Министерство образования Республики Беларусь  
Учреждение образования  
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

КАФЕДРА ИНФОРМАТИКИ

Отчёт по лабораторной работе №2

По теме “Лексический анализ программ

Выполнила: студентка гр. 053503 Зырянова М.М.

Проверил: ассистент кафедры информатики Гриценко Н. Ю.

Минск 2023

**СОДЕРЖАНИЕ**

[1 ЦЕЛЬ РАБОТЫ 3](#_Toc128355173)

[2 КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ 4](#_Toc128355174)

[3 РЕЗУЛЬТАТ РАБОТЫ АНАЛИЗАТОРА 6](#_Toc128355175)

[3.1 Выделенные лексемы 6](#_Toc128355176)

[3.2 Нахождение и обработка ошибок 8](#_Toc128355177)

[ЗАКЛЮЧЕНИЕ 9](#_Toc128355178)

[ПРИЛОЖЕНИЕ А 10](#_Toc128355179)

# **1 ЦЕЛЬ РАБОТЫ**

В ходе данной лабораторной работы необходимо освоить работу с существующими лексическими анализаторами, а также разработать лексичесий анализатор подмножества языка программирования, определенного в лабораторной работе №1. В данной лабораторной работе также определяются лексические правила, выполняется перевод потока символов в поток лексем (токенов).

Лексический анализатор читает поток символов, составляющих исходную программу, и группирует эти символы в значащие последовательности, называемые лексемами.

На вход программы подается текстовый файл, содержащий строки символов анализируемой программы.

Выражение может включать:

* Знаки сложения и умножения («+» и «\*»); Круглые скобки («(» и «)»);
* Константы (например, 5; 3.8; 1e+18, 8.41E–10);
* Имена переменных.

# **2 КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

Первая фаза компиляции называется лексическим анализом или сканированием.

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 вида:

*‹имя\_токена, значение\_атрибута›*

Первый компонент токена, имя\_токена, представляет собой абстрактный символ, использующийся во время синтаксического анализа, а второй компонент, значение атрибута, указывает на запись в таблице идентификаторов, соответствующую данному токену.

С теоретической точки зрения лексический анализатор не является обязательной, необходимой частью компилятора. Его функции могут выполняться на этапе синтаксического разбора. Однако существует несколько причин, исходя из которых в состав практически всех компиляторов включают лексический анализ:

* упрощается работа с текстом исходной программы на этапе синтаксического разбора и сокращается объём обрабатываемой информации, так как лексический анализатор структурирует поступающий на вход исходный текст программы и удаляет всю незначащую информацию;
* для выделения в тексте и разбора лексем можно применять простую, эффективную и теоретически хорошо проработанную технику анализа, в то время как на этапе синтаксического анализа конструкций исходного языка используются достаточно сложные алгоритмы разбора;
* сканер отделяет сложный по конструкции синтаксический анализатор от работы непосредственно с текстом исходной программы, структура которого может варьироваться в зависимости от версии входного языка – при такой конструкции компилятора при переходе от одной версии языка к другой достаточно только перестроить относительно простой сканер.

Функции, выполняемые лексическим анализатором, и состав лексем, которые он выделяет в тексте исходной программы, могут меняться в зависимости от версии компилятора. В основном лексические анализаторы выполняют исключение из текста исходной программы комментариев и незначащих пробелов, а также выделение лексем следующих типов: идентификаторов, строковых, символьных и числовых констант, ключевых (служебных) слов входного языка.

В большинстве компиляторов лексический и синтаксический анализаторы – это взаимосвязанные части. Лексический разбор исходного текста в таком варианте выполняется поэтапно, так что синтаксический анализатор, выполнив разбор очередной конструкции языка, обращается к сканеру за следующей лексемой. При этом он может сообщить информацию о том, какую лексему следует ожидать. В процессе разбора может даже происходить «откат назад», чтобы выполнить анализ текста на другой основе.

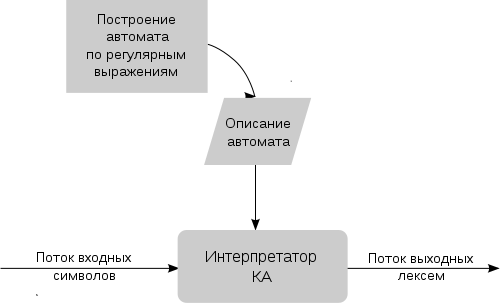


Рисунок 1 – Принцип работы логического анализатора

# **3 РЕЗУЛЬТАТ РАБОТЫ АНАЛИЗАТОРА**

## **3.1 Выделенные лексемы**

В рамках данной лабораторной работы был разработан анализатор, с помощью которого был проведён анализ кода, представленного в лабораторной работе №1. Ниже представлены результаты работы написанного анализатора.

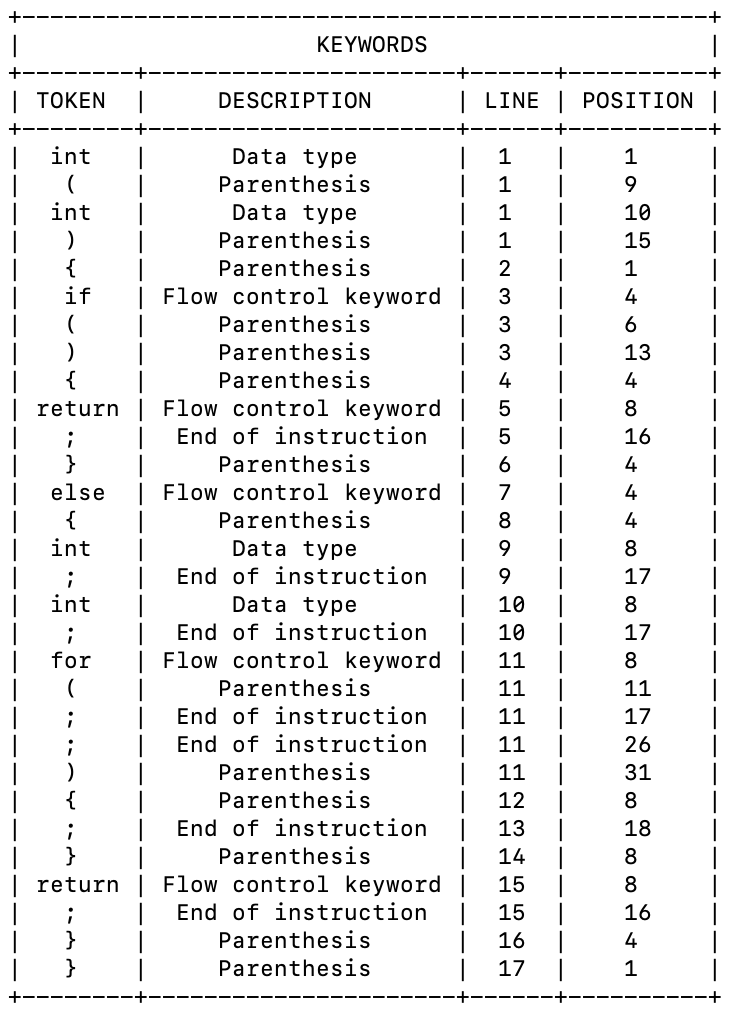


Рисунок 2 – Токены, представляющие собой ключевые слова

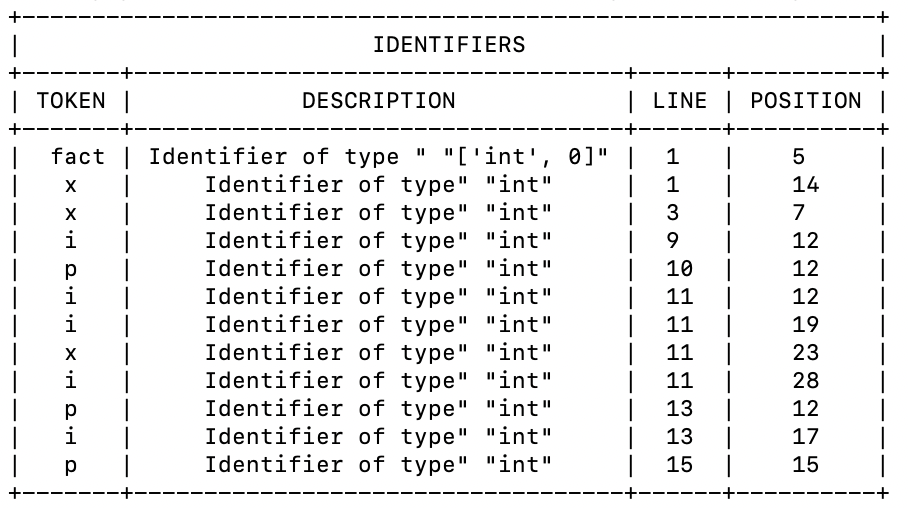


Рисунок 3 – Токены-идентификаторы

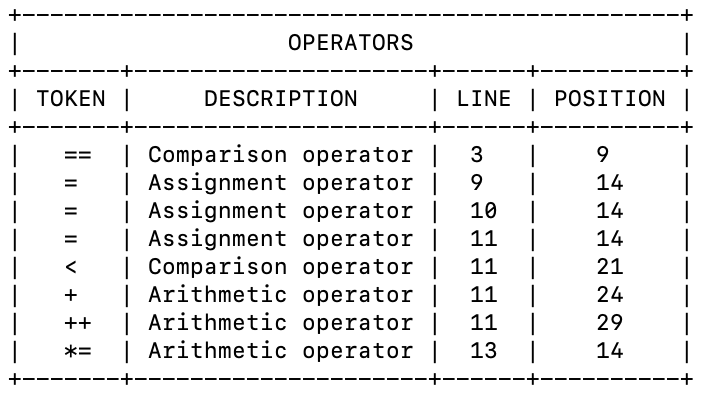


Рисунок 4 – Токены-операторы

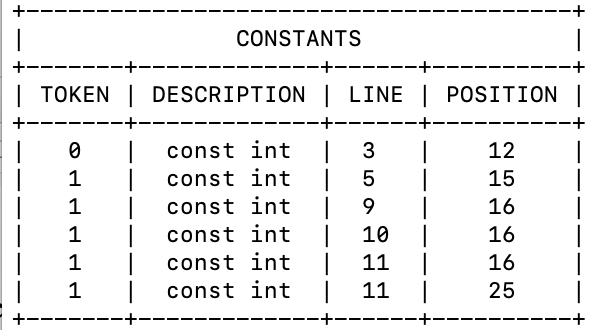


Рисунок 5 – Токены-константы

## **3.2 Нахождение и обработка ошибок**

Иногда исходный код может содержать ряд ошибок, поэтому необходимо находить и исправлять их. Ниже представлены ошибки в коде и результаты работы программы при их наличии.

Если попытаться в качестве оператора использовать знак нижнего подчёркивания, то мы получим лексическую ошибку.



Рисунок 5 – Обработка некорректно введённого оператора

При попытке использовать слово return в качестве имени переменной, мы получим следующую ошибку.



Рисунок 6 – Обработка использования зарезервированного слова

При добавлении лишнего плюса в инкрементный оператор также получаем лексическую ошибку.

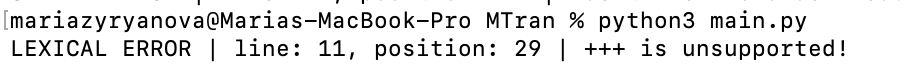


Рисунок 7 – Обработка использования неверного оператора

# **ЗАКЛЮЧЕНИЕ**

В ходе выполнения лабораторной работы был изучен принцип работы типового лексического анализатора. Также была предложена авторская реализация лексического анализатора и продемонстрирована её работа на тестовой программе, включая вычленение токенов из потока байтов, их анализ и обработку лексических ошибок (приведено четыре примера).

Таким образом, сложность лексического анализа в некоторой степени состоит в том, что для корректного определения текущей лексемы (или для определения ошибки) нам нужно понять какие символы идут за ней.

Лексический анализатор представляет собой первую фазу компилятора, его основная задача состоит в чтении входных символов исходной программы, их группировании в лексемы и вывод последовательностей токенов для всех лексем исходной программы. Поток токенов пересылается синтаксическому анализатору для разбора.

# **ПРИЛОЖЕНИЕ А**

**(обязательное)**

**Реализация анализатора**

а) analyzer.py

import re, prettytable, patterns

from patterns import (

TOKEN\_PATTERN,

IDENTIFIER\_PATTERN,

IDENTIFIER\_DESC,

INT\_PATTERN,

FLOAT\_PATTERN,

STRING\_PATTERN

)

from errors import(

LexicalError,

SyntaxError

)

class Analyzer(object):

def \_\_init\_\_(self, source) -> None:

self.lines = source.split('\n')

self.identifiers = {}

self.errors = False

self.functions = {

'read': 'string',

'readint': 'int',

'readfloat': 'float',

'sqrt': 'float',

'pow': 'int'

}

def get\_tokens(self) -> list:

res = []

for i, l in enumerate(self.lines):

matching = re.finditer(TOKEN\_PATTERN, l)

res.extend([(i+1, m.start(0)+1, m.group(0)) for m in matching if m.group(0) != ' '])

return res

def analyze(self, show\_tables=False):

fields = ['TOKEN', 'DESCRIPTION', 'LINE', 'POSITION']

tokens = self.get\_tokens()

identifiers = prettytable.PrettyTable()

constants = prettytable.PrettyTable()

operators = prettytable.PrettyTable()

keywords = prettytable.PrettyTable()

identifiers.title = 'IDENTIFIERS'

constants.title = 'CONSTANTS'

operators.title = 'OPERATORS'

keywords.title = 'KEYWORDS'

identifiers.field\_names = fields

constants.field\_names = fields

operators.field\_names = fields

keywords.field\_names = fields

for index, (line, position, token) in enumerate(tokens):

if token in patterns.keywords:

keywords.add\_row([token, patterns.keywords[token], line, position])

continue

if token in patterns.operators:

operators.add\_row([token, patterns.operators[token], line, position])

continue

if token in patterns.data\_types:

keywords.add\_row([token, patterns.data\_types[token], line, position])

continue

\_identifier = re.match(IDENTIFIER\_PATTERN, token)

\_integer = re.match(INT\_PATTERN, token)

\_float = re.match(FLOAT\_PATTERN, token)

\_string = re.match(STRING\_PATTERN, token)

if \_identifier:

\_constant = False

\_function = False

try:

\_constant = tokens[index-2][2] == 'const'

except:

pass

try:

\_function = tokens[index+1][2] == '('

except:

pass

try:

\_previous = tokens[index-1][2]

if \_previous in patterns.data\_types:

if token in self.identifiers:

self.errors = True

raise SyntaxError('Identifier is duplicated!', line, position)

if \_function:

self.functions[token] = [\_previous, 0]

else:

self.identifiers[token] = \_previous

else:

if token not in self.identifiers and token not in self.functions:

self.errors = True

raise SyntaxError('Identifier is undefined!', line, position)

if \_function:

identifiers.add\_row([token, f'{IDENTIFIER\_DESC} "{"const" if \_constant else ""} "{self.functions[token]}"', line, position])

else:

identifiers.add\_row([token, f'{IDENTIFIER\_DESC}"{"const" if \_constant else ""} "{self.identifiers[token]}"', line, position])

continue

except IndexError:

if token in self.identifiers:

if \_function:

identifiers.add\_row([token, f'{IDENTIFIER\_DESC} {"const " if \_constant else ""} "{self.functions[token]}"', line, position])

else:

identifiers.add\_row([token, f'{IDENTIFIER\_DESC}"{"const" if \_constant else ""} "{self.identifiers[token]}"', line, position])

continue

else:

self.errors = True

raise SyntaxError('Identifier is undefined!', line, position)

if \_integer:

constants.add\_row([token, 'const int', line, position])

continue

if \_float:

constants.add\_row([token, 'const float', line, position])

continue

if \_string:

constants.add\_row(([token, 'const string', line, position]))

continue

raise LexicalError(f'{token} is unsupported!', line, position)

if show\_tables:

print(identifiers, '\n')

print(constants, '\n')

print(keywords, '\n')

print(operators, '\n')

return tokens

б) erorrs.py

class LexicalError(Exception):

def \_\_init\_\_(self, message, line, position):

super().\_\_init\_\_(f'LEXICAL ERROR | line: {line}, position: {position} | {message}')

class SyntaxError(Exception):

def \_\_init\_\_(self, message, line, position):

super().\_\_init\_\_(f'SYNTAX ERROR | line: {line}, position: {position} | {message}')

class SemanticsError(Exception):

def \_\_init\_\_(self, message, line, position):

super().\_\_init\_\_(f'SEMANTICS ERROR | line: {line}, position: {position} | {message}')

в) interpreter.py

from analyzer import Analyzer

class Interpreter(object):

def load(self, src\_path):

with open(src\_path, 'r') as f:

self.source = f.read()

self.analyzer = Analyzer(self.source)

def analyze(self):

try:

self.tokens = self.analyzer.analyze(show\_tables=True)

except Exception as e:

print(e)

г) patterns.py

TOKEN\_PATTERN = r'([a-zA-z]+\[\]|;|\d+\.\d+|\w+|\"\w\*\"|\(|\)|\{|\}|\[|\]|\+=+|-=+|\\*=+|/=+|%=+|\++|-+|\\*+|\/+|%+|!=+|<+|<=+|>+|>=+|=+|;|.)|(.+)"'

FLOAT\_PATTERN = r'^\d+|\d+\.\d+$'

INT\_PATTERN = r'^\d+$'

STRING\_PATTERN = r'^"[a-zA-Z]\*"$'

IDENTIFIER\_PATTERN = r'^[a-zA-Z]\w\*$'

IDENTIFIER\_DESC = 'Identifier of type'

keywords = {

'break': 'Flow control keyword',

'continue': 'Flow control keyword',

'while': 'Flow control keyword',

'for': 'Flow control keyword',

'if': 'Flow control keyword',

'else': 'Flow control keyword',

'return': 'Flow control keyword',

'(': 'Parenthesis',

')': 'Parenthesis',

'{': 'Parenthesis',

'}': 'Parenthesis',

'[': 'Parenthesis',

']': 'Parenthesis',

',': 'Comma separator',

';': 'End of instruction'

}

operators = {

'+': 'Arithmetic operator',

'-': 'Arithmetic operator',

'\*': 'Arithmetic operator',

'/': 'Arithmetic operator',

'%': 'Arithmetic operator',

'++': 'Arithmetic operator',

'--': 'Arithmetic operator',

'+=': 'Arithmetic operator',

'-=': 'Arithmetic operator',

'\*=': 'Arithmetic operator',

'/=': 'Arithmetic operator',

'%=': 'Arithmetic operator',

'=': 'Assignment operator',

'print': 'STD function',

'pow': 'STD function',

'sqrt': 'STD function',

'read': 'STD function',

'readint': 'STD function',

'readfloat': 'STD function',

'==': 'Comparison operator',

'!=': 'Comparison operator',

'<': 'Comparison operator',

'>': 'Comparison operator',

'<=': 'Comparison operator',

'>=': 'Comparison operator'

}

data\_types = {

'string': 'Data type',

'float': 'Data type',

'int': 'Data type',

'string[]': 'Data type',

'float[]': 'Data type',

'float[]': 'Data type'

}

д) main.py

from interpreter import Interpreter

interpreter = Interpreter()

interpreter.load('/Users/mariazyryanova/Desktop/BSUIR/3 course/MTran/files/input.test')

interpreter.analyze()