**ОГЛАВЛЕНИЕ**

[**ВВЕДЕНИЕ** 3](#_Toc41737280)

[**ПРОЕКТИРОВАНИЕ КОМПИЛЯТОРА** 4](#_Toc41737281)

[**Грамматика** 4](#_Toc41737282)

[**Лексический анализатор** 5](#_Toc41737283)

[**Парсер** 6](#_Toc41737284)

[**Машинный код** 9](#_Toc41737285)

[**Компилятор** 11](#_Toc41737286)

[**Тест** 12](#_Toc41737287)

**Приложение**…………………………………………………………………..…13

Изм.

Лист

№ докум.

Подпись

Дата

Лист

2

RU. 02068048.502900.01.01

Разраб.

Фомин А.Е.

Провер.

Монахов Ю.М.

Реценз.

Н. Контр.

Монахов Ю.М.

Утверд.

Компилятор

Лит.

Листов

18

ВлГУ

# ВВЕДЕНИЕ

Задачи курсовой работы заключаются:

Описание программы, предназначенной для компиляции программ на некотором исходном языке программирования под целевую платформу. Исходным языком, предназначенным для компиляции является выбранное мной подмножество языка C++. Язык, выбранный для разработки Pyton. Целевая платформа – PyCharm.

Требования к приложению:

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.

Интерфейс приложения:

* На вход поступает код программы на языке C++
* В процессе работы компилятора выполняется этот код на целевой платформе
* На выход получаем этот файл, очищенный от комментариев и пустых блоков

# ПРОЕКТИРОВАНИЕ КОМПИЛЯТОРА

**Грамматика**  
Что будет уметь наш язык:  
— единственный тип данных — int  
— все переменные — глобальные. Всего есть 28 переменных (a-z)— единственный оператор сравнения — это "<"  
— из конструкций языка — условные операторы if, if/else, while, do/while  
Вот такая получается грамматика:

<program> ::= <statement>

<statement> ::= "if" <paren-expr> <statement> |

"if" <paren-expr> <statement> "else" <statement> |

"while" <paren-expr> <statement> |

"do" <statement> "while" <paren-expr> |

"{" { <statement> } "}" |

<expr> ";" |

";"

<paren-expr> ::= "(" <expr> ")"

<expr> ::= <test> | <id> "=" <expr>

<test> ::= <sum> | <sum> "<" <sum>

<sum> ::= <term> | <sum> "+" <term> | <sum> "-" <term>| <sum> "\*" <term> | <sum> "/" <term>

<term> ::= <id> | <int> | <paren-expr>

<id> ::= "a" | "b" | ... | "z"

<int> ::= <digit>, { <digit> }

<digit> ::= "0" | "1" | ... | "9"   
Вот что эта запись приблизительно означает:  
Программа — это один оператор (statement).  
Операторы бывают условными (if..else...), циклическими (while...) и просто операторами (напр., «a=2+3»).  
Условные и циклические содержат в себе выражение-условие и тело (в виде оператора). Обычные операторы заканчиваются точкой с запятой. Можно группировать операторы в фигурных скобках, тогда получим составной оператор.  
Выражения — это либо сумма, либо присваивание переменной значения.  
Здесь сумма — это последовательность сложений-вычитаний термов.  
Терм — это число, переменная или выражение в скобках.  
Переменные — это символы от a до z. Числа — это набор цифр от 0 до 9.  
Все эти сложности нужны для того, чтобы указать компилятору как именно автоматически анализировать исходный код. Например, встретили слово «if» — значит дальше идет выражение в скобках, а за ним — оператор.

**Лексический анализатор**  
На вход компилятору поступает текстовый файл (исходник). [Лексический анализатор](http://en.wikipedia.org/wiki/Lexical_analysis) нужен для того, чтобы выделить в этом файле токены. Т.е. строчку «a = 3 + 2;» лексический анализатор должен представить в виде «идентификатор: a», «оператор =», «число 3», «оператор +», «число 2», «символ;». Лексический анализатор работает только с последовательностью букв, т.е. для него строчка «a b if =» тоже имеет смысл и является абсолютно корректной.  
Итак, наш лексический анализатор должен узнавать следующие токены:  
— числа  
— идентификаторы-переменные  
— ключевые слова: if, else, while, do  
— символы \*, /,+, -, <, =, {, }, (, ),;  
— конец файла  
Вот как выглядит его исходный код:

class Lexer:  
 NUM, ID, IF, ELSE, WHILE, DO, LBRA, RBRA, LPAR, RPAR, PLUS, MINUS, LESS, \  
 EQUAL, SEMICOLON, MULTIPLY, DIVIDE, EOF = range(18)  
  
 SYMBOLS = {'{': LBRA, '}': RBRA, '=': EQUAL, ';': SEMICOLON, '(': LPAR,  
 ')': RPAR, '+': PLUS, '-': MINUS, '<': LESS, '\*': MULTIPLY, '/': DIVIDE}  
 WORDS = {'if': IF, 'else': ELSE, 'do': DO, 'while': WHILE}  
  
 ch = ' ' # предположим, что первый символ-это пробел  
  
 def error(self, msg):  
 print('def error')  
 print('Lexer error: ', msg)  
 sys.exit(1)  
  
 def getc(self):  
 print('def getc')  
 self.ch = sys.stdin.read(1)  
 def next\_tok(self):  
 print('def next\_tok')  
 self.value = None  
 self.sym = None  
 while self.sym == None:  
 if (self.ch) == '@':  
 self.sym = Lexer.EOF  
 elif self.ch.isspace():  
 self.getc()  
 elif self.ch in Lexer.SYMBOLS:  
 self.sym = Lexer.SYMBOLS[self.ch]  
 self.getc()  
 elif self.ch.isdigit():  
 intval = 0  
 while self.ch.isdigit():  
 intval = intval \* 10 + int(self.ch)  
 self.getc()  
 self.value = intval  
 self.sym = Lexer.NUM  
 elif self.ch.isalpha():  
 ident = ''  
 while self.ch.isalpha():  
 ident = ident + self.ch.lower()  
 self.getc()  
 if ident in Lexer.WORDS:  
 self.sym = Lexer.WORDS[ident]  
 elif len(ident) == 1:  
 self.sym = Lexer.ID  
 self.value = ord(ident) - ord('a')  
 else:  
 self.error('Unknown identifier: ' + ident)  
  
 else:  
 self.error('Unexpected symbol: ' + self.ch)

В методе next\_tok() анализатор получает следующий токен. Тип токена можно получить из атрибута sym, а его значение (если это переменная или число) — из атрибута value.  
  
Анализатор игнорирует пробелы, проверяет, является ли текущий символ специальным символом языка, если нет — проверяет, является ли он числом или идентификатором. Т.е. встретив цифру 1, анализатор продолжит вычитывать символы, пока не встретит не-числовой символ. Аналогично проверяются идентификаторы (там вместо чисел буквы).

**Парсер**  
 Задача парсера, используя токены, полученные от лексического анализатора, сформировать [своего рода дерево](http://en.wikipedia.org/wiki/Abstract_syntax_tree), в котором иерархия и связи отображают структуру кода.

Например:

"if (a < 0) a = 5;"

IF

+-LESS

| +-VAR(a)

| +-NUM(0)

+-SET

+-VAR(a)

+-NUM(5)  
Дерево, которое строится парсером, состоит из узлов. У узла есть тип (IF, LESS, SET, VAR...), значение (если это число или переменная) и несколько дочерних узлов-операндов (в коде — op1, op2, op3). Для if в них хранятся условие и ветки then/else, для циклов — условие и тело цикла.  
Для описания узлов введем класс Node.

Код класса парсера и класса Node:

class Node:  
 def \_\_init\_\_(self, kind, value=None, op1=None, op2=None, op3=None):  
 print('def \_\_init\_\_node', kind, ' ', value, ' ', op1, ' ', op2, ' ', op3, ' ')  
 self.kind = kind  
 self.value = value  
 self.op1 = op1  
 self.op2 = op2  
 self.op3 = op3  
  
  
class Parser:  
 VAR, CONST, ADD, MULT, DIV, SUB, LT, SET, IF1, IF2, WHILE, DO, EMPTY, SEQ, EXPR, PROG = range(16)  
  
 def \_\_init\_\_(self, lexer):  
 print('def \_\_init\_\_parser')  
 self.lexer = lexer  
  
 def error(self, msg):  
 print('def error')  
 print('Parser error:', msg)  
 sys.exit(1)  
  
 def term(self):  
 print('def term')  
 if self.lexer.sym == Lexer.ID:  
 n = Node(Parser.VAR, self.lexer.value)  
 self.lexer.next\_tok()  
 print('term return', n)  
 return n  
 elif self.lexer.sym == Lexer.NUM:  
 n = Node(Parser.CONST, self.lexer.value)  
 self.lexer.next\_tok()  
 print('term return', n)  
 return n  
 else:  
 print('term return', self.paren\_expr())  
 return self.paren\_expr()  
  
 def summa(self):  
 print('def summa')  
 n = self.term()  
 while self.lexer.sym == Lexer.PLUS or self.lexer.sym == Lexer.MINUS or self.lexer.sym == Lexer.MULTIPLY or self.lexer.sym == Lexer.DIVIDE:  
 if self.lexer.sym == Lexer.PLUS:  
 kind = Parser.ADD  
 elif self.lexer.sym == Lexer.MULTIPLY:  
 kind = Parser.MULT  
 elif self.lexer.sym == Lexer.DIVIDE:  
 kind = Parser.DIV  
 else:  
 kind = Parser.SUB  
 self.lexer.next\_tok()  
 n = Node(kind, op1=n, op2=self.term())  
 print('summa return', n)  
 return n  
  
 def test(self):  
 print('def test')  
 n = self.summa()  
 if self.lexer.sym == Lexer.LESS:  
 self.lexer.next\_tok()  
 n = Node(Parser.LT, op1=n, op2=self.summa())  
 print('test return', n)  
 return n  
  
 def expr(self):  
 print('def expr')  
 if self.lexer.sym != Lexer.ID:  
 return self.test()  
 n = self.test()  
 if n.kind == Parser.VAR and self.lexer.sym == Lexer.EQUAL:  
 self.lexer.next\_tok()  
 n = Node(Parser.SET, op1=n, op2=self.expr())  
 print('expr return', n)  
 return n  
  
 def paren\_expr(self):  
 print('def paren\_expr')  
 if self.lexer.sym != Lexer.LPAR:  
 self.error('"(" expected')  
 self.lexer.next\_tok()  
 n = self.expr()  
 if self.lexer.sym != Lexer.RPAR:  
 self.error('")" expected')  
 self.lexer.next\_tok()  
 print('paren\_expr return', n)  
 return n  
  
 def statement(self):  
 print('def statement')  
 if self.lexer.sym == Lexer.IF:  
 n = Node(Parser.IF1)  
 self.lexer.next\_tok()  
 n.op1 = self.paren\_expr()  
 n.op2 = self.statement()  
 if self.lexer.sym == Lexer.ELSE:  
 n.kind = Parser.IF2  
 self.lexer.next\_tok()  
 n.op3 = self.statement()  
 elif self.lexer.sym == Lexer.WHILE:  
 n = Node(Parser.WHILE)  
 self.lexer.next\_tok()  
 n.op1 = self.paren\_expr()  
 n.op2 = self.statement();  
 elif self.lexer.sym == Lexer.DO:  
 n = Node(Parser.DO)  
 self.lexer.next\_tok()  
 n.op1 = self.statement()  
 if self.lexer.sym != Lexer.WHILE:  
 self.error('"while" expected')  
 self.lexer.next\_tok()  
 n.op2 = self.paren\_expr()  
 if self.lexer.sym != Lexer.SEMICOLON:  
 self.error('";" expected')  
 elif self.lexer.sym == Lexer.SEMICOLON:  
 n = Node(Parser.EMPTY)  
 self.lexer.next\_tok()  
 elif self.lexer.sym == Lexer.LBRA:  
 n = Node(Parser.EMPTY)  
 self.lexer.next\_tok()  
 while self.lexer.sym != Lexer.RBRA:  
 n = Node(Parser.SEQ, op1=n, op2=self.statement())  
 self.lexer.next\_tok()  
 else:  
 n = Node(Parser.EXPR, op1=self.expr())  
 if self.lexer.sym != Lexer.SEMICOLON:  
 self.error('";" expected')  
 self.lexer.next\_tok()  
 print('statement return', n)  
 return n  
  
 def parse(self):  
 print('def parse')  
 self.lexer.next\_tok()  
 node = Node(Parser.PROG, op1=self.statement())  
 if (self.lexer.sym != Lexer.EOF):  
 self.error("Invalid statement syntax")  
 print('return node ', node.kind, ' ', node.value)  
 return node

Этот парсер работает рекурсивно, начиная с метода parse().  
Вначале он создает узел «Программа», дочерним узлом которого становится главный оператор программы.  
Операторы формируются в методе statement(). В этом методе проверяется первый токен и в зависимости от него формируется if, if/else, while, do/while, составной оператор (если начинается с фигурной скобки) или просто одиночный оператор, завершающийся точкой с запятой.  
При построении операторов используются методы expr() — получить выражение и paren\_expr() — получить выражение в скобках.  
Выражения строятся из проверок, которые строятся из сумм, которые состоят из термов. А термы в свою очередь состоят из чисел, переменных и выражений в скобках. В доме, который построил Джек.  
Такая вот рекурсия. Как видите, методы соответствуют понятиям описанной выше грамматики.  
На выходе метода parse() получаем объект класса Node, который содержит дерево нашей программы. Это дерево теперь можно преобразовывать в машинный код.

**Машинный код**  
Компилировать будем в простенький байт-код нашей специальной виртуальной машины. Виртуальная машина будет стековой, т.к. они значительно проще регистровых. Вот ее возможные инструкции:

FETCH x - положить на стек значение переменной x

STORE x - сохранить в переменной x значение с вершины стека

PUSH n - положить число n на вершину стека

POP - удалить число с вершины стека

ADD - сложить два числа на вершине стека

SUB - вычесть два числа на вершине стека

LT - сравнить два числа с вершины стека (a < b). Результат - 0 или 1

JZ a - если на вершине стека 0 - перейти к адресу a.

JNZ a - если на вершине стека не 0 - перейти к адресу a.

JMP a - перейти к адресу a

HALT - завершить работу  
Например, операторы «a = 2; b = 5;» преобразуется в такую последовательность инструкций:

PUSH 2 STORE 0 PUSH 5 STORE 1  
Код виртуальной машины очень простой. Он весь описывается в методе run:

IFETCH, ISTORE, IPUSH, IPOP, IADD, IMULT, IDIV, ISUB, ILT, JZ, JNZ, JMP, HALT = range(13)

class VirtualMachine:  
  
 def run(self, program):  
 print('def run')  
 var = [0 for i in xrange(28)]  
 stack = []  
 pc = 0  
 while True:  
 op = program[pc]  
 if pc < len(program) - 1:  
 arg = program[pc + 1]  
  
 if op == IFETCH:  
 stack.append(var[arg]); pc += 2  
 elif op == ISTORE:  
 var[arg] = stack.pop(); pc += 2  
 elif op == IPUSH:  
 stack.append(arg); pc += 2  
 elif op == IPOP:  
 stack.append(arg); stack.pop(); pc += 1  
 elif op == IADD:  
 stack[-2] += stack[-1]; stack.pop(); pc += 1  
 elif op == IMULT:  
 stack[-2] \*= stack[-1]; stack.pop(); pc += 1  
 elif op == IDIV:  
 stack[-2] /= stack[-1]; stack.pop(); pc += 1  
 elif op == ISUB:  
 stack[-2] -= stack[-1]; stack.pop(); pc += 1  
 elif op == ILT:  
 if stack[-2] < stack[-1]:  
 stack[-2] = 1  
 else:  
 stack[-2] = 0  
 stack.pop();  
 pc += 1  
 elif op == JZ:  
 if stack.pop() == 0:  
 pc = arg  
 else:  
 pc += 2  
 elif op == JNZ:  
 if stack.pop() != 0:  
 pc = arg  
 else:  
 pc += 2  
 elif op == JMP:  
 pc = arg;  
 elif op == HALT:  
 break  
  
 print('Execution finished.')  
 for i in xrange(28):  
 if var[i] != 0:  
 print('%c = %d' % (chr(i + ord('a')), var[i]))

**Компилятор**  
Исходный код:

class Compiler:  
 program = []  
 pc = 0  
  
 def gen(self, command):  
 print('def gen')  
 self.program.append(command)  
 self.pc = self.pc + 1  
  
 def compile(self, node):  
 print('def compile')  
 if node.kind == Parser.VAR:  
 self.gen(IFETCH)  
 self.gen(node.value)  
 elif node.kind == Parser.CONST:  
 self.gen(IPUSH)  
 self.gen(node.value)  
 elif node.kind == Parser.ADD:  
 self.compile(node.op1)  
 self.compile(node.op2)  
 self.gen(IADD)  
 elif node.kind == Parser.MULT:  
 self.compile(node.op1)  
 self.compile(node.op2)  
 self.gen(IMULT)  
 elif node.kind == Parser.DIV:  
 self.compile(node.op1)  
 self.compile(node.op2)  
 self.gen(IDIV)  
 elif node.kind == Parser.SUB:  
 self.compile(node.op1)  
 self.compile(node.op2)  
 self.gen(ISUB)  
 elif node.kind == Parser.LT:  
 self.compile(node.op1)  
 self.compile(node.op2)  
 self.gen(ILT)  
 elif node.kind == Parser.SET:  
 self.compile(node.op2)  
 self.gen(ISTORE)  
 self.gen(node.op1.value)  
 elif node.kind == Parser.IF1:  
 self.compile(node.op1)  
 self.gen(JZ);  
 addr = self.pc;  
 self.gen(0);  
 self.compile(node.op2)  
 self.program[addr] = self.pc  
 elif node.kind == Parser.IF2:  
 self.compile(node.op1)  
 self.gen(JZ);  
 addr1 = self.pc;  
 self.gen(0)  
 self.compile(node.op2)  
 self.gen(JMP);  
 addr2 = self.pc;  
 self.gen(0)  
 self.program[addr1] = self.pc  
 self.compile(node.op3)  
 self.program[addr2] = self.pc  
 elif node.kind == Parser.WHILE:  
 addr1 = self.pc  
 self.compile(node.op1)  
 self.gen(JZ);  
 addr2 = self.pc;  
 self.gen(0)  
 self.compile(node.op2)  
 self.gen(JMP);  
 self.gen(addr1);  
 self.program[addr2] = self.pc  
 elif node.kind == Parser.DO:  
 addr = self.pc  
 self.compile(node.op1)  
 self.compile(node.op2)  
 self.gen(JNZ);  
 self.gen(addr);  
 elif node.kind == Parser.SEQ:  
 self.compile(node.op1)  
 self.compile(node.op2)  
 elif node.kind == Parser.EXPR:  
 self.compile(node.op1);  
 self.gen(IPOP)  
 elif node.kind == Parser.PROG:  
 self.compile(node.op1);  
 self.gen(HALT)  
 return self.program

Метод gen() добавляет новый байт (команду или аргумент) в программу (список байт).  
В методе compile() собирается вся программа. Фактически, этот метод рекурсивно обходит дерево узлов и для каждого типа узла генерирует соответствующий код.  
После JMP/JZ, сперва пишем 0, а когда сама ветка скомпилирована и известен адрес, на который надо перейти, чтобы не выполнять эту ветку — значение 0 меняем на фактический адрес.

**Тест**

{i = 3;

a=3; b=5;

a = 1; b = 2; c = a + b;

a = 5; b = 2; c = a - b;

a = 1; b = 2; c = a \* b;

a = 5; b = 2; c = a / b;

a = 5; b = 2; c = b < a;

a = 5; if (a < 10) a = 33;

a = 5; if (10 < a) a = 33; else { a = 1; b = 2; }

a = 10; do { a = a - 2;} while (3 < a);

a = 1; b = 5; while (a < b) { a = a + 3; }}@

**Приложение**

import sys  
  
from pip.\_vendor.msgpack.fallback import xrange  
  
  
class Lexer:  
 NUM, ID, IF, ELSE, WHILE, DO, LBRA, RBRA, LPAR, RPAR, PLUS, MINUS, LESS, \  
 EQUAL, SEMICOLON, MULTIPLY, DIVIDE, EOF = range(18)  
  
 SYMBOLS = {'{': LBRA, '}': RBRA, '=': EQUAL, ';': SEMICOLON, '(': LPAR,  
 ')': RPAR, '+': PLUS, '-': MINUS, '<': LESS, '\*': MULTIPLY, '/': DIVIDE}  
 WORDS = {'if': IF, 'else': ELSE, 'do': DO, 'while': WHILE}  
  
 ch = ' ' # предположим, что первый символ-это пробел  
  
 def error(self, msg):  
 print('def error')  
 print('Lexer error: ', msg)  
 sys.exit(1)  
  
 def getc(self):  
 print('def getc')  
 self.ch = sys.stdin.read(1)  
 def next\_tok(self):  
 print('def next\_tok')  
 self.value = None  
 self.sym = None  
 while self.sym == None:  
 if (self.ch) == '@':  
 self.sym = Lexer.EOF  
 elif self.ch.isspace():  
 self.getc()  
 elif self.ch in Lexer.SYMBOLS:  
 self.sym = Lexer.SYMBOLS[self.ch]  
 self.getc()  
 elif self.ch.isdigit():  
 intval = 0  
 while self.ch.isdigit():  
 intval = intval \* 10 + int(self.ch)  
 self.getc()  
 self.value = intval  
 self.sym = Lexer.NUM  
 elif self.ch.isalpha():  
 ident = ''  
 while self.ch.isalpha():  
 ident = ident + self.ch.lower()  
 self.getc()  
 if ident in Lexer.WORDS:  
 self.sym = Lexer.WORDS[ident]  
 elif len(ident) == 1:  
 self.sym = Lexer.ID  
 self.value = ord(ident) - ord('a')  
 else:  
 self.error('Unknown identifier: ' + ident)  
  
 else:  
 self.error('Unexpected symbol: ' + self.ch)  
  
  
class Node:  
 def \_\_init\_\_(self, kind, value=None, op1=None, op2=None, op3=None):  
 print('def \_\_init\_\_node', kind, ' ', value, ' ', op1, ' ', op2, ' ', op3, ' ')  
 self.kind = kind  
 self.value = value  
 self.op1 = op1  
 self.op2 = op2  
 self.op3 = op3  
  
  
class Parser:  
 VAR, CONST, ADD, MULT, DIV, SUB, LT, SET, IF1, IF2, WHILE, DO, EMPTY, SEQ, EXPR, PROG = range(16)  
  
 def \_\_init\_\_(self, lexer):  
 print('def \_\_init\_\_parser')  
 self.lexer = lexer  
  
 def error(self, msg):  
 print('def error')  
 print('Parser error:', msg)  
 sys.exit(1)  
  
 def term(self):  
 print('def term')  
 if self.lexer.sym == Lexer.ID:  
 n = Node(Parser.VAR, self.lexer.value)  
 self.lexer.next\_tok()  
 print('term return', n)  
 return n  
 elif self.lexer.sym == Lexer.NUM:  
 n = Node(Parser.CONST, self.lexer.value)  
 self.lexer.next\_tok()  
 print('term return', n)  
 return n  
 else:  
 print('term return', self.paren\_expr())  
 return self.paren\_expr()  
  
 def summa(self):  
 print('def summa')  
 n = self.term()  
 while self.lexer.sym == Lexer.PLUS or self.lexer.sym == Lexer.MINUS or self.lexer.sym == Lexer.MULTIPLY or self.lexer.sym == Lexer.DIVIDE:  
 if self.lexer.sym == Lexer.PLUS:  
 kind = Parser.ADD  
 elif self.lexer.sym == Lexer.MULTIPLY:  
 kind = Parser.MULT  
 elif self.lexer.sym == Lexer.DIVIDE:  
 kind = Parser.DIV  
 else:  
 kind = Parser.SUB  
 self.lexer.next\_tok()  
 n = Node(kind, op1=n, op2=self.term())  
 print('summa return', n)  
 return n  
  
 def test(self):  
 print('def test')  
 n = self.summa()  
 if self.lexer.sym == Lexer.LESS:  
 self.lexer.next\_tok()  
 n = Node(Parser.LT, op1=n, op2=self.summa())  
 print('test return', n)  
 return n  
  
 def expr(self):  
 print('def expr')  
 if self.lexer.sym != Lexer.ID:  
 return self.test()  
 n = self.test()  
 if n.kind == Parser.VAR and self.lexer.sym == Lexer.EQUAL:  
 self.lexer.next\_tok()  
 n = Node(Parser.SET, op1=n, op2=self.expr())  
 print('expr return', n)  
 return n  
  
 def paren\_expr(self):  
 print('def paren\_expr')  
 if self.lexer.sym != Lexer.LPAR:  
 self.error('"(" expected')  
 self.lexer.next\_tok()  
 n = self.expr()  
 if self.lexer.sym != Lexer.RPAR:  
 self.error('")" expected')  
 self.lexer.next\_tok()  
 print('paren\_expr return', n)  
 return n  
  
 def statement(self):  
 print('def statement')  
 if self.lexer.sym == Lexer.IF:  
 n = Node(Parser.IF1)  
 self.lexer.next\_tok()  
 n.op1 = self.paren\_expr()  
 n.op2 = self.statement()  
 if self.lexer.sym == Lexer.ELSE:  
 n.kind = Parser.IF2  
 self.lexer.next\_tok()  
 n.op3 = self.statement()  
 elif self.lexer.sym == Lexer.WHILE:  
 n = Node(Parser.WHILE)  
 self.lexer.next\_tok()  
 n.op1 = self.paren\_expr()  
 n.op2 = self.statement();  
 elif self.lexer.sym == Lexer.DO:  
 n = Node(Parser.DO)  
 self.lexer.next\_tok()  
 n.op1 = self.statement()  
 if self.lexer.sym != Lexer.WHILE:  
 self.error('"while" expected')  
 self.lexer.next\_tok()  
 n.op2 = self.paren\_expr()  
 if self.lexer.sym != Lexer.SEMICOLON:  
 self.error('";" expected')  
 elif self.lexer.sym == Lexer.SEMICOLON:  
 n = Node(Parser.EMPTY)  
 self.lexer.next\_tok()  
 elif self.lexer.sym == Lexer.LBRA:  
 n = Node(Parser.EMPTY)  
 self.lexer.next\_tok()  
 while self.lexer.sym != Lexer.RBRA:  
 n = Node(Parser.SEQ, op1=n, op2=self.statement())  
 self.lexer.next\_tok()  
 else:  
 n = Node(Parser.EXPR, op1=self.expr())  
 if self.lexer.sym != Lexer.SEMICOLON:  
 self.error('";" expected')  
 self.lexer.next\_tok()  
 print('statement return', n)  
 return n  
  
 def parse(self):  
 print('def parse')  
 self.lexer.next\_tok()  
 node = Node(Parser.PROG, op1=self.statement())  
 if (self.lexer.sym != Lexer.EOF):  
 self.error("Invalid statement syntax")  
 print('return node ', node.kind, ' ', node.value)  
 return node  
  
  
IFETCH, ISTORE, IPUSH, IPOP, IADD, IMULT, IDIV, ISUB, ILT, JZ, JNZ, JMP, HALT = range(13)  
  
  
class Compiler:  
 program = []  
 pc = 0  
  
 def gen(self, command):  
 print('def gen')  
 self.program.append(command)  
 self.pc = self.pc + 1  
  
 def compile(self, node):  
 print('def compile')  
 if node.kind == Parser.VAR:  
 self.gen(IFETCH)  
 self.gen(node.value)  
 elif node.kind == Parser.CONST:  
 self.gen(IPUSH)  
 self.gen(node.value)  
 elif node.kind == Parser.ADD:  
 self.compile(node.op1)  
 self.compile(node.op2)  
 self.gen(IADD)  
 elif node.kind == Parser.MULT:  
 self.compile(node.op1)  
 self.compile(node.op2)  
 self.gen(IMULT)  
 elif node.kind == Parser.DIV:  
 self.compile(node.op1)  
 self.compile(node.op2)  
 self.gen(IDIV)  
 elif node.kind == Parser.SUB:  
 self.compile(node.op1)  
 self.compile(node.op2)  
 self.gen(ISUB)  
 elif node.kind == Parser.LT:  
 self.compile(node.op1)  
 self.compile(node.op2)  
 self.gen(ILT)  
 elif node.kind == Parser.SET:  
 self.compile(node.op2)  
 self.gen(ISTORE)  
 self.gen(node.op1.value)  
 elif node.kind == Parser.IF1:  
 self.compile(node.op1)  
 self.gen(JZ);  
 addr = self.pc;  
 self.gen(0);  
 self.compile(node.op2)  
 self.program[addr] = self.pc  
 elif node.kind == Parser.IF2:  
 self.compile(node.op1)  
 self.gen(JZ);  
 addr1 = self.pc;  
 self.gen(0)  
 self.compile(node.op2)  
 self.gen(JMP);  
 addr2 = self.pc;  
 self.gen(0)  
 self.program[addr1] = self.pc  
 self.compile(node.op3)  
 self.program[addr2] = self.pc  
 elif node.kind == Parser.WHILE:  
 addr1 = self.pc  
 self.compile(node.op1)  
 self.gen(JZ);  
 addr2 = self.pc;  
 self.gen(0)  
 self.compile(node.op2)  
 self.gen(JMP);  
 self.gen(addr1);  
 self.program[addr2] = self.pc  
 elif node.kind == Parser.DO:  
 addr = self.pc  
 self.compile(node.op1)  
 self.compile(node.op2)  
 self.gen(JNZ);  
 self.gen(addr);  
 elif node.kind == Parser.SEQ:  
 self.compile(node.op1)  
 self.compile(node.op2)  
 elif node.kind == Parser.EXPR:  
 self.compile(node.op1);  
 self.gen(IPOP)  
 elif node.kind == Parser.PROG:  
 self.compile(node.op1);  
 self.gen(HALT)  
 return self.program  
  
  
class VirtualMachine:  
  
 def run(self, program):  
 print('def run')  
 var = [0 for i in xrange(28)]  
 stack = []  
 pc = 0  
 while True:  
 op = program[pc]  
 if pc < len(program) - 1:  
 arg = program[pc + 1]  
  
 if op == IFETCH:  
 stack.append(var[arg]); pc += 2  
 elif op == ISTORE:  
 var[arg] = stack.pop(); pc += 2  
 elif op == IPUSH:  
 stack.append(arg); pc += 2  
 elif op == IPOP:  
 stack.append(arg); stack.pop(); pc += 1  
 elif op == IADD:  
 stack[-2] += stack[-1]; stack.pop(); pc += 1  
 elif op == IMULT:  
 stack[-2] \*= stack[-1]; stack.pop(); pc += 1  
 elif op == IDIV:  
 stack[-2] /= stack[-1]; stack.pop(); pc += 1  
 elif op == ISUB:  
 stack[-2] -= stack[-1]; stack.pop(); pc += 1  
 elif op == ILT:  
 if stack[-2] < stack[-1]:  
 stack[-2] = 1  
 else:  
 stack[-2] = 0  
 stack.pop();  
 pc += 1  
 elif op == JZ:  
 if stack.pop() == 0:  
 pc = arg  
 else:  
 pc += 2  
 elif op == JNZ:  
 if stack.pop() != 0:  
 pc = arg  
 else:  
 pc += 2  
 elif op == JMP:  
 pc = arg;  
 elif op == HALT:  
 break  
  
 print('Execution finished.')  
 for i in xrange(28):  
 if var[i] != 0:  
 print('%c = %d' % (chr(i + ord('a')), var[i]))  
  
  
l = Lexer()  
print('lex')  
p = Parser(l)  
print('parse')  
ast = p.parse()  
print('compile')  
c = Compiler()  
  
program = c.compile(ast)  
print('vm')  
vm = VirtualMachine()  
print('run')  
vm.run(program)