Министерство науки и высшего образования 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 х86

Пояснительная записка

RU. **643.02068048.00001**

АННОТАЦИЯ

В данном документе приведена реализация компилятор подмножества процедурного языка.

Требования к входному языку:

1. Должны присутствовать операторные скобки.
2. Должна игнорироваться идентация 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: выходной язык ассемблер x86.

**Оглавление**

[1. Общие сведения 4](#_Toc41860520)

[2. Лексический анализатор 5](#_Toc41860521)

[3. Синтаксический анализатор 7](#_Toc41860522)

[4. Транслятор 10](#_Toc41860523)

[5. Оптимизация 13](#_Toc41860524)

[6. Тестирование 16](#_Toc41860525)

[7. Приложение 1 18](#_Toc41860526)

1. Общие сведенИя

Курсовая работа была разработана на языке программирования «Python». Также были использованы библиотеки такие, как «rply» и «llvmlite». Программа состоит из лексического анализатора, синтаксического анализатора и кодогенератора.

Лексический анализатор получает входной файл, разбирает на последовательности символов, называемых токенами. Далее поток токенов направляется парсеру.

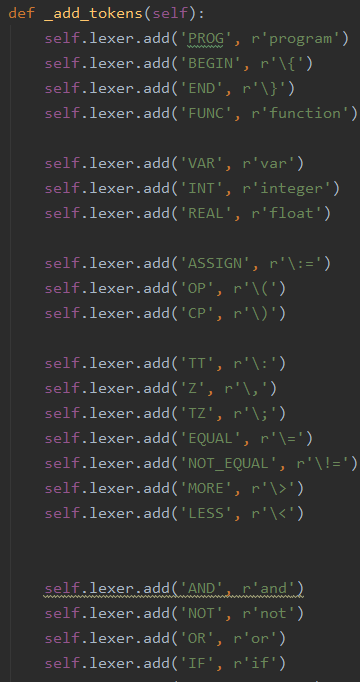
Синтаксический анализатор сопоставляет поток токенов с грамматикой. Абстрактное синтаксическое дерево, в свою очередь, используется для [промежуточного представления](https://ru.wikipedia.org/w/index.php?title=%D0%9F%D1%80%D0%BE%D0%BC%D0%B5%D0%B6%D1%83%D1%82%D0%BE%D1%87%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%B5%D0%B4%D1%81%D1%82%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5&action=edit&redlink=1)программы, которая затем используется в качестве внутреннего представления в [компиляторе](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%B8%D0%BB%D1%8F%D1%82%D0%BE%D1%80)  для оптимизации и генерации кода.

Кодогенератор конвертирует [синтаксически](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D0%BD%D1%82%D0%B0%D0%BA%D1%81%D0%B8%D1%81_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) корректную программу в последовательность инструкций, которые могут выполняться на машине. На вход генератора кода подаётся  [абстрактное синтаксическое дерево](https://ru.wikipedia.org/wiki/%D0%90%D0%B1%D1%81%D1%82%D1%80%D0%B0%D0%BA%D1%82%D0%BD%D0%BE%D0%B5_%D1%81%D0%B8%D0%BD%D1%82%D0%B0%D0%BA%D1%81%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%BE%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE). Дерево преобразуется в линейную последовательность инструкций промежуточного языка.

1. Лексический анализатор

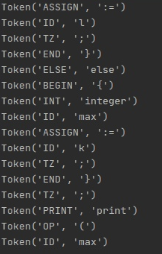
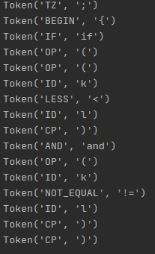
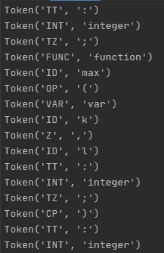
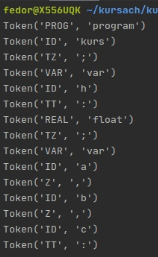
Входной текстовый файл разбирается на токены в соответствии с правилами, описанными лексическим анализатором. Результатом выполнения лексера является перечень всех найденных в исходном файле лексем. Во входном файле присутствуют логические и условный операторы, присваивание, знаки арифметики, цикл, переменные типа integer и float, а также функция. В том числе во входном файле присутствуют комментарии, которые по правилам лексического анализатора, игнорируются при разборе.

На «Рис.1» изображен пример правил обработки лексем из входного текстового файла. На рисунке изображена только часть лексера, полный код программы приведен в части «Приложение 1».



«Рис.1»

Результат работы лексического анализатора, то есть выходной поток токенов изображен на «Рис.2». Далее готовый поток лексем передаётся на работу синтаксическому анализатору.



«Рис.2»

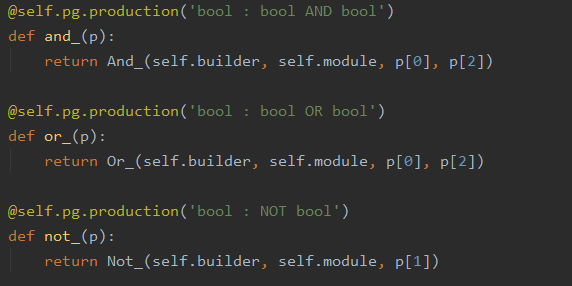
1. Синтаксический анализатор

К синтаксическому анализатору поступает поток токенов. Парсер начинает сопоставлять их с правилами грамматики.

Для каждого токена описаны набор правил, по которому синтаксический анализатор их обрабатывает. После, преобразованный результат отправляется транслятору для дальнейших преобразований.

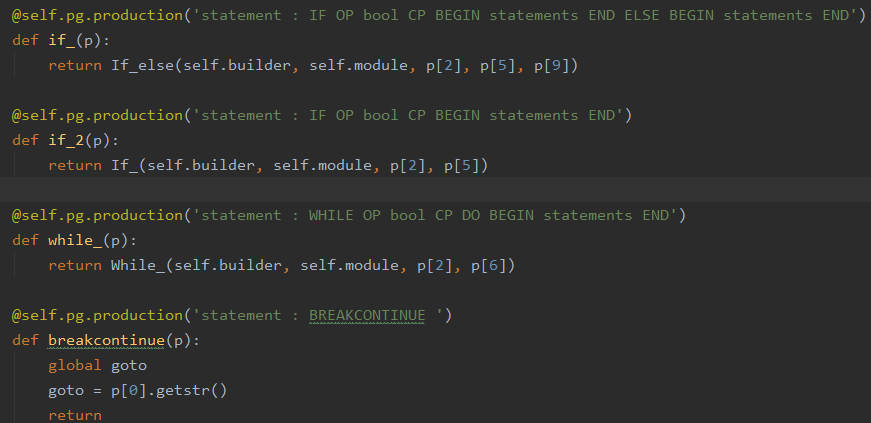
На рисунках ниже будут представлены некоторые правила обработки.

На «Рис.3» изображены правила для логических операторов



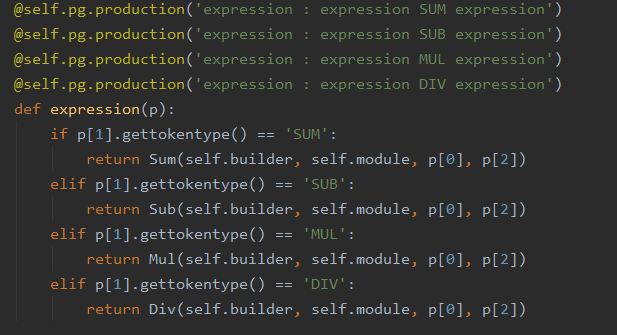
«Рис.3»

На «Рис.4» представлены правила операторов цикла

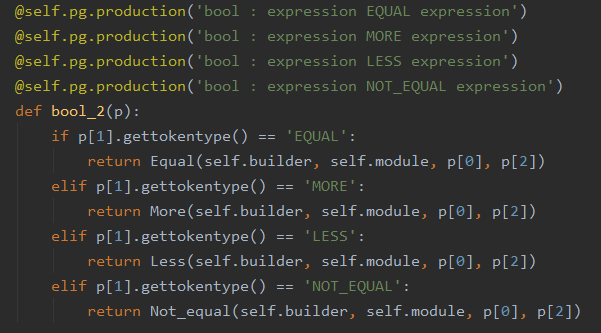


«Рис.4»

На «Рис.5» и «Рис.6» изображены правила для обработки знаков арифметики

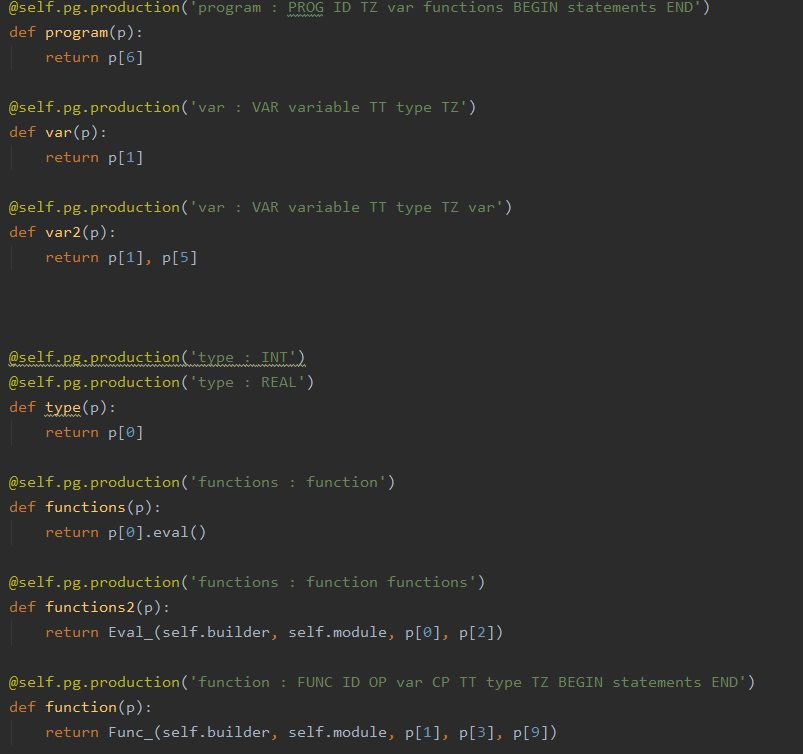


«Рис.5»



«Рис.6»

Также на «Рис.7» изображены правила для обработки начала программы, конца, функций и типов переменных.



«Рис.7»

На рисунках изображена только часть парсера, реализующая основные моменты, полный код программы приведен в части «Приложение 1».

После определения грамматики начинает свою работу транслятор.

1. Транслятор

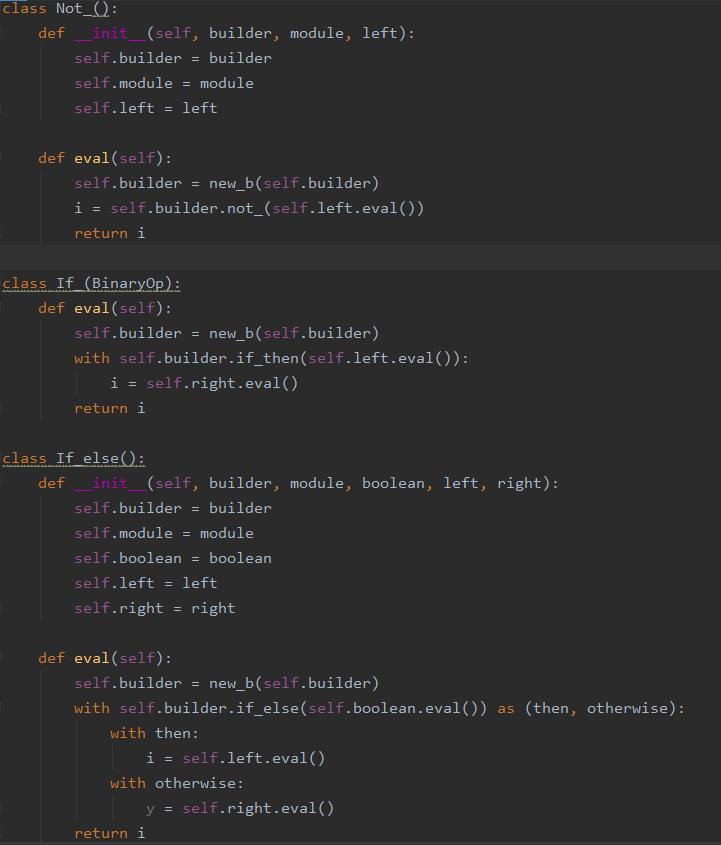
Транслятор выполняет свою часть работы при помощи библиотеки «llvmlite». Выходные данные из синтаксического анализатора передаются в транслятор, где происходят следующие преобразования. Благодаря данной библиотеки он создает промежуточный код «LLVM IR». Результатом выполнения транслятора является файл out.ll, куда записывается сам промежуточный код.

Последним этапом является преобразование кода из файла out.ll в объектный файл с помощью статического компилятора «LLC».

С помощью команд библиотеки “llvmlite” транслятор создает промежуточный код “LLVM IR” для его дальнейшего преобразования в assembler.

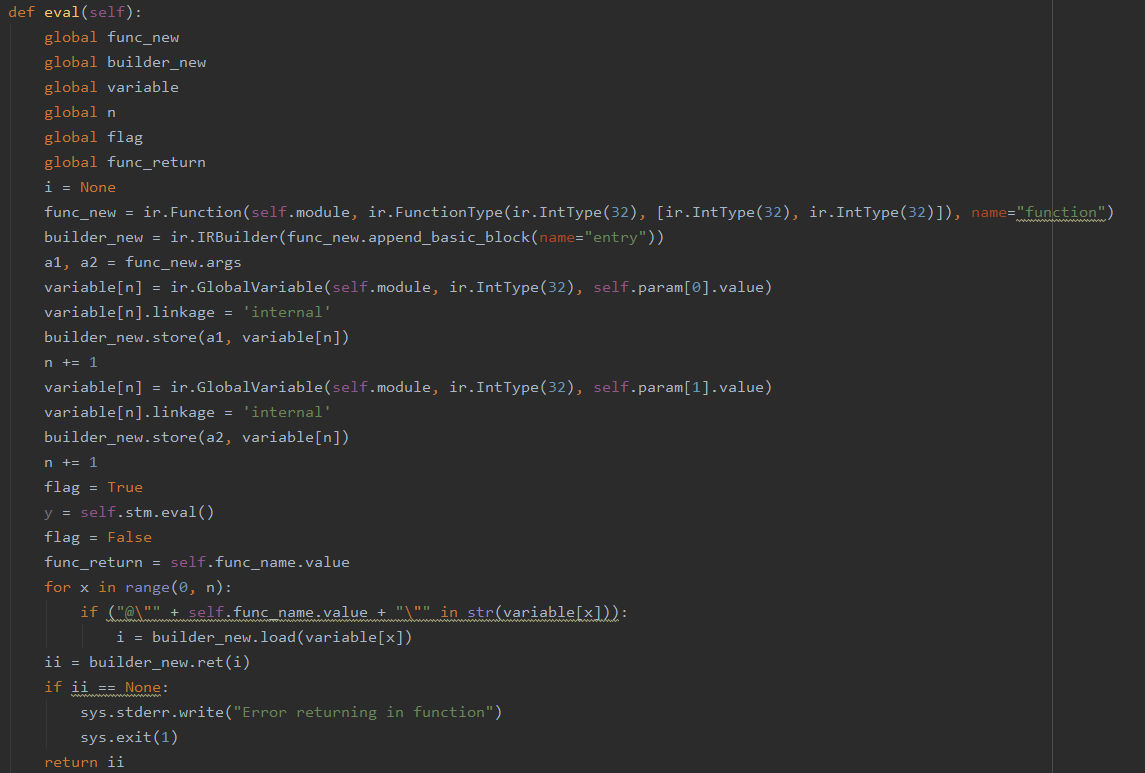
Ниже будут приведены некоторые фрагменты кода, реализующие основные моменты, полный листинг кода приведен в «Приложение 1».

На «Рис.8» изображено преобразование операторов if, else, not



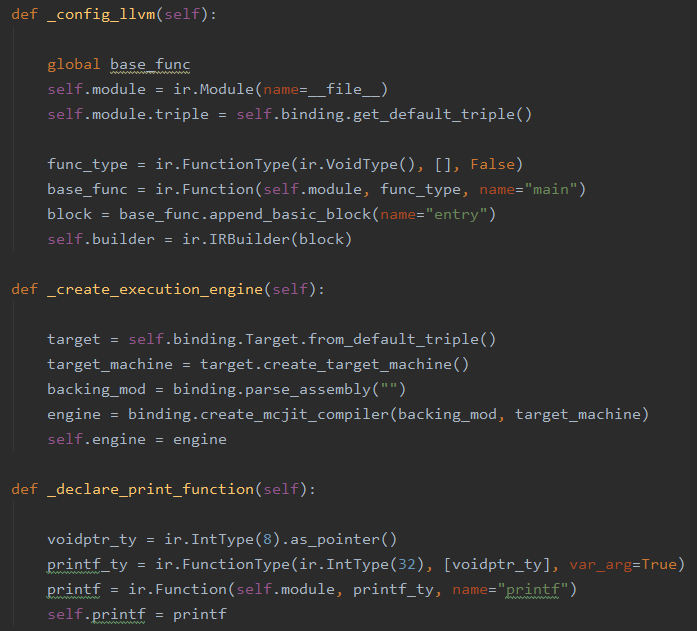
«Рис.8»

На «Рис.9» представлено преобразование функций



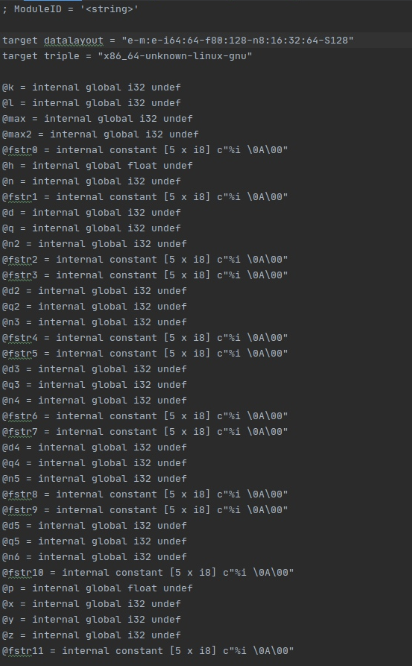
«Рис.9»

И генерация кода изображена на «Рис.10»

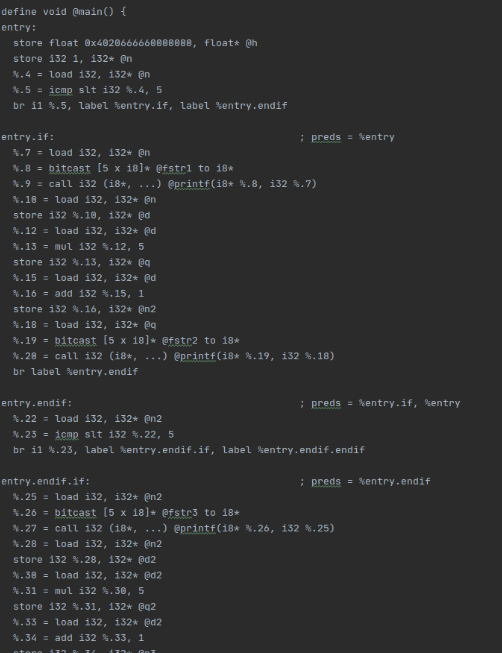


«Рис.10»

Промежуточный код получается очень объёмным, некоторая часть изображена на «Рис.11» и «Рис.12», поэтому нужна оптимизация



«Рис.11»



«Рис.12»

1. Оптимизация

Оптимизация программы является очень важным этапом. С помощью оптимизации программы модифицируется промежуточный код.

Оптимизация была реализована с помощью инструментов библиотеки «llvmlite» и включает в себя глобальную нумерацию значений, избыточное устранение нагрузки, объединение лишних инструкций, устранение мертвых аргументов и мертвого кода в том числе.

Для объединения дублирующиихся глобальных констант в одну общую константу используется метод pm.add\_constant\_merge\_pass(). Это полезно, потому что иногда бывает много строковых констант в программе, независимо от того, доступна или нет существующая строка.

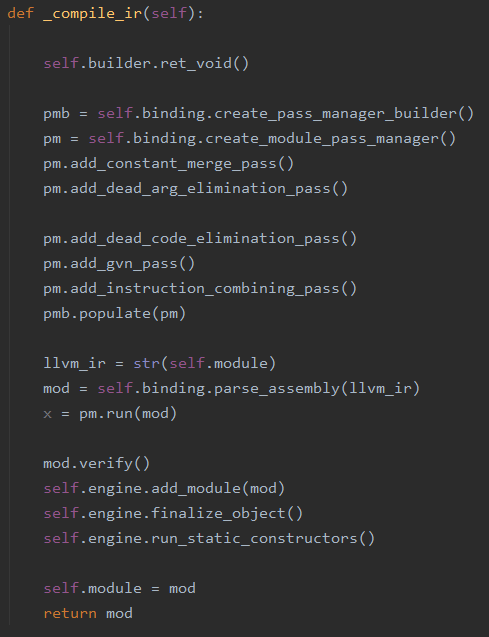
Для удаляения мертвых аргументов из внутренних функций используется метод pm.add\_dead\_arg\_elimination\_pass(). Устранение мертвых аргументов удаляет непосредственно мертвые аргументы, а также аргументы, передаваемые только в вызовы функций, как мертвые аргументы других функций.

Для устранения мертвого кода используется метод pm.add\_dead\_code\_elimination\_pass().

Для глобальной нумерации значений, чтобы исключить полностью и частично избыточные инструкции используется метод pm.add\_gvn\_pass(). Он также выполняет избыточное устранение нагрузки.

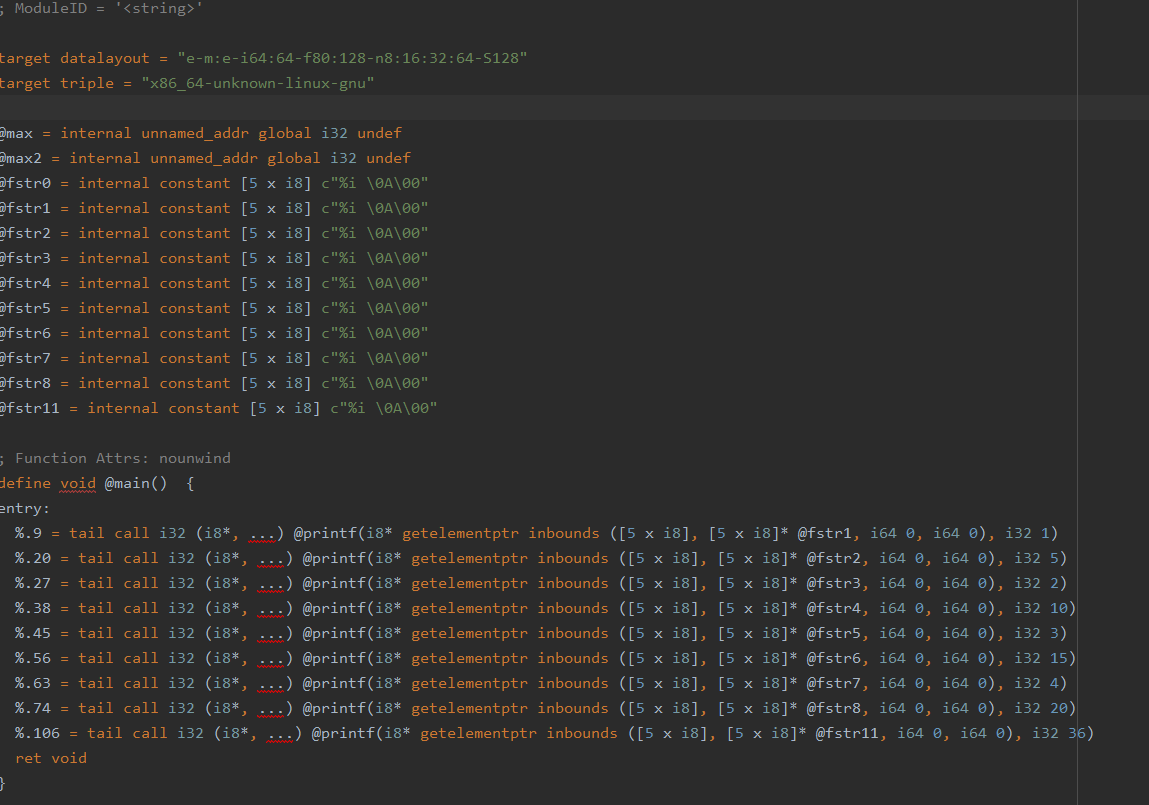
Для объединения инструкций, чтобы сформировать меньше простых инструкций используется метод pm.add\_instruction\_combining\_pass(). Этот проход не изменяет CFG. На этом проходе происходит алгебраическое упрощение.

На «Рис.13» изображен код оптимизации, на котором представлены все методы вместе.



«Рис.13»

После проведения оптимизации промежуточный код изменяется, пример модифицированного промежуточного кода изображен на «Рис.14» и «Рис.15»



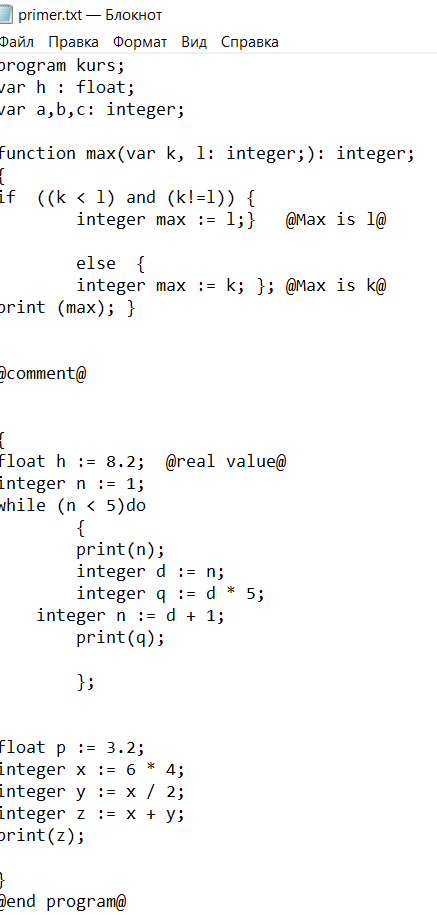
«Рис.14»



«Рис.15»

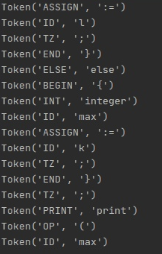
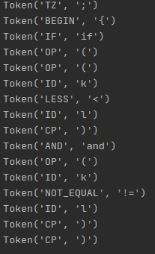
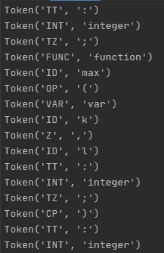
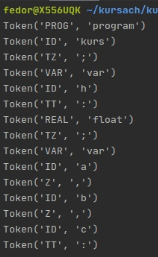
1. Тестирование

На вход поступает текстовый файл primer.txt, содержимое изображено на «Рис.16»



«Рис.16»

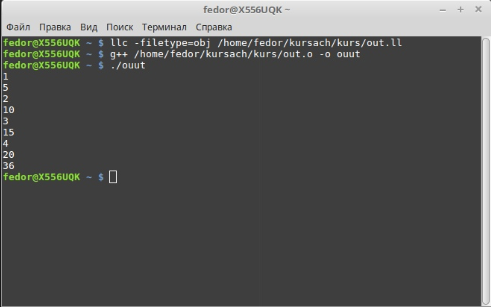
Входной текстовый файл обрабатывается лексером и выдаёт поток токенов как изображено на «рис.17»



«Рис.17»

Далее для выполнения программы был установлен модуль «g++».

Программа выполняется с помощью трех команд как изображено на «Рис.18»



«Рис.18»

При ручном подсчёте значений, которые должна вывести программа, были получены такие же значения, как и при запуске программы, значит, что программа была выполнена корректно.

1. Приложение 1

Ссылка на исходный код: <https://github.com/fedorrrrr/kursach>

**Исходный код:**

**MAIN.py**

**from** Codegener **import** CodeGen  
**from** Lexer **import** Lexer  
**from** Parser **import** Parser  
  
**with** open(**"/home/fedor/kursach/kurs/primer.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(**"/home/fedor/kursach/kurs/out.ll"**)

**Lexer.py**

**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()

**Parser.py**

**from** rply **import** ParserGenerator  
**from** AST **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

**AST.py:**

**from** llvmlite **import** ir  
**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

**Codegener.py**

**from** llvmlite **import** ir, binding  
  
  
**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)