Министерство образования Республики Беларусь

Учреждение образования «Белорусский государственный университет информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: Методы трансляции

ОТЧЁТ

по лабораторной работе

на тему

Лексический анализ

Выполнил

Студент гр. 053501

Хиль В.М.

Проверил

Ассистент кафедры информатики

Гриценко Н.Ю.

Минск 2023

**СОДЕРЖАНИЕ**

[1 Цель работы 3](#_Toc131854195)

[2 Краткие теоретические сведения 4](#_Toc131854196)

[3 Анализ возможных ошибок 5](#_Toc131854197)

[4 Демонстрация работы 6](#_Toc131854198)

[4.1 Результаты работы 6](#_Toc131854199)

[4.2 Лексические ошибки 9](#_Toc131854200)

[Приложение А (обязательное) Код программы 12](#_Toc131854201)

[Приложение Б (обязательное) Константы для обнаружения токенов 16](#_Toc131854202)

[Приложение В (обязательное) Классы для обнаружения ошибок 17](#_Toc131854203)

**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).

1. **КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

Лексический анализ – это первый этап в теории трансляции, на котором исходный код программы преобразуется в последовательность токенов. Токены – это независимые элементы программы, такие как идентификаторы, ключевые слова, символы операций и т.д.

Цель лексического анализа – разбить исходный код на единицы информации, которые можно использовать для дальнейшей обработки. Для этого используется лексер, который сканирует исходный код и разбивает его на токены.

Результатом работы лексического анализа является последовательность токенов, которая подается на вход для дальнейшей обработки, такой как синтаксический анализ. Корректность лексического анализа определяет, насколько хорошо программа может быть обработана дальше. Если лексер обнаруживает ошибку, такую как недопустимый символ или неизвестный идентификатор, он генерирует ошибку лексического анализа.

Одним из важных аспектов лексического анализа является его эффективность. Лексер должен работать быстро, так как этот этап является одним из самых длительных в процессе компиляции. Поэтому разработчики обычно используют алгоритмы, такие как автоматы или грамматические анализаторы, чтобы улучшить эффективность лексического анализа.

В целом, лексический анализ играет ключевую роль в теории трансляции, поскольку он позволяет преобразовать исходный код в формат, который может быть легко обработан дальше. Корректный и эффективный лексический анализ также позволяет сохранять сведения о токенах и их атрибутах, что может быть полезно для дальнейшей обработки и отладки кода.

1. **АНАЛИЗ ВОЗМОЖНЫХ ОШИБОК**

Я рассмотрел лексические ошибки следующих видов:

* Ошибки, при написании специальных символов, которых нет подмножестве рассматриваемого языка: ‘~’, ‘%’, ‘@’, ‘&’, ‘{’, ‘}’.
* Ошибки, когда после буквы латинского алфавита идёт точка ‘.’.
* Ошибки, когда используется любой алфавит, кроме латинского. Например, кириллица.
* Ошибки, когда используется две точки во float-значениях.

1. **ДЕМОНСТРАЦИЯ РАБОТЫ**
   1. **Результаты работы**

На рисунке 1 приведен код, который содержит условную конструкцию if, цикл for и синтаксис объявления функции.

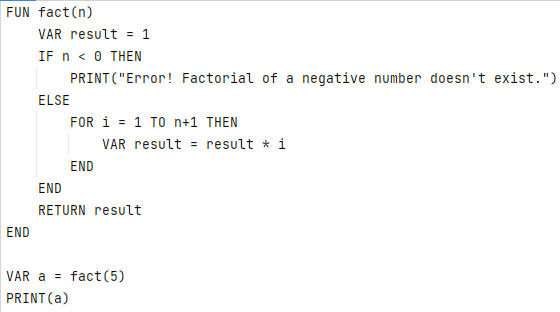


Рисунок 1 – Код на BASIC

На рисунке 2 приведен результат работы лексического анализатора.

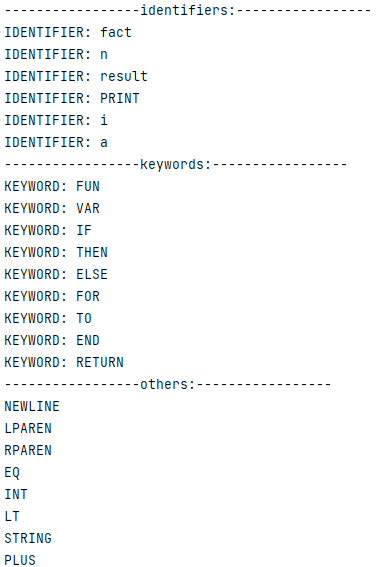


Рисунок 2 – Результат работы лексического анализатора

На рисунке 3 приведен код, который содержит цикл while и синтаксис объявления функции.

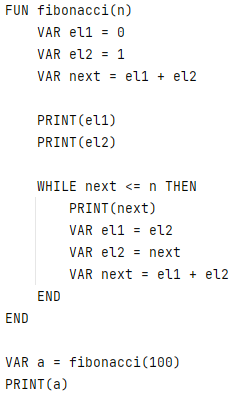


Рисунок 3 – Код на BASIC

На рисунке 4 приведен результат работы лексического анализатора.

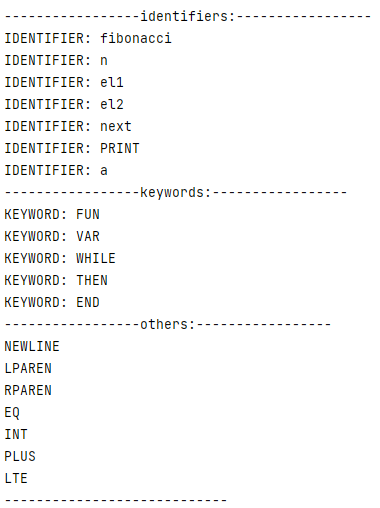


Рисунок 4 – Результат работы лексического анализатора

* 1. **Лексические ошибки**

На рисунке 5 приведен код, где допущена ошибка, при добавлении в код специального символа.

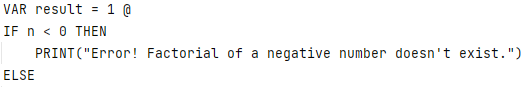


Рисунок 5 – Код на BASIC

На рисунке 6 приведен результат работы лексического анализатора. Он выдал ошибку и указал, что не так.

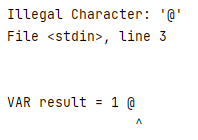


Рисунок 6 – Результат работы лексического анализатора

На рисунке 7 приведен код, где допущен второй тип ошибки.

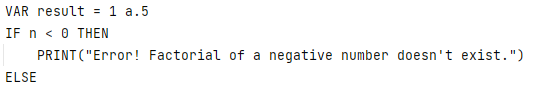


Рисунок 7 – Код на BASIC

На рисунке 8 приведен результат работы лексического анализатора. Он выдал ошибку и указал, что не так.

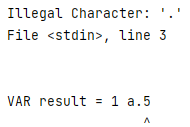


Рисунок 8 - Результат работы лексического анализатора

На рисунке 9 приведен код, где используется кириллица.

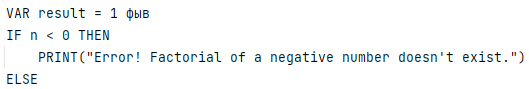


Рисунок 9 – Код на BASIC

На рисунке 10 приведен результат работы лексического анализатора. Он выдал ошибку и указал, что не так.

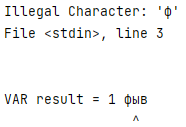


Рисунок 10 – Результат работы лексического анализатора

На рисунке 11 приведен код, где есть четвёртый тип ошибки.

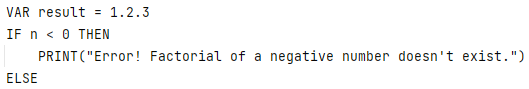


Рисунок 11– Код на BASIC

На рисунке 12 приведен результат работы лексического анализатора. Он выдал ошибку и указал, что не так.

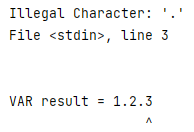


Рисунок 12 - Результат работы лексического анализатора

# ПРИЛОЖЕНИЕ А

**(обязательное)**

**Код программы**

class Lexer:  
 def \_\_init\_\_(self, function, text):  
 self.function = function  
 self.text = text  
 self.position = Position(-1, 0, -1, function, text)  
 self.current\_character = None  
 self.next\_character()  
  
 def next\_character(self):  
 self.position.next\_character(self.current\_character)  
 if self.position.index < len(self.text):  
 self.current\_character = self.text[self.position.index]  
 else:  
 self.current\_character = None  
  
 def make\_tokens(self):  
 tokens = []  
  
 keyword = []  
 identifier = []  
 others = []  
  
 while self.current\_character != None:  
 if self.current\_character in ' \t':  
 self.next\_character()  
 elif self.current\_character == '(':  
 tokens.append(Token(LPAREN, start\_position=self.position))  
 self.next\_character()  
 elif self.current\_character == ')':  
 tokens.append(Token(RPAREN, start\_position=self.position))  
 self.next\_character()  
 elif self.current\_character == '[':  
 tokens.append(Token(LSQUARE, start\_position=self.position))  
 self.next\_character()  
 elif self.current\_character == ']':  
 tokens.append(Token(RSQUARE, start\_position=self.position))  
 self.next\_character()  
 elif self.current\_character == '!':  
 token, error = self.make\_token\_not\_equals()  
 if error:  
 return [], error  
 tokens.append(token)  
 elif self.current\_character == '#':  
 self.skip\_comment()  
 elif self.current\_character in ';\n':  
 tokens.append(Token(NEWLINE, start\_position=self.position))  
 self.next\_character()  
 elif self.current\_character == '^':  
 tokens.append(Token(POW, start\_position=self.position))  
 self.next\_character()  
 elif self.current\_character == '=':  
 tokens.append(self.make\_token\_equals())  
 elif self.current\_character == '<':  
 tokens.append(self.make\_token\_less\_than())  
 elif self.current\_character == '>':  
 tokens.append(self.make\_token\_greater\_than())  
 elif self.current\_character == ',':  
 tokens.append(Token(COMMA, start\_position=self.position))  
 self.next\_character()  
 elif self.current\_character in DIGITS:  
 tokens.append(self.make\_token\_number())  
 elif self.current\_character in LETTERS:  
 tokens.append(self.make\_token\_identifier())  
 elif self.current\_character == '"':  
 tokens.append(self.make\_token\_string())  
 elif self.current\_character == '+':  
 tokens.append(Token(PLUS, start\_position=self.position))  
 self.next\_character()  
 elif self.current\_character == '-':  
 tokens.append(self.make\_token\_minus\_or\_arrow())  
 elif self.current\_character == '\*':  
 tokens.append(Token(MUL, start\_position=self.position))  
 self.next\_character()  
 elif self.current\_character == '/':  
 tokens.append(Token(DIV, start\_position=self.position))  
 self.next\_character()  
 else:  
 start\_position = self.position.copy\_position()  
 char = self.current\_character  
 self.next\_character()  
 return [], IllegalCharError(start\_position, self.position, "'" + char + "'")  
  
 for i in tokens:  
 if i.type == "IDENTIFIER":  
 identifier.append(i)  
 elif i.type == "KEYWORD":  
 keyword.append(i)  
 else:  
 others.append(i)  
  
 tokens.append(Token(EOF, start\_position=self.position))  
  
 if len(identifier) > 0:  
 if len(keyword) > 0:  
 if len(others) > 0:  
 return tokens, identifier, keyword, others, None  
 else:  
 return tokens, None  
  
 def make\_token\_identifier(self):  
 id\_str = ''  
 start\_position = self.position.copy\_position()  
  
 while self.current\_character != None and self.current\_character in LETTERS\_DIGITS + '\_':  
 id\_str += self.current\_character  
 self.next\_character()  
  
 if id\_str in KEYWORDS:  
 token\_type = KEYWORD  
 else:  
 token\_type = IDENTIFIER  
 return Token(token\_type, id\_str, start\_position, self.position)  
  
 def make\_token\_minus\_or\_arrow(self):  
 token\_type = MINUS  
 start\_position = self.position.copy\_position()  
 self.next\_character()  
  
 if self.current\_character == '>':  
 self.next\_character()  
 token\_type = ARROW  
  
 return Token(token\_type, start\_position=start\_position, end\_position=self.position)  
  
 def make\_token\_equals(self):  
 token\_type = EQ  
 start\_position = self.position.copy\_position()  
 self.next\_character()  
  
 if self.current\_character == '=':  
 self.next\_character()  
 token\_type = EE  
  
 return Token(token\_type, start\_position=start\_position, end\_position=self.position)  
  
 def make\_token\_not\_equals(self):  
 start\_position = self.position.copy\_position()  
 self.next\_character()  
  
 if self.current\_character == '=':  
 self.next\_character()  
 return Token(NE, start\_position=start\_position, end\_position=self.position), None  
  
 self.next\_character()  
 return None, ExpectedCharError(start\_position, self.position, "'=' (after '!')")  
  
 def make\_token\_greater\_than(self):  
 token\_type = GT  
 start\_position = self.position.copy\_position()  
 self.next\_character()  
  
 if self.current\_character == '=':  
 self.next\_character()  
 token\_type = GTE  
  
 return Token(token\_type, start\_position=start\_position, end\_position=self.position)  
  
 def make\_token\_less\_than(self):  
 token\_type = LT  
 start\_position = self.position.copy\_position()  
 self.next\_character()  
  
 if self.current\_character == '=':  
 self.next\_character()  
 token\_type = LTE  
  
 return Token(token\_type, start\_position=start\_position, end\_position=self.position)  
  
 def skip\_comment(self):  
 self.next\_character()  
  
 while self.current\_character != '\n':  
 self.next\_character()  
  
 self.next\_character()  
  
 def make\_token\_number(self):  
 number\_string = ''  
 dot\_count = 0  
 start\_position = self.position.copy\_position()  
  
 while self.current\_character != None and self.current\_character in DIGITS + '.':  
 if self.current\_character == '.':  
 if dot\_count == 1:  
 break  
 dot\_count += 1  
 number\_string += self.current\_character  
 self.next\_character()  
  
 if dot\_count == 0:  
 return Token(INT, int(number\_string), start\_position, self.position)  
 else:  
 return Token(FLOAT, float(number\_string), start\_position, self.position)  
  
 def make\_token\_string(self):  
 string = ''  
 start\_position = self.position.copy\_position()  
 escape\_character = False  
 self.next\_character()  
  
 escape\_characters = {  
 'n': '\n',  
 't': '\t'  
 }  
  
 while self.current\_character != None and (self.current\_character != '"' or escape\_character):  
 if escape\_character:  
 string += escape\_characters.get(self.current\_character, self.current\_character)  
 else:  
 if self.current\_character == '\\':  
 escape\_character = True  
 else:  
 string += self.current\_character  
 self.next\_character()  
 escape\_character = False  
  
 self.next\_character()  
 return Token(STRING, string, start\_position, self.position)

# ПРИЛОЖЕНИЕ Б

**(обязательное)**

**Константы для обнаружения токенов**

DIGITS = '0123456789'  
LETTERS = string.ascii\_letters  
LETTERS\_DIGITS = LETTERS + DIGITS  
  
PLUS = 'PLUS'  
MINUS = 'MINUS'  
MUL = 'MUL'  
DIV = 'DIV'  
POW = 'POW'  
EQ = 'EQ'  
LPAREN = 'LPAREN'  
RPAREN = 'RPAREN'  
LSQUARE = 'LSQUARE'  
RSQUARE = 'RSQUARE'  
INT = 'INT'  
FLOAT = 'FLOAT'  
STRING = 'STRING'  
IDENTIFIER = 'IDENTIFIER'  
KEYWORD = 'KEYWORD'  
EE = 'EE'  
NE = 'NE'  
LT = 'LT'  
GT = 'GT'  
LTE = 'LTE'  
GTE = 'GTE'  
COMMA = 'COMMA'  
ARROW = 'ARROW'  
NEWLINE = 'NEWLINE'  
EOF = 'EOF'  
  
KEYWORDS = [  
 'FOR',  
 'TO',  
 'STEP',  
 'WHILE',  
 'FUN',  
 'THEN',  
 'END',  
 'RETURN',  
 'CONTINUE',  
 'BREAK',  
 'VAR',  
 'AND',  
 'OR',  
 'NOT',  
 'IF',  
 'ELIF',  
 'ELSE'  
]

# ПРИЛОЖЕНИЕ В

**(обязательное)**

**Классы для обнаружения ошибок**

from strings\_with\_error import arrow\_string  
  
  
class Error:  
 def \_\_init\_\_(self, start\_position, end\_position, error\_name, details):  
 self.start\_position = start\_position  
 self.end\_position = end\_position  
 self.error\_name = error\_name  
 self.details = details  
  
 def string\_representation(self):  
 result = f'{self.error\_name}: {self.details}\n'  
 result += f'File {self.start\_position.function}, line {self.start\_position.line + 1}'  
 result += '\n\n' + arrow\_string(self.start\_position.function\_text, self.start\_position, self.end\_position)  
 return result  
  
  
class IllegalCharError(Error):  
 def \_\_init\_\_(self, start\_position, end\_position, details):  
 super().\_\_init\_\_(start\_position, end\_position, 'Illegal Character', details)  
  
  
class ExpectedCharError(Error):  
 def \_\_init\_\_(self, start\_position, end\_position, details):  
 super().\_\_init\_\_(start\_position, end\_position, 'Expected Character', details)