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             Е. А. Киселёва

Проверил                          Н. Ю. Гриценко

Минск 2024

СОДЕРЖАНИЕ

[1 Постановка задачи 3](#_Toc158040802)

[2 Краткие теоретические сведения 4](#_Toc158040803)

[3 Результаты выполнения лабораторной работы 5](#_Toc158040804)

[Выводы 7](#_Toc158040805)

[Список использованных источников 8](#_Toc158040806)

[Приложение А (обязательное) Листинг исходного кода 9](#_Toc158040807)

## 1 ПОСТАНОВКА ЗАДАЧИ

Целью выполнения данной лабораторной работы является разработка лексического анализатора подмножества языка программирования 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 сокращает объем обрабатываемой информации. В большинстве компиляторов лексический   
и синтаксический анализаторы – это взаимосвязанные части.

В ходе лабораторной работы разрабатывался лексический анализатор кода программ на C++. Лексемы этого языка программирования используются для построения операторов, определений, объявлений других компонентов, из которых состоит вся программа. Существуют следующие лексические элементы: маркеры и наборы символов, комментарии, идентификаторы, ключевые слова, символы пунктуации, числовые литералы, строковые литералы, литералы-указатели. Идентификаторы включают в себя имена объекта или переменной, имена класса, структуры или объединения, имена перечисленного типа, члены классов, структур, объединений или перечислений, функции или функции члена класса, имена определения типа (typedef), имена макроса, параметров макроса [2]

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

1 Управление потоком: используются конструкции управления потоком, такие как циклы while и условные операторы if, для обработки символов входного потока и применения к ним различных правил и проверок.

2 Обработка ошибок: код включает в себя проверки ошибок для обнаружения некорректных конструкций и выдачи сообщений об ошибках.

3 Статические методы: использование статических методов в классе Analyzer позволяет вызывать эти методы без создания экземпляра класса.

4 Принципы модульности и повторного использования кода: Код разделен на отдельные функции и классы, что облегчает его понимание и изменение, а также повторное использование отдельных частей.

Все вышеперечисленные концепции были использованы для написания лексического анализатора подмножества языка программирования С++.

# 3 РЕЗУЛЬТАТЫ ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ             РАБОТЫ

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

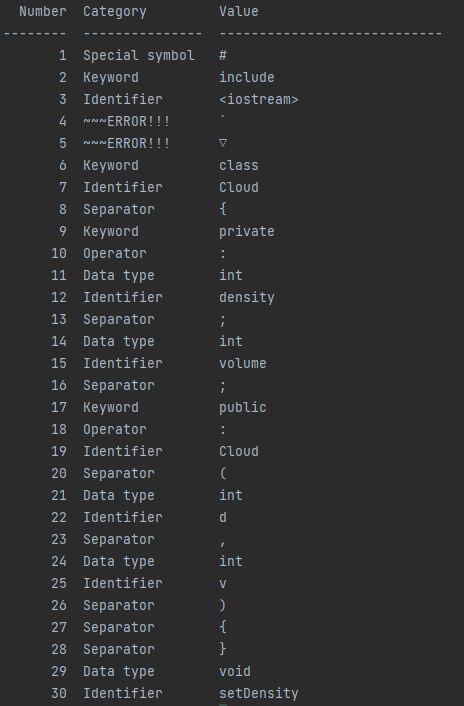


Рисунок 3.1 – Результат работы лексического анализатора

Помимо вывода лексем и их значений программа обрабатывает некоторые ошибки в коде. Если попытаться дать имя переменной, первым символом поставив цифру, то этот токен будет обозначен в таблице лексем как ошибка. Результат нахождения этой ошибки представлен на рисунке 3.2.



Рисунок 3.2 – Результат нахождения ошибки при неправильном наименовании переменной

Если написать символы, которых нет в языке программирования C++, то в таблицу также будет выведена информация об ошибке. Результат нахождения этой ошибки представлен на рисунке 3.3.



Рисунок 3.3 – Результат нахождения ошибки при написании несуществующих в языке символов

Если в коде будет записан числовой литерал с несколькими символами точки, то это тоже обозначится, как ошибка. Результат нахождения этой ошибки представлен на рисунке 3.4.



Рисунок 3.4 – Результат нахождения ошибки в числовом литерале с несколькими символами точки

Если допустить синтаксическую ошибку в коде, например нехватка закрывающихся или открывающихся скобок, нехватка закрывающихся или открывающихся кавычек, то программа выдаст предупреждение о нарушении баланса кода – синтаксической ошибке. Дальнейшая работа программы будет приостановлена. Результат нахождения этой ошибки представлен на рисунке 3.5.



Рисунок 3.5 – Результат синтаксической 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 обнаружения ошибок и демонстрация сообщений о данных ошибках в выходном файле.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Лексический анализатор [Электронный ресурс]. – Режим доступа: https://csc.sibsutis.ru/sites/csc.sibsutis.ru/files/courses/trans/. – Дата доступа: 17.02.2024.
2. Лексические соглашения в С++ [Электронный ресурс]. – Режим доступа <https://learn.microsoft.com/ru-ru/cpp/cpp/lexical-conventions?view=msvc>  
   -170. – Дата доступа: 20.02.2024.

## ПРИЛОЖЕНИЕ А

## (обязательное)

## Листинг исходного кода

Листинг 1 – Программный код главной функции и проверки баланса кода

from analyzer import Analyzer

from tabulate import tabulate

errors = {'missing': [], 'extra': []}

def balance\_check(code):

double\_quotes\_count = 0

single\_quotes\_count = 0

parentheses\_count = 0

curly\_braces\_count = 0

comment\_open\_count = 0

comment\_close\_count = 0

in\_string = False

in\_comment = False

i = 0

while i < len(code):

char = code[i]

if char == '"' and not in\_comment:

double\_quotes\_count += 1

in\_string = not in\_string

elif char == "'" and not in\_comment:

single\_quotes\_count += 1

in\_string = not in\_string

elif char == '(' and not in\_comment and not in\_string:

parentheses\_count += 1

elif char == ')' and not in\_comment and not in\_string:

parentheses\_count -= 1

elif char == '{' and not in\_comment and not in\_string:

curly\_braces\_count += 1

elif char == '}' and not in\_comment and not in\_string:

curly\_braces\_count -= 1

elif char == '/' and i < len(code) - 1 and code[i + 1] == '\*' and not in\_string:

comment\_open\_count += 1

in\_comment = True

i += 1

elif char == '\*' and i < len(code) - 1 and code[i + 1] == '/' and in\_comment and not in\_string:

comment\_close\_count += 1

in\_comment = False

i += 1

i += 1

if (double\_quotes\_count % 2 == 0 and

single\_quotes\_count % 2 == 0 and

parentheses\_count == 0 and

curly\_braces\_count == 0 and

comment\_open\_count == comment\_close\_count):

return True, errors

else:

if double\_quotes\_count % 2 != 0:

errors['missing'].append('"')

if single\_quotes\_count % 2 != 0:

errors['missing'].append("'")

if parentheses\_count != 0:

errors['missing'].append(')')

if curly\_braces\_count != 0:

errors['missing'].append('}')

if comment\_open\_count != comment\_close\_count:

errors['missing'].append('\*/' if comment\_open\_count < comment\_close\_count else '/\*')

return False

if \_\_name\_\_ == '\_\_main\_\_':

with open('input.cpp', 'r', encoding='utf-8') as file:

code = file.read()

balance\_errors = balance\_check(code)

if not balance\_errors:

errors\_message = ", ".join([f"Missing {error}" for error in errors['missing']])

errors\_message += ", " if errors\_message and errors['extra'] else ""

errors\_message += ", ".join([f"Extra {error}" for error in errors['extra']])

print(f"SYNTAX ERROR! {errors\_message}")

else:

result = Analyzer.analyze(code)

headers = ['Number', 'Category', 'Value']

with open("output.txt", "w", encoding='utf-8') as output\_file:

import sys

sys.stdout = output\_file

print(tabulate(list(result), headers=headers))

sys.stdout = sys.\_\_stdout\_\_

Листинг 2 – Программный код, описывающий константные значения

KEYWORDS = {

'break', 'case', 'const', 'continue', 'default', 'do',

'else', 'for', 'if', 'include', 'struct',

'return', 'sizeof', 'endl', 'cin', 'cout',

'static', 'switch', 'typedef', 'while', 'class', 'private',

'public', 'protected'

}

TYPES = {

'auto', 'char', 'double', 'float', 'int', 'long', 'short', 'signed', 'void', 'unsigned'

}

OPERATORS = {

'+', '-', '\*', '/', '%', '&', '|', '^', '!', '~', '++', '--',

'==', '!=', '>', '<', '>=', '<=', '&&', '||', '<<', '>>', '?', ':',

'=', '+=', '-=', '\*=', '/=', '%=', '&=', '|=', '^=', '[', ']', '.', '\'', '"',

}

INVALID\_OPERATORS = {

'~=', '@=', '#=', '$='

}

SEPARATORS = {

'{', '}', '(', ')', ';', ',', '\n', '\t'

}

SPECIAL\_SYMBOLS = {

'#'

}

Листинг 3 – Программный код класса лексического анализатора

import re

from constants import KEYWORDS, INVALID\_OPERATORS, OPERATORS, SPECIAL\_SYMBOLS, SEPARATORS, TYPES

class LexicalAnalyzer:

@staticmethod

def analyze(text: str):

text = LexicalAnalyzer.\_remove\_comments(text)

result = []

position = 0

ID = 1

while position < len(text):

current\_char = text[position]

if current\_char in SPECIAL\_SYMBOLS:

result.append((ID, 'Special symbol', current\_char))

ID += 1

position += 1

continue

if current\_char.isspace():

position += 1

continue

if current\_char == '0' and (text[position + 1] == 'b' or text[position + 1] == 'B'):

binary\_literal = text[position] + text[position + 1] + \

LexicalAnalyzer.\_read\_while\_with\_one(text, position + 2, lambda c: c in ['0', '1'])

result.append((ID, 'Binary Literal', binary\_literal))

ID += 1

position += len(binary\_literal) + 2

continue

if current\_char.isdigit():

literal = LexicalAnalyzer.\_read\_while\_with\_two(text, position,

lambda c: c.isdigit() or c in ['.', 'E', 'E'],

lambda c, next\_c: c in ['E', 'e'] and next\_c in ['+',

'-'])

# Error 1

if literal.count('.') > 1:

result.append((ID, '~~~ERROR!!!', literal))

else:

result.append((ID, 'Numeric Literal', literal))

ID += 1

position += len(literal)

continue

if current\_char.isalpha():

identifier = LexicalAnalyzer.\_read\_while\_with\_one(text, position, lambda c: c.isalnum())

# Error 2

if len(result) > 1 and result[-1][2].isnumeric():

result[-1] = (result[-1][0], '~~~ERROR!!!', result[-1][2] + identifier)

position += len(identifier)

continue

if identifier in KEYWORDS:

result.append((ID, 'Keyword', identifier))

elif identifier == 'nullptr':

result.append((ID, 'Literal Pointer', identifier))

elif identifier == 'std':

result.append((ID, 'Namespace', identifier))

elif identifier in TYPES:

result.append((ID, 'Data type', identifier))

else:

result.append((ID, 'Identifier', identifier))

ID += 1

position += len(identifier)

continue

if len(result) > 1 and result[-1][2] == 'include' and current\_char == '<':

identifier = LexicalAnalyzer.\_read\_while\_with\_one(text, position,

lambda c: c.isalpha() or c in ['.', '>', '<', '"'])

result.append((ID, 'Identifier', identifier))

ID += 1

position += len(identifier)

continue

if current\_char == '"' or current\_char == '\'':

result.append((ID, 'Separator', current\_char))

ID += 1

position += 1

literal = LexicalAnalyzer.\_read\_while\_with\_one(text, position, lambda c: c != current\_char)

parts = re.split(r'(?<!\\)(\\n|\\t)', literal)

while '' in parts:

parts.remove('')

for part in parts:

if part in ['\\n', '\\t']:

result.append((ID, 'Separator', part))

else:

result.append((ID, 'String Literal', part))

ID += 1

result.append((ID, 'Separator', current\_char))

ID += 1

position += len(literal) + 1

continue

if current\_char in OPERATORS:

operator = LexicalAnalyzer.\_read\_while\_with\_one(text, position, lambda c: c in OPERATORS)

# Error 3

if len(result) > 1 and result[-1][2] + operator in INVALID\_OPERATORS:

result[-1] = (result[-1][0], '~~~ERROR!!!', result[-1][2] + operator)

position += len(operator)

continue

result.append((ID, 'Operator', operator))

ID += 1

position += len(operator)

continue

# Error 4

if current\_char not in SEPARATORS:

result.append((ID, '~~~ERROR!!!', current\_char))

else:

result.append((ID, 'Separator', current\_char))

ID += 1

position += 1

return result

@staticmethod

def \_read\_while\_with\_one(text, start, condition):

end = start

while end < len(text) and condition(text[end]):

end += 1

return text[start:end]

@staticmethod

def \_read\_while\_with\_two(text, start, condition1, condition2):

end = start

while end < len(text) - 1 and condition1(text[end]):

if condition2(text[end], text[end + 1]):

end += 2

continue

end += 1

return text[start:end]

@staticmethod

def \_remove\_comments(text):

in\_comment = False

in\_line\_comment = False

result = ''

i = 0

while i < len(text):

if not in\_comment and text[i:i + 2] == '/\*':

in\_comment = True

i += 2

elif in\_comment and text[i:i + 2] == '\*/':

in\_comment = False

i += 2

elif not in\_comment and text[i:i + 2] == '//':

in\_line\_comment = True

i += 2

elif in\_line\_comment and text[i] == '\n':

in\_line\_comment = False

i += 1

elif not in\_comment and not in\_line\_comment:

result += text[i]

i += 1

else:

i += 1

return result