Оглавление

[ВВЕДЕНИЕ 3](#_Toc42195733)

[ПРОЕКТИРОВАНИЕ КОМПИЛЯТОРА 4](#_Toc42195734)

[1.1 ПРОЕКТИРОВАНИЕ СКАНЕРА ТОКЕНОВ 4](#_Toc42195735)

[1.2 Структура данных - Токен 5](#_Toc42195736)

[1.3 Проектирование парсера 6](#_Toc42195737)

[1.4 Проектирование симулятора операций 13](#_Toc42195738)

[1.5 Объединение модулей в единую программу 16](#_Toc42195739)

[1.6 Оптимизация 17](#_Toc42195740)

[2.1 Упаковка приложения 20](#_Toc42195741)

[2.5 Руководство пользователя 21](#_Toc42195742)

[3. Тесты 21](#_Toc42195743)

[ЗАКЛЮЧЕНИЕ 26](#_Toc42195744)

[Приложение 26](#_Toc42195745)

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. Программа интерпертируется.

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

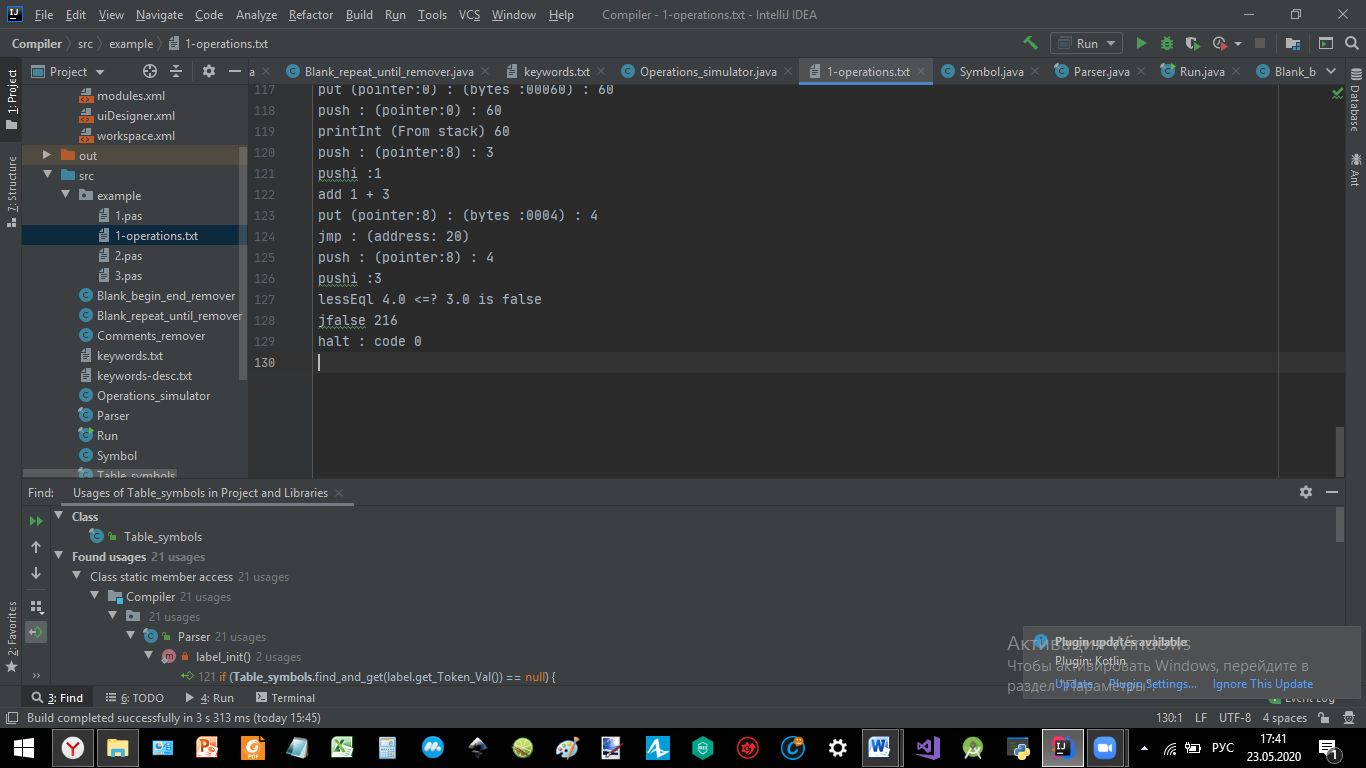
import java.util.LinkedList;  
  
public class Blank\_begin\_end\_remover {  
 private static LinkedList<String> *ScannedList*;  
 private static LinkedList<String> *ClearedList*;  
 private static Boolean *cleared* = false;  
 private static Boolean *useful\_block* = false;  
 private static int *loop\_type* = 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/

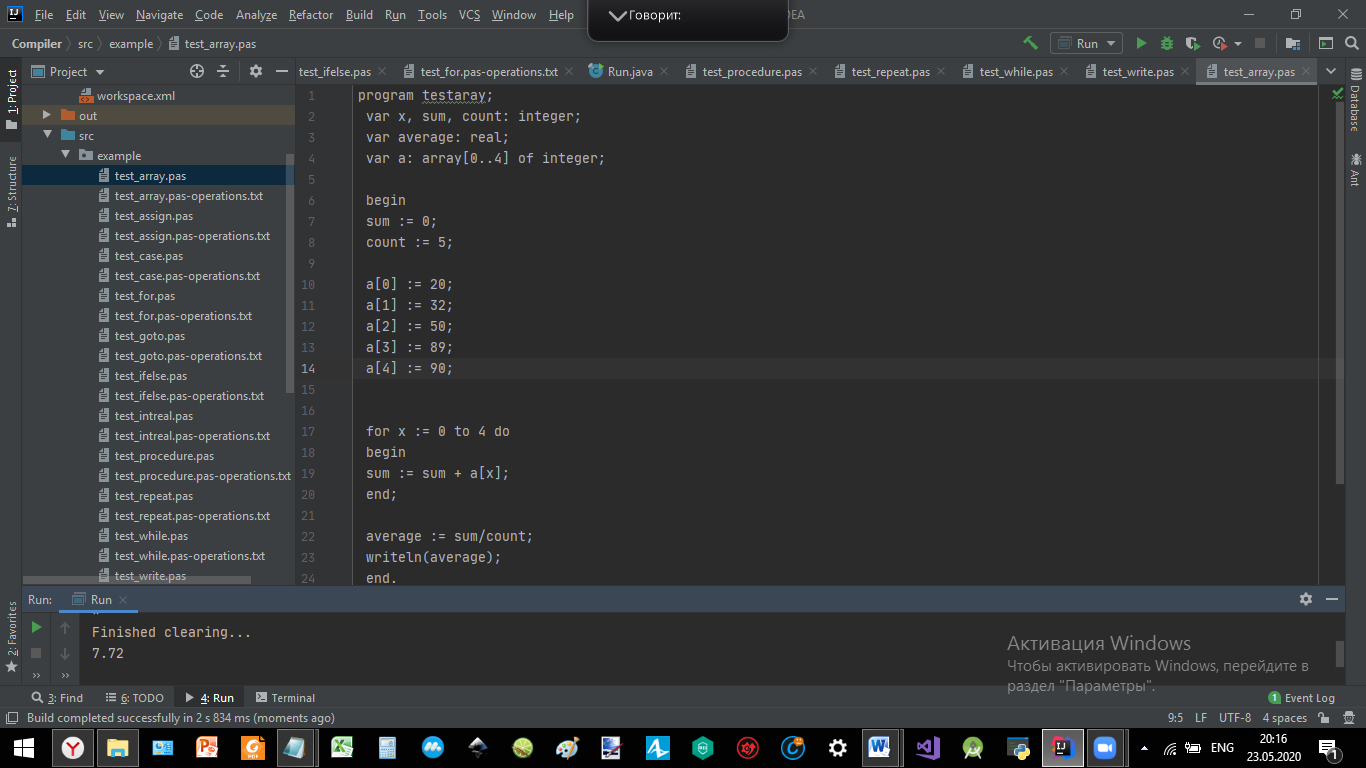
Скомпилировать

javac Run.java

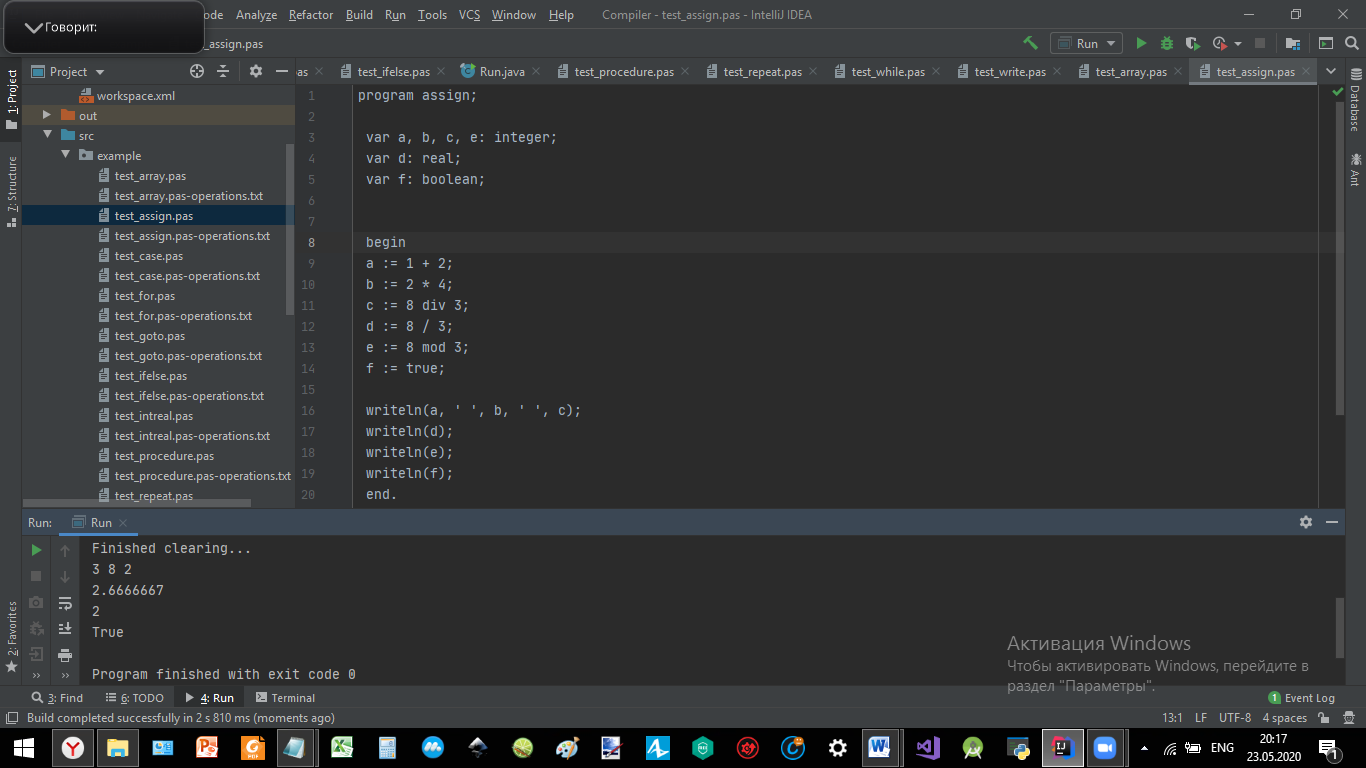
Запустить

java Run

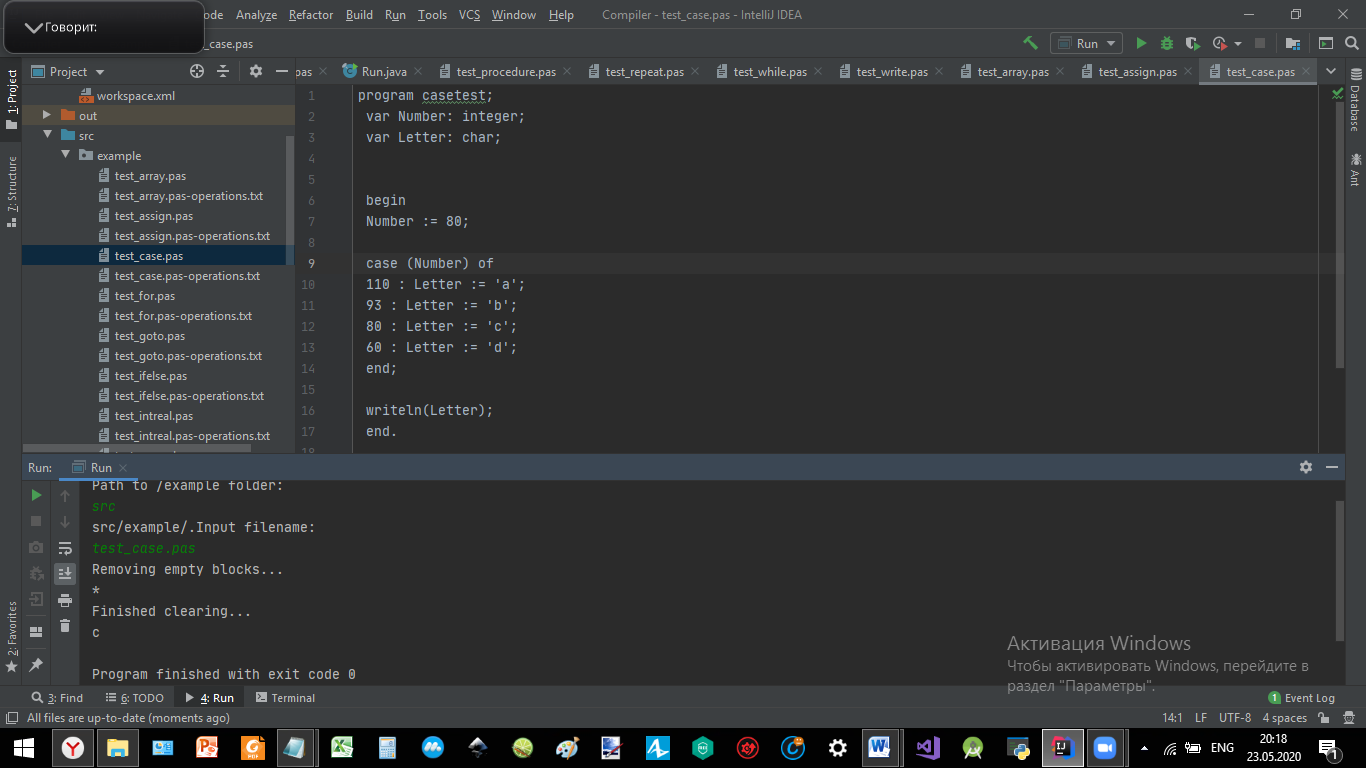
# 3. Тесты



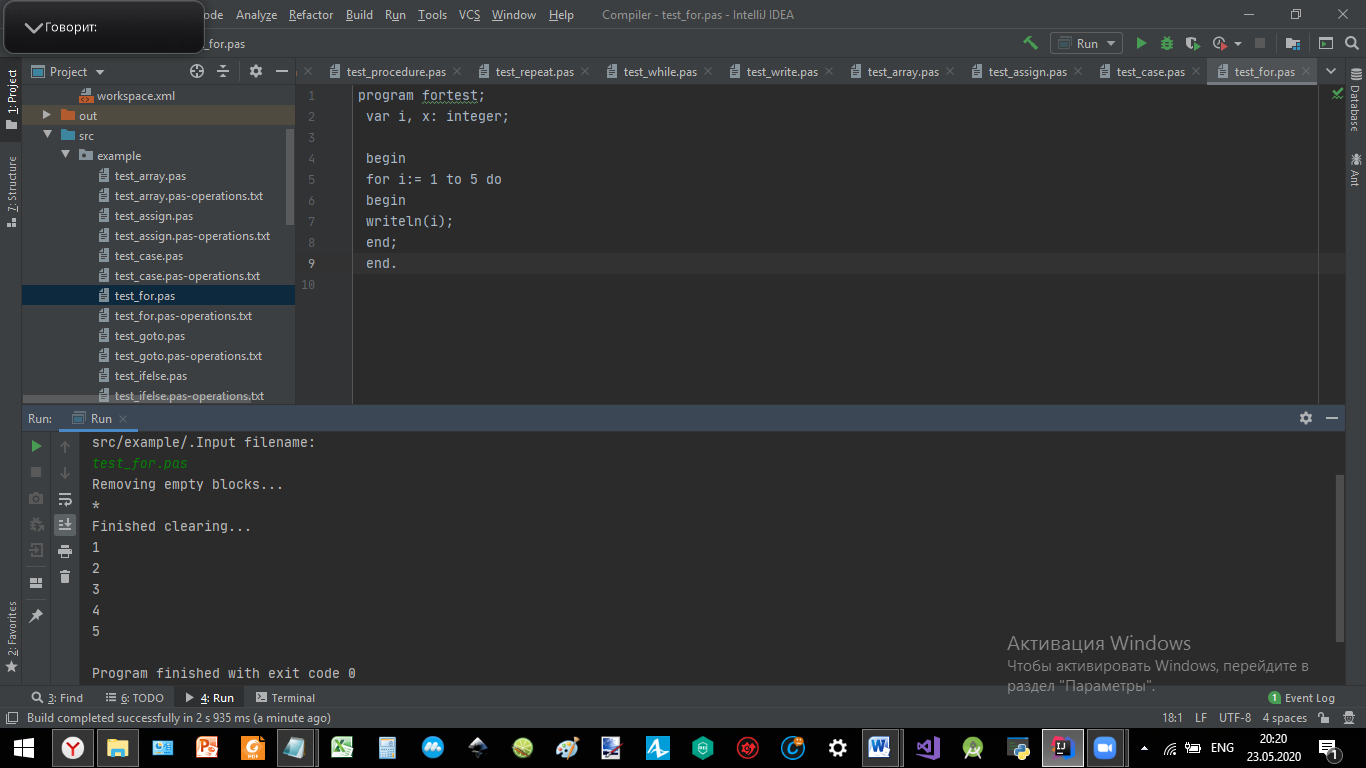
**Рисунок 2. Тест массивов**



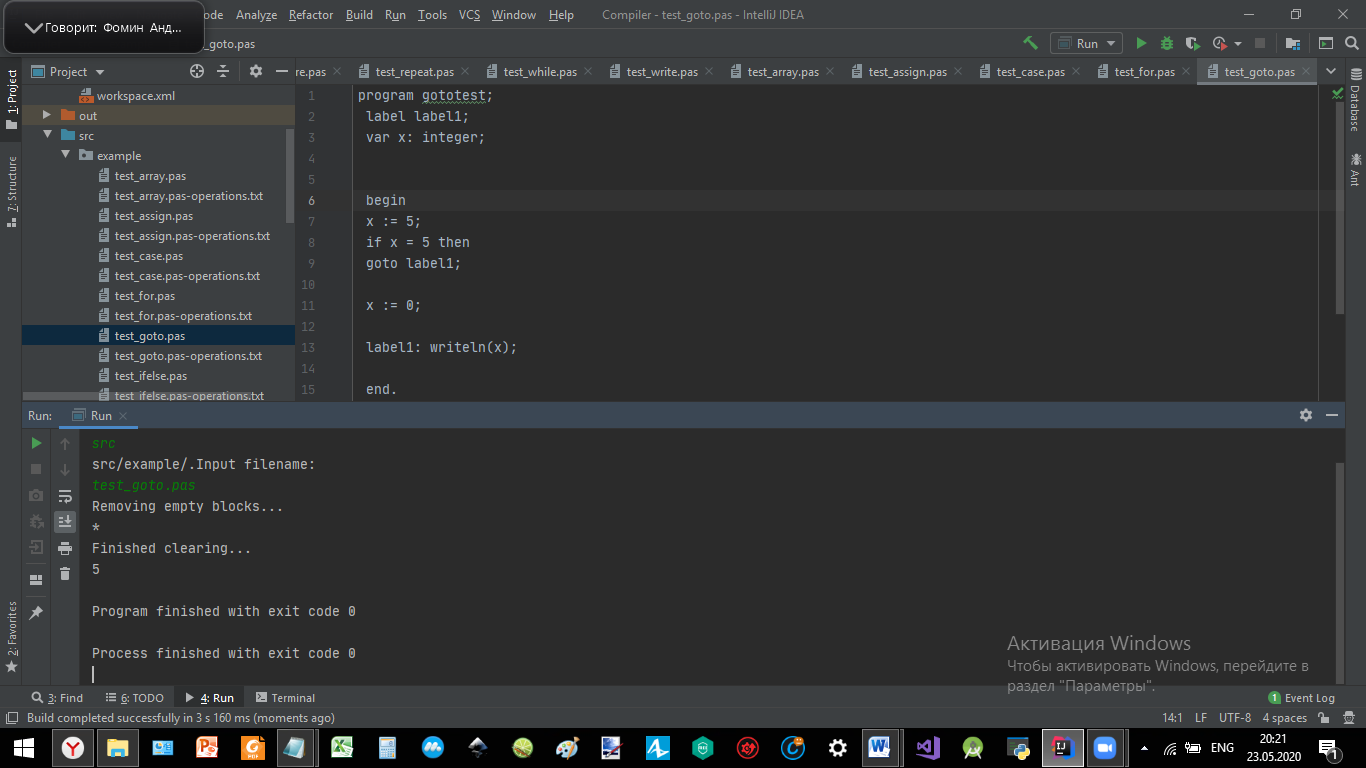
**Рисунок 3. Тест присваивания**



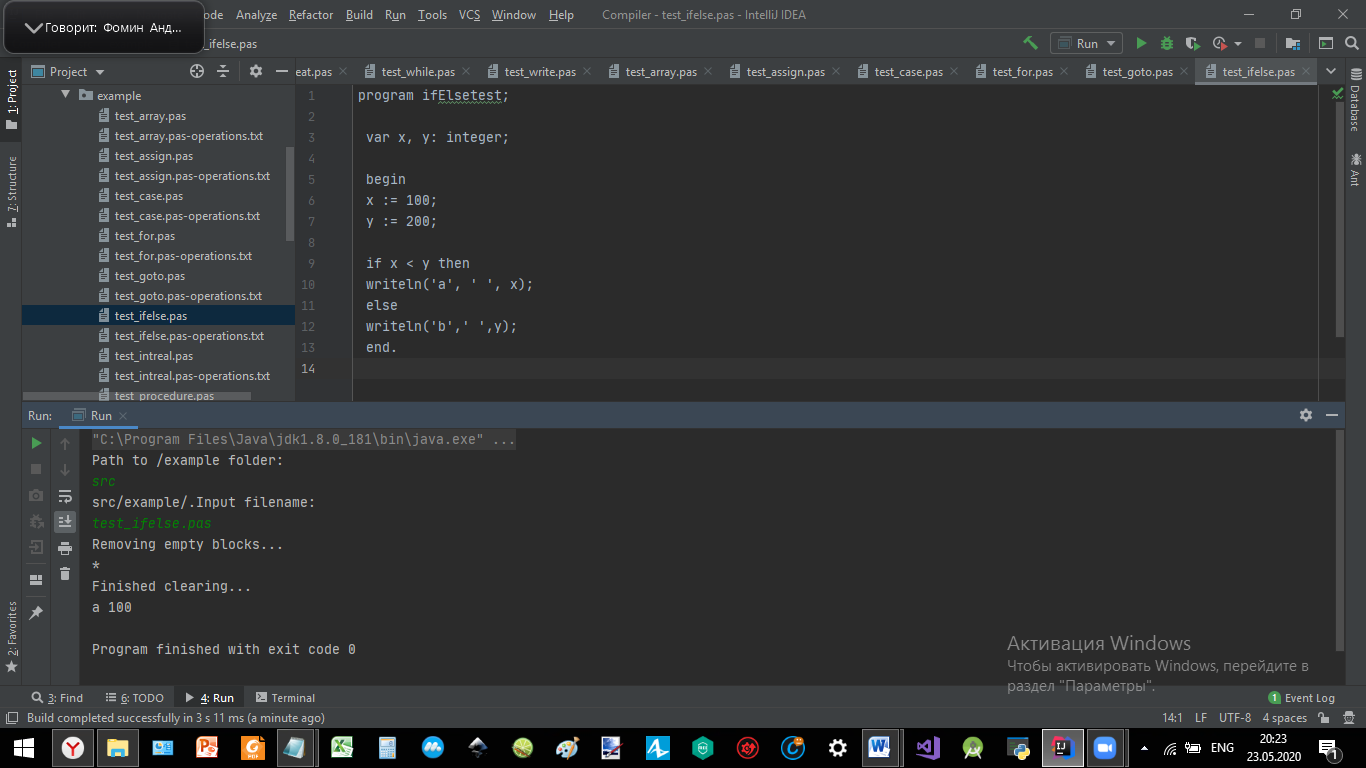
**Рисунок 4. Тест case**



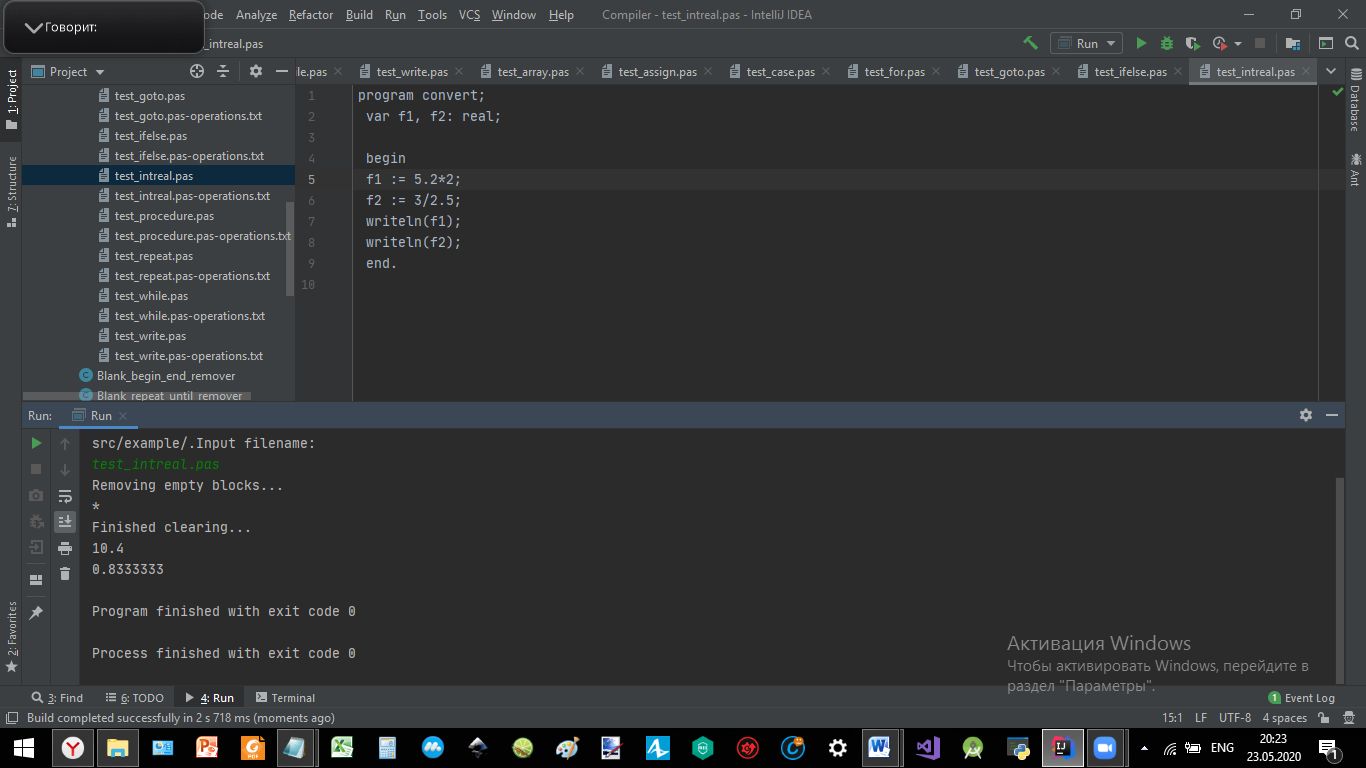
**Рисунок 5. Тест for**



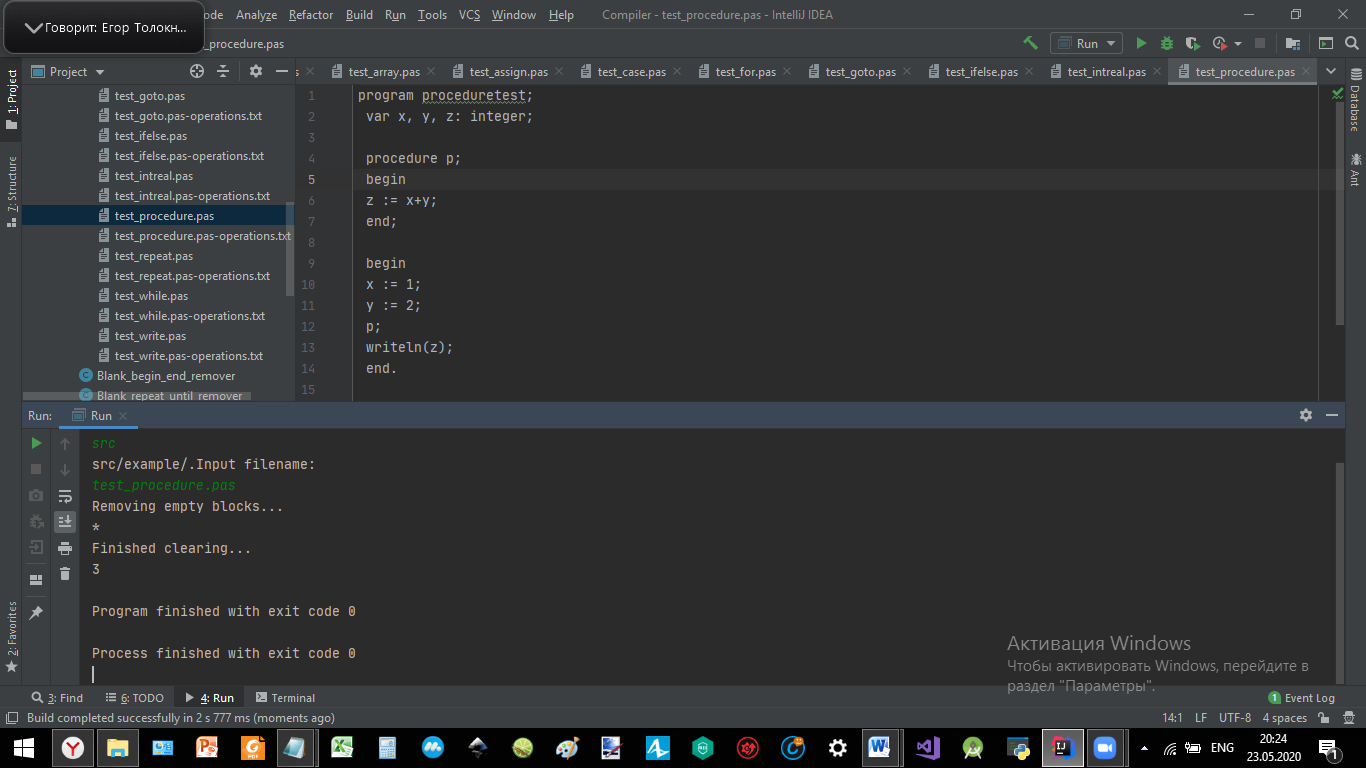
**Рисунок 6. Тест goto**



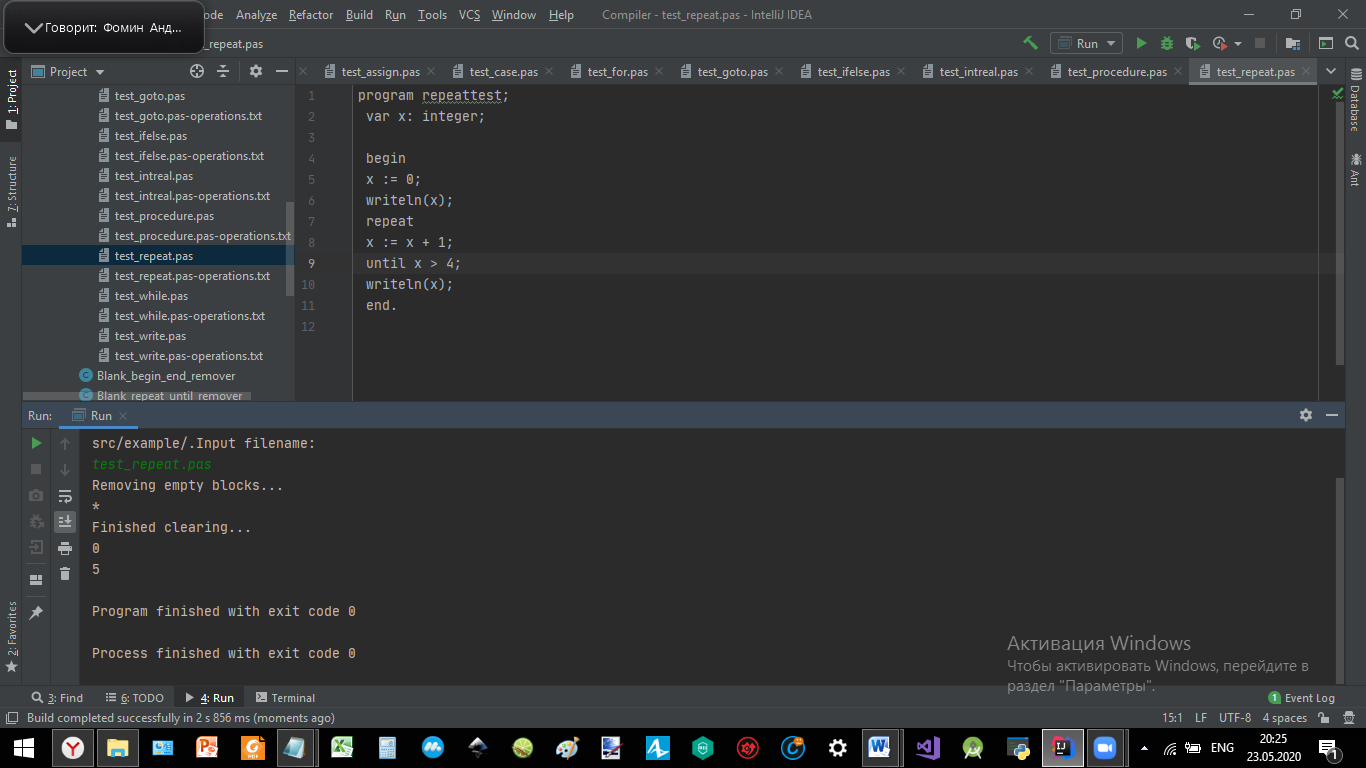
**Рисунок 7. Тест if-else**



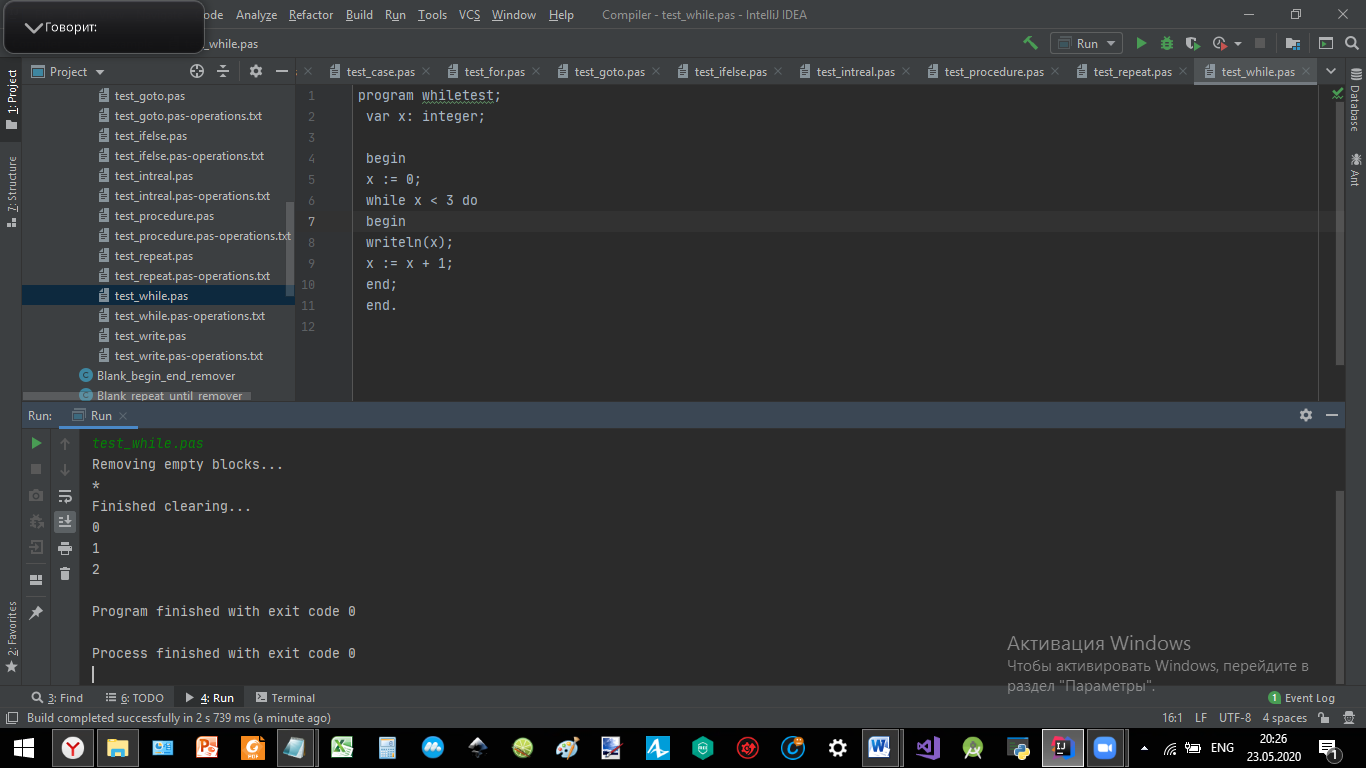
**Рисунок 8. Тест intreal**



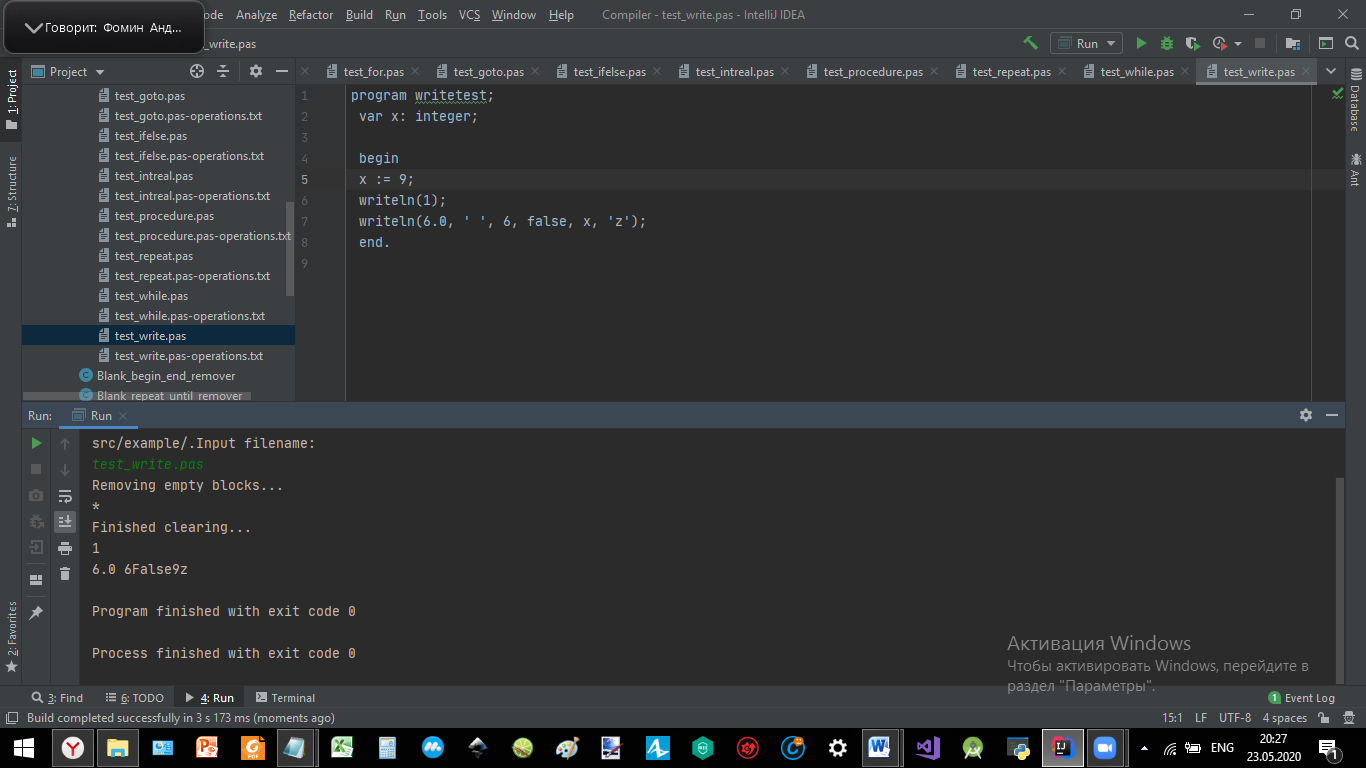
**Рисунок 9. Тест процедуры**



**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

**Github:** https://github.com/egort-33/ISB117