Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей Кафедра информатики

Дисциплина: Операционные среды и системное программирование

ОТЧЕТ

к лабораторной работе № 2

на тему

**ЛЕКСИЧЕСКИЙ АНАЛИЗ**

Выполнил:

студент гр. 153504

Фарисей Ю.А.

Проверил:

Гриценко Н.Ю.

Минск 2024

**СОДЕРЖАНИЕ**

[1 Цель работы 3](#_Toc146836467)

[2 Теоретические сведения 4](#_Toc146836468)

[3 Полученные результаты 4](#_Toc146836469)

[Выводы 7](#_Toc146836470)

[Список использованных источников 8](#_Toc146836471)

[Приложение А (обязательное) листинг кода 9](#_Toc146836472)

# **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 ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ** 

Цель лексического анализа — выделение и классификация лексем в тексте исходной программы. Программа, которая выполняет лексический анализ, называется сканером, или лексическим анализатором. Сканер производит посимвольное чтение файла с исходным текстом программы. [1]

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 вида: ‹*token\_name*, *attribute\_value*›

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.

В большом количестве компиляторов лексический анализ не выделен в отдельный проход. Отдельный проход — это роскошь, которая тянет за собой дополнительный расход памяти. Поэтому большинство компиляторов делает лексический анализ примерно так: лексема = получить очередную лексему. Эта функция вызывается из синтаксического анализатора. Это простое решение экономит память хотя бы потому, что не требуется хранить все встреченные лексемы. [2]

В простейшем случае фазы лексического и синтаксического анализа могут выполняться компилятором последовательно. Но для многих языков программирования информации на этапе лексического анализа может быть недостаточно для однозначного определения типа и границ очередной лексемы. Если невозможно определить границы лексем, то лексический анализ исходного текста должен выполняться поэтапно. Тогда лексический и синтаксический анализаторы должны функционировать параллельно, поочередно обращаясь друг к другу. Лексический анализатор, найдя очередную лексему, передает ее синтаксическому анализатору, тот пытается выполнить анализ считанной части исходной программы и может либо запросить у лексического анализатора следующую лексему, либо потребовать от него вернуться на несколько шагов назад и попробовать выделить лексемы с другими границами. При этом он может сообщить информацию о том, какую лексему следует ожидать. [3]

# **3 ПОЛУЧЕННЫЕ РЕЗУЛЬТАТЫ**

В результате выполнения лабораторной работы был написан парсер на языке программирования python.

Результат работы скрипта представляет собой таблицы, содержащие информацию о токенах (рисунок 1).

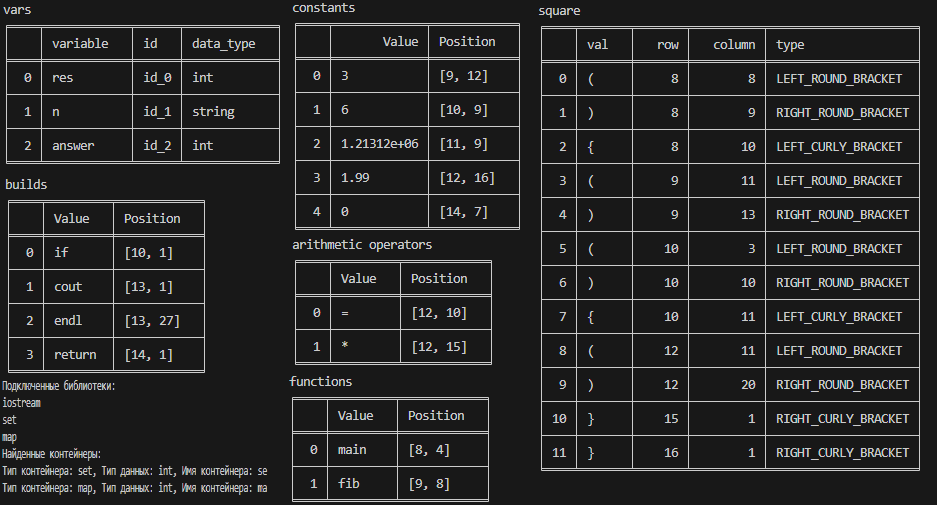


Рисунок 1 – Информация о токенах

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

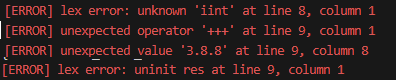


Рисунок 2 – Пример ошибок

# **ВЫВОДЫ**

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).

# **СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

1. Сложные структуры данных. Лексический анализ [Электронный ресурс]. – Режим доступа: https://programm.ws/page.php?id=611.
2. Лексический анализатор [Электронный ресурс]. – Режим доступа: http://www.compiler.su/leksicheskij-analizator.php.
3. Перенаправление ввода/вывода [Электронный ресурс]. – Режим доступа: https://selectel.ru/blog/tutorials/linux-redirection/.

# **ПРИЛОЖЕНИЕ А (обязательное) Листинг кода**

Листинг 1 **–** Файл *main.py*:

from lexer.lexer import Lexer

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

cpp\_file = "2.cpp"

with open(cpp\_file, "r") as file:

code = (file.read())

custom\_lexer = Lexer()

tokens, errors = custom\_lexer.get\_tokens(code)

print('\033[32mOK\033[0m')

Листинг 2 **–** Файл *lexer.py*:

import re

from tabulate import tabulate

from lexer.lexer\_constants import regex\_map, syntax\_types

import pandas as pd

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}"

class Lexer:

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

self.func\_list = []

self.tokens = []

self.errors = []

self.line\_num = 1

self.col\_num = 1

self.cur\_func = ""

def get\_tokens(self, code):

self.tokens = []

self.errors = []

self.line\_num = 1

self.col\_num = 1

self.cur\_func = ""

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()

if len(self.tokens) != 0:

self.func\_list.append((self.cur\_func, self.tokens))

self.tokens = []

if not text[0] in syntax\_types:

print(f"\033[31m[ERROR] lex error: unknown '{text[0]}' at line {self.line\_num}, column {self.col\_num}\033[0m")

exit()

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

self.tokens.append(token)

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

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

self.tokens.append(token)

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

self.cur\_func = text[1]

elif tag == "whitespace":

if '\n' in text:

self.line\_num += 1

self.col\_num = 1

elif tag == "operator":

if len(text) > 2 or self.tokens[-1].token\_type == "ARITHMETIC\_OPERATION" or text not in syntax\_types:

print(

f"\033[31m[ERROR] unexpected operator '{text}' at line {self.line\_num}, column {self.col\_num}\033[0m")

exit()

self.col\_num += len(text)

else:

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

self.tokens.append(token)

self.col\_num += len(text)

elif tag == "string\_value":

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

self.tokens.append(token)

self.col\_num += len(text)

elif tag == "identifier":

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

for item in self.func\_list:

if item[0] == text:

token.token\_type = "FUNCTION"

self.tokens.append(token)

self.col\_num += len(text)

elif tag == "const":

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

print(

f"\033[31m[ERROR] unexpected value '{text}' at line {self.line\_num}, column {self.col\_num}\033[0m")

exit()

self.col\_num += len(text)

else:

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

self.tokens.append(token)

self.col\_num += len(text)

else:

token = CustomToken(text, syntax\_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))

self.func\_list.append((self.cur\_func, self.tokens))

const\_map = dict()

var\_map = dict()

build\_map = dict()

square\_map = dict()

arithmetic\_map = dict()

function\_map = dict()

prev = None

s\_data = []

for token in self.tokens:

if token.token\_type == 'VARIABLE':

if not token.value in var\_map:

if prev == None:

print(f"\033[31m[ERROR] lex error: unknown {token.value} at line {token.line}, column {token.column}\033[0m")

exit()

elif prev.token\_type != 'DATA\_TYPE':

print(f"\033[31m[ERROR] lex error: unknown {token.value} at line {token.line}, column {token.column}\033[0m")

exit()

var\_map[token.value] = [f"id\_{len(var\_map)}", prev.value]

if token.token\_type == 'CONSTANT\_VALUE':

const\_map[token.value] = [token.line, token.column]

print(token.value, ' ---- ', token.token\_type)

if token.token\_type == 'BUILD\_IN':

build\_map[token.value] = [token.line, token.column]

if token.token\_type in ['RIGHT\_CURLY\_BRACKET', 'LEFT\_ROUND\_BRACKET', 'RIGHT\_ROUND\_BRACKET', 'LEFT\_CURLY\_BRACKET']:

square\_map[token.value] = [token.line, token.column, token.token\_type]

s\_data.append({'val': token.value, 'row': token.line, 'column': token.column, 'type': token.token\_type})

if token.token\_type == 'ARITHMETIC\_OPERATION':

arithmetic\_map[token.value] = [token.line, token.column]

if token.token\_type == 'FUNCTION':

function\_map[token.value] = [token.line, token.column]

prev = token

df = pd.DataFrame.from\_dict(var\_map, orient='index', columns=['id', 'data\_type'])

df.reset\_index(inplace=True)

df.columns = ['variable', 'id', 'data\_type']

print('vars')

print(tabulate(df, headers='keys', tablefmt='fancy\_grid'))

df = pd.DataFrame(const\_map.items(), columns=['Value', 'Position'])

print('constants')

print(tabulate(df, headers='keys', tablefmt='fancy\_grid'))

df = pd.DataFrame(build\_map.items(), columns=['Value', 'Position'])

print('builds')

print(tabulate(df, headers='keys', tablefmt='fancy\_grid'))

df = pd.DataFrame(s\_data)

print('square')

print(tabulate(df, headers='keys', tablefmt='fancy\_grid'))

df = pd.DataFrame(arithmetic\_map.items(), columns=['Value', 'Position'])

print('arithmetic operators')

print(tabulate(df, headers='keys', tablefmt='fancy\_grid'))

df = pd.DataFrame(function\_map.items(), columns=['Value', 'Position'])

print('functions')

print(tabulate(df, headers='keys', tablefmt='fancy\_grid'))

return self.func\_list, self.errors