Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

высшего образования

«Владимирский государственный университет

имени Александра Григорьевича и Николая Григорьевича Столетовых»

(ВлГУ)

разработка компилятора подмножества

процедурно-ориентированного языка

Пояснительная записка

RU. 643.02068048.0001-01 81 01

# АННОТАЦИЯ

Компилятор – программный модуль, задачей которого является перевод программы, написанной на одном из языков программирования (исходный язык) в программу на машинном коде, либо в байт-код.

Курсовая работа заключается в создании компилятора подмножества процедурно-ориентированного языка. С исходным кодом проекта можно ознакомиться, перейдя по ссылке: <https://github.com/BentoNight/pascal-compiler>.

В пояснительной записке приведены требования к разрабатываемому проекту и описаны все стадии проектирования компилятора, а именно: построение лексического анализатора, создание синтаксического анализатора, проектирование генератора байт-кода, построение оптимизатора, а также приведены результаты тестирования разработанного компилятора.**СОДЕРЖАНИЕ**

[АННОТАЦИЯ 2](#_Toc11090323)

[1 Требования к проекту 4](#_Toc11090324)

[2 ПРОЕКТИРОВАНИЕ ЛЕКСИЧЕСКОГО АНАЛИЗАТОРА 5](#_Toc11090325)

[2.1 Исходные данные 5](#_Toc11090326)

[2.2 Принципы работы лексического анализатора 5](#_Toc11090327)

[2.3 Пример работы лексического анализатора 7](#_Toc11090328)

[3 ПРОЕКТИРОВАНИЕ СИНТАКСИЧЕСКОГО АНАЛИЗАТОРА 8](#_Toc11090329)

[3.1 Исходные данные 9](#_Toc11090330)

[3.2 Принципы работы синтаксического анализатора 9](#_Toc11090331)

[3.3 Пример синтаксического разбора 12](#_Toc11090332)

[4 ПРОЕКТИРОВАНИЕ ГЕНЕРАТОРА БАЙТ КОДА 13](#_Toc11090333)

[4.1 Исходные данные 13](#_Toc11090334)

[4.2 Принцип работы генератора байт кода 13](#_Toc11090335)

[4.3 Пример работы генератора байт-кода 17](#_Toc11090336)

[5 РАЗРАБОТКА ОПТИМИЗАТОРА 19](#_Toc11090337)

[6 ТЕСТИРОВАНИЕ ГОТОВОГО ПРОЕКТА 19](#_Toc11090338)

[6.1 Пример 1. 19](#_Toc11090339)

[6.2 Пример 2. 19](#_Toc11090340)

[6.3 Пример 3. 19](#_Toc11090341)

[6.4 Пример 4. 19](#_Toc11090342)

[ЗАКЛЮЧЕНИЕ 19](#_Toc11090343)

[Приложение А 19](#_Toc11090344)

**1 Требования к проекту**

Курсовая работа заключается в создании компилятора с заданного подмножества языка Паскаль с незначительными модификациями и упрощениями. Результатами курсовой работы являются программная реализация заданного компилятора и пояснительная записка, оформленная в соответствии с требованиями ГОСТ.

Компилятор должен быть построен из следующих составных частей:

1. лексический анализатор;
2. синтаксический анализатор;
3. оптимизатор;
4. генератор машинного или байт кода.

Входной язык компилятора должен удовлетворять следующим требованиям:

* входная программа может быть разбита на строки произвольным образом, все пробелы и переводы строки должны игнорироваться компилятором;
* текст входной программы может содержать комментарии любой длины, которые должны игнорироваться компилятором;
* входная программа должна представлять собой единый модуль, но требуется предусмотреть вызов функций;
* должны быть предусмотрены следующие варианты операторов входной программы:

1) оператор присваивания (:=);

2) условный оператор (if)

3) оператор цикла (for, while)*;*

4) арифметические операции сложения (*+*), вычитания (*-*), деления (/), умножения (\*);

5) операции сравнения меньше (*<*), больше (*>*), равно (*=*), неравно (<>), больше или равно (>=), меньше или равно (<=);

6) логические операции «и» (*and*), «или» (*or*), «нет» (*not*);

7) break-continue;

8) Для изменения приоритета операций должны использоваться круглые скобки.

* два типа данных: целый (integer) и вещественный (real);
* базовый вывод переменных (writeln);

В качестве выходного (результирующего) языка должен использоваться язык ассемблера или байт-кода, был выбран последний.

# 2 ПРОЕКТИРОВАНИЕ ЛЕКСИЧЕСКОГО АНАЛИЗАТОРА

## **2.1 Исходные данные**

Для выполнения данной части курсовой работы требуется написать программу, которая выполняет лексический анализ входного текста в соответствии с заданием и порождает таблицу лексем с указанием их типов и значений. Программа должна выдавать сообщения о наличие во входном тексте ошибок, которые могут быть обнаружены на этапе лексического анализа.

Программа должна допускать наличие комментариев неограниченной длины во входном файле, которые игнорируются в дальнейшем.

## **2.2 Принципы работы лексического анализатора**

Запущенный лексический анализатор просматривает входной поток символов программы на исходном языке, выделяя символы, входящие в требуемую лексему, до обна­ружения очередного символа, который может ограничивать лексему.

В листинге 1 приведен код метода checkCharacter(char), с помощью которого производится разбор кода входной программы на лексемы. На данном этапе организовано игнорирование идентации текста и пробелов, выделение строк, чисел и операторов, а также реализована поддержка корректной обработки однострочных и многострочных комментариев. Все токены хранятся в структуре ArrayList<Token>.

Листинг 1 – Сканирование входного потока символов

**public** **static** **void** checkCharacter(**char** element){

**if** (*comment*) {

**if** (element == '}')

*comment* = **false**;

**return**;

}

**switch** (***CHAR\_TYPE***.get(String.*valueOf*(element))){

**case** ***LETTER***:

**if** (!*readingNumber*) {

*tokenName* += element;

}

**if** (element == 'E' && *readingNumber*) {

*tokenName* += element;

*sciNotation* = **true**;

}

**break**;

**case** ***DIGIT***:

**if** (*tokenName*.isEmpty()) {

*readingNumber* = **true**;

}

*tokenName* += element;

**break**;

**case** ***SPACE***:

**if** (*readingString*){

*tokenName* += element;

} **else** **if** (*readingColon*) {

*generateToken*(***OPERATORS\_TOKEN***.get(*tokenName*));

*readingColon* = **false**;

} **else** **if** (*readingBool*) {

*generateToken*(***OPERATORS\_TOKEN***.get(*tokenName*));

*readingBool* = **false**;

} **else** **if** (!*readingNumber*) {

*tokenName* = *endOfWord*();

**if** (element == Character.*toChars*(10)[0]){

*lineRow*++;

*lineCol* = 0;

} **else** **if** (element == Character.*toChars*(9)[0]){

*lineCol*+=4;

} **else** **if** (element == Character.*toChars*(32)[0]){

*lineCol*++;

}

} **else** {

*handleNumber*();

}

**break**;

**case** ***OPERATOR***:

**if** (*readingDot* && element == '.') {

**if** (*tokenName*.equals(".")) {

*tokenName* = "";

*generateToken*("TK\_RANGE");

} **else** {

*generateToken*(*tokenName*.substring(0, *tokenName*.length()-2));

*generateToken*("TK\_DOT");

*tokenName* = "";

}

*readingDot* = **false**;

} **else** **if**(*readingString*) {

*tokenName* += element;

} **else** **if** (*readingNumber*) {

**if** (*isFloat* && element == '.') {

*isFloat* = **false**;

*tokenName* = *tokenName*.substring(0,*tokenName*.length()-1);

*handleNumber*();

*generateToken*("TK\_RANGE");

*tokenName* = "";

} **else** **if** (*sciNotation* && (element == '+' || element == '-')) {

*tokenName* += element;

} **else** **if** (element == '.') {

*isFloat* = **true**;

*tokenName* += element;

} **else** {

*handleNumber*();

*generateToken*(***OPERATORS\_TOKEN***.get(String.*valueOf*(element)));

}

} **else** **if** (*readingColon* && element == '=') {

*tokenName* += element;

*generateToken*(***OPERATORS\_TOKEN***.get(*tokenName*));

*readingColon* = **false**;

} **else** **if** (*readingBool*) {

**if** (*tokenName*.equals("<") && ((element == '=') || (element == '>'))) {

*tokenName* += element;

*generateToken*(***OPERATORS\_TOKEN***.get(*tokenName*));

} **else** **if** (*tokenName*.equals(">") && (element == '=')) {

*tokenName* += element;

*generateToken*(***OPERATORS\_TOKEN***.get(*tokenName*));

}

*readingBool* = **false**;

} **else** {

**if** (element == ';') {

*tokenName* = *endOfWord*();

*tokenName* = ";";

*generateToken*(***OPERATORS\_TOKEN***.get(String.*valueOf*(element)));

} **else** **if** (element == ':') {

*tokenName* = *endOfWord*();

*readingColon* = **true**;

*tokenName* += element;

} **else** **if** (element == '<' || element == '>') {

*tokenName* = *endOfWord*();

*readingBool* = **true**;

*tokenName* += element;

} **else** **if** (element == '.') {

*tokenName* += element;

**if** (*tokenName*.equals("end.")){

*generateToken*("TK\_END");

*generateToken*("TK\_DOT");

} **else** {

*readingDot* = **true**;

}

}

**else** **if** (***OPERATORS\_TOKEN***.containsKey(String.*valueOf*(element))) {

*tokenName* = *endOfWord*();

*tokenName* = String.*valueOf*(element);

*generateToken*(***OPERATORS\_TOKEN***.get(*tokenName*));

}

}

**break**;

**case** ***QUOTE***:

**if** (element == '{') {

*comment* = **true**;

**break**;

}

*readingString* = !*readingString*;

*tokenName* += element;

**if** (!*readingString*) {

*tokenName* = *tokenName*.substring(1, *tokenName*.length()-1);

**if** (*tokenName*.length() == 1) {

*generateToken*("TK\_CHARLIT");

} **else** **if** (*tokenName*.length() > 1) {

*generateToken*("TK\_STRLIT");

}

}

**break**;

**default**:

**throw** **new** Error("Unhandled element scanned");

}

}

Для распознавания входящих символов используются hash таблица, например для операторов (сложение, вычитание, сравнение, скобки и т.д.) используется hash таблица, указанная в листинге 2.

Листинге 2. Hash таблица операторов

**private** **static** **final** HashMap<String, String> ***OPERATORS\_TOKEN***;

**static** {

***OPERATORS\_TOKEN*** = **new** HashMap<>();

***OPERATORS\_TOKEN***.put("(", "TK\_OPEN\_PARENTHESIS");

***OPERATORS\_TOKEN***.put(")", "TK\_CLOSE\_PARENTHESIS");

***OPERATORS\_TOKEN***.put("[", "TK\_OPEN\_SQUARE\_BRACKET");

***OPERATORS\_TOKEN***.put("]", "TK\_CLOSE\_SQUARE\_BRACKET");

***OPERATORS\_TOKEN***.put(".", "TK\_DOT");

***OPERATORS\_TOKEN***.put("..", "TK\_RANGE");

***OPERATORS\_TOKEN***.put(":", "TK\_COLON");

***OPERATORS\_TOKEN***.put(";", "TK\_SEMI\_COLON");

***OPERATORS\_TOKEN***.put("+", "TK\_PLUS");

***OPERATORS\_TOKEN***.put("-", "TK\_MINUS");

***OPERATORS\_TOKEN***.put("\*", "TK\_MULTIPLY");

***OPERATORS\_TOKEN***.put("/", "TK\_DIVIDE");

***OPERATORS\_TOKEN***.put("<", "TK\_LESS\_THAN");

***OPERATORS\_TOKEN***.put("<=", "TK\_LESS\_THAN\_EQUAL");

***OPERATORS\_TOKEN***.put(">", "TK\_GREATER\_THAN");

***OPERATORS\_TOKEN***.put(">=", "TK\_GREATER\_THAN\_EQUAL");

***OPERATORS\_TOKEN***.put(":=", "TK\_ASSIGNMENT");

***OPERATORS\_TOKEN***.put(",", "TK\_COMMA");

***OPERATORS\_TOKEN***.put("=", "TK\_EQUAL");

***OPERATORS\_TOKEN***.put("<>", "TK\_NOT\_EQUAL");

}

Все зарезервированные слова (begin, end, while и т.д.) тоже находятся в hash таблице. Они берутся из файла keywords.txt.

## **2.3 Пример работы лексического анализатора**

Рассмотрим ниже пример обработки текстового файла, код которого представлен в листинге 3.

Листинг 3. Пример входного файла

program example;

var y, n, x : integer;

begin

y:=0;

while (y < 10) do

begin

y:=y+1;

if (y = 5) then begin

continue;

end;

writeln(y);

end;

end.

На выходе получим набор токенов, которые показаны в листинге 4.

Листинг 4. Набор токенов

TK\_PROGRAM

TK\_IDENTIFIER

TK\_SEMI\_COLON

TK\_VAR

TK\_IDENTIFIER

TK\_COMMA

TK\_IDENTIFIER

TK\_COMMA

TK\_IDENTIFIER

TK\_COLON

TK\_INTEGER

TK\_SEMI\_COLON

TK\_BEGIN

TK\_IDENTIFIER

TK\_ASSIGNMENT

TK\_INTLIT

TK\_SEMI\_COLON

TK\_WHILE

TK\_OPEN\_PARENTHESIS

TK\_IDENTIFIER

TK\_LESS\_THAN

TK\_INTLIT

TK\_CLOSE\_PARENTHESIS

TK\_DO

TK\_BEGIN

TK\_IDENTIFIER

TK\_ASSIGNMENT

TK\_IDENTIFIER

TK\_PLUS

TK\_INTLIT

TK\_SEMI\_COLON

TK\_IF

TK\_OPEN\_PARENTHESIS

TK\_IDENTIFIER

TK\_EQUAL

TK\_INTLIT

TK\_CLOSE\_PARENTHESIS

TK\_THEN

TK\_BEGIN

TK\_IDENTIFIER

TK\_SEMI\_COLON

TK\_END

TK\_SEMI\_COLON

TK\_WRITELN

TK\_OPEN\_PARENTHESIS

TK\_IDENTIFIER

TK\_CLOSE\_PARENTHESIS

TK\_SEMI\_COLON

TK\_END

TK\_SEMI\_COLON

TK\_END

TK\_DOT

TK\_EOF

Полученные токены будут использоваться в следующем этапе парсера – синтаксическом

анализе.

# 3 ПРОЕКТИРОВАНИЕ СИНТАКСИЧЕСКОГО АНАЛИЗАТОРА

## **3.1 Исходные данные**

После получения массива токенов, следует распарсить их, создать представление, которое будет использоваться в следующем этапе компилятора. В качестве представления используется список команд, которые генератор байт-кода будет выполнять. КС-грамматика языка приведена в Приложении А.

## **3.2 Принципы работы синтаксического анализатора**

Синтаксический анализ файла с исходным кодом и составление списка команд производится в процессе просмотра каждого токена. В листинге 5 приведен код метода funcDeclaration(), который отвечает за разбор сигнатуры функции, а именно ее типа, имени и передаваемых в нее параметров, тела функции.

Листинг 5. Разбор сигнатуры функции

**private** **static** **void** funcDeclaration() {

**if** (*currentToken*.getTokenType().equals("TK\_FUNCTION")) {

*numberfunc*++;

*match*("TK\_FUNCTION");

*currentToken*.setTokenType("TK\_A\_FUNC");

String function = *currentToken*.getTokenValue();

String functionResult = *currentToken*.getTokenValue()+"result";

*match*("TK\_A\_FUNC");

*match*("TK\_OPEN\_PARENTHESIS");

*Region* = function;

**int** kolvoparametrov = *varDeclarationsFunc*();

*match*("TK\_CLOSE\_PARENTHESIS");

*match*("TK\_COLON");

String TK\_RESULT = *currentToken*.getTokenType();

**if** (TK\_RESULT.equals("TK\_INTEGER"))

*genOpCode*(OP\_CODE.***FUNCTIONSTARTINT***);

**else** {

**if** (TK\_RESULT.equals("TK\_REAL"))

*genOpCode*(OP\_CODE.***FUNCTIONSTARTREAL***);

**else**

**throw** **new** Error("Функция '"+function+"' возвращает неизвестный результат (может быть integer или real)");

}

*getToken*();

*Region* = function;

Symbol symbolFunctionResult = **new** Symbol(functionResult,

"TK\_A\_RESULT\_VAR", *Region*,

***STRING\_TYPE\_HASH\_MAP***.get(TK\_RESULT.toLowerCase().substring(3)),

*dp*);

*dp* += 4;

**if** (SymbolTable.*lookup*(functionResult, symbolFunctionResult.getRegion()) == **null**)

SymbolTable.*insert*(symbolFunctionResult);

*match*("TK\_SEMI\_COLON");

Symbol symbolFunction = **new** Symbol(function,

"TK\_A\_FUNC", "Global",

***STRING\_TYPE\_HASH\_MAP***.get(TK\_RESULT.toLowerCase().substring(3)),

*ip*);

symbolFunction.setNumber(*numberfunc*);

symbolFunction.setAmount(kolvoparametrov);

*Region* = function;

*declarations*();

**if** (SymbolTable.*lookup*(function, "Global") == **null**)

SymbolTable.*insert*(symbolFunction);

**else**

**throw** **new** Error(String.*format*("Функция '"+function + "' уже объявлена!"));

*match*("TK\_BEGIN");

*Region* = function;

*statements*();

*match*("TK\_END");

*match*("TK\_SEMI\_COLON");

**if** (TK\_RESULT.equals("TK\_INTEGER")) {

*genOpCode*(OP\_CODE.***FUNCTIONENDINT***);

*genAddress*(symbolFunctionResult.getAddress());

}

**else**

**if** (TK\_RESULT.equals("TK\_REAL"))

*genOpCode*(OP\_CODE.***FUNCTIONENDREAL***);

}

}

В соответствии с составленной КС-грамматикой необходимо произвести разбор тела функции (листинг 6). Когда текущий токен равен одному из операторов, парсер вызывает

функцию, которая описывает текущий оператор, это может цикл for, цикл while, условный if, присваивание, вывод числа, вызов функции.

Листинг 6. Разбор тела функций

**public** **static** **void** statements(){

**while**(!*currentToken*.getTokenType().equals("TK\_END")) {

**switch** (*currentToken*.getTokenType()) {

**case** "TK\_WHILE":

*whileStat*();

**break**;

**case** "TK\_IF":

*ifStat*();

**break**;

**case** "TK\_FOR":

*forStat*();

**break**;

**case** "TK\_WRITELN":

*writeStat*();

**break**;

**case** "TK\_A\_FUNC":

*callfunc*();

**break**;

**case** "TK\_IDENTIFIER":

Symbol symbol = *findSymbol*();

**if** (symbol != **null** && (symbol.getRegion().equals(*Region*) || symbol.getRegion().equals("Global") || symbol.getRegion().equals("FUNCVAR"+*Region*)))

*currentToken*.setTokenType(symbol.getTokenType());

**else** {

**if** (*currentToken*.getTokenValue().equals("break")) {

*genOpCode*(OP\_CODE.***BREAK***);

*match*("TK\_IDENTIFIER");

}

**else**

**if** (*currentToken*.getTokenValue().equals("continue")) {

*genOpCode*(OP\_CODE.***CONTINUE***);

*match*("TK\_IDENTIFIER");

}

**else**

**throw** **new** Error(String.*format*("TK\_IDENTIFIER '"+*currentToken*.getTokenValue()+"' не объявлен"));

}

**break**;

**case** "TK\_A\_FUNC\_VAR":

*assignmentStat*();

**break**;

**case** "TK\_A\_VAR":

*assignmentStat*();

**break**;

**case** "TK\_A\_RESULT\_VAR":

*assignmentStat*();

**break**;

**case** "TK\_SEMI\_COLON":

*match*("TK\_SEMI\_COLON");

**break**;

**default**:

**return**;

}

}

}

Листинг 7. Разбор цикла for

**private** **static** **void** forStat() {

*match*("TK\_FOR");

String varName = *currentToken*.getTokenValue();

*currentToken*.setTokenType("TK\_A\_VAR");

Token token = *currentToken*;

*assignmentStat*();

Symbol symb = *findSymbol*(token);

*genOpCode*(OP\_CODE.***FORSTART***);

*genAddress*(symb.getAddress());

Symbol symbol = SymbolTable.*lookup*(varName, *Region*);

**if** (symbol != **null**) {

*match*("TK\_TO");

**if** (*currentToken*.getTokenType().equals("TK\_INTLIT")) {

*genOpCode*(OP\_CODE.***PUSHINTLIT***);

*genAddress*(Integer.*valueOf*(*currentToken*.getTokenValue()));

*genOpCode*(OP\_CODE.***FORTO***);

*match*("TK\_INTLIT");

}

**else** **if** (*currentToken*.getTokenType().equals("TK\_IDENTIFIER")) {

*genOpCode*(OP\_CODE.***PUSH***);

*genAddress*(*findSymbol*(*currentToken*).getAddress());

*genOpCode*(OP\_CODE.***FORTO***);

*match*("TK\_IDENTIFIER");

}

**else** **throw** **new** Error("Неверный тип данных для цикла for: "+*currentToken*.getTokenType());

*match*("TK\_DO");

*match*("TK\_BEGIN");

*genOpCode*(OP\_CODE.***FORBEGIN***);

*statements*();

*match*("TK\_END");

*genOpCode*(OP\_CODE.***FOREND***);

*match*("TK\_SEMI\_COLON");

}

}

Так как в программе должен быть предусмотрен вызов функций, то каждая переменная должна иметь свой собственный регион объявления. Глобальные переменные – переменные, которые объявлены в самом начале, все остальные переменные будут иметь регион, в котором они инициализированы. Каждый раз, когда парсер находит новое объявление функции, регион меняется на имя этой функции. Если в функции будет вызвана переменная, которая имеет другой регион (кроме глобального) то парсер выдаст ошибку. Поиск текущей переменной представлен в листинге 8.

Листинг 8. Поиск переменной в областях видимости

**private** **static** Symbol findSymbol() {

Symbol symbol = SymbolTable.*lookup*(*currentToken*.getTokenValue(), "FUNCVAR"+*Region*);

**if** (symbol == **null**)

symbol = SymbolTable.*lookup*(*currentToken*.getTokenValue(), *Region*);

**if** (symbol == **null**)

symbol = SymbolTable.*lookup*(*currentToken*.getTokenValue(), "Global");

**return** symbol;

}

**private** **static** Symbol findSymbol(Token token) {

Symbol symbol = SymbolTable.*lookup*(token.getTokenValue(), "FUNCVAR"+*Region*);

**if** (symbol == **null**)

symbol = SymbolTable.*lookup*(token.getTokenValue(), *Region*);

**if** (symbol == **null**)

symbol = SymbolTable.*lookup*(token.getTokenValue(), "Global");

**return** symbol;

}

Сами переменные и функции находятся в символьной таблице (листинг 9).

Листинг 9. Символьная таблица

**public** **final** **class** SymbolTable {

**static** **class** Scope {

Symbol[] symbolTable = **new** Symbol[***HASH\_TABLE\_SIZE***];

Scope next = **null**;

}

**private** **static** **final** **int** ***HASH\_TABLE\_SIZE*** = 256;

**private** **static** Scope *headerScope* = **new** Scope();

**public** **static** **void** insert(Symbol symbol) {

**int** hashValueName = *hash*(symbol.getName());

**int** hashValueRegion = *hash*(symbol.getRegion());

**int** hash = hashValueName + hashValueRegion;

Symbol bucketCursor = *headerScope*.symbolTable[hash];

**if** (bucketCursor == **null**)

*headerScope*.symbolTable[hash] = symbol;

**else** {

**while** (bucketCursor.next != **null**)

bucketCursor = bucketCursor.next;

bucketCursor.next = symbol;

}

}

**public** **static** Symbol lookup(String symbolName, String regionName) {

**int** hashValueName = *hash*(symbolName);

**int** hashValueRegion = *hash*(regionName);

**int** hash = hashValueName + hashValueRegion;

Symbol bucketCursor = *headerScope*.symbolTable[hash];

Scope scopeCursor = *headerScope*;

**while** (scopeCursor != **null**) {

**while** (bucketCursor != **null**) {

**if** (bucketCursor.getName().equals(symbolName) && bucketCursor.getRegion().equals(regionName))

**return** bucketCursor;

bucketCursor = bucketCursor.next;

}

scopeCursor = scopeCursor.next;

}

**return** **null**;

}

**public** **static** **int** hash(String symbolName) {

**int** h = 0;

**for** (**int** i = 0; i < symbolName.length(); i++)

h = h + h + symbolName.charAt(i);

h\*=197;

h = h % ***HASH\_TABLE\_SIZE***/2;

**return** h;

}

**public** **static** **void** openScope() {

Scope innerScope = **new** Scope();

innerScope.next = *headerScope*;

*headerScope* = innerScope;

}

**public** **static** **void** closeScope() {

*headerScope* = *headerScope*.next;

}

**public** **static** Scope getHeaderScope() {

**return** *headerScope*;

}

}

На выходе получаем список команд, которые в дальнейшем используются при построении байт-кода. Список команд, которые генерирует парсер:

***STARTPROGRAM***, ***FUNCTIONSTARTINT***, ***FUNCTIONSTARTREAL***, ***FUNCTIONENDINT***, ***FUNCTIONENDREAL***, ***STARTVARDECL***, ***INTVAR***, ***REALVAR***, ***COMMA***, ***ENDVARDECL***, ***PUSHVARFROMDECL***, ***HALT***, ***BREAK***, ***CONTINUE***, ***PUSHREAL***, ***PUSH***, ***PUSHFLOATLIT***, ***PUSHINTLIT***, ***PUSHINT***, ***FUNCTIONCALL***, ***FORSTART***, ***FORTO***, ***FORBEGIN***, ***FOREND***, ***WHILECMP***, ***WHILEBEGIN***, ***WHILEEND***, ***IFCMP***, ***IFTHEN***, ***IFELSE***, ***IFEND***, ***PRINT\_INT***, ***PRINT\_REAL***, ***PRINT\_NEWLINE***, ***FUNCRETURN***, ***POP***, ***AND***, ***OR***, ***PUSHVARFUNC***, ***ISCALLINT***, ***ISCALLREAL***, ***REPLACERESULT***, ***ADD***, ***XCHG***, ***CVR***, ***FADD***, ***SUB***, ***FSUB***, ***MULT***, ***FMULT***, ***FDIV***, ***DIV***, ***LSSIF***, ***LSS***, ***GTR***, ***LEQ***, ***GEQ***, ***EQL***, ***NEQL***

## **3.3 Пример синтаксического разбора**

Рассмотрим работу синтаксического анализатора при обработке файла, код которого представлен в листинге 10.

Листинг 10. Код входного файла

program example;

var y, n, x : integer;

begin

y:=0;

while (y < 10) do

begin

y:=y+1;

if (y = 5) then

begin

continue;

end;

writeln(y);

end;

end.

На выходе получим такой список команд:

OP\_CODE: STARTPROGRAM

OP\_CODE: PUSHINTLIT

OP\_CODE: POP

OP\_CODE: WHILECMP

OP\_CODE: PUSH

OP\_CODE: PUSHINTLIT

OP\_CODE: LSS

OP\_CODE: WHILEBEGIN

OP\_CODE: PUSH

OP\_CODE: PUSHINTLIT

OP\_CODE: ADD

OP\_CODE: POP

OP\_CODE: IFCMP

OP\_CODE: PUSH

OP\_CODE: PUSHINTLIT

OP\_CODE: EQL

OP\_CODE: IFTHEN

OP\_CODE: CONTINUE

OP\_CODE: IFEND

OP\_CODE: PUSH

OP\_CODE: PRINT\_INT

OP\_CODE: WHILEEND

OP\_CODE: HALT

**4 ПРОЕКТИРОВАНИЕ ГЕНЕРАТОРА БАЙТ КОДА**

## **4.1 Исходные данные**

Для выполнения заключительной части курсовой работы требуется написать программу, которая на основании списка команд синтаксического разбора порождает байт код. В качестве байт кода был выбран LLVM IR.

## **4.2 Принцип работы генератора байт кода**

Генератор байт-кода получает список команд от парсера и постепенно начинает составлять LLVM IR код. Главный метод генератора байт кода называется generate() (листинг 11), он смотрит, какую команду нужно выполнить и запускает функцию, которая записывает все операции этой команды в строку AllProgram. По достижению команды HALT, означающую конец команд, генератор прекращает свою работу. Для каждой инструкции есть свой метод, такие конструкции, как for, while, if состоят из нескольких команд, например оператор ветвления if состоит из таких команд, как IFCMP, IFTHEN, IFELSE, IFEND (Листинг 12).

Листинг 11. Метод generate(), отвечающий за выполнения команд

**public** **static** **void** generate() {

Parser.OP\_CODE opCode;

**do** {

opCode = *getOpCode*();

System.***out***.println(opCode);

**switch** (opCode) {

**case** ***PUSH***:

*push*();

**break**;

**case** ***ISCALLREAL***:

*callint* = **false**;

**break**;

**case** ***ISCALLINT***:

*callint* = **true**;

**break**;

**case** ***PUSHINT***:

*pushint*();

**break**;

**case** ***PUSHREAL***:

*pushreal*();

**break**;

**case** ***PUSHVARFUNC***:

*pushvarfunc*();

**break**;

**case** ***PUSHINTLIT***:

*pushi*();

**break**;

**case** ***PUSHFLOATLIT***:

*pushf*();

**break**;

**case** ***PUSHVARFROMDECL***:

*pushvarfromdecl*();

**break**;

**case** ***REPLACERESULT***:

*replaceresult*();

**break**;

**case** ***POP***:

*pop*();

**break**;

**case** ***CVR***:

*cvr*();

**break**;

**case** ***XCHG***:

*xchg*();

**break**;

**case** ***PRINT\_REAL***:

*printReal*();

**break**;

**case** ***PRINT\_INT***:

*printInt*();

**break**;

**case** ***PRINT\_NEWLINE***:

*printLn*();

**break**;

**case** ***HALT***:

*halt*();

**break**;

**case** ***EQL***:

*eql*(*IsIf*);

**break**;

**case** ***NEQL***:

*neql*(*IsIf*);

**break**;

**case** ***LSS***:

*less*(*IsIf*);

**break**;

**case** ***LEQ***:

*lessEql*(*IsIf*);

**break**;

**case** ***GTR***:

*greater*(*IsIf*);

**break**;

**case** ***GEQ***:

*greaterEql*(*IsIf*);

**break**;

**case** ***WHILECMP***:

*whilecmp*();

**break**;

**case** ***WHILEBEGIN***:

String label6 = *stackNumber*.pop();

*AllProgram*+=";Label %"+label6+"\n";

**break**;

**case** ***WHILEEND***:

*whileend*();

**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** ***FDIV***:

*fdiv*();

**break**;

**case** ***FUNCTIONSTARTINT***:

*startint* = **true**;

*funcstartint*();

*vardecl*="";

**break**;

**case** ***FUNCTIONSTARTREAL***:

*startint* = **false**;

*funcstartreal*();

**break**;

**case** ***FUNCTIONCALL***:

*funccall*();

**break**;

**case** ***FUNCRETURN***:

*funcreturn* = **true**;

**break**;

**case** ***FUNCTIONENDINT***:

*funcendint*();

**break**;

**case** ***FUNCTIONENDREAL***:

*funcendreal*();

**break**;

**case** ***STARTPROGRAM***:

*start*();

**break**;

**case** ***INTVAR***:

*vardecl*+="i32";

*stackNumber*.push(String.*valueOf*(*KolvoVar*));

*KolvoVar*++;

**break**;

**case** ***COMMA***:

*vardecl*+=", ";

**break**;

**case** ***REALVAR***:

*KolvoVar*++;

*stackNumber*.push(String.*valueOf*(*KolvoVar*));

*vardecl*+="double";

**break**;

**case** ***STARTVARDECL***:

*KolvoVar*=0;

**break**;

**case** ***ENDVARDECL***:

*vardecl*+="){\n";

**break**;

**case** ***FORSTART***:

*forstart*();

**break**;

**case** ***FORTO***:

*forto*();

**break**;

**case** ***FORBEGIN***:

**break**;

**case** ***FOREND***:

*forend*();

**break**;

**case** ***IFCMP***:

*IsIf*=**true**;

**break**;

**case** ***IFTHEN***:

*ifthen*();

**break**;

**case** ***IFELSE***:

*ifelse*();

**break**;

**case** ***IFEND***:

*ifend*();

**break**;

**case** ***BREAK***:

*breaK*();

**break**;

**case** ***CONTINUE***:

*continuE*();

**break**;

**default**:

**throw** **new** Error(String.*format*("Unhandled case: %s", opCode));

}

}

**while** (opCode != Parser.OP\_CODE.***HALT***);

}

Листинг 12. Генератор LLVM IR для оператора ветвления if

**private** **static** **void** ifthen() {

*poryadok*.add("then");

**if** (!*dataArrayIf*.isEmpty()) {

*dataArrayTemp* = *dataArrayIf*.pop();

*dataArrayIf*.push(*dataArrayTemp*.clone());

}

**else**

*dataArrayIf*.push(*dataArrayTemp*.clone());

*andIf*=**false**;

String temp = (String) *stackNumber*.pop();

*input*(";Label %"+temp+" ifthen\n");

}

**private** **static** **void** ifelse() {

*dataArrayTemp* = *dataArrayIf*.pop();

*dataArrayIf*.push(*dataArrayTemp*.clone());

*poryadok*.add("else");

String temp = (String) *giveMeNumberVar*();

*ifStackLabels*.push(temp);

String labelunknown = *getIfLabel*();

*input*("br label %"+labelunknown+"\n");

*input*("\n;Label %"+temp+" ifelse\n");

*ifStackLabels*.push(labelunknown);

}

**private** **static** **void** ifend() {

**if** (!*forandIf*.isEmpty()) {

String temp = (String) *giveMeNumberVar*();

**for** (**int** i=0; i<*forandIf*.size(); i++)

*AllProgram*+=temp+"\n"+*forandIf*.get(i);

*schetAnd*=-1;

*forandIf*.clear();

}

String labelexit = (String) *giveMeNumberVar*();

String criptoExitStart=*ifStackLabels*.pop();

**if** (*ifStackLabels*.isEmpty()) {

*AllProgram*=*AllProgram*.replace(criptoExitStart, labelexit);

}

**else** {

String labelElse = *ifStackLabels*.pop();

String criptoExitThen = *ifStackLabels*.pop();

*AllProgram*=*AllProgram*.replace(criptoExitStart, labelexit);

*AllProgram*=*AllProgram*.replace(criptoExitThen, labelElse);

}

**if** (*IsNotBreak* && *IsNotContinue*) {

*input*("br label %"+labelexit+"\n");

}

**else** {

**if** (!*IsNotBreak*)

*IsNotBreak* = **true**;

**if** (!*IsNotContinue*)

*IsNotContinue* = **true**;

}

*input*("\n;Label %"+labelexit+" ifend\n");

*dataArrayTemp* = *dataArrayIf*.pop();

*dataArrayIf*.push(*dataArrayTemp*.clone());

}

После завершения генерации байт-кода он записывается в файл ouput с расширением ll. После этого его можно запустить через команду lli, либо сгенерировать машинный код, используя команду llc для преобразования в объектный файл, а затем слинковать для получения исполняемого файла.

## **4.3 Пример работы генератора байт-кода**

Рассмотрим работу генератора байт-кода при обработке файла, код которого приведен в листинге 13.

Листинг 13. Пример входного файла

program example;

var y, n, x : integer;

begin

y:=0;

while (y < 10) do

begin

y:=y+1;

if (y = 5) then

begin

continue;

end;

writeln(y);

end;

end.

После завершения, генератор выдает код, который показан в листинге 14.

Листинг 14. Полученный LLVM IR код

@.strln = private unnamed\_addr constant [2 x i8] c"\0A\00"

@.strint = private unnamed\_addr constant [4 x i8] c"%i\0A\00"

@.strfloat = private unnamed\_addr constant [4 x i8] c"%f\0A\00"

declare i32 @printf(i8\*, ...)

define i32 @main() {

%1 = alloca i32

store i32 0, i32\* %1

%2 = load i32, i32\* %1

store i32 %2, i32\* %1

br label %3

;Label %3

%4 = alloca i32

store i32 10, i32\* %4

%5 = load i32, i32\* %1

%6 = load i32, i32\* %4

%7 = icmp slt i32 %5, %6

br i1 %7, label %8, label %23

;Label %8

%9 = alloca i32

store i32 1, i32\* %9

%10 = load i32, i32\* %9

%11 = load i32, i32\* %1

%12 = add i32 %10, %11

%13 = alloca i32

store i32 %12, i32\* %13

%14 = load i32, i32\* %13

store i32 %14, i32\* %1

%15 = alloca i32

store i32 5, i32\* %15

%16 = load i32, i32\* %1

%17 = load i32, i32\* %15

%18 = icmp eq i32 %16, %17

br i1 %18, label %19, label %20

;Label %19 ifthen

br label %3

;Label %20 ifend

%21 = load i32, i32\* %1

%22 = call i32 (i8\*, ...) @printf(i8\* getelementptr inbounds ([4 x i8], [4 x i8]\* @.strint, i32 0, i32 0), i32 %21)

br label %3

;Label %23

ret i32 0

}

Запуск программы с помощью команды lli показан на рисунке 1.

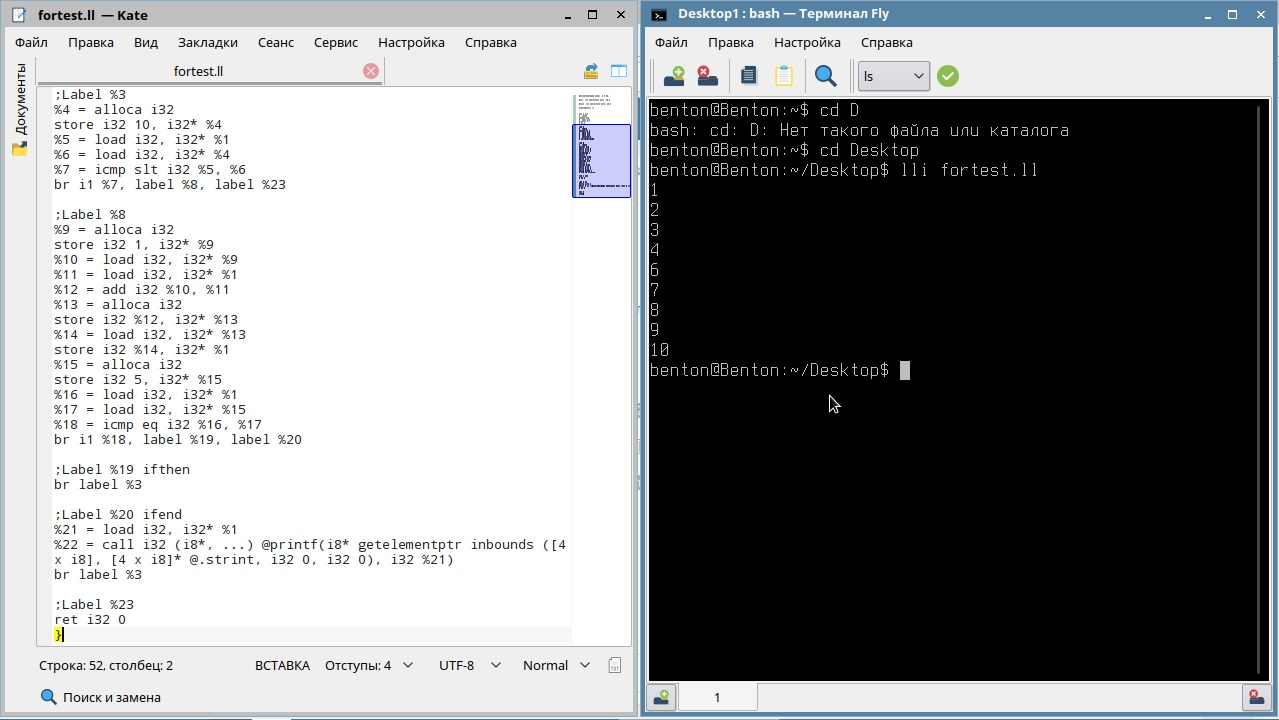


Рисунок 1. Запуск программы

Как видим, полученный код на LLVM IR компилируется без ошибок и выводит верный результат.

# 5 РАЗРАБОТКА ОПТИМИЗАТОРА

Хеллоу

# 6 ТЕСТИРОВАНИЕ ГОТОВОГО ПРОЕКТА

## **6.1 Пример 1**

Пример, который описывает все необходимые требования курсовой работы, приведен в листинге 15. Он сочетает в себе всё, что требовалось реализовать в задании курсовой работы.

Листинг 15.Пример со всеми требованиями

program example;

var x, i, k, d, e: integer; {два типа данных целочисленный}

var z, l: real; {два типа данных вещественный}

{комментарий

любой длины который

игнорируется компилятором!!!}

function test(): integer; {функция}

var y: integer;

begin

testresult:=100; {оператор присваивания}

y:=

10; {идентации не учитываются}

writeln(y); {базовый вывод переменных}

end

function test2(round : integer): integer; {функция с переменной}

begin

test2result:=0;

writeln(round);

end

begin

x:=test(); {вызов функции}

writeln(x); {функция возвращает результат}

test2(5); {вызов функции с переменной}

e:=1;

if (5 < x) then {условный оператор с else}

begin

writeln(1);

end;

else begin

writeln(0);

end;

for i:=0 to 10 do {оператор цикла for}

begin

if (i = 5) then {условный оператор без else}

begin

writeln(666);

break; {оператор break}

end;

writeln(i);

end;

k:=(50+100)\*2-(30\*1+x)\*2; {изменение приоритета операций}

writeln(k);

d:=20;

while (d > 10 ) do {оператор цикла while}

begin

writeln(d);

d:=d-1;

end;

while (e <= 16 ) do {оператор continue}

begin

e:=e\*2;

if (e = 4) then

begin continue;

end;

else

begin

writeln(e);

end;

end; z:=50.16+e\*2+100/5; {переменная real}

writeln(z);

end.

Компилятор переводит код программы из листинга 15 в LLVM IR (листинг 16).

Листинг 16. Полученный LLVM IR

@.strln = private unnamed\_addr constant [2 x i8] c"\0A\00"

@.strint = private unnamed\_addr constant [4 x i8] c"%i\0A\00"

@.strfloat = private unnamed\_addr constant [4 x i8] c"%f\0A\00"

declare i32 @printf(i8\*, ...)

define i32 @Func0(){

%1 = alloca i32

store i32 100, i32\* %1

%2 = load i32, i32\* %1

store i32 %2, i32\* %1

%3 = alloca i32

store i32 10, i32\* %3

%4 = load i32, i32\* %3

store i32 %4, i32\* %3

%5 = load i32, i32\* %3

%6 = call i32 (i8\*, ...) @printf(i8\* getelementptr inbounds ([4 x i8], [4 x i8]\* @.strint, i32 0, i32 0), i32 %5)

%7 = load i32, i32\* %1

ret i32 %7

}

define i32 @Func1(i32){

%2 = alloca i32

store i32 %0, i32\* %2

%3 = alloca i32

store i32 0, i32\* %3

%4 = load i32, i32\* %3

store i32 %4, i32\* %3

%5 = load i32, i32\* %2

%6 = call i32 (i8\*, ...) @printf(i8\* getelementptr inbounds ([4 x i8], [4 x i8]\* @.strint, i32 0, i32 0), i32 %5)

%7 = load i32, i32\* %3

ret i32 %7

}

define i32 @main() {

%1 = call i32 @Func0()

%2 = alloca i32

store i32 %1, i32\* %2

%3 = load i32, i32\* %2

store i32 %3, i32\* %2

%4 = load i32, i32\* %2

%5 = call i32 (i8\*, ...) @printf(i8\* getelementptr inbounds ([4 x i8], [4 x i8]\* @.strint, i32 0, i32 0), i32 %4)

%6 = alloca i32

store i32 5, i32\* %6

%7 = load i32, i32\* %6

%8 = call i32 @Func1(i32 %7)

%9 = alloca i32

store i32 %8, i32\* %9

%10 = alloca i32

store i32 1, i32\* %10

%11 = load i32, i32\* %10

store i32 %11, i32\* %10

%12 = alloca i32

store i32 5, i32\* %12

%13 = load i32, i32\* %12

%14 = load i32, i32\* %2

%15 = icmp slt i32 %13, %14

br i1 %15, label %16, label %20

;Label %16 ifthen

%17 = alloca i32

store i32 1, i32\* %17

%18 = load i32, i32\* %17

%19 = call i32 (i8\*, ...) @printf(i8\* getelementptr inbounds ([4 x i8], [4 x i8]\* @.strint, i32 0, i32 0), i32 %18)

br label %24

;Label %20 ifelse

%21 = alloca i32

store i32 0, i32\* %21

%22 = load i32, i32\* %21

%23 = call i32 (i8\*, ...) @printf(i8\* getelementptr inbounds ([4 x i8], [4 x i8]\* @.strint, i32 0, i32 0), i32 %22)

br label %24

;Label %24 ifend

%25 = alloca i32

store i32 0, i32\* %25

%26 = load i32, i32\* %25

store i32 %26, i32\* %25

;ForStart

%27 = load i32, i32\* %25

%28 = alloca i32

store i32 %27, i32\* %28

br label %29

;label (forcmp) %29

%30 = alloca i32

store i32 10, i32\* %30

%31 = load i32, i32\* %28

%32 = load i32, i32\* %30

%33 = icmp sle i32 %31, %32

br i1 %33, label %34, label %49

;Label (fordo) %34

%35 = alloca i32

store i32 5, i32\* %35

%36 = load i32, i32\* %28

%37 = load i32, i32\* %35

%38 = icmp eq i32 %36, %37

br i1 %38, label %39, label %43

;Label %39 ifthen

%40 = alloca i32

store i32 666, i32\* %40

%41 = load i32, i32\* %40

%42 = call i32 (i8\*, ...) @printf(i8\* getelementptr inbounds ([4 x i8], [4 x i8]\* @.strint, i32 0, i32 0), i32 %41)

br label %99

;Label %43 ifend

%44 = load i32, i32\* %28

%45 = call i32 (i8\*, ...) @printf(i8\* getelementptr inbounds ([4 x i8], [4 x i8]\* @.strint, i32 0, i32 0), i32 %44)

br label %46

;Label (+1) %46

%47 = load i32, i32\* %28

%48 = add nsw i32 %47, 1

store i32 %48, i32\* %28

br label %29

;label (forend) %49

%50 = alloca i32

store i32 50, i32\* %50

%51 = alloca i32

store i32 100, i32\* %51

%52 = load i32, i32\* %51

%53 = load i32, i32\* %50

%54 = add i32 %52, %53

%55 = alloca i32

store i32 %54, i32\* %55

%56 = alloca i32

store i32 2, i32\* %56

%57 = load i32, i32\* %55

%58 = load i32, i32\* %56

%59 = mul i32 %57, %58

%60 = alloca i32

store i32 %59, i32\* %60

%61 = alloca i32

store i32 30, i32\* %61

%62 = alloca i32

store i32 1, i32\* %62

%63 = load i32, i32\* %61

%64 = load i32, i32\* %62

%65 = mul i32 %63, %64

%66 = alloca i32

store i32 %65, i32\* %66

%67 = load i32, i32\* %2

%68 = load i32, i32\* %66

%69 = add i32 %67, %68

%70 = alloca i32

store i32 %69, i32\* %70

%71 = alloca i32

store i32 2, i32\* %71

%72 = load i32, i32\* %70

%73 = load i32, i32\* %71

%74 = mul i32 %72, %73

%75 = alloca i32

store i32 %74, i32\* %75

%76 = load i32, i32\* %60

%77 = load i32, i32\* %75

%78 = sub i32 %76, %77

%79 = alloca i32

store i32 %78, i32\* %79

%80 = load i32, i32\* %79

store i32 %80, i32\* %79

%81 = load i32, i32\* %79

%82 = call i32 (i8\*, ...) @printf(i8\* getelementptr inbounds ([4 x i8], [4 x i8]\* @.strint, i32 0, i32 0), i32 %81)

%83 = alloca i32

store i32 20, i32\* %83

%84 = load i32, i32\* %83

store i32 %84, i32\* %83

br label %85

;Label %85

%86 = alloca i32

store i32 10, i32\* %86

%87 = load i32, i32\* %83

%88 = load i32, i32\* %86

%89 = icmp sgt i32 %87, %88

br i1 %89, label %90, label %99

;Label %90

%91 = load i32, i32\* %83

%92 = call i32 (i8\*, ...) @printf(i8\* getelementptr inbounds ([4 x i8], [4 x i8]\* @.strint, i32 0, i32 0), i32 %91)

%93 = alloca i32

store i32 1, i32\* %93

%94 = load i32, i32\* %83

%95 = load i32, i32\* %93

%96 = sub i32 %94, %95

%97 = alloca i32

store i32 %96, i32\* %97

%98 = load i32, i32\* %97

store i32 %98, i32\* %83

br label %85

;Label %99

br label %100

;Label %100

%101 = alloca i32

store i32 16, i32\* %101

%102 = load i32, i32\* %10

%103 = load i32, i32\* %101

%104 = icmp sle i32 %102, %103

br i1 %104, label %105, label %121

;Label %105

%106 = alloca i32

store i32 2, i32\* %106

%107 = load i32, i32\* %10

%108 = load i32, i32\* %106

%109 = mul i32 %107, %108

%110 = alloca i32

store i32 %109, i32\* %110

%111 = load i32, i32\* %110

store i32 %111, i32\* %10

%112 = alloca i32

store i32 4, i32\* %112

%113 = load i32, i32\* %10

%114 = load i32, i32\* %112

%115 = icmp eq i32 %113, %114

br i1 %115, label %116, label %117

;Label %116 ifthen

br label %100

;Label %117 ifelse

%118 = load i32, i32\* %10

%119 = call i32 (i8\*, ...) @printf(i8\* getelementptr inbounds ([4 x i8], [4 x i8]\* @.strint, i32 0, i32 0), i32 %118)

br label %120

;Label %120 ifend

br label %100

;Label %121

%122 = alloca double

store double 50.16, double\* %122

%123 = alloca i32

store i32 2, i32\* %123

%124 = load i32, i32\* %10

%125 = load i32, i32\* %123

%126 = mul i32 %124, %125

%127 = alloca i32

store i32 %126, i32\* %127

%128 = load i32, i32\* %127

%129 = sitofp i32 %128 to double

%130 = alloca double

store double %129, double\* %130

%131 = load double, double\* %130

%132 = load double, double\* %122

%133 = fadd double %131, %132

%134 = alloca double

store double %133, double\* %134

%135 = alloca i32

store i32 100, i32\* %135

%136 = alloca i32

store i32 5, i32\* %136

%137 = load i32, i32\* %136

%138 = sitofp i32 %137 to double

%139 = alloca double

store double %138, double\* %139

%140 = load i32, i32\* %135

%141 = sitofp i32 %140 to double

%142 = alloca double

store double %141, double\* %142

%143 = load double, double\* %142

%144 = load double, double\* %139

%145 = fdiv double %143, %144

%146 = alloca double

store double %145, double\* %146

%147 = load double, double\* %146

%148 = load double, double\* %134

%149 = fadd double %147, %148

%150 = alloca double

store double %149, double\* %150

%151 = load double, double\* %150

store double %151, double\* %150

%152 = load double, double\* %150

%153 = call i32 (i8\*, ...) @printf(i8\* getelementptr inbounds ([4 x i8], [4 x i8]\* @.strfloat, i32 0, i32 0), double %152)

ret i32 0

}

Теперь полученный LLVM IR можно запустить с помощью команды lli (рисунок 2).

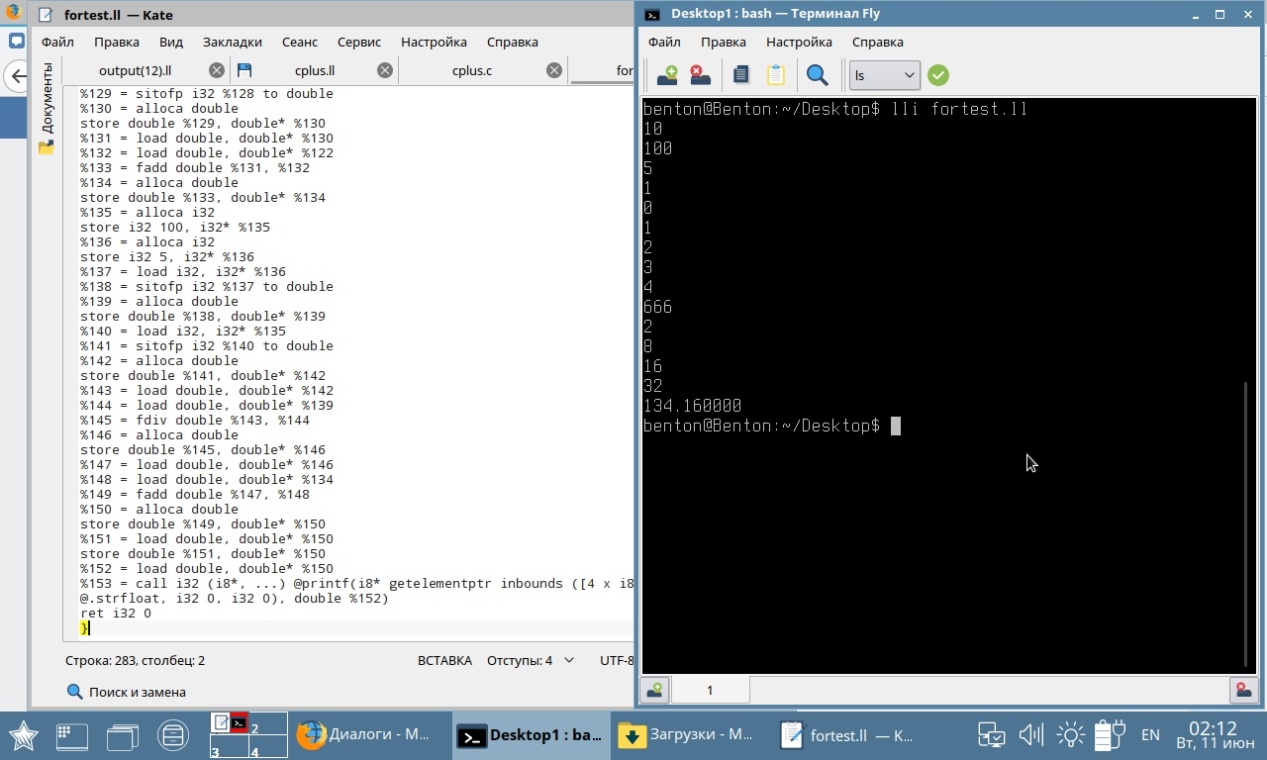


Рисунок 2. Запуск LLVM IR кода

## **6.2 Пример 2**

Компилятор умеет выполнять рекурсию, для примера возьмём рекурсию для подсчёта факториала (листинг 17).

Листинг 17. Код для подсчёта факториала

## **6.3 Пример 3**

# ЗАКЛЮЧЕНИЕ

В процессе выполнения курсовой работы была разработана программа, реализующая компилятор заданного подмножества языка Паскаль с незначительными модификациями. Для её разработки использовался язык программирования java, среда программной разработки Eclipse Oxygen.

**Приложение А**

<pascal program> ->

[<program stat>]

<declarations>

<begin-statement>

<program stat> -> E

<type> ->

integer

real

<var> ->

[varname]

<numb> ->

[intlet]

[reallet]

<num> ->

<var>

<numb>

<declarations> ->

<var decl><declarations>

<type \_\_\_\_\_\_,,\_\_\_\_\_\_>

<function \_\_\_\_\_\_,,\_\_\_\_\_\_>

-> E

<function decl> -> function <name> (params) : <type>;

<declarations>

<begin-statement>

<statement> -> <function call>

<var decl> ->

var[<namelist>: <type>;]^+

<begin\_statement> ->

begin <stats> end

<stats> ->

<while stat> ->

while <cond> <begin\_statement>

<if> ->

if <cond> then <begin\_statement>

if <cond> then <begin\_statement> else <begin\_statement>

<for> ->

for <num> to <num> do <begin\_statement>

<writeStat> ->

writeln (<num>)

<function call> ->

[functionname] (<num>,^+)

<assignment>

<var> := <num>

<cond> ->

<num> > <num>

<num> < <num>

<num> >= <num>

<num> <= <num>

<num> <> <num>

<num> and <num>

<num> || <num>