Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

высшего образования

«Владимирский государственный университет

имени Александра Григорьевича и Николая Григорьевича Столетовых»

(ВлГУ)

разработка компилятора подмножества

процедурно-ориентированного языка

Пояснительная записка

RU. 643.02068048.0001-01 81 01

# АННОТАЦИЯ

Компилятор – программный модуль, функция которого - преобразование исходного текста программы, написанного на одном из языков программирования (исходный язык), в программу на машинном языке 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/Tusya1711/Compiler->

В пояснительной записке приведены требования к разрабатываемому проекту и описаны все стадии проектирования компилятора: построение лексического и синтаксического анализаторов, построение оптимизатора, проектирование генератора байт-кода, а также приведены результаты тестирования разработанного программного модуля.

**СОДЕРЖАНИЕ**

[АННОТАЦИЯ 2](#_Toc11595973)

[СОДЕРЖАНИЕ 3](#_Toc11595974)

[1 Требования к проекту 4](#_Toc11595975)

[2 ПРОЕКТИРОВАНИЕ ЛЕКСИЧЕСКОГО АНАЛИЗАТОРА 5](#_Toc11595976)

[2.1 Исходные данные 5](#_Toc11595977)

[2.2 Принципы работы лексического анализатора 5](#_Toc11595978)

[2.3 Пример работы лексического анализатора 6](#_Toc11595979)

[3 ПРОЕКТИРОВАНИЕ СИНТАКСИЧЕСКОГО АНАЛИЗАТОРА 7](#_Toc11595980)

[3.1 Исходные данные 7](#_Toc11595981)

[3.2 Принципы работы синтаксического анализатора 7](#_Toc11595982)

[3.3 Пример синтаксического разбора 13](#_Toc11595983)

[4 ПРОЕКТИРОВАНИЕ ГЕНЕРАТОРА БАЙТ КОДА 13](#_Toc11595984)

[4.1 Исходные данные 13](#_Toc11595985)

[4.2 Принцип работы генератора байт кода 14](#_Toc11595986)

[4.3 Пример работы генератора байт-кода 17](#_Toc11595987)

[5 РАЗРАБОТКА ОПТИМИЗАТОРА 19](#_Toc11595988)

[6 ТЕСТИРОВАНИЕ ГОТОВОГО ПРОЕКТА 19](#_Toc11595989)

[6.1 Пример 1 (все требования) 19](#_Toc11595990)

[6.2 Пример 2 (рекурсия) 25](#_Toc11595991)

[ЗАКЛЮЧЕНИЕ 28](#_Toc11595992)

[Приложение А 29](#_Toc11595993)

**1 Требования к проекту**

Курсовая работа заключается в создании компилятора подмножества процедурно-ориентированного языка. В данном случае это Паскаль с некоторыми изменениями 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, идентации не должны учитываться компилятором;
* входная программа должна представлять собой единый модуль, но требуется предусмотреть вызов функций;
* текст входной программы может содержать комментарии любой длины, которые должны игнорироваться компилятором;
* должны присутствовать следующие варианты операторов входной программы:

1. оператор присваивания (:=);

2. условный оператор (if);

3. оператор цикла (for, while)*;*

4. арифметические операции: сложение (*+*), вычитание (*-*), деление (/), умножение (\*);

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.

* обязательное присутствие двух типов данных: целый (integer) и вещественный (real);
* базовый вывод переменных (writeln);

В качестве выходного языка должен использоваться язык ассемблера или байт-кода. В данном случае был выбран последний.

# 2 ПРОЕКТИРОВАНИЕ ЛЕКСИЧЕСКОГО АНАЛИЗАТОРА

## **2.1 Исходные данные**

Лексический анализатор – первая стадия работы компилятора. 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. При обнаружении ошибок во входном тексте на этапе лексического анализа программа должна выдавать сообщения об их наличии.

Во входном файле могут присутствовать комментарии произвольной длины, которые должны игнорироваться программой.

## **2.2 Принципы работы лексического анализатора**

После запуска лексического анализатора он просматривает входной поток символов программы на исходном языке и выделяет символы, входящие в требуемую лексему, до обна­ружения очередного символа, который может ограничивать лексему.

В листинге 1 приведен код хеш-таблицы HashMap<String, TYPE>, в которой описываются все буквы, цифры, пробелы и символы для комментариев, используемые данной программой. На следующем этапе организовано игнорирование идентации текста и комментариев, выделение строк, чисел и операторов. Все токены хранятся в структуре ArrayList<Token>.

Листинг 1 – Сканирование входного потока символов

**private static final** HashMap<String, TYPE> ***CHAR\_TYPE***; **static** {  
 ***CHAR\_TYPE*** = **new** HashMap<>();  
 **for** (**int** i = 65; i < 91; i++){  
 String currentChar = String.*valueOf*(Character.*toChars*(i)[0]);  
 ***CHAR\_TYPE***.put(currentChar, TYPE.***LETTER***);  
 ***CHAR\_TYPE***.put(currentChar.toLowerCase(), TYPE.***LETTER***);  
 }  
 **for** (**int** i = 48; i < 58; i++){  
 String currentChar = String.*valueOf*(Character.*toChars*(i)[0]);  
 ***CHAR\_TYPE***.put(currentChar, TYPE.***DIGIT***);  
 }  
 **for** (**int** i = 1; i < 33; i++){  
 String currentChar = String.*valueOf*(Character.*toChars*(i)[0]);  
 ***CHAR\_TYPE***.put(currentChar, TYPE.***SPACE***);  
 }  
 **for** (String key: ***OPERATORS\_TOKEN***.keySet()) {  
 ***CHAR\_TYPE***.put(key, TYPE.***OPERATOR***);  
 }  
 ***CHAR\_TYPE***.put(String.*valueOf*(Character.*toChars*(39)[0]), TYPE.***QUOTE***);  
 ***CHAR\_TYPE***.put(String.*valueOf*(Character.*toChars*(33)[0]), TYPE.***QUOTE***); ***CHAR\_TYPE***.put(String.*valueOf*(Character.*toChars*(63)[0]), TYPE.***QUOTE***); }

В данном программном модуле используется hash таблица для распознавания входящих символов. Например, для операторов (сложение, вычитание, сравнение, скобки и т.д.) используется hash таблица, указанная в листинге 2.

Листинг 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, if, else, while и т.д.) также находятся в hash таблице и 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 i, n: integer;  
begin  
n:=5;  
for i := 1 to n do begin  
writeln (2023);  
end;  
end.

На выходе получим набор токенов, продемонстрированные 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

Далее все полученные токены поступают на следующий этап – парсер.

# 3 ПРОЕКТИРОВАНИЕ СИНТАКСИЧЕСКОГО АНАЛИЗАТОРА

## **3.1 Исходные данные**

Как только массив токенов был получен, необходимо «разобрать» их, т.е. создать список команд, которые будет использоваться на следующем этапе компилятора – в генераторе байт-кода. КС-грамматика языка приведена в Приложении А.

## **3.2 Принципы работы синтаксического анализатора**

По мере просмотра каждого токена происходит синтаксический анализ файла с исходным кодом и составление списка команд. 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** {

String Row = String.*valueOf*(*currentToken*.getLineRow()+1);

String Col = String.*valueOf*(*currentToken*.getLineCol()+1);

**throw** **new** Error(String.*format*("Функция '"+function+"' возвращает неизвестный результат (может быть int или real)"+ "("+Row+" строка, "+Col+" столбец)"));

}

*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);

*arraySymbols*.add(symbolFunctionResult);

*arraySymbols*.add(**new** Symbol("Stop", **null**, **null**, TYPE.***STOP***, 0));

}

*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** {

String Row = String.*valueOf*(*currentToken*.getLineRow()+1);

String Col = String.*valueOf*(*currentToken*.getLineCol()+1);

**throw** **new** Error(String.*format*("Функция '"+function + "' уже объявлена!"+ "("+Row+" строка, "+Col+" столбец)"));

}

*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 быть цикл for, цикл while, условный if, присваивание, вывод числа или 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** {

String Row = String.*valueOf*(*currentToken*.getLineRow()+1);

String Col = String.*valueOf*(*currentToken*.getLineCol()+1);

**throw** **new** Error(String.*format*("'"+*currentToken*.getTokenValue()+"' не объявлен ("+Row+" строка, "+Col+" столбец)"));

}

}

**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**;

}

}

}

Присваивание переменной какому-либо значению состоит из нескольких этапов:

1.поиск символа текущей переменной в символьной таблице;

2. проверка на соответствие типов.

На первом этапе ищется символ и если таковой не найден, то выдаётся ошибка об отсутствии такой переменной. На втором этапе проходит проверка, т.е. нам нельзя присвоить целочисленной переменной дробную часть, а для присвоения вещественной переменной целочисленную переменную, последнюю необходимо преобразовать в вещественный тип. Функция для присваивания приведена в листинге 7.

Листинг 7. Присваивание

**public** **static** **void** assignmentStat() {

Symbol symbol = *findSymbol*();

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

TYPE lhsType = symbol.getDataType();

**int** lhsAddress = symbol.getAddress();

**if** (*currentToken*.getTokenType().equals("TK\_A\_VAR"))

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

**else**

**if** (*currentToken*.getTokenType().equals("TK\_A\_RESULT\_VAR")) {

*match*("TK\_A\_RESULT\_VAR");

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

symbol.setResult(**true**);

}

**else**

**if** (*currentToken*.getTokenType().equals("TK\_A\_FUNC\_VAR"))

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

*match*("TK\_ASSIGNMENT");

TYPE rhsType = *E*();

**if** (lhsType == rhsType) {

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

*genAddress*(lhsAddress);

}

**else** **if** (lhsType == TYPE.***R*** && rhsType == TYPE.***I***) {

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

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

*genAddress*(lhsAddress);

}

**else** {

String Row = String.*valueOf*(*currentToken*.getLineRow()+1);

String Col = String.*valueOf*(*currentToken*.getLineCol()+1);

**throw** **new** Error(String.*format*("Невозможно преобразовать тип (%s) в тип (%s ", lhsType, rhsType + ") ("+Row+" строка, "+Col+" столбец)"));

}

} **else** {

String Row = String.*valueOf*(*currentToken*.getLineRow()+1);

String Col = String.*valueOf*(*currentToken*.getLineCol()+1);

**throw** **new** Error(String.*format*("Неизвестная переменная '%s'", *currentToken*.getTokenValue()+ "("+Row+" строка, "+Col+" столбец)"));

}

}

Оператор ветвления должен начинаться с ключевого слова «if» и содержать в себе цепочку обязательных токенов. Условный оператор if может быть как полноценным (конструкция if-else), так и укороченным. Разбор оператора условного оператора ветвления if показан в листинге 8.

Листинг 8. Разбор условного оператора if

**public** **static** **void** ifStat(){

*match*("TK\_IF");

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

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

*C*();

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

*match*("TK\_THEN");

*match*("TK\_BEGIN");

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

*statements*();

*match*("TK\_END");

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

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

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

*match*("TK\_ELSE");

*match*("TK\_BEGIN");

*statements*();

*match*("TK\_END");

}

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

}

Каждой переменной необходимо иметь свой собственный ареал объявления, так как в программе должен быть предусмотрен вызов функций. Существуют глобальные переменные, т.е. переменные, которые объявлены в самом начале, и все остальные переменные, которые будут иметь ареал, в котором они инициализированы. При нахождении парсером нового объявления функции, ареал меняется на имя этой функции. Парсер выдаст ошибку, если в функции будет вызвана переменная, которая имеет другой ареал (кроме глобального). Поиск текущей переменной представлен в листинге 9.

Листинг 9. 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;

}

Все переменные и функции находятся в символьной таблице (листинг 10).

Листинг 10. 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;

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

}

**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*;

}

}

Список команд, которые в дальнейшем необходимы для построения байт-кода, будет получен на выходе синтаксического анализатора. Список команд, которые генерирует парсер, выглядит следующим образом:

***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 11.

Листинг 11. Код входного файла

program example;  
var i, n: integer;  
begin  
n:=5;  
for i := 1 to n do begin  
writeln (2023);  
end;  
end.

На выходе получим следующий список команд:

COMMAND\_LIST: STARTGLOBALVARS

COMMAND\_LIST: STARTPROGRAM

COMMAND\_LIST: PUSHINTLIT

COMMAND\_LIST: POP

COMMAND\_LIST: PUSHINTLIT

COMMAND\_LIST: POP

COMMAND\_LIST: FORSTART

COMMAND\_LIST: PUSH

COMMAND\_LIST: FORTO

COMMAND\_LIST: FORBEGIN

COMMAND\_LIST: PUSHINTLIT

COMMAND\_LIST: PRINT\_INT

COMMAND\_LIST: FOREND

COMMAND\_LIST: 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 создаёт байт код. В качестве байт кода был выбран LLVM IR.

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

Парсер направляет список команд генератору байт-кода, тот его получает и постепенно начинает составлять LLVM IR код. Главный метод генератора байт кода - generate() (листинг 12). Он смотрит, какую команду нужно выполнить и запускает функцию, которая записывает все операции этой команды в строку AllProgram. Генератор прекращает свою работу, когда достигает команды HALT, означающую конец команд. Для каждой инструкции есть свой метод. Такие конструкции, как for, while, if состоят из нескольких команд. Например, оператор условия while состоит из таких команд, как WHILEBEGIN, WHILECMP, WHILEEND. (Листинг 13).

Листинг 12. Метод generate(), отвечающий за выполнения команд

**public static void** generate() {  
 Parser.COMMAND\_LIST 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***:  
 *eqls*(*IsIf*, **"eq"**);  
 **break**;  
 **case *NEQL***:  
 *eqls*(*IsIf*, **"ne"**);  
 **break**;  
 **case *LSS***:  
 *eqls*(*IsIf*, **"slt"**);  
 **break**;  
 **case *LEQ***:  
 *eqls*(*IsIf*, **"sle"**);  
 **break**;  
 **case *GTR***:  
 *eqls*(*IsIf*, **"sgt"**);  
 **break**;  
 **case *GEQ***:  
 *eqls*(*IsIf*, **"sge"**);  
 **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***:  
 *funcstartint*();  
 *vardecl*=**""**;  
 **break**;  
 **case *FUNCTIONSTARTREAL***:  
 *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"**;  
 String varsDecl=**"%"**+String.*valueOf*(*KolvoVar*);  
 *stackNumber*.push(varsDecl);  
 *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**;  
 **case *AND***:  
 String labelAnd = *stackNumber*.pop();  
   
 *AndOr*.push(**"AND"**);  
 *AllProgram*+=**";Label "**+labelAnd+**" AND\n"**;  
 *stackNumber*.push(labelAnd);  
 **break**;  
 **case *OR***:  
 String labelOr = *stackNumber*.pop();  
   
 *AndOr*.push(**"OR"**);  
 *AllProgram*+=**";Label "**+labelOr+**" OR\n"**;  
 *stackNumber*.push(labelOr);  
 **break**;  
 **case *STARTGLOBALVARS***:  
 *startglobalvars*();  
 **break**;  
 **case *STARTFUNCVARS***:  
 *startfuncvars*();  
 **break**;  
 **default**:  
 **throw new** Error(String.*format*(**"Unhandled case: %s"**, opCode));  
 }  
 }

Листинг 13. Генератор LLVM IR для оператора условия while

**private static void** whilecmp() {  
 *IsIf* = **false**;  
 String label5 = *giveMeVar*();  
   
 *AllProgram* += **"br label "** + label5 + **"\n\n"** + **";Label "** + label5 + **"\n"**;  
 *stackNumber*.push(label5);  
}

**private static void** whileend(){  
 String label7 = *giveMeVar*();  
 String label8 = *stackNumber*.pop();  
   
 **while** (!*AndOr*.isEmpty()) {  
 String isAndisOr = *AndOr*.pop();  
 **if** (isAndisOr.equals(**"AND"**)) {  
 String whileLabel = *whileStackLabels*.pop();  
 *AllProgram*= *AllProgram*.replace(whileLabel, label8);  
 } **else   
 if** (isAndisOr.equals(**"OR"**)) {  
 String whileLabel = *whileStackLabels*.pop();  
 *AllProgram*= *AllProgram*.replace(whileLabel, label8);  
 } **else** {  
 **throw new** Error(**"Ошибка с or and"**);  
 }  
 }  
   
 *AllProgram*+=**"br label "**+label8+**"\n"**;  
 **if** (!*continueStackLabels*.isEmpty()) {  
 String replaceLabel = *continueStackLabels*.pop();  
 *AllProgram* = *AllProgram*.replace(replaceLabel, label8);  
 }  
   
 *AllProgram*+=**"\n;Label "**+label7+**"\n"**;  
 String replaceLabel3 = *whileStackLabels*.pop();  
 *AllProgram* = *AllProgram*.replace(replaceLabel3, label7);  
 **if** (!*breakStackLabels*.isEmpty()) {  
 String replaceLabel = *breakStackLabels*.pop();  
 *AllProgram* = *AllProgram*.replace(replaceLabel, label7);  
 }  
}

После завершения генерации байт-код записывается в файл ouput с расширением ll. Данный файл можно запустить с помощью команды lli либо преобразовать в объектный файл, используя команду llc, 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 14.

Листинг 14. 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 15.

Листинг 15. 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 осуществлялся с помощью сервиса COMPILER EXPLORER (<https://godbolt.org/>), что представлено на рисунке 1.

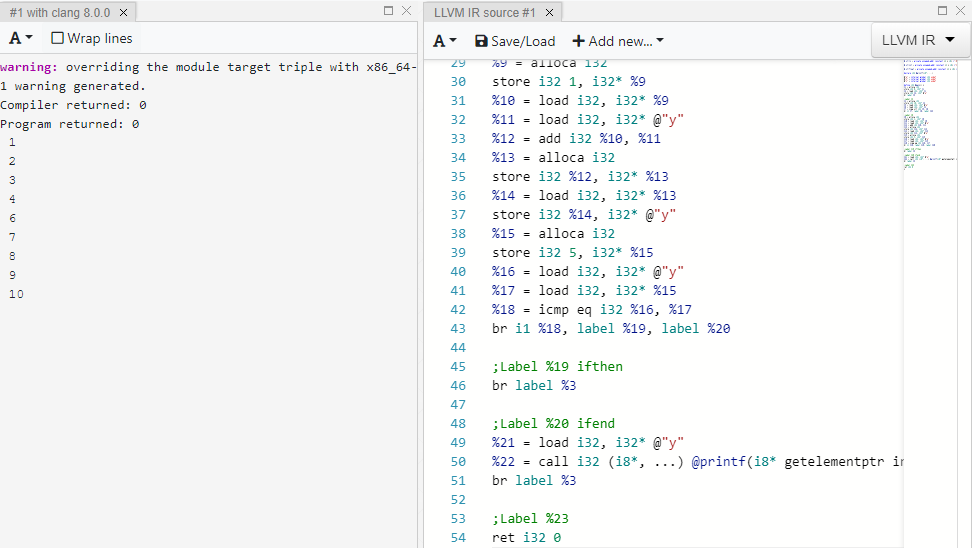


Рисунок 1. Запуск программы

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

# 5 РАЗРАБОТКА ОПТИМИЗАТОРА

Оптимизация кода начинается после генерации байт-кода. Оптимизатор удаляет из кода функции и переменные, которые были инициализированы, ни не использованы в программе.

У каждой переменной (функции) присутствует флаг, который по умолчанию опущен. Во время парсинга если переменная (функция) используется (происходит присваивание, сравнение, вывод, вызов функции или другие операции), то данный флаг поднимается. По окончании синтаксического анализа парсер смотрит, какие переменные (функции) с поднятым флагом и записывает их в массив, которые передаётся генератору кода. Все переменные (функции), имеющие опущенный флаг, больше не используются во время генерации кода.

# 6 ТЕСТИРОВАНИЕ ГОТОВОГО ПРОЕКТА

## **6.1 Пример 1 (все требования)**

Пример, который описывает все необходимые требования курсовой работы, приведен в листинге 16. Он сочетает в себе всё, что требовалось реализовать в задании курсовой работы.

Листинг 16. Пример со всеми требованиями

program example;  
var a, d, e, f, i: integer; !целочисленный тип данных?  
var c: real; !вещественный тип данных?  
function test(): integer; !функция?  
var b: integer;  
begin  
 testresult:=17; !оператор присваивания?  
 b:=  
 354; !идентации не учитываются?  
 writeln(b); !базовый вывод переменных?  
end;  
function test2(mess : integer): integer; !функция с переменной?  
begin test2result:=0;

writeln(mess);  
end;  
 !а это оооочень длинный  
 комментарий  
 и он  
 игнорируется  
 компилятором  
 АААААААААААААААААААААА ?  
begin  
 a:=test(); !вызов функции?

writeln(a); !функция возвращает результат?  
 test2(34); !вызов функции с переменной?  
 e:=1;  
 if (34 < a) then !условный оператор с else?

begin

writeln(1);  
 end;  
 else

begin

writeln(0);  
 end;  
 for i:=0 to 7 do !оператор цикла for?

begin  
 if (i = 3) then !условный оператор без else?

Begin

writeln(2605);  
 break; !оператор break?

end;  
 writeln(i);

end;  
 d:=3;  
 while (d = 18) do !оператор цикла while?

begin  
 writeln(d);

d:=d+3;

end;  
 while (e >= 24 ) do !оператор continue?

Begin

e:=e\*3;  
 if (e = 4) then

begin continue;  
 end;  
 else

begin

writeln(e);  
 end;  
 end;  
 f:=(42-6)\*a + (5+96)\*7 !изменение приоритета операций?

writeln(f);  
 c:=e\*3+240/3-3.659; !переменная real?

writeln(c);  
end.

Компилятор переводит код программы из листинга 16 в байт код LLVM IR (листинг 17).

Листинг 17. 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\*, ...)  
  
@"a" = internal global i32 undef  
@"d" = internal global i32 undef  
@"e" = internal global i32 undef  
@"f" = internal global i32 undef  
@"i" = internal global i32 undef  
@"c" = internal global double undef  
define i32 @Func0(){  
%"testresult" = alloca i32  
%"b" = alloca i32  
%1 = alloca i32  
store i32 17, i32\* %1  
%2 = load i32, i32\* %1  
store i32 %2, i32\* %"testresult"  
%3 = alloca i32  
store i32 354, i32\* %3  
%4 = load i32, i32\* %3  
store i32 %4, i32\* %"b"  
%5 = load i32, i32\* %"b"  
%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\* %"testresult"  
ret i32 %7  
}  
  
define i32 @Func1(i32){  
%2 = alloca i32  
store i32 %0, i32\* %2  
%"test2result" = alloca i32  
%3 = alloca i32  
store i32 0, i32\* %3  
%4 = load i32, i32\* %3  
store i32 %4, i32\* %"test2result"  
%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\* %"test2result"  
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\* @"a"  
%4 = load i32, i32\* @"a"  
%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 34, 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\* @"e"  
%12 = alloca i32  
store i32 34, i32\* %12  
%13 = load i32, i32\* %12  
%14 = load i32, i32\* @"a"  
%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\* @"i"  
  
;ForStart  
%27 = load i32, i32\* @"i"  
%28 = alloca i32  
store i32 %27, i32\* %28  
br label %29  
  
;label (forcmp) %29  
%30 = alloca i32  
store i32 7, 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 3, 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 2605, 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 %49  
  
;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 3, i32\* %50  
%51 = load i32, i32\* %50  
store i32 %51, i32\* @"d"  
br label %52  
  
;Label %52  
%53 = alloca i32  
store i32 18, i32\* %53  
%54 = load i32, i32\* @"d"  
%55 = load i32, i32\* %53  
%56 = icmp eq i32 %54, %55  
br i1 %56, label %57, label %66  
  
;Label %57  
%58 = load i32, i32\* @"d"  
%59 = call i32 (i8\*, ...) @printf(i8\* getelementptr inbounds ([4 x i8], [4 x i8]\* @.strint, i32 0, i32 0), i32 %58)  
%60 = alloca i32  
store i32 3, i32\* %60  
%61 = load i32, i32\* %60  
%62 = load i32, i32\* @"d"  
%63 = add i32 %61, %62  
%64 = alloca i32  
store i32 %63, i32\* %64  
%65 = load i32, i32\* %64  
store i32 %65, i32\* @"d"  
br label %52  
  
;Label %66  
br label %67  
  
;Label %67  
%68 = alloca i32  
store i32 24, i32\* %68  
%69 = load i32, i32\* @"e"  
%70 = load i32, i32\* %68  
%71 = icmp sge i32 %69, %70  
br i1 %71, label %72, label %88  
  
;Label %72  
%73 = alloca i32  
store i32 3, i32\* %73  
%74 = load i32, i32\* @"e"  
%75 = load i32, i32\* %73  
%76 = mul i32 %74, %75  
%77 = alloca i32  
store i32 %76, i32\* %77  
%78 = load i32, i32\* %77  
store i32 %78, i32\* @"e"  
%79 = alloca i32  
store i32 4, i32\* %79  
%80 = load i32, i32\* @"e"  
%81 = load i32, i32\* %79  
%82 = icmp eq i32 %80, %81  
br i1 %82, label %83, label %84  
  
;Label %83 ifthen  
br label %67  
  
;Label %84 ifelse  
%85 = load i32, i32\* @"e"  
%86 = call i32 (i8\*, ...) @printf(i8\* getelementptr inbounds ([4 x i8], [4 x i8]\* @.strint, i32 0, i32 0), i32 %85)  
br label %87  
  
;Label %87 ifend  
br label %67  
  
;Label %88  
%89 = alloca i32  
store i32 42, i32\* %89  
%90 = alloca i32  
store i32 6, i32\* %90  
%91 = load i32, i32\* %89  
%92 = load i32, i32\* %90  
%93 = sub i32 %91, %92  
%94 = alloca i32  
store i32 %93, i32\* %94  
%95 = load i32, i32\* %94  
%96 = load i32, i32\* @"a"  
%97 = mul i32 %95, %96  
%98 = alloca i32  
store i32 %97, i32\* %98  
%99 = alloca i32  
store i32 5, i32\* %99  
%100 = alloca i32  
store i32 96, i32\* %100  
%101 = load i32, i32\* %100  
%102 = load i32, i32\* %99  
%103 = add i32 %101, %102  
%104 = alloca i32  
store i32 %103, i32\* %104  
%105 = alloca i32  
store i32 7, i32\* %105  
%106 = load i32, i32\* %104  
%107 = load i32, i32\* %105  
%108 = mul i32 %106, %107  
%109 = alloca i32  
store i32 %108, i32\* %109  
%110 = load i32, i32\* %109  
%111 = load i32, i32\* %98  
%112 = add i32 %110, %111  
%113 = alloca i32  
store i32 %112, i32\* %113  
%114 = load i32, i32\* %113  
store i32 %114, i32\* @"f"  
%115 = load i32, i32\* @"f"  
%116 = call i32 (i8\*, ...) @printf(i8\* getelementptr inbounds ([4 x i8], [4 x i8]\* @.strint, i32 0, i32 0), i32 %115)  
%117 = alloca i32  
store i32 3, i32\* %117  
%118 = load i32, i32\* @"e"  
%119 = load i32, i32\* %117  
%120 = mul i32 %118, %119  
%121 = alloca i32  
store i32 %120, i32\* %121  
%122 = alloca i32  
store i32 240, i32\* %122  
%123 = alloca i32  
store i32 3, i32\* %123  
%124 = load i32, i32\* %123  
%125 = sitofp i32 %124 to double  
%126 = alloca double  
store double %125, double\* %126  
%127 = load i32, i32\* %122  
%128 = sitofp i32 %127 to double  
%129 = alloca double  
store double %128, double\* %129  
%130 = load double, double\* %129  
%131 = load double, double\* %126  
%132 = fdiv double %130, %131  
%133 = alloca double  
store double %132, double\* %133  
%134 = load i32, i32\* %121  
%135 = sitofp i32 %134 to double  
%136 = alloca double  
store double %135, double\* %136  
%137 = load double, double\* %136  
%138 = load double, double\* %133  
%139 = fadd double %137, %138  
%140 = alloca double  
store double %139, double\* %140  
%141 = alloca double  
store double 3.659, double\* %141  
%142 = load double, double\* %140  
%143 = load double, double\* %141  
%144 = fsub double %142, %143  
%145 = alloca double  
store double %144, double\* %145  
%146 = load double, double\* %145  
store double %146, double\* @"c"  
%147 = load double, double\* @"c"  
%148 = call i32 (i8\*, ...) @printf(i8\* getelementptr inbounds ([4 x i8], [4 x i8]\* @.strfloat, i32 0, i32 0), double %147)  
ret i32 0  
}

Теперь полученный LLVM IR запускаем с помощью сервиса COMPILER EXPLORER (<https://godbolt.org/>) (рисунок 2).

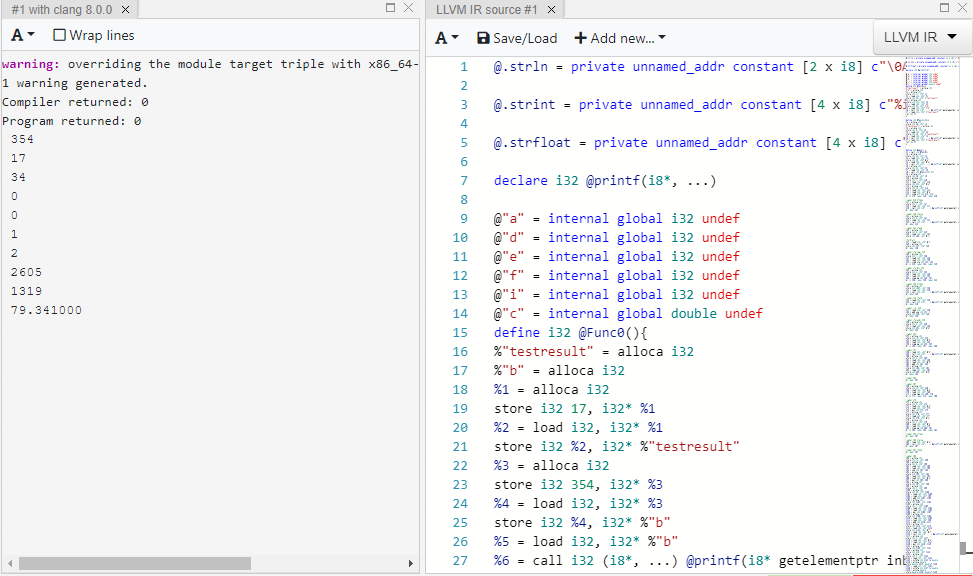


Рисунок 2. Запуск LLVM IR кода

## **6.2 Пример 2 (рекурсия)**

Компилятор умеет выполнять рекурсию, для примера возьмём рекурсию для подсчёта факториала числа 5 (листинг 18).

Листинг 18. Код для подсчёта факториала

program factorial;  
var x, f: integer;  
function fact(a : integer): integer;  
 var num1, num2: integer;  
 begin  
 if (a <= 1) then  
 begin  
 a:=1;  
 end;  
 else  
 begin  
 num1:=a-1;  
 num2:=fact(num1);  
 a:=a\*num2;  
 end;  
 factresult:=a;  
 end;  
begin  
f:=5;  
x:=fact(f);  
writeln(x);  
end.

Полученный код LLVM IR продемонстрирован в листинге 19.

Листинг 19. 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\*, ...)  
  
@"x" = internal global i32 undef  
@"f" = internal global i32 undef  
define i32 @Func0(i32){  
%2 = alloca i32  
store i32 %0, i32\* %2  
%"factresult" = alloca i32  
%"num1" = alloca i32  
%"num2" = alloca i32  
%3 = alloca i32  
store i32 1, i32\* %3  
%4 = load i32, i32\* %2  
%5 = load i32, i32\* %3  
%6 = icmp sle i32 %4, %5  
br i1 %6, label %7, label %10  
  
;Label %7 ifthen  
%8 = alloca i32  
store i32 1, i32\* %8  
%9 = load i32, i32\* %8  
store i32 %9, i32\* %2  
br label %26  
  
;Label %10 ifelse  
%11 = alloca i32  
store i32 1, i32\* %11  
%12 = load i32, i32\* %2  
%13 = load i32, i32\* %11  
%14 = sub i32 %12, %13  
%15 = alloca i32  
store i32 %14, i32\* %15  
%16 = load i32, i32\* %15  
store i32 %16, i32\* %"num1"  
%17 = load i32, i32\* %"num1"  
%18 = call i32 @Func0(i32 %17)  
%19 = alloca i32  
store i32 %18, i32\* %19  
%20 = load i32, i32\* %19  
store i32 %20, i32\* %"num2"  
%21 = load i32, i32\* %2  
%22 = load i32, i32\* %"num2"  
%23 = mul i32 %21, %22  
%24 = alloca i32  
store i32 %23, i32\* %24  
%25 = load i32, i32\* %24  
store i32 %25, i32\* %2  
br label %26  
  
;Label %26 ifend  
%27 = load i32, i32\* %2  
store i32 %27, i32\* %"factresult"  
%28 = load i32, i32\* %"factresult"  
ret i32 %28  
}  
  
  
define i32 @main() {  
%1 = alloca i32  
store i32 5, i32\* %1  
%2 = load i32, i32\* %1  
store i32 %2, i32\* @"f"  
%3 = load i32, i32\* @"f"  
%4 = call i32 @Func0(i32 %3)  
%5 = alloca i32  
store i32 %4, i32\* %5  
%6 = load i32, i32\* %5  
store i32 %6, i32\* @"x"  
%7 = load i32, i32\* @"x"  
%8 = call i32 (i8\*, ...) @printf(i8\* getelementptr inbounds ([4 x i8], [4 x i8]\* @.strint, i32 0, i32 0), i32 %7)  
ret i32 0  
}

Для проверки запускаем LLVM IR с помощью сервиса COMPILER EXPLORER и видим, что программа выдала нам результат, равный 120 (рисунок 3), что действительно является факториалом числа 5.

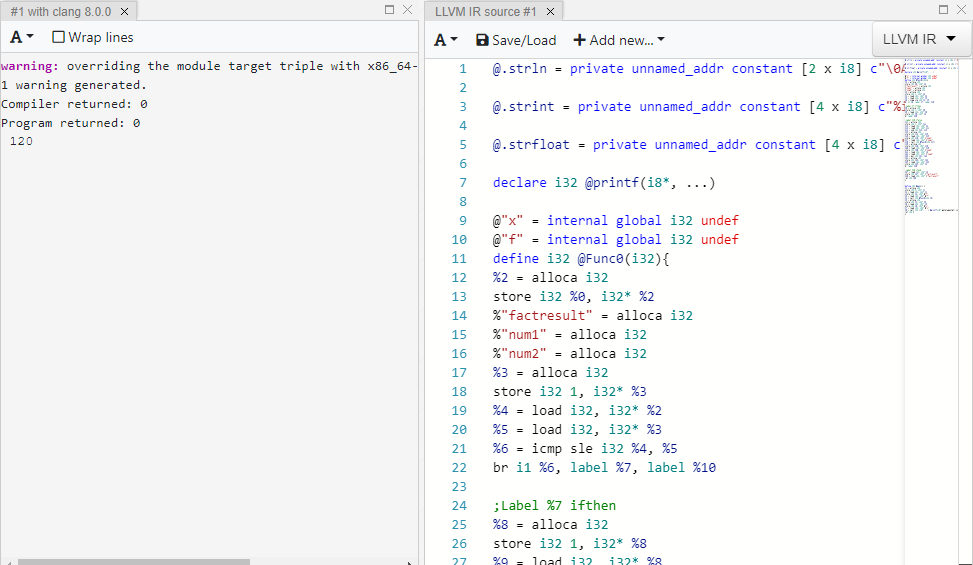


Рисунок 3. Листинг 19

Если входной файл написать с ошибкой, например, присвоить значение переменной, которая не была объявлена, компилятор выдаст ошибку, где будет описана причина её возникновения и её местоположение (строка и столбец) во входном файле (рисунок 4).

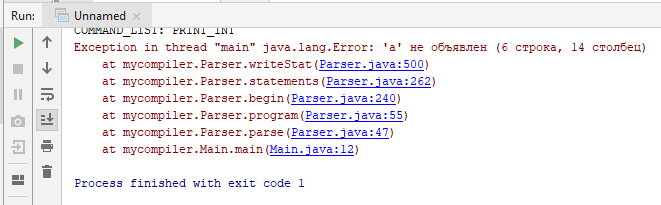


Рисунок 4. Ошибка во время парсинга

# 

# ЗАКЛЮЧЕНИЕ

В процессе выполнения курсовой работы была разработана программа, реализующая компилятор подмножества процедурно – ориентированного языка (Паскаль) с некоторыми изменениями. Для её разработки использовался язык программирования java, среда программной разработки IntelliJ IDEA Community Edition.

**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>