Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина «Методы трансляции»

|  |  |
| --- | --- |
|  |  |

**ОТЧЕТ**

к лабораторной работе № 2

на тему

**ЛЕКСИЧЕСКИЙ АНАЛИЗ**

Выполнил             К. А. Тимофеев

Проверил                          Н. Ю. Гриценко

Минск 2024

СОДЕРЖАНИЕ

[1 Постановка задачи 3](#_Toc158040802)

[2 Краткие теоретические сведения 4](#_Toc158040803)

[3 Результаты выполнения лабораторной работы 6](#_Toc158040804)

[Выводы 8](#_Toc158040805)

[Список использованных источников 9](#_Toc158040806)

[Приложение А (обязательное) Листинг исходного кода 10](#_Toc158040807)

## 1 ПОСТАНОВКА ЗАДАЧИ

Целью выполнения данной лабораторной работы является разработка лексического анализатора подмножества языка программирования в С. Также необходимо определить лексические правила и выполнить перевод потока символов в поток токенов, при определении неверной последовательности символов необходимо обнаружить ошибку и выдать сообщение о ней.

## 2 КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Первая фаза компиляции называется лексическим анализом   
или сканированием. Лексический анализатор читает поток символов, составляющих исходную программу, и группирует эти символы в значащие последовательности, называющиеся лексемами.[1]

Лексема – это структурная единица языка, которая состоит   
из элементарных символов и не содержит в своем составе других структурных единиц языка. Лексемами языка программирования являются идентификаторы, константы, ключевые слова языка, знаки операции. На вход лексического анализатора поступает текст исходной программы, а выходная информация передается для дальнейшей обработки синтаксическому анализатору. Для каждой лексемы анализатор строит выходной токен, где имя токена связано с его значением в коде.

Использование лексического анализатора упрощает работу с текстом исходной программы на этапе синтаксического разбора и сокращается объем обрабатываемой информации. Для выделения в тексте и разбора лексем можно применять простую и эффективную технику анализа, в то время как на этапе синтаксического анализа конструкций исходного языка используются достаточно сложные алгоритмы. Лексический анализатор отделяет сложный по конструкции синтаксический анализатор от работы непосредственно   
с текстом исходной программы. В большинстве компиляторов лексический   
и синтаксический анализаторы – это взаимосвязанные части.[2]

При написании данной лабораторной работы были применены следующие теоретические сведения и концепции:

1 Списки языка программирования *Python*: при помощи использования словарей были представлены наборы ключевых слов, операторов и типов данных в языке программирования *С*. Эти списки используются   
для классификации токенов в программе.

2 Регулярные выражения: при помощи регулярных выражений были произведены проверки соответствия токенов правилам языка.

3 Циклы и ветвления языка программирования *Python*: при помощи разнообразных циклов и ветвлений были организованы разбитие исходного кода на токены, а также классификация каждого токена на основе его типа.

4 Запись в файл и чтение из файла: для работы с файлом кода были использованы встроенные функции языка *Python*.

5 Классы языка программирования *Python*: для представления элементов языка программирования *C* были созданы классы для всех соответствующих категорий ключевых слов языка программирования *C*.

Все вышеперечисленные концепции были использованы для написания лексического анализатора подмножества языка программирования *С*,   
а также для определения лексических правил и перевода потока символов   
в поток токенов.

# 3 РЕЗУЛЬТАТЫ ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ             РАБОТЫ

В ходе лабораторной работы был разработан лексический анализатор для языка программирования С++. Для работы с кодом используются файлы. При запуске программы код анализируется и разбивается на токены. Результат работы лексического анализатора представлен на рисунке 3.1.

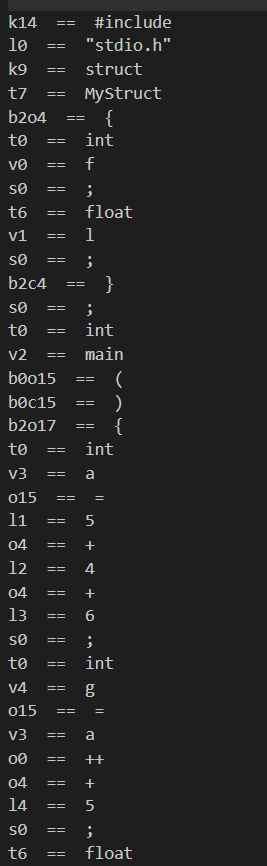


Рисунок 3.1 – Результат работы лексического анализатора

Первая буква в токене обозначает к какому типу он относится. Было установлено следующее соответствие:

– *k –* для ключевых слов;

– *v –* для переменных;

– *l –* для литералов;

– *o –* для операторов;

– *b –* для различных скобок;

– *s –* для пунктуационных знаков;

– *t –* для типов.

Кроме вывода лексем и их значений программа обрабатывает некоторые ошибки в коде. Если совершить попытку дать имя переменной, начиная его с цифры, то это будет обозначено в таблице лексем как ошибка. Результат нахождения данной ошибки представлен на рисунке 3.2.

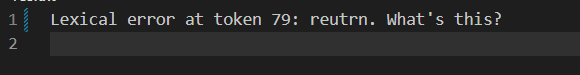


Рисунок 3.2 – Результат нахождения ошибки

Таким образом, по итогу лабораторной работы был разработан лексический анализатор кода, написанного на языке программирования *С*, а также реализовано нахождение разного рода лексических ошибок.

## ВЫВОДЫ

В ходе выполнения данной лабораторной работы был разработан лексический анализатор подмножества языка программирования *С*. Также были определены лексические правила и выполнен перевод потока символов в поток токенов. При определении неверной последовательности символов была реализована возможность обнаружение ошибок и демонстрация сообщений о данных ошибках.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Лексический анализатор [Электронный ресурс]. – Режим доступа: https://csc.sibsutis.ru/sites/csc.sibsutis.ru/files/courses/trans/.

[2] Лексический и синтаксический анализ [Электронный ресурс]. – Режим доступа: http://edu.tsu.ru/eor/resourse/577/html/90.html/.

## ПРИЛОЖЕНИЕ А

## (обязательное)

## Листинг исходного кода

Листинг 1 – Программная код лексического анализатора

import re

class VariableTableCell:

def \_\_init\_\_(self, id: int, name: str, mytype, scopeStart: int):

self.id = id

self.name = name

self.mytype = mytype

self.scopeStart = scopeStart

keywordIdCounter = 0

class MyKeyword:

def \_\_init\_\_(self, word: str):

global keywordIdCounter

self.word = word

self.id = keywordIdCounter

keywordIdCounter += 1

bracketsIdCounter = 0

class MyBrackets:

def \_\_init\_\_(self, symbols: str):

global bracketsIdCounter

self.symbols = symbols

self.openSymbol = symbols[0]

self.closeSymbol = symbols[1]

self.id = bracketsIdCounter

bracketsIdCounter += 1

separatorsCountId = 0

class MySeparator:

def \_\_init\_\_(self, symbol: str):

global separatorsCountId

self.symbol = symbol

self.id = separatorsCountId

separatorsCountId += 1

operatorsCountId = 0

class MyOperator:

def \_\_init\_\_(self, symbol: str):

global operatorsCountId

self.symbol = symbol

self.id = operatorsCountId

operatorsCountId += 1

typeid: int = 0

class MyType:

def \_\_init\_\_(self, name, \*\*kwargs):

global typeid

self.name = name

self.id = typeid

typeid += 1

class Ptr(MyType):

def \_\_init\_\_(self, mytype: type, level: int):

MyType.\_\_init\_\_(self, mytype.Name + "\*" \* level)

self.pointersTo = mytype

self.level = level

literalCountId = 0

class MyLiteral:

def \_\_init\_\_(self, text: str, tokenIndex: int) -> None:

global literalCountId

self.text = text

self.tokenIndex = tokenIndex

self.id = literalCountId

literalCountId += 1

keywords: list[MyKeyword] = [

MyKeyword("for"),

MyKeyword("while"),

MyKeyword("do"),

MyKeyword("return"),

MyKeyword("const"),

MyKeyword("switch"),

MyKeyword("case"),

MyKeyword("if"),

MyKeyword("union"),

MyKeyword("struct"),

MyKeyword("signed"),

MyKeyword("unsigned"),

MyKeyword("default"),

MyKeyword("else"),

MyKeyword("#include"),

]

kwStr = ""

for i in range(len(keywords)):

kwStr += re.escape(keywords[i].word)

if i != len(keywords) - 1:

kwStr += "|"

keywordsPattern = re.compile("(" + kwStr + ")")

literalsPatterns = [

("float", re.compile("\d+\.\d+")),

("int", re.compile("\d+")),

("char", re.compile("('.'|'\\[\w0])")),

("string", re.compile('".\*"')),

]

types: list[MyType] = [

MyType("int"),

MyType("char"),

MyType("short"),

MyType("long"),

MyType("void"),

MyType("double"),

MyType("float"),

]

operators: list[MyOperator] = [

MyOperator("++"),

MyOperator("--"),

MyOperator("\*"),

MyOperator("/"),

MyOperator("+"),

MyOperator("=="),

MyOperator("-"),

MyOperator("<"),

MyOperator(">"),

MyOperator("<="),

MyOperator(">="),

MyOperator("&&"),

MyOperator("||"),

MyOperator("&="),

MyOperator("|="),

MyOperator("="),

]

operatorsStr: str = "("

ttmp: str = ""

for i in range(len(operators)):

ttmp += re.escape(operators[i].symbol)

if i != len(operators) - 1:

ttmp += "|"

operatorsStr += ttmp + ")"

operatorsPattern = re.compile(operatorsStr)

separators: list[MySeparator] = [

MySeparator(";"),

MySeparator(","),

MySeparator(":"),

MySeparator("?"),

]

separatorsStr: str = "("

ttmp: str = ""

for i in range(len(separators)):

ttmp += re.escape(separators[i].symbol)

if i != len(separators) - 1:

ttmp += "|"

separatorsStr += ttmp + ")"

separatorsPattern = re.compile(separatorsStr)

brackets: list[MyBrackets] = [MyBrackets("()"), MyBrackets("[]"), MyBrackets("{}")]

bracketsStr = ""

for i in range(len(brackets)):

bracketsStr += (

re.escape(brackets[i].openSymbol) + "|" + re.escape(brackets[i].closeSymbol)

)

if i != len(brackets) - 1:

bracketsStr += "|"

bracketsPattern = re.compile("(" + bracketsStr + ")")

identifierPattern = re.compile("[A-Za-z\_]\w\*")

def separateByMatches(matches, string):

separated = [string]

for j in range(len(matches)):

last = separated.pop()

split\_results: re.Match = re.search(re.escape(matches[j]), last)

beg = split\_results.start()

end = split\_results.end()

if last[0:beg] != "":

separated.append(last[0:beg])

if last[beg:end] != "":

separated.append(last[beg:end])

if last[end:] != "":

separated.append(last[end:])

return separated

def separate(text) -> list[str]:

pattern = re.compile(r"(\".\*\"|\S+)")

res = re.findall(pattern, text)

i: int = 0

while i < len(res):

piece = res[i]

tmp = re.findall(operatorsPattern, piece)

if tmp:

separated = separateByMatches(tmp, res[i])

res = res[0:i] + separated + res[i + 1 :]

tmp = re.findall(separatorsPattern, res[i])

if tmp:

separated = separateByMatches(tmp, res[i])

res = res[0:i] + separated + res[i + 1 :]

tmp = re.findall(bracketsPattern, res[i])

if tmp:

separated = separateByMatches(tmp, res[i])

res = res[0:i] + separated + res[i + 1 :]

i += 1

return res

def findInTableById(table: list, id: int):

for i in range(len(table) - 1, -1, -1):

if table[i].id == id:

return i

return None

def findInTableByName(table: list, name: str) -> int:

for i in range(len(table) - 1, -1, -1):

print(i)

if table[i].name == name:

return i

return None

def findInTypeTableByName(table: list[MyType], name: str):

for i in range(len(table) - 1, -1, -1):

print(i)

if table[i].name == name:

return i

return None

vars\_id\_counter: int = 0

def createTables(tokens: list[str]):

global vars\_id\_counter

global types

vars\_table: list[VariableTableCell] = []

scope\_starts = [0]

type\_table: list[MyType] = []

literals\_table: list = []

for mytype in types:

type\_table.append(mytype)

for i in range(len(tokens)):

if tokens[i] == "{" or tokens[i] == "for":

scope\_starts.append(i)

if tokens[i] == "}":

scope\_starts.pop()

print(tokens[i])

isLiteral: bool = False

# if not keyword operator or literal or sep then identificator

for typename, pattern in literalsPatterns:

isLiteral = isLiteral or re.match(pattern, tokens[i])

isType: bool = False

for mytype in type\_table:

isType = isType or re.match(re.escape(mytype.name), tokens[i])

isBracket: bool = False

for bracket in brackets:

isBracket = isBracket or (

bracket.openSymbol == tokens[i] or bracket.closeSymbol == tokens[i]

)

if not (

re.match(keywordsPattern, tokens[i])

or re.match(operatorsPattern, tokens[i])

or re.match(separatorsPattern, tokens[i])

or isLiteral

or isType

or isBracket

):

if not re.match(identifierPattern, tokens[i]):

file = open("res.txt", "w")

print(f"Lexical error at token {i}: {tokens[i]}", file=file)

file.close()

exit(0)

continue

if tokens[i - 1] == "struct":

print("it's new type!!!")

index = findInTypeTableByName(type\_table, tokens[i])

if index == None:

type\_table.append(MyType(tokens[i]))

print("it's variable!!!")

index = findInTableByName(type\_table, tokens[i])

if index != None:

continue

if findInTypeTableByName(type\_table, tokens[i - 1]) is None:

if tokens[i] not in [var.name for var in vars\_table]:

file = open("res.txt", "w")

print(

f"Lexical error at token {i}: {tokens[i]}. What's this?",

file=file,

)

file.close()

exit(0)

continue

vars\_table.append(

VariableTableCell(

vars\_id\_counter,

tokens[i],

type\_table[findInTypeTableByName(type\_table, tokens[i - 1])],

scope\_starts[-1],

)

)

vars\_id\_counter += 1

else:

if isLiteral:

literals\_table.append(MyLiteral(tokens[i], i))

for i in range(len(vars\_table)):

print(

f"var{vars\_table[i].id} {vars\_table[i].name} {vars\_table[i].mytype.name} {vars\_table[i].scopeStart}"

)

for i in range(len(type\_table)):

print(f"t{type\_table[i].id} {type\_table[i].name}")

return (type\_table, vars\_table, literals\_table)

def find\_var\_by\_name\_and\_scope(

vars\_table: list[VariableTableCell], name: str, scopeStart: int

):

for cell in vars\_table:

if cell.name == name and cell.scopeStart == scopeStart:

return cell

return None

def change\_to\_ids(

type\_table: list[MyType],

vars\_table,

literals\_table: list[MyLiteral],

original\_tokens: list[str],

):

new\_tokens = original\_tokens.copy()

scope\_starts: dict[str, list[int]] = {}

for br in brackets:

scope\_starts[br.symbols] = []

scope\_starts["{}"].append(0)

for i in range(len(new\_tokens)):

var\_cell = find\_var\_by\_name\_and\_scope(

vars\_table, new\_tokens[i], scope\_starts["{}"][-1]

)

if var\_cell is not None:

new\_tokens[i] = f"v{var\_cell.id}"

continue

type\_index = findInTypeTableByName(type\_table, new\_tokens[i])

if type\_index is not None:

new\_tokens[i] = f"t{type\_table[type\_index].id}"

continue

if i in [it.tokenIndex for it in literals\_table]:

for literal in literals\_table:

if literal.tokenIndex == i:

new\_tokens[i] = f"l{literal.id}"

elif new\_tokens[i] in [op.symbol for op in operators]:

for op in operators:

if new\_tokens[i] == op.symbol:

new\_tokens[i] = f"o{op.id}"

elif new\_tokens[i] in [kw.word for kw in keywords]:

for kw in keywords:

if kw.word == new\_tokens[i]:

new\_tokens[i] = f"k{kw.id}"

elif new\_tokens[i] in [sep.symbol for sep in separators]:

for sep in separators:

if sep.symbol == new\_tokens[i]:

new\_tokens[i] = f"s{sep.id}"

elif new\_tokens[i] in [br.openSymbol for br in brackets]:

for br in brackets:

if br.openSymbol == new\_tokens[i]:

scope\_starts[br.symbols].append(i)

new\_tokens[i] = f"b{br.id}o{scope\_starts[br.symbols][-1]}"

elif new\_tokens[i] in [br.closeSymbol for br in brackets]:

for br in brackets:

if br.closeSymbol == new\_tokens[i]:

poped = scope\_starts[br.symbols].pop()

new\_tokens[i] = f"b{br.id}c{poped}"

return new\_tokens

def analyze(text: str):

tokens = separate(text)

(type\_table, vars\_table, literals\_table) = createTables(tokens)

new\_tokens = change\_to\_ids(type\_table, vars\_table, literals\_table, tokens)

file = open("res.txt", "w")

file.truncate()

print("tokens:\n", file=file)

for i in range(len(new\_tokens)):

print(f"{new\_tokens[i]} == {tokens[i]}", file=file)

print("\n-------------\n", file=file)

print("table of types(t):\n", file=file)

for type in type\_table:

print(f"{type.id} {type.name}", file=file)

print("\n-------------\n", file=file)

print("table of vars(v):\n", file=file)

for var in vars\_table:

print(f"{var.id} {var.name} {var.mytype.name} {var.scopeStart}", file=file)

print("\n-------------\n", file=file)

print("table of keywords(k):\n", file=file)

for kw in keywords:

print(f"{kw.id} {kw.word}", file=file)

print("\n-------------\n", file=file)

print("table of separators(s):\n", file=file)

for sep in separators:

print(f"{sep.id} {sep.symbol}", file=file)

print("\n-------------\n", file=file)

print("table of brackets(b{id}[oc]{firstToken}):\n", file=file)

for br in brackets:

print(f"{br.id} {br.symbols}", file=file)

print("\n-------------\n", file=file)

print("table of operators(o):\n", file=file)

for op in operators:

print(f"{op.id} {op.symbol}", file=file)

file.close()

def clearComments(text: str) -> str:

new\_text = str()

i: int = 0

while i < len(text):

if text[i] == "/":

if text[i + 1] == "/":

while i < len(text) and text[i] != "\n":

i += 1

elif text[i + 1] == "\*":

while i < len(text) and text[i - 2] + text[i - 1] != "\*/":

i += 1

if i >= len(text):

continue

new\_text += text[i]

i += 1

return new\_text

if \_\_name\_\_ == "\_\_main\_\_":

# print("hui")

file = open("test.c")

text: str = file.read()

file.close()

text = clearComments(text)

print(text)

analyze(text)