МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

НИЖЕГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ

УНИВЕРСИТЕТ им. Р.Е.АЛЕКСЕЕВА

Институт радиоэлектроники и информационных технологий

Кафедра информатики и систем управления

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(наименование темы проекта или работы)

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовой работе

по дисциплине

Теория компиляции

(наименование дисциплины)

РУКОВОДИТЕЛЬ:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(подпись) (фамилия, и.,о.)

СТУДЕНТ:

Шеронов И.С. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(подпись) (фамилия, и.,о.)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(шифр группы)

Работа защищена «\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_

С оценкой \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Нижний Новгород 2023

**Оглавление**

Введение

Глава 1 Лексический анализатор код...................................................................4

Глава 2. Синтаксический анализатор код............................................................7

Глава 3. Генератор кода.......................................................................................15

Глава 4. Тесты.......................................................................................................20

4.1 Тест 1……………………………………………………………………...20

4.2 Тест 2……………………………………………………………………...23

4.3 Тест 3……………………………………………………………………...25

Заключение...........................................................................................................27

**Введение**

**Объектом** курсового исследования является компиляция и входящие в этот процесс анализаторы

**Предметом** ‒ процесс преобразования программного кода из одного языка программирования в другой. Компилятор — это программа, которая проводит компиляцию.

**Целью** курсового исследования является изучение процесса компиляции.

Для достижения цели исследования были поставлены следующие **задачи**:

‒ написать лексический анализатор;

‒ написать синтаксический анализатор

‒ написать семантический анализатор

‒ написать генератор кода

**Лексический анализатор код**

PRINT, BEGIN, END, IF, THEN, ELSE, \  
 WHILE, OR, AND, NOT, READ, PROGRAM, \  
 VAR, INT, BOOLEAN, \  
 TRUE, FALSE, DO, \  
 FUNCTION = ('print', 'begin',  
 'end', 'if', 'then',  
 'else', 'while',  
 'or', 'and', 'not', 'read',  
 'program', 'var', 'int',  
 'boolean', 'true', 'false', 'function', 'do')  
  
ALPHABET = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '+', '-',  
 '\*', '/', '(', ')', ':', '=', ';', '<', '>', '!', '.', ',']  
KEYWORDS = [PRINT, BEGIN, END, IF, THEN, ELSE, WHILE,  
 OR, AND, NOT, READ, PROGRAM, VAR,  
 INT, BOOLEAN, TRUE, FALSE, FUNCTION, DO]  
  
TYPES = [INT, BOOLEAN]  
  
BOOL = [TRUE, FALSE]  
  
  
class Tokenizer():  
 def \_\_init\_\_(self, origin):  
 self.origin = origin  
 self.position = 0  
 self.actual = None  
 self.alphabet = ALPHABET  
  
 def selectNext(self):  
 # Final do arquivo  
 if self.position >= len(self.origin):  
 self.actual = None  
 return None  
 char = self.origin[self.position]  
 # Comentarios  
 if char == '{':  
 while char != '}':  
 self.position += 1  
 if self.position == len(self.origin):  
 return None  
 char = self.origin[self.position]  
 self.position += 1  
 char = self.origin[self.position]  
 # Espacos,enter e tabs  
 while char.isspace() and self.position:  
 self.position += 1  
 if self.position == len(self.origin):  
 return None  
 char = self.origin[self.position]  
 # Identificador  
 if char.isalpha():  
 identifier = char  
 while True:  
 self.position += 1  
 if (self.position >= len(self.origin)):  
 break  
 char = self.origin[self.position]  
 if not char.isalpha() and not char.isdigit() and char != '\_':  
 break  
 else:  
 identifier += char  
 if identifier in TYPES:  
 self.actual = Token('TYPE', identifier)  
 elif identifier in BOOL:  
 self.actual = Token('boolean', identifier)  
 elif identifier in KEYWORDS:  
 self.actual = Token(identifier, None)  
 else:  
 self.actual = Token('IDE', identifier)  
 # Caracter invalido  
 elif char not in self.alphabet:  
 raise ValueError("Invalid Char")  
 # Digitos  
 elif char.isdigit():  
 number = char  
 while True:  
 self.position += 1  
 if (self.position >= len(self.origin)):  
 break  
 char = self.origin[self.position]  
 if not char.isdigit():  
 break  
 else:  
 number += char  
 self.actual = Token('int', int(number))  
 # Operacoes e parenteses  
 else:  
 if (char == '+'):  
 self.actual = Token('PLUS', None)  
 elif (char == '-'):  
 self.actual = Token('MINUS', None)  
 elif (char == '\*'):  
 self.actual = Token('MULT', None)  
 elif (char == '/'):  
 self.actual = Token('DIV', None)  
 elif (char == '('):  
 self.actual = Token('OPEN\_PAR', None)  
 elif (char == ')'):  
 self.actual = Token('CLOSE\_PAR', None)  
 elif (char == ';'):  
 self.actual = Token('SEMI\_COLON', None)  
 elif (char == ':'):  
 if (self.position + 1 < len(self.origin)):  
 char = self.origin[self.position + 1]  
 if (char == '='):  
 self.actual = Token('ATRIBUTE', None)  
 self.position += 1  
 else:  
 self.actual = Token('VAR\_DECLARATION', None)  
 elif (char == '>'):  
 self.actual = Token('COMP', ">")  
 elif (char == '<'):  
 self.actual = Token('COMP', "<")  
 elif (char == '='):  
 self.actual = Token('COMP', "=")  
 elif (char == '!'):  
 self.position += 1  
 char = self.origin[self.position]  
 if (char == '='):  
 self.actual = Token('COMP', "!=")  
 else:  
 raise ValueError("Invalid Char")  
 elif (char == '.'):  
 self.actual = Token('END\_PROGRAM', None)  
 elif (char == ','):  
 self.actual = Token('COMMA', None)  
 self.position += 1  
 return self.actual

**Синтаксический анализатор код**

class Parser():  
 def \_\_init\_\_(self, origin):  
 self.tokens = Tokenizer(origin)  
 self.tokens.selectNext()  
  
 def parseProgram(self):  
 token = self.tokens.actual  
 if token.type == "program":  
 token = self.tokens.selectNext()  
 if token.type == "IDE":  
 name\_program = token.value  
 token = self.tokens.selectNext()  
 if token.type == "SEMI\_COLON":  
 self.tokens.selectNext()  
 variables = self.parseVariables()  
 functions = self.parseFunctions()  
 statements = self.parseStatements()  
 result = Program(name\_program,  
 [variables, functions, statements])  
 token = self.tokens.actual  
 if token.type == "END\_PROGRAM":  
 pass  
 else:  
 raise ValueError("Invalid token, expecting a . on position \  
 {}".format(self.tokens.position))  
 else:  
 raise ValueError("Invalid token, expecting a semi colon \  
 or a end on position {}".format(self.tokens.position))  
 else:  
 raise ValueError("Invalid token, expecting a program on \  
 position {}".format(self.tokens.position))  
 return result  
  
 def parseFunctionCall(self):  
 pass  
  
 def parseFunctions(self):  
 token = self.tokens.actual  
 result = Funcs(None, [])  
 while True:  
 if token.type == "function":  
 token = self.tokens.selectNext()  
 if token.type == "IDE":  
 function\_name = token.value  
 func = FuncDec(function\_name, [])  
 self.tokens.selectNext()  
 arguments = self.parseArgumentsFunction(function\_name)  
 self.tokens.selectNext()  
 variables = self.parseVariables()  
 functions = self.parseFunctions()  
 statements = self.parseStatements()  
 func.children.append(arguments)  
 func.children.append(variables)  
 func.children.append(functions)  
 func.children.append(statements)  
 result.children.append(func)  
 token = self.tokens.actual  
 else:  
 raise ValueError("Invalid token, expecting a identifier on position \  
 {}".format(self.tokens.position))  
 elif token.type == "begin":  
 return result  
 else:  
 raise ValueError("Invalid token, expecting a function on position \  
 {}".format(self.tokens.position))  
  
 def parseArgumentsFunction(self, function\_name):  
 token = self.tokens.actual  
 if token.type == "OPEN\_PAR":  
 list\_arguments = []  
 while True:  
 token = self.tokens.selectNext()  
 if token.type == "IDE":  
 list\_arguments.append(token.value)  
 token = self.tokens.selectNext()  
 if token.type == "VAR\_DECLARATION":  
 break  
 elif token.type == "COMMA":  
 pass  
 else:  
 raise ValueError("Invalid token, expecting a : or , on position \  
 {}".format(self.tokens.position))  
 else:  
 raise ValueError("Invalid token, expecting a identifier on position \  
 {}".format(self.tokens.position))  
 token = self.tokens.selectNext()  
 if token.type == "TYPE":  
 arguments = MultiVarDec(None, [])  
 for var\_name in list\_arguments:  
 var\_name = StrVal(var\_name, [])  
 value = StrVal(token.value, [])  
 variable = BinOp(":", [var\_name, value])  
 arguments.children.append(variable)  
 token = self.tokens.selectNext()  
 if token.type == "CLOSE\_PAR":  
 token = self.tokens.selectNext()  
 if token.type == "VAR\_DECLARATION":  
 token = self.tokens.selectNext()  
 if token.type == "TYPE":  
 return\_var\_name = StrVal(function\_name, [])  
 return\_type = StrVal(token.value, [])  
 variable = BinOp(":", [return\_var\_name,  
 return\_type])  
 arguments.children.append(variable)  
 token = self.tokens.selectNext()  
 if token.type == "SEMI\_COLON":  
 return arguments  
 else:  
 raise ValueError("Invalid token, expecting a ; on position \  
 {}".format(self.tokens.position))  
 else:  
 raise ValueError("Invalid token, expecting a type on position \  
 {}".format(self.tokens.position))  
 else:  
 raise ValueError("Invalid token, expecting a : on position \  
 {}".format(self.tokens.position))  
 else:  
 raise ValueError("Invalid token, expecting a ) on position \  
 {}".format(self.tokens.position))  
 else:  
 raise ValueError("Invalid token, expecting a type on position \  
 {}".format(self.tokens.position))  
 else:  
 raise ValueError("Invalid token, expecting a ( on position \  
 {}".format(self.tokens.position))  
  
 def parseVariables(self):  
 token = self.tokens.actual  
 result = MultiVarDec(None, [])  
 if token.type != "begin":  
 if token.type == "var":  
 token = self.tokens.selectNext()  
 while True:  
 list\_vars = []  
 while True:  
 if token.type == "IDE":  
 list\_vars.append(token.value)  
 token = self.tokens.selectNext()  
 if token.type == "COMMA":  
 token = self.tokens.selectNext()  
 elif token.type == "VAR\_DECLARATION":  
 break  
 else:  
 raise ValueError("Invalid token, expecting a , or : on position \  
 {}".format(self.tokens.position))  
 else:  
 raise ValueError("Invalid token, expecting a identifier on position \  
 {}".format(self.tokens.position))  
 token = self.tokens.selectNext()  
 if token.type == "TYPE":  
 for var\_name in list\_vars:  
 var\_name = StrVal(var\_name, [])  
 value = StrVal(token.value, [])  
 variable = VarDec(None, [var\_name, value])  
 result.children.append(variable)  
 token = self.tokens.selectNext()  
 if token.type == "SEMI\_COLON":  
 token = self.tokens.selectNext()  
 if token.type == "begin":  
 break  
 elif token.type == "function":  
 break  
 elif token.type == "IDE":  
 pass  
 else:  
 raise ValueError("Invalid token, expecting a begin \  
 or identifier on position {}"  
 .format(self.tokens.position))  
 else:  
 raise ValueError("Invalid token, expecting a ; on position \  
 {}".format(self.tokens.position))  
 else:  
 raise ValueError("Invalid token, expecting a type on position \  
 {}".format(self.tokens.position))  
 else:  
 raise ValueError("Invalid token, expecting a var on position \  
 {}".format(self.tokens.position))  
 return result  
  
 def parseStatements(self):  
 token = self.tokens.actual  
 if token.type == "begin":  
 result = Statements(None, [])  
 while True:  
 self.tokens.selectNext()  
 result.children.append(self.parseStatement())  
 token = self.tokens.actual  
 if token.type == "SEMI\_COLON":  
 pass  
 elif token.type == "end":  
 break  
 if self.tokens.actual.type == "end":  
 self.tokens.selectNext()  
 pass  
 else:  
 raise ValueError("Invalid token, expecting a end on \  
 position {}".format(self.tokens.position))  
 else:  
 raise ValueError("Invalid token, expecting a begin on \  
 position {}".format(self.tokens.position))  
 return result  
  
 def parseStatement(self):  
 token = self.tokens.actual  
 if token.type == "begin":  
 result = self.parseStatements()  
 elif token.type == "IDE":  
 result = self.parseAtribution()  
 elif token.type == "print":  
 result = self.parsePrint()  
 elif token.type == "if":  
 result = self.parseIf()  
 elif token.type == "while":  
 result = self.parseWhile()  
 else:  
 raise ValueError("Invalid token, expecting a begin,identifier, print, if or while \  
 on position {}".format(self.tokens.position))  
 return result  
  
 def parseAtribution(self):  
 value1 = StrVal(self.tokens.actual.value, [])  
 token = self.tokens.selectNext()  
 if (token.type == "ATRIBUTE"):  
 token = self.tokens.selectNext()  
 if (token.type == "read"):  
 value2 = self.parseRead()  
 else:  
 value2 = self.parseExpression()  
 result = Assignment(None, [value1, value2])  
 else:  
 raise ValueError("Invalid token, expecting a := on position {}"  
 .format(self.tokens.position))  
 return result  
  
 def parsePrint(self):  
 token = self.tokens.selectNext()  
 if token.type == "OPEN\_PAR":  
 self.tokens.selectNext()  
 value = self.parseExpression()  
 token = self.tokens.actual  
 if token.type == "CLOSE\_PAR":  
 result = Print(value, [value])  
 self.tokens.selectNext()  
 else:  
 raise ValueError("Invalid token, expecting a ) on position {}"  
 .format(self.tokens.position))  
 else:  
 raise ValueError("Invalid token, expecting a ( on position {}"  
 .format(self.tokens.position))  
 return result  
  
 def parseRelExpression(self):  
 self.tokens.selectNext()  
 value1 = self.parseExpression()  
 token = self.tokens.actual  
 if token.type == 'COMP':  
 self.tokens.selectNext()  
 value2 = self.parseExpression()  
 result = BinOp(token.value, [value1, value2])  
 else:  
 raise ValueError("Invalid token, expecting a <, >, = or != \  
 on position {}".format(self.tokens.position))  
 return result  
  
 def parseIf(self):  
 comp = self.parseRelExpression()  
 token = self.tokens.actual  
 if (token.type == "then"):  
 self.tokens.selectNext()  
 statement1 = self.parseStatement()  
 token = self.tokens.actual  
 if (token.type == "else"):  
 self.tokens.selectNext()  
 statement2 = self.parseStatement()  
 else:  
 statement2 = NoOp(None, [])  
 result = If(None, [comp, statement1, statement2])  
 else:  
 raise ValueError("Invalid token, expecting a then on \  
 position {}".format(self.tokens.position))  
 return result  
  
 def parseRead(self):  
 token = self.tokens.selectNext()  
 if token.type == "OPEN\_PAR":  
 self.tokens.selectNext()  
 token = self.tokens.actual  
 if token.type == "CLOSE\_PAR":  
 result = Read(None, [])  
 self.tokens.selectNext()  
 else:  
 raise ValueError("Invalid token, expecting a ) on position {}"  
 .format(self.tokens.position))  
 else:  
 raise ValueError("Invalid token, expecting a ( on position {}"  
 .format(self.tokens.position))  
 return result  
  
 def parseWhile(self):  
 comp = self.parseRelExpression()  
 token = self.tokens.actual  
 if (token.type == "do"):  
 self.tokens.selectNext()  
 statement1 = self.parseStatement()  
 token = self.tokens.actual  
 result = While(None, [comp, statement1])  
 else:  
 raise ValueError("Invalid token, expecting a do on \  
 position {}".format(self.tokens.position))  
 return result  
  
 def parseExpression(self):  
 result = self.parseTerm()  
 while True:  
 token = self.tokens.actual  
 if token is None:  
 break  
 if token.type == "PLUS":  
 self.tokens.selectNext()  
 second\_value = self.parseTerm()  
 result = BinOp("+", [result, second\_value])  
 elif token.type == "MINUS":  
 self.tokens.selectNext()  
 second\_value = self.parseTerm()  
 result = BinOp("-", [result, second\_value])  
 elif token.type == "or":  
 self.tokens.selectNext()  
 second\_value = self.parseTerm()  
 result = BinOp("or", [result, second\_value])  
 else:  
 break  
 return result  
  
 def parseTerm(self):  
 result = self.parseFactor()  
 while True:  
 token = self.tokens.actual  
 if token is None:  
 break  
 elif token.type == "MULT":  
 self.tokens.selectNext()  
 second\_value = self.parseFactor()  
 result = BinOp("\*", [result, second\_value])  
 elif token.type == "DIV":  
 self.tokens.selectNext()  
 second\_value = self.parseFactor()  
 result = BinOp("/", [result, second\_value])  
 elif token.type == "and":  
 self.tokens.selectNext()  
 second\_value = self.parseFactor()  
 result = BinOp("and", [result, second\_value])  
 else:  
 break  
 return result  
  
 def parseFactor(self):  
 token = self.tokens.actual  
 if token is None:  
 raise ValueError("Invalid token, expecting a number or opening parentesis on \  
 position {}, got NULL".format(self.tokens.position))  
 if token.type == "int":  
 result = IntVal(token.value, [])  
 self.tokens.selectNext()  
 elif token.type == "boolean":  
 result = BoolVal(token.value, [])  
 self.tokens.selectNext()  
 elif token.type == "OPEN\_PAR":  
 self.tokens.selectNext()  
 result = self.parseExpression()  
 token = self.tokens.actual  
 if token.type != "CLOSE\_PAR":  
 raise ValueError("Invalid token, missing parentesis close on \  
 position {}".format(self.tokens.position))  
 elif token.type == "MINUS":  
 self.tokens.selectNext()  
 result = self.parseFactor()  
 result = UnOp("-", [result])  
 elif token.type == "not":  
 self.tokens.selectNext()  
 result = self.parseFactor()  
 result = UnOp("not", [result])  
 elif token.type == "PLUS":  
 self.tokens.selectNext()  
 result = self.parseFactor()  
 elif token.type == "IDE":  
 identifier = token.value  
 token = self.tokens.selectNext()  
 if token.type == "OPEN\_PAR":  
 token = self.tokens.selectNext()  
 args = []  
 while True:  
 if token.type == "CLOSE\_PAR":  
 break  
 else:  
 arg = self.parseExpression()  
 args.append(arg)  
 token = self.tokens.actual  
 if token.type == "COMMA":  
 self.tokens.selectNext()  
 pass  
 elif token.type == "CLOSE\_PAR":  
 break  
 else:  
 raise ValueError("Invalid token, expecting a , or ) on \  
 position {}".format(self.tokens.position))  
 none\_value = IntVal(None, [])  
 args.append(none\_value)  
 result = FuncCall(identifier, args)  
 self.tokens.selectNext()  
 else:  
 result = Identifier(identifier, [])  
 else:  
 raise ValueError("Invalid token, expecting number or opening parentesis on \  
 position {}".format(self.tokens.position))  
 return result

**Генератор кода**

def generateAsm(self, SymbolTable, whileFlag):  
 pass

def Evaluate(self, SymbolTable, whileFlag=0, nodeId=None):  
 value1\_obj = self.children[0].Evaluate(SymbolTable, whileFlag)  
 self.generateAsm(SymbolTable, "push", whileFlag)  
 value2\_obj = self.children[1].Evaluate(SymbolTable, whileFlag)  
 self.generateAsm(SymbolTable, "pop", whileFlag)  
 value1 = value1\_obj.getValue()  
 value2 = value2\_obj.getValue()  
 if (self.value == "+"):  
 if not self.same\_type(value1\_obj, value2\_obj):  
 raise ValueError("Operands must be the same type")  
 value\_sum = value1 + value2  
 self.generateAsm(SymbolTable, "ADD", whileFlag)  
 result = Value("int")  
 result.setValue(value\_sum)  
 return result  
 elif (self.value == "-"):  
 if not self.same\_type(value1\_obj, value2\_obj):  
 raise ValueError("Operands must be the same type")  
 value\_sub = value1 - value2  
 self.generateAsm(SymbolTable, "SUB", whileFlag)  
 result = Value("int")  
 result.setValue(value\_sub)  
 return result  
 elif (self.value == "or"):  
 if not self.same\_type(value1\_obj, value2\_obj):  
 raise ValueError("Operands must be the same type")  
 value\_or = value1 or value2  
 self.generateAsm(SymbolTable, "OR", whileFlag)  
 result = Value("boolean")  
 result.setValue(value\_or)  
 return result  
 elif (self.value == "\*"):  
 if not self.same\_type(value1\_obj, value2\_obj):  
 raise ValueError("Operands must be the same type")  
 value\_mult = value1 \* value2  
 self.generateAsm(SymbolTable, "IMUL", whileFlag)  
 result = Value("int")  
 result.setValue(value\_mult)  
 return result  
 elif (self.value == "/"):  
 if not self.same\_type(value1\_obj, value2\_obj):  
 raise ValueError("Operands must be the same type")  
 value\_div = value1 // value2  
 result = Value("int")  
 self.generateAsm(SymbolTable, "DIV", whileFlag)  
 result.setValue(value\_div)  
 return result  
 elif (self.value == "and"):  
 if not self.same\_type(value1\_obj, value2\_obj):  
 raise ValueError("Operands must be the same type")  
 value\_and = value1 and value2  
 self.generateAsm(SymbolTable, "AND", whileFlag)  
 result = Value("boolean")  
 result.setValue(value\_and)  
 return result  
 elif (self.value == ">"):  
 if not self.same\_type(value1\_obj, value2\_obj):  
 raise ValueError("Operands must be the same type")  
 self.generateAsm(SymbolTable, "jg", whileFlag, nodeId)  
 value\_bigger = value1 > value2  
 result = Value("boolean")  
 result.setValue(value\_bigger)  
 return result  
 elif (self.value == "<"):  
 if not self.same\_type(value1\_obj, value2\_obj):  
 raise ValueError("Operands must be the same type")  
 self.generateAsm(SymbolTable, "jl", whileFlag, nodeId)  
 value\_smaller = value1 < value2  
 result = Value("boolean")  
 result.setValue(value\_smaller)  
 return result  
 elif (self.value == "="):  
 if not self.same\_type(value1\_obj, value2\_obj):  
 raise ValueError("Operands must be the same type")  
 self.generateAsm(SymbolTable, "je", whileFlag, nodeId)  
 value\_equal = value1 == value2  
 result = Value("boolean")  
 result.setValue(value\_equal)  
 return result  
 elif (self.value == "!="):  
 if not self.same\_type(value1\_obj, value2\_obj):  
 raise ValueError("Operands must be the same type")  
 value\_diff = value1 != value2  
 result = Value("boolean")  
 result.setValue(value\_diff)  
 return result  
 else:  
 return  
  
 def generateAsm(self, SymbolTable, op, whileFlag, nodeId=None):  
 if not whileFlag:  
 asm = ""  
 if op == "push":  
 asm += "PUSH EBX \n"  
 elif op == "pop":  
 asm += "POP EAX \n"  
 elif op == "jl":  
 asm += "CMP EAX, EBX \n"  
 asm += "CALL binop\_jl \n"  
 asm += "CMP EBX, False \n"  
 asm += "JE EXIT\_{0} \n".format(nodeId)  
 elif op == "jg":  
 asm += "CMP EAX, EBX \n"  
 asm += "CALL binop\_jg \n"  
 asm += "CMP EBX, False \n"  
 asm += "JE EXIT\_{0} \n".format(nodeId)  
 elif op == "je":  
 asm += "CMP EAX, EBX \n"  
 asm += "CALL binop\_je \n"  
 asm += "CMP EBX, False \n"  
 asm += "JE EXIT\_{0} \n".format(nodeId)  
 elif op == "IMUL":  
 asm += "IMUL EBX \n"  
 asm += "MOV EBX, EAX \n"  
 else:  
 asm += "{0} EBX, EAX \n".format(op)  
 AssemblyCode.assembly\_code += asm  
  
  
class Assignment(Node):  
 def Evaluate(self, SymbolTable, whileFlag=0):  
 name = self.children[0].Evaluate(SymbolTable, whileFlag).getValue()  
 value = self.children[1].Evaluate(SymbolTable, whileFlag).getValue()  
 self.generateAsm(SymbolTable, whileFlag)  
 SymbolTable.setSymbol(name, value)  
  
 def generateAsm(self, SymbolTable, whileFlag):  
 if not whileFlag:  
 value1 = self.children[0].Evaluate(  
 SymbolTable, whileFlag).getValue()  
 value2 = SymbolTable.id  
 asm = "MOV [{0}\_{1}], EBX \n".format(value1, value2)  
 AssemblyCode.assembly\_code += asm  
  
  
class UnOp(Node):  
 def Evaluate(self, SymbolTable, whileFlag=0):  
 value\_obj = self.children[0].Evaluate(SymbolTable, whileFlag)  
 value = value\_obj.getValue()  
 if (self.value == "-"):  
 result = Value("int")  
 result.setValue(value \* -1)  
 return result  
 elif (self.value == "not"):  
 if value\_obj.type == "boolean":  
 result = Value("boolean")  
 result.setValue(not value)  
 return result  
 else:  
 raise ValueError("Operand must be a boolean")  
 else:  
 return  
  
  
class StrVal(Node):  
 def Evaluate(self, SymbolTable, whileFlag=0):  
 value = Value("string")  
 value.setValue(self.value)  
 return value  
  
  
class IntVal(Node):  
 def Evaluate(self, SymbolTable, whileFlag=0):  
 self.generateAsm(SymbolTable, whileFlag)  
 value = Value("int")  
 value.setValue(self.value)  
 return value  
  
 def generateAsm(self, SymbolTable, whileFlag):  
 if not whileFlag:  
 asm = "MOV EBX, {0} \n".format(self.value)  
 AssemblyCode.assembly\_code += asm  
  
  
class BoolVal(Node):  
 def Evaluate(self, SymbolTable, whileFlag=0):  
 self.generateAsm(SymbolTable, whileFlag)  
 value = Value("boolean")  
 value.setValue(self.value)  
 return value  
  
 def generateAsm(self, SymbolTable, whileFlag):  
 if not whileFlag:  
 asm = "MOV EBX, ${0} \n".format(self.value)  
 AssemblyCode.assembly\_code += asm  
  
  
class Identifier(Node):  
 def Evaluate(self, SymbolTable, whileFlag=0):  
 self.generateAsm(SymbolTable, whileFlag)  
 value = SymbolTable.getSymbol(self.value)  
 return value  
  
 def generateAsm(self, SymbolTable, whileFlag):  
 if not whileFlag:  
 asm = "MOV EBX, [{0}\_{1}] \n".format(self.value, SymbolTable.id)  
 AssemblyCode.assembly\_code += asm  
  
  
class NoOp(Node):  
 def Evaluate(self, SymbolTable, whileFlag=0):  
 return None  
  
  
class Statements(Node):  
 def Evaluate(self, SymbolTable, whileFlag=0):  
 for child in self.children:  
 child.Evaluate(SymbolTable, whileFlag)  
  
  
class Print(Node):  
 def Evaluate(self, SymbolTable, whileFlag=0):  
 value = self.children[0].Evaluate(SymbolTable, whileFlag)  
 self.generateAsm(SymbolTable, whileFlag)  
 print(value.getValue())  
  
 def generateAsm(self, SymbolTable, whileFlag):  
 if not whileFlag:  
 asm = "PUSH EBX \n"  
 asm += "CALL print \n"  
 AssemblyCode.assembly\_code += asm  
  
  
class Read(Node):  
 def Evaluate(self, SymbolTable, whileFlag=0):  
 result = input()  
 value = Value("int")  
 value.setValue(int(result))  
 return value  
  
  
class If(Node):  
 def Evaluate(self, SymbolTable, whileFlag=0):  
 comp = self.children[0].Evaluate(SymbolTable, whileFlag, self.id)  
 if (comp.value):  
 self.children[1].Evaluate(SymbolTable, whileFlag)  
 else:  
 self.children[2].Evaluate(SymbolTable, whileFlag)  
  
 def generateAsm(self, SymbolTable):  
 pass  
  
  
class While(Node):  
 def Evaluate(self, SymbolTable, whileFlag=0):  
 comp = self.children[0]  
 self.generateAsm(SymbolTable, "LOOP", whileFlag, "declare")  
 flag = 0  
 while (comp.Evaluate(SymbolTable, flag, self.id).getValue()):  
 self.children[1].Evaluate(SymbolTable, flag)  
 flag = 1  
 self.generateAsm(SymbolTable, "JMP LOOP", whileFlag, "jump")  
 self.generateAsm(SymbolTable, "EXIT", whileFlag, "declare")  
  
 def generateAsm(self, SymbolTable, label, whileFlag, op):  
 if not whileFlag:  
 if op == "declare":  
 asm = "{0}\_{1}: \n".format(label, self.id)  
 AssemblyCode.assembly\_code += asm  
 else:  
 asm = "{0}\_{1} \n".format(label, self.id)  
 AssemblyCode.assembly\_code += asm  
  
  
class Program(Node):  
 def Evaluate(self, SymbolTable, whileFlag=0):  
 self.generateAssemblyConstants()  
 SymbolTable.createSymbol(self.value, None)  
 for i in range(len(self.children)):  
 if i == 2: # Statements  
 self.generateAsm(SymbolTable, whileFlag)  
 self.children[i].Evaluate(SymbolTable, whileFlag)  
 self.generateEndInterruption()

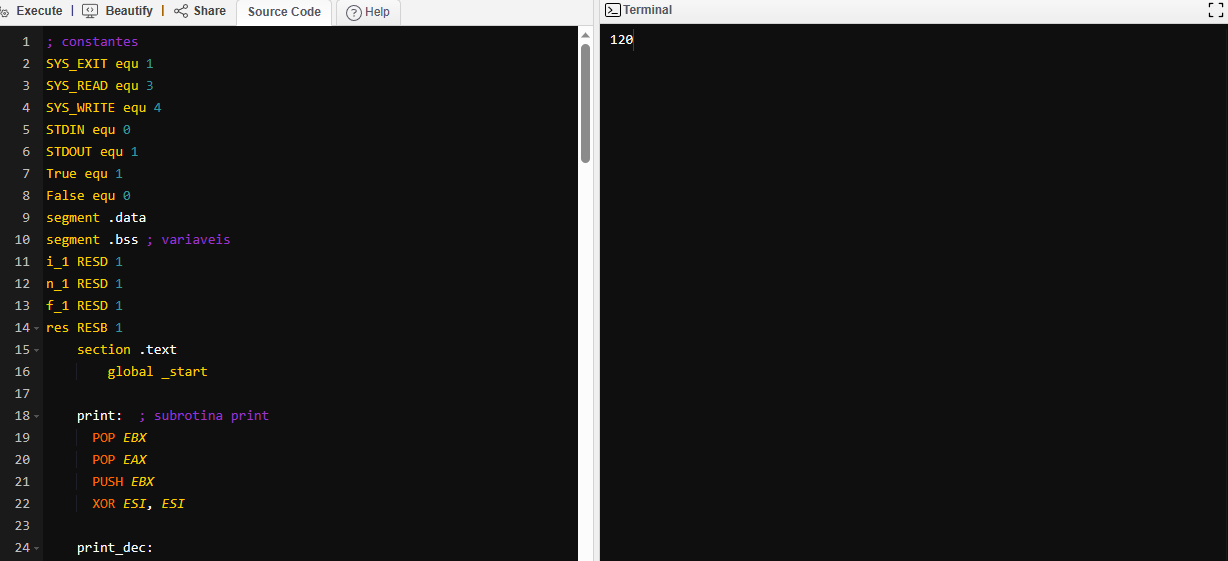
**Тесты**

**Тестовый файл 1**

program teste;  
 var i, n, f: int;  
 begin  
 n := 5;  
 i := 2;  
 f := 1;  
 while i < n + 1 do  
 begin  
 f := f \* i;  
 i := i + 1  
 end;  
 print(f)  
 end.

**Получаемый код**

; constantes   
SYS\_EXIT equ 1   
SYS\_READ equ 3   
SYS\_WRITE equ 4   
STDIN equ 0   
STDOUT equ 1   
True equ 1   
False equ 0   
segment .data   
segment .bss ; variaveis   
i\_1 RESD 1   
n\_1 RESD 1   
f\_1 RESD 1   
res RESB 1  
 section .text  
 global \_start  
  
 print: ; subrotina print  
 POP EBX  
 POP EAX  
 PUSH EBX  
 XOR ESI, ESI  
  
 print\_dec:  
 MOV EDX, 0  
 MOV EBX, 0x000A  
 DIV EBX  
 ADD EDX, '0'  
 PUSH EDX  
 INC ESI  
 CMP EAX, 0  
 JZ print\_next  
 JMP print\_dec  
  
 print\_next:  
 CMP ESI, 0  
 JZ print\_exit  
 DEC ESI  
  
 MOV EAX, SYS\_WRITE  
 MOV EBX, STDOUT  
  
 POP ECX  
 MOV [res], ECX  
 MOV ECX, res  
  
 MOV EDX, 1  
 INT 0x80  
 JMP print\_next  
  
 print\_exit:  
 RET  
  
 ; subrotinas if/while  
 binop\_je:  
 JE binop\_true  
 JMP binop\_false  
  
 binop\_jg:  
 JG binop\_true  
 JMP binop\_false  
  
 binop\_jl:  
 JL binop\_true  
 JMP binop\_false  
  
 binop\_false:  
 MOV EBX, False  
 JMP binop\_exit  
 binop\_true:  
 MOV EBX, True  
 binop\_exit:  
 RET  
  
\_start:   
MOV EBX, 5   
MOV [n\_1], EBX   
MOV EBX, 2   
MOV [i\_1], EBX   
MOV EBX, 1   
MOV [f\_1], EBX   
LOOP\_39:   
MOV EBX, [i\_1]   
PUSH EBX   
MOV EBX, [n\_1]   
PUSH EBX   
MOV EBX, 1   
POP EAX   
ADD EBX, EAX   
POP EAX   
CMP EAX, EBX   
CALL binop\_jl   
CMP EBX, False   
JE EXIT\_39   
MOV EBX, [f\_1]   
PUSH EBX   
MOV EBX, [i\_1]   
POP EAX   
IMUL EBX   
MOV EBX, EAX   
MOV [f\_1], EBX   
MOV EBX, [i\_1]   
PUSH EBX   
MOV EBX, 1   
POP EAX   
ADD EBX, EAX   
MOV [i\_1], EBX   
JMP LOOP\_39   
EXIT\_39:   
MOV EBX, [f\_1]   
PUSH EBX   
CALL print   
MOV EAX, 1   
INT 0x80

**Скомпилированный уже на языке assembler код**

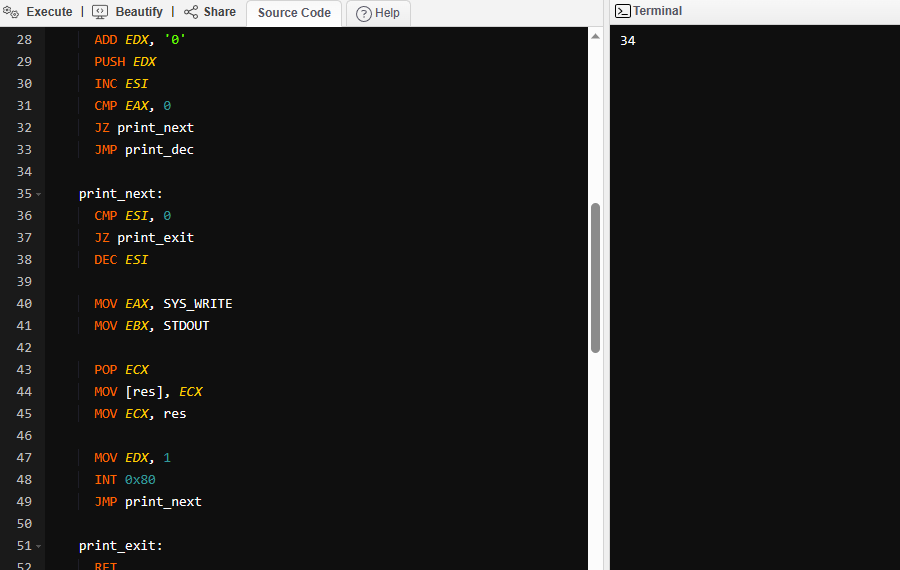
**Тестовый файл 2**

program teste;  
 var a, b: int;  
 x: boolean;  
  
 function fib(n: int): int;  
 begin  
 if n = 0 then  
 fib := 0  
 else  
 if n = 1 then  
 fib := 1  
 else  
 fib := fib(n - 1) + fib(n - 2)  
 end  
  
 begin  
 print(fib(8))  
 end.

**Получаемый код**

; constantes   
SYS\_EXIT equ 1   
SYS\_READ equ 3   
SYS\_WRITE equ 4   
STDIN equ 0   
STDOUT equ 1   
True equ 1   
False equ 0   
segment .data   
segment .bss ; variaveis   
i\_1 RESD 1   
n\_1 RESD 1   
f\_1 RESD 1   
res RESB 1  
 section .text  
 global \_start  
  
 print: ; subrotina print  
 POP EBX  
 POP EAX  
 PUSH EBX  
 XOR ESI, ESI  
  
 print\_dec:  
 MOV EDX, 0  
 MOV EBX, 0x000A  
 DIV EBX  
 ADD EDX, '0'  
 PUSH EDX  
 INC ESI  
 CMP EAX, 0  
 JZ print\_next  
 JMP print\_dec  
  
 print\_next:  
 CMP ESI, 0  
 JZ print\_exit  
 DEC ESI  
  
 MOV EAX, SYS\_WRITE  
 MOV EBX, STDOUT  
  
 POP ECX  
 MOV [res], ECX  
 MOV ECX, res  
  
 MOV EDX, 1  
 INT 0x80  
 JMP print\_next  
  
 print\_exit:  
 RET  
  
 ; subrotinas if/while  
 binop\_je:  
 JE binop\_true  
 JMP binop\_false  
  
 binop\_jg:  
 JG binop\_true  
 JMP binop\_false  
  
 binop\_jl:  
 JL binop\_true  
 JMP binop\_false  
  
 binop\_false:  
 MOV EBX, False  
 JMP binop\_exit  
 binop\_true:  
 MOV EBX, True  
 binop\_exit:  
 RET  
  
\_start:   
MOV EBX, 30   
MOV [n\_1], EBX   
MOV EBX, 4   
MOV [i\_1], EBX   
MOV EBX, [n\_1]   
PUSH EBX   
MOV EBX, [i\_1]   
POP EAX   
ADD EBX, EAX   
MOV [f\_1], EBX   
MOV EBX, [f\_1]   
PUSH EBX   
CALL print   
MOV EAX, 1   
INT 0x80

**Скомпилированный уже на языке assembler код**



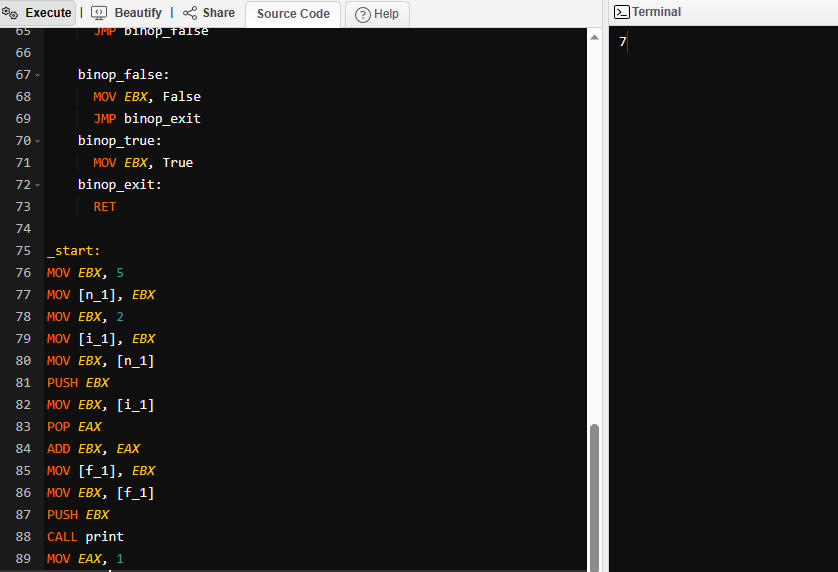
**Тестовый файл 3**

program teste;  
 var i, n, f: int;  
 begin  
 n := 5;  
 i := 2;  
 f := n + i;  
  
 print(f)  
 end.

**Получаемый код**

; constantes   
SYS\_EXIT equ 1   
SYS\_READ equ 3   
SYS\_WRITE equ 4   
STDIN equ 0   
STDOUT equ 1   
True equ 1   
False equ 0   
segment .data   
segment .bss ; variaveis   
i\_1 RESD 1   
n\_1 RESD 1   
f\_1 RESD 1   
res RESB 1  
 section .text  
 global \_start  
  
 print: ; subrotina print  
 POP EBX  
 POP EAX  
 PUSH EBX  
 XOR ESI, ESI  
  
 print\_dec:  
 MOV EDX, 0  
 MOV EBX, 0x000A  
 DIV EBX  
 ADD EDX, '0'  
 PUSH EDX  
 INC ESI  
 CMP EAX, 0  
 JZ print\_next  
 JMP print\_dec  
  
 print\_next:  
 CMP ESI, 0  
 JZ print\_exit  
 DEC ESI  
  
 MOV EAX, SYS\_WRITE  
 MOV EBX, STDOUT  
  
 POP ECX  
 MOV [res], ECX  
 MOV ECX, res  
  
 MOV EDX, 1  
 INT 0x80  
 JMP print\_next  
  
 print\_exit:  
 RET  
  
 ; subrotinas if/while  
 binop\_je:  
 JE binop\_true  
 JMP binop\_false  
  
 binop\_jg:  
 JG binop\_true  
 JMP binop\_false  
  
 binop\_jl:  
 JL binop\_true  
 JMP binop\_false  
  
 binop\_false:  
 MOV EBX, False  
 JMP binop\_exit  
 binop\_true:  
 MOV EBX, True  
 binop\_exit:  
 RET  
  
\_start:   
MOV EBX, 5   
MOV [n\_1], EBX   
MOV EBX, 2   
MOV [i\_1], EBX   
MOV EBX, [n\_1]   
PUSH EBX   
MOV EBX, [i\_1]   
POP EAX   
ADD EBX, EAX   
MOV [f\_1], EBX   
MOV EBX, [f\_1]   
PUSH EBX   
CALL print   
MOV EAX, 1   
INT 0x80

**Скомпилированный уже на языке assembler код**



**Заключение**

Сборка программы имеется на ресурсе GitHub: <https://github.com/SheronovIvan/Compiler.git>

Вывод:

Создание компилятора - это сложный и трудоемкий процесс, требующий глубоких знаний в области теории языков программирования, алгоритмов и структур данных. Однако, успешное создание компилятора может принести огромную пользу, так как позволяет автоматизировать процесс преобразования исходного кода программы в машинный код, что существенно ускоряет процесс разработки программного обеспечения.

В ходе курсовой работы мы изучили основные принципы работы компиляторов, а также рассмотрели различные подходы к их реализации. Мы рассмотрели структуру компилятора и проанализировали каждый из его компонентов, таких как лексический анализатор, синтаксический анализатор, генератор кода.

Также мы рассмотрели различные методы генерации кода, такие как генерация кода виртуальной машины, генерация кода на языке ассемблера и генерация кода на машинном языке.

Таким образом, создание компилятора - это важный этап в разработке программного обеспечения, который позволяет автоматизировать процесс преобразования исходного кода в машинный код. Реализация компилятора требует глубоких знаний в области теории языков программирования, алгоритмов и структур данных, но может принести огромную пользу в долгосрочной перспективе.