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

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 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.

3 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.

4 ДЕМОНСТРАЦИЯ РАБОТЫ

4.1 Результаты работы

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

```
FUN fact(n)
    VAR result = 1
    IF n < 0 THEN
        PRINT("Error! Factorial of a negative number doesn't exist.")
    ELSE
        FOR i = 1 TO n+1 THEN
            VAR result = result * i
        END
    END
    RETURN result
END

VAR a = fact(5)
PRINT(a)
```

Рисунок 1 – Код на BASIC

На рисунке 2 приведен результат работы лексического анализатора.

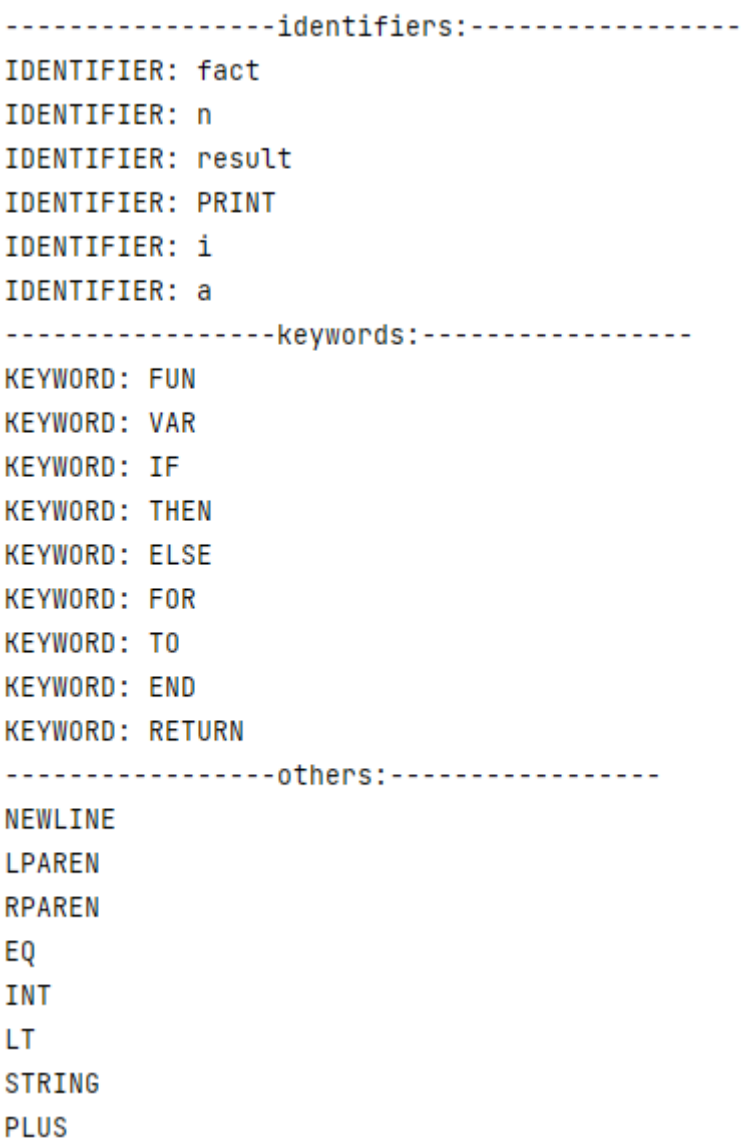

Рисунок 2 – Результат работы лексического анализатора

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

```
FUN fibonacci(n)
    VAR el1 = 0
    VAR el2 = 1
    VAR next = el1 + el2

    PRINT(el1)
    PRINT(el2)

    WHILE next <= n THEN
        PRINT(next)
        VAR el1 = el2
        VAR el2 = next
        VAR next = el1 + el2
    END
END

VAR a = fibonacci(100)
PRINT(a)
```

Рисунок 3 – Код на BASIC

На рисунке 4 приведен результат работы лексического анализатора.

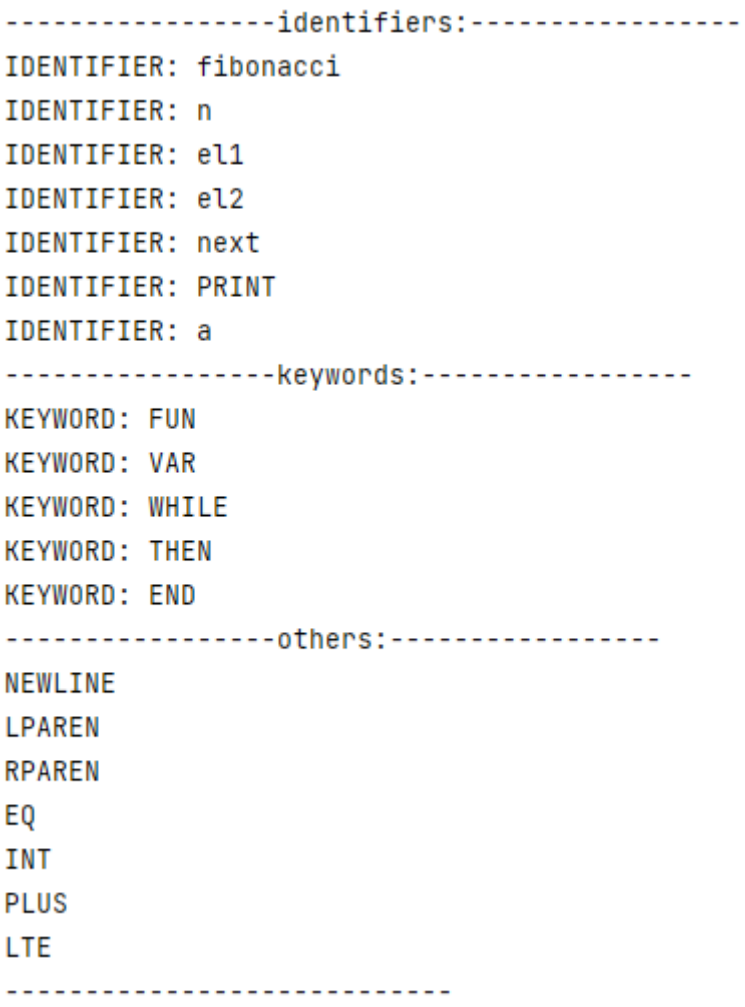

Рисунок 4 – Результат работы лексического анализатора

4.2 Лексические ошибки

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

```
VAR result = 1 @
IF n < 0 THEN
    PRINT("Error! Factorial of a negative number doesn't exist.")
ELSE
```

Рисунок 5 – Код на BASIC

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

```
Illegal Character: '@'
File <stdin>, line 3


VAR result = 1 @
                 ^
```

Рисунок 6 – Результат работы лексического анализатора

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

```
VAR result = 1 a.5
IF n < 0 THEN
    PRINT("Error! Factorial of a negative number doesn't exist.")
ELSE
```

Рисунок 7 – Код на BASIC

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

```
Illegal Character: '.'
File <stdin>, line 3


VAR result = 1 a.5
                 ^
```

Рисунок 8 - Результат работы лексического анализатора

На рисунке 9 приведен код, где используется кириллица.

```
VAR result = 1 фыв
IF n < 0 THEN
    PRINT("Error! Factorial of a negative number doesn't exist.")
ELSE
```

Рисунок 9 – Код на BASIC

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

```
Illegal Character: 'ф'
File <stdin>, line 3


VAR result = 1 фыв
                 ^
```

Рисунок 10 – Результат работы лексического анализатора

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

```
VAR result = 1.2.3
IF n < 0 THEN
    PRINT("Error! Factorial of a negative number doesn't exist.")
ELSE
```

Рисунок 11– Код на BASIC

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

```
Illegal Character: '.'
File <stdin>, line 3


VAR result = 1.2.3
                 ^
```

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)
```