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

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. 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: '(', ')', '[', ']', '{', '}', ',', ';', ':'.

4. Демонстрация работы лексического анализатора

4.1 Результаты работы

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

– Демонстрация используемых ключевых слов представлена на рисунке 1:

```
<---------------------------->
Keywords:
    Token: | include | Coordinates: [1:2]
    Token: | using | Coordinates: [3:1]
    Token: | namespace | Coordinates: [3:7]
    Token: | std | Coordinates: [3:17]
    Token: | cout | Coordinates: [8:4]
    Token: | cin | Coordinates: [9:4]
    Token: | while | Coordinates: [10:4]
    Token: | endl | Coordinates: [16:10]
    Token: | return | Coordinates: [17:4]
<---------------------------->
```

Рисунок 1 – Ключевые слова

– Демонстрация используемых типов переменных представлена на рисунке 2:

```
<---------------------------->
Variable Types:
    Token: | int | Coordinates: [4:1]
    Token: | double | Coordinates: [7:4]
<---------------------------->
```

Рисунок 2 – Типы переменных

– Демонстрация используемых операторов представлена на рисунке 3:

```
<---------------------------->
Operators:
    Token: | < | Coordinates: [1:10]
    Token: | > | Coordinates: [1:19]
    Token: | + | Coordinates: [12:10]
    Token: | / | Coordinates: [13:16]
<---------------------------->
```

Рисунок 3 – Операторы

– Демонстрация используемых разделителей представлена на рисунке 4:

```
<---------------------------->
Delimiters:
    Token: | ; | Coordinates: [3:20]
    Token: | ( | Coordinates: [4:9]
    Token: | ) | Coordinates: [4:10]
    Token: | { | Coordinates: [5:1]
    Token: | , | Coordinates: [6:11]
    Token: | } | Coordinates: [14:4]
<---------------------------->
```

Рисунок 4 – Разделители

– Демонстрация используемых идентификаторов представлена на рисунке 5:

```
<---------------------------->
Identifiers:
    Token: | iostream | Coordinates: [1:11]
    Token: | main | Coordinates: [4:5]
    Token: | num | Coordinates: [6:8]
    Token: | tot | Coordinates: [6:13]
    Token: | val | Coordinates: [7:11]
<---------------------------->
```

Рисунок 5 – Идентификаторы

– Демонстрация используемых целых чисел представлена на рисунке 6:

```
<---------------------------->
Integer Numbers:
    Token: | 0 | Coordinates: [6:17]
    Token: | 10 | Coordinates: [13:17]
<---------------------------->
```

Рисунок 6 – Целые числа

– Демонстрация используемых дробных чисел представлена на рисунке 7:

```
<---------------------------->
Float/Double Numbers:
    Token: | 5.5 | Coordinates: [7:17]
<---------------------------->
```

Рисунок 7 – Дробные числа

– Демонстрация используемых строковых литералов представлена на рисунке 8:

```
<---------------------------->
Strings:
    Token: | "Enter the Number: " | Coordinates: [8:10]
    Token: | "\nTotal Digits = " | Coordinates: [15:10]
<---------------------------->
```

Рисунок 8 – Строковые литералы

Видно, что каждый токен имеет ряд свойств:
– Категория токена;
– значение токена;
– позиция в исходном файле в формате [строка: колонка].

4.2 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”.

```
<---------------------------->
Errors:
    Invalid Token: | asd | Coordinates: [6:20]
<---------------------------->
```

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;”.

```
<---------------------------->
Errors:
    Invalid Token: | Unknown symbol after integer number: a | Coordinates: [6:17]
<---------------------------->
```

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: ;”.

```
<---------------------------->
Errors:
    Invalid Token: | Double quotes unclosed - " | Coordinates: [7:9]
<---------------------------->
```

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;”.

```
<---------------------------->
Errors:
    Invalid Token: | Single quotes unclosed - ' | Coordinates: [7:9]
<---------------------------->
```

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.;”.

```
<---------------------------->
Errors:
    Invalid Token: | Unknown symbol after float number: . | Coordinates: [7:19]
<---------------------------->
```

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

5. 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("<---------------------------->")
```