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](#_bookmark0)
2. [Теоретические сведения 3](#_bookmark1)
3. [Результат выполнения программы 4](#_bookmark2)
4. [Нахождение и локализация лексических ошибок 7](#_bookmark3)

[Приложение А(обязательное) Листинг программного кода 8](#_bookmark4)

# ЦЕЛЬ РАБОТЫ

Необходимо разработать лексический анализатор подмножества языка программирования С, 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 исход- ную 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 попадается идентификатор, надо спра- виться в этой таблице, не появлялся ли он ранее. Если да, то лексема, соответ- ствующая 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 пока- зать скриншоты нахождения 4-х лексических ошибок.

# 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 <ИД1>=(<ИД2>+<ИД3>)\*<ИД4>.

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.

# РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ ПРОГРАММЫ

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

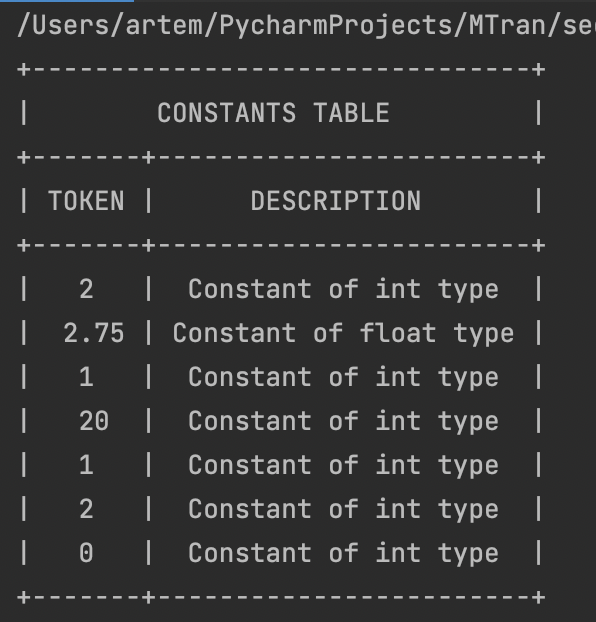


Рисунок 1 – Токены констант

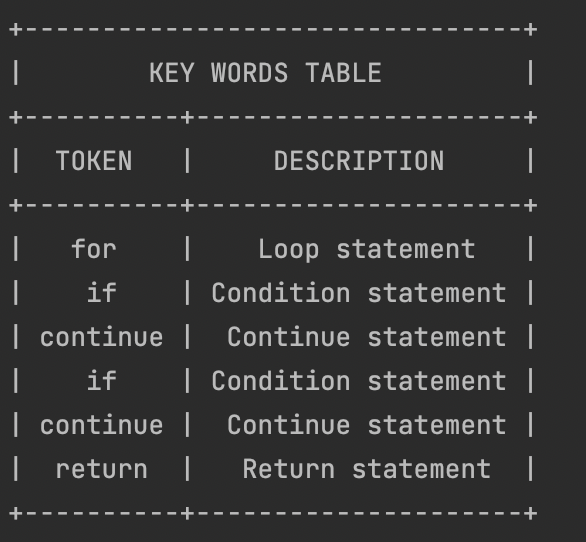


Рисунок 2 – Токены ключевых слов

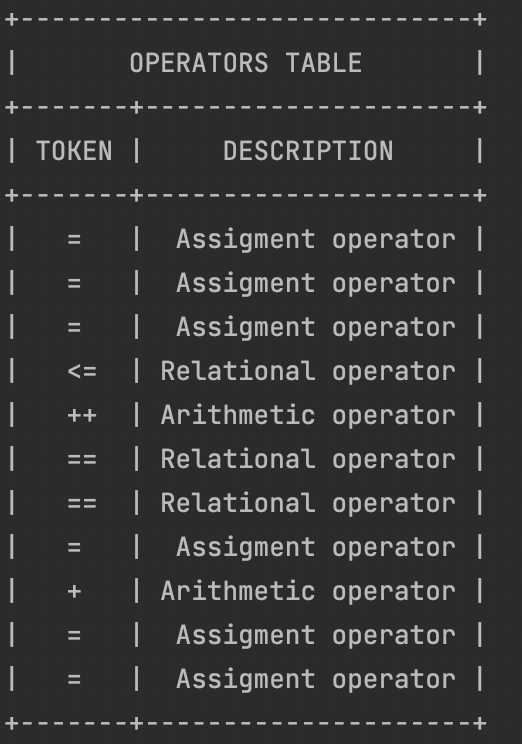


Рисунок 3 – Токены операторов

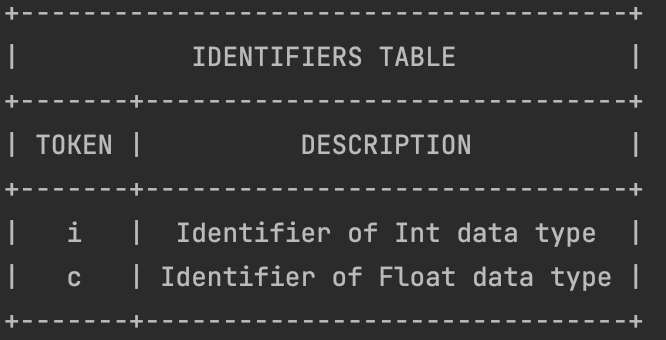


Рисунок 4 – Токены переменных

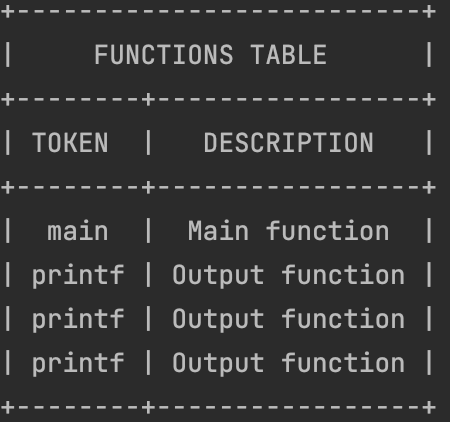


Рисунок 5 – Токены встроенных функций

# 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.



Рисунок 6 – Пропущенный знак «;»



Рисунок 7 – Попытка переопределения переменной «i»



Рисунок 8 – Переменная «b» не определена



Рисунок 9 – Отсутствие закрывающейся скобки



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',  
 '=': 'Assigment operator',  
 '+=': 'Assigment operator',  
 '-=': 'Assigment operator',  
 '\*=': 'Assigment operator',  
 '/=': 'Assigment operator',  
 '%=': 'Assigment operator',  
 '<<=': 'Assigment operator',  
 '>>=': 'Assigment operator',  
 '&=': 'Assigment operator',  
 '^=': 'Assigment operator',  
 '|=': 'Assigment 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;  
}