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. 853504
Шевченя И.В.
Проверил:
Ст. преподаватель КИ Шиманский В. В.

Минск 2021

Содержание

1. **Постановка задачи**

В данной работе ставится задача - освоить работу с существующими лексическими анализаторами. Разработать собственный лексический анализатор подмножества языка программирования, для чего определить лексические правила и выполнить перевод потока символов в поток лексем (токенов). Основной целью работы является реализация лексического анализатора. Лексический анализатор читает поток символов, составляющих исходную программу, и группирует эти символы в значащие последовательности, называемые лексемами.

Исследуемый код языка представлен ниже:

1. Сортировка слиянием

```
void merge(int *a, int n)
{
  int mid = n / 2;
  if (n % 2 == 1)
    mid++;
  int h = 1;
  int *c = (int*)malloc(n * sizeof(int));
  int step;
  while (h < n)
  {
    step = h;
    int i = 0;
    int j = mid;
    int k = 0;
    while (step <= mid)
    {
      while ((i < step) && (j < n) && (j < (mid + step))) {
        if (a[i] < a[j]) {
          c[k] = a[i];
          i++; k++;
        }
        else {
          c[k] = a[j];
          j++; k++;
        }
      }
      while (i < step) {
        c[k] = a[i];
        i++; k++;
      }
      while ((j < (mid + step)) && (j<n)) {
```

```
        c[k] = a[j];
        j++; k++;
      }
      step = step + h;
    }
    h = h * 2;

    for (i = 0; i<n; i++)
      a[i] = c[i];
  }
}

int main()
{
  int a[8];
  for (int i = 0; i < 8; i++)
    a[i] = rand() % 20 – 10;

  for (int i = 0; i < 8; i++)
    printf("%d ", a[i]);

  printf("\n");
  merge(a, 8);
  for (int i = 0; i < 8; i++)
    printf("%d ", a[i]);

  printf("\n");
  getchar();
  return 0;
}
```

2. Перестановка массива в обратном порядке

```
void reverseArray(int arr[], int start, int end)
{
    while (start < end)
    {
        int temp = arr[start];
        arr[start] = arr[end];
        arr[end] = temp;
        start++;
        end--;
    }
}

void printArray(int arr[], int size)
{
    for (int i = 0; i < size; i++)
    cout << arr[i] << " ";
```

```
    cout << endl;
}


int main()
{
    int arr[] = {1, 2, 3, 4, 5, 6};

    int n = sizeof(arr) / sizeof(arr[0]);

    printArray(arr, n);
    rvereseArray(arr, 0, n-1);

    cout << "Reversed array is" << endl;
    printArray(arr, n);

    return 0;
}
```

2. Теория

Лексический анализатор — это программа или часть программы, выполняющая лексический анализ. Лексический анализатор обычно работает в две стадии:

a) сканирование
b) оценка

На первой стадии, сканировании, лексический анализатор обычно реализуется в виде конечного автомата, определяемого регулярными выражениями. В нём кодируется информация о возможных последовательностях символов, которые могут встречаться в токенах . Например, токен «целое число» может содержать любую последовательность десятичных цифр. Во многих случаях первый непробельный символ может использоваться для определения типа следующего токена, после чего входные символы обрабатываются один за другим пока не встретится символ, не входящий во множество допустимых символов для данного токена. В некоторых языках правила разбора лексем несколько более сложные и требуют возвратов назад по читаемой последовательности.

Полученный таким образом токен содержит необработанный исходный текст (строку). Для того чтобы получить токен со значением,

соответствующим типу (напр. целое или дробное число), выполняется оценка этой строки — проход по символам и вычисление значения.

Токен с типом и соответственно подготовленным значением передаётся на вход синтаксического анализатора.

В информатике лексический анализ — процесс аналитического разбора входной последовательности символов на распознанные группы — лексемы, с целью получения на выходе идентифицированных последовательностей, называемых «токенами» (подобно группировке букв в словах). В простых случаях понятия «лексема» и «токен» идентичны, но более сложные токенизаторы дополнительно классифицируют лексемы по различным типам («идентификатор, оператор», «часть речи» и т.п.). Лексический анализ используется в компиляторах и интерпритаторах исходного кода языков программирования, и в различных парсерах естественных языков.

Как правило, лексический анализ производится с точки зрения определённого формального языка или набора языков. Язык, а точнее его грамматика, задаёт определённый набор лексем, которые могут встретиться на входе процесса.

Традиционно принято организовывать процесс лексического анализа, рассматривая входную последовательность символов как поток символов. При такой организации процесс самостоятельно управляет выборкой отдельных символов из входного потока.

Распознавание лексем в контексте грамматики обычно производится путём их идентификации (или классификации) согласно идентификаторам (или классам) токенов, определяемых грамматикой языка. При этом любая последовательность символов входного потока (лексема), которая согласно грамматике не может быть идентифицирована как токен языка, обычно рассматривается как специальный токен-ошибка.

Каждый токен можно представить в виде структуры, содержащей идентификатор токена (или идентификатор класса токена) и, если нужно, последовательность символов лексемы, выделенной из входного потока (строку, число и т. д.).

Цель такой конвертации обычно состоит в том, чтобы подготовить входную последовательность для другой программы, например для граматического анализатора, и избавить его от определения лексических подробностей в контекстно-свободной грамматике (что привело бы к усложнению грамматики).

Фазы

Фаза - это самостоятельная задача в процессе компиляции. Как правило, несколько фаз объединяются в один проход. Фазы лексического анализатора компилятора (также называемый сканером, лексером) переводит входящие данные в форму, более полезную для остальной части компилятора.

Токены и лексемы

Лексический анализатор просматривает входной поток как совокупность основных элементов языка, называемые токенами. Это означает, что токен является лексической неделимой единицей. В C++ ключевые слова **while** и **for** являются токенами (мы не можем сказать wh ile), такие символы как >, >, >= токены, идентификаторы и числа тоже токены и прочее. Исходную строку, содержащую токен, называют лексемой. Обратите внимание, что токен и лексемы похожи, но не одно и то же. Например, токены идентификатор или число могут иметь множество связанных с ними лексем (10, 2.3, 506 - лексемы, число - токен), в тоже время токен совпадает с лексемой состоящей из одного символа. Ситуацию осложняют токены, которые пересекаются с другими (например, какой токен нужно распознать в потоке ">>=": ">", ">>" или ">=" ?). В общем лексический анализатор распознает токен, который соответствует самой длинной лексеме - во многих языках программирование такое поведение указано в спецификации. Таким образом в нашем случае мы получим токен сдвига ">>", а не два токена больше чем.

Выбор множества токенов

Одним из первых проектных решений, которые могут повлиять на структуру всего компилятора, является выбор множества токенов. Вы можете иметь токены для каждаго входного символа или несколько символов могут быть объединены в один токен. Например, символы >, >=, >>, и >>= могут рассматриваться либо как четыре токена, либо как один токен оператор-сравнения. При этом, лексема используется для устранения неоднозначности токена. Первый подход может упростить генерацию кода. Однако, слишком много токенов могут сделать парсер слишком большим и трудным в написании. Нет жестких и быстрых правил для выбора как лучше, но после прочтения книги вы поймете проектные соображения и сможете сделать осознанный выбор. В общем, арифметические операции с одним приоритетом и ассоциативностью могут быть сгруппированы вместе, также ключевые слова для определения типа (вроде int или char) могут быть объединены и т.д.

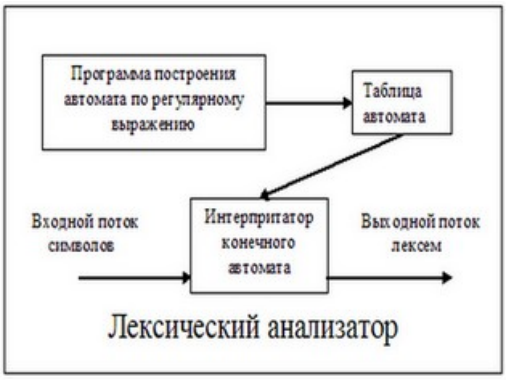


Рис. 2.1. Принцип работы лексера

Анализ текста и разбор кода

Когда программист пытается написать парсер текста, его естественный подход — рекурсивное углубление: найти начало конструкции (например, **{**); найти её конец (например, **}** на том же уровне вложенности); выделить содержимое конструкции, и пропарсить её рекурсивно.

Проблемы с таким подходом — во-первых, избыточная сложность (по одному и тому же фрагменту текста гуляем взад-вперёд); во-вторых, неудобство поддержки (синтаксис языка оказывается рассредоточен по килобайтам и килобайтам ветвистого кода).

Синтаксис языка можно задать декларативно; например, всем знакомы регулярные выражения. Идеально было бы написать стопочку регэкспов для всех конструкций языка, напротив каждого — определение узла, который должен создаться в дереве программы; «универсальный парсер» бы просто подставлял программу в один регэксп за другим, и создавал узлы согласно описанию, один за другим.

Первая из названных проблем решается тем, что поиск всех регэкспов в тексте можно выполнить за один проход, т.е. нет надобности хранить всю программу в памяти целиком — достаточно читать её по одному символу, обрабатывать символ, и тут же забывать.

Вторая — тем, что теперь у нас есть централизованное, формальное описание языка: можем менять регэкспы, вовсе не трогая код; и наоборот, менять код, не рискуя повредить парсер.

Парсер будет читать входную строку символ за символом, и записывать (сдвигать) прочитанные символы в стек. Как только наверху стека соберётся последовательность (символов и переменных), подходящая к прави-

лу грамматики, автомат вытолкнет всю её из стека, и заменит на переменную, стоящую в левой части подошедшего правила (свёртка). Вся работа автомата заключается в последовательности сдвигов и свёрток.

Интересный момент: автомату на самом деле не важно, какие символы лежат в стеке. Всё равно он не может их сравнить с правилами грамматики, потому что видит только верхний; вместо этого он выбирает, какое правило применить для свёртки, по своему текущему состоянию. Стек ему нужен затем, чтобы знать, в какое состояние перейти после свёртки. Для этого он во время сдвига записывает в стек, вместе с символом, своё текущее состояние; а во время свёртки берёт из стека состояние, записанное под всеми стёртыми символами, и в зависимости от него переходит в следующее состояние.

3. **Результат работы программы**

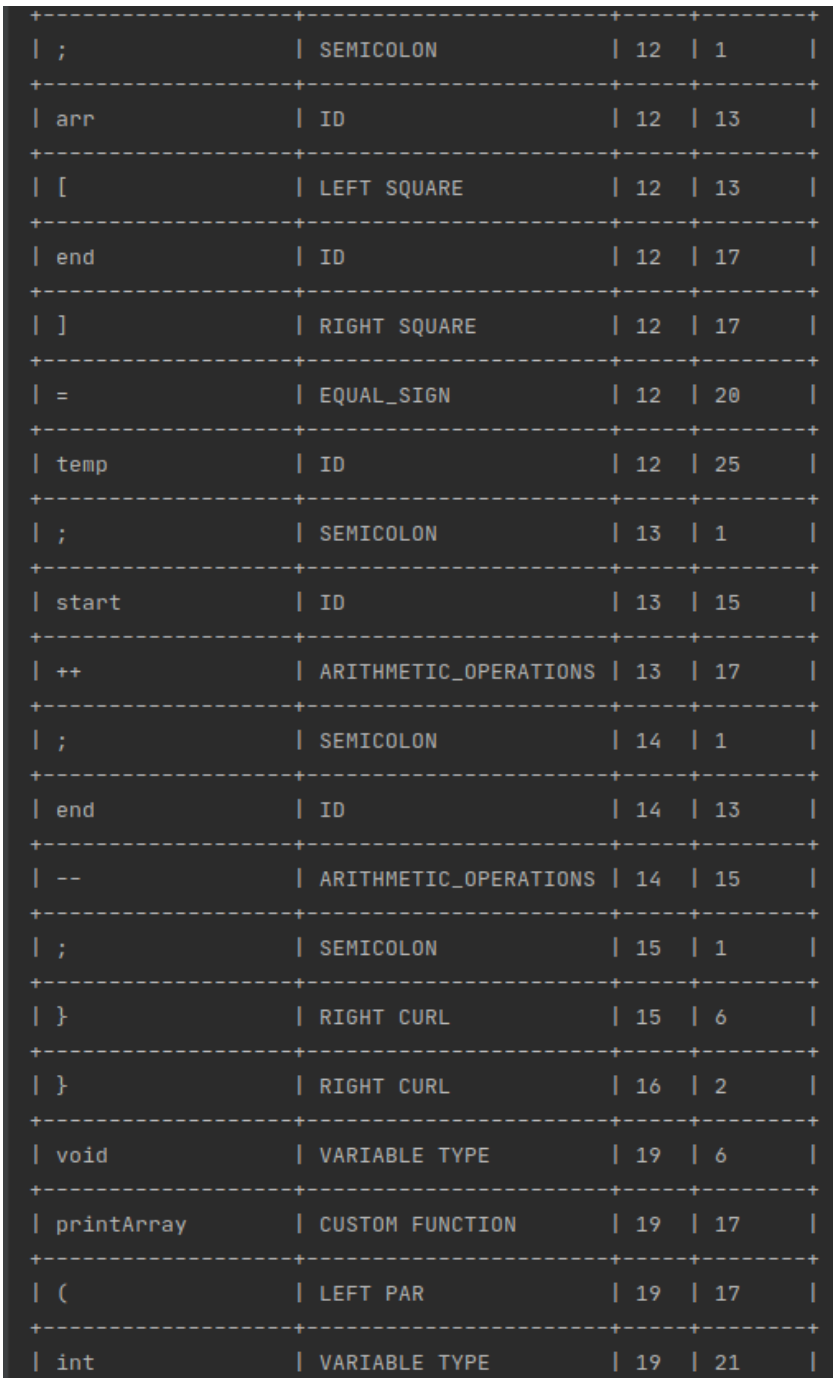
Для анализа программы разложим ее на лексемы и токены. Результат работы программы доступен ниже.

Value (token)	Tag	Row	Column
void	VARIABLE TYPE	6	6
reverseArray	CUSTOM FUNCTION	6	19
(	LEFT PAR	6	19
int	VARIABLE TYPE	6	23
arr	ID	6	27
[	LEFT SQUARE	6	27
]	RIGHT SQUARE	6	28
,	COMMA	6	30
int	VARIABLE TYPE	6	34
start	ID	6	40
,	COMMA	6	41
int	VARIABLE TYPE	6	45
end	ID	6	49
)	RIGHT PAR	6	49
{	LEFT CURL	7	2
while	WHILE	8	11
(	LEFT PAR	8	12
start	ID	8	18
<	LT	8	20

Рис. 1.1. Результат работы анализатора — полная таблица (часть 1)

end	ID	8	24
)	RIGHT PAR	8	24
{	LEFT CURL	9	6
int	VARIABLE TYPE	10	13
temp	ID	10	18
=	EQUAL_SIGN	10	20
arr	ID	10	24
[	LEFT SQUARE	10	24
start	ID	10	30
]	RIGHT SQUARE	10	30
;	SEMICOLON	11	1
arr	ID	11	13
[	LEFT SQUARE	11	13
start	ID	11	19
]	RIGHT SQUARE	11	19
=	EQUAL_SIGN	11	22
arr	ID	11	26
[	LEFT SQUARE	11	26
end	ID	11	30
]	RIGHT SQUARE	11	30
;	SEMICOLON	12	1

Рис. 1.2. Результат работы анализатора — полная таблица (часть 2)

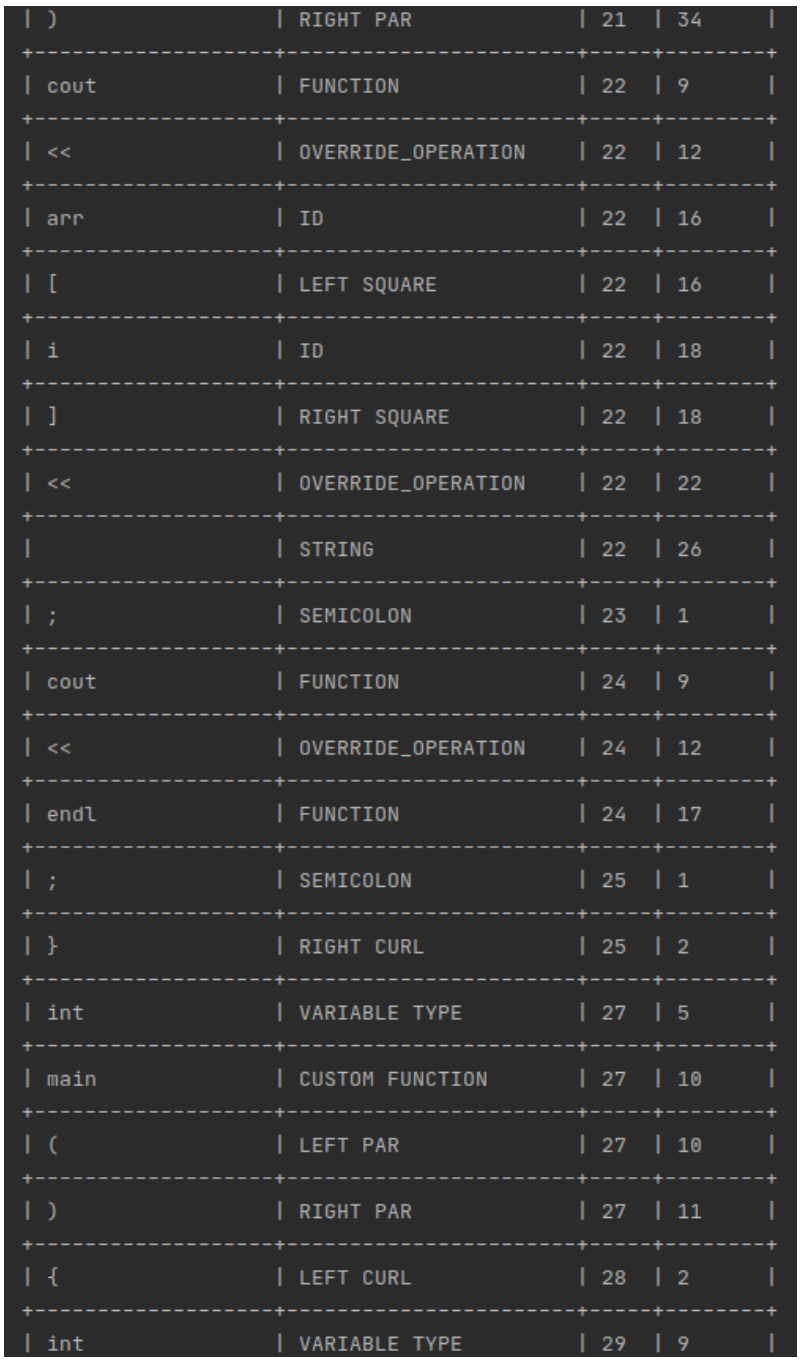


;	SEMICOLON	12	1
arr	ID	12	13
[	LEFT SQUARE	12	13
end	ID	12	17
]	RIGHT SQUARE	12	17
=	EQUAL_SIGN	12	20
temp	ID	12	25
;	SEMICOLON	13	1
start	ID	13	15
++	ARITHMETIC_OPERATIONS	13	17
;	SEMICOLON	14	1
end	ID	14	13
--	ARITHMETIC_OPERATIONS	14	15
;	SEMICOLON	15	1
}	RIGHT CURL	15	6
}	RIGHT CURL	16	2
void	VARIABLE TYPE	19	6
printArray	CUSTOM FUNCTION	19	17
(	LEFT PAR	19	17
int	VARIABLE TYPE	19	21

Рис. 1.3. Результат работы анализатора — полная таблица (часть 3)

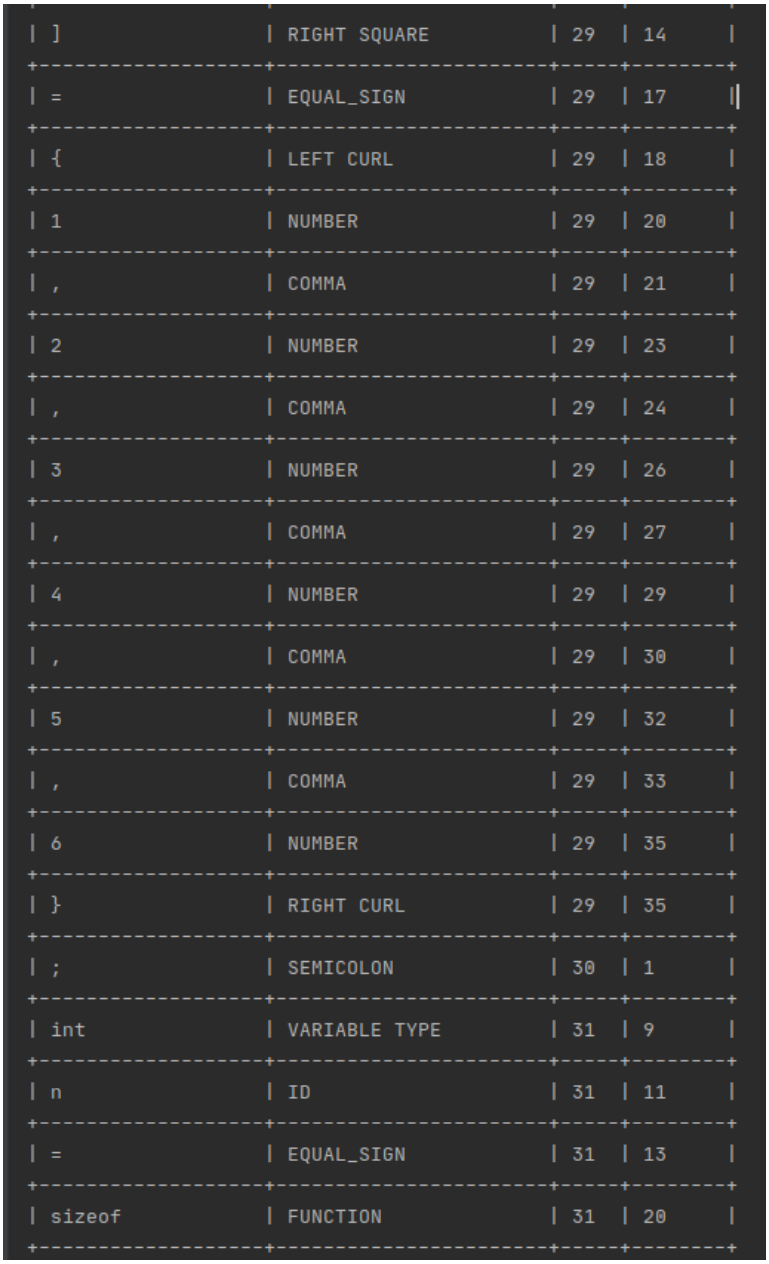
arr	ID	19	25
[	LEFT SQUARE	19	25
]	RIGHT SQUARE	19	26
,	COMMA	19	28
int	VARIABLE TYPE	19	32
size	ID	19	37
)	RIGHT PAR	19	37
{	LEFT CURL	20	2
for	FOR	21	8
(	LEFT PAR	21	9
int	VARIABLE TYPE	21	13
i	ID	21	15
=	EQUAL_SIGN	21	17
0	NUMBER	21	19
;	SEMICOLON	21	20
i	ID	21	22
<	LT	21	24
size	ID	21	29
;	SEMICOLON	21	30
i	ID	21	32
++	ARITHMETIC_OPERATIONS	21	34

Рис. 1.4. Результат работы анализатора — полная таблица (часть 4)



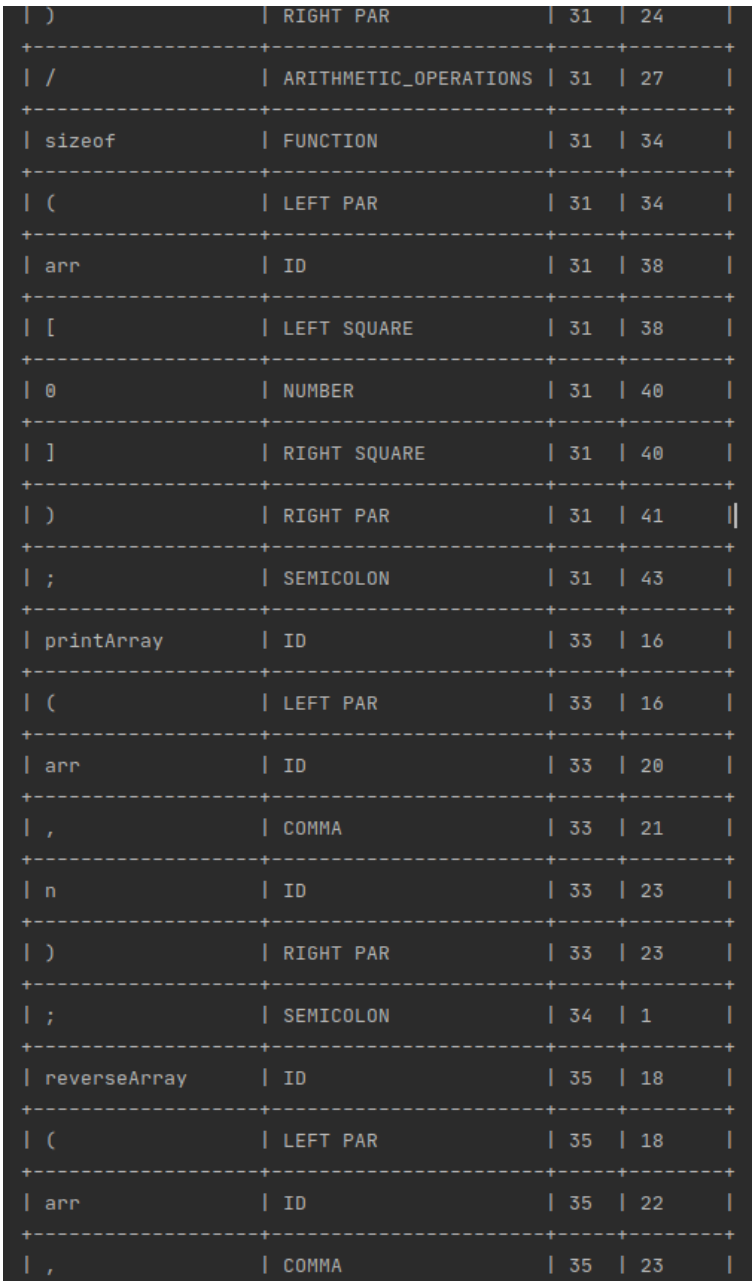
)	RIGHT PAR	21	34
cout	FUNCTION	22	9
<<	OVERRIDE_OPERATION	22	12
arr	ID	22	16
[	LEFT SQUARE	22	16
i	ID	22	18
]	RIGHT SQUARE	22	18
<<	OVERRIDE_OPERATION	22	22
	STRING	22	26
;	SEMICOLON	23	1
cout	FUNCTION	24	9
<<	OVERRIDE_OPERATION	24	12
endl	FUNCTION	24	17
;	SEMICOLON	25	1
}	RIGHT CURL	25	2
int	VARIABLE TYPE	27	5
main	CUSTOM FUNCTION	27	10
(	LEFT PAR	27	10
)	RIGHT PAR	27	11
{	LEFT CURL	28	2
int	VARIABLE TYPE	29	9

Рис. 1.5. Результат работы анализатора — полная таблица (часть 5)



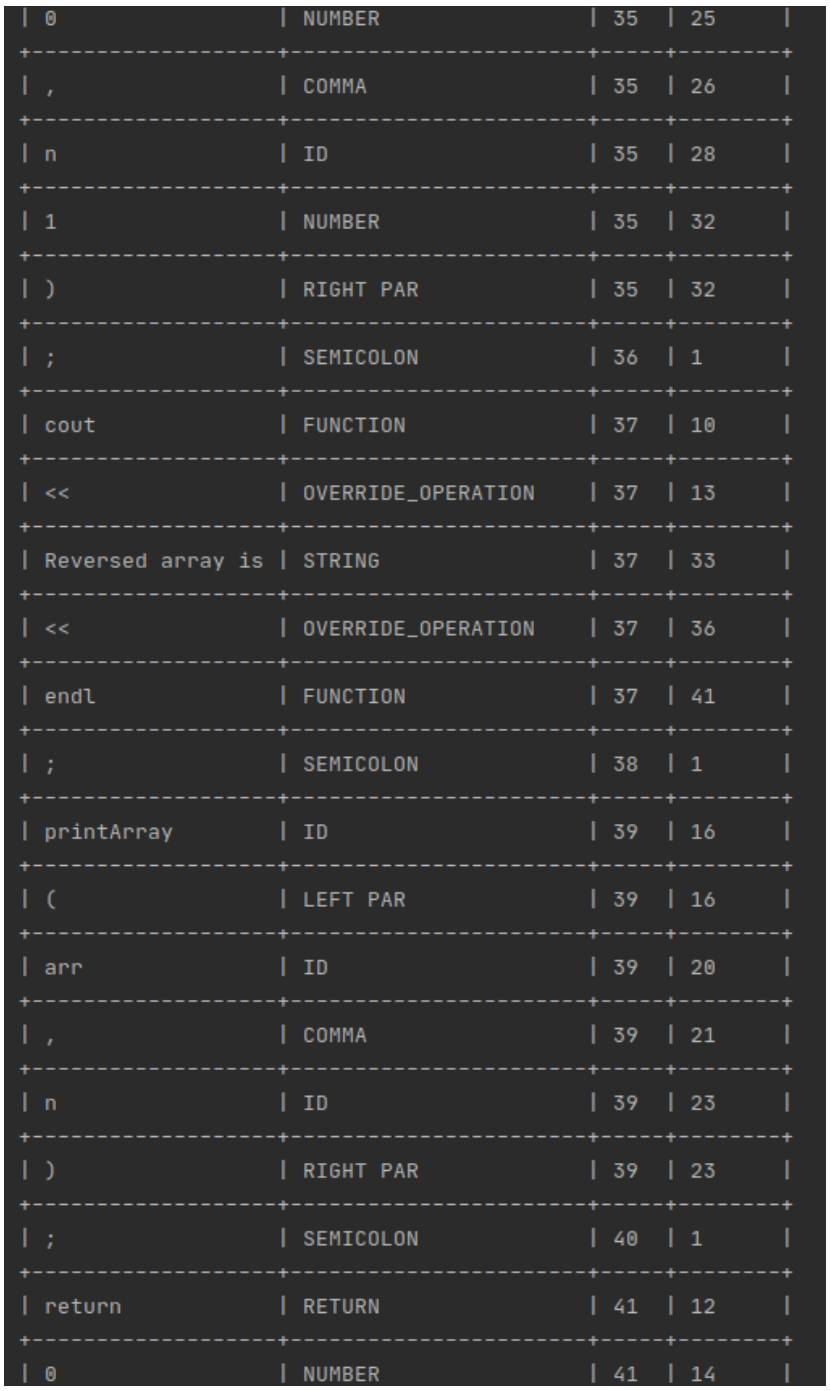
]	RIGHT SQUARE	29	14
=	EQUAL_SIGN	29	17
{	LEFT CURL	29	18
1	NUMBER	29	20
,	COMMA	29	21
2	NUMBER	29	23
,	COMMA	29	24
3	NUMBER	29	26
,	COMMA	29	27
4	NUMBER	29	29
,	COMMA	29	30
5	NUMBER	29	32
,	COMMA	29	33
6	NUMBER	29	35
}	RIGHT CURL	29	35
;	SEMICOLON	30	1
int	VARIABLE TYPE	31	9
n	ID	31	11
=	EQUAL_SIGN	31	13
sizeof	FUNCTION	31	20

Рис. 1.6. Результат работы анализатора — полная таблица (часть 6)



)	RIGHT PAR	31	24
/	ARITHMETIC_OPERATIONS	31	27
sizeof	FUNCTION	31	34
(	LEFT PAR	31	34
arr	ID	31	38
[	LEFT SQUARE	31	38
0	NUMBER	31	40
]	RIGHT SQUARE	31	40
)	RIGHT PAR	31	41
;	SEMICOLON	31	43
printArray	ID	33	16
(	LEFT PAR	33	16
arr	ID	33	20
,	COMMA	33	21
n	ID	33	23
)	RIGHT PAR	33	23
;	SEMICOLON	34	1
reverseArray	ID	35	18
(	LEFT PAR	35	18
arr	ID	35	22
,	COMMA	35	23

Рис. 1.7. Результат работы анализатора — полная таблица (часть 7)

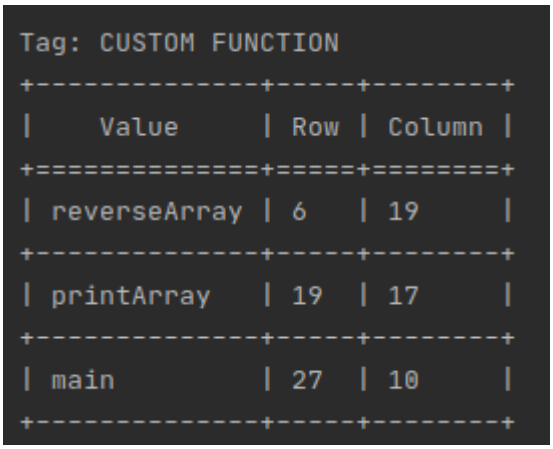


0	NUMBER	35	25
,	COMMA	35	26
n	ID	35	28
1	NUMBER	35	32
)	RIGHT PAR	35	32
;	SEMICOLON	36	1
cout	FUNCTION	37	10
<<	OVERRIDE_OPERATION	37	13
Reversed array is	STRING	37	33
<<	OVERRIDE_OPERATION	37	36
endl	FUNCTION	37	41
;	SEMICOLON	38	1
printArray	ID	39	16
(	LEFT PAR	39	16
arr	ID	39	20
,	COMMA	39	21
n	ID	39	23
)	RIGHT PAR	39	23
;	SEMICOLON	40	1
return	RETURN	41	12
0	NUMBER	41	14

Рис. 1.8. Результат работы анализатора — полная таблица (часть 8)

;	SEMICOLON	42	1
}	RIGHT CURL	42	2

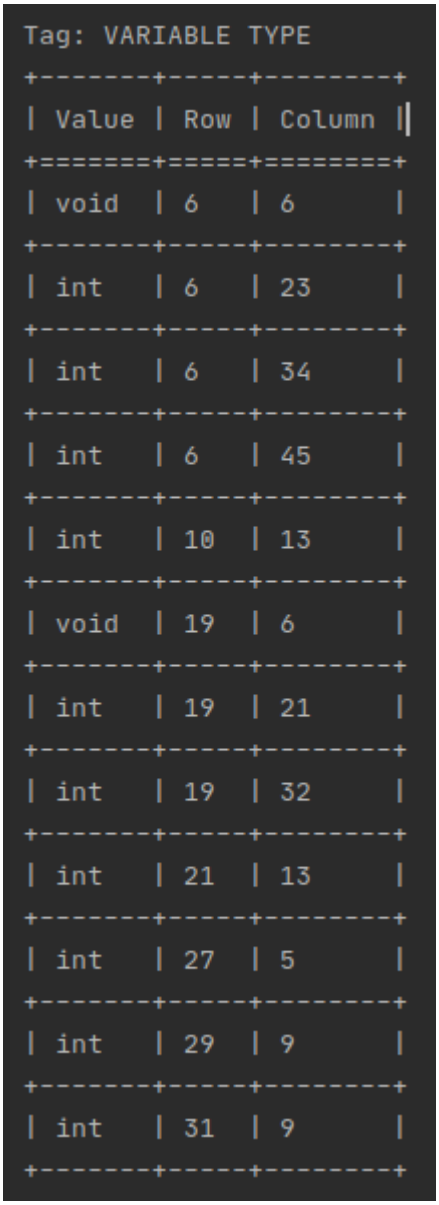
Рис. 1.9. Результат работы анализатора — полная таблица (часть 9)



Tag: CUSTOM FUNCTION

Value	Row	Column
reverseArray	6	19
printArray	19	17
main	27	10

Рис. 1.10. Результат работы анализатора — кастомные функции



Tag: VARIABLE TYPE

Value	Row	Column
void	6	6
int	6	23
int	6	34
int	6	45
int	10	13
void	19	6
int	19	21
int	19	32
int	21	13
int	27	5
int	29	9
int	31	9

Рис. 1.11. Результат работы анализатора — типы переменных

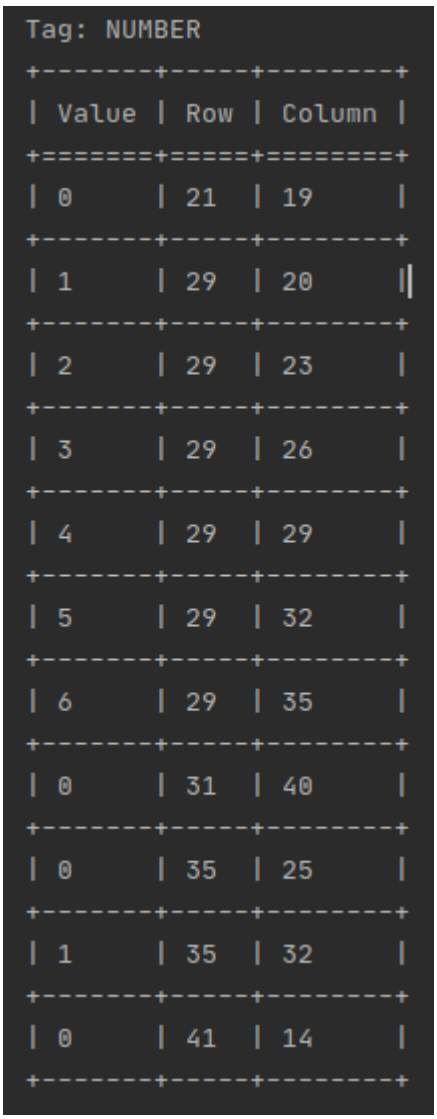


Рис. 1.12. Результат работы анализатора — числовые токены

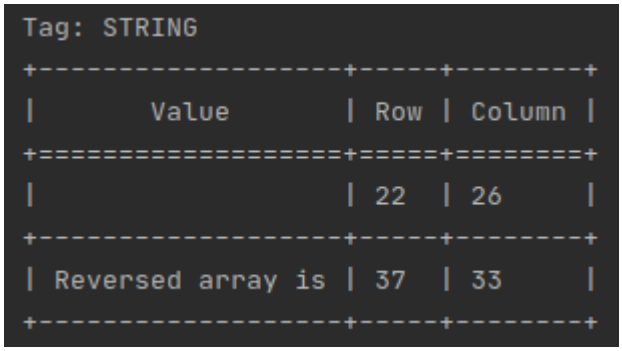
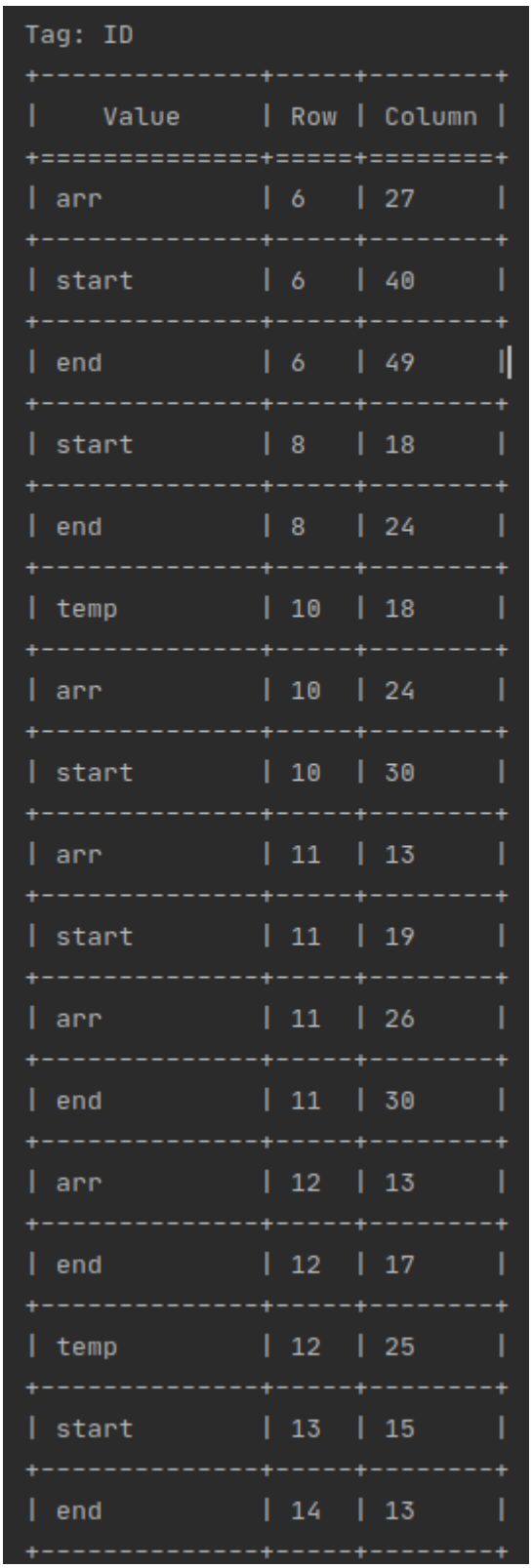


Рис. 1.13. Результат работы анализатора — строковые токены



Tag: ID

Value	Row	Column
arr	6	27
start	6	40
end	6	49
start	8	18
end	8	24
temp	10	18
arr	10	24
start	10	30
arr	11	13
start	11	19
arr	11	26
end	11	30
arr	12	13
end	12	17
temp	12	25
start	13	15
end	14	13

Рис. 1.14. Результат работы анализатора —переменные

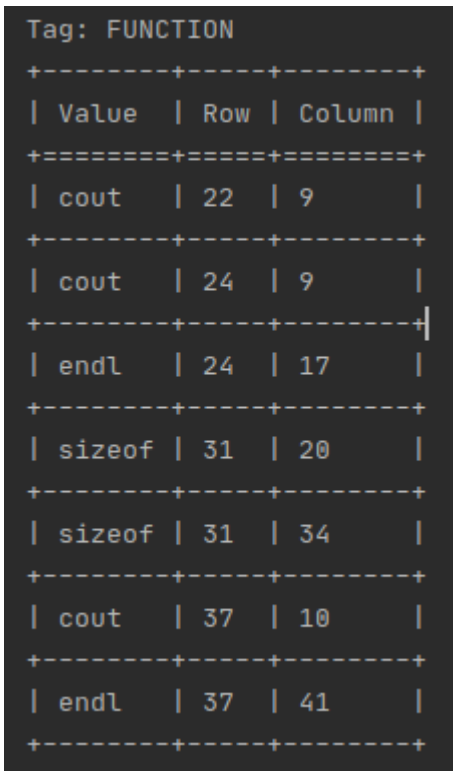


Рис. 1.15. Результат работы анализатора —функциональные токены

Код с ошибками

Рассмотрим тот же код программы с добавленными в него ошибок. При обнаружении их происходит вывод уведомления об ошибке.

Список ошибок:

```
float number = 12.24.12;
```

Рис. 2.1. Ошибка некорректного задания числа с плавающей точкой

```
wwhilee (start < end)
```

Рис. 2.2. Ошибка некорректного имени цикла while

```
arr[end] = temp
```

Рис. 2.3. Ошибка отсутствия символа конца строки „;“

```
start+++;
```

Рис. 2.4. Ошибка некорректного оператора инкремента

```
end----;
```

Рис. 2.5. Ошибка некорректного оператора декремента

```
iff (choice == true) {
    cout << "True result";
}
```

Рис. 2.6. Ошибка некорректного имени оператора if

```
int n === sizeof(arr) / sizeof(arr[0]);
```

Рис. 2.7. Ошибка некорректного оператора присваивания

```
Lexer errors:
    Incorrect format of number: "12.24.12" in line 8, column 29
    Undefined function type 'wwhilee' in line 9, column 13
    Missing end of line:  in line 13, column 24
    Incorrect format of operation: "+++" in line 14, column 18
    Incorrect format of operation: "----" in line 15, column 17
    Incorrect format of operation: "===" in line 32, column 15
    Undefined function type 'iff' in line 42, column 9
```

Рис. 2.8. Результат работы лексера (список ошибок)

Выводы

Я провёл анализ выбранного языка, сформировано его подмножество, составлена таблица токенов с информацией о их расположении в исходном коде и показаны 7 отловленные ошибки в коде программы на языке C++. Результатом работы программы на данном этапе является набор всех токенов, с указанием их места вхождения. Вниманию уделён момент, связанный с возможным лексическим ошибкам. В данной работе рассмотрены 7 возможных ошибки, успешно отлавливаемые лексическим анализатором.

Сложность лексического анализа в некоторой степени состоит в том, что для корректного определения текущей лексемы (или для определения ошибки) нам нужно понять какие символы идут за ней.

Лексический анализатор представляет собой первую фазу компилятора, его основная задача состоит в чтении входных символов исходной программы, их группировании в лексемы и вывод последовательностей токенов для всех лексем исходной программы. Поток токенов пересылается синтаксическому анализатору для разбора.

Приложение А - Код анализатора

```
class Token:
    """
    docstring for Token

    """

    def __init__(self, value, tag, row, col):
        self.value = value
        self.tag = tag
        self.row = row
        self.col = col

    def __str__(self):
        return "<{}, {}, {}, {}>".format(self.value, self.tag,
self.row, self.col)

    def __repr__(self):
        return self.__str__()


class Lexer(dict):
    """
    docstring for Lexer

    """

    def __init__(self, file, *args):
        super().__init__(*args)
        self.pos, self.row, self.col = 0, 1, 1
        self.skip_end = False
        self.variable_type_defined = False
        self.char = ""
        self.file = open(file, "r")
        self.string = self.file.readline()
        self.errors_list = list()

    def errors(self):
        """
        print all errors
```

```
        """
        import sys

        self.file.close()
        sys.stderr.write("Lexer errors:\n")

        for i in self.errors_list:
            sys.stderr.write("\t%s\n" % i)

        sys.stderr.flush()
        exit(1)

    def error(self, text):
        """
        print error

        """
        self.errors_list.append(
            "{} in line {}, column {}".format(text, self.row,
self.col)
        )

    def check_end_of_line(self, pos):
        result = True
        while pos > 0 and self.string[pos] == " ":
            pos -= 1

        if self.string[pos] == ";":
            result = False

        return result

    def empty_line(self):
        line = self.string

        for char in line[:-1]:
            if char != " ":
                return False

        return True

    def skip_line(self):
        self.string = self.file.readline()
        self.skip_end = False
```

```
        self.col = 1
        self.row += 1
        self.pos = 0

    def next_char(self):
        """
        set next char

        """
        if self.pos < len(self.string):
            self.char = self.string[self.pos]
            if self.char != "\n":
                self.col += 1
                self.pos += 1
            else:
                if self.check_end_of_line(self.pos - 1):
                    if not self.skip_end and not
self.empty_line():
                        self.error(f"Missing end of line: ")

                self.skip_line()
        else:
            self.char = "#0"

    def skip_space(self):
        """
        skip spaces

        """
        while self.char.isspace():
            self.next_char()

    @staticmethod
    def compare_signs(lexeme):
        possible_signs = {
            "=": EQUAL_SIGN,
            "==": EQUAL,
            "!=": NOT_EQUAL,
            "<": LT,
            ">": GT,
            "<=": LE,
            ">=": GE,
        }
```

```
        return possible_signs.get(lexeme, None)

    @staticmethod
    def arithmetics_function(lexeme):
        possible_signs = {
            "=": EQUAL_SIGN,
            "==": EQUAL,
            "!=": NOT_EQUAL,
            "<": LT,
            ">": GT,
            "<=": LE,
            ">=": GE,
        }

        return possible_signs.get(lexeme, None)

    @staticmethod
    def logical_operation(lexeme):
        possible_operation = {
            "&&": AND,
            "||": OR,
        }

        return possible_operation.get(lexeme, None)

    def is_build_in_function(self, lexeme):
        possible_func_names = {
            "if": IF,
            "else": ELSE,
            "while": WHILE,
            "for": FOR,
            "break": BREAK,
            "continue": CONTINUE,
            "return": RETURN,
            "printf": FUNC,
            "getchar": FUNC,
            "endl": FUNC,
            "cout": FUNC,
            "sizeof": FUNC,
        }

        if not self.variable_type_defined:
            return possible_func_names.get(lexeme, None)
        else:
```

```
            self.error(f"Undefined function type: {lexeme}")

    def is_function(self):
        if self.string[self.pos - 1] == "(":
            self.variable_type_defined = False
            return True

        return False

    def number_conversion(self, lexeme=""):
        """
        Parsing numbers: float or integer, catch incorrect
number input
        :param lexeme: str
        :return: Token
        """
        if self.char.isdigit():
            count = 0
            sign = 1 if lexeme == "+" or lexeme == "" else -1

            while self.char.isdigit() or self.char == ".":
                if self.char == ".":
                    count += 1

                lexeme += self.char
                self.next_char()

            if count > 1:
                self.error('Incorrect format of number: "%s"' %
lexeme)
                return None
            else:
                return Token(
                    sign * (int(lexeme)) if count == 0 else sign
* (float(lexeme)),
                    NUMBER,
                    self.row,
                    self.col,
                )

    def check_names(self, lexeme):
        """
        Parsing name. Defining functions, variables, build-in
functions
```

```
        :param lexeme: str
        :return: Token
        """

        token = None

        if self.variable_type_defined:
            if self.is_function():
                self.variable_type_defined = False
                token = Token(lexeme, FUNC_DECLARATION,
self.row, self.col)
        else:
            if (func_type := self.is_build_in_function(lexeme))
is not None:
                token = Token(lexeme, func_type, self.row,
self.col)
            elif lexeme in VARIABLE_TYPES:
                self.variable_type_defined = True
                token = Token(lexeme, TYPE, self.row, self.col)

        return token

    def check_operation(self, lexeme):
        """
        Parsing different operations
        :param lexeme: str
        :return: Token
        """

        token = None

        if lexeme in ARITHMETIC_OPERATIONS:
            token = Token(lexeme, "ARITHMETIC_OPERATIONS",
self.row, self.col)
        elif lexeme in OVERRIDE_OPERATION:
            token = Token(lexeme, "OVERRIDE_OPERATION",
self.row, self.col)
        elif (logical_operation :=
self.logical_operation(lexeme)) is not None:
            token = Token(lexeme, logical_operation, self.row,
self.col)
        elif (sign_type := self.compare_signs(lexeme)) is not
None:
            token = Token(lexeme, sign_type, self.row, self.col)
```

```
        return token

    def processing_bracket(self, bracket, *, skip_end=False):
        """
        Return Token for different types of brackets
        :param bracket: str
        :param skip_end: bool
        :return: Token
        """

        possible_brackets = {
            "(": L_PAR,
            ")": R_PAR,
            "[": L_SQUARE,
            "]": R_SQUARE,
            "{": L_CURL,
            "}": R_CURL,
        }

        if skip_end:
            self.skip_end = True

        return Token(bracket, possible_brackets[bracket],
self.row, self.col)

    def check_brackets(self):
        """
        Check which type of brackets is used
        :return: Token
        """

        lexeme = self.char
        token = None

        if lexeme in ("(", ")"):
            token = self.processing_bracket(lexeme,
skip_end=True)
        elif lexeme in ("[", "]"):
            token = self.processing_bracket(lexeme)
        elif lexeme in ("{", "}"):
            token = self.processing_bracket(lexeme,
skip_end=True)
```

```
            self.next_char()

            return token

        def next_token(self):
            """
            Parsing code file and getting tokens
            :return: Token
            """

            self.skip_space()
            lexeme = ""

            if self.char.isalpha() or self.char == "_":
                lexeme = self.char
                self.next_char()

                while self.char.isalpha() or self.char.isdigit():
                    lexeme += self.char
                    self.next_char()

                if (token := self.check_names(lexeme)) is not None:
                    return token

                if not self.variable_type_defined:
                    pos = self.pos

                    while self.string[pos] == " ":
                        pos += 1

                    if self.string[pos] == "(":
                        self.error(f"Undefined function type
'{lexeme}'")
                        return None

                self.variable_type_defined = False
                return Token(lexeme, ID, self.row, self.col)

            elif self.char in "+-*%><=^!?&|":
                lexeme, count = self.char, 1
                self.next_char()

                while self.char in "+-*%><=^!?&|":
                    lexeme += self.char
```

```
                count += 1
                self.next_char()

            if count > 2:
                self.error('Incorrect format of operation: "%s"'
% lexeme)
                return None
            else:
                if lexeme in ("-", "+"):
                    sign = lexeme
                    self.next_char()

                    return self.number_conversion(sign)

                elif (token := self.check_operation(lexeme)) is
not None:
                    return token

            self.error('Undefined operation: "%s"' % lexeme)

        elif self.char.isdigit():
            return self.number_conversion()

        elif self.char in ("(", ")", "{", "}", "[", "]"):
            return self.check_brackets()

        elif self.char == "#0":
            return Token("EOF", None, self.row, self.col)

        elif self.char == "/":
            lexeme = self.char
            self.next_char()
            if self.char in ("/", "*"):
                return self.skip_comments("\n" if self.char ==
"/" else "/")

            return Token(lexeme, "ARITHMETIC_OPERATIONS",
self.row, self.col)

        elif self.char in ('"', "'"):
            character, count = self.char, 0
            self.next_char()

            while self.char != character:
```

```
                count += 1
                condition, lexeme = self.parse_line_end(lexeme)
                if condition:
                    continue

                lexeme += self.char
                self.next_char()

            self.next_char()

            if character == "'":
                if count == 1:
                    return Token(lexeme, CHAR, self.row,
self.col)
            elif character == '"':
                return Token(lexeme, STRING, self.row, self.col)

            self.error("Incorrect quotes: '%s'" % lexeme)

        elif self.char in (";", ","):
            lexeme = self.char
            self.next_char()
            return Token(
                lexeme, SEMICOLON if lexeme == ";" else COMMA,
self.row, self.col
            )

        elif self.char == "\n":
            self.pos -= 1
            self.col -= 1
            self.next_char()
            return None

        elif self.char in self:
            lexeme = self.char
            self.next_char()
            return Token(lexeme, self[lexeme], self.row,
self.col)

        else:
            lexeme = self.char
            self.error('Unknown character: "%s"' % self.char)
            self.next_char()
            return Token(lexeme, UNKNOWN, self.row, self.col)
```

```
        return None

    def parse_line_end(self, lexeme):
        """
        Parsing symbol of line end inside C++ char or string
types
        :param lexeme: str
        :return: (bool, str)
        """

        if self.char == "\\":
            lexeme += self.char
            self.next_char()
            lexeme += self.char
            self.next_char()

            return True, lexeme

        return False, lexeme

    def skip_comments(self, char):
        """
        Base of condition skipping line content in comment
        :param char: str - comment end character
        :return: Token()
        """

        self.skip_end = True

        while self.char != char:
            self.next_char()

        self.next_char()

        return self.next_token()

    def get_token(self):
        """
        Returning token
        :return: Token
        """

        self.next_char()
```

```
        while True:
            result = self.next_token()

            if not result:
                continue

            if result.value == "EOF":
                break

            yield result

    def tokens(self):
        """
        Returning list of parsing tokens
        :return: list
        """

        result = [i for i in self.get_token()]
        return result

    def raw_input(self, user_string):
        """
        Return raw user input
        :param user_string: str
        :return: list
        """

        self.string = user_string
        return self.tokens()


def draw_tags_groups(tokens):
    tokens_copy = copy.deepcopy(tokens)
    tokens_copy.sort(key=lambda x: x.tag)
    tag_names = {*[token.tag for token in tokens_copy]}
    tables = []
    for name in tag_names:
        table = tt.Texttable()
        table.header(["Value", "Row", "Column"])
        for token in filter(lambda x: x.tag == name,
tokens_copy):
            table.add_row((token.value, token.row, token.col))

        tables.append((name, table))
```

```
    for name, table in tables:
        print("Tag:", name)
        print(table.draw())
        print()


def draw_result_table(tokens):
    tab = tt.Texttable()
    headings = ["Value (token)", "Tag", "Row", "Column"]
    tab.header(headings)

    values = list()
    tags = list()
    rows = list()
    columns = list()

    for token in tokens:
        values.append(token.value)
        tags.append(token.tag)
        rows.append(token.row)
        columns.append(token.col)

    for row in zip(values, tags, rows, columns):
        tab.add_row(row)
    s = tab.draw()
    print(s)


def check_if_main_exist(tokens):
    return list(
        filter(lambda x: x.tag == FUNC_DECLARATION and x.value
== "main", tokens)
    )


if __name__ == "__main__":
    path = "main_1.cpp"
    lexer = Lexer(path)
    tokens = lexer.tokens()
    # if len(check_if_main_exist(tokens)) == 0:
    #     lexer.error("Program should have function 'main'")

    if lexer.errors_list:
```

```
    lexer.errors()

draw_result_table(tokens)
draw_tags_groups(tokens)
```