

Министерство образования Республики Беларусь
Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

КАФЕДРА ИНФОРМАТИКИ

Отчёт по лабораторной работе № 2
по теме «Лексический анализ программ»

Выполнил:
студент гр. 053501
Уласевич А.А.

Проверил:
Ассистент кафедры
информатики
Гриценко Н. Ю.

Минск 2023

СОДЕРЖАНИЕ

1 Цель работы.....	2
2 Теоретические сведения.....	3
3 Результат выполнения программы.....	4
4 Нахождение и локализация лексических ошибок	7
Приложение А(обязательное) Листинг программного кода	8

1 ЦЕЛЬ РАБОТЫ

Необходимо разработать лексический анализатор подмножества языка программирования C, определённого в лабораторной работе 1. Программа анализа определяет лексические правила и выполняет перевод потока символов программ лабораторной работы 1 в поток лексем (токенов). На вход программы подается поток, содержащий текст программы.

Лексический анализатор читает поток символов, составляющих исходную программу, и группирует эти символы в значащие последовательности, называемые лексемами.

После того, как в результате лексического анализа лексемы распознаны, информация о некоторых из них собирается и записывается в таблицу имен.

Если позднее во входной цепочке попадает идентификатор, надо справиться в этой таблице, не появлялся ли он ранее. Если да, то лексема, соответствующая новому вхождению этого идентификатора, будет той же, что и у предыдущего вхождения.

При определении неверной последовательности символов, необходимо обнаружить эту ошибку и выдать сообщение о ней. Всего необходимо показать скриншоты нахождения 4-х лексических ошибок.

2 ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Лексический анализатор (ЛА) — это первый этап процесса компиляции, на котором символы, составляющие исходную программу, группируются в отдельные минимальные единицы текста, несущие смысловую нагрузку — лексемы. Задача лексического анализа — выделить лексемы и преобразовать их к виду, удобному для последующей обработки. ЛА использует регулярные грамматики.

ЛА необязательный этап компиляции, но желательный по следующим причинам:

1 Замена идентификаторов, констант, ограничителей и служебных слов лексемами делает программу более удобной для дальнейшей обработки;

2 ЛА уменьшает длину программы, устраняя из ее исходного представления несущественные пробелы и комментарии;

3 Если будет изменена кодировка в исходном представлении программы, то это отразится только на ЛА.

На вход программы подается текстовый файл, содержащий строки символов анализируемой программы. Например, строка может задавать переменной значения арифметического выражения в виде

ПЕРЕМЕННАЯ = ВЫРАЖЕНИЕ.

Выражение может включать:

1 Знаки сложения и умножения («+» и «*»);

2 Круглые скобки («(» и «)»);

3 Константы (например, 5; 3.8; 1e+18, 8.41E-10);

4 Имена переменных.

Имя переменной – это последовательность букв и цифр, начинающаяся с буквы.

Разбор выражения $COST = (PRICE + TAX) * 0.98$.

Проанализируем выражение:

– COST, PRICE, TAX — лексемы-идентификаторы

– 0.98 — лексема-константа; =, +, * — просто лексемы.

Пусть все константы и идентификаторы можно отображать в лексемы типа <идентификатор> (<ИД>). Тогда выходом лексического анализатора будет последовательность лексем $\langle ИД_1 \rangle = (\langle ИД_2 \rangle + \langle ИД_3 \rangle) * \langle ИД_4 \rangle$.

Вторая часть компоненты лексемы (указатель, т.е. номер лексемы в таблице имен) — показана в виде индексов. Символы «=», «+» и «*» трактуются как лексемы, тип которых представляется ими самими. Они не имеют связанных с ними данных и, следовательно, не имеют указателей.

3 РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ ПРОГРАММЫ

В результате работы лексического анализатора в консоли отображены таблицы с токенами, представленные на рисунках 1 – 5.

```
/Users/artem/PycharmProjects/MTran/se
```

CONSTANTS TABLE		
TOKEN	DESCRIPTION	
2	Constant of int type	
2.75	Constant of float type	
1	Constant of int type	
20	Constant of int type	
1	Constant of int type	
2	Constant of int type	
0	Constant of int type	

Рисунок 1 – Токены констант

KEY WORDS TABLE		
TOKEN	DESCRIPTION	
for	Loop statement	
if	Condition statement	
continue	Continue statement	
if	Condition statement	
continue	Continue statement	
return	Return statement	

Рисунок 2 – Токены ключевых слов

OPERATORS TABLE		
TOKEN	DESCRIPTION	
=	Assignment operator	
=	Assignment operator	
=	Assignment operator	
<=	Relational operator	
++	Arithmetic operator	
==	Relational operator	
==	Relational operator	
=	Assignment operator	
+	Arithmetic operator	
=	Assignment operator	
=	Assignment operator	

Рисунок 3 – Токены операторов

IDENTIFIERS TABLE		
TOKEN	DESCRIPTION	
i	Identifier of Int data type	
c	Identifier of Float data type	

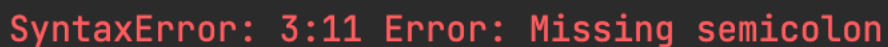
Рисунок 4 – Токены переменных

FUNCTIONS TABLE		
TOKEN	DESCRIPTION	
main	Main function	
printf	Output function	
printf	Output function	
printf	Output function	

Рисунок 5 – Токены встроенных функций

4 НАХОЖДЕНИЕ И ЛОКАЛИЗАЦИЯ ЛЕКСИЧЕСКИХ ОШИБОК

При наличии ошибок в коде программы, лексический анализатор их обнаружит и выведет в консоли уведомление о данной ошибке. Список ошибок представлен на рисунках 6 – 9.



```
SyntaxError: 3:11 Error: Missing semicolon
```

Рисунок 6 – Пропущенный знак «;»



```
SyntaxError: 5:7 Error: redeclaration of 'i'
```

Рисунок 7 – Попытка переопределения переменной «i»



```
SyntaxError: 5:3 Error: 'b' was not declared in this scope
```

Рисунок 8 – Переменная «b» не определена



```
SyntaxError: 2:0 Error: Unmatched opening bracket at token: '{'
```

Рисунок 9 – Отсутствие закрывающейся скобки



```
SyntaxError: 7:18 Expected identifier before '!' token
```

Рисунок 10 – Попытка использования оператора «<!=»

ПРИЛОЖЕНИЕ А
(обязательное)
Листинг программного кода

Файл main.py:

```
import re
import prettytable
import tokens

def extract_tokens(text):
    l = []

    for index, line in enumerate(text.splitlines()):
        for match in re.finditer(tokens.pattern, line):
            l.append((index, match.start(), match.group()))

    return l

with open('/Users/artem/PycharmProjects/MTran/second/2/test.c', 'r') as f:
    text = f.read()

identifiers_table = prettytable.PrettyTable()
constants_table = prettytable.PrettyTable()
keywords_table = prettytable.PrettyTable()
datatypes_table = prettytable.PrettyTable()
operators_table = prettytable.PrettyTable()
directives_table = prettytable.PrettyTable()
functions_table = prettytable.PrettyTable()

identifiers_table.title = 'IDENTIFIERS TABLE'
identifiers_table.field_names = ['TOKEN', 'DESCRIPTION']

constants_table.title = 'CONSTANTS TABLE'
constants_table.field_names = ['TOKEN', 'DESCRIPTION']

keywords_table.title = 'KEY WORDS TABLE'
keywords_table.field_names = ['TOKEN', 'DESCRIPTION']

datatypes_table.title = 'DATA TYPES TABLE'
datatypes_table.field_names = ['TOKEN', 'DESCRIPTION']

operators_table.title = 'OPERATORS TABLE'
```

```

operators_table.field_names = ['TOKEN', 'DESCRIPTION']

directives_table.title = 'DIRECTIVES TABLE'
directives_table.field_names = ['TOKEN', 'DESCRIPTION']

functions_table.title = 'FUNCTIONS TABLE'
functions_table.field_names = ['TOKEN', 'DESCRIPTION']

set_of_tokens = extract_tokens(text)

_ident = []
_err = []
_func = False
_brackets = []

for i, token in enumerate(set_of_tokens):

    if token[2] == "{":
        _brackets.append(token)
    elif token[2] == "}":
        if len(_brackets) == 0:
            raise SyntaxError(str(token[0] + 1) + ":" + str(token[1]) + " Error: Unmatched
closing bracket at token: '" + str(token[2]) + "'")
        else:
            _brackets.pop()

    if i + 1 < set_of_tokens.__len__():
        if token[0] < set_of_tokens[i + 1][0]:
            if not (token[2] in tokens.closing) and not (token[2] in tokens.opened):
                if (set_of_tokens[i + 1][2] in tokens.keywords) \
                    or (set_of_tokens[i + 1][2] in tokens.datatypes) \
                    or (set_of_tokens[i + 1][2] in tokens.closing) or (set_of_tokens[i +
1][2] in _ident):
                    raise SyntaxError( str(token[0] + 1) + ":" + str(token[1]) + " Error:
Missing semicolon")

            if token[2] in tokens.keywords:
                if set_of_tokens[i - 1][2] not in tokens.datatypes:
                    keywords_table.add_row([token[2], tokens.keywords[token[2]]])
                    continue
            else:
                raise SyntaxError(str(token[0] + 1) + ":" + str(token[1]) + " Error: Expected
unqualified-id before '" + str(token[2]) + "'")

            if token[2] in tokens.datatypes:

```

```

        datatypes_table.add_row([token[2], tokens.datatypes[token[2]]])
        continue
    if token[2] in tokens.functions:
        functions_table.add_row([token[2], tokens.functions[token[2]]])
        continue
    if token[2] in directives_table:
        directives_table.add_row([token[2], tokens.directives[token[2]]])
        continue

    if set_of_tokens[i - 1][2] == ")":
        _func = False

    if re.search(tokens.string_pattern.pattern, token[2]):
        if set_of_tokens[i - 1][2] in tokens.operators:
            constants_table.add_row([token[2], "Constant of string type"])
            continue
        elif _func:
            constants_table.add_row([token[2], "Constant of string type"])
            continue
        elif set_of_tokens[i - 1][2] == "return":
            constants_table.add_row([token[2], "Constant of string type"])
            continue
        else:
            raise SyntaxError(
                str(token[0] + 1) + ":" + str(token[1]) + " Error: Expected identifier before
constant")

    if re.search(tokens.char_pattern.pattern, token[2]):
        if set_of_tokens[i - 1][2] in tokens.operators:
            constants_table.add_row([token[2], "Constant of char type"])
            continue
        elif _func:
            constants_table.add_row([token[2], "Constant of char type"])
            continue
        elif set_of_tokens[i - 1][2] == "return":
            constants_table.add_row([token[2], "Constant of char type"])
            continue
        else:
            raise SyntaxError(
                str(token[0] + 1) + ":" + str(token[1]) + " Error: Expected identifier before
constant")

    if re.search(tokens.identifier_pattern.pattern, token[2]):
        if token[2] not in _ident:
            if set_of_tokens[i + 1][2] == "(":

```

```

        _func = True
        functions_table.add_row([token[2], "Function"])
    elif set_of_tokens[i - 1][2] in tokens.datatypes:
        _ident.append(token[2])
        identifiers_table.add_row([token[2], "Identifier of " +
str(tokens.datatypes[set_of_tokens[i - 1][2]])])
        continue
    else:
        raise SyntaxError(str(token[0] + 1) + ":" + str(token[1]) + " Error: '" +
str(token[2]) + "' was not declared in this scope")
    else:
        if set_of_tokens[i - 1][2] in tokens.datatypes:
            raise SyntaxError(str(token[0] + 1) + ":" + str(token[1]) + " Error:
redeclaration of '" + str(token[2]) + "'")

    if token[2] in tokens.operators:
        if token[2] == '++':
            if set_of_tokens[i - 1][2] in _ident or set_of_tokens[i + 1][2] in _ident:
                operators_table.add_row([token[2], tokens.operators[token[2]]])
                continue
            else:
                raise SyntaxError(
                    str(token[0] + 1) + ":" + str(token[1]) + " Expected identifier before '" +
str(token[2]) + "' token")

        if not set_of_tokens[i - 1][2] in _ident:
            raise SyntaxError(
                str(token[0] + 1) + ":" + str(token[1]) + " Expected identifier before '" +
str(token[2]) + "' token")
        else:
            operators_table.add_row([token[2], tokens.operators[token[2]]])
            continue

    if re.search(tokens.float_pattern.pattern, token[2]):
        if set_of_tokens[i - 1][2] in tokens.operators:
            constants_table.add_row([token[2], "Constant of float type"])
            continue
        elif _func:
            constants_table.add_row([token[2], "Constant of float type"])
            continue
        elif set_of_tokens[i - 1][2] == "return":
            constants_table.add_row([token[2], "Constant of float type"])
            continue
        else:
            raise SyntaxError(

```

```
        str(token[0] + 1) + ":" + str(token[1]) + " Error: Expected identifier before  
constant")
```

```
if re.search(tokens.int_pattern.pattern, token[2]):  
    if set_of_tokens[i - 1][2] in tokens.operators:  
        constants_table.add_row([token[2], "Constant of int type"])  
        continue  
    elif _func:  
        constants_table.add_row([token[2], "Constant of int type"])  
        continue  
    elif set_of_tokens[i - 1][2] == "return":  
        constants_table.add_row([token[2], "Constant of int type"])  
        continue  
    else:  
        raise SyntaxError(  
            str(token[0] + 1) + ":" + str(token[1]) + " Error: Expected identifier before  
constant")
```

```
_last = set_of_tokens[-1]
```

```
if len(_brackets) != 0:  
    raise SyntaxError(str(_brackets[-1][0] + 1) + ":" + str(_brackets[-1][1]) + " Error:  
Unmatched opening bracket at token: " + str(_brackets[-1][2]) + """)  
if not _last[2].endswith((':', '}')):  
    raise SyntaxError("Missing semicolon at the end of token:" + str(_last[0]) + " " +  
str(_last[1]))
```

```
print(identifiers_table)  
print(constants_table)  
print(keywords_table)  
print(functions_table)  
print(datatypes_table)  
print(operators_table)  
print(directives_table)
```

Файл tokens.py:

```
import re  
keywords = {  
    'break': 'Break statement',  
    'case': 'Case statement',  
    'const': 'Constant identifier statement',  
    'continue': 'Continue statement',  
    'default': 'keyword',
```

```

'do': 'Loop statement',
'else': 'keyword',
'for': 'Loop statement',
'goto': 'keyword',
'if': 'Condition statement',
'return': 'Return statement',
'sizeof': 'keyword',
'struct': 'keyword',
'switch': 'Switch statement',
'unsigned': 'keyword',
'while': 'Loop statement'
}

```

```

datatypes = {
    'bool': 'Boolean data type',
    'char': 'Char data type',
    'double': 'Double data type',
    'enum': 'Enum data type',
    'float': 'Float data type',
    'int': 'Int data type',
    'long': 'Long data type',
    'short': 'Short data type',
    'string': 'String data type',
}

```

```

operators = {
    '<=': 'Relational operator',
    '>=': 'Relational operator',
    '&&': 'Logical operator',
    '||': 'Logical operator',
    '!': 'Logical operator',
    '==': 'Relational operator',
    '=': 'Assignment operator',
    '+=': 'Assignment operator',
    '-=': 'Assignment operator',
    '*=': 'Assignment operator',
    '/=': 'Assignment operator',
    '%=': 'Assignment operator',
    '<<=': 'Assignment operator',
    '>>=': 'Assignment operator',
    '&=': 'Assignment operator',
    '^=': 'Assignment operator',
    '|=': 'Assignment operator',
    '++': 'Arithmetic operator',
    '--': 'Arithmetic operator',
}

```

```

    '!=': 'Relational operator',
    '+': 'Arithmetic operator',
    '-': 'Arithmetic operator',
    '*': 'Arithmetic operator',
    '/': 'Arithmetic operator',
    '%': 'Arithmetic operator',
    '<': 'Relational operator',
    '>': 'Relational operator'
}

opened = {
    '{': 'Separator',
    '(': 'Separator',
}

closing = {
    ';': 'Separator',
    '}': 'Separator',
}

separators = {
    '(': 'Separator',
    ')': 'Separator',
    '{': 'Separator',
    '}': 'Separator',
    '[': 'Separator',
    ']': 'Separator',
    ';': 'Separator',
    ',': 'Separator',
}

directives = {
    '#include': 'Directive statement',
    '#define': 'Directive statement',
    '#ifdef': 'Directive statement',
    '#ifndef': 'Directive statement',
    '#endif': 'Directive statement',
    '#undef': 'Directive statement',
    '#pragma': 'Directive statement',
}

functions = {
    'main': 'Main function',
    'printf': 'Output function',
    'scanf': 'Input function'
}

```

```
}
```

```
keyword_pattern = re.compile('|'.join(list(keywords.keys())))
datatype_pattern = re.compile('|'.join(list(datatypes.keys())))
operators_pattern = re.compile('|'.join(re.escape(op) for op in operators))
separators_pattern = re.compile('|'.join(re.escape(op) for op in separators))
directives_pattern = re.compile('|'.join(list(directives.keys())))
functions_pattern = re.compile('|'.join(list(functions.keys())))
```

```
identifier_pattern = re.compile(r'\b[a-zA-Z_][a-zA-Z0-9_]*\b')
float_pattern = re.compile(r'\b\d+\.\d+\b')
int_pattern = re.compile(r'\b\d+\b')
string_pattern = re.compile(r'\".*?\"')
char_pattern = re.compile(r"'(?:\\.|[^\\"'])*'")
```

```
pattern = re.compile(identifier_pattern.pattern + '|' + keyword_pattern.pattern + '|'
                    + datatype_pattern.pattern + '|'
                    + operators_pattern.pattern + '|'
                    + separators_pattern.pattern + '|'
                    + directives_pattern.pattern + '|'
                    + functions_pattern.pattern + '|'
                    + float_pattern.pattern + '|'
                    + int_pattern.pattern + '|'
                    + string_pattern.pattern + '|' + char_pattern.pattern)
```

Файл test.c:

```
int main()
{
    int i = 2;

    float c = 2.75;

    for (i = 1; i <!= 20; ++i)
    {
        if (i == 1)
        {
            printf(c);
            continue;

            if (i == 2)
            {
                printf(c);
```



```
        continue;
    }
    c = i + c;
    c = i;
    i = c;
    printf(c);
}
return 0;
}
```