Оглавление

[ВВЕДЕНИЕ 3](#_Toc41661407)

[ПРОЕКТИРОВАНИЕ КОМПИЛЯТОРА 4](#_Toc41661408)

[1.1 ПРОЕКТИРОВАНИЕ СКАНЕРА ТОКЕНОВ 4](#_Toc41661409)

[1.2 Структура данных - Токен 5](#_Toc41661410)

[1.3 Проектирование парсера 6](#_Toc41661411)

[1.4 Проектирование симулятора операций 13](#_Toc41661412)

[1.5 Объединение модулей в единую программу 16](#_Toc41661413)

[1.6 Оптимизация 17](#_Toc41661414)

[2.1 Упаковка приложения 20](#_Toc41661415)

[2.5 Руководство пользователя 21](#_Toc41661416)

[3. Тесты 21](#_Toc41661417)

[ЗАКЛЮЧЕНИЕ 26](#_Toc41661418)

[ПРИЛОЖЕНИЕ 27](#_Toc41661419)

[Run.java 27](#_Toc41661420)

[Token.java 28](#_Toc41661421)

[Token\_reader.java 28](#_Toc41661422)

[Parser.java 34](#_Toc41661423)

[Symbol.java 53](#_Toc41661424)

[Table\_symbols.java 55](#_Toc41661425)

[Operations\_simulator.java 56](#_Toc41661426)

[Comments\_remover.java 63](#_Toc41661427)

[Blank\_begin\_end\_remover.java 64](#_Toc41661428)

[Blank\_repeat\_until\_remover.java 67](#_Toc41661429)

Изм.

Лист

№ докум.

Подпись

Дата

Лист

2

RU. 02068048.502900.01.01

Разраб.

Толокнов Е.А

Провер.

Монахов Ю.М.

Реценз.

Н. Контр.

Монахов Ю.М.

Утверд.

Компилятор

Лит.

Листов

29

ВлГУ

# ВВЕДЕНИЕ

Задачи курсовой работы заключаются в:

Описание программы, предназначенной для компиляции программ на некотором исходном языке программирования под целевую платформу. Исходным языком, предназначенным для компиляции является выбранное мной подмножество синтаксиса языка Pascal. Язык, выбранный для разработки Java. Целевая платформа – Java Virtual Machine.

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 файл с кодом программы на языке Pascal
* В процессе работы компилятора выполняется этот код на целевой платформе
* На выход получаем этот файл, очищенный от комментариев и пустых блоков

# ПРОЕКТИРОВАНИЕ КОМПИЛЯТОРА

# 1.1 ПРОЕКТИРОВАНИЕ СКАНЕРА ТОКЕНОВ

**Сканер токенов** – представляет собой модуль, предназначенный для инициализации списка токенов программы. Все возможно используемые ключевые слова перечислены в keywords.txt. Помимо зарезервированных слов генерируется структура данных hashmap, содержащая символы операторов, которые используются в коде программы (напр. Операции сложения, вычитания, перечисления, точки с запятой и тп).

Отрывок из кода приведен ниже:

*KEYWORDS\_TOKEN* = new HashMap<>();  
 String word;  
 String filename;  
  
  
 filename = Run.*filepath*+"/keywords.txt";  
  
 try {  
  
 Scanner sc = new Scanner(new File(filename));  
 while(sc.hasNext()){  
 word = sc.next();  
 *KEYWORDS\_TOKEN*.put(word, String.*format*("KW~%s", word.toUpperCase()));  
 }  
 } catch (FileNotFoundException e) {  
 e.printStackTrace();  
 }  
}  
  
static {  
 *OPERATORS\_TOKEN* = new HashMap<>();  
 *OPERATORS\_TOKEN*.put("(", "KW~OPEN\_PARENTHESIS");  
 *OPERATORS\_TOKEN*.put(")", "KW~CLOSE\_PARENTHESIS");

Далее заполняется hashmap для распознавания всевозможных символов.

Основной метод модуля – read – читает файл с кодом посимвольно и вызывает метод Char\_type\_checker – для проверки типа этого символа. После прочтения генерируется токен EOF и в основной модуль возвращается список токенов.

Ключевой метод модуля приведен ниже:

public static ArrayList<Token> read(File file) throws FileNotFoundException {  
 // Разделитель для чтения символов  
 Scanner sc = new Scanner(file).useDelimiter("");  
  
 while (sc.hasNext()) {  
 char element = sc.next().toLowerCase().charAt(0);  
  
 *Char\_type\_checker*(element);  
 }  
  
 *token\_Name* = "EOF";  
 *Token\_generate*("KW~EOF");  
  
 return *Array\_List\_tokens*;  
}

Методом Token\_generate непосредственно добавляются токены из программы в возвращаемый список токенов

public static void Token\_generate(String tokenType) {  
 Token t = new Token(tokenType, *token\_Name*);  
 *Array\_List\_tokens*.add(t); // Токен добавляется в список  
  
 *token\_Name* = "";  
}

# 1.2 Структура данных - Токен

Эта структура данных используется парсером и сканером токенов. Представляет собой пару значений – Тип токена и значение токена. Может возвращать оба значения, а также переопределять или устанавливать собственный тип.

/\*  
Например  
KW~FLOATLIT: 0  
 token\_Type: "KW~FLOATLIT"  
 token\_Value: "0"  
 \*/  
  
public final class Token {  
 private String token\_Type;  
 private final String token\_Value;  
  
 public Token(String token\_Type, String token\_Value){  
 this.token\_Type = token\_Type;  
 this.token\_Value = token\_Value;  
  
 }  
  
 @Override  
 public String toString(){  
 return token\_Value;  
 }  
 public String get\_Token\_Type() {  
 return token\_Type;  
 }  
 public void set\_Token\_Type(String tokenType) {  
 this.token\_Type = tokenType;  
 }  
 public String get\_Token\_Val() {  
 return token\_Value;  
 }  
}

# 1.3 Проектирование парсера

Изначально метод парсера set\_Tokens\_list\_Iterator вызывается из основного модуля программы. Этим методом мы получаем итератор от сгенерированного списка токенов, чтобы далее парсер смог прочесть его. Далее управление переходит в парсер, как только в основном модуле вызывается Parser.parse() для получения байтового списка инструкций, которые будут в дальнейшем использованы.

Парсер – модуль программы (класс), предназначенный для чтения списка токенов программы и генерации списка инструкций - байтового представления кодов предусмотренных операций (имеющих тип перечисления).

Каждый считанный токен с помощью метода match сравнивается с предусмотренным шаблоном той или иной операции и генерируются соответствующие коды операций (Например от первых считанных токенов ожидается “Program <identifier>” и символ “;”).

В парсере используется структура данных – символ и таблица символов (Набор символов в области видимости).

Символ и его параметры:

/\*  
Например:  
 var x,y,z : integer;  
 x := 2  
 For variable x:  
 name = "x"  
 tokenType = "TK\_A\_VAR"  
 dataType = TYPE.I  
 value = 2  
 address = 0  
 \*/  
  
public class Symbol {  
 private String name;  
 private String Token\_Type;  
 private final Parser.TYPE Data\_Type;  
 private int Address;  
 private int Address\_return; // Возвращаемый адрес для процедуры  
  
 private Object low\_value\_range; // Нижняя граница массива  
 private Object high\_value\_range; // Верхняя граница массива  
  
 private Parser.TYPE index\_Type; // Тип индекса в массиве  
 private Parser.TYPE value\_Type; // Тип значений в массиве  
  
 Symbol next\_entry\_pointer; // Указатель на следующее вхождение в действующем списке таблицы символов  
  
 public Symbol(String name, String Token\_Type, Parser.TYPE Data\_Type, int address) { //Структура символа  
 this.name = name;  
 this.Token\_Type = Token\_Type;  
 this.Data\_Type = Data\_Type;  
 this.Address = address;  
 }

Таблица символов

public final class Table\_symbols {  
  
 static class Scope {  
 Symbol[] Symbol\_Table = new Symbol[*HASH\_TABLE\_SIZE*]; // Таблица символов для текущей области видимости  
 Scope next\_scope\_pointer = null; // Указатель на следующую внешнюю область  
 }  
  
 private static final int *HASH\_TABLE\_SIZE* = 211;  
 private static final Scope *Scope\_header* = new Scope();  
  
 public static void insert(Symbol symbol) {  
 int hash\_Value = *hash*(symbol.getName());  
  
 Symbol Cursor\_bucket = *Scope\_header*.Symbol\_Table[hash\_Value];  
 if (Cursor\_bucket == null) {  
 // Список пуст  
 *Scope\_header*.Symbol\_Table[hash\_Value] = symbol;  
 } else {  
 // Существующие символы в списке  
 while (Cursor\_bucket.next\_entry\_pointer != null) {  
 Cursor\_bucket = Cursor\_bucket.next\_entry\_pointer;  
 }  
  
 // Добавить в конец списка  
 Cursor\_bucket.next\_entry\_pointer = symbol;  
 }  
 }  
  
 public static Symbol find\_and\_get(String symbol\_Name) {  
 int hash\_Value = *hash*(symbol\_Name);  
 Symbol Cursor\_bucket = *Scope\_header*.Symbol\_Table[hash\_Value];  
 Scope Cursor\_scope = *Scope\_header*;  
  
 while (Cursor\_scope != null) {  
 while (Cursor\_bucket != null) {  
 if (Cursor\_bucket.getName().equals(symbol\_Name)) {  
 return Cursor\_bucket;  
 }  
 Cursor\_bucket = Cursor\_bucket.next\_entry\_pointer;  
 }  
 Cursor\_scope = Cursor\_scope.next\_scope\_pointer;  
 }  
  
 // Символ не существует  
 return null;  
 }  
  
 public static int hash(String symbol\_Name) {  
 int hash\_int\_val = 0;  
 for (int i = 0; i < symbol\_Name.length(); i++) {  
 hash\_int\_val = hash\_int\_val + hash\_int\_val + symbol\_Name.charAt(i);  
 }  
  
 hash\_int\_val = hash\_int\_val % *HASH\_TABLE\_SIZE*;  
  
 return hash\_int\_val;  
 }  
  
  
}

Символ хранит в себе значения и по таблице символов можно связать все упоминания этого символа в программе. Например для goto – goto label1 запишет в символ label1 свой адрес. А при следующем упоминании label1 в качестве метки – label1: statement() – этот label1 уже хранит в себе значение адреса, соответствующего адресу полученному в goto label1, и по этому адресу будет сгенерирован код, указывающий на текущий адрес этого label1. Т.е создастся некоторая связь между адресами в процессе генерации промежуточного кода в список инструкций.

Метод parse считывает изначальный токен из списка и сравнивает его с ожидаемым шаблоном объявления программы, и далее управление передается в метод program().

public static Byte[] parse() {  
 *getToken*(); // Получить изначальный токен  
  
 *tokens\_match*("KW~PROGRAM");  
 *tokens\_match*("KW~IDENTIFIER");  
 *tokens\_match*("KW~SEMI\_COLON"); //Должно соответствовать шаблону - Program <Имя> ;  
  
 *program*();  
  
 return *byteArray*;  
}

Метод program() передает управление в метод declarations() и после в begin()

public static void program() {  
 *declarations*();  
 *begin*();  
}

Метод declarations() описывает инициализацию различных переменных , процедур, меток и также реагирует на начало блока как конец объявлений и возврат из метода. Соответственно токенам вызываются методы var\_init(), procedure\_init(), label\_init().

public static void declarations() { //Различные объявления - переменные, процедуры, метки, начало блока  
 while (true) {  
 switch (*Token\_current*.get\_Token\_Type()) {  
 case "KW~VAR":  
 *var\_init*();  
 break;  
 case "KW~PROCEDURE":  
 *procedure\_init*();  
 break;  
 case "KW~LABEL":  
 *label\_init*();  
 break;  
 case "KW~BEGIN":  
 return;  
 }  
 }  
}

Метод var\_init() описывает инициализацию переменных в исходном языке.

public static void var\_init() {  
 while (true) {  
 if ("KW~VAR".equals(*Token\_current*.get\_Token\_Type())) {  
 *tokens\_match*("KW~VAR");  
 } else {  
  
 break;  
 }  
  
 // Хранение переменных в списке  
 ArrayList<Token> var\_Array = new ArrayList<>();  
  
 while ("KW~IDENTIFIER".equals(*Token\_current*.get\_Token\_Type())) {  
 *Token\_current*.set\_Token\_Type("KW~A\_VAR");  
 var\_Array.add(*Token\_current*);  
  
 *tokens\_match*("KW~A\_VAR");  
  
 if ("KW~COMMA".equals(*Token\_current*.get\_Token\_Type())) {  
 *tokens\_match*("KW~COMMA");  
 }  
 }  
  
 *tokens\_match*("KW~COLON");  
 String Type\_data = *Token\_current*.get\_Token\_Type();  
 *tokens\_match*(Type\_data);  
  
 // Добавление соответствующего типа данных для каждого идентификатора и вставка его в таблицу символов  
 for (Token var : var\_Array) {  
  
 Symbol symbol = new Symbol(var.get\_Token\_Val(),  
 "KW~A\_VAR",  
 *HASH\_MAP\_types*.get(Type\_data.toLowerCase().substring(3)),  
 *data\_pointer*);  
  
 *data\_pointer* += 4;  
  
  
 if (Table\_symbols.*find\_and\_get*(var.get\_Token\_Val()) == null) {  
 Table\_symbols.*insert*(symbol);  
 }  
 }  
  
 if (Type\_data.equals("KW~ARRAY")) {  
 *array\_init*(var\_Array);  
 }  
  
 *tokens\_match*("KW~SEMI\_COLON");  
  
 }  
}

Метод procedure\_init() описывает объявление процедур, где, как и в основном блоке есть begin – statements – end структура. Процедура манипулирует объявленными переменными или вызывает операции вывода на экран, при этом ее можно вызвать из любого места в основном теле программы по ее имени. Так как это процедура, то она выполнится лишь тогда, когда на нее будет передано управление непосредственно из основного блока программы, то есть при первичном анализе токенов должен сохраняться адрес входа в процедуру и вовзрата из процедуры. Это нужно в процессе выполнения программы для перехода в основное тело без предшествующего выполнения самой процедуры, т.к сперва выполняется основное тело до первого вызова процедуры, куда и будет переход.

Генерация JMP для перемещения к процедуре из основного блока.

*generate\_Operation\_Code*(Operations\_code.*JMP*);  
symbol.setAddress\_return(*ip*);  
*generate\_Address*(0);

Аналогично генерируется JMP в то место, откуда был вызов, как только мы вернемся из процедуры.

Объявление процедуры

if (*Token\_current*.get\_Token\_Type().equals("KW~PROCEDURE")) {  
 *tokens\_match*("KW~PROCEDURE");  
 *Token\_current*.set\_Token\_Type("KW~A\_PROC");  
  
 String procedure\_Name = *Token\_current*.get\_Token\_Val();  
  
 *tokens\_match*("KW~A\_PROC");  
 *tokens\_match*("KW~SEMI\_COLON"); //Procedure <имя>;

Тело процедуры

// тело программы  
*tokens\_match*("KW~BEGIN");  
*statements*();  
*tokens\_match*("KW~END");  
*tokens\_match*("KW~SEMI\_COLON");

Метод label\_init() – инициализирует метки, которые будут использованы в п рограмме для перемещения к ним с помощью goto вызовов.

if ("KW~LABEL".equals(*Token\_current*.get\_Token\_Type())) {  
 *tokens\_match*("KW~LABEL");  
} else {  
  
 break;  
}  
  
// Хранение меток (label) в спике  
ArrayList<Token> Array\_labels = new ArrayList<>();  
  
while ("KW~IDENTIFIER".equals(*Token\_current*.get\_Token\_Type())) { //Перечисление объявляемых переменных типа label  
 *Token\_current*.set\_Token\_Type("KW~A\_LABEL");  
 Array\_labels.add(*Token\_current*);  
  
 *tokens\_match*("KW~A\_LABEL");  
  
 if ("KW~COMMA".equals(*Token\_current*.get\_Token\_Type())) {  
 *tokens\_match*("KW~COMMA"); // перечисление через запятую  
 }  
}

После всех объявлений и при обнаружении begin токена, относящегося именно к основному блоку программы, вызывается метод begin() из метода program().

Метод begin() – описывает основной блок программы, ожидая токены begin, а после statements() метода – end, символ точка и токен EOF, который был сгенерирован еще при первичной генерации токенов. После всего этого генерируется код операции выхода из программы – как конца компиляции.

public static void begin() {  
 *tokens\_match*("KW~BEGIN");  
 *statements*();  
 *tokens\_match*("KW~END");  
 *tokens\_match*("KW~DOT");  
 *tokens\_match*("KW~EOF");  
 *generate\_Operation\_Code*(Operations\_code.*HALT*);  
}

Метод statements() – считывает все дальнейшие токены и вызывает соответствующие генераторы кодов операций соответствующих токену (например под токен Writeln вызывается statements\_writeln(), где описывается шаблон этой операции и генерируется дальнейший код).

При прочтении списка токенов вызывается tokens\_match(), что проверяет совпадение типов токенов – ожидаемого и прочитанного. В случае успеха программа продолжает читать токены.

public static void tokens\_match(String tokenType) {  
 if (!tokenType.equals(*Token\_current*.get\_Token\_Type())) {  
 throw new Error(String.*format*("Token type (%s) does not match current token type (%s)", tokenType, *Token\_current*.get\_Token\_Type()));  
 } else {  
  
 *getToken*();  
 }  
}

Метод getToken() – основной метод чтения токенов

public static void getToken() {  
 if (*Token\_iterator*.hasNext()) {  
 *Token\_current* = *Token\_iterator*.next();  
 }  
}

Метод generate\_Operation\_Code – записывает в список инструкций порядковый номер той или иной указанной операции, чтобы потом в симуляторе операций по ее коду из этого списка можно было совершить те или иные операции на целевой платформе. По этому же списку (по его индексам) можно ориентироваться при применении JMP, goto и procedure.

public static void generate\_Operation\_Code(Operations\_code b) {  
  
 *byteArray*[*ip*++] = (byte) (b.ordinal());  
}

Метод generate\_Address – добавляет в список инструкций некоторое значение, которое относится к операции. Метод вызывается после generate\_Operation\_Code, тем самым закрепляя за операцией некоторое значение (Для типов Int, Float. Boolean воспринимается как 1 и 0 (int) , Char – как char представление int - generate\_Address(Token\_current.get\_Token\_Val().charAt(0));)

public static void generate\_Address(int a) {  
  
 byte[] bytes\_int = ByteBuffer.*allocate*(*ADDRESS\_SIZE*).putInt(a).array();  
  
 for (byte b : bytes\_int) {  
 *byteArray*[*ip*++] = b;  
 }  
}  
  
public static void generate\_Address(float a) {  
  
 byte[] intBytes = ByteBuffer.*allocate*(*ADDRESS\_SIZE*).putFloat(a).array();  
  
 for (byte b : intBytes) {  
 *byteArray*[*ip*++] = b;  
 }  
}

# 1.4 Проектирование симулятора операций

В основном методе программы сгенерированный парсером список операций (инструкций с аргументами) передается в класс operations\_simulator в качестве аргумента для его метода - set\_Instructions. Указанным методом симулятор операций получает в свое распоряжение этот список инструкций. Далее в основном методе программы вызывается метод симулятора .simulate() , передающий управление в симулятор операций.

Симулятор операций – модуль, который считывает коды операций из предоставленного списка и вызывает последовательности операций соответственно этому коду.

Например:

switch (oper\_Code) {

case *PRINT\_BOOL*:  
 *printBool*();  
 break;

…

Операции выполняются на целевой платформе JVM как непосредственно последовательности JAVA – функций , процедур, операций над переменными (их значения , как и прочие величины извлекаются из стека).

Стек формируется симуляцией инструкций типа push (вызов stack.push) из списка инструкций, за которой следует значение, предназначенное для помещения в стек, а также из других операций (т.е не только push типа), если это необходимо. Для доступа к величинам стека вызывается stack.pop, с верхушки которого снимается очередное хранящееся значение.

Для вывода информации на экран различными операциями вызывается например System.out.println( stack.pop (непосредственно со стека) или val (вычисляемое значение) или <константное значение>);

Пример одной из таких операций:

private static void printBool() { //Вывод на экран булева значения  
 int val = (int) *stack*.pop();  
 if (val == 1) {  
 *s* += ("printBool " + val + " (True)") + "\n";  
 System.*out*.print("True");  
 } else {  
 *s* += ("printBool " + val + " (False)") + "\n";  
 System.*out*.print("False");  
 }  
}

Переменные представляют собой область памяти, под которую выделяется 4 байта, где и хранятся их значения. При работе с переменными компилятор воспринимает их не как переменные а как участки выделенной памяти и обращается к ним по указателю на адрес (их место в массиве данных) и совершает некоторые действия с их значениями.

Такие операции описаны в методе симулятора put() и get()

private static void put() { //В область памяти записывается последовательность байт, то есть некоторое значение  
 Object val = *stack*.pop();  
 *data\_pointer* = (int) *stack*.pop();  
  
  
 byte[] value\_Bytes;  
 if (val instanceof Integer) {  
 value\_Bytes = ByteBuffer.*allocate*(4).putInt((int) val).array();  
 } else {  
 value\_Bytes = ByteBuffer.*allocate*(4).putFloat((float) val).array();  
 }  
  
 *s* += ("put (pointer:" + *data\_pointer* + ") : (bytes: ");  
 for (byte b : value\_Bytes) {  
 *Array\_data*[*data\_pointer*++] = b;  
 *s* = *s* + b;  
 }  
 *s* += (") : " + val) + "\n";  
}

private static void get() { //Из стека извлекается указатель на некоторый элемент  
 *data\_pointer* = (int) *stack*.pop();  
 int val = *get\_Data*(*data\_pointer*);  
 *s* += ("get " + val + " : " + *data\_pointer*) + "\n";  
 *stack*.push(val);  
}

Как только все операции из списка инструкций будут выполнены, очередь дойдет до вызова halt и выхода из программы, а значит завершения компиляции.

public static void halt() { //Выход из программы с кодом 0, компиляция выполнена  
 *s* += ("halt : code 0") + "\n";  
 System.*out*.print("\nProgram finished with exit code 0\n");  
  
 try {  
 Files.*write*(Paths.*get*(Run.*filepath*+Run.*filename*+"-operations.txt"), *s*.getBytes());  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 System.*exit*(0);  
}

# 1.5 Объединение модулей в единую программу

Основной метод программы – Run – включает в себя набор вызовов и передач аргументов в методы других модулей компилятора.

При запуске потребуется ввести путь к папке, где лежит example (src\).

Далее указать название файла с расширением .pas, где содержится исходный код, предназначенный для компиляции.

Также осуществляет предварительную оптимизацию кода перед непосредственно компиляцией (удаление комментариев, пустых блоков).

import java.io.\*;  
import java.nio.file.Files;  
import java.nio.file.Paths;  
import java.util.\*;  
  
public final class Run {  
 public static String *filepath* ;  
 public static String *filename* ;  
 public static void main(String[] args) throws IOException {  
  
 String str;  
 // "src/example/1.pas";  
 Scanner finput = new Scanner(System.*in*);  
 System.*out*.println("Path to /example folder:");  
  
 *filepath* = finput.nextLine();  
  
  
 System.*out*.println(*filepath*+"/example/"+".Input filename:");  
 *filename* = finput.nextLine();  
 finput.close();  
String finalpath = *filepath*+"/example/"+*filename*;  
 File file = new File(finalpath);  
  
 Scanner in = new Scanner(file);  
  
 str = Comments\_remover.*remover*(in);  
 in.close();  
  
 Files.*write*(Paths.*get*(finalpath), str.getBytes());  
  
 new StringBuilder();  
 StringBuilder code\_optimizer\_str;  
 Scanner in2 = new Scanner(file);  
 code\_optimizer\_str = new StringBuilder();  
 while (in2.hasNext())  
 code\_optimizer\_str.append(in2.nextLine()).append("\n ");  
 in2.close();  
  
  
 int levellength = 0;  
  
 System.*out*.println("Removing empty blocks...");  
 while (code\_optimizer\_str.length() != levellength) {  
  
  
 System.*out*.print("\*");  
  
  
 str = Blank\_begin\_end\_remover.*remover*(code\_optimizer\_str.toString());  
 code\_optimizer\_str = new StringBuilder(str);  
  
  
 levellength = code\_optimizer\_str.length();  
  
  
 str = Blank\_repeat\_until\_remover.*remover*(code\_optimizer\_str.toString());  
 code\_optimizer\_str = new StringBuilder(str);  
  
  
 }  
 System.*out*.println("\nFinished clearing...");  
 Files.*write*(Paths.*get*(finalpath), code\_optimizer\_str.toString().getBytes());  
  
  
 ArrayList<Token> Tokens\_list = Token\_reader.*read*(new File(finalpath));  
  
  
 Parser.*set\_Tokens\_list\_Iterator*(Tokens\_list);  
  
 Byte[] instructions = Parser.*parse*();  
 Operations\_simulator.*set\_Instructions*(instructions);  
  
  
 Operations\_simulator.*simulate*();  
  
  
 }  
  
  
}

# 1.6 Оптимизация

Для того, чтобы программа скомпилировалась успешно, потребуется ввести ее в виде, пригодном для обработки – т.е в исходном коде не должно быть лишних символов и комментариев, т.к компилятор выдаст ошибку.

Комментарии удаляются путем вызова метода класса Comments\_Remover.remover().

Ниже описан вызов для очистки комментариев и сборка строкового представления кода для его дальнейшей оптимизации

str = Comments\_remover.*remover*(in);  
in.close();  
  
Files.*write*(Paths.*get*(finalpath), str.getBytes());  
  
new StringBuilder();  
StringBuilder code\_optimizer\_str;  
Scanner in2 = new Scanner(file);  
code\_optimizer\_str = new StringBuilder();  
while (in2.hasNext())  
 code\_optimizer\_str.append(in2.nextLine()).append("\n ");  
in2.close();

Соответствующий метод в классе Comments\_remover:

import java.util.Scanner;  
  
public class Comments\_remover {  
 public static String remover(Scanner in)  
 {  
 String str = "";  
 String subs, subs\_delete;  
 while(in.hasNext())  
 {  
 subs = in.nextLine();  
 if (subs.contains("//")) {  
 if (subs.contains(" //"))  
 subs\_delete = subs.substring(subs.indexOf(" //"));  
 else  
 subs\_delete = subs.substring(subs.indexOf("//"));  
 subs = subs.replace(subs\_delete, "");  
 }  
 if (subs.contains("{")) {  
 while (in.hasNext()) {  
 subs += in.nextLine();  
 if (subs.contains("//")) {  
 if (subs.contains(" //"))  
 subs\_delete = subs.substring(subs.indexOf(" //"));  
else  
 subs\_delete = subs.substring(subs.indexOf("//"));  
 subs = subs.replace(subs\_delete, "");  
 }  
 if (subs.contains("}"))  
 break;  
 }  
 if (subs.contains(" {"))  
 subs\_delete = subs.substring(subs.indexOf(" {"));  
 else  
 subs\_delete = subs.substring(subs.indexOf("{"));  
 if (subs\_delete.contains("} "))  
 subs\_delete = subs\_delete.substring(0, subs\_delete.indexOf("} ") + 1);  
 else  
 subs\_delete = subs\_delete.substring(0, subs\_delete.indexOf("}") + 1);  
 subs = subs.replace(subs\_delete, "");  
 }  
  
 str += subs + "\r\n";  
 }  
  
  
  
 in.close();  
  
  
  
 return str;  
 }  
}

Далее осуществляется оптимизация – удаление пустых блоков while, repeat, for

int levellength = 0;  
  
System.*out*.println("Removing empty blocks...");  
while (code\_optimizer\_str.length() != levellength) {  
  
  
 System.*out*.print("\*");  
  
  
 str = Blank\_begin\_end\_remover.*remover*(code\_optimizer\_str.toString());  
 code\_optimizer\_str = new StringBuilder(str);  
  
  
 levellength = code\_optimizer\_str.length();  
  
  
 str = Blank\_repeat\_until\_remover.*remover*(code\_optimizer\_str.toString());  
 code\_optimizer\_str = new StringBuilder(str);  
  
  
}  
System.*out*.println("\nFinished clearing...");  
Files.*write*(Paths.*get*(finalpath), code\_optimizer\_str.toString().getBytes());

Соответствующие методы в каждом из классов описаны ниже

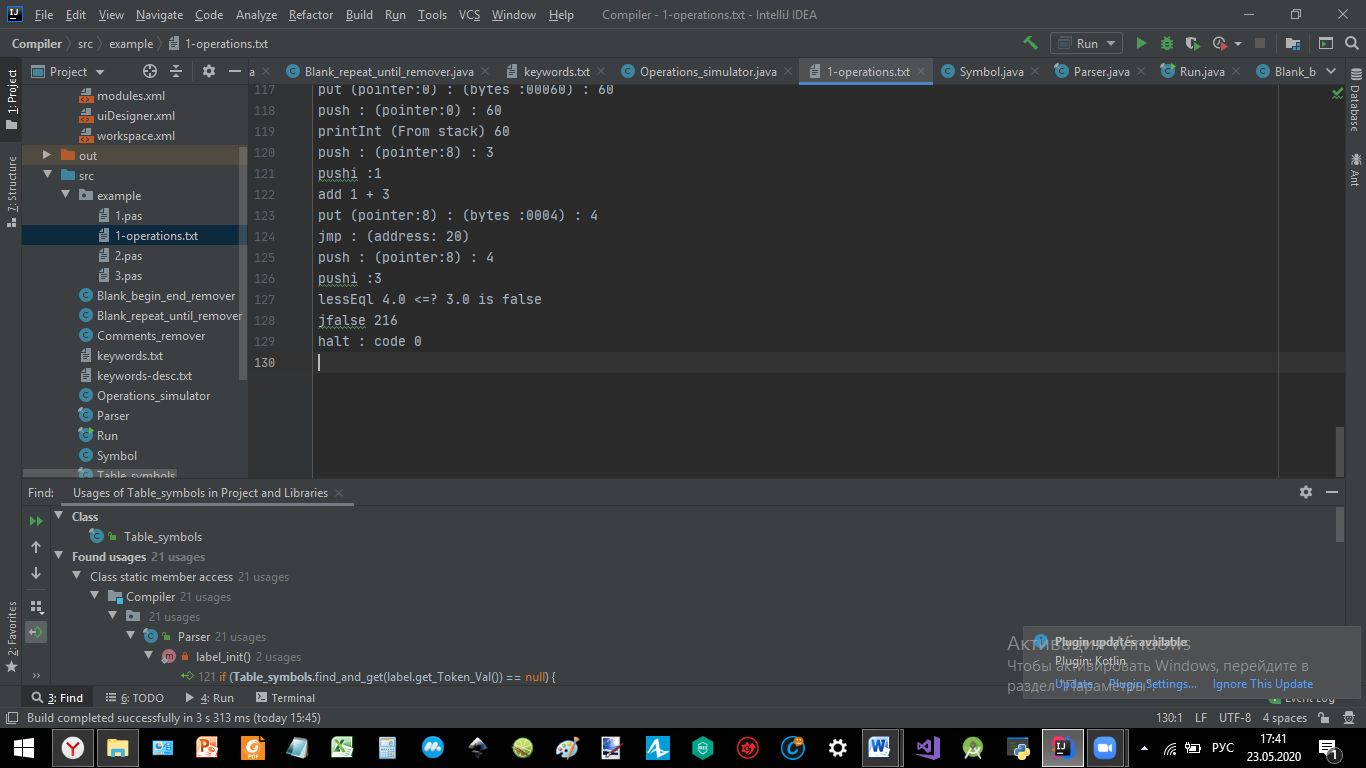
import java.util.LinkedList;  
  
public class Blank\_begin\_end\_remover {  
 private static LinkedList<String> *ScannedList*;  
 private static LinkedList<String> *ClearedList*;  
 private static Boolean *cleared* = false;  
 private static Boolean *useful\_block* = false;  
 private static int *loop\_type* = 0; // 0 - for, 1 - while  
 public static String remover(String input\_str)  
 {  
*…*

import java.util.LinkedList;  
  
public class Blank\_repeat\_until\_remover {  
 private static LinkedList<String> *ScannedList*;  
 private static LinkedList<String> *ClearedList*;  
 private static Boolean *cleared* = false;  
 private static Boolean useful\_block = false;  
 public static String remover(String input\_str)  
 {  
…

После оптимизации, обработанный код записывается в исходный файл, и тот далее отправляется в сканер токенов.

После того, как компиляция завершится, также будет создан файл <filename>-operations.txt, где можно посмотреть порядок вызовов операций компилятора с использующимися в них аргументами.

Пример (Рисунок 1.)



**Рисунок 1. Список операций при выполнении программы**

# 2.1 Упаковка приложения

Приложение находится в папке src/ в виде нескольких файлов проекта .java, с подкаталогом example, где лежат файлы для теста.

# 2.5 Руководство пользователя

Для запуска потребуется в командной строке указать путь к папке src/

cd …src/

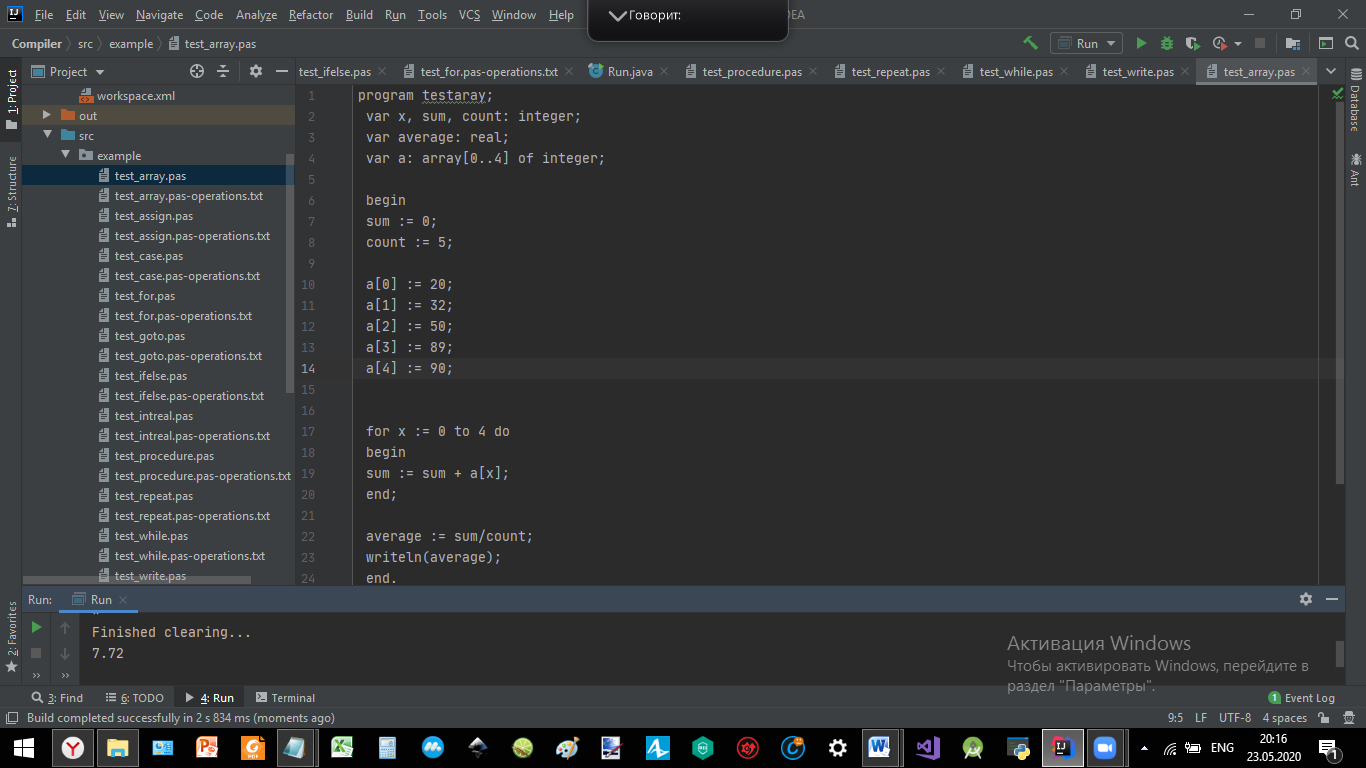
Скомпилировать

javac Run.java

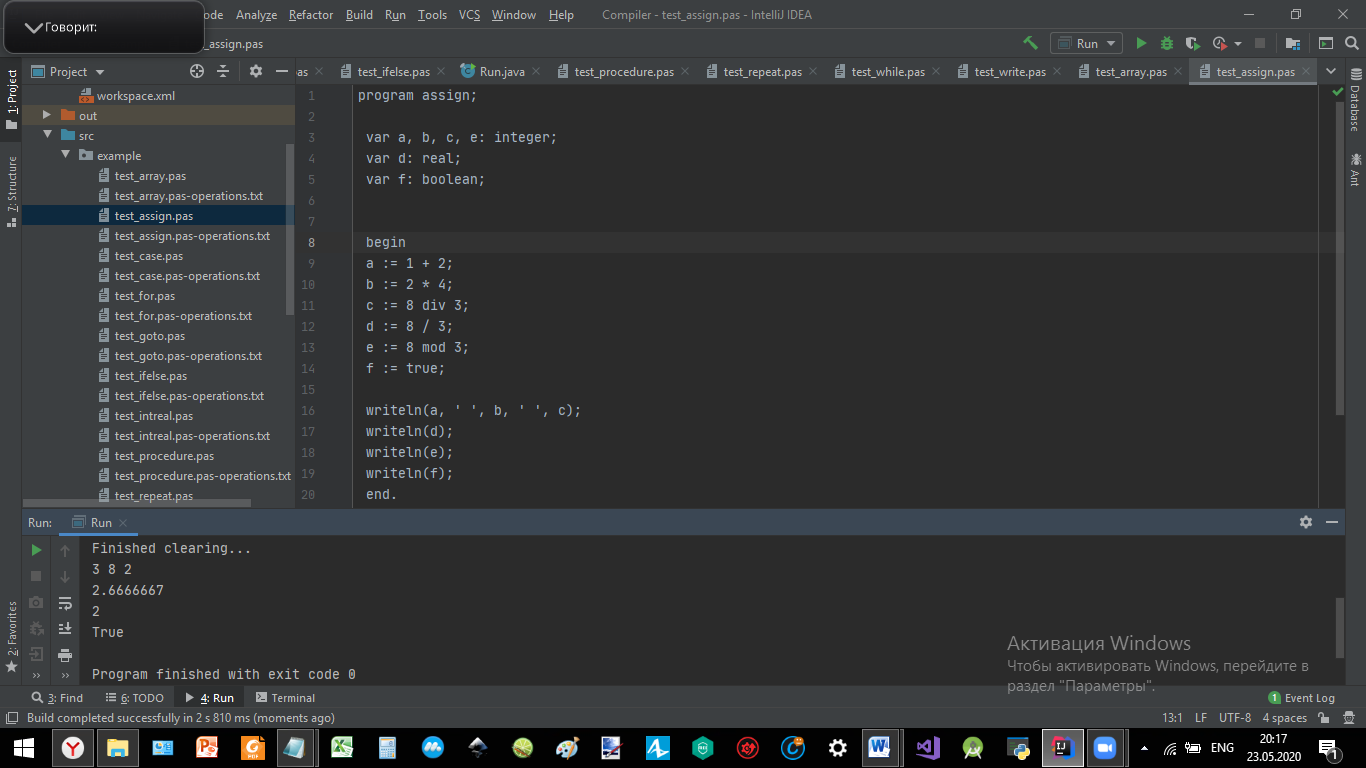
Запустить

java Run

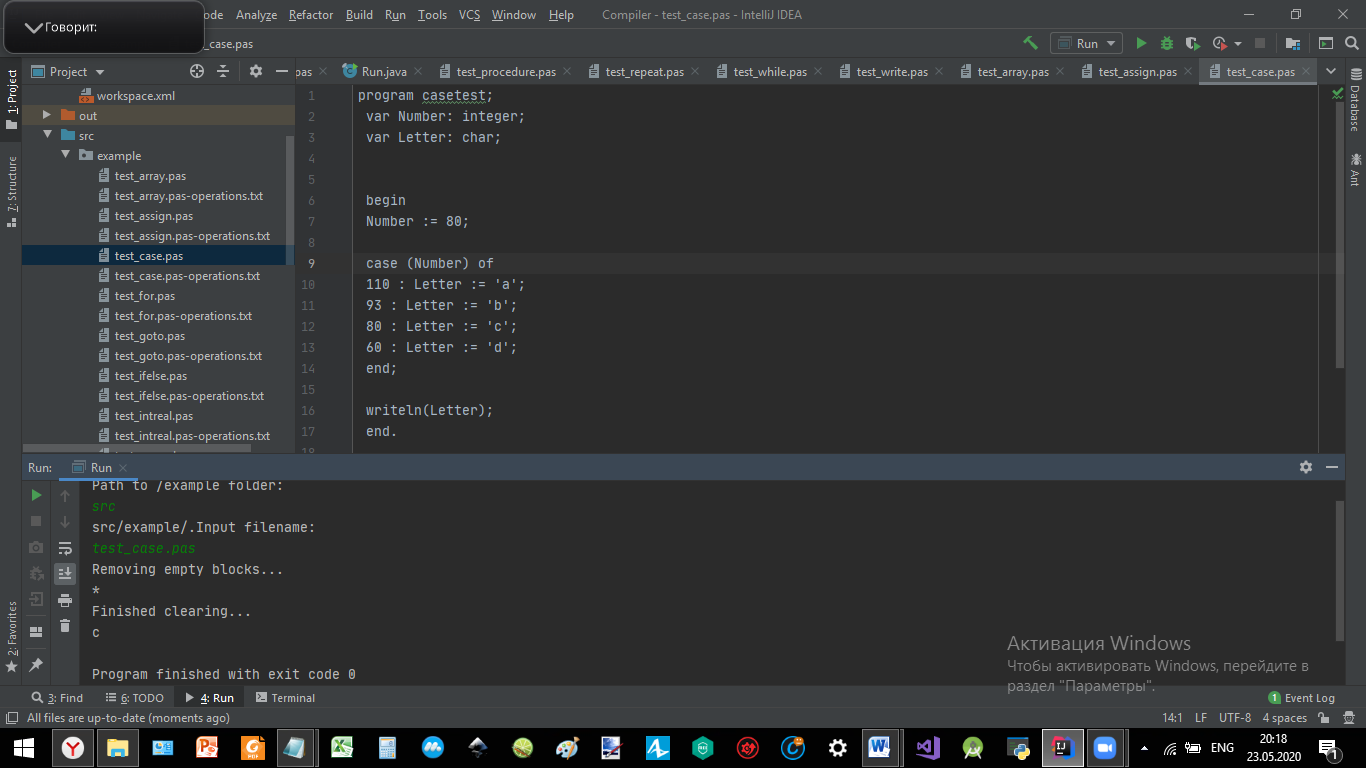
# 3. Тесты



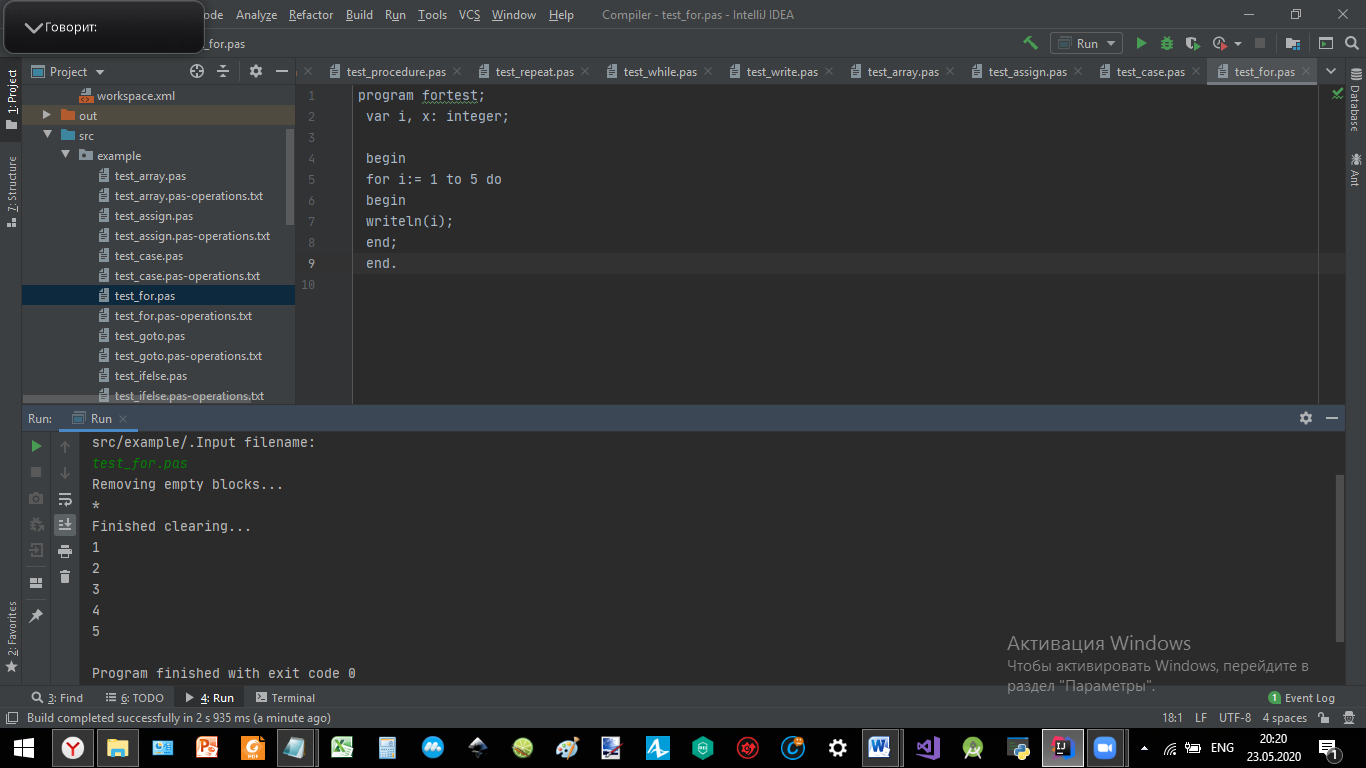
**Рисунок 2. Тест массивов**



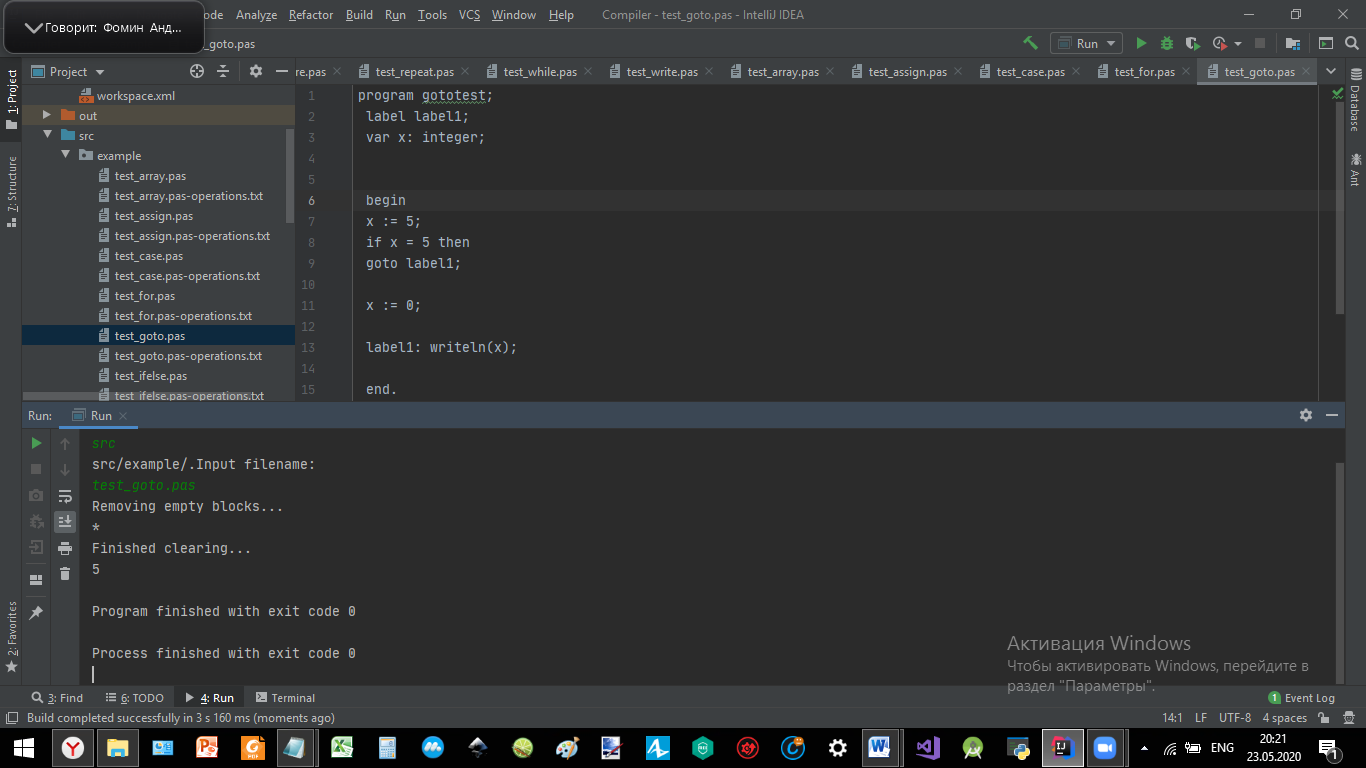
**Рисунок 3. Тест присваивания**



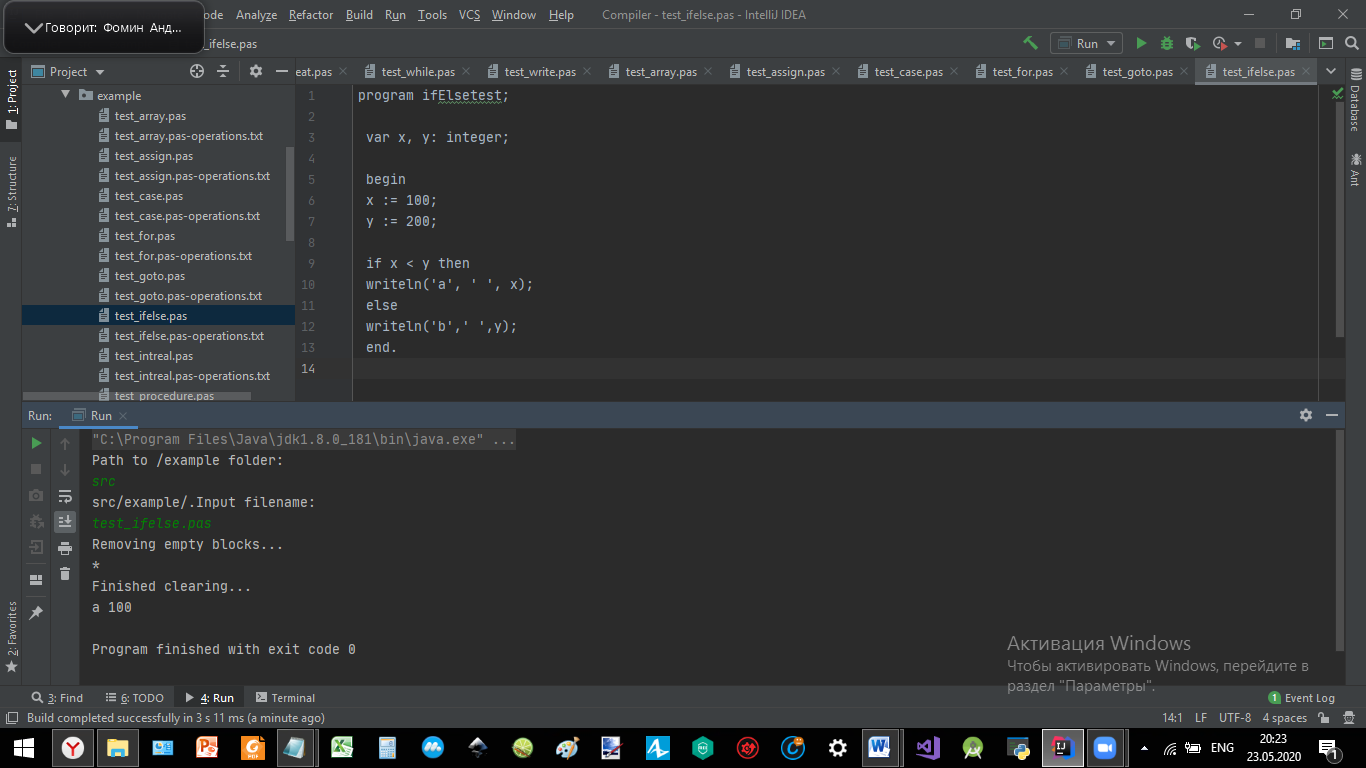
**Рисунок 4. Тест case**



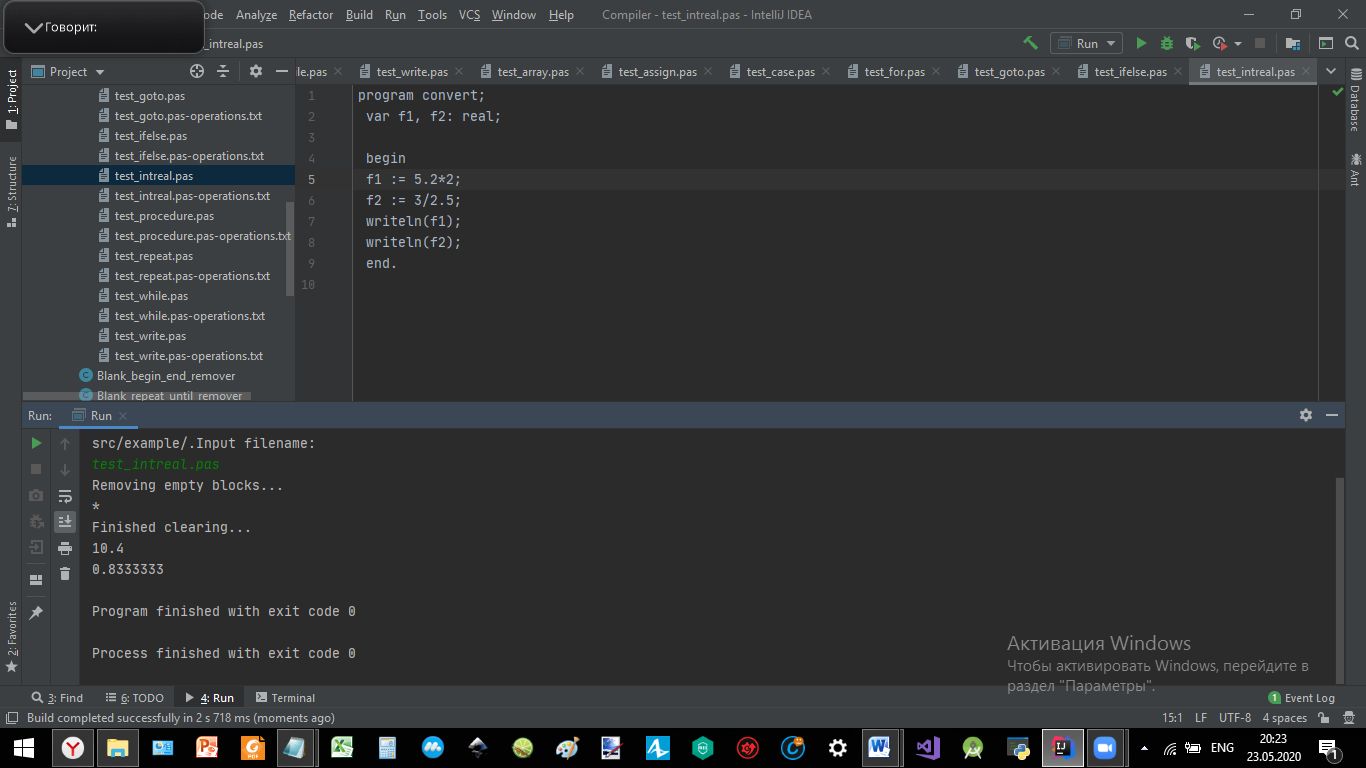
**Рисунок 5. Тест for**



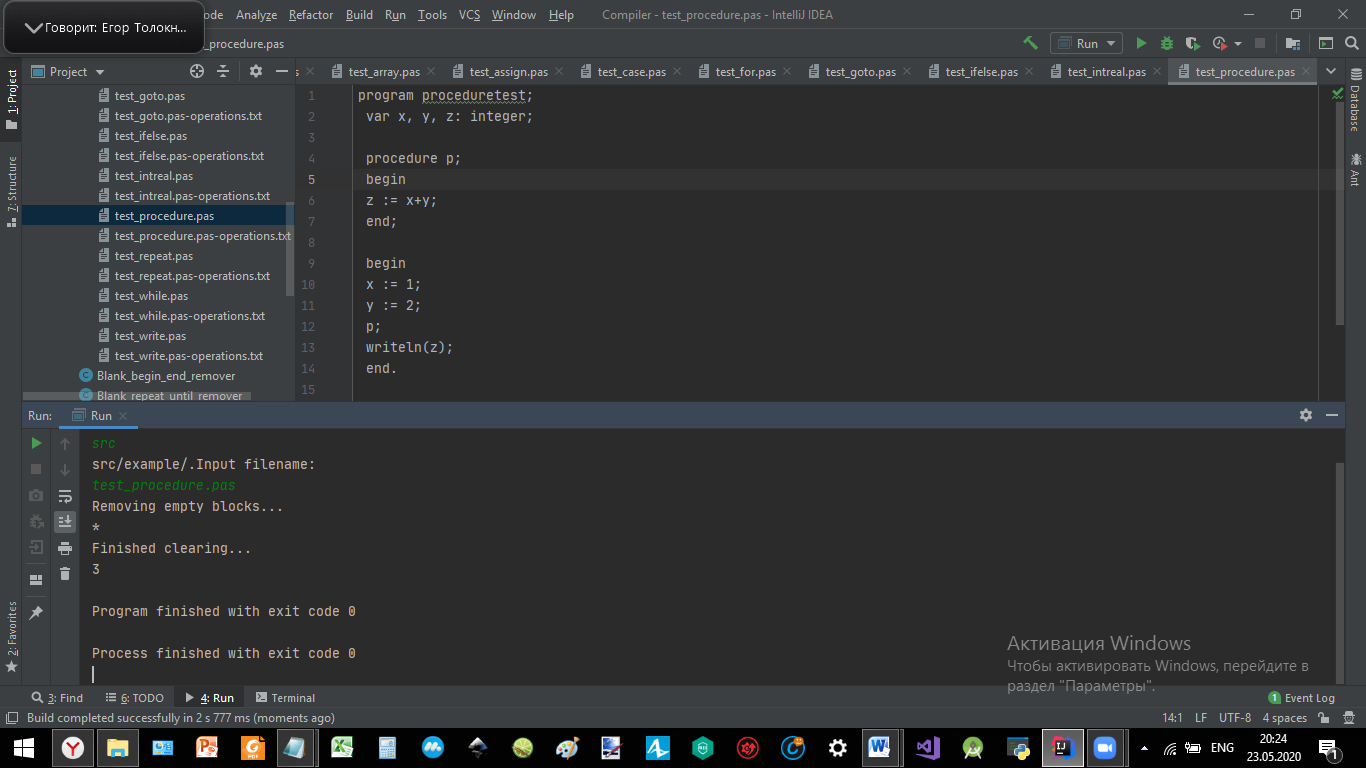
**Рисунок 6. Тест goto**



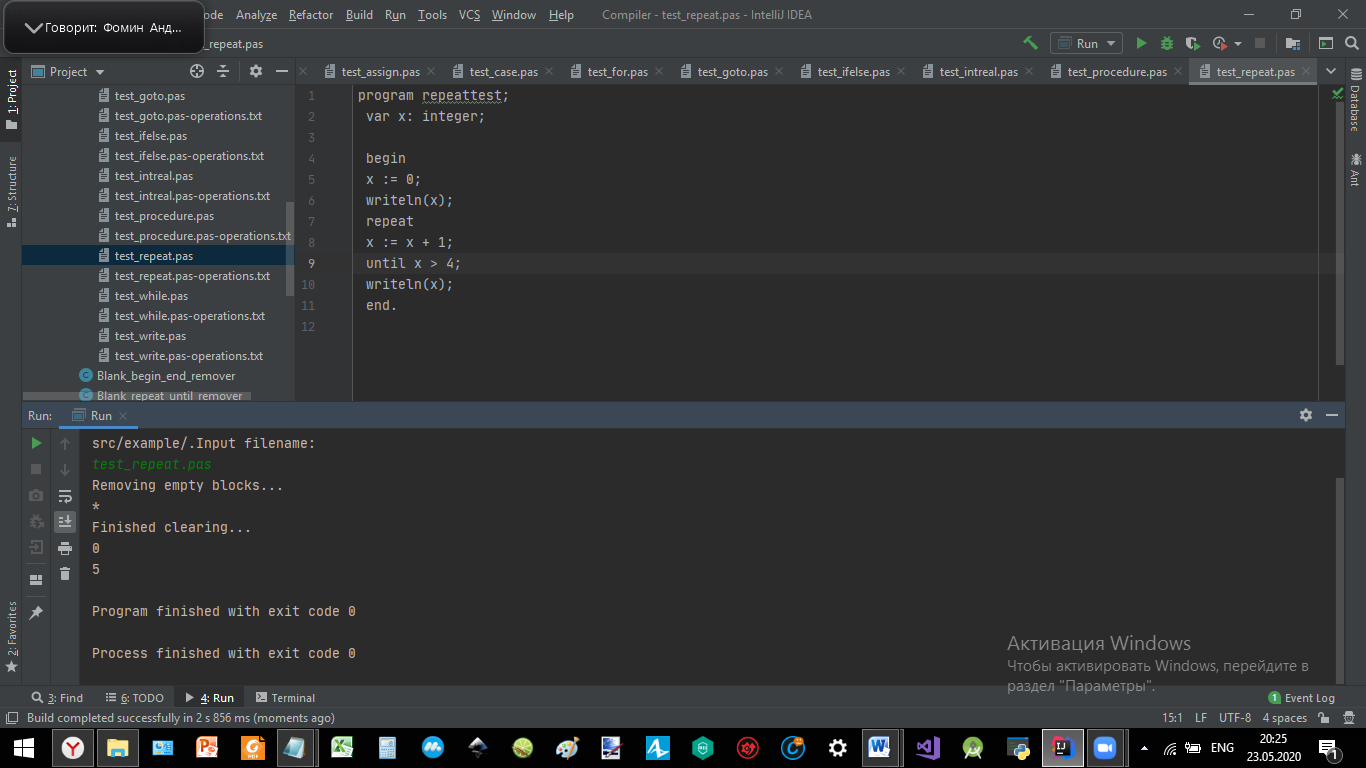
**Рисунок 7. Тест if-else**



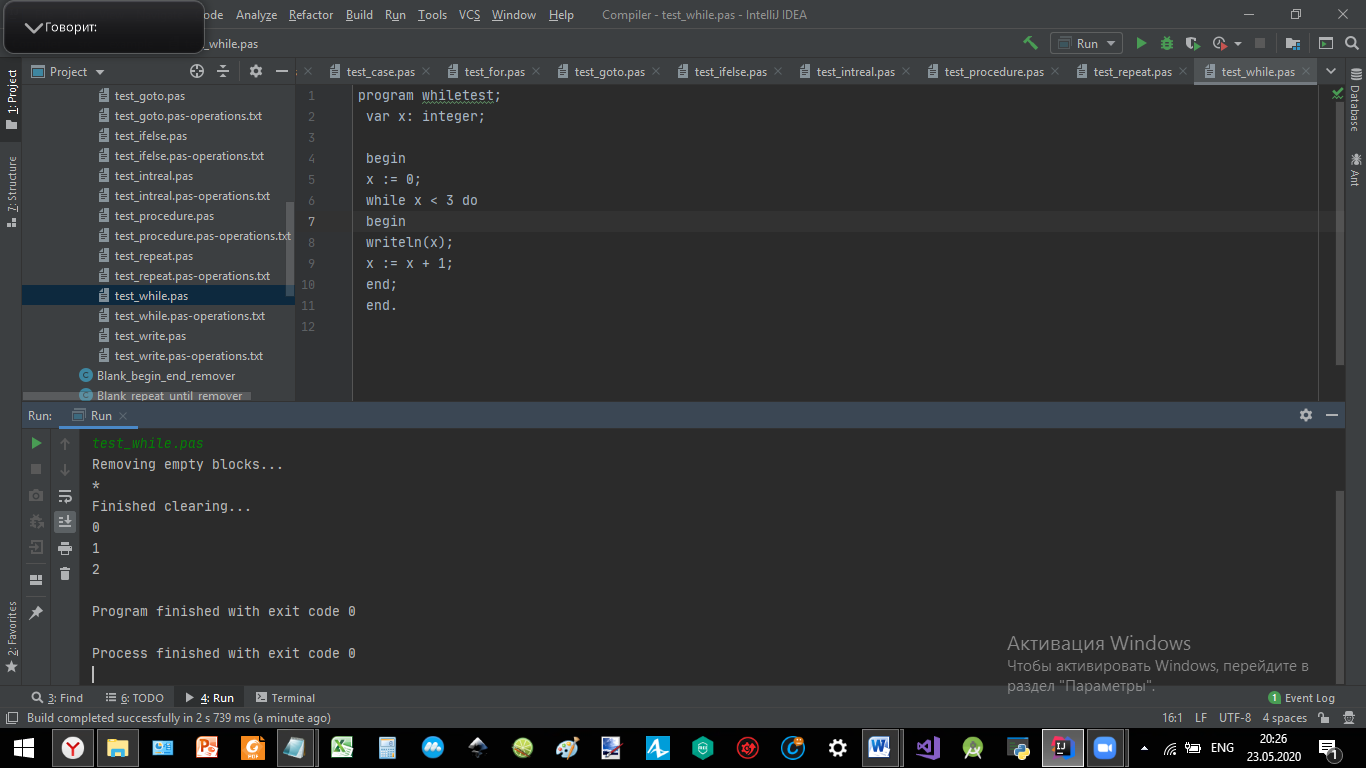
**Рисунок 8. Тест intreal**



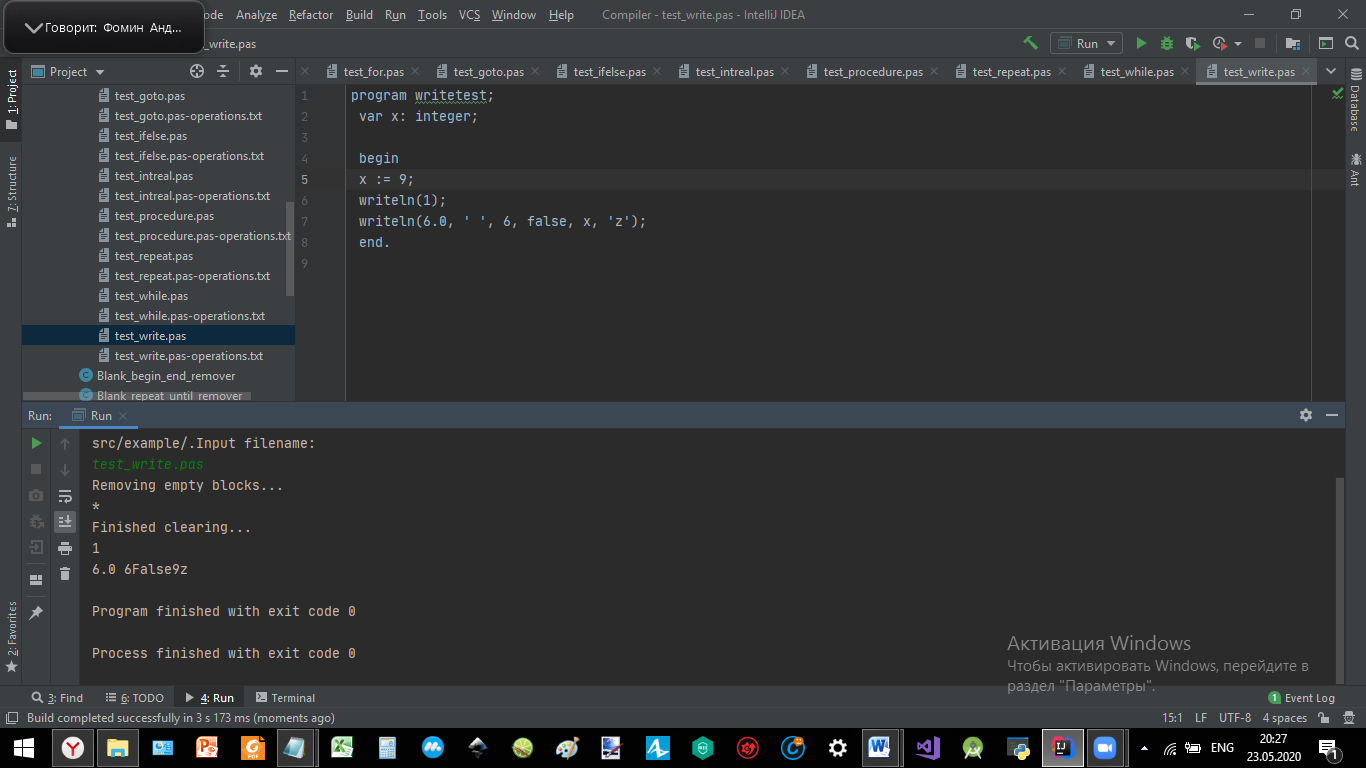
**Рисунок 9. Тест процедуры**



**Рисунок 10. Тест repeat**



**Рисунок 11. Тест while**



**Рисунок 12. Тест write**

# ЗАКЛЮЧЕНИЕ

В данной программе был реализован компилятор подмножества языка Pascal с минимальным набором функционала для выполнения базовых задач, стоящих перед реализацией компилятора.

# ПРИЛОЖЕНИЕ

# Run.java

import java.io.\*;  
import java.nio.file.Files;  
import java.nio.file.Paths;  
import java.util.\*;  
  
public final class Run {  
 public static String *filepath* ;  
 public static String *filename* ;  
 public static void main(String[] args) throws IOException {  
  
 String str;  
  
 Scanner finput = new Scanner(System.*in*);  
 System.*out*.println("Path to /example folder:");  
  
 *filepath* = finput.nextLine();  
  
  
 System.*out*.println(*filepath*+"/example/"+".Input filename:");  
 *filename* = finput.nextLine();  
 finput.close();  
String finalpath = *filepath*+"/example/"+*filename*;  
 File file = new File(finalpath);  
  
 Scanner in = new Scanner(file);  
  
 str = Comments\_remover.*remover*(in);  
 in.close();  
  
 Files.*write*(Paths.*get*(finalpath), str.getBytes());  
  
 new StringBuilder();  
 StringBuilder code\_optimizer\_str;  
 Scanner in2 = new Scanner(file);  
 code\_optimizer\_str = new StringBuilder();  
 while (in2.hasNext())  
 code\_optimizer\_str.append(in2.nextLine()).append("\n ");  
 in2.close();  
  
  
 int levellength = 0;  
  
 System.*out*.println("Removing empty blocks...");  
 while (code\_optimizer\_str.length() != levellength) {  
  
  
 System.*out*.print("\*");  
  
  
 str = Blank\_begin\_end\_remover.*remover*(code\_optimizer\_str.toString());  
 code\_optimizer\_str = new StringBuilder(str);  
  
  
 levellength = code\_optimizer\_str.length();  
  
  
 str = Blank\_repeat\_until\_remover.*remover*(code\_optimizer\_str.toString());  
 code\_optimizer\_str = new StringBuilder(str);  
  
  
 }  
 System.*out*.println("\nFinished clearing...");  
 Files.*write*(Paths.*get*(finalpath), code\_optimizer\_str.toString().getBytes());  
  
  
 ArrayList<Token> Tokens\_list = Token\_reader.*read*(new File(finalpath));  
  
  
 Parser.*set\_Tokens\_list\_Iterator*(Tokens\_list);  
  
 Byte[] instructions = Parser.*parse*();  
 Operations\_simulator.*set\_Instructions*(instructions);  
  
  
 Operations\_simulator.*simulate*();  
  
  
 }  
  
  
}

# Token.java

/\*  
Например  
KW~FLOATLIT: 0  
 token\_Type: "KW~FLOATLIT"  
 token\_Value: "0"  
 \*/  
  
public final class Token {  
 private String token\_Type;  
 private final String token\_Value;  
  
 public Token(String token\_Type, String token\_Value){  
 this.token\_Type = token\_Type;  
 this.token\_Value = token\_Value;  
  
 }  
  
 @Override  
 public String toString(){  
 return token\_Value;  
 }  
 public String get\_Token\_Type() {  
 return token\_Type;  
 }  
 public void set\_Token\_Type(String tokenType) {  
 this.token\_Type = tokenType;  
 }  
 public String get\_Token\_Val() {  
 return token\_Value;  
 }  
  
  
}

# Token\_reader.java

import java.io.File;  
import java.io.FileNotFoundException;  
import java.util.ArrayList;  
import java.util.HashMap;  
import java.util.Scanner;  
  
public final class Token\_reader {  
 private static final HashMap<String, String> *KEYWORDS\_TOKEN*;  
 private static final HashMap<String, String> *OPERATORS\_TOKEN*; //Токены операторов  
 private static final HashMap<String, TYPE> *CHAR\_TYPE*;  
 private static String *token\_Name* = "";  
 private static boolean *String\_isRead* = false;  
 private static boolean *Number\_isRead* = false;  
 private static boolean *isFloat* = false;  
 private static boolean *sci\_Notation* = false;  
 private static boolean *Colon\_isRead* = false;  
 private static boolean *Bool\_isRead* = false;  
 private static boolean *Dot\_isRead* = false;  
 private static final ArrayList<Token> *Array\_List\_tokens* = new ArrayList<>();  
  
 static {  
 *KEYWORDS\_TOKEN* = new HashMap<>();  
 String word;  
 String filename;  
  
  
 filename = Run.*filepath*+"/keywords.txt";  
  
 try {  
  
 Scanner sc = new Scanner(new File(filename));  
 while(sc.hasNext()){  
 word = sc.next();  
 *KEYWORDS\_TOKEN*.put(word, String.*format*("KW~%s", word.toUpperCase()));  
 }  
 } catch (FileNotFoundException e) {  
 e.printStackTrace();  
 }  
 }  
  
 static {  
 *OPERATORS\_TOKEN* = new HashMap<>();  
 *OPERATORS\_TOKEN*.put("(", "KW~OPEN\_PARENTHESIS");  
 *OPERATORS\_TOKEN*.put(")", "KW~CLOSE\_PARENTHESIS");  
 *OPERATORS\_TOKEN*.put("[", "KW~OPEN\_SQUARE\_BRACKET");  
 *OPERATORS\_TOKEN*.put("]", "KW~CLOSE\_SQUARE\_BRACKET");  
 *OPERATORS\_TOKEN*.put(".", "KW~DOT");  
 *OPERATORS\_TOKEN*.put("..", "KW~RANGE");  
 *OPERATORS\_TOKEN*.put(":", "KW~COLON");  
 *OPERATORS\_TOKEN*.put(";", "KW~SEMI\_COLON");  
 *OPERATORS\_TOKEN*.put("+", "KW~PLUS");  
 *OPERATORS\_TOKEN*.put("-", "KW~MINUS");  
 *OPERATORS\_TOKEN*.put("\*", "KW~MULTIPLY");  
 *OPERATORS\_TOKEN*.put("/", "KW~DIVIDE");  
 *OPERATORS\_TOKEN*.put("<", "KW~LESS\_THAN");  
 *OPERATORS\_TOKEN*.put("<=", "KW~LESS\_THAN\_EQUAL");  
 *OPERATORS\_TOKEN*.put(">", "KW~GREATER\_THAN");  
 *OPERATORS\_TOKEN*.put(">=", "KW~GREATER\_THAN\_EQUAL");  
 *OPERATORS\_TOKEN*.put(":=", "KW~ASSIGNMENT");  
 *OPERATORS\_TOKEN*.put(",", "KW~COMMA");  
 *OPERATORS\_TOKEN*.put("=", "KW~EQUAL");  
 *OPERATORS\_TOKEN*.put("<>", "KW~NOT\_EQUAL");  
  
 }  
  
 static {  
 *CHAR\_TYPE* = new HashMap<>();  
  
 for (int i = 65; i < 91; i++){  
 // Добавление строковых символов  
 String Char\_current = String.*valueOf*(Character.*toChars*(i)[0]);  
 *CHAR\_TYPE*.put(Char\_current, TYPE.*LETTER*);  
 *CHAR\_TYPE*.put(Char\_current.toLowerCase(), TYPE.*LETTER*);  
 }  
 for (int i = 48; i < 58; i++){  
 // Добавление цифр  
 String Char\_current = String.*valueOf*(Character.*toChars*(i)[0]);  
 *CHAR\_TYPE*.put(Char\_current, TYPE.*DIGIT*);  
 }  
 for (int i = 1; i < 33; i++){  
 // Добавление пробелов  
 String Char\_current = String.*valueOf*(Character.*toChars*(i)[0]);  
 *CHAR\_TYPE*.put(Char\_current, TYPE.*SPACE*);  
 }  
  
 for (String key: *OPERATORS\_TOKEN*.keySet()) {  
 *CHAR\_TYPE*.put(key, TYPE.*OPERATOR*);  
 }  
  
 // Одиночная кавычка  
 *CHAR\_TYPE*.put(String.*valueOf*(Character.*toChars*(39)[0]), TYPE.*QUOTE*);  
 }  
  
 public static ArrayList<Token> read(File file) throws FileNotFoundException {  
 // Разделитель для чтения символов  
 Scanner sc = new Scanner(file).useDelimiter("");  
  
 while (sc.hasNext()) {  
 char element = sc.next().toLowerCase().charAt(0);  
  
 *Char\_type\_checker*(element);  
 }  
  
 *token\_Name* = "EOF";  
 *Token\_generate*("KW~EOF");  
  
 return *Array\_List\_tokens*;  
 }  
  
 public static void Char\_type\_checker(char element){ //Проверка типа символа - это может строковый литерал, цифра, пробелы, арифметический оператор или кавычка  
 switch (*CHAR\_TYPE*.get(String.*valueOf*(element))){  
  
 case *LETTER*:  
 if (!*Number\_isRead*) {  
 *token\_Name* += element;  
 }  
  
 if (element == 'E' && *Number\_isRead*) {  
 *token\_Name* += element;  
 *sci\_Notation* = true;  
 }  
  
 break;  
 case *DIGIT*:  
 if (*token\_Name*.isEmpty()) {  
 *Number\_isRead* = true;  
 }  
  
 *token\_Name* += element;  
  
 break;  
 case *SPACE*:  
 if (*String\_isRead*){  
 // Добавить в строку  
 *token\_Name* += element;  
 } else if (*Colon\_isRead*) {  
  
 *Token\_generate*(*OPERATORS\_TOKEN*.get(*token\_Name*));  
  
 *Colon\_isRead* = false;  
  
 } else if (*Bool\_isRead*) {  
  
 *Token\_generate*(*OPERATORS\_TOKEN*.get(*token\_Name*));  
  
 *Bool\_isRead* = false;  
  
 } else if (!*Number\_isRead*) {  
 // Конец слова  
 *token\_Name* = *Word\_end*();  
  
 // Проверка на новую строку  
 } else {  
 *Number\_handle*();  
 }  
 break;  
 case *OPERATOR*:  
 if (*Dot\_isRead* && element == '.') {  
 if (*token\_Name*.equals(".")) {  
 *token\_Name* = "";  
 *Token\_generate*("KW~RANGE");  
 } else {  
 *Token\_generate*(*token\_Name*.substring(0, *token\_Name*.length()-2));  
 *Token\_generate*("KW~DOT");  
 *token\_Name* = "";  
 }  
 *Dot\_isRead* = false;  
  
 } else if(*String\_isRead*) {  
 // Добавление в строку  
 *token\_Name* += element;  
 } else if (*Number\_isRead*) {  
 if (*isFloat* && element == '.') {  
 *isFloat* = false;  
 *token\_Name* = *token\_Name*.substring(0, *token\_Name*.length()-1);  
 *Number\_handle*();  
  
  
 *Token\_generate*("KW~RANGE");  
 *token\_Name* = "";  
  
 } else if (*sci\_Notation* && (element == '+' || element == '-')) {  
 *token\_Name* += element;  
 } else if (element == '.') {  
 // Десятичная часть в числе  
 *isFloat* = true;  
 *token\_Name* += element;  
 } else {  
 *Number\_handle*();  
  
  
 *Token\_generate*(*OPERATORS\_TOKEN*.get(String.*valueOf*(element)));  
 }  
 } else if (*Colon\_isRead* && element == '=') {  
  
 // Обработка назначения  
 *token\_Name* += element;  
  
 *Token\_generate*(*OPERATORS\_TOKEN*.get(*token\_Name*));  
  
  
 *Colon\_isRead* = false;  
 } else if (*Bool\_isRead*) {  
 if (*token\_Name*.equals("<") && ((element == '=') || (element == '>'))) {  
 *token\_Name* += element;  
  
 *Token\_generate*(*OPERATORS\_TOKEN*.get(*token\_Name*));  
 } else if (*token\_Name*.equals(">") && (element == '=')) {  
 *token\_Name* += element;  
  
 *Token\_generate*(*OPERATORS\_TOKEN*.get(*token\_Name*));  
 }  
  
 *Bool\_isRead* = false;  
 } else {  
 if (element == ';') {  
 // Перед тем как строка закончится  
 *token\_Name* = *Word\_end*();  
  
 *token\_Name* = ";";  
 *Token\_generate*(*OPERATORS\_TOKEN*.get(String.*valueOf*(element)));  
  
 } else if (element == ':') {  
 *token\_Name* = *Word\_end*();  
 *Colon\_isRead* = true;  
 *token\_Name* += element;  
 } else if (element == '<' || element == '>') {  
 *token\_Name* = *Word\_end*();  
 *Bool\_isRead* = true;  
 *token\_Name* += element;  
 } else if (element == '.') {  
 *token\_Name* += element;  
  
 if (*token\_Name*.equals("end.")){  
 *Token\_generate*("KW~END");  
 *Token\_generate*("KW~DOT");  
 } else {  
 *Dot\_isRead* = true;  
 }  
 } else if (*OPERATORS\_TOKEN*.containsKey(String.*valueOf*(element))) {  
 *token\_Name* = *Word\_end*();  
  
  
 *token\_Name* = String.*valueOf*(element);  
 *Token\_generate*(*OPERATORS\_TOKEN*.get(*token\_Name*));  
 }  
 }  
 break;  
 case *QUOTE*:  
 // Найдена кавычка открывающая / закрывающая  
 *String\_isRead* = !*String\_isRead*;  
 *token\_Name* += element;  
  
 if (!*String\_isRead*) {  
 // Удаление хвостовых кавычек  
 *token\_Name* = *token\_Name*.substring(1, *token\_Name*.length()-1);  
  
 // Найдена закрывающая  
 if (*token\_Name*.length() == 1) {  
  
 *Token\_generate*("KW~CHARLIT");  
 }  
 }  
 break;  
 default:  
 throw new Error("Unhandled element scanned");  
 }  
 }  
  
 public static String Word\_end(){ //проверяет конец слова и возвращает имя токена  
 if(*KEYWORDS\_TOKEN*.containsKey(*token\_Name*)){  
  
 *Token\_generate*(*KEYWORDS\_TOKEN*.get(*token\_Name*));  
 } else {  
 if (*token\_Name*.length() > 0) {  
  
 if (*token\_Name*.equals("true") || *token\_Name*.equals("false")) {  
  
 *Token\_generate*("KW~BOOLLIT");  
 } else {  
  
 *Token\_generate*("KW~IDENTIFIER");  
 }  
 }  
 }  
  
 *clearStatuses*();  
  
 return *token\_Name*;  
 }  
  
 public static void clearStatuses() { //сброс статусов читаемого слова  
 *String\_isRead* = false;  
 *Number\_isRead* = false;  
 *isFloat* = false;  
 *sci\_Notation* = false;  
 *Colon\_isRead* = false;  
 *Bool\_isRead* = false;  
 }  
  
 public static void Token\_generate(String tokenType) {  
 Token t = new Token(tokenType, *token\_Name*);  
 *Array\_List\_tokens*.add(t); // Токен добавляется в список  
  
 *token\_Name* = "";  
 }  
  
 public static void Number\_handle() { //Целочисленный тип или вещественный  
 *Number\_isRead* = false;  
 if (*isFloat*) {  
  
 *Token\_generate*("KW~FLOATLIT");  
 *isFloat* = false;  
 } else {  
  
 *Token\_generate*("KW~INTLIT");  
 }  
 }  
  
 enum TYPE {  
 *LETTER*, *DIGIT*, *SPACE*, *OPERATOR*, *QUOTE* }  
}

# Parser.java

import java.nio.ByteBuffer;  
import java.util.ArrayList;  
import java.util.HashMap;  
import java.util.Iterator;  
  
public final class Parser {  
 enum TYPE {  
 *INT*, *REAL*, *BOOL*, *CHAR*, *PROCEDURE*, *LABEL*, *ARRAY* // integer, real, boolean, char, procedure, label, array  
 }  
  
 private static int *data\_pointer* = 0; // Указатель на область памяти для переменных  
  
 private static final HashMap<String, TYPE> *HASH\_MAP\_types*;  
  
 static {  
 *HASH\_MAP\_types* = new HashMap<>();  
 *HASH\_MAP\_types*.put("integer", TYPE.*INT*);  
 *HASH\_MAP\_types*.put("real", TYPE.*REAL*);  
 *HASH\_MAP\_types*.put("boolean", TYPE.*BOOL*);  
 *HASH\_MAP\_types*.put("char", TYPE.*CHAR*);  
 *HASH\_MAP\_types*.put("array", TYPE.*ARRAY*);  
  
 }  
  
 enum Operations\_code {  
 *PUSHI*, *PUSH*, *POP*, *PUSHF*,  
 *JMP*, *JFALSE*, *JTRUE*,  
 *CVR*, *CVI*,  
 *DUP*, *XCHG*, *REMOVE*,  
 *ADD*, *SUB*, *MULT*, *DIV*, *MOD* , *NEG*,  
 *OR*, *AND*,  
 *FADD*, *FSUB*, *FMULT*, *FDIV*, *FNEG*,  
 *EQL*, *NEQL*, *GEQ*, *LEQ*, *GTR*, *LSS*,  
 *FGTR*, *FLSS*,  
 *HALT*,  
 *PRINT\_INT*, *PRINT\_CHAR*, *PRINT\_BOOL*, *PRINT\_REAL*, *PRINT\_NEWLINE*,  
 *GET*, *PUT* }  
  
 private static final int *ADDRESS\_SIZE* = 4; // Сколько байт выделяется под одну переменную  
  
 private static Token *Token\_current*;  
 private static Iterator<Token> *Token\_iterator*;  
  
 private static final int *INSTRUCTION\_SIZE* = 1000;  
  
 private static final Byte[] *byteArray* = new Byte[*INSTRUCTION\_SIZE*];  
 private static int *ip* = 0;  
  
 public static Byte[] parse() {  
 *getToken*(); // Получить изначальный токен  
  
 *tokens\_match*("KW~PROGRAM");  
 *tokens\_match*("KW~IDENTIFIER");  
 *tokens\_match*("KW~SEMI\_COLON"); //Должно соответствовать шаблону - Program <Имя> ;  
  
 *program*();  
  
 return *byteArray*;  
 }  
  
  
 public static void program() {  
 *declarations*();  
 *begin*();  
 }  
  
  
 public static void declarations() { //Различные объявления - переменные, процедуры, метки, начало блока  
 while (true) {  
 switch (*Token\_current*.get\_Token\_Type()) {  
 case "KW~VAR":  
 *var\_init*();  
 break;  
 case "KW~PROCEDURE":  
 *procedure\_init*();  
 break;  
 case "KW~LABEL":  
 *label\_init*();  
 break;  
 case "KW~BEGIN":  
 return;  
 }  
 }  
 }  
  
  
 private static void label\_init() {  
 while (true) {  
 if ("KW~LABEL".equals(*Token\_current*.get\_Token\_Type())) {  
 *tokens\_match*("KW~LABEL");  
 } else {  
  
 break;  
 }  
  
 // Хранение меток (label) в спике  
 ArrayList<Token> Array\_labels = new ArrayList<>();  
  
 while ("KW~IDENTIFIER".equals(*Token\_current*.get\_Token\_Type())) { //Перечисление объявляемых переменных типа label  
 *Token\_current*.set\_Token\_Type("KW~A\_LABEL");  
 Array\_labels.add(*Token\_current*);  
  
 *tokens\_match*("KW~A\_LABEL");  
  
 if ("KW~COMMA".equals(*Token\_current*.get\_Token\_Type())) {  
 *tokens\_match*("KW~COMMA"); // перечисление через запятую  
 }  
 }  
  
 // Вставка всех меток в таблицу символов  
 for (Token label : Array\_labels) {  
  
  
 Symbol symbol = new Symbol(label.get\_Token\_Val(),  
 "KW~A\_LABEL",  
 TYPE.*LABEL*,  
 0);  
  
 if (Table\_symbols.*find\_and\_get*(label.get\_Token\_Val()) == null) {  
 Table\_symbols.*insert*(symbol);  
 }  
 }  
  
 *tokens\_match*("KW~SEMI\_COLON"); // Объявление заканчивается ;  
 }  
 }  
  
  
 private static void procedure\_init() {  
  
 if (*Token\_current*.get\_Token\_Type().equals("KW~PROCEDURE")) {  
 *tokens\_match*("KW~PROCEDURE");  
 *Token\_current*.set\_Token\_Type("KW~A\_PROC");  
  
 String procedure\_Name = *Token\_current*.get\_Token\_Val();  
  
 *tokens\_match*("KW~A\_PROC");  
 *tokens\_match*("KW~SEMI\_COLON"); //Procedure <имя>;  
  
 // Генерация пустого пространства для перемещения через тело программы  
 *generate\_Operation\_Code*(Operations\_code.*JMP*);  
 int body\_jump\_hole = *ip*;  
 *generate\_Address*(0);  
  
 Symbol symbol = new Symbol(procedure\_Name,  
 "KW~A\_PROC",  
 TYPE.*PROCEDURE*,  
 *ip*);  
  
 // тело программы  
 *tokens\_match*("KW~BEGIN");  
 *statements*();  
 *tokens\_match*("KW~END");  
 *tokens\_match*("KW~SEMI\_COLON");  
  
 // пустое пространство для возвращения процедуры  
 *generate\_Operation\_Code*(Operations\_code.*JMP*);  
 symbol.setAddress\_return(*ip*);  
 *generate\_Address*(0);  
  
 if (Table\_symbols.*find\_and\_get*(procedure\_Name) == null) {  
 Table\_symbols.*insert*(symbol);  
 }  
  
 // заполнение пустого пространства для перемещения через тело программы  
 int save = *ip*;  
  
 *ip* = body\_jump\_hole;  
 *generate\_Address*(save);  
 *ip* = save;  
 }  
 }  
  
  
 public static void var\_init() {  
 while (true) {  
 if ("KW~VAR".equals(*Token\_current*.get\_Token\_Type())) {  
 *tokens\_match*("KW~VAR");  
 } else {  
  
 break;  
 }  
  
 // Хранение переменных в списке  
 ArrayList<Token> var\_Array = new ArrayList<>();  
  
 while ("KW~IDENTIFIER".equals(*Token\_current*.get\_Token\_Type())) {  
 *Token\_current*.set\_Token\_Type("KW~A\_VAR");  
 var\_Array.add(*Token\_current*);  
  
 *tokens\_match*("KW~A\_VAR");  
  
 if ("KW~COMMA".equals(*Token\_current*.get\_Token\_Type())) {  
 *tokens\_match*("KW~COMMA");  
 }  
 }  
  
 *tokens\_match*("KW~COLON");  
 String Type\_data = *Token\_current*.get\_Token\_Type();  
 *tokens\_match*(Type\_data);  
  
 // Добавление соответствующего типа данных для каждого идентификатора и вставка его в таблицу символов  
 for (Token var : var\_Array) {  
  
 Symbol symbol = new Symbol(var.get\_Token\_Val(),  
 "KW~A\_VAR",  
 *HASH\_MAP\_types*.get(Type\_data.toLowerCase().substring(3)),  
 *data\_pointer*);  
  
 *data\_pointer* += 4;  
  
  
 if (Table\_symbols.*find\_and\_get*(var.get\_Token\_Val()) == null) {  
 Table\_symbols.*insert*(symbol);  
 }  
 }  
  
 if (Type\_data.equals("KW~ARRAY")) {  
 *array\_init*(var\_Array);  
 }  
  
 *tokens\_match*("KW~SEMI\_COLON");  
  
 }  
 }  
  
  
 private static void array\_init(ArrayList<Token> var\_Array) {  
 *tokens\_match*("KW~OPEN\_SQUARE\_BRACKET");  
 String v1 = *Token\_current*.get\_Token\_Val();  
 TYPE index\_type1 = *get\_Litera\_Type*(*Token\_current*.get\_Token\_Type());  
 *tokens\_match*(*Token\_current*.get\_Token\_Type());  
  
 *tokens\_match*("KW~RANGE");  
  
 String v2 = *Token\_current*.get\_Token\_Val();  
 TYPE index\_type2 = *get\_Litera\_Type*(*Token\_current*.get\_Token\_Type());  
 *tokens\_match*(*Token\_current*.get\_Token\_Type());  
 *tokens\_match*("KW~CLOSE\_SQUARE\_BRACKET");  
 *tokens\_match*("KW~OF");  
  
 String val\_Type = *Token\_current*.get\_Token\_Type();  
 *tokens\_match*(val\_Type);  
  
 if (index\_type1 != index\_type2) {  
 throw new Error(String.*format*("Array index LHS type (%s) is not equal to RHS type: (%s)", index\_type1, index\_type2));  
 } else {  
  
 assert index\_type1 != null;  
 switch (index\_type1) {  
 case *INT*:  
 int i1 = Integer.*parseInt*(v1);  
 int i2 = Integer.*parseInt*(v2);  
 if (i1 > i2) {  
 throw new Error(String.*format*("Array range is invalid: %d..%d", i1, i2));  
 }  
  
 Symbol first\_Integer\_Array = Table\_symbols.*find\_and\_get*(var\_Array.get(0).get\_Token\_Val());  
 if (first\_Integer\_Array != null) {  
 *data\_pointer* = first\_Integer\_Array.getAddress();  
 }  
  
 for (Token var : var\_Array) {  
 Symbol symbol = Table\_symbols.*find\_and\_get*(var.get\_Token\_Val());  
 if (symbol != null) {  
  
 int elementSize = 4;  
 int size = elementSize \* (i2 - i1 + 1);  
  
 symbol.set\_Address(*data\_pointer*);  
 symbol.set\_Low(i1);  
 symbol.set\_High(i2);  
 symbol.set\_Token\_Type("KW~AN\_ARRAY");  
 symbol.set\_Index\_Type(TYPE.*INT*);  
 symbol.set\_Value\_Type(*HASH\_MAP\_types*.get(val\_Type.toLowerCase().substring(3)));  
  
 *data\_pointer* += size;  
 }  
 }  
  
 break;  
 case *CHAR*:  
 char c1 = v1.toCharArray()[0];  
 char c2 = v2.toCharArray()[0];  
 if (c1 > c2) {  
 throw new Error(String.*format*("Array range is invalid: %c..%c", c1, c2));  
 }  
  
 Symbol first\_Char\_Array = Table\_symbols.*find\_and\_get*(var\_Array.get(0).get\_Token\_Val());  
 if (first\_Char\_Array != null) {  
 *data\_pointer* = first\_Char\_Array.getAddress();  
 }  
  
 for (Token var : var\_Array) {  
 Symbol symbol = Table\_symbols.*find\_and\_get*(var.get\_Token\_Val());  
 if (symbol != null) {  
 int size = c2 - c1 + 1;  
  
 symbol.set\_Address(*data\_pointer*);  
 symbol.set\_Low(c1);  
 symbol.set\_High(c2);  
 symbol.set\_Token\_Type("KW~AN\_ARRAY");  
 symbol.set\_Index\_Type(TYPE.*CHAR*);  
 symbol.set\_Value\_Type(*HASH\_MAP\_types*.get(val\_Type.toLowerCase().substring(3)));  
  
 *data\_pointer* += size;  
 }  
 }  
  
 break;  
 case *REAL*:  
 throw new Error("Array index type: real is invalid");  
 }  
  
 }  
  
 }  
  
  
 public static void begin() {  
 *tokens\_match*("KW~BEGIN");  
 *statements*();  
 *tokens\_match*("KW~END");  
 *tokens\_match*("KW~DOT");  
 *tokens\_match*("KW~EOF");  
 *generate\_Operation\_Code*(Operations\_code.*HALT*);  
 }  
  
  
 public static void statements() {  
 while (!*Token\_current*.get\_Token\_Type().equals("KW~END")) {  
 switch (*Token\_current*.get\_Token\_Type()) {  
 case "KW~CASE":  
 *Statement\_case*();  
 break;  
 case "KW~GOTO":  
 *Statement\_goto*();  
 break;  
 case "KW~WHILE":  
 *Statement\_while*();  
 break;  
 case "KW~REPEAT":  
 *Statement\_repeat*();  
 break;  
 case "KW~IF":  
 *Statement\_if*();  
 break;  
 case "KW~FOR":  
 *Statement\_for*();  
 break;  
 case "KW~WRITELN":  
 *Statement\_writeln*();  
 break;  
  
 case "KW~IDENTIFIER":  
 Symbol symbol = Table\_symbols.*find\_and\_get*(*Token\_current*.get\_Token\_Val());  
 if (symbol != null) {  
 // Назначение типа токена как переменная процедура или метка  
 *Token\_current*.set\_Token\_Type(symbol.getToken\_Type());  
 }  
 break;  
 case "KW~A\_VAR":  
 *Statement\_assign*();  
 break;  
 case "KW~A\_PROC":  
 *Statement\_procedure*();  
 break;  
 case "KW~A\_LABEL":  
 *Statement\_label*();  
 break;  
 case "KW~AN\_ARRAY":  
 *Statement\_arrayAssignment*();  
 break;  
 case "KW~SEMI\_COLON":  
 *tokens\_match*("KW~SEMI\_COLON");  
 break;  
 default:  
 return;  
 }  
 }  
  
 }  
  
 private static void Statement\_label() {  
 Symbol symbol = Table\_symbols.*find\_and\_get*(*Token\_current*.get\_Token\_Val());  
 *tokens\_match*("KW~A\_LABEL");  
 *tokens\_match*("KW~COLON");  
 if (symbol != null) {  
 int hole = symbol.getAddress();  
 int save = *ip*;  
  
 // Заполнение пустого пространства под goto  
 *ip* = hole;  
 *generate\_Address*(save);  
  
 *ip* = save;  
  
 *statements*();  
 }  
 }  
  
 private static void Statement\_procedure() {  
 Symbol symbol = Table\_symbols.*find\_and\_get*(*Token\_current*.get\_Token\_Val());  
 if (symbol != null) {  
 int address = symbol.getAddress();  
 *tokens\_match*("KW~A\_PROC");  
 *tokens\_match*("KW~SEMI\_COLON");  
 // Вызов процедуры  
 *generate\_Operation\_Code*(Operations\_code.*JMP*);  
 *generate\_Address*(address);  
  
 int restore = *ip*;  
  
 // Заполнение пустого пространства под возврат и восстановление указателя  
 *ip* = symbol.getAddress\_return();  
 *generate\_Address*(restore);  
 *ip* = restore;  
 }  
 }  
  
 private static void Statement\_goto() {  
 *tokens\_match*("KW~GOTO");  
 Symbol symbol = Table\_symbols.*find\_and\_get*(*Token\_current*.get\_Token\_Val());  
 *Token\_current*.set\_Token\_Type("KW~A\_LABEL");  
 *tokens\_match*("KW~A\_LABEL");  
 *generate\_Operation\_Code*(Operations\_code.*JMP*);  
 int hole = *ip*;  
 *generate\_Address*(0);  
  
 // Пустое пространство для перемещения  
 if (symbol != null) {  
 symbol.set\_Address(hole);  
 }  
  
 *tokens\_match*("KW~SEMI\_COLON");  
  
 }  
  
  
 // for <имя переменной> := <начало> to <конец> do <действия>  
 private static void Statement\_for() {  
 *tokens\_match*("KW~FOR");  
  
 String varName = *Token\_current*.get\_Token\_Val();  
 *Token\_current*.set\_Token\_Type("KW~A\_VAR");  
 *Statement\_assign*();  
  
 int target = *ip*;  
  
  
 Symbol symbol = Table\_symbols.*find\_and\_get*(varName);  
 if (symbol != null) {  
 int address = symbol.getAddress();  
 *tokens\_match*("KW~TO");  
  
 // Генерация кодов операций для x <= <верхняя граница>  
 *generate\_Operation\_Code*(Operations\_code.*PUSH*);  
 *generate\_Address*(address);  
 *generate\_Operation\_Code*(Operations\_code.*PUSHI*);  
 *generate\_Address*(Integer.*parseInt*(*Token\_current*.get\_Token\_Val()));  
  
 *generate\_Operation\_Code*(Operations\_code.*LEQ*);  
 *tokens\_match*("KW~INTLIT");  
  
 *tokens\_match*("KW~DO");  
  
 *generate\_Operation\_Code*(Operations\_code.*JFALSE*);  
 int hole = *ip*;  
 *generate\_Address*(0);  
  
 *tokens\_match*("KW~BEGIN");  
 *statements*();  
 *tokens\_match*("KW~END");  
 *tokens\_match*("KW~SEMI\_COLON");  
  
 // Генерация кодов операций для увеличения x в цикле  
 *generate\_Operation\_Code*(Operations\_code.*PUSH*);  
 *generate\_Address*(address);  
 *generate\_Operation\_Code*(Operations\_code.*PUSHI*);  
 *generate\_Address*(1);  
 *generate\_Operation\_Code*(Operations\_code.*ADD*);  
  
 *generate\_Operation\_Code*(Operations\_code.*POP*);  
 *generate\_Address*(address);  
  
  
 *generate\_Operation\_Code*(Operations\_code.*JMP*);  
 *generate\_Address*(target);  
  
 int save = *ip*;  
 *ip* = hole;  
 *generate\_Address*(save);  
 *ip* = save;  
 }  
 }  
  
 // repeat <действия> until <условие>  
 private static void Statement\_repeat() {  
 *tokens\_match*("KW~REPEAT");  
 int target = *ip*;  
 *statements*();  
 *tokens\_match*("KW~UNTIL");  
 *Condition\_type*();  
 *generate\_Operation\_Code*(Operations\_code.*JFALSE*);  
 *generate\_Address*(target);  
 }  
  
  
 // while <условие> do <действия>  
 private static void Statement\_while() {  
 *tokens\_match*("KW~WHILE");  
 int target = *ip*;  
 *Condition\_type*();  
 *tokens\_match*("KW~DO");  
  
 *generate\_Operation\_Code*(Operations\_code.*JFALSE*);  
 int hole = *ip*;  
 *generate\_Address*(0);  
  
 *tokens\_match*("KW~BEGIN");  
 *statements*();  
 *tokens\_match*("KW~END");  
 *tokens\_match*("KW~SEMI\_COLON");  
  
  
 *generate\_Operation\_Code*(Operations\_code.*JMP*);  
 *generate\_Address*(target);  
  
 int save = *ip*;  
 *ip* = hole;  
 *generate\_Address*(save);  
 *ip* = save;  
  
 }  
  
 // if <условие> then <действия>  
 // if <условие> then <действия> else <действия в ином случае>  
 public static void Statement\_if() {  
 *tokens\_match*("KW~IF");  
 *Condition\_type*();  
 *tokens\_match*("KW~THEN");  
 *generate\_Operation\_Code*(Operations\_code.*JFALSE*);  
 int hole1 = *ip*;  
 *generate\_Address*(0);  
 *statements*();  
  
 if (*Token\_current*.get\_Token\_Type().equals("KW~ELSE")) {  
 *generate\_Operation\_Code*(Operations\_code.*JMP*);  
 int hole2 = *ip*;  
 *generate\_Address*(0);  
 int save = *ip*;  
 *ip* = hole1;  
 *generate\_Address*(save);  
 *ip* = save;  
 hole1 = hole2;  
 *statements*();  
 *tokens\_match*("KW~ELSE");  
 *statements*();  
 }  
  
 int save = *ip*;  
 *ip* = hole1;  
 *generate\_Address*(save);  
 *ip* = save;  
 }  
  
 //case <выражение с некоторым значением> of (как правило набор из нескольких пар) <значение> : <действие>  
 public static void Statement\_case() {  
 *tokens\_match*("KW~CASE");  
 *tokens\_match*("KW~OPEN\_PARENTHESIS");  
 Token eToken = *Token\_current*;  
  
 TYPE t1 = *Expression\_type*();  
  
 if (t1 == TYPE.*REAL*) {  
 throw new Error("Invalid type of real for case E");  
 }  
  
 *tokens\_match*("KW~CLOSE\_PARENTHESIS");  
 *tokens\_match*("KW~OF");  
  
 ArrayList<Integer> labelsArrayList = new ArrayList<>();  
  
 while (*Token\_current*.get\_Token\_Type().equals("KW~INTLIT") ||  
 *Token\_current*.get\_Token\_Type().equals("KW~CHARLIT") ||  
 *Token\_current*.get\_Token\_Type().equals("KW~BOOLLIT")) {  
  
 TYPE t2 = *Expression\_type*();  
 *operation\_selector*("KW~EQUAL", t1, t2);  
 *tokens\_match*("KW~COLON");  
  
 // Пустое пространство выделяется под JFALSE к следующей case когда операция сравнения (eql) возвращает false  
 *generate\_Operation\_Code*(Operations\_code.*JFALSE*);  
 int hole = *ip*;  
 *generate\_Address*(0);  
 *statements*();  
  
 *generate\_Operation\_Code*(Operations\_code.*JMP*);  
 labelsArrayList.add(*ip*);  
 *generate\_Address*(0);  
  
 // Заполнение пустого пространства JFALSE  
 int save = *ip*;  
 *ip* = hole;  
 *generate\_Address*(save);  
  
 *ip* = save;  
  
 // Вставка в стек токена для подготовки к eql в следующей case  
  
 if (!*Token\_current*.get\_Token\_Val().equals("KW~END")) {  
 Symbol symbol = Table\_symbols.*find\_and\_get*(eToken.get\_Token\_Val());  
 if (symbol != null) {  
 *generate\_Operation\_Code*(Operations\_code.*PUSH*);  
 *generate\_Address*(symbol.getAddress());  
 }  
 }  
 }  
  
 *tokens\_match*("KW~END");  
 *tokens\_match*("KW~SEMI\_COLON");  
  
 int save = *ip*;  
  
 // Заполнение всех пустых пространств выделенных под ссылки для JMP  
 for (Integer labelHole : labelsArrayList) {  
 *ip* = labelHole;  
 *generate\_Address*(save);  
 }  
  
 *ip* = save;  
 }  
  
 //Вывод на экран  
 public static void Statement\_writeln() {  
 *tokens\_match*("KW~WRITELN");  
 *tokens\_match*("KW~OPEN\_PARENTHESIS");  
  
 while (true) {  
 Symbol symbol = Table\_symbols.*find\_and\_get*(*Token\_current*.get\_Token\_Val());  
 TYPE t;  
  
 if (symbol != null) {  
 if (symbol.getData\_Type() == TYPE.*ARRAY*) {  
  
 *Token\_current*.set\_Token\_Type("KW~AN\_ARRAY");  
 *access\_array\_type*(symbol);  
  
 *generate\_Operation\_Code*(Operations\_code.*GET*);  
  
 t = symbol.getValue\_Type();  
  
 } else {  
  
 *Token\_current*.set\_Token\_Type("KW~A\_VAR");  
  
 t = symbol.getData\_Type();  
 *generate\_Operation\_Code*(Operations\_code.*PUSH*);  
 *generate\_Address*(symbol.getAddress());  
 *tokens\_match*("KW~A\_VAR");  
 }  
 } else {  
  
 t = *get\_Litera\_Type*(*Token\_current*.get\_Token\_Type());  
 assert t != null;  
 switch (t) {  
 case *REAL*:  
 *generate\_Operation\_Code*(Operations\_code.*PUSHF*);  
 *generate\_Address*(Float.*parseFloat*(*Token\_current*.get\_Token\_Val()));  
 break;  
 case *INT*:  
 *generate\_Operation\_Code*(Operations\_code.*PUSHI*);  
 *generate\_Address*(Integer.*parseInt*(*Token\_current*.get\_Token\_Val()));  
 break;  
 case *BOOL*:  
 *generate\_Operation\_Code*(Operations\_code.*PUSHI*);  
 if (*Token\_current*.get\_Token\_Val().equals("true")) {  
 *generate\_Address*(1);  
 } else {  
 *generate\_Address*(0);  
 }  
 break;  
 case *CHAR*:  
 *generate\_Operation\_Code*(Operations\_code.*PUSHI*);  
 *generate\_Address*(*Token\_current*.get\_Token\_Val().charAt(0));  
 break;  
 }  
  
 *tokens\_match*(*Token\_current*.get\_Token\_Type());  
 }  
  
 assert t != null;  
 switch (t) {  
 case *INT*:  
 *generate\_Operation\_Code*(Operations\_code.*PRINT\_INT*);  
 break;  
 case *CHAR*:  
 *generate\_Operation\_Code*(Operations\_code.*PRINT\_CHAR*);  
 break;  
 case *REAL*:  
 *generate\_Operation\_Code*(Operations\_code.*PRINT\_REAL*);  
 break;  
 case *BOOL*:  
 *generate\_Operation\_Code*(Operations\_code.*PRINT\_BOOL*);  
 break;  
  
  
  
 default:  
 throw new Error("Cannot write unknown type");  
  
 }  
  
 switch (*Token\_current*.get\_Token\_Type()) {  
 case "KW~COMMA":  
 *tokens\_match*("KW~COMMA");  
 break;  
 case "KW~CLOSE\_PARENTHESIS":  
 *tokens\_match*("KW~CLOSE\_PARENTHESIS");  
 *generate\_Operation\_Code*(Operations\_code.*PRINT\_NEWLINE*);  
 return;  
 default:  
 throw new Error(String.*format*("Current token type (%s) is neither KW~COMMA nor KW~CLOSE\_PARENTHESIS", *Token\_current*.get\_Token\_Type()));  
 }  
  
 }  
 }  
  
 public static void Statement\_assign() {  
 Symbol symbol = Table\_symbols.*find\_and\_get*(*Token\_current*.get\_Token\_Val());  
  
 if (symbol != null) {  
 TYPE lhsType = symbol.getData\_Type();  
 int lhsAddress = symbol.getAddress();  
  
 *tokens\_match*("KW~A\_VAR");  
  
 *tokens\_match*("KW~ASSIGNMENT");  
  
 TYPE rhsType = *Expression\_type*();  
 if (lhsType == rhsType) {  
 *generate\_Operation\_Code*(Operations\_code.*POP*);  
 *generate\_Address*(lhsAddress);  
 } else {  
 throw new Error(String.*format*("LHS type (%s) is not equal to RHS type: (%s)", lhsType, rhsType));  
 }  
 }  
  
  
 }  
  
  
 private static void Statement\_arrayAssignment() {  
 Symbol symbol = Table\_symbols.*find\_and\_get*(*Token\_current*.get\_Token\_Val());  
 if (symbol != null) {  
  
 *access\_array\_type*(symbol);  
  
 *tokens\_match*("KW~ASSIGNMENT");  
  
  
 TYPE rhsType = *Expression\_type*();  
  
 if (symbol.getValue\_Type() == rhsType) {  
 *generate\_Operation\_Code*(Operations\_code.*PUT*);  
 }  
  
 }  
  
 }  
  
 private static void access\_array\_type(Symbol symbol) {  
 *tokens\_match*("KW~AN\_ARRAY");  
 *tokens\_match*("KW~OPEN\_SQUARE\_BRACKET");  
 TYPE t;  
  
  
 Symbol varSymbol = Table\_symbols.*find\_and\_get*(*Token\_current*.get\_Token\_Val());  
 if (varSymbol != null) {  
 t = varSymbol.getData\_Type();  
  
  
 if (t != symbol.getIndex\_Type()) {  
 throw new Error(String.*format*("Incompatible index type: (%s, %s)", t, symbol.getIndex\_Type()));  
 }  
  
 *Token\_current*.set\_Token\_Type("KW~A\_VAR");  
 *generate\_Operation\_Code*(Operations\_code.*PUSH*);  
 *generate\_Address*(varSymbol.getAddress());  
 *tokens\_match*("KW~A\_VAR");  
  
 *tokens\_match*("KW~CLOSE\_SQUARE\_BRACKET");  
  
 *generate\_Operation\_Code*(Operations\_code.*PUSHI*);  
  
 switch (t) {  
 case *INT*:  
  
 int i1 = (int) symbol.get\_Low();  
  
 *generate\_Address*(i1);  
 *generate\_Operation\_Code*(Operations\_code.*XCHG*);  
 *generate\_Operation\_Code*(Operations\_code.*SUB*);  
  
 // push element size  
 *generate\_Operation\_Code*(Operations\_code.*PUSHI*);  
 *generate\_Address*(4);  
  
 *generate\_Operation\_Code*(Operations\_code.*MULT*);  
  
 *generate\_Operation\_Code*(Operations\_code.*PUSHI*);  
 *generate\_Address*(symbol.getAddress());  
  
 *generate\_Operation\_Code*(Operations\_code.*ADD*);  
  
 break;  
 case *CHAR*:  
 char c1 = (char) symbol.get\_Low();  
  
 *generate\_Address*(c1);  
 *generate\_Operation\_Code*(Operations\_code.*XCHG*);  
 *generate\_Operation\_Code*(Operations\_code.*SUB*);  
  
 *generate\_Operation\_Code*(Operations\_code.*PUSHI*);  
 *generate\_Address*(symbol.getAddress());  
  
 *generate\_Operation\_Code*(Operations\_code.*ADD*);  
  
 break;  
 }  
 } else {  
  
  
 String index = *Token\_current*.get\_Token\_Val();  
 t = *Expression\_type*();  
  
 if (t != symbol.getIndex\_Type()) {  
 throw new Error(String.*format*("Incompatible index type: (%s, %s)", t, symbol.getIndex\_Type()));  
 }  
  
 *tokens\_match*("KW~CLOSE\_SQUARE\_BRACKET");  
  
 *generate\_Operation\_Code*(Operations\_code.*PUSHI*);  
  
 switch (t) {  
 case *INT*:  
  
 int i1 = (int) symbol.get\_Low();  
 int i2 = (int) symbol.get\_High();  
  
  
 if (Integer.*parseInt*(index) < i1 || Integer.*parseInt*(index) > i2) {  
 throw new Error(String.*format*("Index %d is not within range %d to %d",  
 Integer.*valueOf*(index), i1, i2));  
 }  
  
 *generate\_Address*(i1);  
 *generate\_Operation\_Code*(Operations\_code.*XCHG*);  
 *generate\_Operation\_Code*(Operations\_code.*SUB*);  
  
  
 *generate\_Operation\_Code*(Operations\_code.*PUSHI*);  
 *generate\_Address*(4);  
  
 *generate\_Operation\_Code*(Operations\_code.*MULT*);  
  
 *generate\_Operation\_Code*(Operations\_code.*PUSHI*);  
 *generate\_Address*(symbol.getAddress());  
  
 *generate\_Operation\_Code*(Operations\_code.*ADD*);  
  
 break;  
 case *CHAR*:  
 char c1 = (char) symbol.get\_Low();  
 char c2 = (char) symbol.get\_High();  
  
  
 if (index.toCharArray()[0] < c1 || index.toCharArray()[0] > c2) {  
 throw new Error(String.*format*("Index %c is not within range %c to %c",  
 index.toCharArray()[0], c1, c2));  
 }  
  
 *generate\_Address*(c1);  
 *generate\_Operation\_Code*(Operations\_code.*XCHG*);  
 *generate\_Operation\_Code*(Operations\_code.*SUB*);  
  
 *generate\_Operation\_Code*(Operations\_code.*PUSHI*);  
 *generate\_Address*(symbol.getAddress());  
  
 *generate\_Operation\_Code*(Operations\_code.*ADD*);  
  
 break;  
 }  
  
 }  
 }  
  
  
 public static void Condition\_type() {  
 TYPE expr1 = *Expression\_type*();  
 while (*Token\_current*.get\_Token\_Type().equals("KW~LESS\_THAN") ||  
 *Token\_current*.get\_Token\_Type().equals("KW~GREATER\_THAN") ||  
 *Token\_current*.get\_Token\_Type().equals("KW~LESS\_THAN\_EQUAL") ||  
 *Token\_current*.get\_Token\_Type().equals("KW~GREATER\_THAN\_EQUAL") ||  
 *Token\_current*.get\_Token\_Type().equals("KW~EQUAL") ||  
 *Token\_current*.get\_Token\_Type().equals("KW~NOT\_EQUAL")) {  
 String pred = *Token\_current*.get\_Token\_Type();  
 *tokens\_match*(pred);  
 TYPE expr2 = *Terminal\_type*();  
  
 expr1 = *operation\_selector*(pred, expr1, expr2);  
 }  
  
 }  
  
  
 public static TYPE Expression\_type() {  
 TYPE term1 = *Terminal\_type*();  
 while (*Token\_current*.get\_Token\_Type().equals("KW~PLUS") || *Token\_current*.get\_Token\_Type().equals("KW~MINUS")) {  
 String operator\_tok = *Token\_current*.get\_Token\_Type();  
 *tokens\_match*(operator\_tok);  
 TYPE term2 = *Terminal\_type*();  
  
 term1 = *operation\_selector*(operator\_tok, term1, term2);  
 }  
  
 return term1;  
 }  
  
  
 public static TYPE Terminal\_type() {  
 TYPE factor1 = *Factor\_type*();  
 while (*Token\_current*.get\_Token\_Type().equals("KW~MULTIPLY") ||  
 *Token\_current*.get\_Token\_Type().equals("KW~DIVIDE") ||  
 *Token\_current*.get\_Token\_Type().equals("KW~DIV")||  
 *Token\_current*.get\_Token\_Type().equals("KW~MOD")) {  
 String operator\_tok = *Token\_current*.get\_Token\_Type();  
 *tokens\_match*(operator\_tok);  
 TYPE factor2 = *Factor\_type*();  
  
 factor1 = *operation\_selector*(operator\_tok, factor1, factor2);  
 }  
 return factor1;  
 }  
  
  
 public static TYPE Factor\_type() {  
 switch (*Token\_current*.get\_Token\_Type()) {  
 case "KW~IDENTIFIER":  
 Symbol symbol = Table\_symbols.*find\_and\_get*(*Token\_current*.get\_Token\_Val());  
 if (symbol != null) {  
 if (symbol.getToken\_Type().equals("KW~A\_VAR")) {  
 // variable  
 *Token\_current*.set\_Token\_Type("KW~A\_VAR");  
  
 *generate\_Operation\_Code*(Operations\_code.*PUSH*);  
 *generate\_Address*(symbol.getAddress());  
  
 *tokens\_match*("KW~A\_VAR");  
 return symbol.getData\_Type();  
 } else if (symbol.getToken\_Type().equals("KW~AN\_ARRAY")) {  
 *Token\_current*.set\_Token\_Type("KW~AN\_ARRAY");  
  
 *access\_array\_type*(symbol);  
 *generate\_Operation\_Code*(Operations\_code.*GET*);  
  
 return symbol.getValue\_Type();  
 }  
 } else {  
 throw new Error(String.*format*("Symbol not found (%s)", *Token\_current*.get\_Token\_Val()));  
 }  
 case "KW~INTLIT":  
 *generate\_Operation\_Code*(Operations\_code.*PUSHI*);  
 *generate\_Address*(Integer.*parseInt*(*Token\_current*.get\_Token\_Val()));  
  
 *tokens\_match*("KW~INTLIT");  
 return TYPE.*INT*;  
 case "KW~FLOATLIT":  
 *generate\_Operation\_Code*(Operations\_code.*PUSHF*);  
 *generate\_Address*(Float.*parseFloat*(*Token\_current*.get\_Token\_Val()));  
  
 *tokens\_match*("KW~FLOATLIT");  
 return TYPE.*REAL*;  
 case "KW~BOOLLIT":  
 *generate\_Operation\_Code*(Operations\_code.*PUSHI*);  
 *generate\_Address*(Boolean.*parseBoolean*(*Token\_current*.get\_Token\_Val()) ? 1 : 0);  
  
 *tokens\_match*("KW~BOOLLIT");  
 return TYPE.*BOOL*;  
 case "KW~CHARLIT":  
 *generate\_Operation\_Code*(Operations\_code.*PUSHI*);  
 *generate\_Address*(*Token\_current*.get\_Token\_Val().charAt(0));  
  
 *tokens\_match*("KW~CHARLIT");  
 return TYPE.*CHAR*;  
  
 case "KW~NOT":  
 *tokens\_match*("KW~NOT");  
 return *Factor\_type*();  
 case "KW~OPEN\_PARENTHESIS":  
 *tokens\_match*("KW~OPEN\_PARENTHESIS");  
 TYPE type\_ = *Expression\_type*();  
 *tokens\_match*("KW~CLOSE\_PARENTHESIS");  
 return type\_;  
 default:  
 throw new Error("Unknown data type");  
 }  
  
 }  
  
  
 public static TYPE operation\_selector(String op, TYPE t1, TYPE t2) {  
 switch (op) {  
 case "KW~PLUS":  
 if (t1 == TYPE.*INT* && t2 == TYPE.*INT*) {  
 *generate\_Operation\_Code*(Operations\_code.*ADD*);  
 return TYPE.*INT*;  
 } else if (t1 == TYPE.*INT* && t2 == TYPE.*REAL*) {  
 *generate\_Operation\_Code*(Operations\_code.*XCHG*);  
 *generate\_Operation\_Code*(Operations\_code.*CVR*);  
 *generate\_Operation\_Code*(Operations\_code.*FADD*);  
 return TYPE.*REAL*;  
 } else if (t1 == TYPE.*REAL* && t2 == TYPE.*INT*) {  
 *generate\_Operation\_Code*(Operations\_code.*CVR*);  
 *generate\_Operation\_Code*(Operations\_code.*FADD*);  
 return TYPE.*REAL*;  
 } else if (t1 == TYPE.*REAL* && t2 == TYPE.*REAL*) {  
 *generate\_Operation\_Code*(Operations\_code.*FADD*);  
 return TYPE.*REAL*;  
 }  
 case "KW~MINUS":  
 if (t1 == TYPE.*INT* && t2 == TYPE.*INT*) {  
 *generate\_Operation\_Code*(Operations\_code.*SUB*);  
 return TYPE.*INT*;  
 } else if (t1 == TYPE.*INT* && t2 == TYPE.*REAL*) {  
 *generate\_Operation\_Code*(Operations\_code.*XCHG*);  
 *generate\_Operation\_Code*(Operations\_code.*CVR*);  
 *generate\_Operation\_Code*(Operations\_code.*FSUB*);  
 return TYPE.*REAL*;  
 } else if (t1 == TYPE.*REAL* && t2 == TYPE.*INT*) {  
 *generate\_Operation\_Code*(Operations\_code.*CVR*);  
 *generate\_Operation\_Code*(Operations\_code.*FSUB*);  
 return TYPE.*REAL*;  
 } else if (t1 == TYPE.*REAL* && t2 == TYPE.*REAL*) {  
 *generate\_Operation\_Code*(Operations\_code.*FSUB*);  
 return TYPE.*REAL*;  
 }  
 case "KW~MULTIPLY":  
 if (t1 == TYPE.*INT* && t2 == TYPE.*INT*) {  
 *generate\_Operation\_Code*(Operations\_code.*MULT*);  
 return TYPE.*INT*;  
 } else if (t1 == TYPE.*INT* && t2 == TYPE.*REAL*) {  
 *generate\_Operation\_Code*(Operations\_code.*XCHG*);  
 *generate\_Operation\_Code*(Operations\_code.*CVR*);  
 *generate\_Operation\_Code*(Operations\_code.*FMULT*);  
 return TYPE.*REAL*;  
 } else if (t1 == TYPE.*REAL* && t2 == TYPE.*INT*) {  
 *generate\_Operation\_Code*(Operations\_code.*CVR*);  
 *generate\_Operation\_Code*(Operations\_code.*FMULT*);  
 return TYPE.*REAL*;  
 } else if (t1 == TYPE.*REAL* && t2 == TYPE.*REAL*) {  
 *generate\_Operation\_Code*(Operations\_code.*FMULT*);  
 return TYPE.*REAL*;  
 }  
 case "KW~DIVIDE":  
 if (t1 == TYPE.*INT* && t2 == TYPE.*INT*) {  
 *generate\_Operation\_Code*(Operations\_code.*CVR*);  
 *generate\_Operation\_Code*(Operations\_code.*XCHG*);  
 *generate\_Operation\_Code*(Operations\_code.*CVR*);  
 *generate\_Operation\_Code*(Operations\_code.*XCHG*);  
 *generate\_Operation\_Code*(Operations\_code.*FDIV*);  
 return TYPE.*REAL*;  
 } else if (t1 == TYPE.*INT* && t2 == TYPE.*REAL*) {  
 *generate\_Operation\_Code*(Operations\_code.*XCHG*);  
 *generate\_Operation\_Code*(Operations\_code.*CVR*);  
 *generate\_Operation\_Code*(Operations\_code.*FDIV*);  
 return TYPE.*REAL*;  
 } else if (t1 == TYPE.*REAL* && t2 == TYPE.*INT*) {  
 *generate\_Operation\_Code*(Operations\_code.*CVR*);  
 *generate\_Operation\_Code*(Operations\_code.*FDIV*);  
 return TYPE.*REAL*;  
 } else if (t1 == TYPE.*REAL* && t2 == TYPE.*REAL*) {  
 *generate\_Operation\_Code*(Operations\_code.*FDIV*);  
 return TYPE.*REAL*;  
 }  
 case "KW~DIV":  
 if (t1 == TYPE.*INT* && t2 == TYPE.*INT*) {  
 *generate\_Operation\_Code*(Operations\_code.*DIV*);  
 return TYPE.*INT*;  
 }  
 case "KW~MOD":  
 if (t1 == TYPE.*INT* && t2 == TYPE.*INT*) {  
 *generate\_Operation\_Code*(Operations\_code.*MOD*);  
 return TYPE.*INT*;  
 }  
 case "KW~LESS\_THAN":  
 return *Boolean\_selector*(Operations\_code.*LSS*, t1, t2);  
 case "KW~GREATER\_THAN":  
 return *Boolean\_selector*(Operations\_code.*GTR*, t1, t2);  
 case "KW~LESS\_THAN\_EQUAL":  
 return *Boolean\_selector*(Operations\_code.*LEQ*, t1, t2);  
 case "KW~GREATER\_THAN\_EQUAL":  
 return *Boolean\_selector*(Operations\_code.*GEQ*, t1, t2);  
 case "KW~EQUAL":  
 return *Boolean\_selector*(Operations\_code.*EQL*, t1, t2);  
 case "KW~NOT\_EQUAL":  
 return *Boolean\_selector*(Operations\_code.*NEQL*, t1, t2);  
 }  
  
 return null;  
 }  
  
 public static TYPE Boolean\_selector(Operations\_code pred, TYPE t1, TYPE t2) {  
 if (t1 == t2) {  
 *generate\_Operation\_Code*(pred);  
 return TYPE.*BOOL*;  
 } else if (t1 == TYPE.*INT* && t2 == TYPE.*REAL*) {  
 *generate\_Operation\_Code*(Operations\_code.*XCHG*);  
 *generate\_Operation\_Code*(Operations\_code.*CVR*);  
 *generate\_Operation\_Code*(pred);  
 return TYPE.*BOOL*;  
 } else if (t1 == TYPE.*REAL* && t2 == TYPE.*INT*) {  
 *generate\_Operation\_Code*(Operations\_code.*CVR*);  
 *generate\_Operation\_Code*(pred);  
 return TYPE.*BOOL*;  
 }  
  
 return null;  
 }  
//Ниже перечислены операции заполняющие выделенные области памяти в байтах  
  
 public static void generate\_Operation\_Code(Operations\_code b) {  
  
 *byteArray*[*ip*++] = (byte) (b.ordinal());  
 }  
  
 public static void generate\_Address(int a) {  
  
 byte[] bytes\_int = ByteBuffer.*allocate*(*ADDRESS\_SIZE*).putInt(a).array();  
  
 for (byte b : bytes\_int) {  
 *byteArray*[*ip*++] = b;  
 }  
 }  
  
 public static void generate\_Address(float a) {  
  
 byte[] intBytes = ByteBuffer.*allocate*(*ADDRESS\_SIZE*).putFloat(a).array();  
  
 for (byte b : intBytes) {  
 *byteArray*[*ip*++] = b;  
 }  
 }  
  
 public static void getToken() {  
 if (*Token\_iterator*.hasNext()) {  
 *Token\_current* = *Token\_iterator*.next();  
 }  
 }  
  
 public static void tokens\_match(String tokenType) {  
 if (!tokenType.equals(*Token\_current*.get\_Token\_Type())) {  
 throw new Error(String.*format*("Token type (%s) does not match current token type (%s)", tokenType, *Token\_current*.get\_Token\_Type()));  
 } else {  
  
 *getToken*();  
 }  
 }  
  
 public static TYPE get\_Litera\_Type(String tokenType) {  
 switch (tokenType) {  
 case "KW~INTLIT":  
 return TYPE.*INT*;  
 case "KW~FLOATLIT":  
 return TYPE.*REAL*;  
 case "KW~CHARLIT":  
 return TYPE.*CHAR*;  
 case "KW~BOOLLIT":  
 return TYPE.*BOOL*;  
 default:  
 return null;  
 }  
 }  
  
 public static void set\_Tokens\_list\_Iterator(ArrayList<Token> tokenArrayList) {  
 *Token\_iterator* = tokenArrayList.iterator();  
 }  
}

# Symbol.java

/\*  
Например:  
 var x,y,z : integer;  
 x := 2  
 For variable x:  
 name = "x"  
 tokenType = "TK\_A\_VAR"  
 dataType = TYPE.I  
 value = 2  
 address = 0  
 \*/  
  
public class Symbol {  
 private String name;  
 private String Token\_Type;  
 private final Parser.TYPE Data\_Type;  
 private int Address;  
 private int Address\_return; // Возвращаемый адрес для процедуры  
  
 private Object low\_value\_range; // Нижняя граница массива  
 private Object high\_value\_range; // Верхняя граница массива  
  
 private Parser.TYPE index\_Type; // Тип индекса в массиве  
 private Parser.TYPE value\_Type; // Тип значений в массиве  
  
 Symbol next\_entry\_pointer; // Указатель на следующее вхождение в действующем списке таблицы символов  
  
 public Symbol(String name, String Token\_Type, Parser.TYPE Data\_Type, int address) { //Структура символа  
 this.name = name;  
 this.Token\_Type = Token\_Type;  
 this.Data\_Type = Data\_Type;  
 this.Address = address;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public Parser.TYPE getData\_Type() {  
 return Data\_Type;  
 }  
  
 public int getAddress() {  
 return Address;  
 }  
  
 public void set\_Address(int address) {  
 this.Address = address;  
 }  
  
 public String getToken\_Type() {  
 return Token\_Type;  
 }  
  
 public void set\_Token\_Type(String tokenType) {  
 this.Token\_Type = tokenType;  
 }  
  
 public int getAddress\_return() {  
 return Address\_return;  
 }  
  
 public void setAddress\_return(int address\_return) {  
 this.Address\_return = address\_return;  
 }  
  
 public Object get\_Low() {  
 return low\_value\_range;  
 }  
  
 public void set\_Low(Object low\_range) {  
 this.low\_value\_range = low\_range;  
 }  
  
 public Object get\_High() {  
 return high\_value\_range;  
 }  
  
 public void set\_High(Object high\_range) {  
 this.high\_value\_range = high\_range;  
 }  
  
 public Parser.TYPE getIndex\_Type() {  
 return index\_Type;  
 }  
  
 public void set\_Index\_Type(Parser.TYPE indexType) {  
 this.index\_Type = indexType;  
 }  
  
 public Parser.TYPE getValue\_Type() {  
 return value\_Type;  
 }  
  
 public void set\_Value\_Type(Parser.TYPE valueType) {  
 this.value\_Type = valueType;  
 }  
}

# Table\_symbols.java

public final class Table\_symbols {  
  
 static class Scope {  
 Symbol[] Symbol\_Table = new Symbol[*HASH\_TABLE\_SIZE*]; // Таблица символов для текущей области видимости  
 Scope next\_scope\_pointer = null; // Указатель на следующую внешнюю область  
 }  
  
 private static final int *HASH\_TABLE\_SIZE* = 211;  
 private static final Scope *Scope\_header* = new Scope();  
  
 public static void insert(Symbol symbol) {  
 int hash\_Value = *hash*(symbol.getName());  
  
 Symbol Cursor\_bucket = *Scope\_header*.Symbol\_Table[hash\_Value];  
 if (Cursor\_bucket == null) {  
 // Список пуст  
 *Scope\_header*.Symbol\_Table[hash\_Value] = symbol;  
 } else {  
 // Существующие символы в списке  
 while (Cursor\_bucket.next\_entry\_pointer != null) {  
 Cursor\_bucket = Cursor\_bucket.next\_entry\_pointer;  
 }  
  
 // Добавить в конец списка  
 Cursor\_bucket.next\_entry\_pointer = symbol;  
 }  
 }  
  
 public static Symbol find\_and\_get(String symbol\_Name) {  
 int hash\_Value = *hash*(symbol\_Name);  
 Symbol Cursor\_bucket = *Scope\_header*.Symbol\_Table[hash\_Value];  
 Scope Cursor\_scope = *Scope\_header*;  
  
 while (Cursor\_scope != null) {  
 while (Cursor\_bucket != null) {  
 if (Cursor\_bucket.getName().equals(symbol\_Name)) {  
 return Cursor\_bucket;  
 }  
 Cursor\_bucket = Cursor\_bucket.next\_entry\_pointer;  
 }  
 Cursor\_scope = Cursor\_scope.next\_scope\_pointer;  
 }  
  
 // Символ не существует  
 return null;  
 }  
  
 public static int hash(String symbol\_Name) {  
 int hash\_int\_val = 0;  
 for (int i = 0; i < symbol\_Name.length(); i++) {  
 hash\_int\_val = hash\_int\_val + hash\_int\_val + symbol\_Name.charAt(i);  
 }  
  
 hash\_int\_val = hash\_int\_val % *HASH\_TABLE\_SIZE*;  
  
 return hash\_int\_val;  
 }  
  
  
}

# Operations\_simulator.java

import java.io.IOException;  
import java.nio.ByteBuffer;  
import java.nio.file.Files;  
import java.nio.file.Paths;  
import java.util.Arrays;  
import java.util.Stack;  
  
public class Operations\_simulator {  
  
 private static int *ip* = 0;  
 public static String *s* = "";  
 private static int *data\_pointer* = 0;  
  
 private static final Stack<Object> *stack* = new Stack<>();  
  
 private static final Byte[] *Array\_data* = new Byte[1000];  
  
 private static Byte[] *instructions*;  
  
 public static void simulate() {  
 Parser.Operations\_code oper\_Code;  
  
 do {  
 oper\_Code = *get\_Oper\_Code*();  
  
 switch (oper\_Code) { //Вызов соответствующей целевому языку операции по ее "коду"  
 case *PUSH*:  
 *push*();  
 break;  
 case *PUSHI*:  
 *pushi*();  
 break;  
 case *PUSHF*:  
 *pushf*();  
 break;  
 case *POP*:  
 *pop*();  
 break;  
 case *GET*:  
 *get*();  
 break;  
 case *PUT*:  
 *put*();  
 break;  
 case *CVR*:  
 *cvr*();  
 break;  
 case *XCHG*:  
 *xchg*();  
 break;  
 case *JMP*:  
 *jmp*();  
 break;  
 case *PRINT\_REAL*:  
 *printReal*();  
 break;  
 case *PRINT\_INT*:  
 *printInt*();  
 break;  
 case *PRINT\_BOOL*:  
 *printBool*();  
 break;  
 case *PRINT\_CHAR*:  
 *printChar*();  
 break;  
  
 case *PRINT\_NEWLINE*:  
 System.*out*.println();  
 break;  
 case *HALT*:  
 *halt*();  
 break;  
 case *EQL*:  
 *eql*();  
 break;  
 case *NEQL*:  
 *neql*();  
 break;  
 case *LSS*:  
 *less*();  
 break;  
 case *LEQ*:  
 *lessEql*();  
 break;  
 case *GTR*:  
 *greater*();  
 break;  
 case *GEQ*:  
 *greaterEql*();  
 break;  
 case *JFALSE*:  
 *jfalse*();  
 break;  
 case *JTRUE*:  
 *jtrue*();  
 break;  
 case *ADD*:  
 *add*();  
 break;  
 case *FADD*:  
 *fadd*();  
 break;  
 case *SUB*:  
 *sub*();  
 break;  
 case *FSUB*:  
 *fsub*();  
 break;  
 case *MULT*:  
 *mult*();  
 break;  
 case *FMULT*:  
 *fmult*();  
 break;  
 case *DIV*:  
 *div*();  
 break;  
  
 case *MOD*:  
 *mod*();  
 break;  
 case *FDIV*:  
 *fdiv*();  
 break;  
 default:  
 throw new Error(String.*format*("Unhandled case: %s", oper\_Code));  
 }  
  
 }  
 while (oper\_Code != Parser.Operations\_code.*HALT*);  
  
 }  
  
 private static void pushf() { //В стек компилятора добавляется число с плавающей точкой  
 float val = *get\_Float\_val*();  
 *s* += ("pushf " + val) + "\n";  
 *stack*.push(val);  
 }  
  
 private static void get() { //Из стека извлекается указатель на некоторый элемент  
 *data\_pointer* = (int) *stack*.pop();  
 int val = *get\_Data*(*data\_pointer*);  
 *s* += ("get " + val + " : " + *data\_pointer*) + "\n";  
 *stack*.push(val);  
 }  
  
 private static void put() { //В область памяти записывается последовательность байт, то есть некоторое значение  
 Object val = *stack*.pop();  
 *data\_pointer* = (int) *stack*.pop();  
  
  
 byte[] value\_Bytes;  
 if (val instanceof Integer) {  
 value\_Bytes = ByteBuffer.*allocate*(4).putInt((int) val).array();  
 } else {  
 value\_Bytes = ByteBuffer.*allocate*(4).putFloat((float) val).array();  
 }  
  
 *s* += ("put (pointer:" + *data\_pointer* + ") : (bytes: ");  
 for (byte b : value\_Bytes) {  
  
 *Array\_data*[*data\_pointer*++] = b;  
  
  
  
 *s* = *s* + b;  
 }  
 *s* += (") : " + val) + "\n";  
 }  
  
 private static void jtrue() { //Проверка истинности  
 if (*stack*.pop().toString().equals("true")) {  
 *ip* = *get\_Addr\_Val*();  
 *s* += ("jtrue " + *ip*) + "\n";  
 } else {  
 *get\_Addr\_Val*();  
 *s* += ("jtrue (false) ") + "\n";  
 }  
 }  
  
 private static void jfalse() { //Проверка ложности  
 if (*stack*.pop().toString().equals("false")) {  
 *ip* = *get\_Addr\_Val*();  
 *s* += ("jfalse " + *ip*) + "\n";  
 } else {  
 *get\_Addr\_Val*();  
 *s* += ("jfalse (false) ") + "\n";  
 }  
 }  
  
 private static void eql() { //Проверка равны ли элементы между собой  
 Integer i2 = (Integer) *stack*.pop();  
 Float val2 = (float) i2;  
  
 Integer i1 = (Integer) *stack*.pop();  
 Float val1 = (float) i1;  
  
 *stack*.push(val1.equals(val2));  
  
 *s* += ("eql " + val1 + " =? " + val2 + " is " + val1.equals(val2)) + "\n";  
 }  
  
 private static void neql() {//Проверка не равны ли элементы между собой  
 Integer i2 = (Integer) *stack*.pop();  
 Float val2 = (float) i2;  
  
 Integer i1 = (Integer) *stack*.pop();  
 Float val1 = (float) i1;  
  
 *stack*.push(!val1.equals(val2));  
 *s* += ("neql " + val1 + " !=? " + val2 + " is " + !val1.equals(val2)) + "\n";  
 }  
  
 private static void less() { //Проверка меньше ли первый элемент чем второй  
 Integer i2 = (Integer) *stack*.pop();  
 float val2 = (float) i2;  
  
 Integer i1 = (Integer) *stack*.pop();  
 float val1 = (float) i1;  
  
 *stack*.push(val1 < val2);  
 *s* += ("less " + val1 + " <? " + val2 + " is " + (val1 < val2)) + "\n";  
 }  
  
 private static void greater() { //Проверка больше ли первый элемент чем второй  
 Integer i2 = (Integer) *stack*.pop();  
 float val2 = (float) i2;  
  
 Integer i1 = (Integer) *stack*.pop();  
 float val1 = (float) i1;  
  
 *stack*.push(val1 > val2);  
 *s* += ("greater " + val1 + " >? " + val2 + " is " + (val1 > val2)) + "\n";  
 }  
  
 private static void lessEql() { //Проверка меньше/равен ли первый элемент второму  
 Integer i2 = (Integer) *stack*.pop();  
 float val2 = (float) i2;  
  
 Integer i1 = (Integer) *stack*.pop();  
 float val1 = (float) i1;  
  
 *stack*.push(val1 <= val2);  
 *s* += ("lessEql " + val1 + " <=? " + val2 + " is " + (val1 <= val2)) + "\n";  
 }  
  
 private static void greaterEql() { //Проверка больше/равен ли первый элемент второму  
 Integer i2 = (Integer) *stack*.pop();  
 float val2 = (float) i2;  
  
 Integer i1 = (Integer) *stack*.pop();  
 float val1 = (float) i1;  
  
 *stack*.push(val1 >= val2);  
 *s* += ("greaterEql " + val1 + " >=? " + val2 + " is " + (val1 >= val2)) + "\n";  
 }  
  
 private static void printReal() { //Вывод на экран числа с плавающей точкой  
  
 Object val = *stack*.pop();  
  
 if (val instanceof Integer) {  
 byte[] valArray = ByteBuffer.*allocate*(4).putInt((int) val).array();  
 *s* += ("printReal " + Arrays.*toString*(valArray) + " : " + " : " + ByteBuffer.*wrap*(valArray) + " : " + ByteBuffer.*wrap*(valArray).getFloat()) + "\n";  
 System.*out*.print(ByteBuffer.*wrap*(valArray).getFloat());  
 } else {  
 *s* += (val) + "\n";  
 System.*out*.print(val);  
 }  
  
 }  
  
 private static void printBool() { //Вывод на экран булева значения  
 int val = (int) *stack*.pop();  
 if (val == 1) {  
 *s* += ("printBool " + val + " (True)") + "\n";  
 System.*out*.print("True");  
 } else {  
 *s* += ("printBool " + val + " (False)") + "\n";  
 System.*out*.print("False");  
 }  
 }  
  
 public static void printInt() { //Вывод на экран целочисленного значения  
  
 int x = (Integer) *stack*.pop();  
 *s* += ("printInt (From stack) " + x) + "\n";  
 System.*out*.print(x);  
  
 }  
  
 public static void printChar() { //Вывод на экран символа  
  
 int x = (Integer) *stack*.pop();  
 *s* += ("printChar (From stack) " + Character.*toChars*(x)[0]) + "\n";  
 System.*out*.print(Character.*toChars*(x)[0]);  
  
 }  
  
  
  
  
 public static void add() { //Операция сложения  
 int val1 = (int) *stack*.pop();  
 int val2 = (int) *stack*.pop();  
 *stack*.push(val1 + val2);  
 *s* += ("add " + val1 + " + " + val2) + "\n";  
 }  
  
 private static void fadd() { //Операция сложения чисел с плавающей точкой  
 float val1 = (float) *stack*.pop();  
 float val2 = (float) *stack*.pop();  
 *stack*.push(val1 + val2);  
 *s* += ("fadd " + val1 + " + " + val2) + "\n";  
 }  
  
  
 public static void sub() { //Операция вычитания  
 int val1 = (int) *stack*.pop();  
 int val2 = (int) *stack*.pop();  
 *stack*.push(val2 - val1);  
 *s* += ("sub " + val2 + " - " + val1) + "\n";  
 }  
  
 public static void fsub() { //Операция вычитания чисел с плавающей точкой  
 float val1 = (float) *stack*.pop();  
 float val2 = (float) *stack*.pop();  
 *stack*.push(val2 - val1);  
 *s* += ("fsub " + val2 + " - " + val1) + "\n";  
 }  
  
 public static void mult() { //Операция умножения  
 int val1 = (int) *stack*.pop();  
 int val2 = (int) *stack*.pop();  
 *stack*.push(val1 \* val2);  
 *s* += ("mult " + val1 + " \* " + val2) + "\n";  
 }  
  
 public static void fmult() { //Операция умножения чисел с плавающей точкой  
 float val1 = (float) *stack*.pop();  
 float val2 = (float) *stack*.pop();  
 *stack*.push(val1 \* val2);  
 *s* += ("fmult " + val1 + " \* " + val2) + "\n";  
 }  
  
 public static void fdiv() { //Операция деления чисел с плавающей точкой  
 float val2 = (float) *stack*.pop();  
 float val1 = (float) *stack*.pop();  
  
 *stack*.push(val1 / val2);  
 *s* += ("fdiv " + val1 + " / " + val2) + "\n";  
 }  
  
 public static void div() { //Операция деления нацело  
 int val2 = (int) *stack*.pop();  
 int val1 = (int) *stack*.pop();  
 *stack*.push(val1 / val2);  
 *s* += ("div " + val1 + " / " + val2) + "\n";  
 }  
 public static void mod() { //Операция деления с остатком нацело  
 int val2 = (int) *stack*.pop();  
 int val1 = (int) *stack*.pop();  
 *stack*.push((int)val1 % val2);  
 *s* += ("mod " + val1 + " % " + val2) + "\n";  
 }  
  
 public static void cvr() { //Извлечение значения с плавающей точкой из строкового представления элемента стека  
 float val = Float.*parseFloat*(String.*valueOf*(*stack*.pop()));  
 *stack*.push(val);  
 *s* += ("cvr (from Stack) : " + val) + "\n";  
 }  
  
 public static void xchg() { //Обмен пары значений с верхушки стека  
 Object val1 = *stack*.pop();  
 Object val2 = *stack*.pop();  
 *stack*.push(val1);  
 *stack*.push(val2);  
 *s* += ("xchg : pop " + val1 + ", pop " + val2 + ", push " + val1 + " , push " + val2) + "\n";  
 }  
  
 public static void pushi() { //Добавление элемента в стек  
 int val = *get\_Addr\_Val*();  
 *stack*.push(val);  
 *s* += ("pushi :" + val) + "\n";  
 }  
  
 public static void push() { //Добавление элемента по указателю в стек  
 *data\_pointer* = *get\_Addr\_Val*();  
 int val = *get\_Data*(*data\_pointer*);  
 *stack*.push(val);  
 *s* += ("push : (pointer:" + *data\_pointer* + ") : " +val) + "\n";  
 }  
  
 public static void pop() { //Извлечение элемента из стека по указателю  
 Object val = *stack*.pop();  
 *data\_pointer* = *get\_Addr\_Val*();  
  
  
 byte[] value\_Bytes;  
 if (val instanceof Integer) {  
 value\_Bytes = ByteBuffer.*allocate*(4).putInt((int) val).array();  
 }  
  
 else {  
 value\_Bytes = ByteBuffer.*allocate*(4).putFloat((float) val).array();  
 }  
  
 *s* += ("put (pointer:" + *data\_pointer* + ") : (bytes :");  
 for (byte b : value\_Bytes) {  
 *Array\_data*[*data\_pointer*++] = b;  
 *s* += b;  
 }  
  
 *s* += (") : " + val) + "\n";  
 }  
  
 public static void jmp() {  
  
 *ip* = *get\_Addr\_Val*();  
 *s* += ("jmp : (address: " + *ip*) + ") " + "\n";  
 }  
  
  
 public static void halt() { //Выход из программы с кодом 0, компиляция выполнена  
 *s* += ("halt : code 0") + "\n";  
 System.*out*.print("\nProgram finished with exit code 0\n");  
  
 try {  
 if (!Files.*exists*(Paths.*get*((Run.*filepath*+"/example/" + Run.*filename* + "-operations.txt"))))  
 Files.*createFile*(Paths.*get*((Run.*filepath*+"/example/" + Run.*filename* + "-operations.txt")));  
 Files.*write*(Paths.*get*(Run.*filepath*+"/example/" + Run.*filename* + "-operations.txt"), *s*.getBytes());  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 System.*exit*(0);  
 }  
  
 public static int get\_Addr\_Val() { //Получение значения из байтового представления в виде целочисленного значения  
 byte[] valArray = new byte[4];  
 for (int i = 0; i < 4; i++) {  
 valArray[i] = *instructions*[*ip*++];  
 }  
  
 return ByteBuffer.*wrap*(valArray).getInt();  
 }  
  
 public static float get\_Float\_val() { //Получение значения из байтового представления в виде числа с плавающей точкой  
 byte[] valArray = new byte[4];  
 for (int i = 0; i < 4; i++) {  
 valArray[i] = *instructions*[*ip*++];  
 }  
  
 return ByteBuffer.*wrap*(valArray).getFloat();  
 }  
  
 public static int get\_Data(int dp) { //Получение значения по указателю  
 byte[] valArray = new byte[4];  
 for (int i = 0; i < 4; i++) {  
 valArray[i] = *Array\_data*[dp++];  
 }  
  
 return ByteBuffer.*wrap*(valArray).getInt();  
 }  
  
  
 public static Parser.Operations\_code get\_Oper\_Code() {  
 return Parser.Operations\_code.*values*()[*instructions*[*ip*++]];  
 } //Получение непосредственно "кода" операции  
  
 public static void set\_Instructions(Byte[] instructions) {  
 Operations\_simulator.*instructions* = instructions;  
 }  
  
}

# Comments\_remover.java

import java.util.Scanner;  
  
public class Comments\_remover {  
 public static String remover(Scanner in)  
 {  
 String str = "";  
 String subs, subs\_delete;  
 while(in.hasNext())  
 {  
 subs = in.nextLine();  
 if (subs.contains("//")) {  
 if (subs.contains(" //"))  
 subs\_delete = subs.substring(subs.indexOf(" //"));  
 else  
 subs\_delete = subs.substring(subs.indexOf("//"));  
 subs = subs.replace(subs\_delete, "");  
 }  
 if (subs.contains("{")) {  
 while (in.hasNext()) {  
 subs += in.nextLine();  
 if (subs.contains("//")) {  
 if (subs.contains(" //"))  
 subs\_delete = subs.substring(subs.indexOf(" //"));  
else  
 subs\_delete = subs.substring(subs.indexOf("//"));  
 subs = subs.replace(subs\_delete, "");  
 }  
 if (subs.contains("}"))  
 break;  
 }  
 if (subs.contains(" {"))  
 subs\_delete = subs.substring(subs.indexOf(" {"));  
 else  
 subs\_delete = subs.substring(subs.indexOf("{"));  
 if (subs\_delete.contains("} "))  
 subs\_delete = subs\_delete.substring(0, subs\_delete.indexOf("} ") + 1);  
 else  
 subs\_delete = subs\_delete.substring(0, subs\_delete.indexOf("}") + 1);  
 subs = subs.replace(subs\_delete, "");  
 }  
  
 str += subs + "\r\n";  
 }  
  
  
  
 in.close();  
  
  
  
 return str;  
 }  
}

# Blank\_begin\_end\_remover.java

import java.util.LinkedList;  
  
public class Blank\_begin\_end\_remover {  
 private static LinkedList<String> *ScannedList*;  
 private static LinkedList<String> *ClearedList*;  
 private static Boolean *cleared* = false;  
 private static Boolean *useful\_block* = false;  
 private static int *loop\_type* = 0; // 0 - for, 1 - while  
 public static String remover(String input\_str)  
 {  
 *ScannedList* = new LinkedList<>();  
 *ClearedList* = new LinkedList<>();  
 *loop\_type* = 0;  
*cleared* = false;  
 *useful\_block* = false;  
 String str = new String() ;  
 String splitter = input\_str;  
  
  
 String[] Operators;  
  
 Operators = splitter.split(" ");  
  
  
 for (String i : Operators  
 ) {  
 if ((((String) (i)).length() > 0) && (i != "\r") && (i != ""))  
 *ScannedList*.addLast(i);  
 }  
  
  
 while (*cleared* == false) {  
 *cleared* = true;  
  
 while (!*ScannedList*.isEmpty()) {  
 if (*ScannedList*.getFirst() .contains("begin") )  
 *clearsp*();  
  
  
 else if ((*ScannedList*.getFirst() .contains("end"))) {  
 *ClearedList*.addLast(*ScannedList*.getFirst());  
  
 *ScannedList*.removeFirst();  
 } else {  
 *ClearedList*.addLast(*ScannedList*.getFirst());  
  
 *ScannedList*.removeFirst();  
 }  
 }  
 if (*cleared* == false) {  
  
  
 *ScannedList*.addAll(*ClearedList*);  
  
 *ClearedList*.clear();  
 }  
 }  
 *ScannedList*.addAll(*ClearedList*);  
  
 *ClearedList*.clear();  
  
 while (!*ScannedList*.isEmpty()) {  
 if ( *ScannedList*.getFirst() .contains( "for"))  
 *loop\_type* = 0;  
 if ( *ScannedList*.getFirst() .contains( "while"))  
 *loop\_type* = 1;  
  
 if (*ScannedList*.getFirst() .contains( "begin")) {  
 *ClearedList*.addLast(*ScannedList*.getFirst());  
  
 *ScannedList*.removeFirst();  
  
  
 if (*ScannedList*.getFirst() .contains("end")) //reached end with just spaces  
 {  
 // r.pop\_front();  
 *ScannedList*.removeFirst(); //;  
 *ClearedList*.removeLast();  
 // p.pop\_back();  
 } else {  
 *ClearedList*.addLast(*ScannedList*.getFirst());  
  
 *ScannedList*.removeFirst();  
 }  
  
  
 } else if (*ScannedList*.getFirst() .contains( "end")) {  
 *ClearedList*.addLast(*ScannedList*.getFirst());  
  
 *ScannedList*.removeFirst();  
 } else {  
 *ClearedList*.addLast(*ScannedList*.getFirst());  
  
 *ScannedList*.removeFirst();  
 }  
 }  
 str ="";  
 for (int i = 0; i< *ClearedList*.size(); i++) {  
 str+=(*ClearedList*.get(i)+" ");  
  
 }  
  
  
  
 return str;  
 }  
 private static void clearsp() {  
 *useful\_block* =false;  
  
 *ClearedList*.addLast(*ScannedList*.getFirst());  
  
 *ScannedList*.removeFirst();  
  
  
 if (*ScannedList*.getFirst() .contains( "begin"))  
 *clearsp*();  
  
  
 else if ((*ScannedList*.getFirst() .contains( "end")) &&(*useful\_block* ==false))//reached end with just spaces  
 {  
 *ScannedList*.removeFirst();  
 for (int i = 0; i< *ClearedList*.size(); i++)  
 if ( (*ClearedList*.get(i)) .contains( "for"))  
 *loop\_type* = 0;  
 else if ( (*ClearedList*.get(i)) .contains( "while"))  
 *loop\_type* = 1;  
  
if (*loop\_type* == 0) {  
if (!*ClearedList*.isEmpty())  
 while (!(*ClearedList*.getLast().contains("for"))) {  
  
  
 *ClearedList*.removeLast();  
 }  
  
}  
else if (*loop\_type* == 1) {  
 if (!*ClearedList*.isEmpty())  
 while (!(*ClearedList*.getLast().contains("while"))) {  
  
  
 *ClearedList*.removeLast();  
 }  
}  
 *ClearedList*.removeLast();  
  
  
  
 *cleared* = false;  
 } else if (*cleared* == true) {  
 *ClearedList*.addLast(*ScannedList*.getFirst());  
 if (!(*ScannedList*.getFirst().equals("\n")))  
 *useful\_block* = true;  
 *ScannedList*.removeFirst();  
 }  
 }  
}

# Blank\_repeat\_until\_remover.java

import java.util.LinkedList;  
  
public class Blank\_repeat\_until\_remover {  
 private static LinkedList<String> *ScannedList*;  
 private static LinkedList<String> *ClearedList*;  
 private static Boolean *cleared* = false;  
 private static Boolean *useful\_block* = false;  
 public static String remover(String input\_str)  
 {  
 *ScannedList* = new LinkedList<>();  
 *ClearedList* = new LinkedList<>();  
 *cleared* = false;  
 *useful\_block* = false;  
 String str = "";  
 String splitter = input\_str;  
  
  
 String[] Operators;  
  
 Operators = splitter.split(" ");  
  
  
 for (String i : Operators  
 ) {  
 if ((((String) (i)).length() > 0) && (i != "\r") && (i != ""))  
 *ScannedList*.addLast(i);  
 }  
  
  
 while (*cleared* == false) {  
 *cleared* = true;  
  
 while (!*ScannedList*.isEmpty()) {  
 if (*ScannedList*.getFirst() .contains("repeat") )  
 *clearsp*();  
  
  
 else if ((*ScannedList*.getFirst() .contains("until"))) {  
 *ClearedList*.addLast(*ScannedList*.getFirst());  
  
 *ScannedList*.removeFirst();  
 } else {  
 *ClearedList*.addLast(*ScannedList*.getFirst());  
  
 *ScannedList*.removeFirst();  
 }  
 }  
 if (*cleared* == false) {  
  
  
 *ScannedList*.addAll(*ClearedList*);  
  
 *ClearedList*.clear();  
 }  
 }  
 *ScannedList*.addAll(*ClearedList*);  
  
 *ClearedList*.clear();  
  
 while (!*ScannedList*.isEmpty()) {  
  
  
 if (*ScannedList*.getFirst() .contains( "repeat")) {  
 *ClearedList*.addLast(*ScannedList*.getFirst());  
  
 *ScannedList*.removeFirst();  
  
  
 if (*ScannedList*.getFirst() .contains("until")) //reached until with just spaces  
 {  
  
 *ScannedList*.removeFirst();  
 *ClearedList*.removeLast();  
  
 } else {  
 *ClearedList*.addLast(*ScannedList*.getFirst());  
  
 *ScannedList*.removeFirst();  
 }  
  
  
 } else if (*ScannedList*.getFirst() .contains( "until")) {  
 *ClearedList*.addLast(*ScannedList*.getFirst());  
  
 *ScannedList*.removeFirst();  
 } else {  
 *ClearedList*.addLast(*ScannedList*.getFirst());  
  
 *ScannedList*.removeFirst();  
 }  
 }  
 str ="";  
 for (int i = 0; i< *ClearedList*.size(); i++) {  
 str+=(*ClearedList*.get(i)+" ");  
  
 }  
  
  
  
 return str;  
 }  
 private static void clearsp() {  
 *useful\_block* =false;  
 *ClearedList*.addLast(*ScannedList*.getFirst());  
  
 *ScannedList*.removeFirst();  
 if (*ScannedList*.getFirst() .contains( "repeat"))  
 *clearsp*();  
 else if ((*ScannedList*.getFirst() .contains( "until")) &&(*useful\_block* ==false))//reached until with just spaces  
 {  
 *ScannedList*.removeFirst();  
 while (!(*ScannedList*.getFirst() .contains( ";")))  
 *ScannedList*.removeFirst();  
 *ScannedList*.removeFirst();  
 *ClearedList*.removeLast();  
  
  
  
 *cleared* = false;  
 } else if (*cleared* == true) {  
 *ClearedList*.addLast(*ScannedList*.getFirst());  
 if (!(*ScannedList*.getFirst().equals("\n")))  
 *useful\_block* = true;  
 *ScannedList*.removeFirst();  
 }  
 }  
}