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. 053502

Аралин И.О.

Проверил

Ассистент кафедры информатики

Гриценко Н.Ю.

Минск 2023

**СОДЕРЖАНИЕ**

[1. Цель работы 3](#_Toc129266355)

[2. Краткие теоретические сведения 4](#_Toc129266356)

[3. Виды токенов лексического анализатора 5](#_Toc129266357)

[4. Демонстрация работы лексического анализатора 6](#_Toc129266358)

[4.1 Результаты работы 6](#_Toc129266359)

[4.2 Лексические ошибки 8](#_Toc129266360)

[5. Выводы 10](#_Toc129266361)

Приложение А (информационное) [Код тестовой программы 11](#_Toc129266362)

Приложение Б (информационное) [Код лексического анализатора 12](#_Toc129266363)

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. **Виды токенов лексического анализатора**

Для выбранного подмножества языка можно выделить следующие виды токенов:

* Токены, соответствующие переменным, например, variable1.
* Токены, соответствующие числам с плавающей запяятой, например, 5.5.
* Токены, соответствующий строковым литералам, например, “Hello world!”.
* Токены, соответствующие целым числам, например, 123.
* Токены, соответствующие ключевым словам: auto, break, case, const, continue, default, do, else, enum, extern, for, goto, if, register, return, sizeof, static, struct, switch, typedef, union, volatile, while, include, using, std::, std, namespace, cin, cout.
* Токены, соответствующие типам данных: bool, char, double, float, int, long, short, signed, unsigned, void, string.
* Токены, соответствующие операторам: +, -, \*, /, %, ++, --, ==, !=, <, <=, >, >=, !, &&, ||, &, |, ^, ~, <<, >>, +=, -=, \*=, /=, %=, &=, |=, ^=, <<=, >>=.
* Токены, соответствующие разделителям: '(', ')', '[', ']', '{', '}', ',', ';', ':'.

1. **Демонстрация работы лексического анализатора**
   1. **Результаты работы**

Рассмотрим результат лексического анализа тестовой программы (см. приложение А) программой-анализатором (см. приложение Б):

* Демонстрация используемых ключевых слов представлена на рисунке 1:

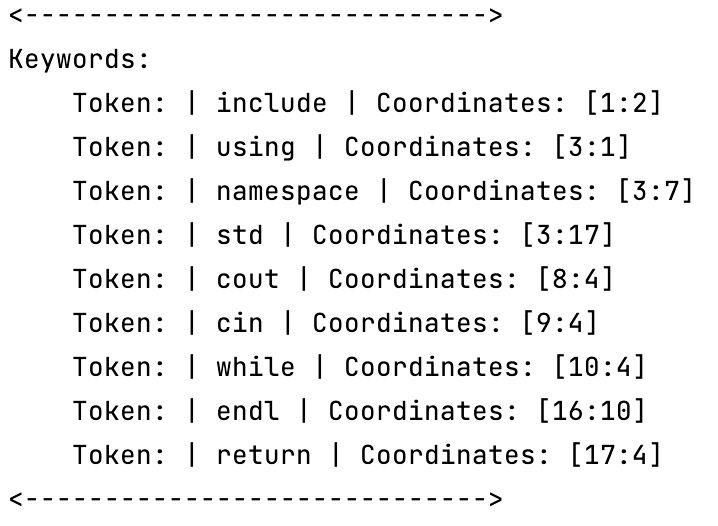


Рисунок 1 – Ключевые слова

* Демонстрация используемых типов переменных представлена на рисунке 2:

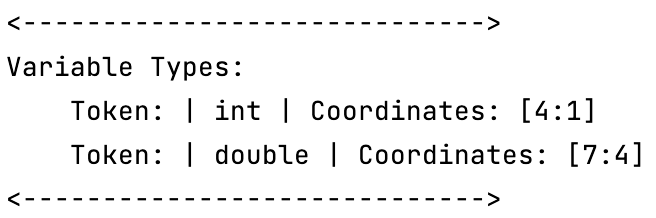


Рисунок 2 – Типы переменных

* Демонстрация используемых операторов представлена на рисунке 3:

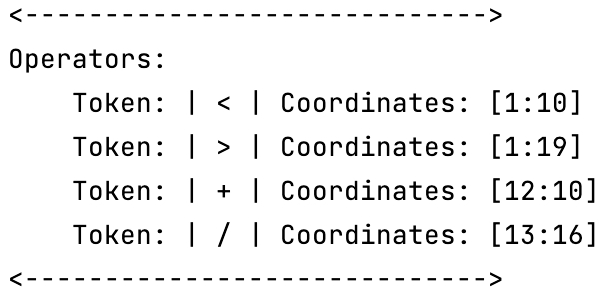


Рисунок 3 – Операторы

* Демонстрация используемых разделителей представлена на рисунке 4:

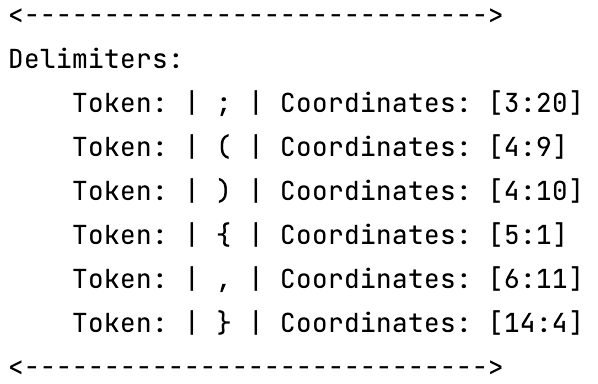


Рисунок 4 – Разделители

* Демонстрация используемых идентификаторов представлена на рисунке 5:

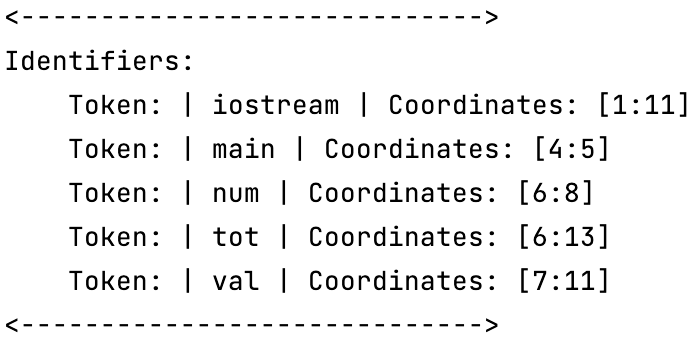


Рисунок 5 – Идентификаторы

* Демонстрация используемых целых чисел представлена на рисунке 6:

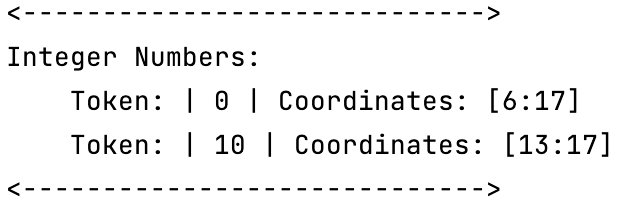


Рисунок 6 – Целые числа

* Демонстрация используемых дробных чисел представлена на рисунке 7:

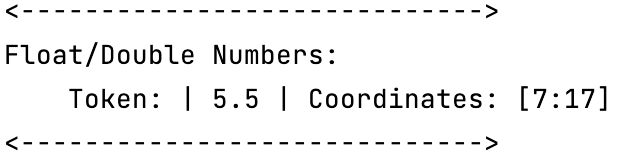


Рисунок 7 – Дробные числа

* Демонстрация используемых строковых литералов представлена на рисунке 8:

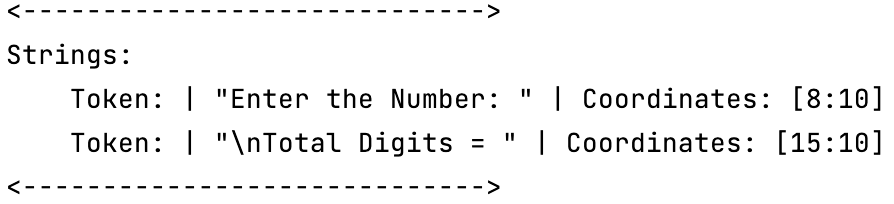


Рисунок 8 – Строковые литералы

Видно, что каждый токен имеет ряд свойств:

* Категория токена;
* значение токена;
* позиция в исходном файле в формате [строка: колонка].
  1. **Лексические ошибки**

Ошибка неожиданного символа – производится, когда лексер встречает символ, не использующийся в подмножестве языка программирования. Результат анализа ошибки представлен на рисунке 9. Входная программа: “int num, tot=0; asd”.

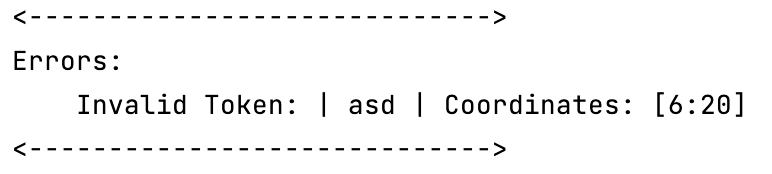


Рисунок 9 – Пример ошибки неожиданного символа

Ошибка недопустимых символов после целого числа – производится, когда лексер встречает неверное написание целого числа. Результат анализа ошибки представлен на рисунке 10. Входная программа: “int num, tot=0asd;”.

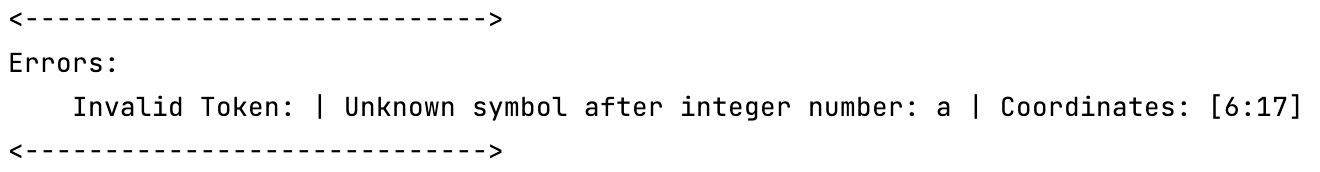


Рисунок 10 – Пример ошибки недопустимых символов после целого числа

Ошибка, когда двойные кавычки открыты, но не закрыты – производится, когда лексер встречает неверную расстановку двойных кавычек в строке. Результат анализа ошибки представлен на рисунке 11. Входная программа: “cout<<"Enter the Number: ;”.

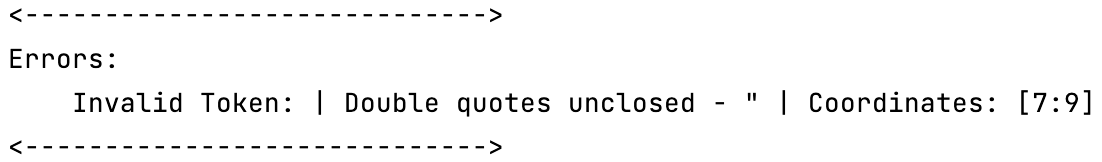


Рисунок 11 – Пример ошибки с двойными кавычками

Ошибка, когда одинарные кавычки открыты, но не закрыты – производится, когда лексер встречает неверную расстановку одинарных кавычек в строке. Результат анализа ошибки представлен на рисунке 12. Входная программа: “cout<<'a;”.

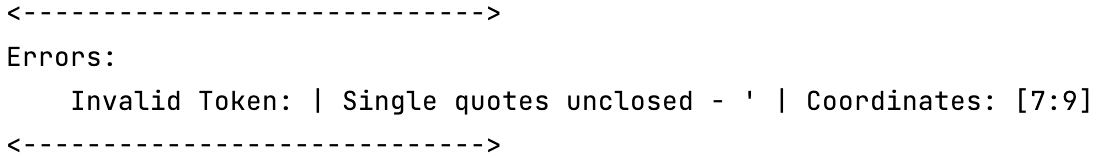


Рисунок 12 – Пример ошибки с одинарными кавычками

Ошибка недопустимых символов после числа с плавающей запятой – производится, когда лексер встречает неверное написание числа с плавающей запятой. Результат анализа ошибки представлен на рисунке 13. Входная программа: “double val = 5.5.;”.

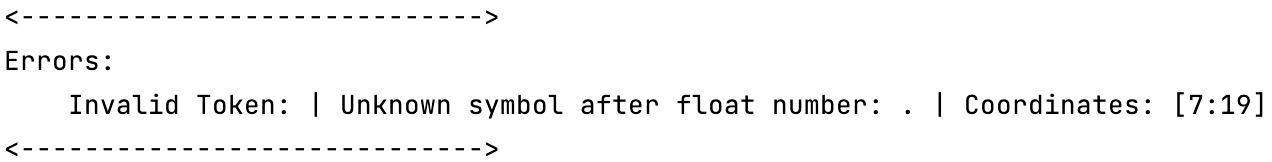


Рисунок 13 – Пример ошибки недопустимых символов после числа с плавающей запятой

1. **Выводы**

Таким образом, в ходе лабораторной работы было изучено понятие лексического анализа в теории трансляции. Было создано приложение-лексер для выбранного подмножества языка. В процессе работы были исследованы способы определения лексических единиц, особенности их хранения и обработки. Было замечено, что для лексического анализа эффективно использовать сканирование исходного текста посимвольно.

В результате был получен ценный практический и теоретический опыт работы с лексическим анализом и созданием лексеров. Это знание может быть полезным для дальнейшей работы с компиляторами и интерпретаторами языков программирования, а также для разработки собственных лексических анализаторов для других языков программирования.

**ПРИЛОЖЕНИЕ А**

**(информационное)**

# Код тестовой программы

#include <iostream>

using namespace std;

int main()

{

int num, tot=0;

cout<<"Enter the Number: ";

cin>>num;

while (num > 0)

{

tot++;

num = num/10;

}

cout<<"\nTotal Digits = "<<tot;

cout<<endl;

return 0;

}

**ПРИЛОЖЕНИЕ Б**

**(информационное)**

# Код лексического анализатора

import re

# Define regular expressions for each token category

keywords = ['break', 'case', 'const', 'continue', 'default', 'endl',

'do', 'else', 'enum', 'extern', 'for', 'goto',

'if', 'register', 'return', 'namespace', 'cin', 'cout',

'sizeof', 'static', 'struct', 'switch', 'typedef', 'union',

'volatile', 'while', 'include', 'using', 'std', 'std::']

var\_types = ['bool', 'char', 'double', 'float', 'int', 'long', 'short', 'signed', 'unsigned', 'void', 'string']

operators = ['+', '-', '\*', '/', '%', '++', '--', '==', '!=', '<', '<=', '>',

'>=', '!', '&&', '||', '&', '|', '^', '~', '<<', '>>', '+=',

'-=', '\*=', '/=', '%=', '&=', '|=', '^=', '<<=', '>>=']

delimiters = ['(', ')', '[', ']', '{', '}', ',', ';', ':']

identifiers = r'[a-zA-Z\_][a-zA-Z0-9\_]\*'

int\_numbers = r'\d+(\.\d\*)?([Ee][+-]?\d+)?'

float\_numbers = r'\d+\.\d+([Ee][+-]?\d+)?'

strings = r'"([^"\\]|\\.)\*"|\'([^\'\\]|\\.)\*\''

# Define a regular expression for matching all tokens

token\_pattern = re.compile('|'.join([re.escape(keyword) for keyword in keywords] +

[re.escape(operator) for operator in operators] +

[re.escape(delimiter) for delimiter in delimiters] +

[identifiers, int\_numbers, float\_numbers, strings]))

# Read the input C++ file

with open('main.cpp', 'r') as file:

code = file.readlines()

# Initialize dictionary to hold tokens for each category

token\_categories = {

'Keywords': [],

'Variable Types': [],

'Operators': [],

'Delimiters': [],

'Identifiers': [],

'Integer Numbers': [],

'Float/Double Numbers': [],

'Strings': [],

'Errors': []

}

# Find all tokens in the code and categorize them

for line\_no, line in enumerate(code, 1):

line\_end = False

for match in token\_pattern.finditer(line):

token = match.group(0)

col\_start = match.start() + 1

col\_end = match.end()

if token in keywords and line\_end is False:

category = 'Keywords'

elif token in var\_types and line\_end is False:

category = 'Variable Types'

elif token in operators and line\_end is False:

category = 'Operators'

elif token in delimiters and line\_end is False:

category = 'Delimiters'

if token == ';' and line.\_\_contains\_\_('for') is False and line[(col\_end + 1):].\_\_contains\_\_(';') is False:

line\_end = True

elif re.match(identifiers, token) and line\_end is False:

category = 'Identifiers'

elif re.match(float\_numbers, token) and line\_end is False:

category = 'Float/Double Numbers'

# Bad float/double errors detect

if line[col\_end] not in (',', ';', ' ', '+', '-', '\*', '=', '/', '%', '(', ')', '[', ']', '}'):

token\_categories['Errors'].append(("Unknown symbol after float number: " + line[col\_end], line\_no, col\_end, col\_end))

elif re.match(int\_numbers, token) and line\_end is False:

category = 'Integer Numbers'

if line[col\_start - 2] == '.':

category = 'Errors'

# Bad integer errors detect

if line[col\_end] not in (',', ';', ' ', '+', '-', '\*', '=', '/', '%', '(', ')', '[', ']', '}'):

token\_categories['Errors'].append(("Unknown symbol after integer number: " + line[col\_end], line\_no, col\_end, col\_end))

elif re.match(strings, token) and line\_end is False:

category = 'Strings'

else:

category = 'Errors'

token\_categories[category].append((token, line\_no, col\_start, col\_end))

# Quotes errors detect

if line.count("'") % 2 != 0:

token\_categories['Errors'].append(("Single quotes unclosed - '", line\_no, line.rfind("'"), line.rfind("'")))

for item in token\_categories['Strings']:

if item[1] == line\_no and item[2] == line.rfind("'") - 1:

token\_categories['Strings'].pop(token\_categories['Strings'].index(item))

if line.count('"') % 2 != 0:

token\_categories['Errors'].append(('Double quotes unclosed - "', line\_no, line.rfind('"'), line.rfind('"')))

for item in token\_categories['Strings']:

if item[1] == line\_no and item[2] == line.rfind('"') - 1:

token\_categories['Strings'].pop(token\_categories['Strings'].index(item))

# Semicolon errors detect

if re.match('^(?! \*#)(?:(?!for\b|while\b|if\b|else\b|'

'do\b|switch\b|case\b|default\b|struct\b|'

'class\b|namespace\b|typedef\b|return\b)'

'[^;{}]\*;|.\*\{.\*\}.\*|.\*\}.\*|.\*\{.\*)', line) is None\

and re.match('^(?:\s\*|.\*//.\*)$|^\s\*#\s\*include\s+<.\*>\s\*$', line) is None\

and re.match('^\s\*if\s\*\([^;]+?\)[^{;]\*$', line) is None\

and re.match('^\s\*[a-zA-Z\_]+\s+[a-zA-Z\_]+'

'\s\*\(\s\*([a-zA-Z\_]+\s+[\*&]\*'

'\s\*[a-zA-Z\_]+[\[\]]\*\s\*,'

'\s\*)\*([a-zA-Z\_]+\s+[\*&]\*'

'\s\*[a-zA-Z\_]+[\[\]]\*)?\s\*\)\s\*$', line) is None\

and line.\_\_contains\_\_(';') is False\

and line.\_\_contains\_\_('while') is False:

token\_categories['Errors'].append(('; was expected at end of the line',

line\_no, line[str.\_\_len\_\_(line) - 1], line[str.\_\_len\_\_(line) - 1]))

print("<----------------------------->")

for category, tokens in token\_categories.items():

print(f'{category}:')

if category != "Errors":

for token in tokens:

print(f'\tToken: | {token[0]} | Coordinates: [{token[1]}:{token[2]}]')

else:

for token in tokens:

print(f'\tInvalid Token: | {token[0]} | Coordinates: [{token[1]}:{token[2]}]')

print("<----------------------------->")