Министерство образования Республики Беларусь

Учреждения образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ

И РАДИОЭЛЕКТРОНИКИ

КАФЕДРА ИНФОРМАТИКИ

Отчет по лабораторной работе №2

По теме «Лексический анализ»

Выполнил:

студент гр. 053502

Песоцкий В.А.

Проверил:

Гриценко Н. Ю.

Минск 2023

**СОДЕРЖАНИЕ**

1. Цель работы3
2. Теория4
3. Демонстрация работы5

Примечание. Код программы9

**1 ЦЕЛЬ РАБОТЫ**

Разработка лексического анализатора подмножества языка программирования, определенного в лабораторной работе 1. Программа определяет лексические правила и выполняет перевод потока символов программы лабораторной работы 1 в поток лексем(токенов).

**2 ТЕОРИЯ**

Лексический анализатор (или сканер) – часть компилятора, которая читает литеры программы на исходном языке и строит из них слова (лексемы) исходного языка. На вход лексического анализатора поступает текст исходной программы, а выходная информация передается для дальнейшей обработки компилятором на этапе синтаксического анализа и разбора.

Лексема (лексическая единица языка) – структурная единица языка, которая состоит из элементарных символов языка и не содержит в своем составе других структурных единиц языка. Лексемами языков программирования являются идентификаторы, константы, ключевые слова языка, знаки операций и т.п. Состав возможных лексем каждого конкретного языка программирования определяется синтаксисом этого языка.

Она также называется последовательностью символов (буквенно-цифровых) в токене. Есть несколько предопределенных правил для каждой лексемы, которая будет определена как действительный токен. Эти правила определяются грамматическими правилами посредством шаблона.

Перед тем как пройти до синтаксического и семантического анализаторов, все символы должны быть разбиты на токены, определены их типы. На рисунке 2.1 представленна обобщенная стандартная модель компилятора.

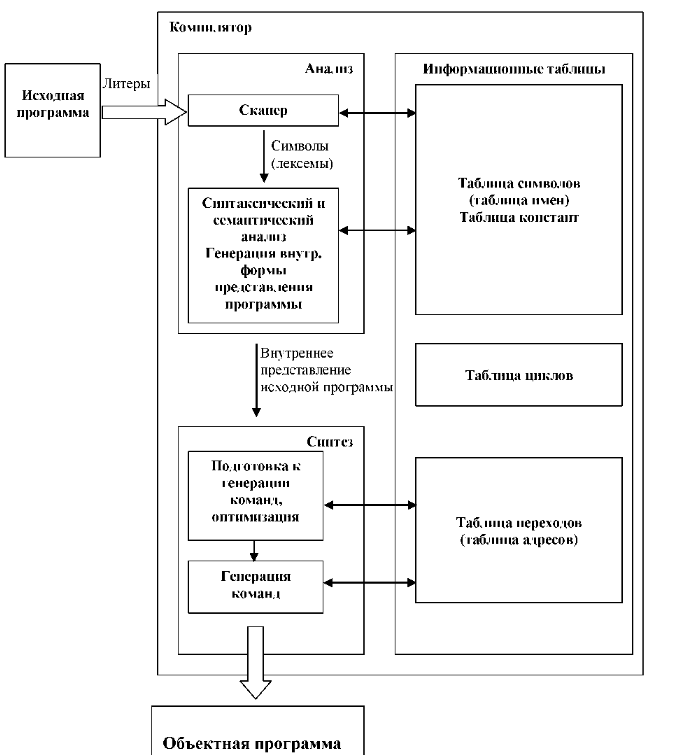


Рисунок 2.1 - Обобщенная стандартная модель компилятора

# **3 ДЕМОНСТРАЦИЯ РАБОТЫ**

На рисунке 3.1 представлены все ключевые слова, константы, возможные паттерны применения строковых выражений и операторы в языке C#.

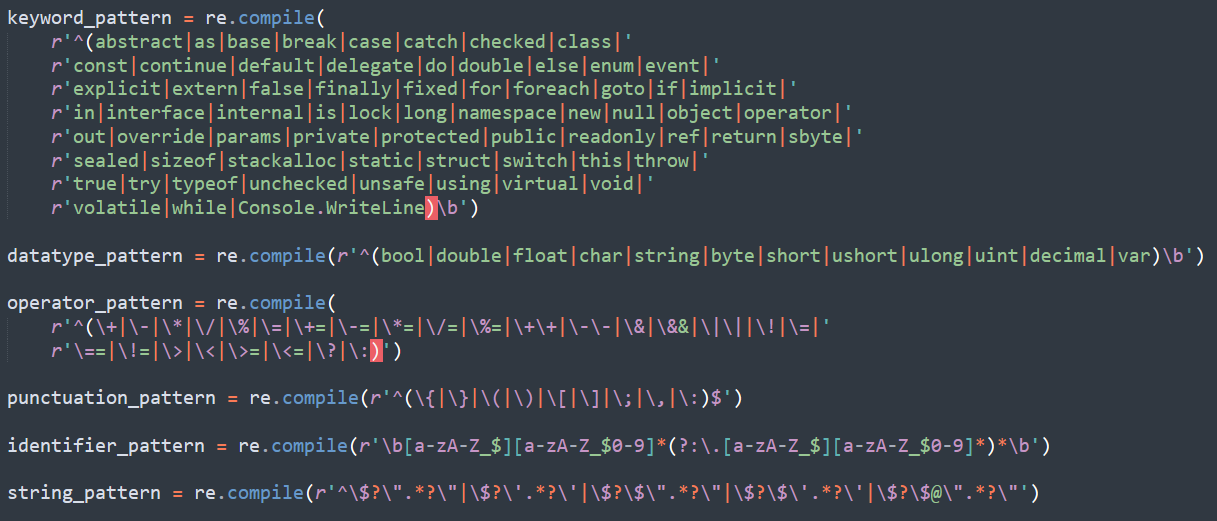
****

Рисунок 3.1 - Перечисление регулярных выражений для лексического анализа

В качестве входного кода для лексического анализа, используем алгоритм пузырьковой сортировки. Символов в алгоритме пузырьковой сортировки слишком много, поэтому ниже будут представлены таблицы в неполном объеме. На рисунке 3.2 представлена таблица переменных.

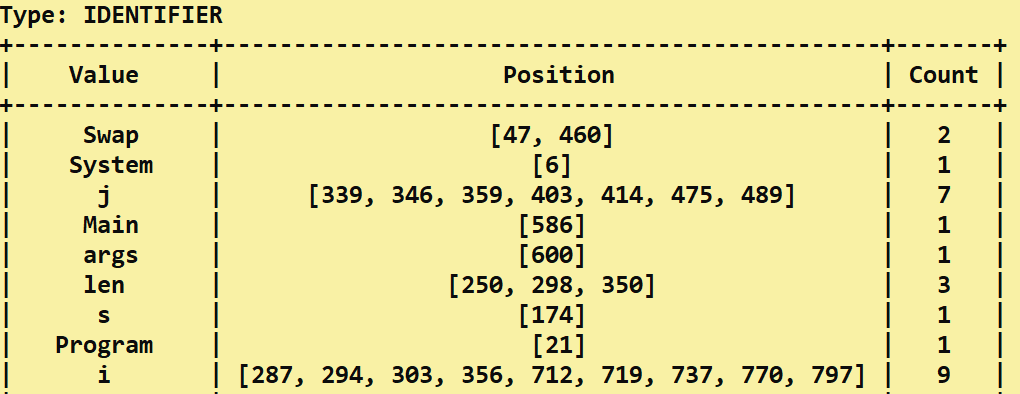


Рисунок 3.2 - Таблица переменных

На рисунке 3.3 представлена таблица ключевых слов.

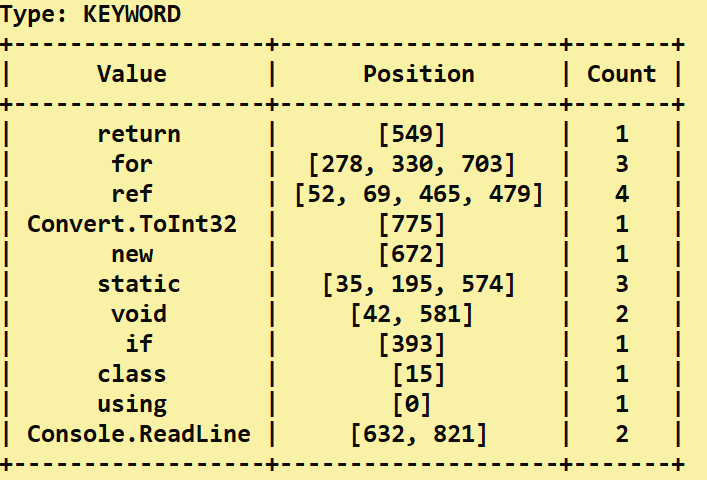


Рисунок 3.3 - Таблица ключевых слов

На рисунке 3.4 представлена таблица операторов.

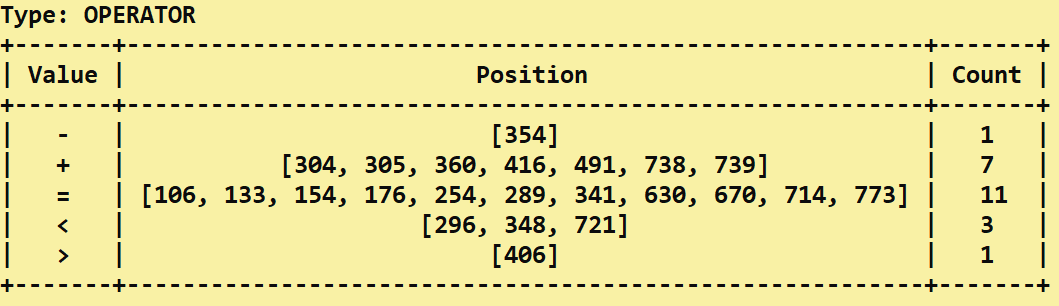


Рисунок 3.4 - Таблица операторов

На рисунке 3.5 представлена таблица символов пунктуации.

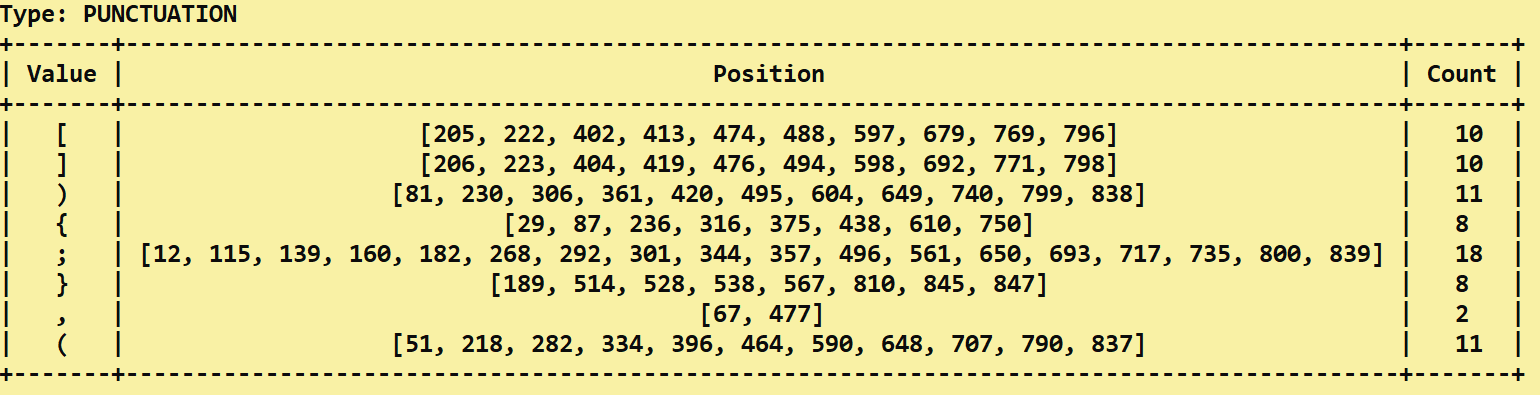


Рисунок 3.5 - Таблица символов пунктуации

На рисунке 3.6 представлена таблица символов пунктуации.

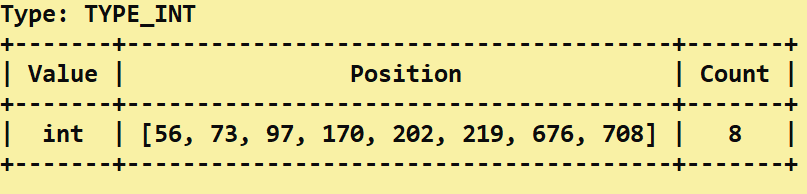


Рисунок 3.6 - Таблица определения одного из типов данных

Далее будет продемонстрировано определение определенных лексических ошибок в коде (если они присутствуют). Если ошибок в коде несколько, то приложение с анализатором будет прерываться на первой, ждать ее исправления, а далее покажет следующую ошибку.

На рисунке 3.7 отображена ошибка некорректного использования знака пунктуации.

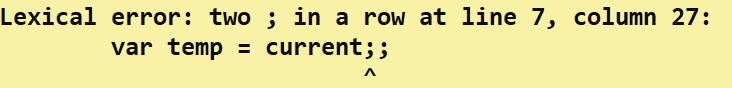


Рисунок 3.7 - Лексическая ошибка «повторное использование знака пунктуации “;”»

 На рисунке 3.8 отображена лексическая ошибка, которая возникает при некорректном использовании типа данных. В данном случае дважды определен тип «int».

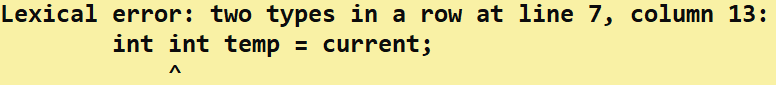


Рисунок 3.8 - Лексическая ошибка “type”

На рисунке 3.9 отображена лексическая ошибка, которая возникает при неверном использовании оператора инкремента.

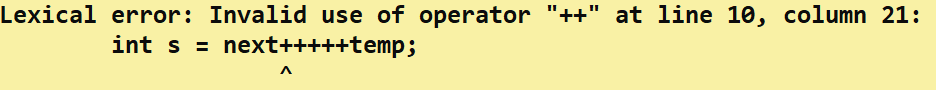


Рисунок 3.9 - Лексическая ошибка “оператор ++”

На рисунке 3.10 отображена лексическая ошибка, которая возникает, если в случайном месте кода остались ненужные символы, случайно поставленные буквы.

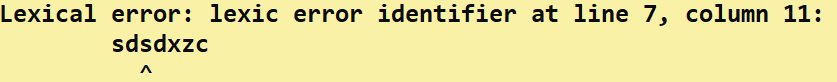


Рисунок 3.10 - Лексическая ошибка «неверное использование идентификатора»

На рисунке 3.11 продемонстрирована лексическая ошибка, которая возникает, если использовать символы, которых не существует в множестве языка C#.

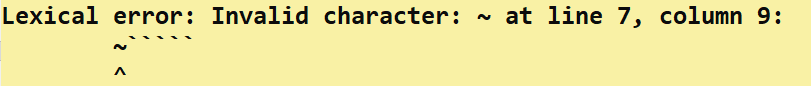


Рисунок 3.11 - Лексическая ошибка «некорректный символ»

На рисунке 3.12 продемонстрирована лексическая ошибка, которая возникает, при отсутствии необходимого знака пунктуации.

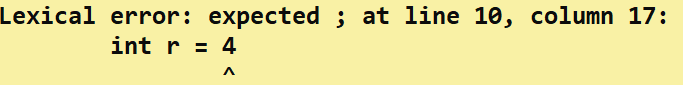


Рисунок 3.12 - Лексическая ошибка «;»

Приложение. Текст программы

**import** **re**

**from** **prettytable** **import** PrettyTable

**import** **sys**

# Определение регулярных выражений для лексического анализа

keyword\_pattern = re.compile(

r'^(abstract|as|base|break|case|catch|checked|class|'

r'const|continue|default|delegate|do|double|else|enum|event|'

r'explicit|extern|false|finally|fixed|for|foreach|goto|if|implicit|'

r'in|interface|internal|is|lock|long|namespace|new|null|object|operator|'

r'out|override|params|private|protected|public|readonly|ref|return|sbyte|'

r'sealed|sizeof|stackalloc|static|struct|switch|this|throw|'

r'true|try|typeof|unchecked|unsafe|using|virtual|void|'

r'volatile|while|Console.WriteLine)\b')

datatype\_pattern = re.compile(r'^(bool|double|float|char|string|byte|short|ushort|ulong|uint|decimal|var)\b')

operator\_pattern = re.compile(

r'^(\+|\-|\\*|\/|\%|\=|\+=|\-=|\\*=|\/=|\%=|\+\+|\-\-|\&|\&&|\|\||\!|\=|'

r'\==|\!=|\>|\<|\>=|\<=|\?|\:)')

punctuation\_pattern = re.compile(r'^(\{|\}|\(|\)|\[|\]|\;|\,|\:)$')

identifier\_pattern = re.compile(r'\b[a-zA-Z\_$][a-zA-Z\_$0-9]\*(?:\.[a-zA-Z\_$][a-zA-Z\_$0-9]\*)\*\b')

string\_pattern = re.compile(r'^\$?\".\*?\"|\$?**\'**.\*?**\'**|\$?\$\".\*?\"|\$?\$**\'**.\*?**\'**|\$?\$@\".\*?\"')

# Определение класса для лексемы

**class** **Token**:

**def** **\_\_init\_\_**(self, type, value, pos):

self.type = type

self.value = value

self.pos = pos

**def** **repr**(self):

**return** 'Token(type={type}, value={value}, pos={pos})'.format(

type=self.type, value=repr(self.value), pos=self.pos)

**class** **Lexic**:

**def** **\_\_init\_\_**(self, text):

self.text = text

self.pos = **0**

self.lst\_type = None

self.lst\_data\_type = None

**def** **error**(self):

**raise** **Exception**('Invalid character at position ' + str(self.pos))

**def** **report\_error**(self, message):

line\_num = self.text.count('**\n**', **0**, self.pos) + **1**

line\_start = self.text.rfind('**\n**', **0**, self.pos) + **1**

line\_end = self.text.find('**\n**', self.pos)

**if** line\_end == -**1**:

line\_end = len(self.text)

line = self.text[line\_start:line\_end]

col\_num = self.pos - line\_start + **1**

**print**('Lexical error: {} at line {}, column {}:**\n**{}**\n**{}^'.format(

message, line\_num, col\_num, line, ' ' \* (col\_num - **1**)))

**def** **get\_next\_token**(self):

**if** self.pos >= len(self.text):

**return** Token('EOF', '', self.pos)

current\_char = self.text[self.pos]

# установка начальной позиции

pos\_start = self.pos

# Определение типа лексемы

**if** current\_char == ' ' **or** current\_char == '**\t**':

self.pos += **1**

**return** self.get\_next\_token()

**elif** current\_char == '**\n**':

self.pos += **1**

**return** Token('NEWLINE', '**\n**', pos\_start)

**elif** keyword\_pattern.match(self.text[self.pos:]):

**if** self.lst\_type == 'IDENTIFIER':

self.pos -= **14**

self.report\_error('unexpected IDENTIFIER')

sys.exit()

match = keyword\_pattern.match(self.text[self.pos:])

self.lst\_type='KEYWORD'

self.pos += match.end()

**return** Token('KEYWORD', match.group(), pos\_start)

**elif** re.match(r'^int\b', self.text[self.pos:]):

**if** self.lst\_type == 'TYPE\_INT':

self.report\_error('two types in a row')

sys.exit()

**if** self.lst\_type == 'IDENTIFIER':

self.pos -=**14**

self.report\_error('lexic error identifier')

sys.exit()

**if** self.lst\_type != 'PUNCTUATION' **and** self.lst\_type != 'KEYWORD':

self.report\_error('expected PUNCTUATION or KEYWORD before TYPE')

sys.exit()

match = re.match(r'^int\b', self.text[self.pos:])

self.lst\_type='TYPE\_INT'

self.lst\_data\_type='TYPE\_INT'

self.pos += match.end()

**return** Token('TYPE\_INT', match.group(), pos\_start)

**elif** datatype\_pattern.match(self.text[self.pos:]):

concreteType = ((datatype\_pattern.match(self.text[self.pos:])).group()).upper()

bool\_reg = re.compile(r'^(true|false|1|0)$')

**print**(concreteType)

**if** self.lst\_type == f'TYPE\_{concreteType}':

self.report\_error('Two types in a row')

sys.exit()

**if** self.lst\_type == 'IDENTIFIER':

self.pos -= **14**

self.report\_error('lexic error identifier')

sys.exit()

**if** self.lst\_type != 'PUNCTUATION' **and** self.lst\_type != 'KEYWORD':

self.pos -= **5**

self.report\_error('expected ; before TYPE')

sys.exit()

self.lst\_type=f'TYPE\_{concreteType}'

self.lst\_data\_type =f'TYPE\_{concreteType}'

'''if self.lst\_data\_type == 'TYPE\_BOOL':

print(bool\_reg.match(self.text[self.pos:]))

if not bool\_reg.match(self.text[self.pos:]):

self.pos +=9

self.report\_error('datatype error');

sys.exit()'''

match = datatype\_pattern.match(self.text[self.pos:])

self.pos += match.end()

**return** Token(f'TYPE\_{concreteType}', match.group(), pos\_start)

**elif** operator\_pattern.match(current\_char):

**if** current\_char == '+':

**if** self.text[self.pos+**1**:self.pos+**3**] == '++':

self.report\_error('Invalid use of operator "++"')

self.pos += **2**

sys.exit()

**if** current\_char == '-':

**if** self.text[self.pos+**1**:self.pos+**3**] == '--':

self.report\_error('Invalid use of operator "--"')

self.pos += **2**

sys.exit()

**if** current\_char == '=':

**if** self.text[self.pos+**1**:self.pos+**3**] == '+=':

self.report\_error('Invalid use of operator "+="')

self.pos += **2**

sys.exit()

**if** current\_char == '=':

**if** self.text[self.pos+**1**:self.pos+**3**] == '%=':

self.report\_error('Invalid use of operator "%="')

self.pos += **2**

sys.exit()

**if** current\_char == '=':

**if** self.text[self.pos+**1**:self.pos+**3**] == '\*=':

self.report\_error('Invalid use of operator "\*="')

self.pos += **2**

sys.exit()

self.lst\_type = 'OPERATOR'

self.pos += **1**

**return** Token('OPERATOR', current\_char, pos\_start)

**elif** punctuation\_pattern.match(current\_char):

skobka = (punctuation\_pattern.match(current\_char)).group()

**if** current\_char == ';':

**if** self.text[self.pos+**1**] == ';':

self.report\_error('two ; in a row')

self.pos += **2**

sys.exit()

**if** current\_char == ',':

**if** self.text[self.pos+**1**] == ',':

self.report\_error('two , in a row')

self.pos += **2**

sys.exit()

**if** current\_char == '.':

**if** self.text[self.pos+**1**] == '.':

self.report\_error('two . in a row')

self.pos += **2**

sys.exit()

**if** self.lst\_type == 'IDENTIFIER' **and** re.match(if\_condition\_regex, self.text[self.pos:]):

**print**(re.match(if\_condition\_regex, self.text[self.pos:]))

self.report\_error('incorrect keyword before condition')

sys.exit()

self.lst\_type = 'PUNCTUATION'

self.pos += **1**

**return** Token('PUNCTUATION', current\_char, pos\_start)

**elif** re.match(r'^(\+|\-)?(\d+(\.\d\*)?|\.\d+)([eE][-+]?\d+)?', self.text[self.pos:]):

match = re.match(r'^(\+|\-)?(\d+(\.\d\*)?|\.\d+)([eE][-+]?\d+)?', self.text[self.pos:])

**if** match **is** None:

self.report\_error('Invalid number format')

sys.exit()

**return** self.get\_next\_token()

self.lst\_type = 'NUMBER'

**if** self.lst\_type == 'NUMBER':

**if** self.text[self.pos+**1**:self.pos+**2**] != ';':

self.report\_error('expected ;')

sys.exit()

**if** self

self.pos += match.end()

**return** Token('NUMBER', match.group(), pos\_start)

**elif** identifier\_pattern.match(self.text[self.pos:]):

**if** self.lst\_type == 'IDENTIFIER':

self.report\_error('Two IDENTIFIERs in a row')

sys.exit()

match = identifier\_pattern.match(self.text[self.pos:])

**if** match.group() **in** keyword\_pattern.findall(self.text):

self.report\_error('Identifier cannot be a keyword')

sys.exit()

self.lst\_type = 'IDENTIFIER'

self.pos += match.end()

**return** Token('IDENTIFIER', match.group(), pos\_start)

**elif** string\_pattern.match(self.text[self.pos:]):

**if** self.lst\_data\_type != 'TYPE\_STRING'**:**

self.report\_error('incorrect datatype assignment')

sys.exit()

match = string\_pattern.match(self.text[self.pos:])

**if** match **is** None:

self.report\_error('Invalid string format')

sys.exit()

**return** self.get\_next\_token()

self.lst\_type = 'STRING'

self.pos += match.end()

**return** Token('STRING', match.group(), pos\_start)

**else**:

self.report\_error('Invalid character: ' + current\_char)

self.pos += **1**

sys.exit()

**return** self.get\_next\_token()

filename = 'Program.cs'

**with** open(filename, 'r', encoding='utf-8-sig') **as** f:

text = f.read()

f.close()

lexer = Lexic(text)

tokens = []

**while** True:

token = lexer.get\_next\_token()

tokens.append(token)

**if** token.type == 'EOF':

**break**

'''def display\_tokens(tokens):

table = PrettyTable(['Type', 'Value', 'Position'])

for token in tokens:

table.add\_row([token.type, token.value, token.pos])

print(table)'''

**def** **display\_tokens**(tokens):

token\_types = set(token.type **for** token **in** tokens)

**for** token\_type **in** token\_types:

table = PrettyTable(['Value', 'Position'])

**for** token **in** tokens:

**if** token.type == token\_type:

table.add\_row([token.value, token.pos])

**print**(f'Type: {token\_type}')

**print**(table)

**print**()

display\_tokens(tokens)