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             А. К. Хрищанович

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 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.

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

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

2 Приведение типов: при помощи приведения типов были созданы проверки строк, чтобы выяснить, является ли она целым числом, символом, числом с плавающей точкой, логическим значением или строкой соответственно.

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 кодом были использованы функции записи в файл и чтения из файла для удобства работы.

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

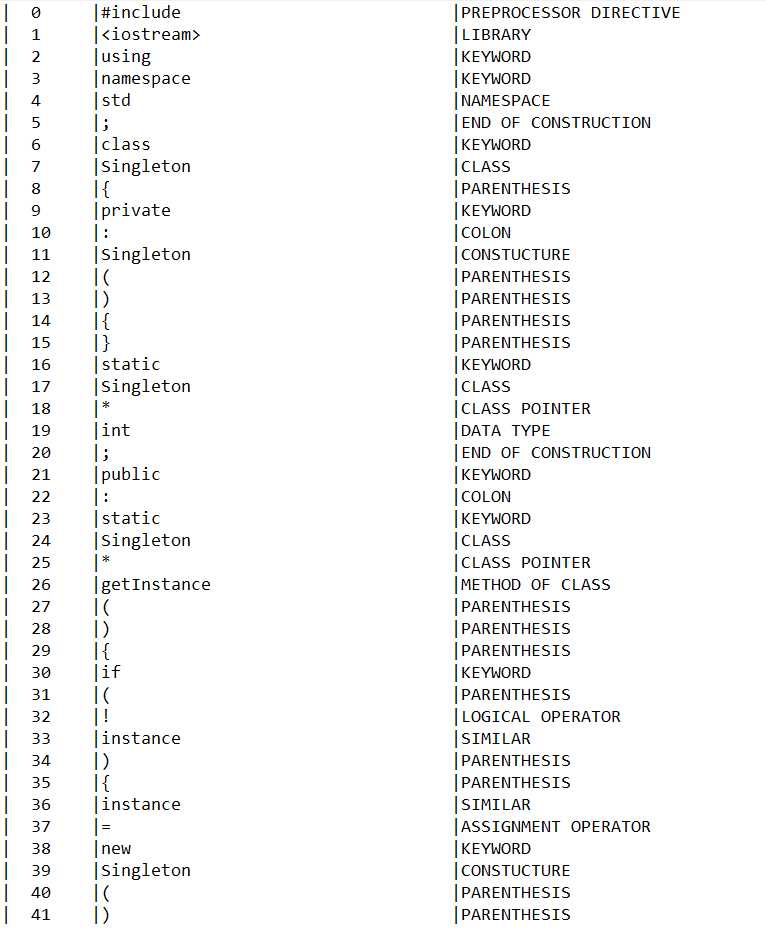


Рисунок 3.1 – Результат работы лексического анализатора

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 при неправильном названии переменной

Если совершить ошибку при написании ключевого слова, то программа выдаст ошибку и предложит возможно похожее слово, из словаря ключевых слов. Результат нахождения данной ошибки представлен на рисунке 3.3.



Рисунок 3.3 – Результат нахождения ошибки при неправильном написании директивы препроцессора

Если в коде будет некорректно представлено имя токена, программа также выдаст ошибку. Результат нахождения данной ошибки представлен на рисунке 3.4.



Рисунок 3.4 – Результат нахождения ошибки при пропущенном символе при подключении библиотеки

Если допустить синтаксическую ошибку в коде, к которой относятся недостающие закрывающиеся или открывающиеся скобки, нехватка закрывающихся или открывающихся кавычек, программа выдаст предупреждение о том, что нарушен баланс кода. Дальнейшая работа программы будет приостановлена. Результат нахождения данной 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 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/. – Дата доступа: 04.02.2024.
2. Введение в С++ [Электронный ресурс]. – Режим доступа: <https://metanit.com/cpp/tutorial/2.5.php>. – Дата доступа: 28.01.2024.
3. Типы данных [Электронный ресурс]. – Режим доступа: <https://metanit.com/cpp/tutorial/2.3.php>. – Дата доступа: 28.01.2024.
4. Операторы в С++ [Электронный ресурс]. – Режим доступа: <https://learn.microsoft.com/ru-ru/cpp/c-operators>. – Дата доступа: 04.02.2024.
5. Функции С++ [Электронный ресурс]. – Режим доступа: https://metanit.com/cpp/tutorial/3.1.php. – Дата доступа: 04.02.2024.
6. Классы С++ [Электронный ресурс]. – Режим доступа: <https://ravesli.com/urok-113-klassy-obekty-i-metody-klassov/>. – Дата доступа: 04.02.2024.
7. The C++ Standard Template Library (STL) [Электронный ресурс]. – Режим доступа: https://www.geeksforgeeks.org/the-c-standard-template-library-stl/. – Дата доступа: 30.01.2024.

## ПРИЛОЖЕНИЕ А

## (обязательное)

## Листинг исходного кода

Листинг 1 – Программная код функции классификации

from constants import keywords, data\_types, operators, standart\_function, standart\_libraries, containers, classes, \

special\_symbols

from type\_cheking import is\_integer\_type, is\_character\_type, is\_float\_type, is\_bool, is\_string

list\_data\_types = []

list\_containers = []

lexical\_error\_tokens = []

allowed\_symbols = 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ\_1234567890'

def check\_match(element: str, identifiers):

ident = [iden[1] for iden in identifiers]

keyws = list(keywords.keys())

datas = list(data\_types.keys())

conts = list(containers.keys())

base = ident + datas + keyws + conts

max\_match = ""

max\_count = 0

for string in base:

count = 0

i = 0

while i < min(len(string), len(element)):

if string[i] == element[i]:

count += 1

else:

break

i += 1

if count > max\_count:

max\_count = count

max\_match = string

return max\_match

def classify\_token(token, prev\_token, next\_token, token\_table):

if token in keywords:

return keywords[token]

elif token in standart\_function:

return standart\_function[token]

elif token in standart\_libraries:

return standart\_libraries[token]

elif token in containers:

list\_containers.append(token)

return containers[token]

elif token in classes:

return classes[token]

elif token in operators:

if token == '~':

if token\_table[next\_token] == 'ClASS':

token\_table[token] = 'TILDA'

return 'TILDA'

else:

return 'BITWISE OPERATOR'

if token == '<':

if prev\_token in containers:

return 'SIGN'

else:

return 'COMPARISON OPERATOR'

if token == '>':

if prev\_token in data\_types or (prev\_token in token\_table and token\_table[prev\_token] == 'CLASS'):

token\_table[token] = 'SIGN'

print(token\_table)

return 'SIGN'

else:

return 'COMPARISON OPERATOR'

if token == '\*':

if prev\_token in token\_table and token\_table[prev\_token] == 'CLASS':

token\_table[token] = 'CLASS POINTER'

return 'CLASS POINTER'

if len(list\_data\_types) != 0:

if next\_token == '(':

return 'ARITHMETIC OPERATOR'

if prev\_token == ')':

if next\_token == '(':

return 'ARITHMETIC OPERATOR'

return 'ARITHMETIC OPERATOR'

if ((prev\_token in token\_table and f'VARIABLE ({list\_data\_types[-1].upper()})' in token\_table[

prev\_token]) \

or (prev\_token in token\_table and f'FUNCTION (POINTER {list\_data\_types[-1].upper()})' in

token\_table[prev\_token]) \

or (prev\_token in token\_table and f'POINTER' in token\_table[

prev\_token]) \

or (prev\_token in token\_table and f'METHOD OF CLASS') \

or (prev\_token in token\_table and f'OBJECT' in token\_table[prev\_token])) \

and ((next\_token in token\_table and f'VARIABLE ({list\_data\_types[-1].upper()})' in token\_table[

next\_token]) \

or (next\_token in token\_table and f'FUNCTION (POINTER {list\_data\_types[-1].upper()})' in

token\_table[next\_token]) \

or (next\_token in token\_table and f'POINTER' in token\_table[

next\_token]) \

or (next\_token in token\_table and f'METHOD OF CLASS')):

return 'ARITHMETIC OPERATOR'

if prev\_token in data\_types and (next\_token in token\_table and f'VARIABLE ({list\_data\_types[-1].upper()})' in token\_table[next\_token]) \

or (next\_token in token\_table and f'FUNCTION (POINTER {list\_data\_types[-1].upper()})' in token\_table[next\_token]) \

or (next\_token in token\_table and f'POINTER' in token\_table[next\_token]) \

or (next\_token in token\_table and f'METHOD OF CLASS'):

token\_table[token] = 'ASTERIK'

return 'ASTERIK'

elif prev\_token in data\_types or (next\_token in token\_table and f'VARIABLE ({list\_data\_types[-1].upper()})' in token\_table[next\_token]) \

or (next\_token in token\_table and f'FUNCTION (POINTER {list\_data\_types[-1].upper()})' in token\_table[next\_token]) \

or (next\_token in token\_table and f'POINTER' in token\_table[next\_token]) \

or (next\_token in token\_table and f'METHOD OF CLASS'):

token\_table[token] = 'ASTERIK'

return 'ASTERIK'

return 'ARITHMETIC OPERATOR'

if token in ('++', '--', '&&', '=='):

if prev\_token:

if prev\_token in ('+', '-', '&', '=', '++', '--', '&&', '=='):

return 'LEXICAL ERROR'

if token in ('+', '-', '&', '='):

if prev\_token:

if prev\_token in ('&&', '++', '--', '=='):

return 'LEXICAL ERROR'

if token == '&':

if prev\_token in data\_types or (prev\_token in token\_table and token\_table[prev\_token] == 'CLASS') \

or (prev\_token in classes) or prev\_token in keywords:

token\_table[token] = 'REFERENCE OPERATOR'

return 'REFERENCE OPERATOR'

else:

token\_table[token] = 'BITWISE OPERATOR'

return 'BITWISE OPERATOR'

if token == '.':

if prev\_token:

if is\_float\_type(prev\_token):

return 'LEXICAL ERROR'

token\_table[token] = operators[token]

return operators[token]

elif token in data\_types:

if prev\_token in ('/\*', '//'):

return 'STRING OF COMMENT'

list\_data\_types.append(token)

return data\_types[token]

# elif token not in data\_types:

# list\_data\_types.append('')

elif token.endswith('[]'):

return 'ARRAY'

elif is\_string(token):

if prev\_token == '#include':

return 'HEADER FILE'

return 'STRING'

elif is\_integer\_type(token):

if prev\_token == '.':

return 'LEXICAL ERROR'

return 'INTEGER'

elif is\_float\_type(token):

return 'FLOAT'

elif is\_bool(token):

return 'BOOLEAN'

elif is\_character\_type(token):

return 'CHAR'

elif token.isidentifier():

if prev\_token in ('/\*', '//'):

return 'STRING OF COMMENT'

if not all(char in allowed\_symbols for char in token):

if token[0].isdigit():

lexical\_error\_tokens.append(token)

return 'LEXICAL ERROR! Identifier cannot start with a digit.'

return 'LEXICAL ERROR! Token contains invalid symbols.'

if any(char in token for char in special\_symbols):

return 'LEXICAL ERROR! Special symbols detected!'

if token[0].isdigit():

lexical\_error\_tokens.append(token)

return 'LEXICAL ERROR! Identifier cannot start with a digit.'

if token in token\_table:

if next\_token == '(' and token\_table[token] == 'CLASS':

return 'CONSTUCTURE'

return token\_table[token]

else:

if prev\_token == '#define':

token\_table[token] = 'VARIABLE'

return 'VARIABLE'

if prev\_token == '~' and next\_token == '(':

return 'DESCTRUCTURE'

if prev\_token == '::':

return 'METHOD'

if prev\_token == '>' and (prev\_token in token\_table and token\_table[prev\_token] == 'SIGN'):

if len(list\_containers) != 0:

token\_table[token] = f'CONTAINER {list\_containers[-1].upper()}'

return f'CONTAINER {list\_containers[-1].upper()}'

elif prev\_token == 'using':

token\_table[token] = 'ALIAS'

return 'ALIAS'

elif next\_token == '[':

if prev\_token in data\_types:

token\_table[token] = f'ARRAY ({list\_data\_types[-1]})'

return f'ARRAY ({list\_data\_types[-1]})'

elif prev\_token == '\*':

if prev\_token in token\_table and token\_table[prev\_token] == 'ASTERIK':

token\_table[token] = f'ARRAY (POINTER {list\_data\_types[-1]})'

return f'ARRAY (POINTER {list\_data\_types[-1]})'

elif next\_token == '(':

if prev\_token in data\_types:

token\_table[token] = f'FUNCTION ({list\_data\_types[-1].upper()})'

return f'FUNCTION ({list\_data\_types[-1].upper()})'

elif prev\_token == '\*':

if prev\_token in token\_table and token\_table[prev\_token] == 'CLASS POINTER':

token\_table[token] = f'METHOD OF CLASS'

return f'METHOD OF CLASS'

if len(list\_data\_types) != 0:

token\_table[token] = f'FUNCTION (POINTER {list\_data\_types[-1].upper()})'

else:

token\_table[token] = f'FUNCTION (POINTER)'

return f'FUNCTION (POINTER)'

return f'FUNCTION (POINTER {list\_data\_types[-1].upper()})'

else:

if prev\_token == '\*' and (prev\_token in token\_table and token\_table[prev\_token] == 'CLASS POINTER'):

token\_table[token] = f'METHOD OF CLASS'

return f'METHOD OF CLASS'

if prev\_token in data\_types or prev\_token == ',':

if len(list\_data\_types) != 0:

token\_table[token] = f'VARIABLE ({list\_data\_types[-1].upper()})'

return f'VARIABLE ({list\_data\_types[-1].upper()})'

else:

token\_table[token] = f'VARIABLE'

return f'VARIABLE'

elif prev\_token == '\*':

if len(list\_data\_types) != 0:

# token\_table[token] = f'POINTER ({list\_data\_types[-1].upper()})'

# return f'POINTER ({list\_data\_types[-1].upper()})'

token\_table[token] = f'POINTER'

return f'POINTER'

else:

token\_table[token] = 'IDENTIFICATOR'

return 'IDENTIFICATOR'

if next\_token == '{':

if prev\_token in 'class':

token\_table[token] = 'CLASS'

return 'CLASS'

if prev\_token in 'struct':

token\_table[token] = 'STRUCTURE'

return 'STRUCTURE'

elif prev\_token in token\_table and token\_table[prev\_token] == 'REFERENCE OPERATOR':

token\_table[token] = 'REFERENCE'

return 'REFERENCE'

elif prev\_token in token\_table and token\_table[prev\_token] == 'CLASS' or prev\_token in classes:

token\_table[token] = f'OBJECT OF {prev\_token}'

return f'OBJECT OF {prev\_token}'

elif prev\_token == '.':

return 'METHOD'

else:

match = check\_match(token, token\_table.items())

# if prev\_token in token\_table and token\_table[prev\_token] == 'SIMILAR':

# token\_table[token] = 'IDENTIFIER'

# return 'IDENTIFIER'

if match:

token\_table[token] = 'SIMILAR'

return f'LEXICAL ERROR! SIMILAR TO: {match}'

else:

return f'LEXICAL ERROR! UNRECOGNIZED IDENTIFIER: {token}'

else:

if prev\_token in ('/\*', '//'):

return 'STRING OF COMMENT'

elif prev\_token == '#include':

return 'HEADER FILE'

elif prev\_token == '.':

return 'METHOD'

elif token.endswith(('LL', 'll', 'UL', 'ul')):

if is\_integer\_type(token[:-2]):

return 'INTEGER'

elif is\_float\_type(token[:-2]):

return 'FLOAT'

else:

token\_table[token] = 'IDENTIFICATOR'

return 'IDENTIFICATOR'

elif token.endswith(('ULL', 'ull')):

if is\_integer\_type(token[:-3]):

return 'INTEGER'

elif is\_float\_type(token[:-3]):

return 'FLOAT'

else:

token\_table[token] = 'IDENTIFICATOR'

return 'IDENTIFICATOR'

elif token.endswith(('L', 'l', 'F', 'f', 'D', 'd')):

if is\_integer\_type(token[:-1]):

return 'INTEGER'

elif is\_float\_type(token[:-1]):

return 'FLOAT'

else:

if token[0].isdigit():

return 'LEXICAL ERROR! Identifier cannot start with a digit.'

if any(char in token for char in special\_symbols):

return 'LEXICAL ERROR! Special symbols detected!'

token\_table[token] = 'IDENTIFICATOR'

return 'IDENTIFICATOR'

else:

match = check\_match(token, token\_table.items())

# if prev\_token in token\_table and token\_table[prev\_token] == 'SIMILAR':

# token\_table[token] = 'IDENTIFIER'

# return 'IDENTIFIER'

if match:

token\_table[token] = 'SIMILAR'

return f'LEXICAL ERROR! SIMILAR TO: {match}'

else:

return f'LEXICAL ERROR! UNRECOGNIZED IDENTIFIER: {token}'

Листинг 2 – Программная код, описывающий константные значения

keywords = {  
 '#include': 'INCLUSION DIRECTIVE',  
 'using': 'KEYWORD',  
 'namespace': 'KEYWORD',  
 'std': 'NAMESPACE',  
 'break': 'KEYWORD',  
 'continue': 'KEYWORD',  
 'while': 'KEYWORD',  
 'for': 'KEYWORD',  
 'if': 'KEYWORD',  
 'else': 'KEYWORD',  
 'return': 'KEYWORD',  
 'switch': 'KEYWORD',  
 'case': 'KEYWORD',  
 'default':'KEYWORD',  
 'std::swap': 'KEYWORD',  
 'private:': 'KEYWORD',  
 'public:': 'KEYWORD',  
 '(': 'PARENTHESIS',  
 ')': 'PARENTHESIS',  
 '{': 'PARENTHESIS',  
 '}': 'PARENTHESIS',  
 '[': 'PARENTHESIS',  
 ']': 'PARENTHESIS',  
 ',': 'COMA',  
 ';': 'END OF CONSTRUCTION',  
 'std::cout': 'OUTPUT',  
 'std::endl': 'OUTPUT MANIPULATOR',  
 'std::vector': 'CONTAINER',  
 'std::list': 'CONTAINER',  
 'std::deque': 'CONTAINER',  
 'std::queue': 'CONTAINER',  
 'std::stack': 'CONTAINER',  
 'std::set': 'CONTAINER',  
 'std::unordered\_set': 'CONTAINER',  
 'std::multiset': 'CONTAINER',  
 'std::map': 'CONTAINER',  
 'std::unordered\_map': 'CONTAINER',  
 'std::multimap': 'CONTAINER',  
 'std::bitset': 'CONTAINER',  
 'class': 'CLASS',  
 'struct': 'STRUCT',  
 'const': 'CONST'  
}  
operators = {  
 '.': 'OPERATOR',  
 '+': 'ARITHMETIC OPERATOR',  
 '-': 'ARITHMETIC OPERATOR',  
 '\*': 'ARITHMETIC OPERATOR',  
 '/': 'ARITHMETIC OPERATOR',  
 '%': 'ARITHMETIC OPERATOR',  
 '++': 'ARITHMETIC OPERATOR',  
 '--': 'ARITHMETIC OPERATOR',  
 '+=': 'ARITHMETIC OPERATOR',  
 '-=': 'ARITHMETIC OPERATOR',  
 '\*=': 'ARITHMETIC OPERATOR',  
 '/=': 'ARITHMETIC OPERATOR',  
 '%=': 'ARITHMETIC OPERATOR',  
 '=': 'ADDIGNMENT OPERATOR',  
 'pow': 'FUNCTION',  
 'sqrt': 'FUNCTION',  
 'sin': 'FUNCTION',  
 'cos': 'FUNCTION',  
 'log': 'FUNCTION',  
 'exp': 'FUNCTION',  
 'round': 'FUNCTION',  
 '==': 'COMPARISON OPERATOR',  
 '!=': 'COMPARISON OPERATOR',  
 '<': 'COMPARISON OPERATOR',  
 '>': 'COMPARISON OPERATOR',  
 '<=': 'COMPARISON OPERATOR',  
 '>=': 'COMPARISON OPERATOR',  
 '>>': 'BITWISE OPERATOR',  
 '<<': 'BITWISE OPERATOR',  
 '<<': 'I/O OPERATOR',  
 '>>': 'I/O OPERATOR',  
 '&&': 'LOGICAL OPERATOR',  
 '||': 'LOGICAL OPERATOR',  
 '!': 'LOGICAL OPERATOR',  
 '&': 'BITWISE OPERATOR',  
 '|': 'BITWISE OPERATOR',  
 '^': 'BITWISE OPERATOR',  
 '~': 'BITWISE OPERATOR',  
 '?': 'TERNARY OPERATOR'  
}  
data\_types = {  
 'bool': 'DATA TYPE',  
 'signed char': 'DATA TYPE',  
 'char': 'DATA TYPE',  
 'short': 'DATA TYPE',  
 'int': 'DATA TYPE',  
 'long': 'DATA TYPE',  
 'long long': 'DATA TYPE',  
 'unsigned char': 'DATA TYPE',  
 'unsigned short': 'DATA TYPE',  
 'unsigned int': 'DATA TYPE',  
 'unsigned long': 'DATA TYPE',  
 'unsigned long long': 'DATA TYPE',  
 'float': 'DATA TYPE',  
 'double': 'DATA TYPE',  
 'long double': 'DATA TYPE',  
 'wchar\_t': 'DATA TYPE',  
 'char8\_t': 'DATA TYPE',  
 'char16\_t': 'DATA TYPE',  
 'char32\_t': 'DATA TYPE',  
 'std::string': 'DATA TYPE',}

Листинг 3 – Программный код функций чтения и записи

def read\_code\_from\_file(file\_path):

with open(file\_path, 'r') as file:

code = file.read()

return code

def write\_output\_to\_file(output, file\_path):

with open(file\_path, 'w') as file:

file.write(output)

Листинг 4 – Программный код главной функции

from parsing import tokenize

from classification import classify\_token

from function import read\_code\_from\_file, write\_output\_to\_file

file\_path\_input = 'test.cpp'

file\_path\_output = 'output.txt'

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

code = read\_code\_from\_file(file\_path\_input)

tokens = tokenize(code)

token\_table = {}

output = ''

output += "+----------------------------------------------------------------+\n"

output += "| № | Элемент | Информация |\n"

output += "+----------------------------------------------------------------+\n"

for i, token in enumerate(tokens):

classification = classify\_token(token, tokens[i - 1] if i > 0 else None,

tokens[i + 1] if i < len(tokens) - 1 else None, token\_table)

output += f"| {i:<3} |{token:<23} |{classification:<30} \n"

output += "+----------------------------------------------------------------+\n"

write\_output\_to\_file(output, file\_path\_output)

Листинг 5 – Программный код, реализующий разделение на токены

def tokenize(code):

tokens = []

current\_token = ''

inside\_string = False

inside\_comment = False

line\_number = 1

char\_position = 0

i = 0

while i < len(code):

char = code[i]

char\_position += 1

if char == '\n':

line\_number += 1

char\_position = 0

inside\_comment = False

if char == '\\':

if i + 1 < len(code):

next\_char = code[i + 1]

if next\_char in ('r', 'n', 't', '\\'):

current\_token += char + next\_char

i += 1

else:

current\_token += char

else:

current\_token += char

elif (char == '"' or char == "'") and not inside\_string:

if current\_token:

tokens.append(current\_token)

current\_token = ''

current\_token += char

inside\_string = True

elif (char == '"' or char == "'") and inside\_string:

current\_token += char

tokens.append(current\_token)

current\_token = ''

inside\_string = False

elif inside\_string:

current\_token += char

elif char == '/' and i + 1 < len(code) and code[i + 1] == '/':

if current\_token:

tokens.append(current\_token)

current\_token = ''

current\_token += char + code[i + 1]

i += 1

tokens.append(current\_token)

current\_token = ''

i += 1

while i < len(code) and code[i] != '\n':

current\_token += code[i]

i += 1

tokens.append(current\_token.strip())

current\_token = ''

elif char == '/' and i + 1 < len(code) and code[i + 1] == '\*':

if current\_token:

tokens.append(current\_token)

current\_token = ''

tokens.append('/\*')

inside\_comment = True

i += 1

elif char == '\*' and i + 1 < len(code) and code[i + 1] == '/':

if current\_token:

tokens.append(current\_token)

current\_token = ''

current\_token = '\*/'

inside\_comment = False

tokens.append(current\_token)

current\_token = ''

i += 1

elif inside\_comment:

current\_token += char

elif char in ('№','@','#','$'):

current\_token += char

i += 1

while i < len(code) and not code[i].isspace():

current\_token += code[i]

i += 1

tokens.append(current\_token)

current\_token = ''

elif char in ('(', ')', '{', '}', '[', ']', ',', ';', '<', '>'):

if current\_token:

tokens.append(current\_token)

current\_token = ''

tokens.append(char)

elif char.isspace():

if current\_token:

tokens.append(current\_token)

current\_token = ''

elif char in ('(', ')', '{', '}', '[', ']', ',', ';', '<', '>'):

if current\_token:

tokens.append(current\_token)

current\_token = ''

tokens.append(char)

elif char.isspace():

if current\_token:

tokens.append(current\_token)

current\_token = ''

elif char == '.':

if current\_token.isdigit() and i + 1 < len(code) and code[i + 1].isdigit():

current\_token += char

elif current\_token:

tokens.append(current\_token)

tokens.append(char)

current\_token = ''

else:

tokens.append(char)

elif char.isalnum() or char == '\_':

current\_token += char

elif char.isdigit() and i + 1 < len(code) and code[i + 1] in ('e', 'E'):

current\_token += char

current\_token += code[i + 1]

i += 1

if i + 1 < len(code) and code[i + 1] in ('+', '-'):

current\_token += code[i + 1]

i += 1

while i + 1 < len(code) and code[i + 1].isdigit():

current\_token += code[i + 1]

i += 1

tokens.append(current\_token)

current\_token = ''

elif char.isdigit() and current\_token.endswith(('l', 'L', 'u', 'U')):

tokens.append(current\_token[:-1])

tokens.append(current\_token[-1])

current\_token = ''

current\_token += char

elif current\_token.endswith(('ll', 'LL', 'uu', 'UU', 'el', 'EL')):

print('ct', current\_token)

tokens.append(current\_token[:-2])

tokens.append(current\_token[-2])

current\_token = ''

current\_token += char

else:

if current\_token:

tokens.append(current\_token)

current\_token = ''

tokens.append(char)

i += 1

if current\_token:

tokens.append(current\_token)

combined\_tokens = []

i = 0

while i < len(tokens):

if tokens[i - 1] == '#include' and tokens[i] == '<' and tokens[i + 1] != '<':

end\_index = i + 1

while end\_index < len(tokens) and tokens[end\_index] != '>':

end\_index += 1

if end\_index < len(tokens) and tokens[end\_index] == '>':

combined\_tokens.append('<' + ''.join(tokens[i + 1:end\_index]) + '>')

i = end\_index + 1

if tokens[i:i + 2] == ['<', '<']:

combined\_tokens.append('<<')

i += 2

elif tokens[i:i + 2] == ['>', '>']:

combined\_tokens.append('>>')

i += 2

elif tokens[i:i + 2] == [':', ':']:

combined\_tokens.append('::')

i += 2

elif tokens[i:i + 2] == ['-', '>']:

combined\_tokens.append('->')

i += 2

elif tokens[i:i + 2] == ['+', '+']:

combined\_tokens.append('++')

i += 2

elif tokens[i:i + 2] == ['-', '-']:

combined\_tokens.append('--')

i += 2

elif tokens[i:i + 2] == ['+', '=']:

combined\_tokens.append('+=')

i += 2

elif tokens[i:i + 2] == ['-', '=']:

combined\_tokens.append('-=')

i += 2

elif tokens[i:i + 2] == ['\*', '=']:

combined\_tokens.append('\*=')

i += 2

elif tokens[i:i + 2] == ['/', '=']:

combined\_tokens.append('/=')

i += 2

elif tokens[i:i + 2] == ['%', '=']:

combined\_tokens.append('%=')

i += 2

elif tokens[i:i + 2] == ['=', '=']:

combined\_tokens.append('==')

i += 2

elif tokens[i:i + 2] == ['!', '=']:

combined\_tokens.append('!=')

i += 2

elif tokens[i:i + 2] == ['>', '=']:

combined\_tokens.append('>=')

i += 2

elif tokens[i:i + 2] == ['<', '=']:

combined\_tokens.append('<=')

i += 2

elif tokens[i:i + 2] == ['>', '>']:

combined\_tokens.append('!=')

i += 2

elif tokens[i:i + 2] == ['&', '&']:

combined\_tokens.append('&&')

i += 2

elif tokens[i:i + 2] == ['|', '|']:

combined\_tokens.append('||')

i += 2

elif tokens[i:i + 3] == ['>', '>', '=']:

combined\_tokens.append('>>=')

i += 3

elif tokens[i:i + 3] == ['<', '<', '=']:

combined\_tokens.append('<<=')

i += 3

elif tokens[i:i + 2] == ['&', '=']:

combined\_tokens.append('&=')

i += 2

elif tokens[i:i + 2] == ['|', '=']:

combined\_tokens.append('|=')

i += 2

elif tokens[i:i + 2] == ['^', '=']:

combined\_tokens.append('^=')

i += 2

elif tokens[i:i + 2] == ['/', '/']:

combined\_tokens.append('//')

i += 2

elif tokens[i:i + 2] == ['/', '\*']:

combined\_tokens.append('/\*')

i += 2

elif tokens[i:i + 2] == ['\*', '/']:

combined\_tokens.append('\*/')

i += 2

elif tokens[i:i + 3] == ['unsigned', 'long', 'long']:

combined\_tokens.append('unsigned long long')

i += 3

elif tokens[i:i + 2] == ['unsigned', 'char']:

combined\_tokens.append('unsigned char')

i += 2

elif tokens[i:i + 2] == ['unsigned', 'short']:

combined\_tokens.append('unsigned short')

i += 2

elif tokens[i:i + 2] == ['unsigned', 'int']:

combined\_tokens.append('unsigned int')

i += 2

elif tokens[i:i + 2] == ['unsigned', 'long']:

combined\_tokens.append('unsigned long')

i += 2

elif tokens[i:i + 4] == ['unsigned', 'long', 'long', 'double']:

combined\_tokens.append('unsigned long long double')

i += 4

elif tokens[i:i + 2] == ['long', 'long']:

combined\_tokens.append('long long')

i += 2

elif tokens[i:i + 2] == ['long', 'int']:

combined\_tokens.append('long int')

i += 2

elif tokens[i:i + 2] == ['short', 'int']:

combined\_tokens.append('short int')

i += 2

elif tokens[i:i + 2] == ['long', 'double']:

combined\_tokens.append('long double')

i += 2

elif tokens[i:i + 2] == ['signed', 'char']:

combined\_tokens.append('signed char')

i += 2

else:

combined\_tokens.append(tokens[i])

i += 1

return combined\_tokens

Листинг 6 – Программный код, реализующий проверку типов

def is\_integer\_type(s):

try:

int(s)

return True

except ValueError:

return False

def is\_float\_type(s):

try:

float(s)

return True

except ValueError:

return False

def is\_character\_type(s):

return isinstance(s, str) and s.startswith('"') and s.endswith('"') and len(s) == 1

def is\_bool(s):

return s.lower() in ['true', 'false']

def is\_string(s):

return (isinstance(s, str) and s.startswith('"') and s.endswith('"')) or (isinstance(s, str) and s.startswith("'") and s.endswith("'"))

def is\_library(s):

return isinstance(s, str) and s.startswith('<') and s.endswith('>')