Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

КАФЕДРА ИНФОРМАТИКИ

Отчёт по лабораторной работе №2

“Лексический анализ”

Выполнил:

студент гр. 853504

Кузьма В. В.

Проверил:

Ст. преподаватель КИ Шиманский В. В.

Минск 2021

**Содержание**

[**1. Цель работы 3**](#_Toc64740995)

[**2. Краткие теоретические сведения 4**](#_Toc64740996)

[**3. Выделение лексем (токенов) 5**](#_Toc64741001)

**4. Лексические ошибки программы 9**

[**Приложение. Текст программ 10**](#_Toc64741002)

# **Цель работы**

# Разработка лексического анализатора подмножества языка программирования, определённого в лабораторной работе 1. Программа анализа определяет лексические правила и выполняет перевод потока символов программ лабораторной работы 1в поток лексем (токенов).

На вход программы подается текстовый файл, содержащий строки текста программы.

Например, строка присваивания переменной значения арифметического выражения в виде:

**ПЕРЕМЕННАЯ = ВЫРАЖЕНИЕ**

Выражение может включать:

* **Знаки сложения и умножения** («+» и «\*»);
* **Круглые скобки** («(» и «)»);
* **Константы** (например, 5; 3.8; 1e+18, 8.41E–10);
* **Имена переменных**.

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

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

Проанализируем выражение:

* ***COST***, ***PRICE*** и ***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.

# **Краткие теоретические сведения**

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. Обычно при работе лексический анализатор взаимодействует также с таблицей символов. Когда лексический анализатор встречается с лексемой, составляющей идентификатор, эту лексему требуется внести в таблицу символов. В некоторых случаях лексический анализатор может получать из таблицы символов некоторую информацию об идентификаторах, которая может помочь ему верно определить передаваемый синтаксическому анализатору токен.

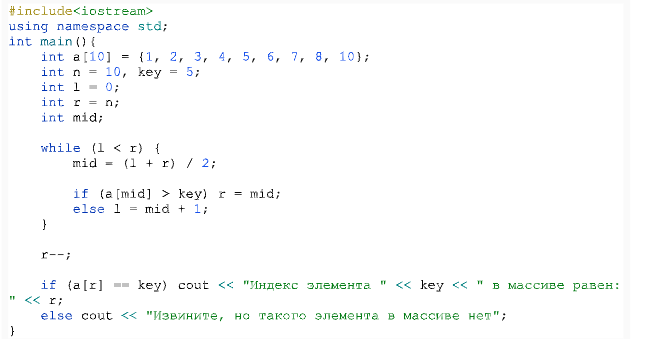
Обычно взаимодействие реализуется как вызов лексического анализатора синтаксическим анализатором. Этот вызов, представленный как команда getNextToken, заставляет лексический анализатор читать символы из входного потока, пока он не сможет идентифицировать очередную лексему и вернуть синтаксическому анализатору корректный токен.

Поскольку лексический анализатор является частью компилятора, которая читает исходный текст, он может заодно выполнять и некоторые другие действия, помимо идентификации лексем. Одной из таких задач является отбрасывание комментариев и пробельных символов (пробел, символы табуляции и новой строки, а также, возможно, некоторые другие символы, использующиеся для отделения токенов друг от друга во входном потоке). Еще одной задачей является синхронизация сообщений об ошибках, генерируемых компилятором, с исходной программой. Например, лексический анализатор может отслеживать количество символов новой строки, чтобы каждое сообщение об ошибке сопровождалось номером строки, в которой она обнаружена. В некоторых компиляторах лексический анализатор создает копию исходной программы с сообщениями об ошибках, вставленными в соответствующие места исходного текста.

Если исходная программа использует макропрепроцессор, то раскрытие макросов также может выполняться лексическим анализатором

# **Выделение лексем (токенов)**

Пример бинарного поиска из 1-ой лабораторной работы:



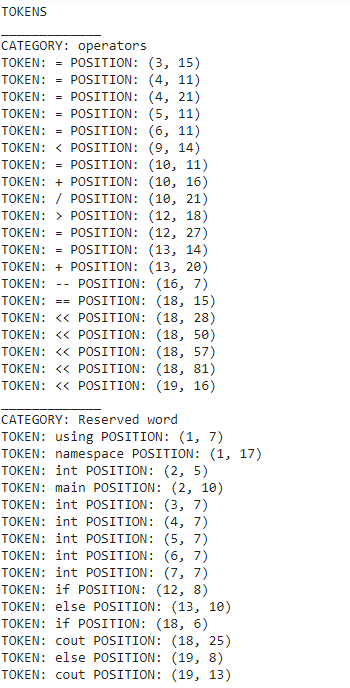


Рис 1. Пример работы программы без ошибок

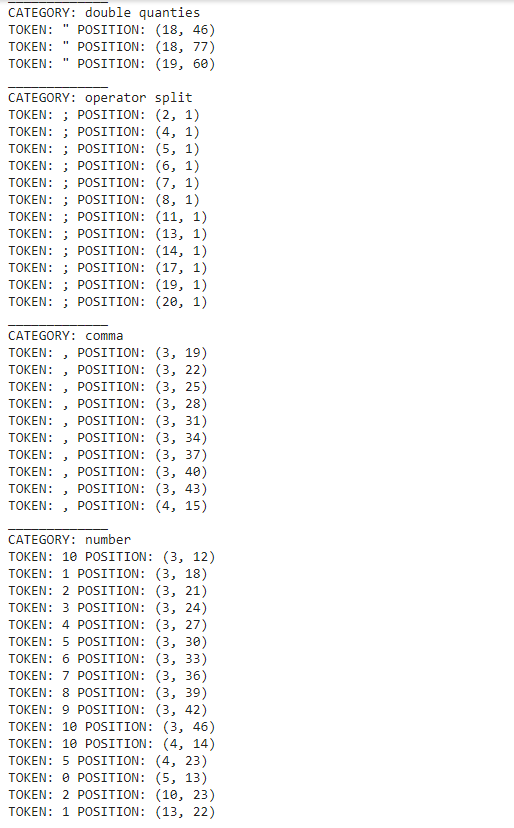


Рис 2. Пример работы программы без ошибок

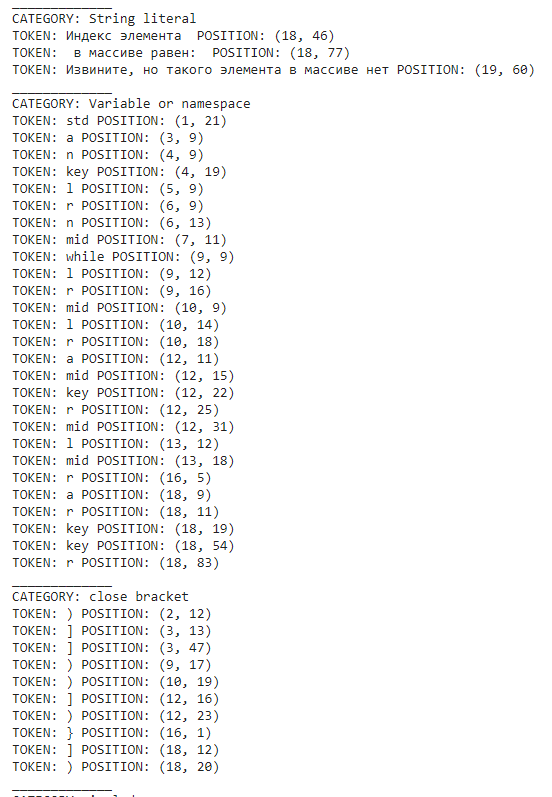


Рис 3. Пример работы программ без ошибок

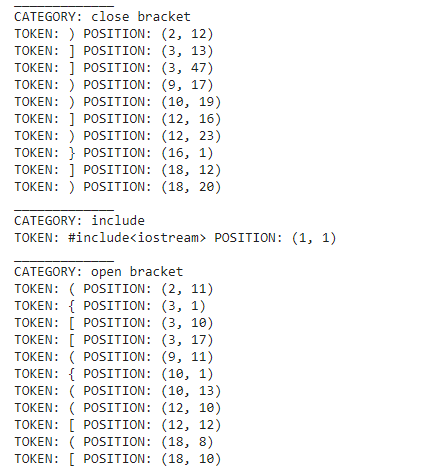


Рис 4. Пример работы программы без ошибок

1. **Лексические ошибки программы**

Добавим некоторые ошибки в программу для просмотра некоторых ошибочных ситуаций.

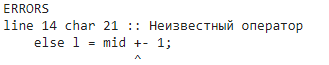


Рис 5. Недопустимый оператор +-

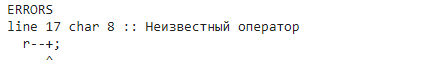


Рис 6. Недопустимый оператор --+

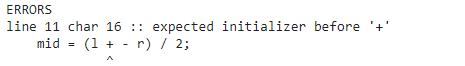


Рис 7. Недопустимо смешение операторов



Рис 8. Недопустимо смешение операторов



Рис 9. Недопустимый оператор ;

**Приложение. Текст программ**

def tokenize(token, category, tokens):

  if len(token) == 0 or category == "whitespace" or "\n" in token or " " in token:

    pass

  elif category == "letter":

    tokenize\_letter(token, tokens)

  else:

    tokens.append({"token": token, "category": category, "row": get\_row(), "column": get\_column()})

def get\_row():

  global row

  return row

def get\_column():

  global col

  return col + 1

def tokenize\_letter(token, tokens):

  RESERVED\_LIST = ['using', 'namespace', 'int', 'main', 'if', 'else', 'cin', 'cout', 'return', 'for', 'struct']

  if token in RESERVED\_LIST:

    tokens.append({"token": token, "category": "Reserved word", "row": get\_row(), "column": get\_column()})

  else:

    tokens.append({"token": token, "category": "Variable or namespace", "row": get\_row(), "column": get\_column()})

def tokenize\_directiva(file, tokens):

  global col

  if col == 1:

    token = '#'

    c = 'q'

    while c != '\n':

      c = read\_symbol(file)

      token += c

    token = token[:-1]

    tokens.append({"token": token, "category": "include", "row": get\_row() - 1, "column": get\_column()})

def tokenize\_string(file, tokens):

  token = ''

  c = 'q'

  while c != '"':

    c = read\_symbol(file)

    token += c

  token = token[:-1]

  tokens.append({"token":token, "category": "String literal", "row": get\_row(), "column": get\_column()})

  tokens.append({"token":'"', "category": "double quanties", "row": get\_row(), "column": get\_column()})

def tokenize\_char(file, tokens):

  token = ''

  c = 'q'

  while c != '"':

    c = read\_symbol(file)

    token += c

  token = token[:-1]

  tokens.append({"token":token, "category": "char literal", "row": get\_row(), "column": get\_column()})

  tokens.append({"token":'"', "category": "single quanties", "row": get\_row(), "column": get\_column()})

def get\_category(c):

  if c.isnumeric():

    return "number"

  if c == '#':

    return "Directiva"

  if c.isalpha():

    return "letter"

  if c == ' ':

    return "whitespace"

  if c == ';':

    return "operator split"

  if c == ',':

    return 'comma'

  if c == '.':

    return 'dot'

  if c in "([{":

    return "open bracket"

  if c in ")]}":

    return "close bracket"

  if c in "+/=<>-%":

    return "operators"

  if c == '"':

    return 'double quanties'

  if c == "'":

    return 'single quanties'

  return 'Unknown'

def read\_symbol(file):

  global pos, row, col

  c = file.read(1)

  if  c == '\n':

    row += 1

    col = 0

  elif c:

    col += 1

  else:

    return 0

  return c

operators = ['=', '<', '>', '+', '-', '<<', '==', '--', '++', '/']

def errorizer(lines, tokens):

  print("ERRORS")

  for i in range(1, len(tokens)):

    if tokens[i]['category'] == 'operators' and tokens[i]['token'] not in operators:

      print(f"line {tokens[i]['row']} char {tokens[i]['column']} :: Неизвестный оператор")

      print(lines[tokens[i]['row'] - 1][:-1])

      print(' ' \*  tokens[i]['column'] + '^')

    elif tokens[i]['category'] == tokens[i - 1]['category'] and tokens[i]['category'] != 'Reserved word':

      print(f"line {tokens[i - 1]['row']} char {tokens[i - 1]['column']} :: expected initializer before '{tokens[i - 1]['token']}'")

      print(lines[tokens[i]['row'] - 1][:-1])

      print(' ' \* tokens[i - 1]['column'] + '^')

  for i in range(len(lines)):

    if len(lines[i]) == 1:

      continue

    if lines[i][-2] == ';' and (lines[i][-3] == '{' or lines[i][-3] == '}' or lines[i][-3] == '>'):

      print(f"line {i + 1} char {len(lines[i])} :: Unxpected ';'")

      print(lines[tokens[i]['row']][:-1])

      print(' ' \* (len(lines[i]) - 2) + '^')

    if lines[i][-2] != ';' and (lines[i][-3] != '{' and lines[i][-3] != '}' and lines[i][-3] != '>'):

      print(f"line {i + 1} char {len(lines[i])} :: Expected ';'")

      print(lines[tokens[i]['row']][:-1])

      print(' ' \* (len(lines[i]) - 2) + '^')

col, row = 0, 1

file = open("main.cpp")

f = open("main.cpp")

Lines = f.readlines()

tokens = []

pattern = ""

pred\_category, category = "", ""

while True:

  c = read\_symbol(file)

  if c == 0:

    break

  pred\_category = category

  category = get\_category(c)

  if category == "Directiva":

    tokenize\_directiva(file, tokens)

    continue

  elif category == "double quanties":

    tokenize\_string(file, tokens)

    continue

  elif category == "single quanties":

    tokenize\_char(file, tokens)

    continue

  elif category == "":

    tokenize\_string(file, tokens)

  elif pred\_category != category:

    pass

    tokenize(pattern, pred\_category, tokens)

    pattern = ""

  pattern += c

errorizer(Lines, tokens)

categories = []

for token in tokens:

  categories.append(token['category'])

categories = set(categories)

print('TOKENS')

for category in categories:

  print("\_\_\_\_\_\_\_\_\_\_\_\_\_")

  print(f"CATEGORY: {category}")

  for token in tokens:

    if token['category'] == category:

      print(f"TOKEN: {token['token']} POSITION: ({token['row']}, {token['column']})")