baseNotTerminal) != -1:
            quotationMarkCount += 1

    return notTerminal + "'" * quotationMarkCount

def __findPrefixMatches(self, rightRules: list[list[str]],
    prefix: str) -> list[int]:
    indexList = []
    for i in range(len(rightRules)):
        if self.__comparePrefixes(rightRules[i], prefix):
            indexList.append(i)

    return indexList

def __comparePrefixes(self, rightRule: list[str], prefix:
    str) -> bool:
    if len(rightRule) < len(prefix):
        return False

```



```

    for i in range(len(prefix)):
        if prefix[i] != rightRule[i]:
            return False

    return True

def __replaceProducts(self, notTerminal: str,
    replaceableNotTerminal: str) -> None:
    flagReplace = False
    newRightRules = []
    rightRules = self.rules[notTerminal]
    for i in range(len(rightRules)):
        if replaceableNotTerminal not in rightRules[i]:
            newRightRules.append(rightRules[i])
            continue

        flagReplace = True
        j = rightRules[i].index(replaceableNotTerminal)
        for substitutedRightRule in
            self.rules[replaceableNotTerminal]:
                newRightRule = rightRules[i][:j]
                if substitutedRightRule[0] != "":
                    newRightRule.extend(substitutedRightRule)
                newRightRule.extend(rightRules[i][j + 1:])
                newRightRules.append(newRightRule)

    if flagReplace:
        self.rules[notTerminal] = newRightRules
        print(f"\nПосле замены {replaceableNotTerminal}: ",
            end="")
        self.__printProduct(notTerminal, newRightRules)

def __removeDirectLeftRecursion(self, notTerminal: str) ->
    bool:
    self.rules[notTerminal].sort(
        key=lambda rightRule: rightRule[0] != notTerminal
    )
    newNotTerminal = notTerminal + "'"
    rightRulesForNewNotTerminal = []
    rightRules = []

```

```

    for rightRule in deepcopy(self.rules[notTerminal]):
        if rightRule[0] != notTerminal:
            if rightRule[0] == "":
                rightRule = [newNotTerminal]
            else:
                rightRule.append(newNotTerminal)
            rightRules.append(rightRule)
        else:
            rightRule = rightRule[1:]
            rightRule.append(newNotTerminal)
            rightRulesForNewNotTerminal.append(rightRule)

    if len(rightRulesForNewNotTerminal):
        rightRulesForNewNotTerminal.append([""])
        indexNotTerminal =
            self.notTerminals.index(notTerminal)
        self.notTerminals = \
            self.notTerminals[:indexNotTerminal + 1] +
            [newNotTerminal] + \
            self.notTerminals[indexNotTerminal + 1:]
        self.rules[newNotTerminal] =
            rightRulesForNewNotTerminal
        self.rules[notTerminal] = rightRules

        removedFlag = True
    else:
        removedFlag = False

    return removedFlag

def __printProduct(self, notTerminal: str, rightRules:
    list[list[str]]):
    print(f"{notTerminal} -> ", end="")
    for i in range(len(rightRules)):
        print(f"{" | " if i != 0 else
            ""}{Grammar.__joinListWithSymbol(rightRules[i], "
            ")}", end="")
    print()

@staticmethod

```

```

def __joinListWithSymbol(arr: list[str], symbol: str) -> str:
    return reduce(lambda elemPrev, elem:
        f"{elemPrev}{symbol}{elem}", arr)

def readGrammarFromFile(fileName: str) -> Grammar:
    with open(fileName, encoding="utf-8") as f:
        lines = [line[:-1] for line in f.readlines()]

    notTerminals = lines[0].split(" ")
    terminals = lines[1].split(" ")
    start = lines[-1]
    rules = {}
    for notTerminal in notTerminals:
        rules[notTerminal] = []

    for rule in lines[2:-1]:
        rule = rule.split(" ")
        rules[rule[0]].append(rule[2:])

    return Grammar(
        notTerminals=notTerminals,
        terminals=terminals,
        rules=rules,
        start=start,
    )

```