# Лабораторная работая №1

# По дисциплине

# «Проектирование компиляторов»

# Тема:

# «Программирование лексического анализатора»

**Студент:** Владислав Бураков

**Группа:** 4101BN

**Вариант:** 3

TSI, 2014

**1. Теоретическое обоснование**

Цель данной лабораторной работы – произвести анализ фрагмента исходного кода на языке Pascal. На вход подается упомянутый фрагмент, проанализировав который, сканнер должен распознать лексемы, заданные правилами регулярной грамматики, и строит таблицы для идентификаторов и литерал (числовых и строковых).

Язык реализации лексического анализатора – Python (версия 3.4)

**2. Текст исходного кода индивидуального варианта**

**procedure** TCalcDisplay.Draw;

var

Color: **Byte**;

I: Integer;

B: TDrawBuffer;

begin

Key := UpCase(Key);

**if**(Status=csError) **and** (Key<>'C') **then**

Key:=' ';

Color := GetColor(1);

I := Size.X - Length(Number) - 2;

MoveChar(B, ' ', Color, Size.X);

MoveChar(B[I], Sign, Color, 1);

MoveStr(B[I + 1], Number, Color);

WriteBuf(0, 0, Size.X, 1, B);

end;

**3. Грамматические правила для лексем, содержащихся в данном фрагменте**

**PROCEDURE -> procedure**

**VAR -> var**

**BEGIN -> begin**

**IF -> if**

**AND -> and**

**THEN -> then**

**END -> end**

**LPAREN -> (**

**RPAREN -> )**

**LCURLY -> {**

**RCURLY -> }**

**COMMA -> ,**

**DOT -> .**

**SEMICOLON -> ;**

**COLON -> :**

**LSQUARE -> [**

**RSQUARE -> ]**

**ASSIGN -> :->**

**OPERATOR -> - | + | \* | / | == | <> | > | < | >= | <=**

**WS -> (\s | \t | \n)+**

**ID -> [a-zA-Z\_][a-zA-Z0-9\_]\***

**STRING -> "[^"]\*" | '[^']\*'**

**INTEGER -> [0-9]+**

**REAL -> [0-9]\.[0-9]+**

**3. Диаграмма состояний конечного автомата**



**4. Исходной текст лексического анализатора**

**import** string

**from** enum **import** Enum

TokenType = Enum("TokenType", "LPAREN RPAREN IF AND OR THEN COMMA ID DOT COLON SEMICOLON PROCEDURE VAR BEGIN END ASSIGN OPERATOR WS STRING INTEGER REAL LSQUARE RSQUARE UNKNOWN")

**class** Token:

**def** \_\_init\_\_(self, kind, val, index, table):

self.kind = kind

self.val = val

self.idx = index

self.table = table

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

**return** "Kind: {0}, value: \"{1}\", idx: {2}".format(self.kind, self.val, self.idx)

**class** IDEntry:

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

**return** "Identifier({0}), value: \"{1}\"".format(self.kind, self.val)

**def** \_\_init\_\_(self, kind, value):

self.kind = TokenType.ID

self.val = value

**class** LiteralEntry:

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

**return** "Literal ({0}), value: \"{1}\"".format(self.kind, self.val)

**def** \_\_init\_\_(self, kind, value):

self.kind = kind

self.val = value

**class** KeywordTableEntry:

**def** \_\_init\_\_(self, val, kind, isSep = False):

self.val = val

self.kind = kind

self.isSep = isSep

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

**return** "Keyword ({0}), value: \"{1}\", is a separator: {2}".format(self.kind, self.val, self.isSep)

**class** FileStream:

**def** \_\_init\_\_(self, filename):

self.fd = open(filename, "r")

self.pos = 0

self.lines = []

self.lines.append(self.fd.readline())

self.cur\_line = 0

self.eof = False

**def** current\_line\_num(self):

**return** self.cur\_line + 1

**def** getChar(self):

char = None

**if** self.pos == len(self.\_get\_current\_line()) **and** self.cur\_line == len(self.lines) - 1 **and** self.eof:

**return** None

char = self.\_get\_current\_line()[self.pos]

self.pos += 1

**if** self.pos >= len(self.\_get\_current\_line()):

**if** self.cur\_line == len(self.lines) -1:

line = self.fd.readline()

**if** line **is** "":

self.eof = True

line = line.rstrip("\n")

**if** line **is** **not** "":

self.lines.append(line)

**if** **not** self.eof:

self.pos = 0

self.cur\_line += 1

**return** char

**def** \_get\_current\_line(self):

**return** self.lines[self.cur\_line]

**def** putChar(self, how\_many = 1):

**while** how\_many > 0:

**if** self.lines **is** []:

**raise** Exception("No lines")

**elif** self.pos == 0 :

**if** self.cur\_line == 0:

**raise** Exception("Cannot put char: zeroeth position within only line")

**else**:

self.cur\_line -= 1

self.pos = len(self.\_get\_current\_line()) - 1

**else**:

self.pos -= 1

how\_many -= 1

**class** Lexer:

**def** \_\_init\_\_(self, stream):

self.line = None

self.inp = None

self.sym\_table = [ ]

self.stream = stream

self.lexeme = ""

self.char = None

self.keywords = [

KeywordTableEntry(":=", TokenType.ASSIGN),

KeywordTableEntry("==", TokenType.OPERATOR),

KeywordTableEntry(">=", TokenType.OPERATOR),

KeywordTableEntry("<=", TokenType.OPERATOR),

KeywordTableEntry("<>", TokenType.OPERATOR),

KeywordTableEntry("<", TokenType.OPERATOR),

KeywordTableEntry(">", TokenType.OPERATOR),

KeywordTableEntry("if", TokenType.IF),

KeywordTableEntry("and", TokenType.AND),

KeywordTableEntry("or", TokenType.OR),

KeywordTableEntry("then", TokenType.THEN),

KeywordTableEntry("procedure", TokenType.PROCEDURE),

KeywordTableEntry("var", TokenType.VAR),

KeywordTableEntry("begin", TokenType.BEGIN),

KeywordTableEntry("end", TokenType.END),

KeywordTableEntry("(", TokenType.LPAREN, True),

KeywordTableEntry(")", TokenType.RPAREN, True),

KeywordTableEntry("[", TokenType.LSQUARE, True),

KeywordTableEntry("]", TokenType.RSQUARE, True),

KeywordTableEntry(",", TokenType.COMMA, True),

KeywordTableEntry(".", TokenType.DOT, True),

KeywordTableEntry(";", TokenType.SEMICOLON, True),

KeywordTableEntry(":", TokenType.COLON, True),

KeywordTableEntry("+", TokenType.OPERATOR),

KeywordTableEntry("-", TokenType.OPERATOR),

KeywordTableEntry("\\", TokenType.OPERATOR),

KeywordTableEntry("\*", TokenType.OPERATOR),

]

self.identifiers = [

]

self.literals = [

]

**def** getIndexOfKw(self, val):

**for** i **in** range(len(self.keywords)):

**if** self.keywords[i].val == val:

**return** i

**return** None

**def** isId(self):

**return** (lambda s: s **in** string.ascii\_letters)(self.char)

**def** readId(self):

self.getChar()

**while** self.char **and** (self.char **in** string.ascii\_letters **or** self.char **in** string.digits):

self.lexeme += self.char

self.getChar()

self.putChar()

**def** processLexeme(self, kind = TokenType.ID):

**if** kind **is** TokenType.ID:

kwIndex = self.getIndexOfKw(self.lexeme)

#first check if exists in the keyword table

**if** kwIndex **is** **not** None:

kwEntry = self.keywords[kwIndex]

**return** Token(kwEntry.kind, self.lexeme, kwIndex, self.keywords)

**else**:

#true ID, not a keyword

entry = IDEntry(kind, self.lexeme)

self.identifiers.append(entry)

**return** Token(TokenType.ID, self.lexeme, self.identifiers.index(entry), self.identifiers)

**else**:

#literal

entry = LiteralEntry(kind, self.lexeme)

self.literals.append(entry)

**return** Token(kind, self.lexeme, self.literals.index(entry), self.literals)

**def** getChar(self):

self.char = self.stream.getChar()

**def** putChar(self, howMany = 1):

self.stream.putChar(howMany)

**def** handleNumber(self):

**while** self.char **and** self.char **in** string.digits **or** (self.char **is** "." **and** "." **not** **in** self.lexeme):

self.lexeme += self.char

self.getChar()

self.putChar()

**if** "." **in** self.lexeme:

**return** self.processLexeme(TokenType.REAL)

**return** self.processLexeme(TokenType.INTEGER)

**def** get\_token(self):

**global** line\_num

**while** True:

self.lexeme = ""

self.getChar()

**if** **not** self.char:

**return** None

**if** self.char **in** string.whitespace:

**while** self.char **and** self.char **in** string.whitespace:

self.getChar()

**if** self.char:

self.putChar()

**continue**

**elif** self.char **is** "\"" **or** self.char **is** "\'":

op\_quote = self.char

self.getChar()

escape = self.char **is** "\\"

**while** self.char **and** (escape **or** self.char **is** **not** op\_quote):

escape = self.char **is** "\\"

**if** **not** escape:

self.lexeme += self.char

self.getChar()

**return** self.processLexeme(TokenType.STRING)

**elif** self.isId():

self.putChar()

self.readId()

**return** self.processLexeme(TokenType.ID)

**elif** self.char **is** "-" **or** self.char **is** "+":

self.lexeme += self.char

self.getChar()

**if** self.char **in** string.digits:

**return** self.handleNumber()

**else**:

#put back both "-"(or "+") and current char

self.putChar(2)

#get back "-" (or "+") and fall-through to the last if

self.getChar()

self.lexeme = ""

**elif** self.char **in** string.digits:

**return** self.handleNumber()

**elif** self.char **is** ":" **or** self.char **is** "<" **or** self.char **is** ">" **or** self.char **is** "=":

self.lexeme += self.char

self.getChar()

**if** self.char **is** "=":

self.lexeme += self.char

**return** self.processLexeme()

**elif** self.char **is** ">":

self.lexeme += self.char

**return** self.processLexeme()

**else**:

self.putChar()

**return** self.processLexeme()

#keywords get processed here ("default" case in C/C++)

**if** self.getIndexOfKw(self.char) **is** **not** None:

self.lexeme += self.char

**return** self.processLexeme()

**else**:

**while** self.char **and** self.char **not** **in** string.digits **and** self.char **not** **in** string.ascii\_letters:

self.lexeme += self.char

self.getChar()

self.putChar()

**return** Token(TokenType.UNKNOWN, self.lexeme, 0, None)

**def** printTable(self, table):

i = 0

**for** entry **in** table:

**print**("{0}) {1}".format(i, entry))

i += 1

**4.1 Текст программы, использующей лексический анализатор**

from lexer import \*

**def** main():

stream = FileStream("input.pas")

lexer = Lexer(stream)

**while** True:

line = lexer.get\_token()

**if** line == None:

**break**

**elif** line.kind == TokenType.UNKNOWN:

**print**("\nError: Unknown lexeme {0} at line {1}".format(line.val, lexer.stream.current\_line\_num()))

**return**

**print**(line)

**print**("\nSuccess: got a stream of tokens")

**print**("\nPrinting a table of keywords:\n")

lexer.printTable(lexer.keywords)

**print**("\nPrinting a table of literals:\n")

lexer.printTable(lexer.literals)

**print**("\nPrinting a table of IDs:\n")

lexer.printTable(lexer.identifiers)

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

main()

**5.1 Исходные таблицы лексем (вывод управляющей программы)**

**0) Keyword (TokenType.ASSIGN), value: ":=", is a separator: False**

**1) Keyword (TokenType.OPERATOR), value: "==", is a separator: False**

**2) Keyword (TokenType.OPERATOR), value: ">=", is a separator: False**

**3) Keyword (TokenType.OPERATOR), value: "<=", is a separator: False**

**4) Keyword (TokenType.OPERATOR), value: "<>", is a separator: False**

**5) Keyword (TokenType.OPERATOR), value: "<", is a separator: False**

**6) Keyword (TokenType.OPERATOR), value: ">", is a separator: False**

**7) Keyword (TokenType.IF), value: "if", is a separator: False**

**8) Keyword (TokenType.AND), value: "and", is a separator: False**

**9) Keyword (TokenType.OR), value: "or", is a separator: False**

**10) Keyword (TokenType.THEN), value: "then", is a separator: False**

**11) Keyword (TokenType.PROCEDURE), value: "procedure", is a separator: False**

**12) Keyword (TokenType.VAR), value: "var", is a separator: False**

**13) Keyword (TokenType.BEGIN), value: "begin", is a separator: False**

**14) Keyword (TokenType.END), value: "end", is a separator: False**

**15) Keyword (TokenType.LPAREN), value: "(", is a separator: True**

**16) Keyword (TokenType.RPAREN), value: ")", is a separator: True**

**17) Keyword (TokenType.LSQUARE), value: "[", is a separator: True**

**18) Keyword (TokenType.RSQUARE), value: "]", is a separator: True**

**19) Keyword (TokenType.COMMA), value: ",", is a separator: True**

**20) Keyword (TokenType.DOT), value: ".", is a separator: True**

**21) Keyword (TokenType.SEMICOLON), value: ";", is a separator: True**

**22) Keyword (TokenType.COLON), value: ":", is a separator: True**

**23) Keyword (TokenType.OPERATOR), value: "+", is a separator: False**

**24) Keyword (TokenType.OPERATOR), value: "-", is a separator: False**

**25) Keyword (TokenType.OPERATOR), value: "\", is a separator: False**

**26) Keyword (TokenType.OPERATOR), value: "\*", is a separator: False**

**5.2 Таблицы распознанных лексем (литералы и идентификаторы)**

**Printing a table of literals:**

**0) Literal (TokenType.STRING), value: "C"**

**1) Literal (TokenType.STRING), value: " "**

**2) Literal (TokenType.INTEGER), value: "1"**

**3) Literal (TokenType.INTEGER), value: "2"**

**4) Literal (TokenType.STRING), value: " "**

**5) Literal (TokenType.INTEGER), value: "1"**

**6) Literal (TokenType.INTEGER), value: "1"**

**7) Literal (TokenType.INTEGER), value: "0"**

**8) Literal (TokenType.INTEGER), value: "0"**

**9) Literal (TokenType.INTEGER), value: "1"**

**Printing a table of IDs:**

**0) Identifier(TokenType.ID), value: "TCalcDisplay"**

**1) Identifier(TokenType.ID), value: "Draw"**

**2) Identifier(TokenType.ID), value: "Color"**

**3) Identifier(TokenType.ID), value: "Byte"**

**4) Identifier(TokenType.ID), value: "I"**

**5) Identifier(TokenType.ID), value: "Integer"**

**6) Identifier(TokenType.ID), value: "B"**

**7) Identifier(TokenType.ID), value: "TDrawBuffer"**

**8) Identifier(TokenType.ID), value: "Key"**

**9) Identifier(TokenType.ID), value: "UpCase"**

**10) Identifier(TokenType.ID), value: "Key"**

**11) Identifier(TokenType.ID), value: "Status"**

**12) Identifier(TokenType.ID), value: "="**

**13) Identifier(TokenType.ID), value: "csError"**

**14) Identifier(TokenType.ID), value: "Key"**

**15) Identifier(TokenType.ID), value: "Key"**

**16) Identifier(TokenType.ID), value: "Color"**

**17) Identifier(TokenType.ID), value: "GetColor"**

**18) Identifier(TokenType.ID), value: "I"**

**19) Identifier(TokenType.ID), value: "Size"**

**20) Identifier(TokenType.ID), value: "X"**

**21) Identifier(TokenType.ID), value: "Length"**

**22) Identifier(TokenType.ID), value: "Number"**

**23) Identifier(TokenType.ID), value: "MoveChar"**

**24) Identifier(TokenType.ID), value: "B"**

**25) Identifier(TokenType.ID), value: "Color"**

**26) Identifier(TokenType.ID), value: "Size"**

**27) Identifier(TokenType.ID), value: "X"**

**28) Identifier(TokenType.ID), value: "MoveChar"**

**29) Identifier(TokenType.ID), value: "B"**

**30) Identifier(TokenType.ID), value: "I"**

**31) Identifier(TokenType.ID), value: "Sign"**

**32) Identifier(TokenType.ID), value: "Color"**

**33) Identifier(TokenType.ID), value: "MoveStr"**

**34) Identifier(TokenType.ID), value: "B"**

**35) Identifier(TokenType.ID), value: "I"**

**36) Identifier(TokenType.ID), value: "Number"**

**37) Identifier(TokenType.ID), value: "Color"**

**38) Identifier(TokenType.ID), value: "WriteBuf"**

**39) Identifier(TokenType.ID), value: "Size"**

**40) Identifier(TokenType.ID), value: "X"**

**41) Identifier(TokenType.ID), value: "B"**

**6. Результаты работы лексического анализатора**

**Выводы таблиц из п. 5.1 и п. 5.2 опущены.**

**6.1 Исходной код без ошибок**

**Kind: TokenType.PROCEDURE, value: "procedure", idx: 11**

**Kind: TokenType.ID, value: "TCalcDisplay", idx: 0**

**Kind: TokenType.DOT, value: ".", idx: 20**

**Kind: TokenType.ID, value: "Draw", idx: 1**

**Kind: TokenType.SEMICOLON, value: ";", idx: 21**

**Kind: TokenType.VAR, value: "var", idx: 12**

**Kind: TokenType.ID, value: "Color", idx: 2**

**Kind: TokenType.COLON, value: ":", idx: 22**

**Kind: TokenType.ID, value: "Byte", idx: 3**

**Kind: TokenType.SEMICOLON, value: ";", idx: 21**

**Kind: TokenType.ID, value: "I", idx: 4**

**Kind: TokenType.COLON, value: ":", idx: 22**

**Kind: TokenType.ID, value: "Integer", idx: 5**

**Kind: TokenType.SEMICOLON, value: ";", idx: 21**

**Kind: TokenType.ID, value: "B", idx: 6**

**Kind: TokenType.COLON, value: ":", idx: 22**

**Kind: TokenType.ID, value: "TDrawBuffer", idx: 7**

**Kind: TokenType.SEMICOLON, value: ";", idx: 21**

**Kind: TokenType.BEGIN, value: "begin", idx: 13**

**Kind: TokenType.ID, value: "Key", idx: 8**

**Kind: TokenType.ASSIGN, value: ":=", idx: 0**

**Kind: TokenType.ID, value: "UpCase", idx: 9**

**Kind: TokenType.LPAREN, value: "(", idx: 15**

**Kind: TokenType.ID, value: "Key", idx: 10**

**Kind: TokenType.RPAREN, value: ")", idx: 16**

**Kind: TokenType.SEMICOLON, value: ";", idx: 21**

**Kind: TokenType.IF, value: "if", idx: 7**

**Kind: TokenType.LPAREN, value: "(", idx: 15**

**Kind: TokenType.ID, value: "Status", idx: 11**

**Kind: TokenType.ID, value: "=", idx: 12**

**Kind: TokenType.ID, value: "csError", idx: 13**

**Kind: TokenType.RPAREN, value: ")", idx: 16**

**Kind: TokenType.AND, value: "and", idx: 8**

**Kind: TokenType.LPAREN, value: "(", idx: 15**

**Kind: TokenType.ID, value: "Key", idx: 14**

**Kind: TokenType.OPERATOR, value: "<>", idx: 4**

**Kind: TokenType.STRING, value: "C", idx: 0**

**Kind: TokenType.RPAREN, value: ")", idx: 16**

**Kind: TokenType.THEN, value: "then", idx: 10**

**Kind: TokenType.ID, value: "Key", idx: 15**

**Kind: TokenType.ASSIGN, value: ":=", idx: 0**

**Kind: TokenType.STRING, value: " ", idx: 1**

**Kind: TokenType.SEMICOLON, value: ";", idx: 21**

**Kind: TokenType.ID, value: "Color", idx: 16**

**Kind: TokenType.ASSIGN, value: ":=", idx: 0**

**Kind: TokenType.ID, value: "GetColor", idx: 17**

**Kind: TokenType.LPAREN, value: "(", idx: 15**

**Kind: TokenType.INTEGER, value: "1", idx: 2**

**Kind: TokenType.RPAREN, value: ")", idx: 16**

**Kind: TokenType.SEMICOLON, value: ";", idx: 21**

**Kind: TokenType.ID, value: "I", idx: 18**

**Kind: TokenType.ASSIGN, value: ":=", idx: 0**

**Kind: TokenType.ID, value: "Size", idx: 19**

**Kind: TokenType.DOT, value: ".", idx: 20**

**Kind: TokenType.ID, value: "X", idx: 20**

**Kind: TokenType.OPERATOR, value: "-", idx: 24**

**Kind: TokenType.ID, value: "Length", idx: 21**

**Kind: TokenType.LPAREN, value: "(", idx: 15**

**Kind: TokenType.ID, value: "Number", idx: 22**

**Kind: TokenType.RPAREN, value: ")", idx: 16**

**Kind: TokenType.OPERATOR, value: "-", idx: 24**

**Kind: TokenType.INTEGER, value: "2", idx: 3**

**Kind: TokenType.SEMICOLON, value: ";", idx: 21**

**Kind: TokenType.ID, value: "MoveChar", idx: 23**

**Kind: TokenType.LPAREN, value: "(", idx: 15**

**Kind: TokenType.ID, value: "B", idx: 24**

**Kind: TokenType.COMMA, value: ",", idx: 19**

**Kind: TokenType.STRING, value: " ", idx: 4**

**Kind: TokenType.COMMA, value: ",", idx: 19**

**Kind: TokenType.ID, value: "Color", idx: 25**

**Kind: TokenType.COMMA, value: ",", idx: 19**

**Kind: TokenType.ID, value: "Size", idx: 26**

**Kind: TokenType.DOT, value: ".", idx: 20**

**Kind: TokenType.ID, value: "X", idx: 27**

**Kind: TokenType.RPAREN, value: ")", idx: 16**

**Kind: TokenType.SEMICOLON, value: ";", idx: 21**

**Kind: TokenType.ID, value: "MoveChar", idx: 28**

**Kind: TokenType.LPAREN, value: "(", idx: 15**

**Kind: TokenType.ID, value: "B", idx: 29**

**Kind: TokenType.LSQUARE, value: "[", idx: 17**

**Kind: TokenType.ID, value: "I", idx: 30**

**Kind: TokenType.RSQUARE, value: "]", idx: 18**

**Kind: TokenType.COMMA, value: ",", idx: 19**

**Kind: TokenType.ID, value: "Sign", idx: 31**

**Kind: TokenType.COMMA, value: ",", idx: 19**

**Kind: TokenType.ID, value: "Color", idx: 32**

**Kind: TokenType.COMMA, value: ",", idx: 19**

**Kind: TokenType.INTEGER, value: "1", idx: 5**

**Kind: TokenType.RPAREN, value: ")", idx: 16**

**Kind: TokenType.SEMICOLON, value: ";", idx: 21**

**Kind: TokenType.ID, value: "MoveStr", idx: 33**

**Kind: TokenType.LPAREN, value: "(", idx: 15**

**Kind: TokenType.ID, value: "B", idx: 34**

**Kind: TokenType.LSQUARE, value: "[", idx: 17**

**Kind: TokenType.ID, value: "I", idx: 35**

**Kind: TokenType.OPERATOR, value: "+", idx: 23**

**Kind: TokenType.INTEGER, value: "1", idx: 6**

**Kind: TokenType.RSQUARE, value: "]", idx: 18**

**Kind: TokenType.COMMA, value: ",", idx: 19**

**Kind: TokenType.ID, value: "Number", idx: 36**

**Kind: TokenType.COMMA, value: ",", idx: 19**

**Kind: TokenType.ID, value: "Color", idx: 37**

**Kind: TokenType.RPAREN, value: ")", idx: 16**

**Kind: TokenType.SEMICOLON, value: ";", idx: 21**

**Kind: TokenType.ID, value: "WriteBuf", idx: 38**

**Kind: TokenType.LPAREN, value: "(", idx: 15**

**Kind: TokenType.INTEGER, value: "0", idx: 7**

**Kind: TokenType.COMMA, value: ",", idx: 19**

**Kind: TokenType.INTEGER, value: "0", idx: 8**

**Kind: TokenType.COMMA, value: ",", idx: 19**

**Kind: TokenType.ID, value: "Size", idx: 39**

**Kind: TokenType.DOT, value: ".", idx: 20**

**Kind: TokenType.ID, value: "X", idx: 40**

**Kind: TokenType.COMMA, value: ",", idx: 19**

**Kind: TokenType.INTEGER, value: "1", idx: 9**

**Kind: TokenType.COMMA, value: ",", idx: 19**

**Kind: TokenType.ID, value: "B", idx: 41**

**Kind: TokenType.RPAREN, value: ")", idx: 16**

**Kind: TokenType.SEMICOLON, value: ";", idx: 21**

**Kind: TokenType.END, value: "end", idx: 14**

**Kind: TokenType.SEMICOLON, value: ";", idx: 21**

**Success: got a stream of tokens**

**6.1 Ошибка в исходном коде**

**Во второй строке присутствует символ #**

**Kind: TokenType.PROCEDURE, value: "procedure", idx: 11**

**Kind: TokenType.ID, value: "TCalcDisplay", idx: 0**

**Kind: TokenType.DOT, value: ".", idx: 20**

**Kind: TokenType.ID, value: "Draw", idx: 1**

**Kind: TokenType.SEMICOLON, value: ";", idx: 21**

**Kind: TokenType.VAR, value: "var", idx: 12**

**Error: Unknown lexeme # at line 3**

**7. Выводы о проделанной работе**

**Несмотря на то, что лексемы, содержащиеся в исходном коде, распознаются, данная реализация лексического анализатора не является легко расширяемой. Универсальным решением данной проблемы мог быть программный продукт, который принимает грамматику в виде строки/текстового файла и на основе этого генерирует соответствующий конечный автомат. Частично это решается использованием генераторов лексических анализаторов Flex/Lex, однако на выходе программист получает реализацию на языке C, который не является ОО, что опять-таки не позволяет с легкостью расширить/доделать сканнер «под себя» (к примеру, репортирование ошибок).**

**Лучшим языком на роль реализации подходил бы язык со статической типизацией типа Java или C++. Несмотря на достаточно высокую скорость разработки на Python, отсутствие проверки на этапе компиляции и конструкции switch доставило немало неудобств.**