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

[разработка компилятора подмножества 1](#_Toc11010807)

[процедурно-ориентированного языка 1](#_Toc11010808)

[АННОТАЦИЯ 2](#_Toc11010809)

[ПРИЛОЖЕНИЕ А 22](#_Toc11010810)

[Лист регистрации изменений 23](#_Toc11010811)

**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)*;*

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 Построение 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 индентации 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 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.) также находятся в hash таблице. Они берутся из файла keywords.txt.

**2.3 Пример**

Рассмотрим ниже пример обработки текстового 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 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

**3 ПРОЕКТИРОВАНИЕ СИНТАКСИЧЕСКОГО АНАЛИЗАТОРА**

**3.1 Исходные данные**

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

Синтаксический анализ файла с исходным кодом и составление списка команд производится в процессе просмотра каждого токена. В листинге 3 приведен пример кода метода funcDeclaration(), который отвечает за разбор сигнатуры функции, а именно ее типа, имени и передаваемых в нее параметров, тело функции.

Листинг 3. 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 4). 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. Пример вызова функции для цикла for представлен в листинге 5.

Листинг 4. 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 5. 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 6.

Листинг 6. 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 7).

Листинг 7. 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 Результаты**

Рассмотрим работу синтаксического анализатора при обработке следующего 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 СОЗДАНИЕ ГЕНЕРАТОРА ОБЪЕКТНОГО КОДА**

**4.1 Исходные данные**

Для выполнения заключительной части курсовой работы требуется написать программу, которая на основании дерева синтаксического разбора порождает объектный код и затем выполняет его оптимизацию. В качестве исходного дерева синтаксического разбора рекомендуется взять дерево, которое порождает программа, построенная по заданию предыдущего раздела работы.

Результатом работы должна быть построенная на основании заданного предложения грамматики программа на объектном языке. В качестве объектного языка предлагается взять язык ассемблера для процессоров типа Intel 80x86 в реальном режиме. Все встречающиеся в исходной программе идентификаторы считать простыми скалярными переменными, не требующими выполнения преобразования типов.

Разработка данной части работы разбивается на два этапа – построение списка триад и генерация ассемблерного кода.

**4.2 Построение списка триад**

При построении списка триад производится рекурсивный проход дерева вывода, построенного синтаксическим анализатором, описанным в разделе 4. При этом вводятся следующие дополнительные типы триад:

– Триада *if* (*a*,*b*), где операнды *a* и *b* обязательно являются ссылками на триады. Смысл данной триады состоит в следующем: если результат вычисления триады *a*, являющейся логическим выражением, равен нулю, то производится переход по ссылке *b*. Иначе производится последовательный переход к следующей по списку триаде.

– Триада jmp (1, a), где первый операнд не несет смысловой нагрузки, а второй указывает ссылку на триаду, к которой на следующем этапе должен быть произведен безусловный переход.

Выражения исходного языка, не несущие семантической нагрузки, не порождают новых триад, но для таких узлов дерева вывода производится рекурсивный вызов функции построения триад.

Остальные выражения однозначно преобразуются в одну или несколько последовательных триад.

По завершении построения списка триад производится его оптимизация в виде исключения лишних операций. При этом используется еще один дополнительный тип триады – SAME (a, 0), где второй операнд не несет смысла, а первый указывает на триаду, которой идентична триада, которая была заменена на данное выражение.

**4.3 Генерация ассемблерного кода**

Генерация ассемблерного кода на основе списка триад не требует дополнительных преобразований, каждая триада может быть однозначно заменена на некоторую последовательность ассемблерных команд. При этом основной проблемой является правильное распределение регистров микропроцессора во время выполнения ассемблерного кода. Решение данного вопроса соответствует методу, предложенному в методических указаниях к курсовой работе [2].

Перед вставкой списка ассемблерных комнд, соответствующих текущей триаде, проверяется необходимость изменения содержимого аккумулятора. Если в нем уже содержится необходимое значение, его изменение является лишним, иначе первой из порожденных команд является команда *mov.* Аналогично, при необходимости сохранения результата выполнения операции (в случае, если он вызывается в следующих далее операциях), он сохраняется в одном из регистров.

**4.4 Результаты**

За генерацию списка триад отвечают модули Triad, TriListMaker и TriListOptimizer. Класс Triad определяет структуру, соответствующую одной триаде. Модуль TriListMaker генерирует первоначальный список триад на основе синтаксического дерева вывода, а модуль TriListOptimizer производит его оптимизацию в виде исключения лишних операций. Как только заканчивается оптимизация триад, результирующий список триад отправляется модулю AsmGenerator - модулю, отвечающему за генерацию ассемблерного кода. Результатом работы модуля AsmGenerator является результирующий объектный код на основе входного текстового файла. Результаты данных операций выводятся на экран при помощи модулей TriTab и AsmTab соответственно.

**6 Описание работы программы**

**Заключение**

В процессе выполнения курсовой работы была разработана программа, реализующая компилятор заданного подмножества языка Паскаль с незначительными модификациями. Для ее 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>