**Федеральное агентство по образованию**Государственное образовательное учреждение высшего профессионального   
образования **«Тихоокеанский Государственный университет»**

Факультет компьютерных и фундаментальных наук

Кафедра ПОВТАС

**Лабораторная работа №1**

по дисциплине: «Теория и практика ЯП»

на тему: «Построение программы лексического анализатора»  
Вариант №4

Выполнил: студент группы ПИИ(м)-21

Забавин А.С.

Проверил: кандидат физико-математических наук, доцент ка­федры ПОВТАС

Вихтенко Э.М.

# Постановка задачи

На вход программы подается файл, содержащий тест на входном языке программирования. Результатом работы программы должен быть файл, содержащий последовательность кодов лексем входной программы, а также один или несколько файлов, содержащие все таблицы лексем.

Отчет по работе должен содержать описание правил записи перечисленных выше элементов заданного входного языка, стандартные таблицы лексем, диаграмма состояний сканера, листинг программы и комментарии к нему, пример лексического разбора.

Каждый вариант задания представляет собой пару: входной язык и машинный, или выходной, язык. В качестве входного языка предлагается один из языков программирования высокого уровня, в который должно быть обязательно включено следующее подмножество языковых конструкций и операторов:

* идентификаторы;
* числовые константы целого типа и вещественного типа,
* представленные с фиксированной и плавающей точкой;
* символьные (строковые) константы;
* переменные с индексами (массивы и элементы массивов);
* комментарии (строчные и блочные);
* имена функций пользователя;
* арифметические операции;
* операции сравнения (меньше, больше, равно, не равно, меньше или
* равно, больше или равно);
* операторы описания данных (идентификаторов и массивов);
* операторы описания процедур и функций (если предусмотрены в
* языке);
* операторы условного и безусловного перехода;
* метки (если предусмотрены в языке).

# Краткая теория

Лексический анализ всегда предшествует синтаксическому анализу и заключается в предварительной обработке программы и ее перекодировании к виду, удобному для синтаксического анализа. Лексический анализ принято отделять от синтаксического анализа для сокращения времени компиляции, поскольку синтаксис (морфология) слов всегда проще, чем синтаксис программ.

Лексический анализ сводится к разбиению текста программы на лексические единицы (слова, лексемы) – конструкции, которые выступают в качестве терминальных символов для синтаксического анализа. Лексемы еще иногда называют символами или атомами.

Большинство лексем в языках программирования можно сгруппировать в классы (см. Таблица 1).

Таблица . Классы лексем подмножества языка PL/1

|  |  |
| --- | --- |
| **Типы лексем** | **Обозначение** |
| Служебные слова | W |
| Идентификаторы | I |
| Операции | O |
| Разделители | R |
| Константы | N |
| C |

Для оптимизации и ускорения поиска данных часто таблицу констант разделяют на две: таблицу N (для числовых констант) и таблицу С (для символьных констант).

Лексема представляется в виде пары (класс, значение). Значение может быть непосредственным (литеральным) или представлять собой ссылку на некоторую таблицу, в которой хранятся значения лексем.

Таблицы служебных слов, разделителей и операций определяются входным языком и должны быть сформированы при построении сканера, они являются постоянными и неизменяемыми. Таблицы идентификаторов и констант являются временными и создаются непосредственно в процессе лексического разбора исходной программы

Лексический анализатор представляет собой процессор или, иначе говоря, сканер (конечный автомат) для разбора и классификации лексем, связанный с некоторыми семантическими процедурами. На вход такого процессора последовательно подаются символы исходной программы, причем каждый входной символ вызывает изменение состояния сканера. Если анализируемый символ означает конец разбираемой лексемы, то с переходом связывается некоторая семантическая процедура, позволяющая либо определить код лексемы по таблице (в случае служебных слов, разделителей, операций), либо пополнить таблицы (для констант и идентификаторов), а затем в выходную строку выдать очередной код лексемы во внутреннем представлении.

Для эффективного разбора и оптимизации сканера, а также для его программной реализации в алфавите автомата выделяют несколько подмножеств символов, по которым сканер выполняет одинаковые переходы и действия.

# Результаты работы

В качестве исходного был выбран язык программирования C++. Первым делом, была построена диаграмма состояний сканера (см. Рис. 1).

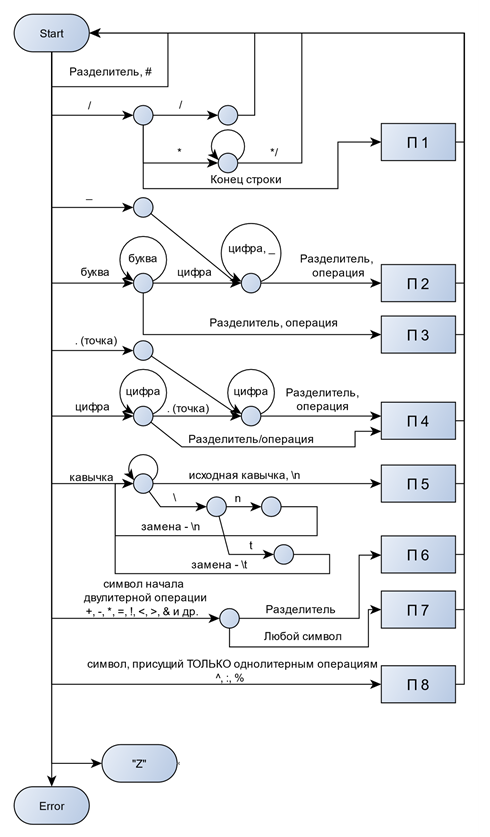


Рис. . Диаграмма состояний сканера

Были определены на языке программирования Python3 постоянные таблицы, используемые лексическим анализатором

Код программы представлен на листингах 1 и 2.

class **\_StackQueue**:

def **\_\_new\_\_**(cls, \*args, \*\*kwargs):

if cls is \_StackQueue:

raise TypeError(*'От этого класса нельзя наследоваться напрямую, используйте субклассы'*)

return super().\_\_new\_\_(cls)

def **\_\_init\_\_**(*self*, initlist=None):

*self*.\_items = []

if initlist is not None:

for x in initlist:

*self*.add(x)

*@property*

def **is\_empty**(*self*):

if *self*.\_items and len(*self*.\_items):

return False

return True

def **pop**(*self*):

if *self*.is\_empty:

return None

else:

return *self*.\_items.pop()

def **clear**(*self*):

del *self*.\_items

*self*.\_items = []

def **\_\_len\_\_**(*self*):

return len(*self*.\_items)

len = property(lambda s: len(s))

def **\_\_str\_\_**(*self*):

return *self*.read()

class **CharStack**(\_StackQueue):

is\_stack = True

def **top**(*self*):

try:

return *self*.\_items[-1]

except:

pass

return *''*

def **bottom**(*self*):

try:

return *self*.\_items[0]

except:

pass

return *''*

def **add**(*self*, value):

*self*.\_items.append(str(value))

def **read**(*self*):

return *''*.join(*self*.\_items)

def **\_\_iter\_\_**(*self*):

return *self*

def **\_\_next\_\_**(*self*):

x = *self*.pop()

if x and x != *''*:

return x

raise StopIteration

class **CharQueue**(\_StackQueue):

is\_stack = False

def **first**(*self*):

try:

return *self*.\_items[-1]

except:

pass

return *''*

def **last**(*self*):

try:

return *self*.\_items[0]

except:

pass

return *''*

def **add**(*self*, value):

*self*.\_items.insert(0, str(value))

def **read**(*self*):

return *''*.join(reversed(*self*.\_items))

def **\_\_iter\_\_**(*self*):

return *self*

def **\_\_next\_\_**(*self*):

x = *self*.pop()

if x and x != *''*:

return x

raise StopIteration

class **VarStack**(\_StackQueue):

is\_stack = True

def **top**(*self*):

try:

return *self*.\_items[-1]

except:

pass

return None

def **bottom**(*self*):

try:

return *self*.\_items[0]

except:

pass

return None

def **add**(*self*, value):

*self*.\_items.append(value)

def **read**(*self*):

return *''*.join(map(repr, *self*.\_items))

def **\_\_iter\_\_**(*self*):

return *self*

def **\_\_next\_\_**(*self*):

x = *self*.pop()

if x is not None:

return x

raise StopIteration

class **VarQueue**(\_StackQueue):

is\_stack = False

def **first**(*self*):

try:

return *self*.\_items[-1]

except:

pass

return None

def **last**(*self*):

try:

return *self*.\_items[0]

except:

pass

return None

def **add**(*self*, value):

*self*.\_items.insert(0, value)

def **read**(*self*):

return *''*.join(map(repr, *self*.\_items))

def **\_\_iter\_\_**(*self*):

return *self*

def **\_\_next\_\_**(*self*):

x = *self*.pop()

if x is not None:

return x

raise StopIteration

Листинг 1. Файл shared\_classes.py определяющий структуры стека и очреди

class **Lexem**:

LEXEM\_CLASSES = {

*'W'*: *'Служебные слова'*,

*'I'*: *'Идентификаторы'*,

*'O'*: *'Операции'*,

*'R'*: *'Разделители'*,

*'N'*: *'Константы числовые'*,

*'C'*: *'Константы символьные'*,

*'?'*: *'Другие лексемы'*,

}

def **\_\_init\_\_**(*self*, class\_code: str, lex\_id: int=-1, text: str=*''*, order: int=-1, linen: int=0, charn: int=0, slinen: int=0, scharn: int=0):

if class\_code not in Lexem.LEXEM\_CLASSES.keys():

raise ValueError(*'Неподходящий код класса лексемы'*)

*self*.class\_code = class\_code

*self*.lex\_id = lex\_id

*self*.text = text

*self*.order = order

*self*.linen = linen

*self*.charn = charn

*self*.slinen = slinen

*self*.scharn = scharn

*@property*

def **fcode**(*self*):

return *f'{self.class\_code}{self.lex\_id}'*

*@property*

def **fdesc**(*self*):

return *f'{Lexem.LEXEM\_CLASSES[self.class\_code]}'*

def **\_\_repr\_\_**(*self*):

clsname = *self*.\_\_class\_\_.\_\_name\_\_

return *f'<{clsname}: {self.fcode}>'*

def **\_\_str\_\_**(*self*):

return *f'"{self.text}" ({self.fcode}: {self.fdesc})'*

class **LexicalAnalyzerC**:

STATES = [

*'S'*, # S - Start state - Начальный статус по умолчанию

*'Z'*, # Z - sucsess end state - Статус успешного завершения

*'F'*, # F - Fail (error) state - Статус ошибки

*'OLC'*, # OLC - One-Line Comment state - Статус однострочного комментария

*'MLC'*, # MLC - Multi-Line Comment state - Статус многострочного комментария

*'MCOP'*, # MCOP - Multi-Char OPerators - Статус многосимвольного оператора

*'LETT'*, # LETT - LETTers state - Статус букв, которые могут быть как идентификаторами переменных так и служебными словами

*'IDENT'*, # IDENT - IDENTifiers state - Статус идентификаторов переменных

*'INT'*, # INT - INTeger state - Статус целых чисел

*'REAL'*, # REAL - REAL (float) state - Статус вещественных чисел

*'CHAR'*, # CHAR - CHAR constants state - Статус строковых констант

]

EXCEPT\_CHARS = (*' '*, *'\n'*, *'\t'*, *''*)

TABLE\_SPEC\_WORDS = {

*'char'*: 0,

*'double'*: 1,

*'float'*: 2,

*'int'*: 3,

*'string'*: 4,

*'const'*: 5,

*'long'*: 6,

*'short'*: 7,

*'signed'*: 8,

*'unsigned'*: 9,

*'void'*: 10,

*'auto'*: 11,

*'if'*: 12,

*'else'*: 13,

*'case'*: 14,

*'switch'*: 15,

*'for'*: 16,

*'while'*: 17,

*'do'*: 18,

*'break'*: 19,

*'continue'*: 20,

*'return'*: 21,

*'goto'*: 22,

*'using'*: 23,

*'namespace'*: 24,

}

TABLE\_SEPARATORS = {

*'('*: 0,

*')'*: 1,

*';'*: 2,

*'['*: 3,

*']'*: 4,

*'{'*: 5,

*'}'*: 6,

*','*: 7,

# '.': 8,

}

TABLE\_OPERATIONS = {

*'+'*: 0,

*'++'*: 1,

*'-'*: 2,

*'--'*: 3,

*'\*'*: 4,

*'\*\*'*: 5,

*'^'*: 6,

*'/'*: 7,

*'%'*: 8,

*'=='*: 9,

*'!='*: 10,

*'<'*: 11,

*'>'*: 12,

*'<='*: 13,

*'>='*: 14,

*'&'*: 15,

*'&&'*: 16,

*'||'*: 17,

*'!'*: 18,

*'<<'*: 19,

*'>>'*: 20,

*':'*: 21,

*'='*: 22,

*'::'*: 23,

*'.'*: 24,

}

SINGLE\_CHAR\_OPERATIONS = {k: v for k, v in TABLE\_OPERATIONS.items() if len(k) == 1}

MULTY\_CHAR\_OPERATIONS = {k: v for k, v in TABLE\_OPERATIONS.items() if len(k) > 1}

IMPLICIT\_SEPARATORS = list(set(list(TABLE\_SEPARATORS.keys()) + list(map(lambda x: x[0], TABLE\_OPERATIONS.keys())) + list(EXCEPT\_CHARS)))

*@staticmethod*

def **is\_left\_op**(optext):

*"""*

*Левосторонний оператор*

*https://ru.wikipedia.org/wiki/Операторы\_в\_C\_и\_C++*

**:param** *optext:*

*"""*

if optext in (*'++'*, *'--'*, *'()'*, *'[]'*, *'.'*, *'->'*, *'.\*'*, *'->\*'*, *'\*'*, *'/'*, *'%'*, *'+'*,

*'-'*, *'<<'*, *'>>'*, *'<'*, *'<='*, *'>'*,

*'>='*, *'=='*, *'!='*, *'&'*, *'^'*, *'|'*, *'&&'*, *'||'*, *','*, ):

return True

return False

*@staticmethod*

def **is\_right\_op**(optext):

*"""*

*Правосторонний оператор*

*https://ru.wikipedia.org/wiki/Операторы\_в\_C\_и\_C++*

**:param** *optext:*

*"""*

if optext in (*'~~~++'*, *'~~~--'*, *'+'*, *'-'*, *'!'*, *'~'*, *'(type)'*, *'\*'*,

*'&'*, *'sizeof'*, *'new'*, *'new[]'*, *'delete'*, *'delete[]'*,

*'?:'*, *'='*, *'+='*, *'-='*, *'\*='*, *'/='*,

*'%='*, *'<<='*, *'>>='*, *'&='*, *'|='*, *'^='*, *'throw'*, ):

return True

return False

*@property*

def **TABLE\_IDENTIFICATORS**(*self*):

return *self*.\_TABLE\_IDENTIFICATORS

*@TABLE\_IDENTIFICATORS.setter*

def **TABLE\_IDENTIFICATORS**(*self*, value):

*self*.\_TABLE\_IDENTIFICATORS = value

*@property*

def **TABLE\_CHAR\_CONSTANTS**(*self*):

return *self*.\_TABLE\_CHAR\_CONSTANTS

*@TABLE\_CHAR\_CONSTANTS.setter*

def **TABLE\_CHAR\_CONSTANTS**(*self*, value):

*self*.\_TABLE\_CHAR\_CONSTANTS = value

*@property*

def **TABLE\_NUM\_CONSTANTS**(*self*):

return *self*.\_TABLE\_NUM\_CONSTANTS

*@TABLE\_NUM\_CONSTANTS.setter*

def **TABLE\_NUM\_CONSTANTS**(*self*, value):

*self*.\_TABLE\_NUM\_CONSTANTS = value

def **\_\_init\_\_**(*self*, program\_filename: str):

*self*.program\_filename = program\_filename

*self*.\_current\_state = *'S'*

*self*.current\_state\_verbose = *''*

*self*.\_lex\_count = 0

*self*.\_start\_linen = 0

*self*.\_current\_linen = 0

*self*.\_current\_line = CharQueue()

*self*.\_start\_сharn = 0

*self*.\_current\_сharn = 0

*self*.\_current\_char = *''*

*self*.\_buffer = CharStack()

*self*.\_\_ALL\_registry = []

*self*.\_analysed\_lines = CharQueue()

with open(*self*.program\_filename, *'rt'*, encoding=*'utf-8'*) as f:

for line in f:

*self*.\_analysed\_lines.add(line)

*self*.TABLE\_IDENTIFICATORS = {}

*self*.TABLE\_CHAR\_CONSTANTS = {}

*self*.TABLE\_NUM\_CONSTANTS = {}

def **get\_reg\_lexem\_by\_order**(*self*, order: int):

if len(*self*.ALL\_registry):

lx = *self*.ALL\_registry[order - 1]

if lx.order != order:

for lx in *self*.ALL\_registry:

if lx.order == order:

return lx

return lx

return None

*@property*

def **current\_state**(*self*):

return *self*.\_current\_state

*@current\_state.setter*

def **current\_state**(*self*, value):

state\_verbose = *''*

if isinstance(value, (tuple, list)):

state\_verbose = value[1]

value = value[0]

if value not in *self*.STATES:

raise ValueError(*'Статус не из списка допустимых'*)

*self*.current\_state\_verbose = state\_verbose

*self*.\_current\_state = value

*@property*

def **current\_char**(*self*):

return *self*.\_current\_char

*@current\_char.setter*

def **current\_char**(*self*, value):

*self*.\_current\_char = value

*self*.\_current\_сharn += 1

def **add\_buff**(*self*, state=None, item=None, preclean=False):

if not item:

item = *self*.\_current\_char

if *self*.\_buffer.is\_empty and item in *self*.EXCEPT\_CHARS:

return

if preclean:

*self*.clear\_buff(state)

if state:

*self*.current\_state = state

if item:

*self*.\_buffer.add(item)

def **clear\_buff**(*self*, state=None):

if state:

*self*.current\_state = state

*self*.\_start\_linen = *self*.\_current\_linen

*self*.\_start\_сharn = *self*.\_current\_сharn

*self*.\_buffer.clear()

def **read\_buff**(*self*, top=False):

if top:

return *self*.\_buffer.top()

return *self*.\_buffer.read()

*@property*

def **is\_buff\_empty**(*self*):

return *self*.\_buffer.is\_empty

*@property*

def **ALL\_registry**(*self*):

return *self*.\_\_ALL\_registry

*@ALL\_registry.setter*

def **ALL\_registry**(*self*, value):

*self*.\_\_ALL\_registry = value

W\_registry = property(lambda s: [x for x in s.ALL\_registry if x.class\_code == *'W'*])

N\_registry = property(lambda s: [x for x in s.ALL\_registry if x.class\_code == *'N'*])

C\_registry = property(lambda s: [x for x in s.ALL\_registry if x.class\_code == *'C'*])

def **add\_lexem**(*self*, lexem: Lexem):

*self*.\_lex\_count += 1

if not lexem.text:

lexem.text = *self*.current\_char

if lexem.linen == 0:

lexem.linen = *self*.\_current\_linen

if lexem.charn == 0:

lexem.charn = *self*.\_current\_сharn

if lexem.slinen == 0:

lexem.slinen = *self*.\_start\_linen

if lexem.scharn == 0:

lexem.scharn = *self*.\_start\_сharn

if lexem.order == -1:

lexem.order = *self*.\_lex\_count

if lexem.lex\_id <= 0 and lexem.text:

lex\_id = *''*

if lexem.class\_code == *"W"*:

lex\_id = *self*.TABLE\_SPEC\_WORDS.get(lexem.text, 0)

elif lexem.class\_code == *"O"*:

lex\_id = *self*.TABLE\_OPERATIONS.get(lexem.text, 0)

elif lexem.class\_code == *"R"*:

lex\_id = *self*.TABLE\_SEPARATORS.get(lexem.text, 0)

elif lexem.class\_code == *"I"*:

clen = len(*self*.TABLE\_IDENTIFICATORS)

lexem.text = lexem.text.strip()

lex\_id = *self*.TABLE\_IDENTIFICATORS.get(lexem.text, 0)

if lex\_id == 0:

lex\_id = clen + 1

*self*.TABLE\_IDENTIFICATORS[lexem.text] = lex\_id

elif lexem.class\_code == *"C"*:

clen = len(*self*.TABLE\_CHAR\_CONSTANTS)

lexem.text = lexem.text.strip()

lex\_id = *self*.TABLE\_CHAR\_CONSTANTS.get(lexem.text, 0)

if lex\_id == 0:

lex\_id = clen + 1

*self*.TABLE\_CHAR\_CONSTANTS[lexem.text] = lex\_id

elif lexem.class\_code == *"N"*:

clen = len(*self*.TABLE\_NUM\_CONSTANTS)

lexem.text = lexem.text.strip()

lex\_id = *self*.TABLE\_NUM\_CONSTANTS.get(lexem.text, 0)

if lex\_id == 0:

lex\_id = clen + 1

*self*.TABLE\_NUM\_CONSTANTS[lexem.text] = lex\_id

lexem.lex\_id = lex\_id

*self*.ALL\_registry.append(lexem)

def **get\_table\_rows\_cols**(*self*) -> list:

th = *'''| № | Текст лексемы | Код | Описание лексемы | Строка:Символ |'''*

twidth = len(th)

thds = th.split(*'|'*)

thds = list(filter(lambda x: x != *''*, thds))

n\_charbuf = len(thds[0]) - 1

text\_charbuf = len(thds[1]) - 1

code\_charbuf = len(thds[2]) - 1

desc\_charbuf = len(thds[3]) - 1

linechar\_charbuf = len(thds[4]) - 1

def **get\_row**(n=*''*, text=*''*, code=*''*, desc=*''*, linechar=*''*):

n\_ = *f"{n:>{n\_charbuf}} "*

if text.count(*'-'*) != text\_charbuf:

text = text[:(text\_charbuf - 5)].replace(*'\n'*, *'\\n'*)

text\_ = *f"{text:>{text\_charbuf}} "*

code\_ = *f"{code:>{code\_charbuf}} "*

desc\_ = *f"{desc:>{desc\_charbuf}} "*

linechar\_ = *f"{linechar:>{linechar\_charbuf}} "*

return *'|'* + *'|'*.join([n\_, text\_, code\_, desc\_, linechar\_]) + *'|'*

ret\_list = [*'='* \* twidth,

th,

*'='* \* twidth,

]

for x in *self*.ALL\_registry:

lenes = *f'{x.linen}:{x.charn}'*

if (x.slinen or x.scharn) and not (x.linen == x.slinen and x.charn == x.scharn):

lenes = *f'{x.slinen}:{x.scharn}'* + *'-'* + lenes

ret\_list.append(get\_row(x.order, x.text, x.fcode, x.fdesc, lenes))

ret\_list.append(get\_row(*'-'* \* n\_charbuf, *'-'* \* text\_charbuf, *'-'* \* code\_charbuf, *'-'* \* desc\_charbuf, *'-'* \* linechar\_charbuf))

return ret\_list

def **\_\_str\_\_**(*self*):

if *self*.current\_state == *'Z'*:

return *'\n'*.join(*self*.get\_table\_rows\_cols()) + *f'\n\n{self.current\_state\_verbose}'*

if not *self*.current\_state == *'F'*:

return *'\n'*.join(*self*.get\_table\_rows\_cols())

return *f'Ошибка парсера (строка {self.\_current\_linen}, символ {self.\_current\_сharn}): {self.current\_state\_verbose}'*

def **run\_analysis**(*self*):

*"""*

*Начать анализ. Заполнить таблицы лексем.*

*"""*

*self*.\_time\_start = time.time()

# Цикл парсинга строк программы

while not *self*.\_analysed\_lines.is\_empty or not *self*.\_current\_line.is\_empty:

# Парсер вернул ошибку

if *self*.current\_state == *'F'*:

break

# Парсер чистит буфер за строку если не парсим многострочный комментарий

if *self*.current\_state != *'MLC'*:

*self*.clear\_buff()

*self*.\_current\_line = CharQueue(*self*.\_analysed\_lines.pop())

*self*.\_current\_linen += 1

*self*.\_current\_сharn = 0

*self*.current\_char = *self*.\_current\_line.pop()

*self*.add\_buff()

# Цикл парсинга символов в строке

full\_loop\_iter = True # Произошла ли полная итерация

ret\_or\_add\_chars\_to\_buff = False # Вернуть в буфер считанные дополнительные символы или взять новый символ

next\_chars = [] # Дополнительные символы считанные из строки но пока не добавленные в буфер

while (*self*.current\_char or not *self*.is\_buff\_empty) and not *self*.\_current\_line.is\_empty:

# Заполним буфер если был continue переход

if ret\_or\_add\_chars\_to\_buff:

if not len(next\_chars): # Берем следующий символ из буфера

*self*.current\_char = *self*.\_current\_line.pop()

*self*.add\_buff()

else: # Возвращаем в буфер прочитанные в итерации символы если они есть

for nc in next\_chars:

*self*.current\_char = nc

*self*.add\_buff()

elif not full\_loop\_iter: # Если оптимизационный continue случился - берем следующий символ в начале петли, иначе он берется в конце петли

*self*.current\_char = *self*.\_current\_line.pop()

*self*.add\_buff()

ret\_or\_add\_chars\_to\_buff = False

next\_chars = []

full\_loop\_iter = False

# / Заполним буфер если был continue переход

#-----------------------------------------

#-----------------------------------------

#-----------------------------------------

# print(self.read\_buff())

# print("=======")

# Не анализируем символы

if *self*.current\_char in *self*.EXCEPT\_CHARS and *self*.current\_state not in (*'LETT'*, *'IDENT'*, *'INT'*, *'REAL'*, *'MCOP'*, ):

continue

# Состояние когда окончены предыдущие парсинги лексем

elif *self*.current\_state == *'S'*:

# Установка стартовой позиции между успешными лексемами

*self*.\_start\_linen = *self*.\_current\_linen

*self*.\_start\_сharn = *self*.\_current\_сharn or 1

# Инклюд

if *self*.current\_char == *'#'*:

next\_chars.extend(x for x in *self*.\_current\_line if x != *'\n'*)

*self*.\_current\_сharn += len(next\_chars)

*self*.add\_lexem(Lexem(*'?'*, text=*self*.current\_char + *''*.join(next\_chars)))

*self*.clear\_buff(*'S'*)

continue # Однозначаное определение лексемы

# Многострочный комментарий, Однострочный комментарий, иное (операция деления)

elif *self*.current\_char == *'/'*:

next\_chars.append(*self*.\_current\_line.pop())

*self*.\_current\_сharn += len(next\_chars)

if next\_chars[-1] == *'\*'*: # Многострочный комментарий

*self*.add\_buff(*'MLC'*, next\_chars[-1])

continue

elif next\_chars[-1] == *'/'*: # Однострочный комментарий

*self*.add\_buff(*'OLC'*, next\_chars[-1])

full\_loop\_iter = True

continue

else: # Операция деления

*self*.add\_lexem(Lexem(*'O'*, text=*'/'*))

*self*.add\_buff(*'S'*, next\_chars[-1], preclean=True)

continue # Однозначаное определение лексемы

elif *self*.current\_char == *"'"* or *self*.current\_char == *'"'*:

next\_chars.append(*self*.\_current\_line.pop())

*self*.\_current\_сharn += len(next\_chars)

if *self*.current\_char == next\_chars[-1]:

*self*.add\_lexem(Lexem(*'C'*, text=*self*.current\_char + next\_chars[-1]))

*self*.clear\_buff(*'S'*)

continue # Однозначаное определение лексемы

else:

*self*.add\_buff(*'CHAR'*, next\_chars[-1])

continue

elif *self*.current\_char in map(lambda x: x[0], *self*.MULTY\_CHAR\_OPERATIONS.keys()):

*self*.current\_state = *'MCOP'*

continue

elif *self*.current\_char in *self*.SINGLE\_CHAR\_OPERATIONS:

*self*.add\_lexem(Lexem(*'O'*))

*self*.clear\_buff(*'S'*)

continue # Однозначаное определение лексемы

elif *self*.current\_char in *self*.TABLE\_SEPARATORS:

*self*.add\_lexem(Lexem(*'R'*))

*self*.clear\_buff(*'S'*)

continue # Однозначаное определение лексемы

elif *self*.current\_char.isalpha():

*self*.current\_state = *'LETT'*

continue

elif *self*.current\_char.isdigit():

*self*.clear\_buff(*'INT'*)

*self*.add\_buff()

continue

elif *self*.current\_char == *'.'*:

*self*.clear\_buff(*'REAL'*)

*self*.add\_buff()

continue

elif *self*.current\_char == *'\_'*:

*self*.clear\_buff(*'IDENT'*)

*self*.add\_buff()

continue

elif ord(*self*.current\_char) == 65279: # BOM (Byte Order Mark) ? 'ZERO WIDTH NO-BREAK SPACE'

*self*.clear\_buff()

continue

else:

*self*.current\_state = *'F'*

break # Переход на следующую строку и выход

# Состояние когда идет анализ какойто лексемы

else:

if *self*.current\_state == *'MLC'*:

if *self*.current\_char == *'\*'*: # Многострочный комментарий, Закрывающая звездочка

next\_chars.append(*self*.\_current\_line.pop())

*self*.\_current\_сharn += len(next\_chars)

if next\_chars[-1] == *'/'*: # Многострочный комментарий, Закрывающий слеш

*self*.add\_buff(item=next\_chars[-1])

*self*.add\_lexem(Lexem(*'?'*, text=*self*.read\_buff()))

*self*.clear\_buff(*'S'*)

continue # Однозначаное определение лексемы

if *self*.current\_state == *'OLC'*:

next\_chars.append(*self*.\_current\_line.pop())

if not next\_chars[-1] or next\_chars[-1] == *'\n'*: # строчный комментарий

*self*.add\_lexem(Lexem(*'?'*, text=*self*.read\_buff()))

*self*.clear\_buff(*'S'*)

continue

ret\_or\_add\_chars\_to\_buff = True

continue

elif *self*.current\_state == *'LETT'* or *self*.current\_state == *'IDENT'*:

if *self*.current\_char.isdigit() or *self*.current\_char == *'\_'*:

*self*.current\_state = *'IDENT'*

continue

elif *self*.current\_char != *' '* and *self*.current\_char.isalpha():

continue

elif *self*.current\_char in *self*.IMPLICIT\_SEPARATORS:

stripchars = *''*.join(*self*.IMPLICIT\_SEPARATORS)

scharn = *self*.\_start\_сharn

buff = *self*.read\_buff()

bunocu = buff.lstrip(stripchars)

bunocu\_ = bunocu.rstrip(stripchars)

charn = scharn + len(bunocu\_) - 1

bunocu = bunocu\_

if *self*.current\_state == *'LETT'*:

if bunocu in *self*.TABLE\_SPEC\_WORDS:

*self*.add\_lexem(Lexem(*'W'*, text=bunocu, charn=charn, scharn=scharn))

*self*.add\_buff(*'S'*, *self*.current\_char, preclean=True)

full\_loop\_iter = True

continue # Однозначаное определение лексемы

else:

*self*.add\_lexem(Lexem(*'I'*, text=bunocu, charn=charn, scharn=scharn))

*self*.add\_buff(*'S'*, *self*.current\_char, preclean=True)

full\_loop\_iter = True

continue # Однозначаное определение лексемы

elif *self*.current\_state == *'IDENT'*:

*self*.add\_lexem(Lexem(*'I'*, text=bunocu, charn=charn, scharn=scharn))

*self*.add\_buff(*'S'*, *self*.current\_char, preclean=True)

full\_loop\_iter = True

continue # Однозначаное определение лексемы

*self*.add\_buff(*'S'*, *self*.current\_char, preclean=True)

full\_loop\_iter = True

continue

elif *self*.current\_state == *'INT'* or *self*.current\_state == *'REAL'*:

if *self*.current\_char.isdigit():

continue

elif *self*.current\_char == *'.'* and *self*.current\_state == *'INT'*:

*self*.current\_state = *'REAL'*

continue

elif *self*.current\_char in *self*.IMPLICIT\_SEPARATORS:

stripchars = *''*.join(*self*.IMPLICIT\_SEPARATORS)

scharn = *self*.\_start\_сharn

buff = *self*.read\_buff()

bunocu = buff.lstrip(stripchars)

bunocu\_ = bunocu.rstrip(stripchars)

charn = scharn + len(bunocu\_) - 1

bunocu = bunocu\_

*self*.add\_lexem(Lexem(*'N'*, text=bunocu, charn=charn, scharn=scharn))

*self*.add\_buff(*'S'*, *self*.current\_char, preclean=True)

full\_loop\_iter = True

continue # Однозначаное определение лексемы

else:

*self*.clear\_buff((*'F'*, *f"Символ {self.current\_char} не может быть "*

*f"лексемой в данном контексте "*

*f"({self.current\_state})"*))

break # Переход на следующую строку и выход

elif *self*.current\_state == *'CHAR'*:

buff = *self*.read\_buff()

firstchar = buff.lstrip()[0]

if *self*.current\_char == *'\\'*:

next\_chars.append(*self*.\_current\_line.pop())

*self*.\_current\_сharn += len(next\_chars)

if next\_chars[-1] == *'\n'*:

*self*.clear\_buff(*'S'*)

break # Переход на следующую строку

ret\_or\_add\_chars\_to\_buff = True

continue

elif (*self*.\_analysed\_lines.is\_empty and *self*.\_current\_line.is\_empty and \

not *self*.current\_char) or *self*.current\_char == *'\n'*:

*self*.clear\_buff((*'F'*, *f"Обрыв строки посередине строковой константы: {self.read\_buff()}"*))

break # Переход на следующую строку и выход

elif *self*.current\_char == firstchar:

*self*.add\_lexem(Lexem(*'C'*, text=*self*.read\_buff()))

*self*.clear\_buff(*'S'*)

continue # Однозначаное определение лексемы

elif *self*.current\_state == *'MCOP'*:

if *self*.current\_char == *' '* or *self*.current\_char in *self*.IMPLICIT\_SEPARATORS:

stripchars = *' '* + *''*.join(*self*.TABLE\_SEPARATORS.keys()) + *''*.join(*self*.EXCEPT\_CHARS)

scharn = *self*.\_start\_сharn

buff = *self*.read\_buff()

full\_op = buff.lstrip(stripchars)

bunocu\_ = full\_op.rstrip(stripchars)

charn = scharn + len(bunocu\_) - 1

full\_op = bunocu\_

oneop = full\_op[0]

if full\_op in *self*.MULTY\_CHAR\_OPERATIONS:

*self*.add\_lexem(Lexem(*'O'*, text=full\_op, scharn=scharn, charn=charn))

*self*.clear\_buff(*'S'*)

continue # Однозначаное определение лексемы

elif oneop in *self*.SINGLE\_CHAR\_OPERATIONS and *self*.current\_char not in *self*.SINGLE\_CHAR\_OPERATIONS:

*self*.add\_lexem(Lexem(*'O'*, text=oneop, charn=scharn, scharn=scharn))

*self*.add\_buff(*'S'*, *self*.current\_char, preclean=True)

full\_loop\_iter = True

continue # Однозначаное определение лексемы

else: # Не многолитерная не однолитерная

*self*.clear\_buff((*'F'*, *f"оператор {full\_op} не является оператором!"*))

break # Переход на следующую строку

else: # Не многолитерная не однолитерная

*self*.clear\_buff((*'F'*, *f"{self.read\_buff()} не является оператором!"*))

break # Переход на следующую строку

pass

#-----------------------------------------

#-----------------------------------------

#-----------------------------------------

# Заполним буфер поумолчанию

if ret\_or\_add\_chars\_to\_buff:

if not len(next\_chars): # Берем следующий символ из буфера

*self*.current\_char = *self*.\_current\_line.pop()

*self*.add\_buff()

else: # Возвращаем в буфер прочитанные в итерации символы если они есть

for nc in next\_chars:

*self*.current\_char = nc

*self*.add\_buff()

else: # Берем следующий символ из буфера

*self*.current\_char = *self*.\_current\_line.pop()

*self*.add\_buff()

ret\_or\_add\_chars\_to\_buff = False

next\_chars = []

# / Заполним буфер поумолчанию

full\_loop\_iter = True

else:

*self*.\_time\_end = time.time()

*self*.current\_state = (*'Z'*, *f'Лексический анализ завершен без ошибок. Время выполнения: {self.\_time\_end - self.\_time\_start} с.'*)

pass

if \_\_name\_\_ == *"\_\_main\_\_"*:

anali = LexicalAnalyzerC(*r'input\_code.cpp'*)

anali.run\_analysis()

with open(anali.program\_filename, *'rt'*, encoding=*'utf-8'*) as f:

print(f.read())

print(anali)

pass

Листинг 2. Файл main.py – основной код программы

# Вывод

Программа была написана на ЯП Python 3. В качестве входного языка был выбран C++. Программа принимает на вход пусть к файлу и с помощью алгоритма конечного автомата преобразует код программы в набор лексем определенных в исходных данных задачи (преобразуемого языка). Набор лексем выводится ввиде текстовой таблицы. Полный код программы доступен на https://github.com/PaiNt-git/study.teor\_pl/tree/main/src/lab1\_lexical\_analysis