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»

Выполнил А. А. Сивый

Проверил Н. Ю. Гриценко

Минск 2024

**СОДЕРЖАНИЕ**

1 Цели работы3

2 Результат работы программы4

3 Нахождение и локализация лексических ошибок программы5

Вывод6

Список использованных источников7

Приложение А (обязательное) Листинг программного кода8

**1 ЦЕЛИ РАБОТЫ**

1 Разработать лексический анализатор подмножества языка программирования, определенного в лабораторной работе 1.

2 Предусмотреть возможность определения неверной последовательности символов. Показать нахождение 4-х лексических ошибок.

**2 РЕЗУЛЬТАТ РАБОТЫ ПРОГРАММЫ**

В результате выполнения лабораторной работы была разработана программа, которая выполняет перевод потока символов в поток лексем (токенов).

На вход программе передадим файл, содержащий код программы 1 из лабораторной работы 1. Результат работы программы продемонстрирован на рисунке 2.1.

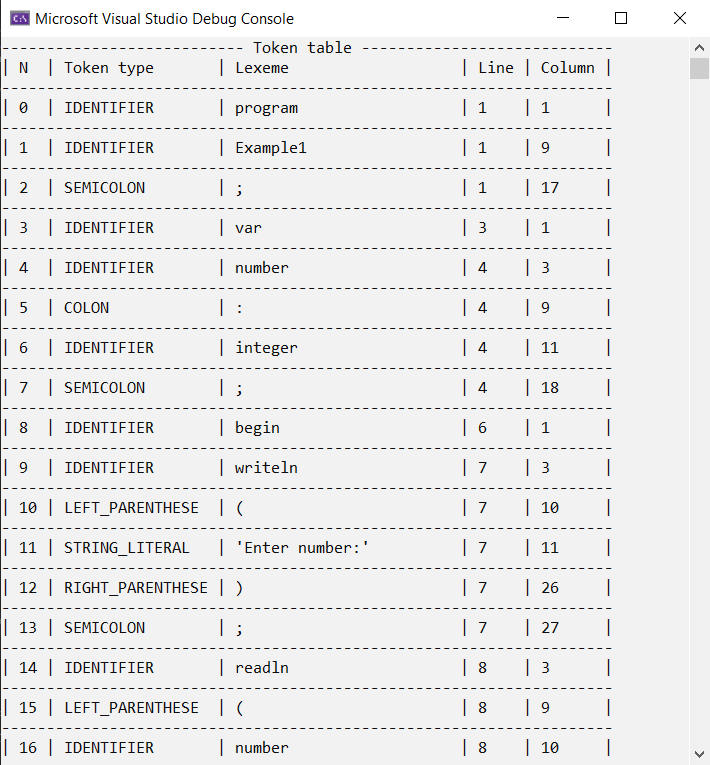


Рисунок 2.1 – Результат лексического анализа первой программы.

На выходе данной программы мы получаем набор токенов, представляющих идентификаторы, ключевые слова, операторы, разделители и т.д.

**3 НАХОЖДЕНИЕ И ЛОКАЛИЗАЦИЯ ЛЕКСИЧЕСКИХ**

**ОШИБОК ПРОГРАММЫ**

На этапе лексического анализа входной последовательности могут быть определены следующие типы ошибок:

1 Наличие символов, не предусмотренных языком программирования. Например, символ «#» не предусмотрен языком Pascal.

2 Неправильный формат литералов, например числовые литералы не должны содержать букв.

3 Незавершенные литералы или комментарии. Например, для многострочного комментария забыта закрывающая скобка.

4 Неправильные имена идентификаторов. Идентификаторы должны начинаться с буквы, могут содержать цифры и некоторые специальные символы.

Добавим в исходный текст программы непредусмотренный символ, изменим имя идентификатора, а также припишем в числовом литерале две точки. Результат работы программы продемонстрирован на рисунке 3.1.

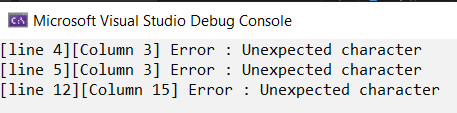


Рисунок 3.1 – Обнаруженные ошибки во время лексического анализа.

Добавим в текст программы незакрытый комментарий. Результат представлен на рисунке 3.2.

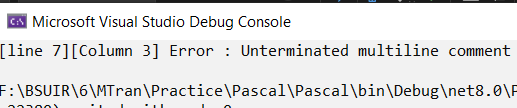


Рисунок 3.2 – Обнаружение незакрытого комментария.

Выявление ошибок на ранних стадиях играет важную роль в выявлении недочетов в программах, позволяя улучшить качество обработки данных.

**ВЫВОД**

Лексический анализ – это первый и важный этап обработки текста программы, который позволяет выделить лексемы (минимальные смысловые единицы) из исходного кода. На это этапе происходит разбиение текста на отдельные слова, числа, знаки препинания и другие элементы. Лексемы не зависят от контекста и могут быть использованы в разных частях текста. Результат работы лексического анализатора – список токенов, которые представляют собой более сложные структуры, образованные из лексем. Корректное выделение лексем обеспечивает успешную дальнейшую обработку текста программы.

**СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

[1] Crafting Interpreters [Электронный ресурс] – Режим доступа: <https://craftinginterpreters.com/> – Дата доступа: 15.02.2024

[2] Free Pascal Wiki [Электронный ресурс] – Режим доступа: [https://wiki.freepascal.org](https://wiki.freepascal.org/Operators) – Дата доступа: 16.02.2024

**ПРИЛОЖЕНИЕ А**

**(обязательное)**

**Листинг исходного кода**

Листинг 1 – Программный код разработанного приложения.

public List<Token> ScanTokens()

{

while (!IsAtEnd())

{

\_column += \_current - \_start;

\_start = \_current;

ScanToken();

}

\_tokens.Add(new Token(TokenType.EOF, "", null, \_line, 1));

return \_tokens;

}

private bool IsAtEnd()

{

return \_current >= \_source.Length;

}

private void ScanToken()

{

char c = Advance();

switch (c)

{

// Single-character tokens

case '(':

AddToken(TokenType.LEFT\_PARENTHESE);

break;

case ')':

AddToken(TokenType.RIGHT\_PARENTHESE);

break;

case '[':

AddToken(TokenType.LEFT\_ANGLE\_BRACKET);

break;

case ']':

AddToken(TokenType.RIGHT\_ANGLE\_BRACKET);

break;

case ',':

AddToken(TokenType.COMMA);

break;

case '.':

AddToken(TokenType.DOT);

break;

case '-':

AddToken(TokenType.MINUS);

break;

case '+':

AddToken(TokenType.PLUS);

break;

case ':':

AddToken(Match('=') ? TokenType.ASSIGN : TokenType.COLON);

break;

case ';':

AddToken(TokenType.SEMICOLON);

break;

case '\*':

AddToken(TokenType.STAR);

break;

case '=':

AddToken(TokenType.EQUAL);

break;

// One- or two-character tokens

case '<':

if (Match('='))

AddToken(TokenType.LESS\_EQUAL);

else if (Match('>'))

AddToken(TokenType.NOT\_EQUAL);

else if (Match('<'))

AddToken(TokenType.SHIFT\_LEFT);

else

AddToken(TokenType.LESS);

break;

case '>':

if (Match('='))

AddToken(TokenType.GREATER\_EQUAL);

else if (Match('>'))

AddToken(TokenType.SHIFT\_RIGHT);

else

AddToken(TokenType.GREATER);

break;

case '/':

if (Match('/')) // if comment

while (Peek() != '\n' && !IsAtEnd())

Advance();

else

AddToken(TokenType.SLASH);

break;

case '{': // multiline comment

while (Peek() != '}' && !IsAtEnd())

Advance();

if (IsAtEnd())

Pascal.Error(\_line, \_column, "Unterminated multiline comment");

break;

case ' ':

case '\r':

case '\t':

break;

case '\n':

\_column = 0;

\_line++;

break;

case '"':

case '\'':

String(c);

break;

default:

if (IsDigit(c))

Number();

else if (IsAlpha(c) || c == '\_')

Identifier();

else

Pascal.Error(\_line, \_column, "Unexpected character");

break;

}

}