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**

Студент Е. В. Ласевич

Преподаватель Н. Ю. Гриценко

Минск 2024

**СОДЕРЖАНИЕ**

[1 Формулировка задачи 3](#_Toc158641914)

[2 Теоретические сведения 4](#_Toc158641915)

[3 Описание работы программы 5](#_Toc158641916)

[Список использованных источников 8](#_Toc158641917)

[Приложение А (обязательное) Исходный код программы 9](#_Toc158641918)

1 ФОРМУЛИРОВКА ЗАДАЧИ

Целью выполнения лабораторной работы является разработка лексического анализатора подмножества языка программирования, определенного в лабораторной работе 1.

В качестве задачи необходимо определить лексические правила, выполнить перевод потока символов в поток лексем, а также показать скриншоты нахождения четырех лексических ошибок разработанным анализатором.

2 ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Лексический анализатор – это компонент компилятора, который выполняет первый этап процесса компиляции, а именно, разбивает исходный код программы на лексемы или токены.

Лексема (лексическая единица языка) – это 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. Состав возможных лексем каждого конкретного языка программирования определяется синтаксисом этого языка. [1]

Цель лексического анализатора заключается в том, чтобы преобразовать исходный код программы в последовательность лексем, которые затем будут использоваться в следующих этапах компиляции, таких как синтаксический анализ и семантический анализ.

Лексический анализатор выполняет следующие задачи:

1 Разбиение исходного кода на лексемы. Лексический анализатор сканирует исходный код программы и выделяет последовательность лексем. Он использует набор правил, называемых лексическими правилами или регулярными выражениями, для определения, какие символы образуют лексемы и как их классифицировать.

2 Игнорирование незначащих символов. Лексический анализатор игнорирует незначащие символы, такие как пробелы, табуляции и комментарии. Он пропускает эти символы и сосредотачивается только на значимых символах, которые образуют лексемы.

3 Проверка на корректность лексем. Лексический анализатор также проверяет, что каждая лексема соответствует правилам языка программирования. Если встречается некорректная лексема, то генерируется ошибка, которая может быть обработана в следующих этапах компиляции.

Таким образом, лексический анализатор играет важную роль в процессе компиляции, обеспечивая правильное разбиение исходного кода на лексемы, которые затем будут использоваться для дальнейшего анализа и преобразования программы. [2]

3 ОПИСАНИЕ РАБОТЫ ПРОГРАММЫ

В ходе лабораторной работы был разработан лексический анализатор языка *Kotlin* на языке программирования *Python*. При запуске кода скрипт считывает данные из файла, который передаётся в качестве параметра, после чего на экран выведется информация, полученная в ходе анализа.

В случае, если программа не нашла лексические ошибки в консоль выводится таблица имен. Пример этой таблицы представлен на рисунке 3.1.

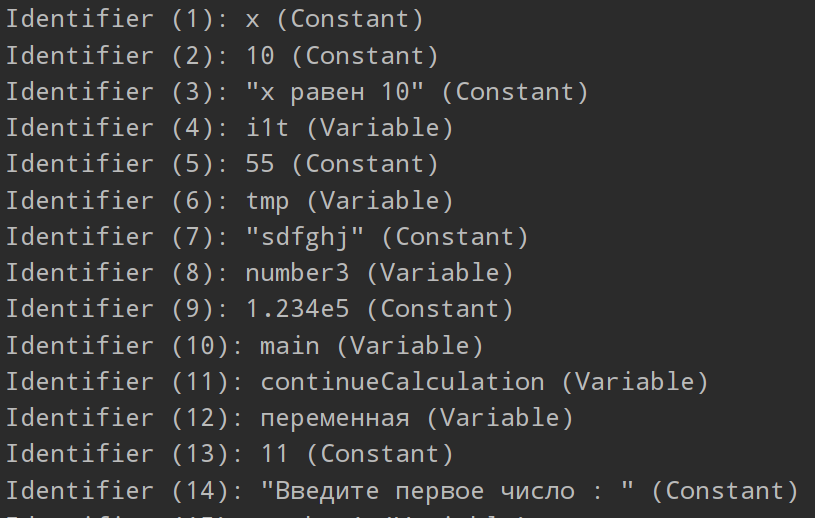


Рисунок 3.1 – Таблица имен

В случае же нахождения ошибки в консоль будет выведена информация об этой ошибке с указанием номера строки анализируемого кода, в которой она была найдена. Пример лексических ошибок при объявлении переменной изображен на рисунке 3.2.

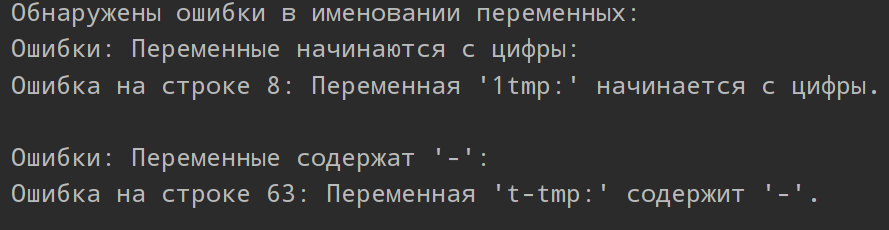


Рисунок 3.2 – Ошибки объявления переменных

Пример обработки ошибки, в которой не закрыты кавычки строки изображен на рисунке 3.3



Рисунок 3.3 – Ошибка не закрытые кавычки

Пример обработки ошибок, где используются не корректные числа изображен на рисунке 3.4.

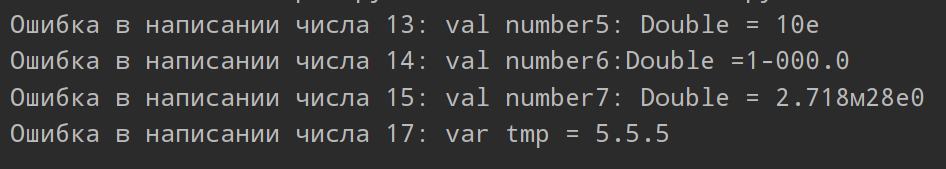


Рисунок 3.4 – Ошибки написания чисел

Пример обработки ошибки, в которой есть опечатка в ключевом слове изображен на рисунке 3.5.



Рисунок 3.5 – Ошибка опечатка в ключевом слове

Пример обработки ошибки, неправильного использования оператора изображен на рисунке 3.6

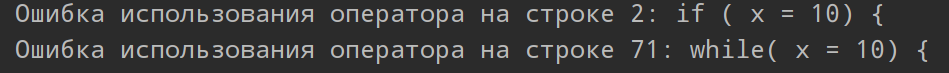


Рисунок 3.6 – Неверное использование оператора

Таким образом в ходе лабораторной работы был разработан лексический анализатор, который способен выводить в консоль таблицу имен, а также найденные лексические ошибки.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Назначение лексического анализатора [Электронный ресурс]. – Электронные данные. – Режим доступа: https://it.wireading.ru/hIzAVYomlf.

[2] [Электронный ресурс]. – Электронные данные. – Режим доступа: https://nauchniestati.ru/spravka/teoriya-kompilyaczii-leksicheskij-analizator/.

ПРИЛОЖЕНИЕ А

(обязательное)

Исходный код программы

**Файл main.py:**

import sys  
import check\_numbers  
import check\_keywords  
import check\_operators  
import check\_strings  
from check\_variables import \*  
from fix\_file import \*  
  
  
class Lexer:  
 def \_\_init\_\_(self):  
 self.keywords = {  
 "control": {"if", "else", "for", "while", "do", "return", "continue", "break", "when", "in", "is", "as",  
 "println", "readLine"},  
 "modifiers": {"var", "val", "fun", "package", "import"},  
 "types": {"Boolean", "Byte", "Char", "Double", "Float", "Int", "Long", "Short", "String", "Any", "Unit"},  
 "miscellaneous": {"true", "false", "null", "throw", "try", "catch", "finally", "object", "typealias"}  
 }  
 self.operators = {  
 "arithmetic": {"+", "-", "\*", "/", "%"},  
 "assignment": {"=", "+=", "-=", "\*=", "/=", "%=", "&=", "|=", "^=", "<<=", ">>="},  
 "comparison": {"==", "!=", "<", ">", "<=", ">="},  
 "logical": {"&&", "||", "!"},  
 "bitwise": {"&", "|", "^", "~", "<<", ">>"},  
 "ternary": {"?:", "?."},  
 "other": {"::", "->", "..", "!!", "@"}  
 }  
 self.symbols = {'(', ')', '{', '}', ':', ',', ';', '.', '[', ']', '?', '->', '?:', '?'}  
  
 def analyze(self, input\_code):  
 token\_table = []  
 identifier\_table = []  
 keyword\_table = set()  
 operator\_table = set()  
 in\_string = False  
 id\_counter = 1  
 identifiers = {} # Словарь для отслеживания идентификаторов и их типов  
 for line in input\_code:  
 tokens = []  
 current\_token = ''  
 for char in line.strip():  
 if char == '"':  
 in\_string = not in\_string  
 if in\_string:  
 current\_token += char  
 else:  
 current\_token += char  
 tokens.append(current\_token)  
 current\_token = ''  
 elif char == ' ' and not in\_string:  
 if current\_token:  
 tokens.append(current\_token)  
 current\_token = ''  
 elif char in {'(', ')'}:  
 if current\_token:  
 tokens.append(current\_token)  
 current\_token = ''  
 tokens.append(char)  
 else:  
 current\_token += char  
 if current\_token:  
 tokens.append(current\_token)  
  
 processed\_tokens, id\_counter = self.process\_tokens(tokens, in\_string, id\_counter, identifiers)  
 token\_table.extend(processed\_tokens)  
 identifier\_table.extend(  
 [(token\_type, token\_value, token\_info) for token\_type, token\_value, token\_info in processed\_tokens if  
 token\_type == "Identifier"])  
  
 # Заполнение таблиц ключевых слов и операторов  
 for token\_type, token\_value, token\_info in processed\_tokens:  
 if token\_type.startswith("Keyword"):  
 keyword\_table.add(token\_value)  
 elif token\_type.startswith("Operator"):  
 operator\_table.add(token\_value)  
  
 return token\_table, identifier\_table, keyword\_table, operator\_table  
  
 def process\_tokens(self, tokens, in\_string, id\_counter, identifiers):  
 processed\_tokens = []  
 is\_variable\_declaration = False # Флаг для отслеживания объявления переменной  
  
 for token in tokens:  
 if in\_string:  
 processed\_tokens.append(("String", token, ""))  
 elif token in self.keywords["control"]:  
 processed\_tokens.append(("Keyword (control)", token, ""))  
 is\_variable\_declaration = False # Сбрасываем флаг при встрече ключевого слова  
 elif token in self.keywords["modifiers"]:  
 processed\_tokens.append(("Keyword (modifiers)", token, ""))  
 is\_variable\_declaration = True # Устанавливаем флаг при встрече модификатора  
 elif token in self.keywords["types"]:  
 processed\_tokens.append(("Keyword (types)", token, ""))  
 elif token in self.keywords["miscellaneous"]:  
 processed\_tokens.append(("Keyword (miscellaneous)", token, ""))  
 is\_variable\_declaration = False # Сбрасываем флаг при встрече ключевого слова  
 elif token in self.operators["arithmetic"]:  
 processed\_tokens.append(("Operator (arithmetic)", token, ""))  
 is\_variable\_declaration = False # Сбрасываем флаг при встрече оператора  
 elif token in self.operators["assignment"]:  
 processed\_tokens.append(("Operator (assignment)", token, ""))  
 is\_variable\_declaration = False # Сбрасываем флаг при встрече оператора  
 elif token in self.operators["comparison"]:  
 processed\_tokens.append(("Operator (comparison)", token, ""))  
 is\_variable\_declaration = False # Сбрасываем флаг при встрече оператора  
 elif token in self.operators["logical"]:  
 processed\_tokens.append(("Operator (logical)", token, ""))  
 is\_variable\_declaration = False # Сбрасываем флаг при встрече оператора  
 elif token in self.operators["bitwise"]:  
 processed\_tokens.append(("Operator (bitwise)", token, ""))  
 is\_variable\_declaration = False # Сбрасываем флаг при встрече оператора  
 elif token in self.operators["ternary"]:  
 processed\_tokens.append(("Operator (ternary)", token, ""))  
 is\_variable\_declaration = False # Сбрасываем флаг при встрече оператора  
 elif token in self.operators["other"]:  
 processed\_tokens.append(("Operator (other)", token, ""))  
 is\_variable\_declaration = False # Сбрасываем флаг при встрече оператора  
 elif token in self.symbols:  
 processed\_tokens.append(("Symbol", token, ""))  
 is\_variable\_declaration = False # Сбрасываем флаг при встрече символа  
 else:  
 # Если это идентификатор, проверяем, был ли он уже встречен ранее  
 if token in identifiers:  
 # Если да, используем тот же тип  
 processed\_tokens.append((identifiers[token], token, ""))  
 else:  
 # Иначе, добавляем новый идентификатор и определяем его тип  
 if is\_variable\_declaration:  
 # Если идентификатор в контексте объявления переменной, он является переменной  
 identifiers[token] = "Identifier"  
 token\_info = "Variable"  
 else:  
 # Иначе он является константой  
 identifiers[token] = "Constant"  
 token\_info = "Constant"  
 processed\_tokens.append(("Identifier", token, token\_info))  
 id\_counter += 1  
  
 return processed\_tokens, id\_counter  
  
  
def print\_table(title, data):  
 print(f"{title}:")  
 for index, item in enumerate(data, 1):  
 print(f"{index}: {item}")  
  
  
def print\_tokens(token\_table):  
 for index, token\_info in enumerate(token\_table, 1):  
 print(f"Token {index}: {token\_info}")  
  
  
from collections import OrderedDict  
  
  
def print\_identifiers(identifier\_table):  
 unique\_identifiers = OrderedDict()  
  
 for index, (token\_type, token\_value, token\_info) in enumerate(identifier\_table, 1):  
 if token\_value not in unique\_identifiers:  
 unique\_identifiers[token\_value] = token\_info  
  
 for index, (identifier, info) in enumerate(unique\_identifiers.items(), 1):  
 print(f"Identifier ({index}): {identifier} ({info})")  
  
  
def main():  
 lexer = Lexer()  
 filename = "corrected\_test.kt"  
 with open(filename, 'r', encoding="UTF-8") as file:  
 input\_code = file.readlines()  
 token\_table, identifier\_table, keyword\_table, operator\_table = lexer.analyze(input\_code)  
  
 print("All tokens:")  
 print\_tokens(token\_table)  
 # print\_table("Identifier Table", identifier\_table)  
 print\_identifiers(identifier\_table)  
 print()  
 print\_table("Keyword Table", keyword\_table)  
 print()  
 print\_table("Operator Table", operator\_table)  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 filename = "test.kt"  
 has\_errors, digit\_start\_errors, hyphen\_errors = check\_variable\_naming\_errors(filename)  
 print\_errors(has\_errors, digit\_start\_errors, hyphen\_errors)  
 if check\_strings.check\_quotes(filename):  
 sys.exit(0)  
 if has\_errors:  
 sys.exit(0)  
 with open(filename, 'r', encoding="UTF-8") as file:  
 lines = file.readlines()  
 has\_errors = check\_numbers.check\_numbers\_in\_lines(lines)  
 if has\_errors:  
 sys.exit(0)  
 fixed\_file()  
 filename = "corrected\_test.kt"  
 code = check\_keywords.read\_code\_from\_file(filename)  
 error\_type = "опечатка в ключевом слове"  
 error = check\_keywords.find\_typo\_in\_keywords(code)  
 if check\_keywords.print\_error(error\_type, error):  
 sys.exit(0)  
 if check\_operators.check\_lexical\_errors(filename):  
 sys.exit(0)  
  
 main()

**Файл check\_keywords.py:**

import sys  
def read\_code\_from\_file(filename):  
 with open(filename, 'r') as file:  
 code = file.readlines()  
 return code  
  
def find\_typo\_in\_keywords(code):  
 keywords = ['if', 'else', 'while', 'for', 'when', 'return', 'fun', 'class', 'interface', 'object', 'package', 'import', 'val', 'var']  
 for line\_number, line in enumerate(code, 1):  
 words = line.split()  
 for word in words:  
 if word not in keywords:  
 min\_diff = float('inf')  
 closest\_keyword = None  
 for keyword in keywords:  
 if len(word) == len(keyword):  
 diff\_count = sum(c1 != c2 for c1, c2 in zip(word, keyword))  
 if diff\_count < min\_diff:  
 min\_diff = diff\_count  
 closest\_keyword = keyword  
 if min\_diff == 1:  
 return line\_number, word, closest\_keyword  
 return None  
  
def print\_error(error\_type, error):  
 if error:  
 line\_number, word, closest\_keyword = error  
 print("Обнаружена лексическая ошибка типа '{}' в строке {}: {}, ближайшее ключевое слово: {}".format(error\_type, line\_number, word, closest\_keyword))  
 sys.exit(0)

**Файл check\_variables.py:**

def check\_variable\_naming\_errors(filename):  
 has\_errors = False  
 digit\_start\_errors = []  
 hyphen\_errors = []  
 with open(filename, 'r') as file:  
 lines = file.readlines()  
 for i, line in enumerate(lines):  
 words = line.split()  
 for j, word in enumerate(words):  
 if word in ['var', 'val'] and j < len(words) - 1:  
 next\_word = words[j + 1]  
 if next\_word[0].isdigit():  
 has\_errors = True  
 digit\_start\_errors.append((i + 1, next\_word))  
 elif '-' in next\_word:  
 has\_errors = True  
 hyphen\_errors.append((i + 1, next\_word))  
 return has\_errors, digit\_start\_errors, hyphen\_errors  
  
  
def print\_errors(has\_errors, digit\_start\_errors, hyphen\_errors):  
 if has\_errors:  
 print("Обнаружены ошибки в именовании переменных:")  
 if digit\_start\_errors:  
 print("Ошибки: Переменные начинаются с цифры:")  
 for error in digit\_start\_errors:  
 print(f"Ошибка на строке {error[0]}: Переменная '{error[1]}' начинается с цифры.")  
 if hyphen\_errors:  
 print("\nОшибки: Переменные содержат '-':")  
 for error in hyphen\_errors:  
 print(f"Ошибка на строке {error[0]}: Переменная '{error[1]}' содержит '-'.")  
  
 else:  
 pass

**Файл check\_strings.py:**

def read\_code\_from\_file(filename):  
 with open(filename, 'r') as file:  
 code = file.readlines()  
 return code  
  
def are\_quotes\_closed(line):  
 stack = []  
 for i, char in enumerate(line):  
 if char == '"':  
 if stack and stack[-1][0] == '"':  
 stack.pop()  
 else:  
 stack.append(('"', i))  
 elif char == "'":  
 if stack and stack[-1][0] == "'":  
 stack.pop()  
 else:  
 stack.append(("'", i))  
 if stack:  
 return False, stack[-1][1] # Возвращаем False и позицию символа в строке  
 return True, None  
  
def check\_quotes(filename):  
 code = read\_code\_from\_file(filename)  
 has\_errors = False  
 for line\_number, line in enumerate(code, 1):  
 result, error\_position = are\_quotes\_closed(line)  
 if not result:  
 print("Ошибка: Незакрытые кавычки в строке {} файла {}".format(line\_number, filename))  
 has\_errors = True  
 break  
 if not has\_errors:  
 pass  
 return has\_errors