Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

КАФЕДРА ИНФОРМАТИКИ

Отчёт по лабораторной работе № 2

по теме “Лексический анализ”

Выполнил:

студент гр. 053501

Волковский О. А.

Проверил:

Ассистент кафедры информатики Гриценко Н. Ю.

Минск 2023

Содержание

1 Цель работы 2

2 Теория 4

3 Примеры выделения лексем(токенов) 5

4 Нахождение лексических ошибок лексем(токенов) 6  
Вывод 7  
Приложение. Код программ 8

1 Цель работы

Освоение работы с существующими лексическими анализаторами (по желанию). Разработка лексического анализатора подмножества языка программирования, определенного в лабораторной работе 1. Определяются лексические правила. Выполняется перевод потока символов в поток лексем(токенов).

2 Теория

Лексический анализатор (сканер) читает поток символов, составляющих исходную программу, и группирует эти символы в значащие последовательности, называющиеся лексемами.

Лексема – это структурная единица языка, которая состоит из элементарных символов языка и не содержит в своём составе других структурных единиц языка. Лексемами языков программирования являются идентификаторы, константы, ключевые слова языка, знаки операций и т.п.

На вход лексического анализатора поступает текст исходной программы, а выходная информация передаётся для дальнейшей обработки синтаксическому анализатору. Для каждой лексемы сканер строит выходной токен (англ. token – знак, символ) вида: ‹имя\_токена, значение\_атрибута›

Одним из решений, которые могут повлиять на структуру всего компилятора, является выбор множества токенов. Вы можете иметь токены для каждого входного символа или несколько символов могутбыть объединены в один токен. Например, символы >, >=, >>, и >>= могут рассматриваться либо как четыре токена, либо как один токен оператор сравнения. При этом, лексема используется для устранения неоднозначности токена. Первый подход может упростить генерацию кода. Однако, слишком много токенов могут сделать парсер слишком большим и трудным в написании. Все возможные токены указаны lexer\_constants.regex\_map.

Далее необходимо вариант парсинга: рекурсивный или декларативный. Проблемы первым подходом — во-первых, избыточная сложность (по одному и тому же фрагменту текста гуляем взад-вперёд); во-вторых, неудобство поддержки (синтаксис языка оказывается рассредоточен по килобайтам и килобайтам ветвистого кода). Синтаксис языка можно задать декларативно. Например, всем знакомы регулярные выражения. Первая из названных проблем решается тем, что поиск всех реджексов в тексте можно выполнить за один проход, т.е. нет надобности хранить всю программу в памяти целиком — достаточно читать её по одному символу, обрабатывать символ, и тут же забывать. Вторая — тем, что теперь у нас есть централизованное, формальное

описание языка: можем менять реджексы, вовсе не трогая код; и наоборот, менять код, не рискуя повредить парсер. Парсер будет проверять все возможные реджексы пытаясь проверить строку с какого-то элемента на соответствие. При совпадении будет создаваться новый токен.

1. Примеры выделения лексем(токенов)

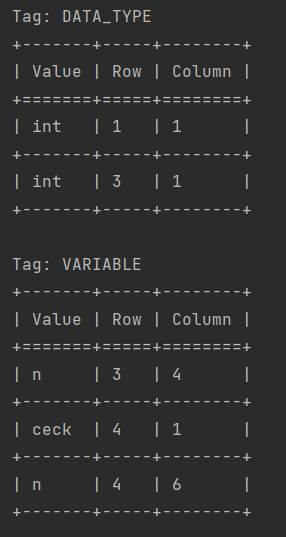


Рисунок 1.1 – переменные и типы данных

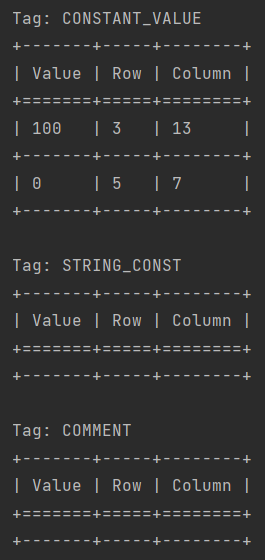


Рисунок 1.2 – константы

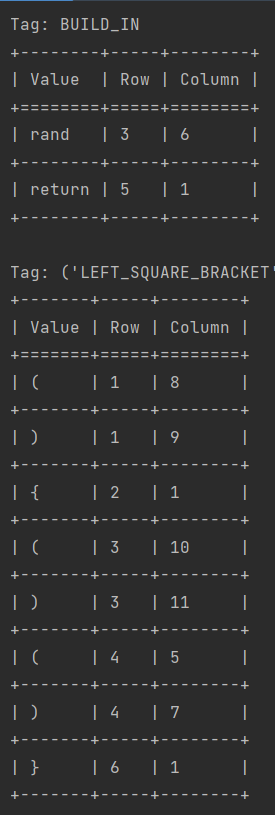


Рисунок 1.3 – встроенные выражения и скобки

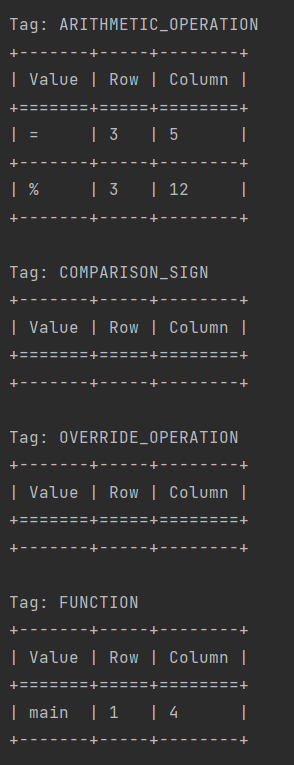


Рисунок 1.4 – операции и функции

1. Нахождение лексических ошибок

В нашем случае есть 2 возможных лексических ошибки – неверный оператор, неверное задание константного значение числа, а также некоторые другие проблемы, связанные с другими. Любая другая ошибка в коде предполагает рассмотрение смежных лексем, а значит относится к работе синтаксического анализатора.



Рисунок 2.1– пример некорректного инкремента



Рисунок 2.2 – пример некорректного декремента



Рисунок 2.3 – пример некорректного задания числа с плавающей точкой



Рисунок 2.4 – пример некорректного задания константного значения

Вывод

В результате выполнения лабораторной работы была изучена теория о лексических анализаторах и лексемах. Рассмотрена стадия лексического анализа.

Разработан лексический анализатор подмножества языка программирования Python, определенного в лабораторной работе 1. Программа анализатора определяет лексические правила и выполняет перевод потока символов программ лабораторной работы 1 в поток лексем (токенов).

Приложение. Код программы.

Файл CustomToken.py

class CustomToken:

def \_\_init\_\_(self, value, token\_type, line, column):

self.token\_type = token\_type

self.value = value

self.line = line

self.column = column

def \_\_str\_\_(self):

return f"token with value: {self.value}, type: {self.token\_type}, line {self.line}, column {self.column}"

Файл Lexer.py  
  
 import re

from CustomToken import CustomToken

from lexer\_constants import regex\_map, tokens\_types

class Lexer:

def \_\_init\_\_(self):

self.tokens = ()

self.errors = []

self.line\_num = 1

self.col\_num = 1

self.func\_table = {}

def get\_tokens(self, code):

self.tokens = []

self.errors = []

self.line\_num = 1

self.col\_num = 1

i = 0

while i < len(code):

match = None

for key, value in regex\_map.items():

tag, pattern = key, value

regex = re.compile(pattern)

match = regex.match(code[i:])

if match:

text = match.group(0)

if tag == "function":

text = text.split()

token = CustomToken(text[0], tokens\_types[text[0]], self.line\_num, self.col\_num)

self.tokens.append(token)

self.col\_num += len(text[0])

token = CustomToken(text[1], tokens\_types[tag], self.line\_num, self.col\_num)

self.func\_table[text[1]] = 1

self.tokens.append(token)

self.col\_num += len(text[1])

elif tag == "whitespace":

if '\n' in text:

self.line\_num += 1

self.col\_num = 1

elif tag == "operator":

if len(text) > 2:

self.errors.append(

f"Syntax error: unexpected operator '{text}' at line {self.line\_num}, column {self.col\_num}")

self.col\_num += len(text)

else:

token = CustomToken(text, tokens\_types[text], self.line\_num, self.col\_num)

self.tokens.append(token)

self.col\_num += len(text)

elif tag == "identifier" or tag == "string\_value":

token = CustomToken(text, tokens\_types[tag], self.line\_num, self.col\_num)

self.tokens.append(token)

self.col\_num += len(text)

elif tag == "const":

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

self.errors.append(

f"Syntax error: unexpected value '{text}' at line {self.line\_num}, column {self.col\_num}")

self.col\_num += len(text)

else:

token = CustomToken(text, tokens\_types[tag], self.line\_num, self.col\_num)

self.tokens.append(token)

self.col\_num += len(text)

else:

token = CustomToken(text, tokens\_types[text], self.line\_num, self.col\_num)

self.tokens.append(token)

self.col\_num += len(text)

break

if not match:

self.errors.append(

f"Syntax error: unexpected character '{code[i]}' at line {self.line\_num}, column {self.col\_num}")

self.col\_num += 1

i += 1

else:

i += len(match.group(0))

for i in range(len(self.tokens)):

if self.tokens[i].value in self.func\_table:

self.tokens[i].token\_type = tokens\_types["function"]

return self.tokens, self.errors  
  
Файл lexer\_constants.py  
  
tokens\_types = {

"int": "VARIABLE\_TYPE",

"float": "VARIABLE\_TYPE",

"double": "VARIABLE\_TYPE",

"char": "VARIABLE\_TYPE",

"void": "VARIABLE\_TYPE",

"string": "VARIABLE\_TYPE",

"identifier": "VARIABLE",

"const": "CONSTANT\_VALUE",

"string\_value": "CONSTANT\_VALUE\_STRING",

'comment': 'COMMENT',

"if": "BUILD\_IN",

"rand": "BUILD\_IN",

"else": "BUILD\_IN",

"while": "BUILD\_IN",

"for": "BUILD\_IN",

"break": "BUILD\_IN",

"continue": "BUILD\_IN",

"return": "BUILD\_IN",

"sizeof": "BUILD\_IN",

"cout": "BUILD\_IN",

"endl": "BUILD\_IN",

"[": "LEFT\_SQUARE\_BRACKET",

"]": "RIGHT\_SQUARE\_BRACKET",

"{": "LEFT\_CURLY\_BRACKET",

"}": "RIGHT\_CURLY\_BRACKET",

"(": "LEFT\_ROUND\_BRACKET",

")": "RIGHT\_ROUND\_BRACKET",

",": "COMMA",

";": "SEMICOLON",

"+": "ARITHMETIC\_OPERATION",

"-": "ARITHMETIC\_OPERATION",

"++": "ARITHMETIC\_OPERATION",

"--": "ARITHMETIC\_OPERATION",

"\*": "ARITHMETIC\_OPERATION",

"/": "ARITHMETIC\_OPERATION",

"%": "ARITHMETIC\_OPERATION",

"=": "ARITHMETIC\_OPERATION",

"+=": "ARITHMETIC\_OPERATION",

"-=": "ARITHMETIC\_OPERATION",

"<": "COMPARISON\_SIGN",

">": "COMPARISON\_SIGN",

"<=": "COMPARISON\_SIGN",

">=": "COMPARISON\_SIGN",

"==": "COMPARISON\_SIGN",

"!=": "COMPARISON\_SIGN",

"<<": "OVERRIDE\_OPERATION",

">>": "OVERRIDE\_OPERATION",

"function": "FUNCTION"

}

# Define regular expressions for C++ tokens

regex\_map = {

'function': r'(\w+)\s+(\w+)(?=\()([^)]\*?)\s\*',

'int': r'int\b',

'float': r'float\b',

'double': r'double\b',

'char': r'char\b',

'bool': r'bool\b',

'string': r'string\b',

'string\_value': r'\".+?\"',

'if': r'if\b',

'else': r'else\b',

'for': r'for\b',

'while': r'while\b',

'do': r'do\b',

'break': r'break\b',

'continue': r'continue\b',

'return': r'return\b',

'void': r'void\b',

'sizeof': r'sizeof\b',

'cin': r'cin\b',

'rand': r'rand\b',

'cout': r'cout\b',

'endl': r'endl\b',

'identifier': r'[a-zA-Z\_]\w\*',

'const': r'\d+(\.\d+)\*',

'operator': r'[+\-\*/%<>&|^=!]+|<<|>>',

'semicolon': r';',

'comma': r',',

'comment': r'//.\*?$|/\\*.\*?\\*/',

'whitespace': r'[\t|\n|\s|\r]+',

'left\_round\_bracket': r'\(',

'right\_round\_bracket': r'\)',

'left\_square\_bracket': r'\[',

'right\_square\_bracket': r'\]',

'left\_curly\_bracket': r'\{',

'right\_curly\_bracket': r'\}'

}