Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

высшего образования «Владимирский государственный университет

имени Александра Григорьевича и Николая Григорьевича Столетовых»

(ВлГУ)

Компилятор подмножества процедурного языка в ассемблер

Пояснительная записка

RU. 02068048.502900.01.01

#### АННОТАЦИЯ

В данном программном документе приведён код компилятора подмножества процедурного языка. Код программы реализован на языке программирования **Python** в интегрированной среде разработки **PyCharm**.

Разработанная программа состоит из лексического анализатора, синтаксического анализатора и генератора объектного кода.

Лексический анализатор считывает текстовый файл, разбирает содержимое на токены и, затем, направляет их синтаксическому анализатору.

В свою очередь синтаксический анализатор сопоставляет поток токенов с описанной грамматикой.

Генератор объектного кода переделывает полученную программу в последовательность инструкций, которые в дальнейшем могут выполняться на машине.

Требования к входному языку:

1. Должны присутствовать операторные скобки.
2. Должна игнорироваться индентация программы.
3. Должны поддерживаться комментарии любой длины.
4. Входная программа должна представлять собой единый модуль, но также должна быть поддержка вызова функций.

Операторы:

1. Оператор присваивания.
2. Арифметика (\*, /, +, -, >, <, =).
3. Логические операторы (И, ИЛИ, НЕ).
4. Условный оператор (ЕСЛИ).
5. Операторы цикла (while, break, continue).
6. Базовый вывод (строковый литерал, переменная).
7. Типы (целочисленный 32 бита, с плавающей запятой 32 бита).

Требования к выходному языку:

1. В ассемблере.

Оформление программного документа произведено по требованиям ЕСПД (ГОСТ 19.101-77 [[1]](#footnote-1), ГОСТ 19.103-77 [[2]](#footnote-2), ГОСТ 19.104-78\* [[3]](#footnote-3), ГОСТ 19.105-78\* [[4]](#footnote-4), ГОСТ 19.106-78\* [[5]](#footnote-5), ГОСТ 19.401-78 [[6]](#footnote-6), ГОСТ 19.604-78\* [[7]](#footnote-7)).

**СОДЕРЖАНИЕ**

1. Лексический анализатор ..5

2. Синтаксический анализатор ..6

3. Транслятор ..6

4. Оптимизатор ..7

5. Работоспособность ..8

6. Код программы………………………………………………………………………..10

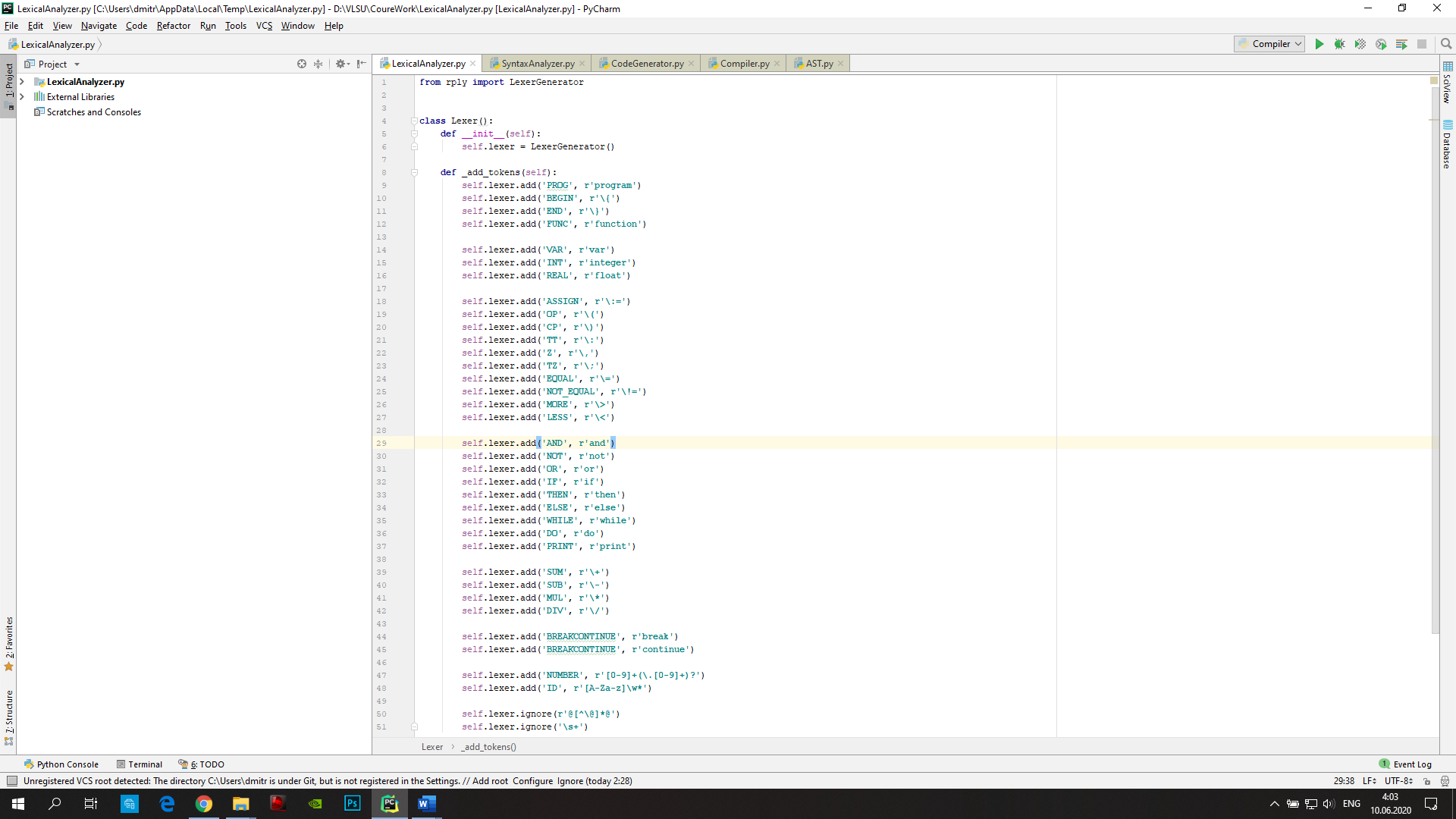
Лист регистрации изменений 21

**1. ЛЕКСИЧЕСКИЙ АНАЛИЗАТОР**

На начальном этапе содержимое текстового файла ***TestCode.txt*** разбивается на токены в соответствии с описанными правилами лексического анализатора. Результатом работы является последовательный список всех найденных лексем.

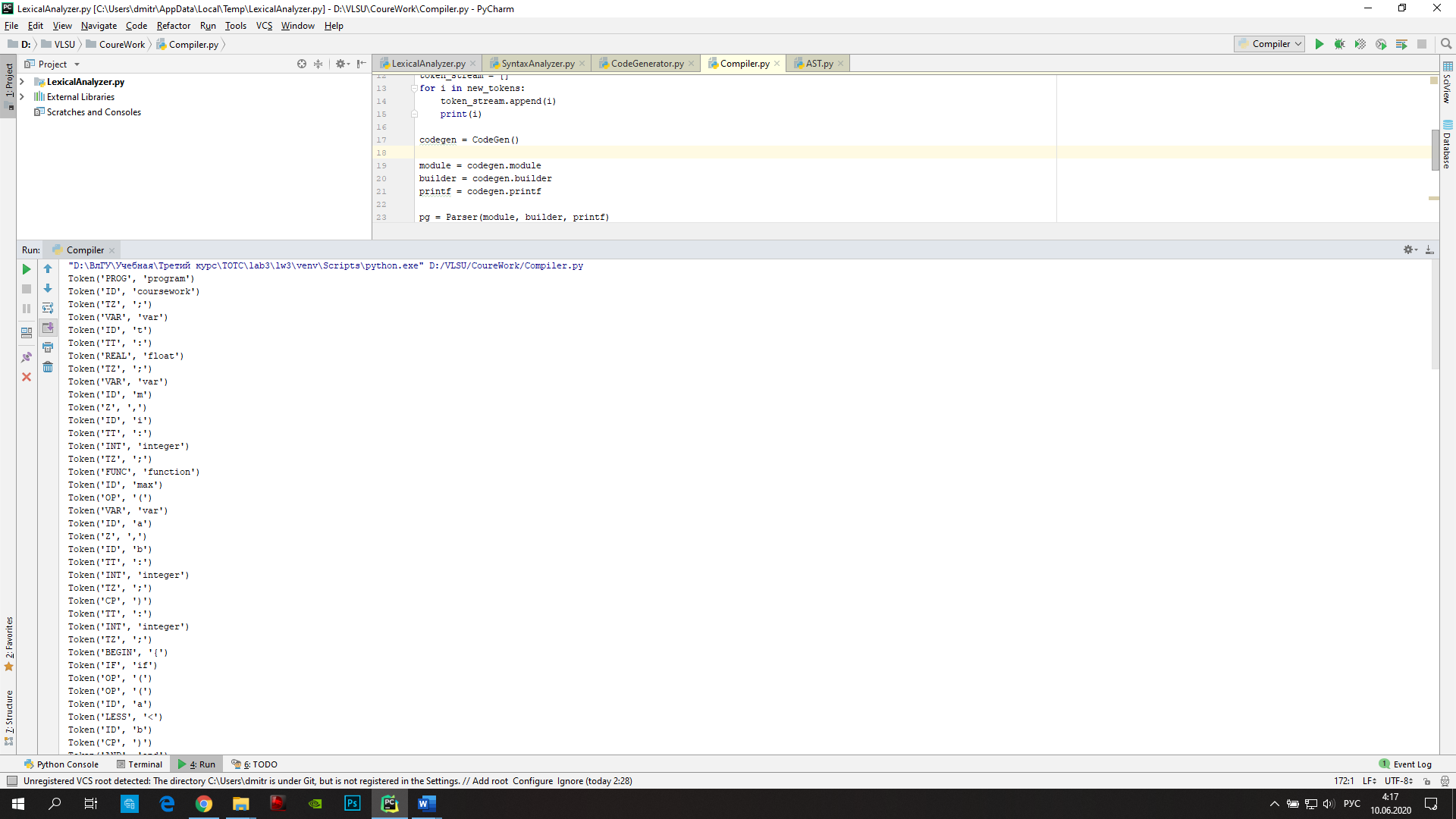
В текстовом файле имеются операторы присваивания, цикла, условные и логические, арифметика, типы и функции. Также присутствуют комментарии, которые игнорируются при разборе.

На рисунке 1 представлен небольшой пример правил разбора лексем из текстового файла.



*Рисунок 1 – разбор лексем*

А на рисунке 2 представлен пример полученных токенов.



*Рисунок 2 – полученные токены*

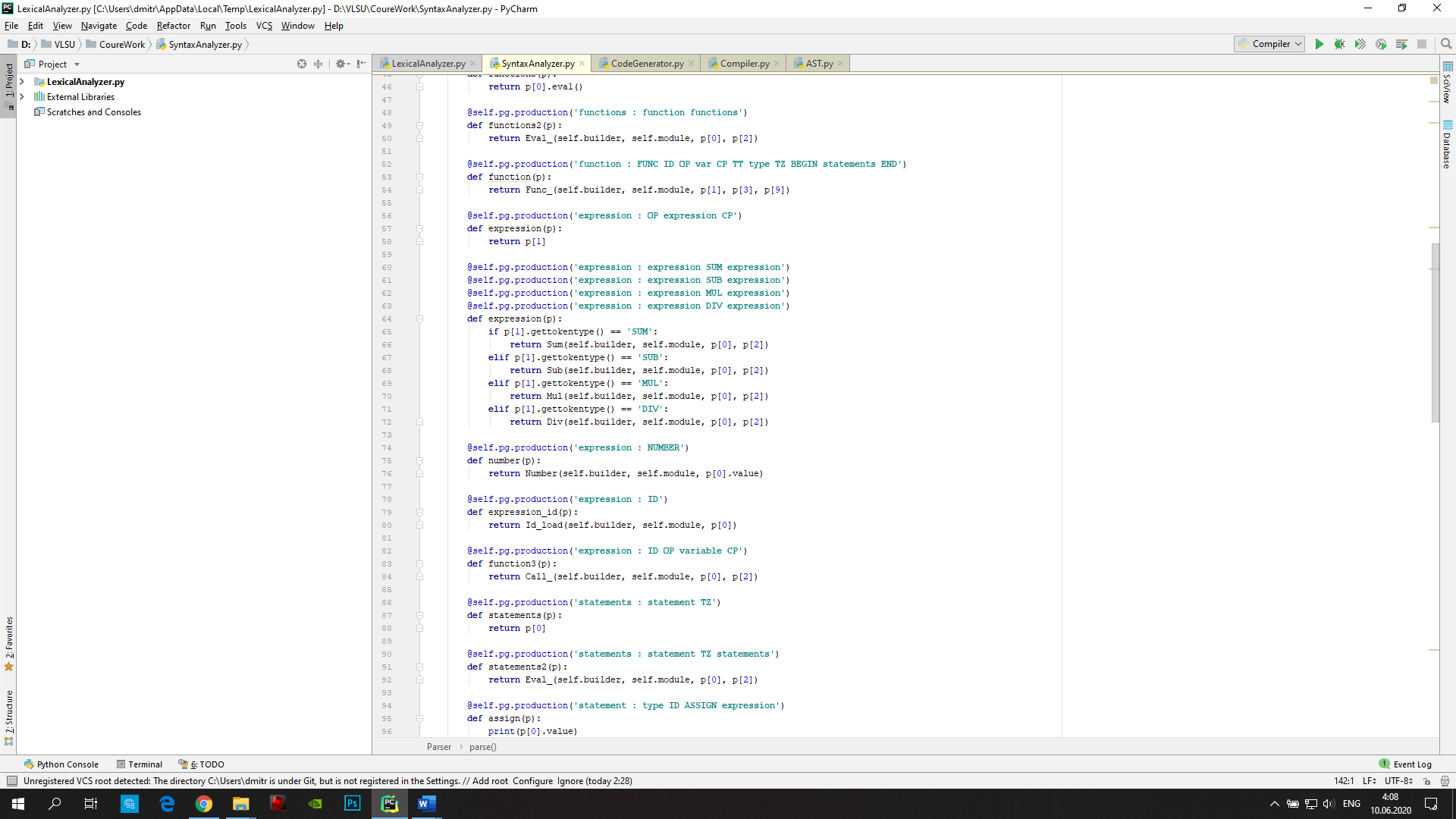
После этого полученный поток токенов направляется в синтаксический анализатор.

1. **СИНТАКСИЧЕСКИЙ АНАЛИЗАТОР**

Когда синтаксический анализатор принимает поток токенов, то он сразу начинает сопоставлять их уже с новыми с правилами – грамматики.

Для каждого токена описаны определённые наборы правил, по котором синтаксический анализатор их может обработать. При помощи этих правил, токены преобразуются в синтаксическое дерево разбора. После завершения, итоговый результат отправляется транслятору для дальнейших преобразований.

На рисунке 3 представлено одно из правил обработки потока токенов.



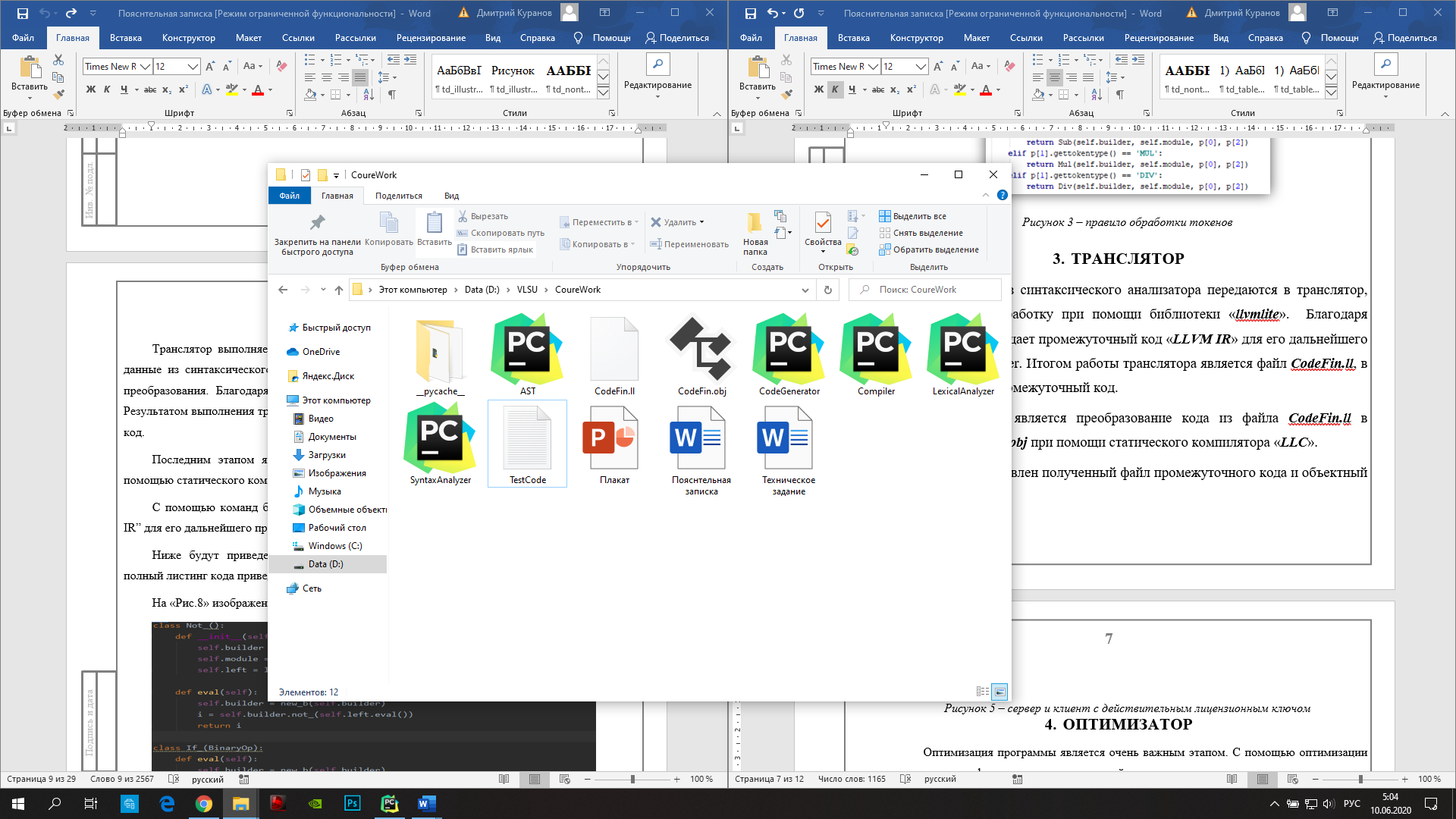
*Рисунок 3 – правило обработки токенов*

1. **ТРАНСЛЯТОР**

Выходные данные из синтаксического анализатора передаются в транслятор, который продолжает обработку при помощи библиотеки «***llvmlite***». Благодаря данной библиотеке он создает промежуточный код «***LLVM IR***» для его дальнейшего преобразования в *assembler*. Итогом работы транслятора является файл ***CodeFin.ll***, в который записывается промежуточный код.

Финальным этапом является преобразование кода из файла ***CodeFin.ll*** в объектный файл ***CodeFin.obj*** при помощи статического компилятора «***LLC***».

На рисунке 4 представлен полученный файл промежуточного кода и объектный файл.



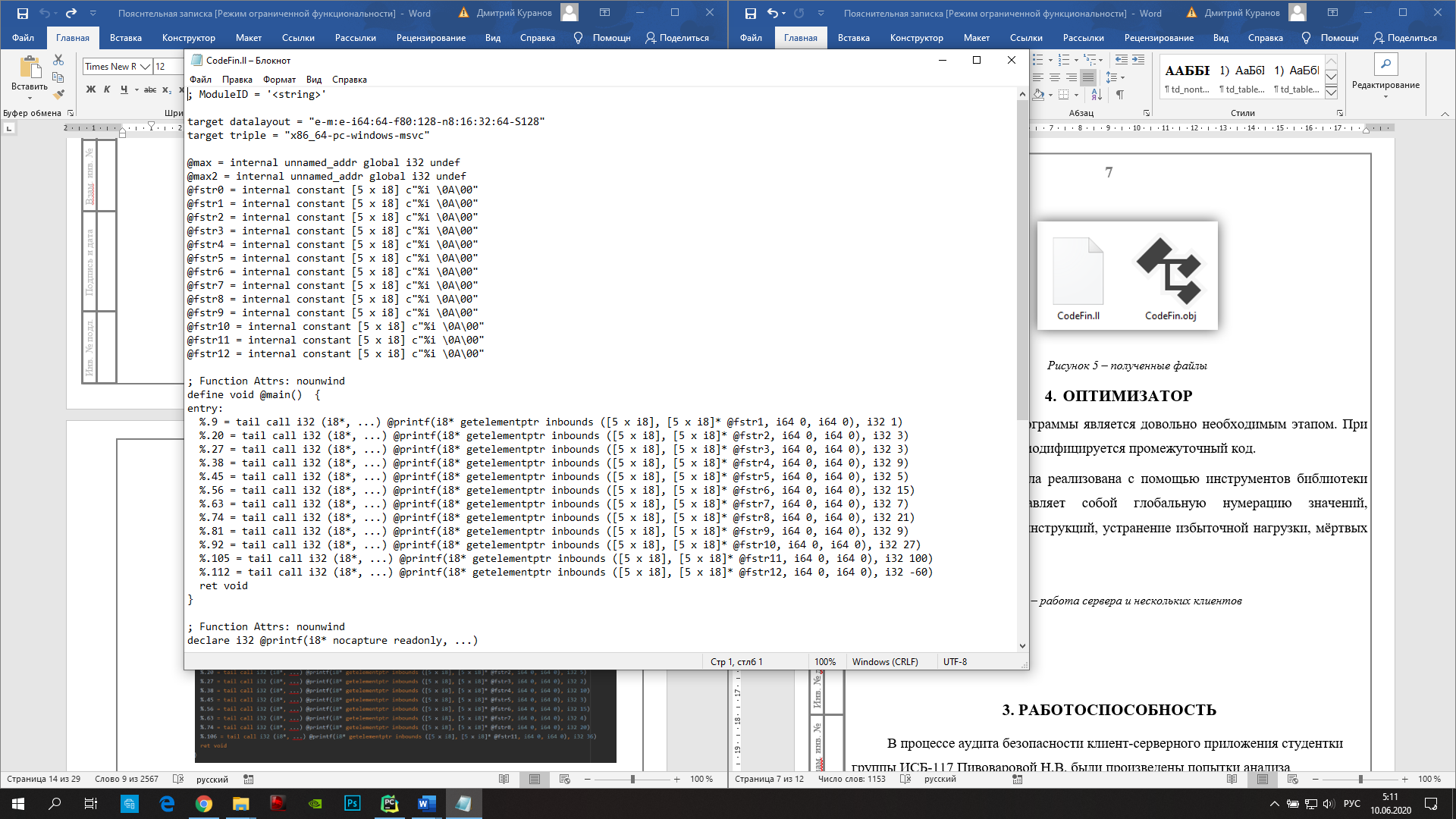
*Рисунок 4 – полученные файлы*

1. **ОПТИМИЗАТОР**

Оптимизация программы является довольно необходимым этапом. При помощи оптимизации модифицируется промежуточный код.

Оптимизация была реализована с помощью инструментов библиотеки «***llvmlite***» и представляет собой глобальную нумерацию значений, объединение лишних инструкций, устранение избыточной нагрузки, мёртвых аргументов и кода.

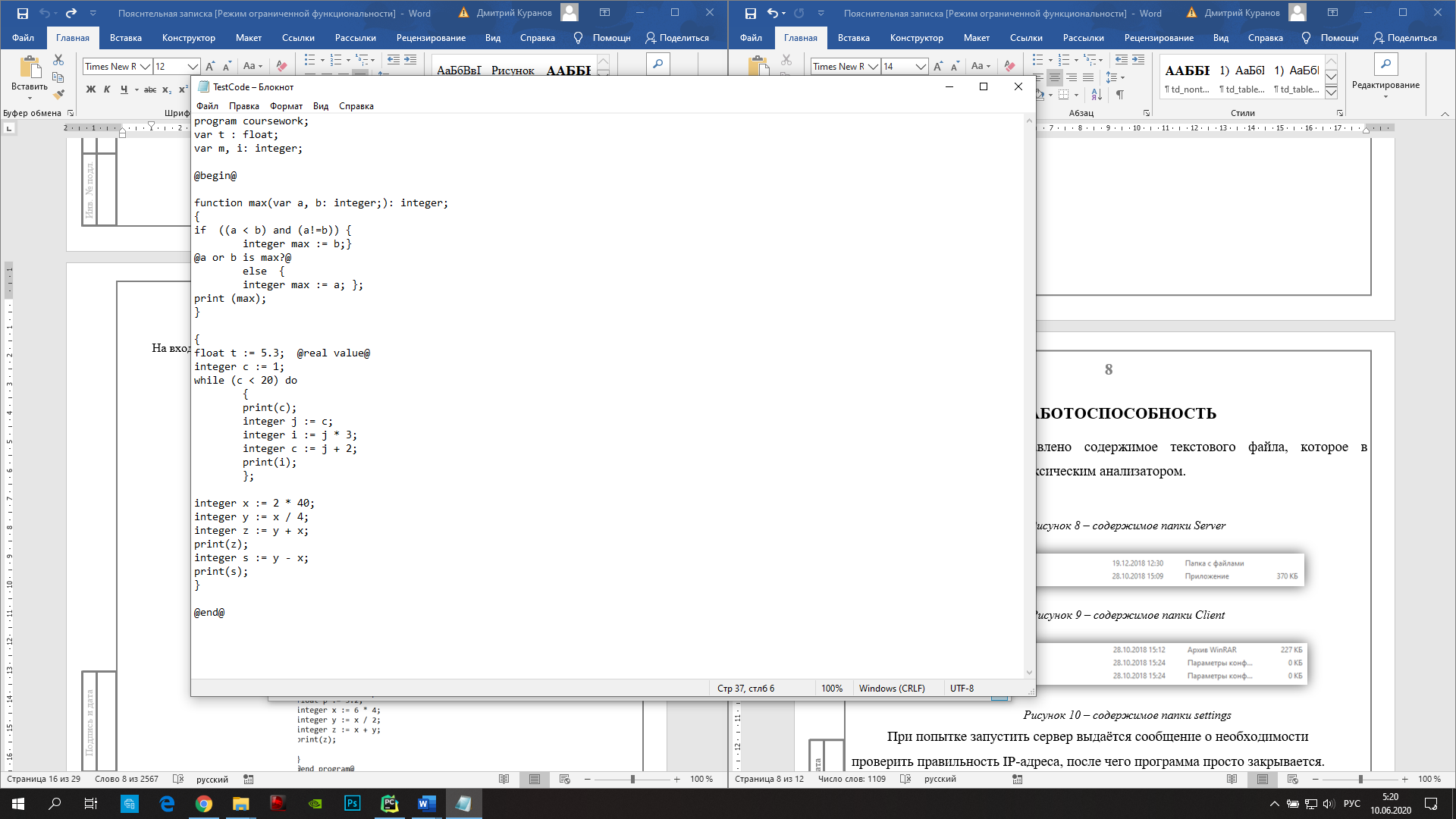
На рисунке 5 представлена небольшая часть оптимизированного кода.



*Рисунок 5 – небольшая часть оптимизированного кода*

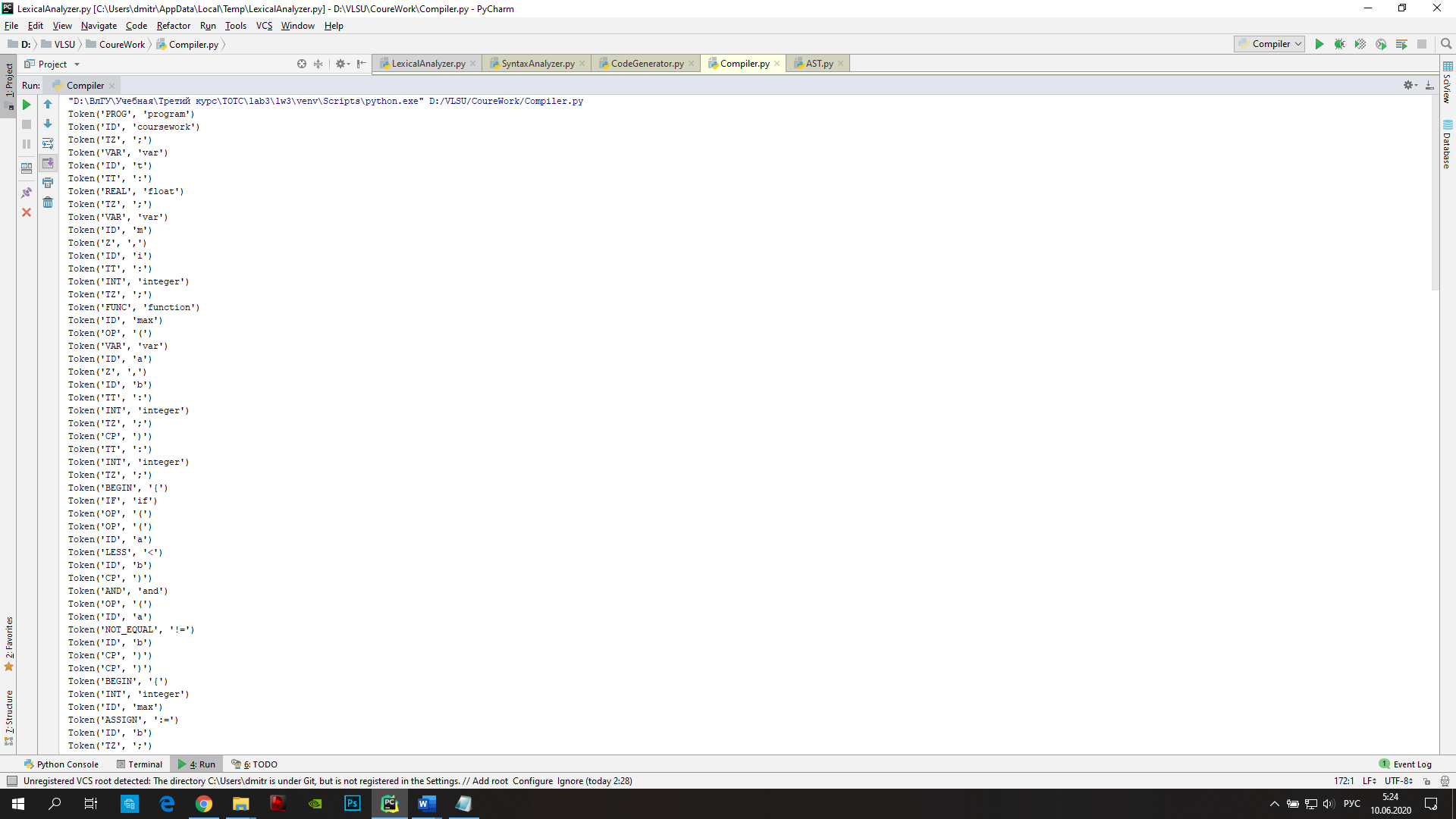
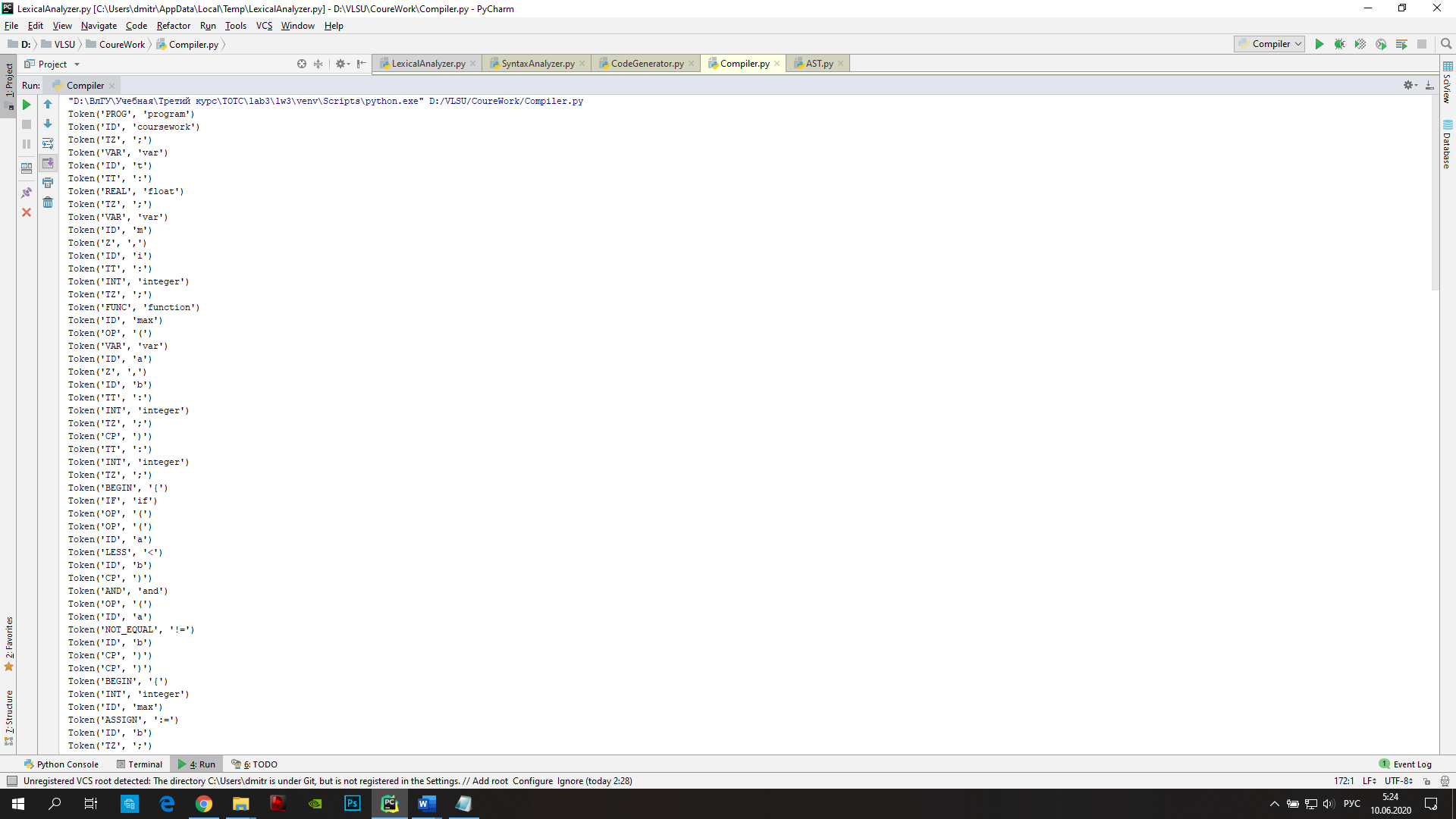
1. **РАБОТОСПОСОБНОСТЬ**

На рисунке 6 представлено содержимое текстового файла, которое в дальнейшем будет считано лексическим анализатором.



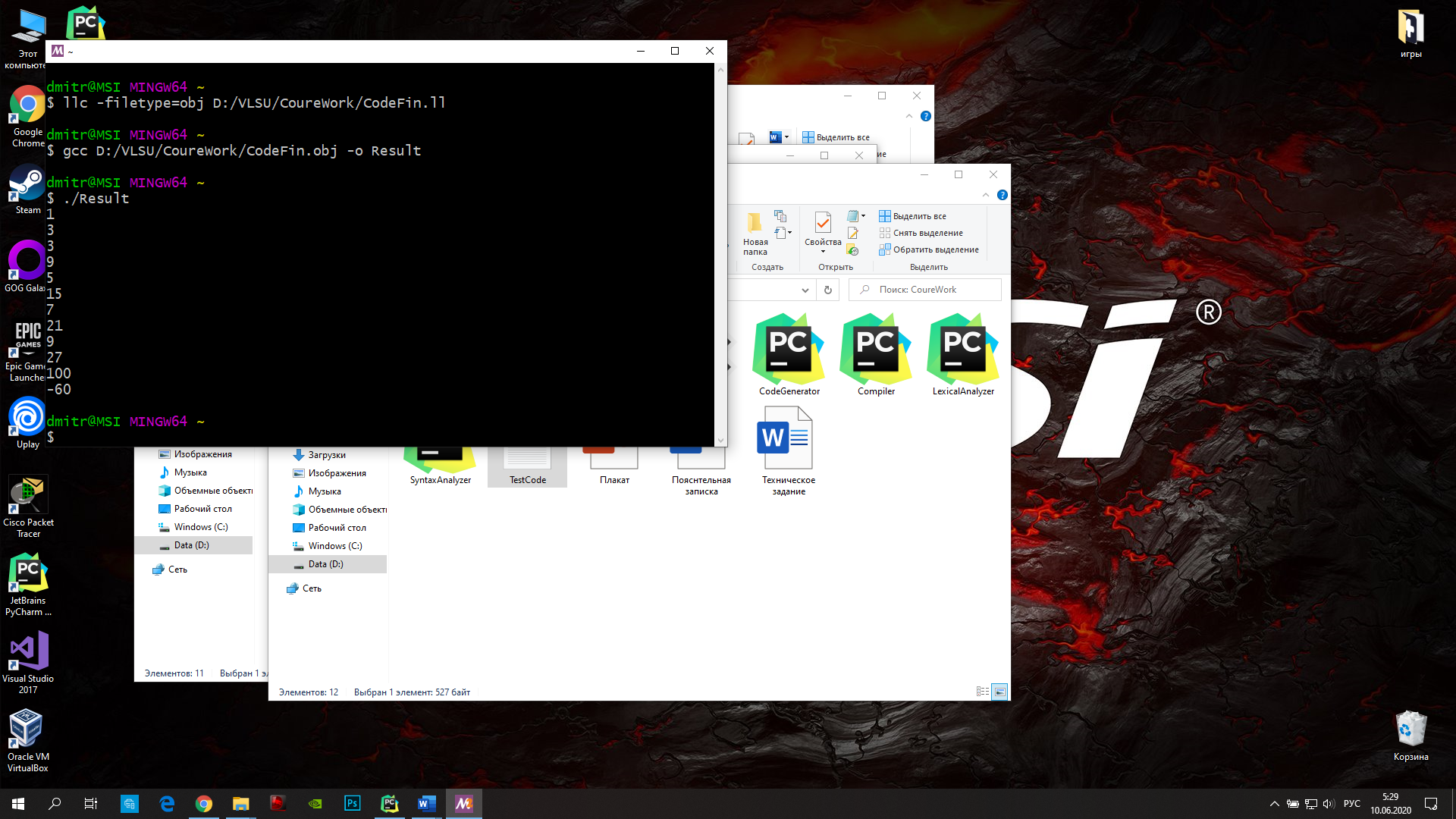
*Рисунок 6 – содержимое папки Server*

На рисунке 7 представлен (не весь) полученный поток токенов, который обработал лексический анализатор.



*Рисунок 7 – токены*

На рисунке 8 представлено конечное выполнение программы при помощи специально установленной «***GCC***».



*Рисунок 8 – результат работы программы*

Полученные значения соответствуют ручным подсчётам, благодаря чему можно сделать вывод о правильной работоспособности программы.

1. **КОД ПРОГРАММЫ**

**LexicalAnalyzer**

**from** rply **import** LexerGenerator  
  
**class** Lexer():  
 **def** \_\_init\_\_(self):  
 self.lexer = LexerGenerator()  
  
 **def** \_add\_tokens(self):  
 self.lexer.add(**'PROG'**, **r'program'**)  
 self.lexer.add(**'BEGIN'**, **r'\{'**)  
 self.lexer.add(**'END'**, **r'\}'**)  
 self.lexer.add(**'FUNC'**, **r'function'**)  
  
 self.lexer.add(**'VAR'**, **r'var'**)  
 self.lexer.add(**'INT'**, **r'integer'**)  
 self.lexer.add(**'REAL'**, **r'float'**)  
  
 self.lexer.add(**'ASSIGN'**, **r'\:='**)  
 self.lexer.add(**'OP'**, **r'\('**)  
 self.lexer.add(**'CP'**, **r'\)'**)  
 self.lexer.add(**'TT'**, **r'\:'**)  
 self.lexer.add(**'Z'**, **r'\,'**)  
 self.lexer.add(**'TZ'**, **r'\;'**)  
 self.lexer.add(**'EQUAL'**, **r'\='**)  
 self.lexer.add(**'NOT\_EQUAL'**, **r'\!='**)  
 self.lexer.add(**'MORE'**, **r'\>'**)  
 self.lexer.add(**'LESS'**, **r'\<'**)  
  
 self.lexer.add(**'AND'**, **r'and'**)  
 self.lexer.add(**'NOT'**, **r'not'**)  
 self.lexer.add(**'OR'**, **r'or'**)  
 self.lexer.add(**'IF'**, **r'if'**)  
 self.lexer.add(**'THEN'**, **r'then'**)  
 self.lexer.add(**'ELSE'**, **r'else'**)  
 self.lexer.add(**'WHILE'**, **r'while'**)  
 self.lexer.add(**'DO'**, **r'do'**)  
 self.lexer.add(**'PRINT'**, **r'print'**)  
  
 self.lexer.add(**'SUM'**, **r'\+'**)  
 self.lexer.add(**'SUB'**, **r'\-'**)  
 self.lexer.add(**'MUL'**, **r'\\*'**)  
 self.lexer.add(**'DIV'**, **r'\/'**)  
  
 self.lexer.add(**'BREAKCONTINUE'**, **r'break'**)  
 self.lexer.add(**'BREAKCONTINUE'**, **r'continue'**)  
  
 self.lexer.add(**'NUMBER'**, **r'[0-9]+(\.[0-9]+)?'**)  
 self.lexer.add(**'ID'**, **r'[A-Za-z]\w\*'**)  
  
 self.lexer.ignore(**r'@[^\@]\*@'**)  
 self.lexer.ignore(**'\s+'**)  
  
 **def** get\_lexer(self):  
 self.\_add\_tokens()  
 **return** self.lexer.build()

**SyntaxAnalyzer**

**from** rply **import** ParserGenerator  
**from** CodeGenerator **import** \*  
  
**class** Parser():  
 **def** \_\_init\_\_(self, module, builder, printf):  
 self.pg = ParserGenerator(  
 [**'PROG'**, **'BEGIN'**, **'END'**, **'OP'**, **'CP'**, **'TZ'**, **'TT'**, **'Z'**, **'FUNC'**, **'VAR'**, **'INT'**, **'REAL'**, **'NUMBER'**, **'ID'**,  
 **'ASSIGN'**, **'EQUAL'**, **'NOT\_EQUAL'**, **'AND'**, **'NOT'**, **'OR'**, **'MORE'**, **'LESS'**, **'SUM'**, **'SUB'**, **'MUL'**, **'DIV'**,  
 **'IF'**, **'ELSE'**, **'WHILE'**, **'DO'**, **'BREAKCONTINUE'**, **'PRINT'**],  
 precedence=[(**"left"**, [**'MUL'**, **'DIV'**]), (**"left"**, [**'SUM'**, **'SUB'**]) ])  
  
 self.module = module  
 self.builder = builder  
 self.printf = printf  
  
 **def** parse(self):  
  
 @self.pg.production(**'program : PROG ID TZ var functions BEGIN statements END'**)  
 **def** program(p):  
 **return** p[6]  
  
 @self.pg.production(**'var : VAR variable TT type TZ'**)  
 **def** var(p):  
 **return** p[1]  
  
 @self.pg.production(**'var : VAR variable TT type TZ var'**)  
 **def** var2(p):  
 **return** p[1], p[5]  
  
 @self.pg.production(**'variable : ID'**)  
 **def** variable(p):  
 **return** p[0]  
  
 @self.pg.production(**'variable : ID Z variable'**)  
 **def** variable2(p):  
 **return** p[0], p[2]  
  
 @self.pg.production(**'type : INT'**)  
 @self.pg.production(**'type : REAL'**)  
 **def** type(p):  
 **return** p[0]  
  
 @self.pg.production(**'functions : function'**)  
 **def** functions(p):  
 **return** p[0].eval()  
  
 @self.pg.production(**'functions : function functions'**)  
 **def** functions2(p):  
 **return** Eval\_(self.builder, self.module, p[0], p[2])  
  
 @self.pg.production(**'function : FUNC ID OP var CP TT type TZ BEGIN statements END'**)  
 **def** function(p):  
 **return** Func\_(self.builder, self.module, p[1], p[3], p[9])  
  
 @self.pg.production(**'expression : OP expression CP'**)  
 **def** expression(p):  
 **return** p[1]  
  
 @self.pg.production(**'expression : expression SUM expression'**)  
 @self.pg.production(**'expression : expression SUB expression'**)  
 @self.pg.production(**'expression : expression MUL expression'**)  
 @self.pg.production(**'expression : expression DIV expression'**)  
 **def** expression(p):  
 **if** p[1].gettokentype() == **'SUM'**:  
 **return** Sum(self.builder, self.module, p[0], p[2])  
 **elif** p[1].gettokentype() == **'SUB'**:  
 **return** Sub(self.builder, self.module, p[0], p[2])  
 **elif** p[1].gettokentype() == **'MUL'**:  
 **return** Mul(self.builder, self.module, p[0], p[2])  
 **elif** p[1].gettokentype() == **'DIV'**:  
 **return** Div(self.builder, self.module, p[0], p[2])  
  
 @self.pg.production(**'expression : NUMBER'**)  
 **def** number(p):  
 **return** Number(self.builder, self.module, p[0].value)  
  
 @self.pg.production(**'expression : ID'**)  
 **def** expression\_id(p):  
 **return** Id\_load(self.builder, self.module, p[0])  
  
 @self.pg.production(**'expression : ID OP variable CP'**)  
 **def** function3(p):  
 **return** Call\_(self.builder, self.module, p[0], p[2])  
  
 @self.pg.production(**'statements : statement TZ'**)  
 **def** statements(p):  
 **return** p[0]  
  
 @self.pg.production(**'statements : statement TZ statements'**)  
 **def** statements2(p):  
 **return** Eval\_(self.builder, self.module, p[0], p[2])  
  
 @self.pg.production(**'statement : type ID ASSIGN expression'**)  
 **def** assign(p):  
 print(p[0].value)  
 **return** Id\_save(self.builder, self.module, p[0], p[1], p[3])  
  
 @self.pg.production(**'statement : IF OP bool CP BEGIN statements END ELSE BEGIN statements END'**)  
 **def** if\_(p):  
 **return** If\_else(self.builder, self.module, p[2], p[5], p[9])  
  
 @self.pg.production(**'statement : IF OP bool CP BEGIN statements END'**)  
 **def** if\_2(p):  
 **return** If\_(self.builder, self.module, p[2], p[5])  
  
 @self.pg.production(**'statement : WHILE OP bool CP DO BEGIN statements END'**)  
 **def** while\_(p):  
 **return** While\_(self.builder, self.module, p[2], p[6])  
  
 @self.pg.production(**'statement : BREAKCONTINUE '**)  
 **def** breakcontinue(p):  
 **global** goto  
 goto = p[0].getstr()  
 **return** @self.pg.production(**'statement : PRINT OP expression CP'**)  
 **def** print\_(p):  
 **return** Print(self.builder, self.module, self.printf, p[2])  
  
 @self.pg.production(**'bool : OP bool CP'**)  
 **def** bool\_(p):  
 **return** p[1]  
  
 @self.pg.production(**'bool : expression EQUAL expression'**)  
 @self.pg.production(**'bool : expression MORE expression'**)  
 @self.pg.production(**'bool : expression LESS expression'**)  
 @self.pg.production(**'bool : expression NOT\_EQUAL expression'**)  
 **def** bool\_2(p):  
 **if** p[1].gettokentype() == **'EQUAL'**:  
 **return** Equal(self.builder, self.module, p[0], p[2])  
 **elif** p[1].gettokentype() == **'MORE'**:  
 **return** More(self.builder, self.module, p[0], p[2])  
 **elif** p[1].gettokentype() == **'LESS'**:  
 **return** Less(self.builder, self.module, p[0], p[2])  
 **elif** p[1].gettokentype() == **'NOT\_EQUAL'**:  
 **return** Not\_equal(self.builder, self.module, p[0], p[2])  
  
 @self.pg.production(**'bool : bool AND bool'**)  
 **def** and\_(p):  
 **return** And\_(self.builder, self.module, p[0], p[2])  
  
 @self.pg.production(**'bool : bool OR bool'**)  
 **def** or\_(p):  
 **return** Or\_(self.builder, self.module, p[0], p[2])  
  
 @self.pg.production(**'bool : NOT bool'**)  
 **def** not\_(p):  
 **return** Not\_(self.builder, self.module, p[1])  
  
 @self.pg.error  
 **def** error\_handle(token):  
 **raise** ValueError(token)  
  
 **def** get\_parser(self):  
 **return** self.pg.build()  
  
  
goto = **None  
  
  
class** While\_():  
 **def** \_\_init\_\_(self, builder, module, boolean, right):  
 self.builder = builder  
 self.module = module  
 self.boolean = boolean  
 self.right = right  
  
 **def** eval(self):  
 **global** goto  
 i = **None** self.builder = new\_b(self.builder)  
 **for** x **in** range(5):  
 **with** self.builder.if\_then(self.boolean.eval()):  
 **if** goto == **"break"**:  
 **break  
 elif** goto == **"continue"**:  
 **continue** i = self.right.eval()  
 **return** i

**CodeGenerator**

**from** llvmlite **import** ir, binding  
**import** sys  
  
type = **None  
  
class** Number():  
 **def** \_\_init\_\_(self, builder, module, value):  
 self.builder = builder  
 self.module = module  
 self.value = value  
  
 **def** eval(self):  
 **if** type == ir.IntType(32):  
 i = ir.Constant(ir.IntType(32), int(self.value))  
 **else**:  
 i = ir.Constant(ir.FloatType(), float(self.value))  
 **return** i  
  
  
**class** BinaryOp():  
 **def** \_\_init\_\_(self, builder, module, left, right):  
 **global** type  
 self.builder = builder  
 self.module = module  
 self.left = left  
 self.right = right  
  
  
builder\_new = **None**flag = **False  
  
def** new\_b(b):  
 **if** flag == **True**:  
 b = builder\_new  
 **return** b  
  
**class** Sum(BinaryOp):  
 **def** eval(self):  
 self.builder = new\_b(self.builder)  
 **if** type == ir.IntType(32):  
 i = self.builder.add(self.left.eval(), self.right.eval())  
 **else**:  
 i = self.builder.fadd(self.left.eval(), self.right.eval())  
 **return** i  
  
**class** Sub(BinaryOp):  
 **def** eval(self):  
 self.builder = new\_b(self.builder)  
 **if** type == ir.IntType(32):  
 i = self.builder.sub(self.left.eval(), self.right.eval())  
 **else**:  
 i = self.builder.fsub(self.left.eval(), self.right.eval())  
 **return** i  
  
**class** Mul(BinaryOp):  
 **def** eval(self):  
 self.builder = new\_b(self.builder)  
 **if** type == ir.IntType(32):  
 i = self.builder.mul(self.left.eval(), self.right.eval())  
 **else**:  
 i = self.builder.fmul(self.left.eval(), self.right.eval())  
 **return** i  
  
**class** Div(BinaryOp):  
 **def** eval(self):  
 self.builder = new\_b(self.builder)  
 **if** type == ir.IntType(32):  
 i = self.builder.sdiv(self.left.eval(), self.right.eval())  
 **else**:  
 i = self.builder.fdiv(self.left.eval(), self.right.eval())  
 **return** i  
  
**class** Equal(BinaryOp):  
 **def** eval(self):  
 self.builder = new\_b(self.builder)  
 i = self.builder.icmp\_signed(**'=='**, self.left.eval(), self.right.eval())  
 **return** i  
  
  
**class** More(BinaryOp):  
 **def** eval(self):  
 self.builder = new\_b(self.builder)  
 i = self.builder.icmp\_signed(**'>'**, self.left.eval(), self.right.eval())  
 **return** i  
  
**class** Less(BinaryOp):  
 **def** eval(self):  
 self.builder = new\_b(self.builder)  
 i = self.builder.icmp\_signed(**'<'**, self.left.eval(), self.right.eval())  
 **return** i  
  
**class** Not\_equal(BinaryOp):  
 **def** eval(self):  
 self.builder = new\_b(self.builder)  
 i = self.builder.icmp\_signed(**'!='**, self.left.eval(), self.right.eval())  
 **return** i  
  
**class** And\_(BinaryOp):  
 **def** eval(self):  
 self.builder = new\_b(self.builder)  
 i = self.builder.and\_(self.left.eval(), self.right.eval())  
 **return** i  
  
**class** Or\_(BinaryOp):  
 **def** eval(self):  
 self.builder = new\_b(self.builder)  
 i = self.builder.or\_(self.left.eval(), self.right.eval())  
 **return** i  
  
**class** Not\_():  
 **def** \_\_init\_\_(self, builder, module, left):  
 self.builder = builder  
 self.module = module  
 self.left = left  
  
 **def** eval(self):  
 self.builder = new\_b(self.builder)  
 i = self.builder.not\_(self.left.eval())  
 **return** i  
  
**class** If\_(BinaryOp):  
 **def** eval(self):  
 self.builder = new\_b(self.builder)  
 **with** self.builder.if\_then(self.left.eval()):  
 i = self.right.eval()  
 **return** i  
  
**class** If\_else():  
 **def** \_\_init\_\_(self, builder, module, boolean, left, right):  
 self.builder = builder  
 self.module = module  
 self.boolean = boolean  
 self.left = left  
 self.right = right  
  
 **def** eval(self):  
 self.builder = new\_b(self.builder)  
 **with** self.builder.if\_else(self.boolean.eval()) **as** (then, otherwise):  
 **with** then:  
 i = self.left.eval()  
 **with** otherwise:  
 y = self.right.eval()  
 **return** i  
  
**class** Eval\_(BinaryOp):  
 **def** eval(self):  
 i = self.left.eval()  
 y = self.right.eval()  
 **return** i  
  
  
values = [**None**]  
variable = [**None**] \* 100  
n = 0  
value\_num = [**None**] \* 100  
count = [1] \* 100  
  
**class** Id\_save():  
 **def** \_\_init\_\_(self, builder, module, type\_, left, right):  
 self.builder = builder  
 self.module = module  
 self.type\_ = type\_  
 self.left = left  
 self.right = right  
  
 **def** eval(self):  
 **global** variable  
 **global** n  
 **global** values  
 **global** value\_num  
 **global** count  
 i = **None** superval = self.left.value  
 values.append(superval)  
 **for** x **in** range(0, len(values) - 1):  
 **if** values[x] == self.left.value:  
 count[x] += 1  
 value\_num[x] = values[x] + str(count[x])  
 superval = value\_num[x]  
 **break  
 global** type  
 **if** self.type\_.value == **"integer"**:  
 type = ir.IntType(32)  
 **elif** self.type\_.value == **"float"**:  
 type = ir.FloatType()  
 variable[n] = ir.GlobalVariable(self.module, type, superval)  
 variable[n].linkage = **'internal'** self.builder = new\_b(self.builder)  
 i = self.builder.store(self.right.eval(), variable[n])  
 n += 1  
 **if** i == **None**:  
 sys.stderr.write(**"Error storing variable: %s\n"** % self.left.value)  
 sys.exit(1)  
 **return** i  
  
**def** num(x):  
 **for** i **in** x:  
 **if** i.isdigit():  
 **return True  
 return False  
  
class** Id\_load():  
 **def** \_\_init\_\_(self, builder, module, left):  
 self.builder = builder  
 self.module = module  
 self.left = left  
  
 **def** eval(self):  
 **global** variable  
 **global** n  
 **global** value\_num  
 **global** count  
 i = **None** check = self.left.value  
 **if** num(self.left.value):  
 check = self.left.value[:-1]  
 **if** num(check):  
 check = check[:-1]  
 **for** x **in** range(0, len(value\_num) - 1):  
 **if** value\_num[x] == check + str(count[x]):  
 check = check + str(count[x])  
 **break  
 for** x **in** range(0, n):  
 **if** (**"@\""** + check + **"\"" in** str(variable[x])):  
 self.builder = new\_b(self.builder)  
 i = self.builder.load(variable[x])  
 **if** i == **None**:  
 sys.stderr.write(**"Error loading variable: %s\n"** % self.left.value)  
 sys.exit(1)  
 **return** i  
  
  
func\_num = 0  
  
  
**class** Print():  
 **def** \_\_init\_\_(self, builder, module, printf, value):  
 self.builder = builder  
 self.module = module  
 self.printf = printf  
 self.value = value  
  
 **def** eval(self):  
 value = self.value.eval()  
 voidvariable\_ty = ir.IntType(8).as\_pointer()  
 fmt = **"%i \n\0"** c\_fmt = ir.Constant(ir.ArrayType(ir.IntType(8), len(fmt)),  
 bytearray(fmt.encode(**"utf8"**)))  
 **global** func\_num  
 global\_fmt = ir.GlobalVariable(self.module, c\_fmt.type, name=**"fstr"** + str(func\_num))  
 func\_num += 1  
 global\_fmt.linkage = **'internal'** global\_fmt.global\_constant = **True** global\_fmt.initializer = c\_fmt  
 self.builder = new\_b(self.builder)  
 fmt\_arg = self.builder.bitcast(global\_fmt, voidvariable\_ty)  
 self.builder.call(self.printf, [fmt\_arg, value])  
  
  
func\_new = **None**func\_return = 0   
  
  
**class** Func\_():  
 **def** \_\_init\_\_(self, builder, module, func\_name, param, stm):  
 self.builder = builder  
 self.module = module  
 self.func\_name = func\_name  
 self.param = param  
 self.stm = stm  
  
 **def** eval(self):  
 **global** func\_new  
 **global** builder\_new  
 **global** variable  
 **global** n  
 **global** flag  
 **global** func\_return  
 i = **None** func\_new = ir.Function(self.module, ir.FunctionType(ir.IntType(32), [ir.IntType(32), ir.IntType(32)]), name=**"function"**)  
 builder\_new = ir.IRBuilder(func\_new.append\_basic\_block(name=**"entry"**))  
 a1, a2 = func\_new.args  
 variable[n] = ir.GlobalVariable(self.module, ir.IntType(32), self.param[0].value)  
 variable[n].linkage = **'internal'** builder\_new.store(a1, variable[n])  
 n += 1  
 variable[n] = ir.GlobalVariable(self.module, ir.IntType(32), self.param[1].value)  
 variable[n].linkage = **'internal'** builder\_new.store(a2, variable[n])  
 n += 1  
 flag = **True** y = self.stm.eval()  
 flag = **False** func\_return = self.func\_name.value  
 **for** x **in** range(0, n):  
 **if** (**"@\""** + self.func\_name.value + **"\"" in** str(variable[x])):  
 i = builder\_new.load(variable[x])  
 ii = builder\_new.ret(i)  
 **if** ii == **None**:  
 sys.stderr.write(**"Error returning in function"**)  
 sys.exit(1)  
 **return** ii  
  
  
**class** Call\_():  
 **def** \_\_init\_\_(self, builder, module, func\_name, param):  
 self.builder = builder  
 self.module = module  
 self.func\_name = func\_name  
 self.param = param  
  
 **def** eval(self):  
 **global** func\_new  
 self.builder = new\_b(self.builder)  
 **for** x **in** range(0, n):  
 **if** (**"@\""** + self.param[0].value + **"\"" in** str(variable[x])):  
 a1 = self.builder.load(variable[x])  
 **for** x **in** range(0, n):  
 **if** (**"@\""** + self.param[1].value + **"\"" in** str(variable[x])):  
 a2 = self.builder.load(variable[x])  
 i = self.builder.call(func\_new, [a1, a2])  
 **return** i  
  
  
**class** CodeGen():  
 **def** \_\_init\_\_(self):  
 self.binding = binding  
 self.binding.initialize()  
 self.binding.initialize\_native\_target()  
 self.binding.initialize\_native\_asmprinter()  
 self.\_config\_llvm()  
 self.\_create\_execution\_engine()  
 self.\_declare\_print\_function()  
  
 **def** \_config\_llvm(self):  
 **global** base\_func  
 self.module = ir.Module(name=\_\_file\_\_)  
 self.module.triple = self.binding.get\_default\_triple()  
  
 func\_type = ir.FunctionType(ir.VoidType(), [], **False**)  
 base\_func = ir.Function(self.module, func\_type, name=**"main"**)  
 block = base\_func.append\_basic\_block(name=**"entry"**)  
 self.builder = ir.IRBuilder(block)  
  
 **def** \_create\_execution\_engine(self):  
 target = self.binding.Target.from\_default\_triple()  
 target\_machine = target.create\_target\_machine()  
 backing\_mod = binding.parse\_assembly(**""**)  
 engine = binding.create\_mcjit\_compiler(backing\_mod, target\_machine)  
 self.engine = engine  
  
 **def** \_declare\_print\_function(self):  
 voidptr\_ty = ir.IntType(8).as\_pointer()  
 printf\_ty = ir.FunctionType(ir.IntType(32), [voidptr\_ty], var\_arg=**True**)  
 printf = ir.Function(self.module, printf\_ty, name=**"printf"**)  
 self.printf = printf  
  
 **def** \_compile\_ir(self):  
 self.builder.ret\_void()  
  
 pmb = self.binding.create\_pass\_manager\_builder()  
 pm = self.binding.create\_module\_pass\_manager()  
 pm.add\_constant\_merge\_pass()  
 pm.add\_dead\_arg\_elimination\_pass()  
  
 pm.add\_dead\_code\_elimination\_pass()  
 pm.add\_gvn\_pass()  
 pm.add\_instruction\_combining\_pass()  
 pmb.populate(pm)  
  
 llvm\_ir = str(self.module)  
 mod = self.binding.parse\_assembly(llvm\_ir)  
 x = pm.run(mod)  
  
 mod.verify()  
 self.engine.add\_module(mod)  
 self.engine.finalize\_object()  
 self.engine.run\_static\_constructors()  
  
 self.module = mod  
 **return** mod  
  
 **def** create\_ir(self):  
 self.\_compile\_ir()  
  
 **def** save\_ir(self, filename):  
 S = str(self.module).replace(**r'local\_unnamed\_addr #0'**, **''**)  
 S = S.replace(**r'local\_unnamed\_addr #1'**, **''**)  
 S = S.replace(**'source\_filename = "<string>"'**, **''**)  
 S = S.replace(**'x86\_64-pc-win32'**, **'x86\_64-pc-linux-gnu'**)  
 **with** open(filename, **'w'**) **as** output\_file:  
 output\_file.write(S)

**Compiler**

**from** CodeGenerator **import** CodeGen  
**from** LexicalAnalyzer **import** Lexer  
**from** SyntaxAnalyzer **import** Parser  
  
**with** open(**"TestCode.txt"**) **as** file:  
 text\_input = file.read()  
  
lexer = Lexer().get\_lexer()  
tokens = lexer.lex(text\_input)  
new\_tokens = lexer.lex(text\_input)  
  
token\_stream = []  
**for** i **in** new\_tokens:  
 token\_stream.append(i)  
 print(i)  
  
codegen = CodeGen()  
  
module = codegen.module  
builder = codegen.builder  
printf = codegen.printf  
  
pg = Parser(module, builder, printf)  
pg.parse()  
parser = pg.get\_parser()  
parse = parser.parse(tokens).eval()  
  
codegen.create\_ir()  
codegen.save\_ir(**"CodeFin.ll"**)

**Команды из терминала**

llc -filetype=obj D:/VLSU/CoureWork/CodeFin.ll

gcc D:/VLSU/CoureWork/CodeFin.obj -o Result

./Result

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Лист регистрации изменений | | | | | | | | | |
| Номера листов (страниц) | | | | | Всего  листов  (страниц)  в докум | №  документа | Входящий  № сопрово  дительного  документа  и дата | Подп. | Дата |
| Изм | изменен  ных | заме  ненных | новых | Анулиро  ванных |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |

1. ГОСТ 19.101-77 ЕСПД. Виды программ и программных документов [↑](#footnote-ref-1)
2. ГОСТ 19.103-77 ЕСПД. Обозначение программ и программных документов [↑](#footnote-ref-2)
3. ГОСТ 19.104-78\* ЕСПД. Основные надписи [↑](#footnote-ref-3)
4. ГОСТ 19.105-78\* ЕСПД. Общие требования к программным документам [↑](#footnote-ref-4)
5. ГОСТ 19.106-78\* ЕСПД. Общие требования к программным документам, выполненным печатным способом [↑](#footnote-ref-5)
6. ГОСТ 19.401-78 ЕСПД. Текст программы. Требования к содержанию и оформлению [↑](#footnote-ref-6)
7. ГОСТ 19.604-78\* ЕСПД. Правила внесения изменений в программные документы, выполненные печатным способом [↑](#footnote-ref-7)