**Содержание**

[Введение 4](#_Toc434487177)

[1 Техническое задание 5](#_Toc434487178)

[1.1 Назначение разработки 5](#_Toc434487179)

[1.2 Требования к программе 5](#_Toc434487180)

[1.3 Стадии и этапы разработки 6](#_Toc434487181)

[2 Описание программы 8](#_Toc434487182)

[2.1 Общие сведения 8](#_Toc434487183)

[2.2 Описание логической структуры 8](#_Toc434487184)

[2.3 Структура программы 16](#_Toc434487185)

[3 Текст программы компилятора 17](#_Toc434487186)

[4 Контрольный пример 46](#_Toc434487187)

[4.1 Входная программа 46](#_Toc434487188)

[4.2 Выходная программа 47](#_Toc434487189)

[4.3 Результат работы программы 62](#_Toc434487190)

[Заключение 63](#_Toc434487191)

[Список использованных источников 64](#_Toc434487192)

# Введение

Компилятор - программный модуль, задачей которого является перевод программы, написанной на одном из языков программирования (исходный язык) в программу на язык ассемблера или язык машинных команд.

Большинство компиляторов переводят программу с некоторого высокоуровневого языка программирования в машинный код, который может быть непосредственно выполнен компьютером.

Целью курсовой работы является изучение составных частей, основных принципов построения и функционирования компиляторов, практическое освоение методов построения грамматики и простейших компиляторов.

В курсовой работе необходимо разработать грамматику языка программирования, компилятор и дополнительно реализовать возможности, предусмотренные индивидуальным вариантом.

Результатам курсовой работы является программная реализация заданного компилятора и пояснительная записка, оформленная в соответствии с требованиями стандартов и задания на курсовую работу.

# 1 Техническое задание

## 1.1 Назначение разработки

Компилятор – это программа, которая осуществляет перевод исходной программы на входном языке в эквивалентную ей объектную программу на языке машинных команд или языке ассемблере.

Основные функции компилятора:

1. проверка исходной цепочки символов на принадлежность к входному языку;
2. генерация выходной цепочки символов на языке машинных команд или ассемблере.

## 1.2 Требования к программе

1.2.1 Требования к функциональным характеристикам

Код исходной программы должен соответствовать разработанной грамматике, которая включает данные целого, логического, символьного типов, одномерный массив элементов простого типа, а также операторы: присваивание, цикл while, условный, ввода-вывода, операторные скобки и оператор Repeat по типу Паскаля.

1.2.2 Требования к надежности

В результате ввода кода программы, который не будет соответствовать грамматике, программа должна выводить сообщение об ошибке и завершаться.

1.2.3 Требования к составу и параметрам технических средств

ПК пользователя должен удовлетворять следующим требованиям:

* процессор – Intel Pentium 1,5 ГГц;
* объем оперативной памяти – 1 Гб;
* дисковая подсистема – 20 Гб.

1.2.4 Требования к информационной и программной совместимости

Данный программный продукт должен быть совместим с операционными системами Windows XP/Vista/windows 7/windows 8.1/ windows 10.

1.2.5 Требования к маркировке и упаковке

Требований к маркировке и упаковке не предъявляется.

1.2.6 Требования к транспортированию и хранению

Транспортировка и хранение программы могут быть произведены на любом носителе информации.

## 1.3 Стадии и этапы разработки

При разработке любой программы необходимо руководствоваться следующим алгоритмом работы:

1 Постановка задачи:

* сбор информации о задаче;
* формулировка условия задачи;
* определение конечных целей решения задачи;
* определение формы выдачи результатов;
* описание данных (их типов, диапазонов величин, структуры и т. п.).

2 Анализ и исследование задачи, модели:

* анализ существующих аналогов;
* анализ технических и программных средств;
* разработка математической модели;
* разработка структур данных.

3 Разработка алгоритма:

* выбор метода проектирования алгоритма;
* выбор формы записи алгоритма (блок-схемы, псевдокод и др.);
* выбор тестов и метода тестирования;
* проектирование алгоритма.

4 Программирование:

* выбор языка программирования;
* уточнение способов организации данных;
* запись алгоритма на выбранном языке программирования.

5 Тестирование и отладка:

* синтаксическая отладка;
* отладка семантики и логической структуры;
* тестовые расчеты и анализ результатов тестирования;
* совершенствование программы.

6 Анализ результатов решения задачи и уточнение в случае необходимости математической модели с повторным выполнением этапов 2-5.

7 Сопровождение программы:

* доработка программы для решения конкретных задач;
* составление документации к решенной задаче, к математической модели, к алгоритму, к программе, к набору тестов, к использованию.

Стадии разработки компилятора:

1. Разработка лексического анализатора.
2. Разработка процедур рекурсивного спуска синтаксического анализа.
3. Генерация кода и сборка компилятора.

# 2 Описание программы

## 2.1 Общие сведения

Данный программный продукт является однопроходным компилятором. Применяется для компиляции программы, написанной на разработанном языке в программу на ассемблере. Программа написана на языке программирования высокого уровня – С++ и может быть редактирована в следующих программ: Блокнот, Notepad++, Microsoft Visual Studio 2015, Dev-C++, Code::Blocks, и т.п.

## 2.2 Описание логической структуры

Основными этапами программы является разработка лексического и синтаксического анализаторов, некоторые семантические действия и генерация кода. Рассмотрим каждый из них более подробно.

2.2.1 Лексический анализ

Лексический анализ – это этап, который позволяет преобразовать исходную строку в новую, с меньшим числом правил, убрав несущественную для синтаксического анализа информацию. Для этого в исходной строке выделяют элемент подстроки – лексемы, которое образуют однотипное множество, а замен их в строке одним символом, который является именем всего множества. Он называется токеном. Токен – терминальный символ в новой грамматике.

С точки зрения теории токен образует лексемы, которые неразличимы при синтаксическом анализе.

Выделим следующие токены:

* идентификаторы – I
* ключевые слова – K
* литералы – L
* числа (без знака) – C
* разделители – R

Лексический анализатор будет производить разбор исходного текста в соответствии со следующим графом:



К ключевым словам относятся: and, array, begin, bool, break, case, char, default, else, end, end\_var, false, if, int, let, not, of, or, program, read, readln, switch, then, true, var, while, write, writeln, repeat, until.

К однолитерным разделителям относятся: «#», «+», «-», «\*», «/», «(», «)», «>», «<», «=», «[», «]», «.», «,», «:», «;», «'», «"», «!», «?», «&», «^», «|».

К двулитерным разделителям относятся: «==», «!=», «<=», «>=».

В компиляторе для каждого токена строятся таблицы. Ключевые слова и разделители строятся до вызова процедуры лексического анализа и являются постоянными, остальные же строятся в процессе. Каждая лексема вносится в таблицу только один раз. Каждый токен должен иметь атрибут – номер строки, где хранится символ в таблице.

Лексический анализатор необязательный этап компиляции, но желательный по следующим причинам:

1. Замена идентификаторов, констант, ограничителей и ключевых слов лексемами делает программу более удобной для дальнейшей обработки.
2. Лексический анализатор уменьшает длину программы, устраняя из ее исходного представления несущественный пробелы и комментарии.
3. Если будет изменена кодировка в исходном представлении программы, то это отразится только на лексическом анализаторе.
4. Входные данные.

Входными данными является текстовый документ, содержащий программу на языке, соответствующем заданной грамматике

1. Выходные данные.

Выходными данными является документ с расширением \*.asm, в котором исходная программа записана на ассемблере. Далее с помощью tasm.exe и tlink.exe получаем файл с расширением .exe.

2.2.2 Синтаксический анализ

Задача синтаксического анализатора - провести разбор текста программы, сопоставив его с эталоном разработанной грамматики. Для синтаксического разбора используются контекстно-свободные грамматики.

Один из эффективных методов синтаксического анализа – метод рекурсивного спуска. В основе метода рекурсивного спуска лежит левосторонний разбор строки языка. Исходной сентенциальной формой является начальный символ грамматики, а целевой - заданная строка языка. На каждом шаге разбора правило грамматики применяется к самому левому нетерминалу сентенции. Данный процесс соответствует построению дерева разбора цепочки сверху вниз (от корня к листьям).

Синтаксический анализатор проверяет следующую грамматику:

<программа> → ’program’ I ‘;’ <блок объявлений> <блок операторов> ’.’

<блок объявлений> → ’var’{<тип>I (‘[‘ C ‘]’ | ) {‘,’ I (‘[‘ C ‘]’ | ) } ‘;’} ‘endvar’ ‘;’

<тип> → ’bool’ | ’char’ | ’int’

<блок операторов> → ’begin’ {<оператор> ‘;’} ’end’

<оператор> → <блок операторов> | <присваивание> | <условие> | <цикл> | <ввод> | <вывод> | < repeat > |

<присваивание> → ('let' | ) I ('[' E ']' | ) '=' (E | El | L | I ('[' E ']' | ))

<условие> → ’if’ El ‘then’ <оператор> (‘else’ <оператор > | )

<цикл while> → ’while’ ‘(‘ El ‘)’ <оператор >

<ввод> → ’read’ ‘(’ I[‘[’E’]’] | ε)

<вывод> → ’write’ ‘(’ (((E | Ichar | <литерал>) {‘,’ (E | Ichar | <литерал>)}) | ) ’)’

< repeat > → ‘repeat’ <оператор> {‘,’ <оператор>} ‘until’ El

E – арифметическое выражение

E → T {+T | -T}

T → F {\*F | /F}

F → I ( ‘[‘ E ‘]’ | ) | ‘(‘ E ‘)’ | N

El – логическое выражение

El → Tl {’or’ Tl}

Tl → Fl{’and’ Fl}

Fl → I (‘[‘ E ‘]’ | ) | ‘(‘ El ‘)’ | [E Zn E] | ’true’ | ’false’ | ‘not’ Fl

Zn → ‘>’ | ‘<’ | ‘==’ | ‘!=’ | ‘<=’ | ‘>=’

2.2.3 Семантический анализ

В ходе семантическогог анализа дерево разбора обрабатывается с целью установления его семантики (смысла) — например, привязка идентификаторов к их декларациям, типам, проверка совместимости, определение типов выражений и т. д. Результат обычно называется «промежуточным представлением/кодом», и может быть дополненным деревом разбора, новым деревом, абстрактным набором команд или чем-то ещё, удобным для дальнейшей обработки.

Ниже представлены правила грамматики, но уже в аннотированном виде, т.е. с семантической нагрузкой. В квадратных скобках будут записаны семантические действия. Для генерации ассемблерного кода будет введено обозначение REC(), где в скобках будут записаны команды на языке ассемблера.

Некоторые правила не нуждаются в аннотировании, например, правила для <программа>, <блок операторов>, <оператор>, поэтому не будем записывать их повторно.

<блок объявлений> → ’var’ { <тип> I [1] (‘[‘ [2] C [3] ‘]’ | ) [5] { ‘,’ I [4] (‘[‘ [2] C [3] ‘]’ | ) [5] } ‘;’ [6] } ‘end\_var’

[1]: in = j;

[2]: t = 1;

[3]: s = Tab\_C[j]];

[4]: ip = j;

После вызова процедуры для правила <тип> в глобальную переменную a будет занесено числовое значение типа(3 – bool, 6 – char, 13 – int), а в переменные s и t размер и обозначение массива, соответственно.

[5]: if (t == 1) { Tab\_I[i].тип = 1; Tab\_I[i].размер = s; Tab\_I[i].типМассива = a; } else Tab\_I[i].тип = a;

[6]: for ( i = in; I < ip; i++) { REC (Tab\_I[i].имя);

if (Tab\_I[i].тип == 1) u=Tab\_I[i].типМассива; else u=Tab\_I[i].тип;

if (u == 13) REC (‘ dw ‘); else REC (‘ db ‘);

if (Tab\_I[i].тип == 1) REC (‘%s DUP (?)’, Tab\_I[i].размер) else REC (‘(?)’); }

<тип> → ’bool’[1] | ’char’[2] | ’int’[3]

[1]: a = 3; [2]: a = 6; [3]: a = 13;

<присваивание> → ('let' | ) [1] I [2] ('[' E [3] ']' | ) '=' (E [4]| El [5] | L [6] | I [7] ('[' E ']' [8] | ))

[1]: REC (mov di, 0);

[2]: id = j;

[3]: REC (pop di);

[4]: REC (‘shl di, 1’); REC (‘pop ax’); REC (‘mov %s[di], ax‘, Tab\_I[id].имя);

[5]: REC (pop ax); REC (‘mov %s[di], ax‘, Tab\_I[id].имя);

[6]: REC (‘mov al, %s‘, Tab\_L[j][1]); REC (‘mov %s[di], al’, Tab\_I[id].имя)

[7]: id2 = j; REC (‘mov dx, 0’);

[8]: REC (‘pop dx’); REC (‘mov ax, %s[dx]’, Tab\_I[id2]); REC (‘mov %s[di], ax‘, Tab\_I[id].имя);

<условие> → ’if’ El [1] ‘then’ <оператор > [2] (‘else’ <оператор > | ) [3]

[1]: thn = metka; els = metka; REC (‘pop ax’); REC (‘cmp ax, 0’); REC (‘jz %s‘, thn);

[2]: REC (‘jmp %s‘, els); REC (‘%s: ’, thn);

[3]: REC (‘%s: ’, els);

<цикл while> → ’while’ [1] ‘(‘ El [2] ‘)’ <оператор > [3]

[1]: a = metka; REC (‘%s: ’, a);

[2]: REC (‘pop ax’); REC (‘cmp ax, 0’); b = metka; REC(‘jz %s‘, b);

[3]: REC (‘jmp %s‘, a); REC (‘%s: ’, b);

<ввод> →[0] 'read' '(' ((I [1] ('[' E [2] ']' | ) [3] {',' I [4] ('[' E [5] ']' | ) [6]}) | ) | ')'

[0]: Подготовка – объявление буфера для чтения REC (‘@buf db 255, 0, 253 DUP (?)’);

[1], [4]: id = j; REC (‘mov di, 0’);

[2], [5]: REC (‘pop di’); if (Tab\_I[id].тип == 13) REC (‘shl di, 1’);

[3], [6]: чтение: REC (‘mov ah, 0Ah’); REC (‘lea dx, @buf’); REC (‘int21h’); затем вызывается функция преобразования символа в значение и запись: REC (‘mov %s[di], ax‘, Tab\_I[id].имя);

<вывод> → ’write’ ‘(’ (((E | Ichar | <литерал>)[1] {‘,’ (E | Ichar | <литерал>) [2]}) | ) ’)’

[1], [2]: вызов функции преобразования значения в символ и функции вывода на экран 09h.

< repeat > → ‘repeat’[1] <оператор> {‘;’ <оператор>} ‘until’ El[2]

[1]: a = metka; REC (‘%s :’, a);

[2]: REC (‘pop ax’); REC (‘cmp ax, 0’); REC (‘jne %s‘, a);

2.2.4 Генерация кода

Генерация кода – это перевод транслятором внутреннего представления исходной программы в цепочку символов выходного языка. Поскольку выходным языком компилятора (в отличие от транслятора) может быть только язык ассемблера или машинный язык, то генерация кода порождает результирующую объектную программу на языке ассемблера или непосредственно на машинном языке (в машинных кодах). Эта результирующая программа всегда представляет собой линейную последовательность команд. Поэтому генерация объектного кода в любом случае должна выполнять действия, связанные с преобразованием сложных синтаксических структур в линейные цепочки. Обычно генерация объектного кода выполняется после того как выполнен синтаксический анализ программы и все необходимые действия по подготовке к генерации кода: распределено адресное пространство под переменные и подпрограммы, проверено соответствие имён и типов переменных констант и функций в синтаксических конструкциях исходной программы и т.д.

В программе используются следующие процедуры:

generateAsmLabel – процедура для генерации уникальных меток.

lex – процедура лексического анализатора.

scan(false) – процедура, которая заглядывает в поток лексем, но не извлекает их.

scan(true) – процедура, которая извлекает лексему.

error(int errorCode, string oper) – процедура обработки ошибок.

f\_program – процедура рекурсивного спуска для <программа>.

f\_variables – процедура рекурсивного спуска для <блок объявлений>.

f\_type – процедура рекурсивного спуска для <тип>.

f\_operator – процедура рекурсивного спуска для <оператор>.

f\_operatorsBlock – процедура рекурсивного спуска для <блок операторов>.

f\_if – процедура рекурсивного спуска для <условие>.

f\_while – процедура рекурсивного спуска для <цикл>.

f\_let – процедура рекурсивного спуска для <присваивание>.

f\_read – процедура рекурсивного спуска для <ввод>.

f\_write – процедура рекурсивного спуска для <вывод>.

f\_repeat – процедура рекурсивного спуска для < repeat>.

int addIdentifierToTable(string currenIdentifier, char &token) – процедура, добавляющая идентификатор в таблицу и возвращающая пару токен-позиция.

int addLiteralToTable(string currentLiteral) – процедура для добавления литерала в таблицу, возвращающая позицию литерала.

int addNumberToTable(int currentNumber) – процедура для добавления числа в таблицу, возвращающая позицию этого числа.

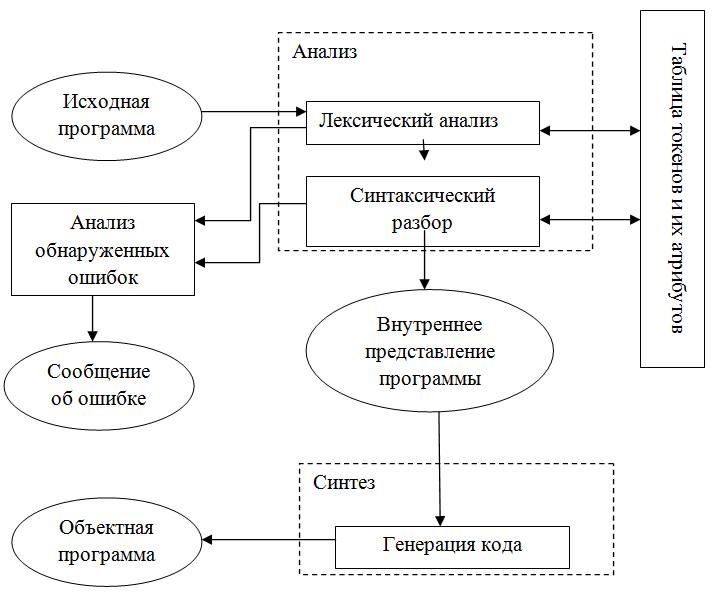
int singleDelimiter(string sDelimiter) – процедура для поиска однолитерного разделителя в таблице.

int doubleDelimiter(string dDelimiter) – процедура для поиска двулитерного разделителя в таблице.

E(); T(); F() – процедуры для разбора арифметического выражения.

El(); Tl(); Fl(); Z() – процедуры для разбора логического выражения.

## 2.3 Структура программы



# 3 Текст программы компилятора

#include <iostream>

#include <cstdio>

#include <algorithm>

#include <vector>

#include <tchar.h>

#include <string>

#include <sstream>

using namespace std;

vector<char> identifiers;

vector<char> numbers;

vector<int> FollowE;

vector<int> FollowT;

vector<int> FollowL;

vector<int> SINGLE;

vector<int> DOUBLE;

vector<int> LITERACY;

int MAX\_TOKENS = 1000;

int MAX\_SINGLE = 23;

int MAX\_DOUBLE = 4;

int MAX\_KEYWORDS = 30;

string tmp;

struct Token {

char a;

int n;

};

struct Identifier {

string name;

int identifierType;

int arrayType;

int arraySize;

};

class LexicalParser {

public:

LexicalParser();

~LexicalParser();

vector<Identifier> identifiersToken;

vector<int> numbersToken;

vector<char> singleDelimitersToken;

vector<string> literalsToken,

keywordsToken,

doubleDelimitersToken;

int totalNumbers,

totalLiterals,

totalIdentifiers;

FILE \*lex\_file, \*lex\_tokens;

fpos\_t lex\_filePosition;

Token lex(bool changePosition);

int addIdentifierToTable(string currenIdentifier, char &token);

int addLiteralToTable(string currentLiteral);

int addNumberToTable(int currentNumber);

int singleDelimiter(string sDelimiter);

int doubleDelimiter(string dDelimiter);

bool in(char x, vector<char> array);

void initializeTokens();

void error();

};

LexicalParser::LexicalParser() {

lex\_file = fopen("sourceCode.txt", "r");

lex\_tokens = fopen("tokens.txt", "w+");

totalIdentifiers = 0;

totalLiterals = 0;

totalNumbers = 0;

lex\_filePosition = 0;

initializeTokens();

}

LexicalParser::~LexicalParser() {

fclose(lex\_file);

}

Token LexicalParser::lex(bool changePosition) {

Token result;

int state = 0,

lexemeNumber;

char x, T;

string currentLexeme;

fsetpos(lex\_file, &lex\_filePosition);

result.a = 'G';

result.n = -1;

while (true) {

switch (state) {

case 0 :

x = fgetc(lex\_file);

// перевод строки (\xA, \xD) или пробел (\x20) или символ табуляции (\x9)

if ((x == '\xA') || (x == '\xD') || (x == '\x20') || (x == '\x9'))

x = fgetc(lex\_file);

if (in(x, identifiers)) // char

state = 1;

if (in(x, numbers)) // number

state = 2;

if (x == '\x27') // \x27 = '

state = 3;

if (x == '\x2F') // \x2F = /

state = 5;

if (in(x, singleDelimitersToken) && (state == 0)) // single delimiter и не ' (для literal)

state = 4;

if (feof(lex\_file)) // Конец файла

state = 6;

break;

case 1 : // identifier

while (in(x, identifiers) || in(x, numbers)) {

currentLexeme += x;

if (feof(lex\_file))

break;

x = fgetc(lex\_file);

}

fseek(lex\_file, -1, SEEK\_CUR);

lexemeNumber = addIdentifierToTable(currentLexeme, T);

result.a = T;

result.n = lexemeNumber;

state = 7;

break;

case 2 : // number

do {

currentLexeme += x;

if (feof(lex\_file))

break;

x = fgetc(lex\_file);

} while (in(x, numbers));

fseek(lex\_file, -1, SEEK\_CUR);

lexemeNumber = addNumberToTable(atoi(currentLexeme.c\_str()));

result.a = 'C';

result.n = lexemeNumber;

state = 7;

break;

case 3 : // literal

currentLexeme = x;

do {

if (feof(lex\_file))

break;

x = fgetc(lex\_file);

currentLexeme += x;

} while (x != '\x27'); // \x27 = '

currentLexeme = currentLexeme.substr(1, currentLexeme.length() - 2);

lexemeNumber = addLiteralToTable(currentLexeme);

result.a = 'L';

result.n = lexemeNumber;

state = 7;

break;

case 4 : // delimiters

currentLexeme = x;

if ((x == '>') || (x == '<') || (x == '=') || (x == '!')) { // double delimiter

x = fgetc(lex\_file);

if (x == '=') { // double

currentLexeme += x;

lexemeNumber = doubleDelimiter(currentLexeme);

if (lexemeNumber != -1) {

result.a = 'D';

result.n = lexemeNumber;

}

else

error();

}

else { // single

lexemeNumber = singleDelimiter(currentLexeme);

if (lexemeNumber != -1) {

fseek(lex\_file, -1, SEEK\_CUR);

result.a = 'R';

result.n = lexemeNumber;

}

else

error();

}

}

else { // only single

lexemeNumber = singleDelimiter(currentLexeme);

if (lexemeNumber != -1) {

result.a = 'R';

result.n = lexemeNumber;

}

else

error();

}

state = 7;

break;

case 5 : // comment

currentLexeme = x;

x = fgetc(lex\_file);

switch (x) {

case '\x2F' : // \x2F = /

do {

currentLexeme += x;

x = fgetc(lex\_file);

} while (!((x == '\xA') || feof(lex\_file)));

currentLexeme = "";

state = 0;

break;

case '\x2A': // \x2A = \*

do {

currentLexeme += x;

x = fgetc(lex\_file);

} while (!((x == '\x2F') && (currentLexeme[currentLexeme.length() - 1] == '\x2A')));

currentLexeme += x;

state = 0;

break;

default :

result.a = 'R';

result.n = 4;

fseek(lex\_file, -1, SEEK\_CUR);

state = 7;

break;

};

break;

case 6 : // end of file

result.a = 'R';

result.n = singleDelimiter(tmp = x);

state = 7;

break;

case 7 : // result

if (changePosition == true)

fgetpos(lex\_file, &lex\_filePosition);

fseek(lex\_tokens, 0, SEEK\_SET);

fputc(result.a, lex\_tokens);

fputc(result.n, lex\_tokens);

fclose(lex\_tokens);

return result;

break;

}

}

}

int LexicalParser::addIdentifierToTable(string currenIdentifier, char &token) {

int result;

bool isAdded = false,

isKeyword = false;

// search in keywords

for (int i = 0; i < MAX\_KEYWORDS; i++)

if (currenIdentifier == keywordsToken[i]) {

isKeyword = true;

result = i;

token = 'K';

break;

}

// not found in keywords

if (isKeyword == false) {

token = 'I';

for (int i = 0; i < totalIdentifiers; i++)

if (identifiersToken[i].name == currenIdentifier) {

isAdded = true;

result = i;

break;

}

// add new identifier

if (isAdded == false) {

result = totalIdentifiers;

Identifier newIdentifier;

newIdentifier.name = currenIdentifier;

identifiersToken.push\_back(newIdentifier);

totalIdentifiers++;

}

}

return result;

}

int LexicalParser::addLiteralToTable(string currentLiteral) {

int result;

bool isAdded = false;

// search in literals

for (int i = 0; i < totalLiterals; i++)

if (literalsToken[i] == currentLiteral) {

isAdded = true;

result = i;

break;

}

// add new literal

if (isAdded == false) { // not found in literals

result = totalLiterals;

literalsToken.push\_back(currentLiteral);

totalLiterals++;

}

return result;

}

int LexicalParser::addNumberToTable(int currentNumber) {

int result;

bool isAdded = false;

// search in numbers

for (int i = 0; i < totalNumbers; i++)

if (numbersToken[i] == currentNumber) {

isAdded = true;

result = i;

break;

}

// add new number

if (isAdded == false) { // not found

result = totalNumbers;

numbersToken.push\_back(currentNumber);

totalNumbers++;

}

return result;

}

int LexicalParser::singleDelimiter(string sDelimiter) {

for (int i = 0; i < MAX\_SINGLE; i++)

if (singleDelimitersToken[i] == sDelimiter[0])

return i;

return -1;

}

int LexicalParser::doubleDelimiter(string dDelimiter) {

for (int i = 0; i < MAX\_DOUBLE; i++)

if (doubleDelimitersToken[i] == dDelimiter)

return i;

return -1;

}

bool LexicalParser::in(char x, vector<char> array) {

return (find(array.begin(), array.end(), x) != array.end());

}

void LexicalParser::initializeTokens() {

char\* line;

FILE\* strm;

// request a change in capacity

keywordsToken.reserve(MAX\_KEYWORDS);

singleDelimitersToken.reserve(MAX\_SINGLE);

doubleDelimitersToken.reserve(MAX\_DOUBLE);

// load from file double delimiters

strm = fopen("doubleDelimiters.txt", "r");

while (fgets(line, 1024, strm)) {

string tmp = line;

doubleDelimitersToken.push\_back(tmp.substr(0, tmp.length() - 1));

}

fclose(strm);

// load from file single delimiters

strm = fopen("singleDelimiters.txt", "r");

while (fgets(line, 1024, strm)) {

char tmp = line[0];

singleDelimitersToken.push\_back(tmp);

}

fclose(strm);

// load from file keywords

strm = fopen("keywords.txt", "r");

while (fgets(line, 1024, strm)) {

string tmp = line;

keywordsToken.push\_back(tmp.substr(0, tmp.length() - 1));

}

fclose(strm);

// init elements identifiers and numbers

identifiers.push\_back('\_');

for (char i = 'a'; i <= 'z'; i++)

identifiers.push\_back(i);

for (char i = 'A'; i <= 'Z'; i++)

identifiers.push\_back(i);

for (char i = '0'; i <= '9'; i++)

numbers.push\_back(i);

// follow E arithmetic

FollowE.push\_back(6);

FollowE.push\_back(7);

FollowE.push\_back(8);

FollowE.push\_back(9);

FollowE.push\_back(11);

FollowE.push\_back(13);

FollowE.push\_back(14);

FollowE.push\_back(15);

FollowE.push\_back(16);

// follow T arithmetic

FollowT.push\_back(1);

FollowT.push\_back(2);

FollowT.push\_back(6);

FollowT.push\_back(7);

FollowT.push\_back(8);

FollowT.push\_back(9);

FollowT.push\_back(11);

FollowT.push\_back(13);

FollowT.push\_back(14);

FollowT.push\_back(15);

FollowT.push\_back(16);

// follow logic

FollowL.push\_back(6);

FollowL.push\_back(11);

FollowL.push\_back(13);

FollowL.push\_back(15);

//

SINGLE.push\_back(7);

SINGLE.push\_back(8);

SINGLE.push\_back(9);

SINGLE.push\_back(18);

//

DOUBLE.push\_back(0);

DOUBLE.push\_back(1);

DOUBLE.push\_back(2);

DOUBLE.push\_back(3);

//

LITERACY.push\_back(2);

LITERACY.push\_back(4);

LITERACY.push\_back(8);

LITERACY.push\_back(9);

LITERACY.push\_back(10);

LITERACY.push\_back(12);

LITERACY.push\_back(16);

LITERACY.push\_back(19);

LITERACY.push\_back(20);

LITERACY.push\_back(21);

LITERACY.push\_back(22);

LITERACY.push\_back(23);

LITERACY.push\_back(25);

LITERACY.push\_back(26);

LITERACY.push\_back(27);

//

for (int i = 0; i < 0; i++) {}

}

void LexicalParser::error() {

cout << "Incorrect symbol" << endl;

exit(1);

}

class SyntaxParser {

public:

SyntaxParser();

~SyntaxParser();

char t\_a;

int t\_n,

labelCount,

cmpType,

currenIdentifier,

baseType,

arraySize;

FILE \*asm\_code, \*asm\_variables;

void scan(bool arg);

bool inFollow(int x, vector<int> array);

string generateAsmLabel();

void E();

void T();

void F();

void EL();

void TL();

void FL();

void Z();

void f\_program();

void f\_variables();

void f\_type();

void f\_operatorsBlock();

void f\_operator();

void f\_if();

void f\_let();

void f\_while();

void f\_read(bool ln);

void f\_readInt(int idenNum);

void f\_readBool(int idenNum);

void f\_readChar(int idenNum);

void f\_write(bool ln);

void f\_writeInt();

void f\_writeLiteral();

void f\_writeChar(int idenNum);

void f\_writeBool();

void error(int errorCode, string oper);

void f\_repeat();

void compile();

LexicalParser lexicalParser;

};

SyntaxParser::SyntaxParser() {

labelCount = 0;

asm\_code = fopen("asmKod.asm", "w+");

asm\_variables = fopen("asmData.asm", "w+");

}

SyntaxParser::~SyntaxParser() {

fclose(asm\_code);

fclose(asm\_variables);

}

void SyntaxParser::scan(bool arg) {

Token scannedToken = lexicalParser.lex(arg);

t\_a = scannedToken.a;

t\_n = scannedToken.n;

}

bool SyntaxParser::inFollow(int x, vector<int> array) {

return (find(array.begin(), array.end(), x) != array.end());

}

string SyntaxParser::generateAsmLabel() {

stringstream ss;

ss << labelCount++;

return "@" + ss.str();

}

// E -> T {+T | -T}

void SyntaxParser::E() {

T(); // T

scan(false);

while (!((t\_a == 'R') && inFollow(t\_n, FollowE)))

if ((t\_a == 'R') && (t\_n == 1)) { // +

scan(true);

T(); // T

fprintf(asm\_code, " pop bx \n");

fprintf(asm\_code, " pop ax \n");

fprintf(asm\_code, " add ax, bx \n");

fprintf(asm\_code, " push ax \n");

} else if ((t\_a == 'R') && (t\_n == 2)) { // -

scan(true);

T(); // T

fprintf(asm\_code, " pop bx \n");

fprintf(asm\_code, " pop ax \n");

fprintf(asm\_code, " sub ax, bx \n");

fprintf(asm\_code, " push ax \n");

} else if (!( ((t\_a == 'R') && inFollow(t\_n, SINGLE)) || ((t\_a == 'D') && inFollow(t\_n, DOUBLE)) || ((t\_a == 'K') && inFollow(t\_n, LITERACY)) ))

error(2, "E arithmetic");

else

break;

}

// T -> F {\*F | /F}

void SyntaxParser::T() {

F(); // F

scan(false);

while (!((t\_a == 'R') && inFollow(t\_n, FollowT)))

if ((t\_a == 'R') && (t\_n == 3)) { // \*

scan(true);

F(); // F

scan(false);

fprintf(asm\_code, " pop bx \n");

fprintf(asm\_code, " pop ax \n");

fprintf(asm\_code, " mul bx \n");

fprintf(asm\_code, " push ax \n");

} else if ((t\_a == 'R') && (t\_n == 4)) { // /

scan(true);

F(); // F

scan(false);

fprintf(asm\_code, " pop bx \n");

fprintf(asm\_code, " pop ax \n");

fprintf(asm\_code, " xor dx, dx \n");

fprintf(asm\_code, " div bx \n");

fprintf(asm\_code, " push ax \n");

} else if (!( ((t\_a == 'R') && inFollow(t\_n, SINGLE)) || ((t\_a == 'D') && inFollow(t\_n, DOUBLE)) || ((t\_a == 'K') && inFollow(t\_n, LITERACY)) ))

error(3, "T arithmetic");

else

break;

}

// F -> '(' E ')' | I ('[' E ']' | eps) | C

void SyntaxParser::F() {

int idenNum;

scan(false);

if (t\_a == 'C') { // C

scan(true);

fprintf(asm\_code, " mov ax, %d \n", lexicalParser.numbersToken[t\_n]);

fprintf(asm\_code, " push ax \n");

} else if (t\_a == 'I') { // I ('[' E ']')

fprintf(asm\_code, " mov di, 0 \n");

idenNum = t\_n;

scan(true);

scan(false);

if ((t\_a == 'R') && (t\_n == 10)) { // '[' E ']'

scan(true);

E();

fprintf(asm\_code, " pop di \n");

fprintf(asm\_code, " shl di, 1 \n");

scan(true);

if (!((t\_a == 'R') && (t\_n == 11))) // ']'

error(4, "F arithmetic");

}

if ((lexicalParser.identifiersToken[idenNum].identifierType == 3) || (lexicalParser.identifiersToken[idenNum].arrayType == 3)) { // bool

fprintf(asm\_code, " xor ax, ax \n");

fprintf(asm\_code, " mov al, %s[di] \n", lexicalParser.identifiersToken[idenNum].name.c\_str());

} else

fprintf(asm\_code, " mov ax, %s[di] \n", lexicalParser.identifiersToken[idenNum].name.c\_str());

fprintf(asm\_code, " push ax \n");

} else if ((t\_a == 'R') && (t\_n == 5)) { // '(' E ')'

scan(true);

E();

scan(true);

if (!((t\_a == 'R') && (t\_n == 6))) // ')'

error(5, "F");

}

}

// El -> Tl {'or' Tl}

void SyntaxParser::EL() {

TL();

scan(false);

while (!(((t\_a == 'R') && inFollow(t\_n, FollowL)) || ((t\_a == 'K') && ((t\_n == 8) || (t\_n == 22))))) // K8 = else; K22 = then

if ((t\_a == 'K') && (t\_n == 17)) { // K17 = or

scan(true);

TL();

fprintf(asm\_code, " pop bx \n");

fprintf(asm\_code, " pop ax \n");

fprintf(asm\_code, " or ax, bx \n");

fprintf(asm\_code, " push ax \n");

} else

error(6, "E logic");

}

// Tl -> Fl {'and' Fl}

void SyntaxParser::TL() {

FL();

scan(false);

while (!(((t\_a == 'R') && inFollow(t\_n, FollowL)) || ((t\_a == 'K') && ((t\_n == 8) || (t\_n == 17) || (t\_n == 22))))) // K8 = else; K17 = or; K22 = then

if ((t\_a == 'K') && (t\_n == 0)) { // K1 = and

scan(true);

FL();

fprintf(asm\_code, " pop bx \n");

fprintf(asm\_code, " pop ax \n");

fprintf(asm\_code, " and ax, bx \n");

fprintf(asm\_code, " push ax \n");

}

}

// Fl -> 'true' | 'false' | 'not' Fl | '(' El ')' | '[' E Zn E ']' | I ('[' E ']' | eps)

void SyntaxParser::FL() {

int idenNum;

string trueLabel, falseLabel;

scan(false);

if (t\_a == 'C')

error(30, "F logic");

// 'true' | 'false' | I ('[' E ']' | eps)

if (((t\_a == 'K') && ((t\_n == 11) || (t\_n == 23))) || (t\_a == 'I')) { // K11 = false; K23 = true

scan(true);

if (t\_a == 'K') // 'true' | 'false'

switch (t\_n) {

case 11 : fprintf(asm\_code, " push 0 \n"); break; // 'false'

case 23 : fprintf(asm\_code, " push 1 \n"); break; // 'true'

}

if (t\_a == 'I') { // I ('[' E ']' | eps)

fprintf(asm\_code, " mov di, 0 \n");

idenNum = t\_n;

scan(false);

if ((t\_a == 'R') && (t\_n == 10)) { // '[' E ']'

scan(true);

E();

fprintf(asm\_code, " pop di \n");

scan(true);

if (!((t\_a == 'R') && (t\_n == 11))) // ']'

error(4, "F logic");

}

fprintf(asm\_code, " xor ah, ah \n");

fprintf(asm\_code, " mov al, %s[di] \n", lexicalParser.identifiersToken[idenNum].name.c\_str());

fprintf(asm\_code, " push ax \n");

}

} else {

if ((t\_a == 'K') && (t\_n == 15)) { // 'not' Fl

scan(true);

FL();

fprintf(asm\_code, " pop ax \n");

fprintf(asm\_code, " not ax \n");

fprintf(asm\_code, " push ax \n");

} else {

if ((t\_a == 'R') && (t\_n == 10)) { // '[' E Zn E ']'

scan(true);

E();

Z();

E();

trueLabel = generateAsmLabel();

falseLabel = generateAsmLabel();

fprintf(asm\_code, " pop bx \n");

fprintf(asm\_code, " pop ax \n");

fprintf(asm\_code, " cmp ax, bx \n");

switch (cmpType) {

case 0 : fprintf(asm\_code, " je %s \n", trueLabel.c\_str()); break; // ==

case 1 : fprintf(asm\_code, " jne %s \n", trueLabel.c\_str()); break; // !=

case 2 : fprintf(asm\_code, " jbe %s \n", trueLabel.c\_str()); break; // <=

case 3 : fprintf(asm\_code, " jae %s \n", trueLabel.c\_str()); break; // >=

case 7 : fprintf(asm\_code, " ja %s \n", trueLabel.c\_str()); break; // >

case 8 : fprintf(asm\_code, " jb %s \n", trueLabel.c\_str()); break; // <

}

fprintf(asm\_code, " push 0 \n");

fprintf(asm\_code, " jmp %s \n", falseLabel.c\_str());

fprintf(asm\_code, "%s: \n", trueLabel.c\_str());

fprintf(asm\_code, " push 1 \n");

fprintf(asm\_code, "%s: \n", falseLabel.c\_str());

scan(true);

if (!((t\_a == 'R') && (t\_n == 11))) // ']'

error(4, "F logic");

} else if ((t\_a == 'R') && (t\_n == 5)) { // '(' El ')'

scan(true);

EL();

scan(true);

if (!((t\_a == 'R') && (t\_n == 6))) // ')'

error(5, "F logic");

}

}

}

}

// Zn -> '>' | '<' | '==' | '!=' | '>=' | '<='

void SyntaxParser::Z() {

scan(false);

if (((t\_a == 'R') && inFollow(t\_n, SINGLE)) || ((t\_a == 'D') && inFollow(t\_n, DOUBLE))) { // '>' | '<' | '==' | '!=' | '>=' | '<='

cmpType = t\_n;

scan(true);

} else

error(8, "Z logic");

}

// <program> -> 'program' I ';' <variables> <operators\_block> '.'

void SyntaxParser::f\_program() {

fprintf(asm\_code, " .model small \n");

fprintf(asm\_code, " include asmData.asm \n");

scan(true);

if (!((t\_a == 'K') && (t\_n == 18))) // 'program'

error(9, "program");

scan(true);

if (!(t\_a == 'I')) // I

error(10, "program");

scan(true);

if (!((t\_a == 'R') && (t\_n == 15))) // ';'

error(16, "program");

f\_variables(); // <variables>

fprintf(asm\_code, " .stack 100h \n");

fprintf(asm\_code, " .code \n");

fprintf(asm\_code, " .386 \n");

fprintf(asm\_code, "start: \n");

fprintf(asm\_code, " mov ax, @data \n");

fprintf(asm\_code, " mov ds, ax \n");

f\_operatorsBlock();

scan(true);

if (!((t\_a == 'R') && (t\_n == 12))) // '.'

error(11, "program");

fprintf(asm\_code, "error: \n");

fprintf(asm\_code, " mov ax, 4C00h \n");

fprintf(asm\_code, " int 21h \n");

fprintf(asm\_code, " end start \n");

}

// <variables> -> 'var' { <type> I ('[' C ']' | eps) { ',' I ('[' C ']' | eps) } ';' } 'endvar' ';'

void SyntaxParser::f\_variables() {

int u;

currenIdentifier = 0;

int varListBegin = 0;

int varListEnd = 0;

fprintf(asm\_variables, " .data \n");

fprintf(asm\_variables, "; SUPPORTING VARIABLES \n");

fprintf(asm\_variables, " @buffer db 6 \n");

fprintf(asm\_variables, " blength db (?) \n");

fprintf(asm\_variables, " @buf db 256 DUP (?) \n");

fprintf(asm\_variables, " clrf db 0Dh, 0Ah, \"$\" \n");

fprintf(asm\_variables, " output db 6 DUP (?), \"$\" \n");

fprintf(asm\_variables, " err\_msg db \"Input error, try again\", 0Dh, 0Ah, \"$\" \n");

fprintf(asm\_variables, " @true db \"true\" \n");

fprintf(asm\_variables, " @@true db \"true$\" \n");

fprintf(asm\_variables, " @false db \"false\" \n");

fprintf(asm\_variables, " @@false db \"false$\" \n");

fprintf(asm\_variables, "; USING VARIABLES \n");

scan(true);

if (!((t\_a == 'K') && (t\_n == 24))) // var

error(12, "var");

// { <type> I ('[' C ']' | eps) { ',' I ('[' C ']' | eps) } ';' }

scan(false);

while (!((t\_a == 'K') && (t\_n == 10))) { // 'endvar'

f\_type();

scan(true);

if (t\_a != 'I') // I

error(13, "var");

currenIdentifier++;

if (t\_n != currenIdentifier)

error(1, "var 1");

varListBegin = t\_n;

varListEnd = t\_n;

scan(false);

if ((t\_a == 'R') && (t\_n == 10)) { // ('[' C ']' | eps)

scan(true); // '['

scan(true);

if (t\_a != 'C') // 'C'

error(19, "arrayType");

arraySize = lexicalParser.numbersToken[t\_n];

scan(true);

if (!((t\_a == 'R') && (t\_n == 11))) // ']'

error(4, "arrayType");

lexicalParser.identifiersToken[currenIdentifier].identifierType = 1;

lexicalParser.identifiersToken[currenIdentifier].arrayType = baseType;

lexicalParser.identifiersToken[currenIdentifier].arraySize = arraySize;

} else

lexicalParser.identifiersToken[currenIdentifier].identifierType = baseType;

scan(false);

while (!((t\_a == 'R') && (t\_n == 15))) { // { ',' I ('[' C ']' | eps) }

scan(true); // ','

scan(true);

if (t\_a != 'I') // I

error(13, "var 2");

currenIdentifier++;

if (t\_n != currenIdentifier)

error(1, "var");

varListEnd = t\_n;

scan(false);

if ((t\_a == 'R') && (t\_n == 10)) { // ('[' C ']' | eps)

scan(true); // '['

scan(true);

if (t\_a != 'C') // 'C'

error(19, "arrayType");

arraySize = lexicalParser.numbersToken[t\_n];

scan(true);

if (!((t\_a == 'R') && (t\_n == 11))) // ']'

error(4, "arrayType");

lexicalParser.identifiersToken[currenIdentifier].identifierType = 1;

lexicalParser.identifiersToken[currenIdentifier].arrayType = baseType;

lexicalParser.identifiersToken[currenIdentifier].arraySize = arraySize;

} else

lexicalParser.identifiersToken[currenIdentifier].identifierType = baseType;

scan(false);

}

scan(true);

if (!((t\_a == 'R') && (t\_n == 15))) // ;

error(16, "var");

for (int i = varListBegin; i <= varListEnd; i++) {

fprintf(asm\_variables, "%s", lexicalParser.identifiersToken[i].name.c\_str());

if (lexicalParser.identifiersToken[i].identifierType == 1)

u = lexicalParser.identifiersToken[i].arrayType;

else

u = lexicalParser.identifiersToken[i].identifierType;

fprintf(asm\_variables, (u == 13) ? " dw " : " db ");

if (lexicalParser.identifiersToken[i].identifierType == 1)

fprintf(asm\_variables, "%d DUP (?) \n", lexicalParser.identifiersToken[i].arraySize);

else

fprintf(asm\_variables, "%s", "(?) \n");

if (varListEnd != lexicalParser.totalIdentifiers - 1)

for (int i = varListEnd+1; i < lexicalParser.totalIdentifiers; i++)

error(23, "var");

}

scan(false);

}

scan(true); // 'endvar'

scan(true);

if (!((t\_a == 'R') && (t\_n == 15))) // ';'

error(16, "var");

}

// <type> -> 'bool' | 'char' | 'int'

void SyntaxParser::f\_type() {

scan(true);

if (t\_a != 'K')

error(18, "baseType");

else {

if ((t\_n == 3) || (t\_n == 6) || (t\_n == 13)) // | 'bool' | 'char' | 'int'

baseType = t\_n;

else

error(18, "baseType");

}

}

// <operators\_block> -> 'begin' { <operator> ';' } 'end'

void SyntaxParser::f\_operatorsBlock() {

scan(true);

if (!((t\_a == 'K') && (t\_n == 2))) // 'begin'

error(24, "begin...end");

scan(false);

while (!((t\_a == 'K') && (t\_n == 9))) { // 'end'

f\_operator();

scan(true);

if (!((t\_a == 'R') && (t\_n == 15))) // ';'

error(16, "begin...end");

scan(false);

}

scan(true); // 'end'

}

// <operators> -> <operatorsBlock> | <if> | <let> | <read> | <readln> | <while> | <write> | <writeln> | <repeat> | eps

void SyntaxParser::f\_operator() {

scan(false);

if (!((t\_a == 'R') && (t\_n == 15))) // ';'

if (t\_a == 'K')

switch (t\_n) {

case 2 : f\_operatorsBlock(); break; // <operatos\_block>

case 12 : f\_if(); break; // <if>

case 14 : f\_let(); break; // <let>

case 19 : f\_read(false); break; // <read>

case 20 : f\_read(true); break; // <readln>

case 25 : f\_while(); break; // <while>

case 26 : f\_write(false); break; // <write>

case 27 : f\_write(true); break; // <writeln>

case 28 : f\_repeat(); break; // <repeat>

default : f\_let(); break;

}

else

f\_let();

}

// <if> -> 'if' El 'then' <operator> ('else' <operator> | eps)

void SyntaxParser::f\_if() {

fprintf(asm\_code, "; IF() \n");

scan(true); // 'if'

EL();

string thenLabel = generateAsmLabel();

string elseLabel = generateAsmLabel();

fprintf(asm\_code, " pop ax \n");

fprintf(asm\_code, " cmp ax, 0 \n");

fprintf(asm\_code, " jz %s \n", thenLabel.c\_str());

scan(true);

if (!((t\_a == 'K') && (t\_n == 22))) // 'then'

error(26, "if");

fprintf(asm\_code, "; IF THEN \n");

f\_operator();

fprintf(asm\_code, " jmp %s \n", elseLabel.c\_str());

fprintf(asm\_code, "%s: \n", thenLabel.c\_str());

scan(false);

if ((t\_a == 'K') && ( t\_n == 8)) { // 'else' <operator> | eps

fprintf(asm\_code, "; IF ELSE \n");

scan(true);

f\_operator();

}

fprintf(asm\_code, "%s: \n", elseLabel.c\_str());

}

// <let> -> ('let' | eps) I ('[' E ']' | eps) '=' (E | El | L | I ('[' E ']' | eps))

void SyntaxParser::f\_let() {

fprintf(asm\_code, "; LET() \n");

scan(false);

if ((t\_a == 'K') && (t\_n == 14)) // 'let'

scan(true);

scan(true);

if (t\_a != 'I')

error(13, "let");

int typeCode, idenNum = t\_n;

if (lexicalParser.identifiersToken[t\_n].identifierType == 1) // array

typeCode = lexicalParser.identifiersToken[t\_n].arrayType;

else

typeCode = lexicalParser.identifiersToken[t\_n].identifierType;

fprintf(asm\_code, " mov di, 0 \n");

scan(false);

if ((t\_a == 'R') && (t\_n == 10)) { // '[' E ']' | eps)

if (lexicalParser.identifiersToken[idenNum].identifierType != 1)

error(27, "let");

scan(true); // '['

E();

fprintf(asm\_code, " pop di \n");

scan(true);

if (!((t\_a == 'R') && (t\_n == 11))) // ']'

error(4, "let");

}

fprintf(asm\_code, " push di \n");

scan(true);

if (!((t\_a == 'R') && (t\_n == 9))) // '='

error(28, "let");

switch (typeCode) {

case 3 : // El

EL();

fprintf(asm\_code, " pop ax \n");

fprintf(asm\_code, " pop di \n");

fprintf(asm\_code, " mov %s[di], al \n", lexicalParser.identifiersToken[idenNum].name.c\_str());

break;

case 6 : // L | I ('[' E ']' | eps))

scan(true);

if (t\_a == 'L') {

fprintf(asm\_code, " mov al, \"%s\" \n", lexicalParser.literalsToken[t\_n][0]);

fprintf(asm\_code, " pop di \n");

fprintf(asm\_code, " mov %s[di], al \n", lexicalParser.identifiersToken[idenNum].name.c\_str());

} else { // I

int sIdenNum = t\_n;

fprintf(asm\_code, " mov di, 0 \n");

scan(false);

if ((t\_a == 'R') && (t\_n == 10)) { // '[' E ']'

scan(true); // '['

E();

fprintf(asm\_code, " pop di \n");

scan(true);

if (!((t\_a == 'R') && (t\_n == 11))) // ']'

error(4, "let");

}

scan(false);

fprintf(asm\_code, " mov al, %s[di] \n", lexicalParser.identifiersToken[sIdenNum].name.c\_str());

fprintf(asm\_code, " pop di \n");

fprintf(asm\_code, " mov %s[di], al \n", lexicalParser.identifiersToken[idenNum].name.c\_str());

}

scan(false);

break;

case 13 :

E();

fprintf(asm\_code, " pop ax \n");

fprintf(asm\_code, " pop di \n");

if (lexicalParser.identifiersToken[idenNum].identifierType == 1) // array

fprintf(asm\_code, " shl di, 1 \n");

fprintf(asm\_code, " mov %s[di], ax \n", lexicalParser.identifiersToken[idenNum].name.c\_str());

break;

}

}

// <while> -> 'while' '(' El ')' <operator>

void SyntaxParser::f\_while() {

fprintf(asm\_code, "; WHILE() \n");

scan(true); // 'while'

scan(true);

if (!((t\_a == 'R') && (t\_n == 5))) // '('

error(25, "while");

string repeatLabel = generateAsmLabel();

fprintf(asm\_code, "%s: \n", repeatLabel.c\_str());

EL();

fprintf(asm\_code, " pop ax \n");

fprintf(asm\_code, " cmp ax, 0 \n");

string endLabel = generateAsmLabel();

fprintf(asm\_code, " jz %s \n", endLabel.c\_str());

scan(true);

if (!(t\_a == 'R') && (t\_n == 6)) // ')'

error(5, "while");

f\_operator();

fprintf(asm\_code, " jmp %s \n", repeatLabel.c\_str());

fprintf(asm\_code, "%s: \n", endLabel.c\_str());

}

// <read> -> 'read' '(' ((I ('[' E ']' | eps) {',' I ('[' E ']' | eps)}) | eps) | eps ')'

void SyntaxParser::f\_read(bool ln) {

int typeCode, idenNum;

fprintf(asm\_code, (ln) ? "; READLN() \n" : "; READ() \n");

scan(true); // 'read'

scan(true);

if (!((t\_a == 'R') && (t\_n == 5))) // '('

error(25, "read(ln)");

scan(false);

if ((t\_a == 'R') && (t\_n == 6)) // ')'

scan(true);

else { // ((I ('[' E ']' | eps) {',' I ('[' E ']' | eps)}) | eps)

if (t\_a != 'I')

error(13, "read(ln)");

idenNum = t\_n;

if (lexicalParser.identifiersToken[t\_n].identifierType == 1) // array

typeCode = lexicalParser.identifiersToken[t\_n].arrayType;

else

typeCode = lexicalParser.identifiersToken[t\_n].identifierType;

scan(true);

fprintf(asm\_code, " push 0 \n"); // mov di, 0

scan(false);

if ((t\_a == 'R') && (t\_n == 10)) { // ('[' E ']' | eps)

scan(true);

E();

scan(true);

if (!((t\_a == 'R') && (t\_n == 11))) // ']'

error(5, "read(ln)");

}

switch (typeCode) {

case 3 : f\_readBool(idenNum); break;// bool

case 6 : f\_readChar(idenNum); break; // char

case 13 : f\_readInt(idenNum); break; // int

}

scan(false);

while (!((t\_a == 'R') && (t\_n == 6))) { // {',' I ('[' E ']' | eps)}

scan(true);

if (!((t\_a == 'R') && (t\_n == 13))) // ','

error(14, "read(ln)");

scan(false);

if (t\_a != 'I')

error(13, "read(ln)");

idenNum = t\_n;

if (lexicalParser.identifiersToken[t\_n].identifierType == 1) // array

typeCode = lexicalParser.identifiersToken[t\_n].arrayType;

else

typeCode = lexicalParser.identifiersToken[t\_n].identifierType;

scan(true);

fprintf(asm\_code, " push 0 \n"); // mov di, 0

scan(false);

if ((t\_a == 'R') && (t\_n == 10)) { // ('[' E ']' | eps)

scan(true);

E();

scan(true);

if (!((t\_a == 'R') && (t\_n == 11))) // ']'

error(5, "read(ln)");

}

switch (typeCode) {

case 3 : f\_readBool(idenNum); break;// bool

case 6 : f\_readChar(idenNum); break; // char

case 13 : f\_readInt(idenNum); break; // int

}

scan(false);

}

}

scan(true);

if (ln) {

fprintf(asm\_code, " lea dx, clrf \n");

fprintf(asm\_code, " mov ah, 9 \n");

fprintf(asm\_code, " int 21h \n");

}

}

void SyntaxParser::f\_readInt(int idenNum) {

fprintf(asm\_code, "; READ INTEGER \n");

string sLabel = generateAsmLabel();

string lerror = generateAsmLabel();

string lstart = generateAsmLabel();

string lend = generateAsmLabel();

fprintf(asm\_code, "%s: \n", lstart.c\_str());

fprintf(asm\_code, " mov ah, 0Ah \n");

fprintf(asm\_code, " lea dx, @buffer \n");

fprintf(asm\_code, " int 21h \n");

fprintf(asm\_code, " mov ax, 0 \n");

fprintf(asm\_code, " mov cx, 0 \n");

fprintf(asm\_code, " mov cl, byte ptr[blength] \n");

fprintf(asm\_code, " mov bx, cx \n");

fprintf(asm\_code, "%s: \n", sLabel.c\_str());

fprintf(asm\_code, " dec bx \n");

fprintf(asm\_code, " mov al, @buf[bx] \n");

fprintf(asm\_code, " cmp al, \"9\" \n");

fprintf(asm\_code, " ja %s \n", lerror.c\_str());

fprintf(asm\_code, " cmp al, \"0\" \n");

fprintf(asm\_code, " jb %s \n", lerror.c\_str());

fprintf(asm\_code, " loop %s \n", sLabel.c\_str());

sLabel = generateAsmLabel();

fprintf(asm\_code, " mov cl, byte ptr[blength] \n");

fprintf(asm\_code, " mov di, 0 \n");

fprintf(asm\_code, " mov ax, 0 \n");

fprintf(asm\_code, "%s: \n", sLabel.c\_str());

fprintf(asm\_code, " mov bl, @buf[di] \n");

fprintf(asm\_code, " inc di \n");

fprintf(asm\_code, " sub bl, 30h \n");

fprintf(asm\_code, " add ax, bx \n");

fprintf(asm\_code, " mov si, ax \n");

fprintf(asm\_code, " mov bx, 10 \n");

fprintf(asm\_code, " mul bx \n");

fprintf(asm\_code, " loop %s \n", sLabel.c\_str());

fprintf(asm\_code, " mov ax, si \n");

fprintf(asm\_code, " pop di \n");

fprintf(asm\_code, " shl di, 1 \n");

fprintf(asm\_code, " mov %s[di],ax \n", lexicalParser.identifiersToken[idenNum].name.c\_str());

fprintf(asm\_code, " jmp %s \n", lend.c\_str());

fprintf(asm\_code, "%s: \n", lerror.c\_str());

fprintf(asm\_code, " lea dx, err\_msg \n");

fprintf(asm\_code, " mov ah, 9 \n");

fprintf(asm\_code, " int 21h \n");

fprintf(asm\_code, " jmp %s \n", lstart.c\_str());

fprintf(asm\_code, "%s: \n", lend.c\_str());

}

void SyntaxParser::f\_readBool(int idenNum) {

fprintf(asm\_code, "; READ BOOLEAN \n");

string start = generateAsmLabel();

string l4 = generateAsmLabel();

string l5 = generateAsmLabel();

string le = generateAsmLabel();

string lt = generateAsmLabel();

string lf = generateAsmLabel();

string lend = generateAsmLabel();

string lerror = generateAsmLabel();

fprintf(asm\_code, "%s: \n", start.c\_str());

fprintf(asm\_code, " mov ah, 0Ah \n");

fprintf(asm\_code, " lea dx, @buffer \n");

fprintf(asm\_code, " int 21h \n");

fprintf(asm\_code, " cmp blength, 4 \n");

fprintf(asm\_code, " je %s \n", l4.c\_str());

fprintf(asm\_code, " cmp blength, 5 \n");

fprintf(asm\_code, " je '%s \n", l5.c\_str());

fprintf(asm\_code, " jmp %s \n", lerror.c\_str());

fprintf(asm\_code, "%s: \n", l4.c\_str());

fprintf(asm\_code, " lea si, @true \n");

fprintf(asm\_code, " lea di, @buf \n");

fprintf(asm\_code, " mov cx, 4 \n");

fprintf(asm\_code, " repe cmpsb \n");

fprintf(asm\_code, " jz %s \n", le.c\_str());

fprintf(asm\_code, " jmp %s \n", lerror.c\_str());

fprintf(asm\_code, "%s: \n", l5.c\_str());

fprintf(asm\_code, " lea si, @false \n");

fprintf(asm\_code, " lea di, @buf \n");

fprintf(asm\_code, " mov cx, 5 \n");

fprintf(asm\_code, " repe cmpsb \n");

fprintf(asm\_code, " jz %s \n", le.c\_str());

fprintf(asm\_code, " jmp %s \n", lerror.c\_str());

fprintf(asm\_code, "%s: \n", le.c\_str());

fprintf(asm\_code, " cmp @buf[0], \"t\" \n");

fprintf(asm\_code, " je %s \n", lt.c\_str());

fprintf(asm\_code, " push 0 \n");

fprintf(asm\_code, " jmp %s \n", lend.c\_str());

fprintf(asm\_code, "%s: \n", lt.c\_str());

fprintf(asm\_code, " push 1 \n");

fprintf(asm\_code, " jmp %s \n", lend.c\_str());

fprintf(asm\_code, "%s: \n", lerror.c\_str());

fprintf(asm\_code, " lea dx, err\_msg \n");

fprintf(asm\_code, " mov ah, 9 \n");

fprintf(asm\_code, " int 21h \n");

fprintf(asm\_code, " jmp %s \n", start.c\_str());

fprintf(asm\_code, "%s: \n", lend.c\_str());

fprintf(asm\_code, " pop ax \n");

fprintf(asm\_code, " pop di \n");

fprintf(asm\_code, " mov %s[di], al \n", lexicalParser.identifiersToken[idenNum].name.c\_str());

}

void SyntaxParser::f\_readChar(int idenNum) {

fprintf(asm\_code, "; READ CHAR \n");

fprintf(asm\_code, " mov ah, 0Ah \n");

fprintf(asm\_code, " lea dx, @buffer \n");

fprintf(asm\_code, " int 21h \n");

fprintf(asm\_code, " xor dx, dx \n");

fprintf(asm\_code, " mov dl, @buf[0] \n");

fprintf(asm\_code, " pop di \n");

fprintf(asm\_code, " mov %s[di], dl \n", lexicalParser.identifiersToken[idenNum].name.c\_str());

}

// <write> -> 'write' '(' (((E | L) {',' (E | L)} ) | eps) ')'

void SyntaxParser::f\_write(bool ln) {

int typeCode, idenNum;

fprintf(asm\_code, (ln) ? "; WRITELN() \n" : "; WRITE() \n" );

scan(true); // 'write'

scan(true);

if (!((t\_a == 'R') && (t\_n == 5))) // '('

error(25, "write(ln)");

scan(false);

if ((t\_a == 'R') && (t\_n == 6)) // ')'

scan(true);

else { // ((E | L) {',' (E | L)} )

if (t\_a == 'L') { // L

f\_writeLiteral();

scan(true);

} else {

if (t\_a == 'I') { // E

idenNum = t\_n;

if (lexicalParser.identifiersToken[t\_n].identifierType == 1) // array

typeCode = lexicalParser.identifiersToken[t\_n].arrayType;

else

typeCode = lexicalParser.identifiersToken[t\_n].identifierType;

switch (typeCode) {

case 3 : // bool

EL();

f\_writeBool();

break;

case 6 : // char

fprintf(asm\_code, " mov di, 0 \n");

scan(false);

if ((t\_a == 'R') && (t\_n == 10)) {

scan(true);

E();

fprintf(asm\_code, " pop di \n");

scan(true);

if (!((t\_a == 'R') && (t\_n == 11)))

error(5, "write(ln)");

}

f\_writeChar(idenNum);

scan(true);

break;

case 13 : // int

E();

f\_writeInt();

break;

}

}

}

// {',' (E | L)}

scan(false);

while (!((t\_a == 'R') && (t\_n == 6))) { // ')'

scan(true);

if (!((t\_a == 'R') && (t\_n == 13))) // ','

error(14, "write(ln)");

scan(false);

if (t\_a == 'L') { // L

f\_writeLiteral();

scan(true);

} else {

if (t\_a == 'I') { // E

idenNum = t\_n;

if (lexicalParser.identifiersToken[t\_n].identifierType == 1) // array

typeCode = lexicalParser.identifiersToken[t\_n].arrayType;

else

typeCode = lexicalParser.identifiersToken[t\_n].identifierType;

switch (typeCode) {

case 3 : // bool

EL();

f\_writeBool();

break;

case 6 : // char

scan(true);

fprintf(asm\_code, " mov di, 0 \n");

scan(false);

if ((t\_a == 'R') && (t\_n == 10)) {

scan(true);

E();

fprintf(asm\_code, " pop di \n");

scan(true);

if (!((t\_a == 'R') && (t\_n == 11)))

error(5, "write(ln)");

}

f\_writeChar(idenNum);

scan(false);

break;

case 13 : // int

E();

f\_writeInt();

break;

}

}

}

scan(false);

}

scan(true);

}

if (ln) {

fprintf(asm\_code, " lea dx, clrf \n");

fprintf(asm\_code, " mov ah, 9 \n");

fprintf(asm\_code, " int 21h \n");

}

}

void SyntaxParser::f\_writeInt() {

fprintf(asm\_code, "; WRITE ARIPHMETICS \n");

string l1 = generateAsmLabel();

string l2 = generateAsmLabel();

fprintf(asm\_code, " pop ax \n");

fprintf(asm\_code, " mov bx, 10 \n");

fprintf(asm\_code, " mov di, 0 \n");

fprintf(asm\_code, " mov si, ax \n");

fprintf(asm\_code, " cmp ax, 0 \n");

string sLabel = generateAsmLabel();

fprintf(asm\_code, " jns %s \n", sLabel.c\_str());

fprintf(asm\_code, " neg si \n");

fprintf(asm\_code, " mov ah, 2 \n");

fprintf(asm\_code, " mov dl, \"-\" \n");

fprintf(asm\_code, " int 21h \n");

fprintf(asm\_code, " mov ax, si \n");

fprintf(asm\_code, "%s: \n", sLabel.c\_str());

fprintf(asm\_code, " mov dx, 0 \n");

fprintf(asm\_code, " div bx \n");

fprintf(asm\_code, " add dl, 30h \n");

fprintf(asm\_code, " mov output[di], dl \n");

fprintf(asm\_code, " inc di \n");

fprintf(asm\_code, " cmp al, 0 \n");

fprintf(asm\_code, " jnz %s \n", sLabel.c\_str());

fprintf(asm\_code, " mov cx, di \n");

fprintf(asm\_code, " dec di \n");

fprintf(asm\_code, " mov ah, 2 \n");

sLabel = generateAsmLabel();

fprintf(asm\_code, "%s: \n", sLabel.c\_str());

fprintf(asm\_code, " mov dl, output[di] \n");

fprintf(asm\_code, " dec di \n");

fprintf(asm\_code, " int 21h \n");

fprintf(asm\_code, " loop %s \n", sLabel.c\_str());

}

void SyntaxParser::f\_writeLiteral() {

fprintf(asm\_code, "; WRITE LITERAL \n");

string text = generateAsmLabel();

fprintf(asm\_variables, "%sp db \"%s$\" \n", text.c\_str(), lexicalParser.literalsToken[t\_n].c\_str());

fprintf(asm\_code, " lea dx, %sp \n", text.c\_str());

fprintf(asm\_code, " mov ah, 9 \n");

fprintf(asm\_code, " int 21h \n");

}

void SyntaxParser::f\_writeChar(int idenNum) {

fprintf(asm\_code, "; WRITE CHAR \n");

fprintf(asm\_code, " mov ax, 0 \n");

fprintf(asm\_code, " mov al, %s[di] \n", lexicalParser.identifiersToken[idenNum].name.c\_str());

fprintf(asm\_code, " mov dl, al \n");

fprintf(asm\_code, " mov ah, 2 \n");

fprintf(asm\_code, " int 21h \n");

}

void SyntaxParser::f\_writeBool() {

string l0 = generateAsmLabel();

string l1 = generateAsmLabel();

fprintf(asm\_code, "; WRITE BOOLEAN \n");

fprintf(asm\_code, " pop ax \n");

fprintf(asm\_code, " cmp ax, 0 \n");

fprintf(asm\_code, " je %s \n", l0.c\_str());

fprintf(asm\_code, " lea dx, @@true \n");

fprintf(asm\_code, " jmp %s \n", l1.c\_str());

fprintf(asm\_code, "%s: \n", l0.c\_str());

fprintf(asm\_code, " lea dx, @@false \n");

fprintf(asm\_code, "%s: \n", l1.c\_str());

fprintf(asm\_code, " mov ah, 9 \n");

fprintf(asm\_code, " int 21h \n");

}

void SyntaxParser::error(int errorCode, string oper) {

printf("Current lexeme: %c%d . Operator: %s ", t\_a, t\_n, oper.c\_str());

switch (errorCode) {

case 1: printf("Redefenition of identifier %s. \n", lexicalParser.identifiersToken[t\_n].name.c\_str()); break;

case 2: printf("Expected + or -. \n"); break;

case 3: printf("Expected \* or /. \n"); break;

case 4: printf("Expected ]. \n"); break;

case 5: printf("Expected ). \n"); break;

case 6: printf("Expected or. \n"); break;

case 7: printf("Expected and. \n"); break;

case 8: printf("Expected compare operation. \n"); break;

case 9: printf("Expected program \n"); break;

case 10: printf("Expected %prograt\_name% \n"); break;

case 11: printf("Expected .. \n"); break;

case 12: printf("Expected var \n"); break;

case 13: printf("Expected identifier. \n"); break;

case 14: printf("Expected ,. \n"); break;

case 15: printf("Expected :. \n"); break;

case 16: printf("Expected ;. \n"); break;

case 17: printf("Expected identifier or begin. \n"); break;

case 18: printf("Expected %type\_name%. \n"); break;

case 19: printf("Expected array. \n"); break;

case 20: printf("Expected [. \n"); break;

case 21: printf("Expected number. \n"); break;

case 22: printf("Expected of. \n"); break;

case 23: printf("Undeclared identifier %s. \n", lexicalParser.identifiersToken[t\_n].name.c\_str()); break;

case 24: printf("Expected begin. \n"); break;

case 25: printf("Expected (. \n"); break;

case 26: printf("Expected then. \n"); break;

case 27: printf("Unexpected [. \n"); break;

case 28: printf("Expected =. \n"); break;

case 29: printf("Expected do. \n"); break;

case 30: printf("Expected true or false in assignment. \n"); break;

case 31: printf("Expected end. \n"); break;

case 32: printf("Expected }."); break;

case 33: printf("Expected {."); break;

}

fclose(asm\_code);

fclose(asm\_variables);

exit(1);

}

// <repeat> -> 'repeat' <operator> 'until' El

void SyntaxParser::f\_repeat() {

fprintf(asm\_code, "; REPEAT() \n");

string lstart = generateAsmLabel();

scan(true);

fprintf(asm\_code, "%s:", lstart.c\_str());

scan(false);

while (!((t\_a == 'K') && (t\_n == 29))) { // 'until'

f\_operator();

scan(true);

if (!((t\_a == 'R') && (t\_n == 15))) // ';'

error(14, "repeat");

scan(false);

}

scan(true);

EL();

fprintf(asm\_code, " pop ax \n");

fprintf(asm\_code, " cmp ax, 0 \n");

fprintf(asm\_code, " jne %s \n", lstart.c\_str());

}

void SyntaxParser::compile() {

f\_program();

fclose(asm\_code);

fclose(asm\_variables);

fclose(lexicalParser.lex\_file);

cout << "Build successfully" << endl;

}

int main(int argc, char\*\* argv) {

SyntaxParser syntaxParser;

syntaxParser.compile();

system("pause");

return 0;

}

# 4 Контрольный пример

## 4.1 Входная программа

program prg; // Вариант 5

var int x, y, a, b, c;

int mas[10], i;

char s[2];

endvar;

begin

i=0;

repeat

write('Enter char: ');

readln(s[i]);

i=i+1;

until [i < 3];

write('Enter x: ');

readln(x);

write('Enter y: ');

readln(y);

mas[x-2]=4\*y-3\*(y+2);

c=mas[x-2];

if [x<y] and [y>=5] then

a=2\*(x+y)

else

a=10\*x;

if ([x != 2] or true) then

begin

b=x-c;

writeln('b = ', b);

end;

x=0;

while ([x < 45])

begin

x=x+5;

y=(y+2)\*2;

end;

writeln('y = ', y);

writeln('a = ', a);

writeln('x = ', x);

repeat

x=x-x/4;

writeln('x= ', x);

until [x > 25];

i=0;

writeln('You enter char: ');

repeat

writeln('',s[i]);

i=i+1;

until [i < 3];

end.

## 4.2 Выходная программа

**asmData.asm**

.data

; SUPPORTING VARIABLES

@buffer db 6

blength db (?)

@buf db 256 DUP (?)

clrf db 0Dh, 0Ah, "$"

output db 6 DUP (?), "$"

err\_msg db "Input error, try again", 0Dh, 0Ah, "$"

@true db "true"

@@true db "true$"

@false db "false"

@@false db "false$"

; USING VARIABLES

x dw (?)

y dw (?)

a dw (?)

b dw (?)

c dw (?)

mas dw 10 DUP (?)

i dw (?)

s db 2 DUP (?)

@1p db "Enter char: $"

@4p db "Enter x: $"

@10p db "Enter y: $"

@26p db "b = $"

@35p db "y = $"

@40p db "a = $"

@45p db "x = $"

@51p db "x= $"

@58p db "You enter char: $"

@60p db "$"

**asmKod.asm**

.model small

include asmData.asm

.stack 100h

.code

.386

start:

mov ax, @data

mov ds, ax

; LET()

mov di, 0

push di

mov ax, 0

push ax

pop ax

pop di

mov i[di], ax

; REPEAT()

@0:; WRITE()

; WRITE LITERAL

lea dx, @1p

mov ah, 9

int 21h

; READLN()

push 0

mov di, 0

mov ax, i[di]

push ax

; READ CHAR

mov ah, 0Ah

lea dx, @buffer

int 21h

xor dx, dx

mov dl, @buf[0]

pop di

mov s[di], dl

lea dx, clrf

mov ah, 9

int 21h

; LET()

mov di, 0

push di

mov di, 0

mov ax, i[di]

push ax

mov ax, 1

push ax

pop bx

pop ax

add ax, bx

push ax

pop ax

pop di

mov i[di], ax

mov di, 0

mov ax, i[di]

push ax

mov ax, 3

push ax

pop bx

pop ax

cmp ax, bx

jb @2

push 0

jmp @3

@2:

push 1

@3:

pop ax

cmp ax, 0

jne @0

; WRITE()

; WRITE LITERAL

lea dx, @4p

mov ah, 9

int 21h

; READLN()

push 0

; READ INTEGER

@7:

mov ah, 0Ah

lea dx, @buffer

int 21h

mov ax, 0

mov cx, 0

mov cl, byte ptr[blength]

mov bx, cx

@5:

dec bx

mov al, @buf[bx]

cmp al, "9"

ja @6

cmp al, "0"

jb @6

loop @5

mov cl, byte ptr[blength]

mov di, 0

mov ax, 0

@9:

mov bl, @buf[di]

inc di

sub bl, 30h

add ax, bx

mov si, ax

mov bx, 10

mul bx

loop @9

mov ax, si

pop di

shl di, 1

mov x[di],ax

jmp @8

@6:

lea dx, err\_msg

mov ah, 9

int 21h

jmp @7

@8:

lea dx, clrf

mov ah, 9

int 21h

; WRITE()

; WRITE LITERAL

lea dx, @10p

mov ah, 9

int 21h

; READLN()

push 0

; READ INTEGER

@13:

mov ah, 0Ah

lea dx, @buffer

int 21h

mov ax, 0

mov cx, 0

mov cl, byte ptr[blength]

mov bx, cx

@11:

dec bx

mov al, @buf[bx]

cmp al, "9"

ja @12

cmp al, "0"

jb @12

loop @11

mov cl, byte ptr[blength]

mov di, 0

mov ax, 0

@15:

mov bl, @buf[di]

inc di

sub bl, 30h

add ax, bx

mov si, ax

mov bx, 10

mul bx

loop @15

mov ax, si

pop di

shl di, 1

mov y[di],ax

jmp @14

@12:

lea dx, err\_msg

mov ah, 9

int 21h

jmp @13

@14:

lea dx, clrf

mov ah, 9

int 21h

; LET()

mov di, 0

mov di, 0

mov ax, x[di]

push ax

mov ax, 2

push ax

pop bx

pop ax

sub ax, bx

push ax

pop di

push di

mov ax, 4

push ax

mov di, 0

mov ax, y[di]

push ax

pop bx

pop ax

mul bx

push ax

mov ax, 3

push ax

mov di, 0

mov ax, y[di]

push ax

mov ax, 2

push ax

pop bx

pop ax

add ax, bx

push ax

pop bx

pop ax

mul bx

push ax

pop bx

pop ax

sub ax, bx

push ax

pop ax

pop di

shl di, 1

mov mas[di], ax

; LET()

mov di, 0

push di

mov di, 0

mov di, 0

mov ax, x[di]

push ax

mov ax, 2

push ax

pop bx

pop ax

sub ax, bx

push ax

pop di

shl di, 1

mov ax, mas[di]

push ax

pop ax

pop di

mov c[di], ax

; IF()

mov di, 0

mov ax, x[di]

push ax

mov di, 0

mov ax, y[di]

push ax

pop bx

pop ax

cmp ax, bx

jb @16

push 0

jmp @17

@16:

push 1

@17:

mov di, 0

mov ax, y[di]

push ax

mov ax, 5

push ax

pop bx

pop ax

cmp ax, bx

jae @18

push 0

jmp @19

@18:

push 1

@19:

pop bx

pop ax

and ax, bx

push ax

pop ax

cmp ax, 0

jz @20

; IF THEN

; LET()

mov di, 0

push di

mov ax, 2

push ax

mov di, 0

mov ax, x[di]

push ax

mov di, 0

mov ax, y[di]

push ax

pop bx

pop ax

add ax, bx

push ax

pop bx

pop ax

mul bx

push ax

pop ax

pop di

mov a[di], ax

jmp @21

@20:

; IF ELSE

; LET()

mov di, 0

push di

mov ax, 10

push ax

mov di, 0

mov ax, x[di]

push ax

pop bx

pop ax

mul bx

push ax

pop ax

pop di

mov a[di], ax

@21:

; IF()

mov di, 0

mov ax, x[di]

push ax

mov ax, 2

push ax

pop bx

pop ax

cmp ax, bx

jne @22

push 0

jmp @23

@22:

push 1

@23:

push 1

pop bx

pop ax

or ax, bx

push ax

pop ax

cmp ax, 0

jz @24

; IF THEN

; LET()

mov di, 0

push di

mov di, 0

mov ax, x[di]

push ax

mov di, 0

mov ax, c[di]

push ax

pop bx

pop ax

sub ax, bx

push ax

pop ax

pop di

mov b[di], ax

; WRITELN()

; WRITE LITERAL

lea dx, @26p

mov ah, 9

int 21h

mov di, 0

mov ax, b[di]

push ax

; WRITE ARIPHMETICS

pop ax

mov bx, 10

mov di, 0

mov si, ax

cmp ax, 0

jns @29

neg si

mov ah, 2

mov dl, "-"

int 21h

mov ax, si

@29:

mov dx, 0

div bx

add dl, 30h

mov output[di], dl

inc di

cmp al, 0

jnz @29

mov cx, di

dec di

mov ah, 2

@30:

mov dl, output[di]

dec di

int 21h

loop @30

lea dx, clrf

mov ah, 9

int 21h

jmp @25

@24:

@25:

; LET()

mov di, 0

push di

mov ax, 0

push ax

pop ax

pop di

mov x[di], ax

; WHILE()

@31:

mov di, 0

mov ax, x[di]

push ax

mov ax, 45

push ax

pop bx

pop ax

cmp ax, bx

jb @32

push 0

jmp @33

@32:

push 1

@33:

pop ax

cmp ax, 0

jz @34

; LET()

mov di, 0

push di

mov di, 0

mov ax, x[di]

push ax

mov ax, 5

push ax

pop bx

pop ax

add ax, bx

push ax

pop ax

pop di

mov x[di], ax

; LET()

mov di, 0

push di

mov di, 0

mov ax, y[di]

push ax

mov ax, 2

push ax

pop bx

pop ax

add ax, bx

push ax

mov ax, 2

push ax

pop bx

pop ax

mul bx

push ax

pop ax

pop di

mov y[di], ax

jmp @31

@34:

; WRITELN()

; WRITE LITERAL

lea dx, @35p

mov ah, 9

int 21h

mov di, 0

mov ax, y[di]

push ax

; WRITE ARIPHMETICS

pop ax

mov bx, 10

mov di, 0

mov si, ax

cmp ax, 0

jns @38

neg si

mov ah, 2

mov dl, "-"

int 21h

mov ax, si

@38:

mov dx, 0

div bx

add dl, 30h

mov output[di], dl

inc di

cmp al, 0

jnz @38

mov cx, di

dec di

mov ah, 2

@39:

mov dl, output[di]

dec di

int 21h

loop @39

lea dx, clrf

mov ah, 9

int 21h

; WRITELN()

; WRITE LITERAL

lea dx, @40p

mov ah, 9

int 21h

mov di, 0

mov ax, a[di]

push ax

; WRITE ARIPHMETICS

pop ax

mov bx, 10

mov di, 0

mov si, ax

cmp ax, 0

jns @43

neg si

mov ah, 2

mov dl, "-"

int 21h

mov ax, si

@43:

mov dx, 0

div bx

add dl, 30h

mov output[di], dl

inc di

cmp al, 0

jnz @43

mov cx, di

dec di

mov ah, 2

@44:

mov dl, output[di]

dec di

int 21h

loop @44

lea dx, clrf

mov ah, 9

int 21h

; WRITELN()

; WRITE LITERAL

lea dx, @45p

mov ah, 9

int 21h

mov di, 0

mov ax, x[di]

push ax

; WRITE ARIPHMETICS

pop ax

mov bx, 10

mov di, 0

mov si, ax

cmp ax, 0

jns @48

neg si

mov ah, 2

mov dl, "-"

int 21h

mov ax, si

@48:

mov dx, 0

div bx

add dl, 30h

mov output[di], dl

inc di

cmp al, 0

jnz @48

mov cx, di

dec di

mov ah, 2

@49:

mov dl, output[di]

dec di

int 21h

loop @49

lea dx, clrf

mov ah, 9

int 21h

; REPEAT()

@50:; LET()

mov di, 0

push di

mov di, 0

mov ax, x[di]

push ax

mov di, 0

mov ax, x[di]

push ax

mov ax, 4

push ax

pop bx

pop ax

xor dx, dx

div bx

push ax

pop bx

pop ax

sub ax, bx

push ax

pop ax

pop di

mov x[di], ax

; WRITELN()

; WRITE LITERAL

lea dx, @51p

mov ah, 9

int 21h

mov di, 0

mov ax, x[di]

push ax

; WRITE ARIPHMETICS

pop ax

mov bx, 10

mov di, 0

mov si, ax

cmp ax, 0

jns @54

neg si

mov ah, 2

mov dl, "-"

int 21h

mov ax, si

@54:

mov dx, 0

div bx

add dl, 30h

mov output[di], dl

inc di

cmp al, 0

jnz @54

mov cx, di

dec di

mov ah, 2

@55:

mov dl, output[di]

dec di

int 21h

loop @55

lea dx, clrf

mov ah, 9

int 21h

mov di, 0

mov ax, x[di]

push ax

mov ax, 25

push ax

pop bx

pop ax

cmp ax, bx

ja @56

push 0

jmp @57

@56:

push 1

@57:

pop ax

cmp ax, 0

jne @50

; LET()

mov di, 0

push di

mov ax, 0

push ax

pop ax

pop di

mov i[di], ax

; WRITELN()

; WRITE LITERAL

lea dx, @58p

mov ah, 9

int 21h

lea dx, clrf

mov ah, 9

int 21h

; REPEAT()

@59:; WRITELN()

; WRITE LITERAL

lea dx, @60p

mov ah, 9

int 21h

mov di, 0

mov di, 0

mov ax, i[di]

push ax

pop di

; WRITE CHAR

mov ax, 0

mov al, s[di]

mov dl, al

mov ah, 2

int 21h

lea dx, clrf

mov ah, 9

int 21h

; LET()

mov di, 0

push di

mov di, 0

mov ax, i[di]

push ax

mov ax, 1

push ax

pop bx

pop ax

add ax, bx

push ax

pop ax

pop di

mov i[di], ax

mov di, 0

mov ax, i[di]

push ax

mov ax, 3

push ax

pop bx

pop ax

cmp ax, bx

jb @61

push 0

jmp @62

@61:

push 1

@62:

pop ax

cmp ax, 0

jne @59

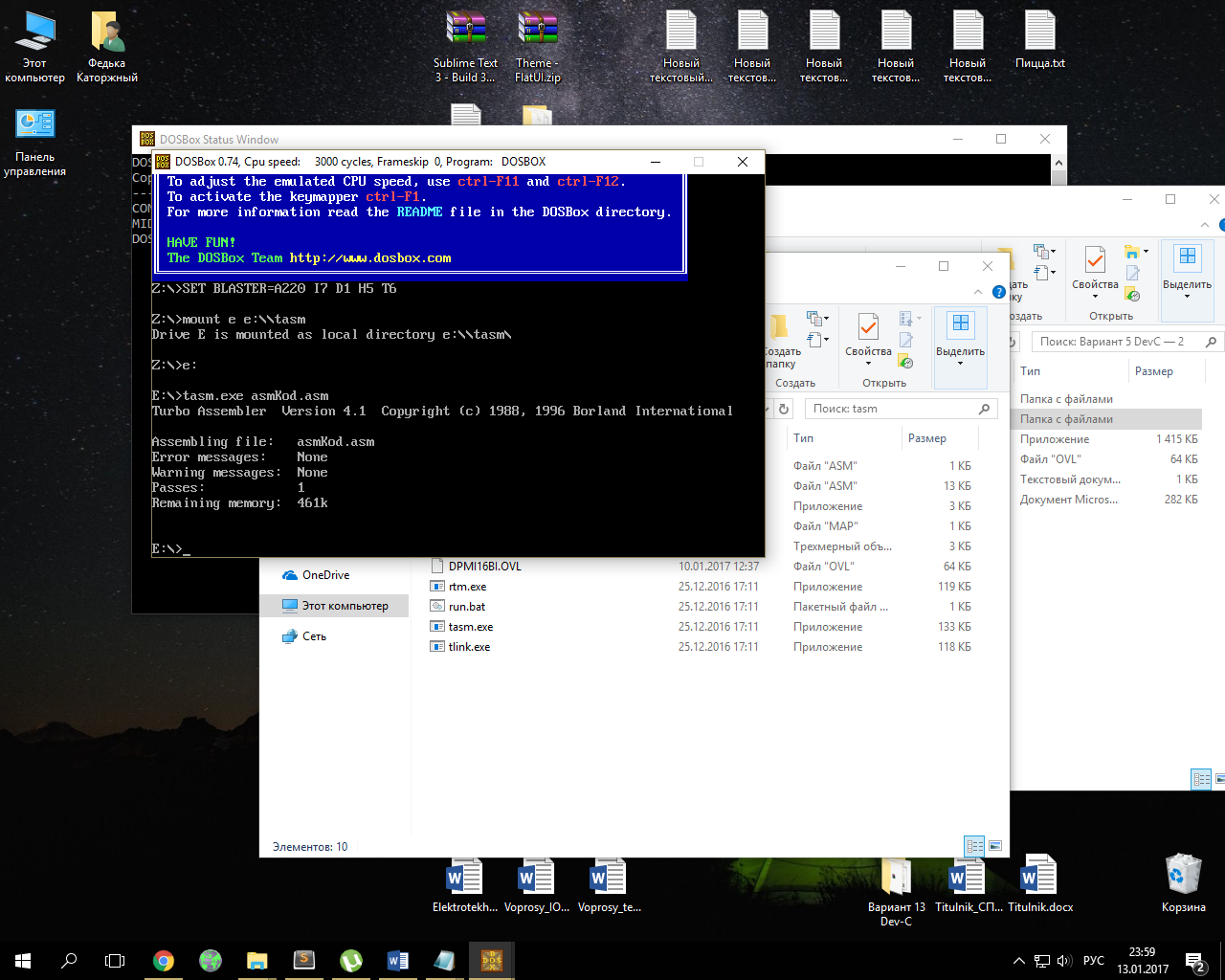
error:

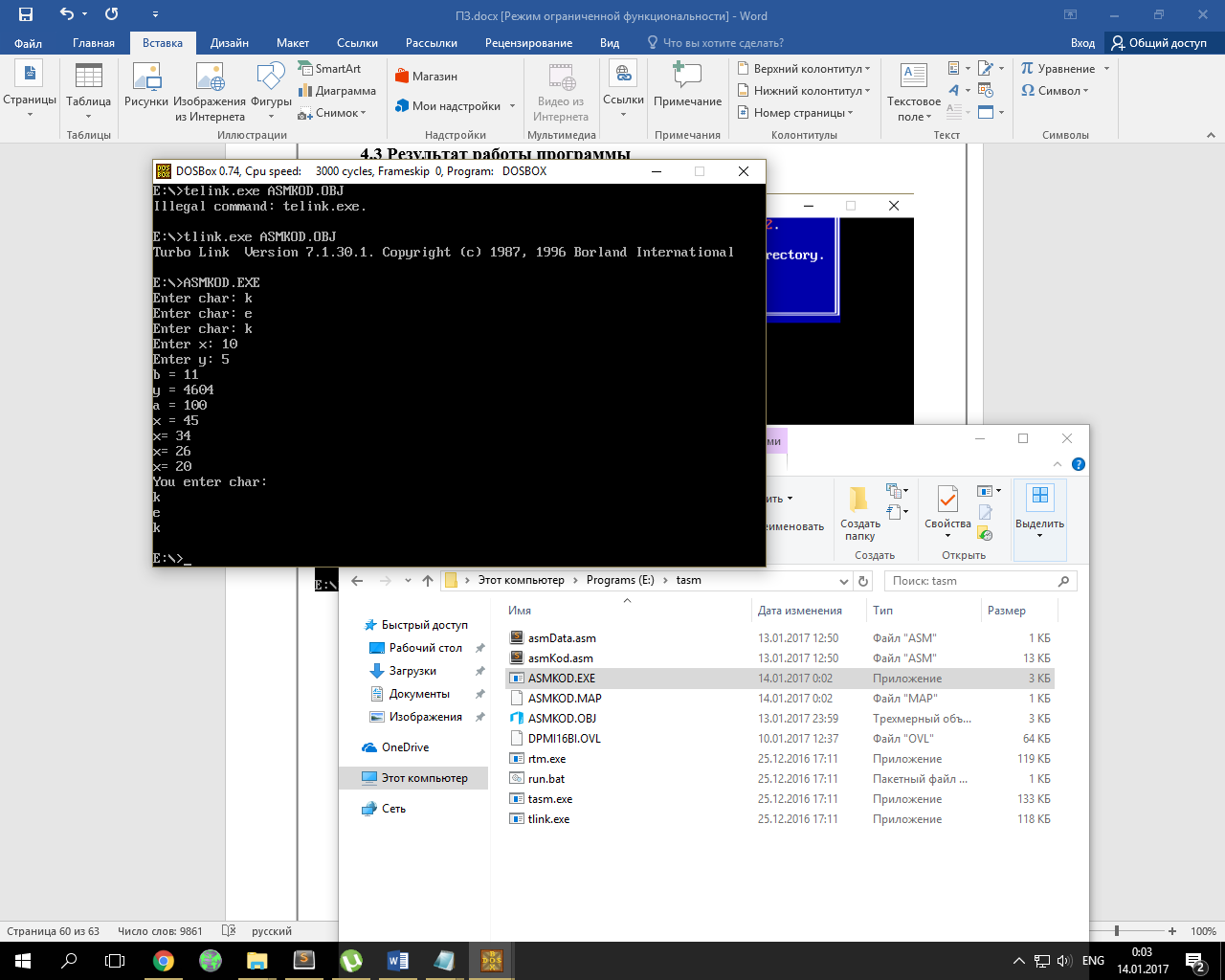
mov ax, 4C00h

int 21h

end start

## 4.3 Результат работы программы





# Заключение

В результате выполнения курсовой работы для заданного входного языка была разработана грамматика языка программирования, включающая следующие возможности:

* использование типов данных (целый, символьный, логический, одномерный массив; литералы);
* операторы (присваивания, цикла, выбора, ввода, вывода);
* возможность использования комментариев.

Разработан компилятор для указанного выше языка включающий следующие характеристиками:

* лексический анализ;
* синтаксический анализ;
* генератор кода;
* организацию таблиц;
* реализацию индивидуального задания.

Отдельные части компилятора, разработанные в данной курсовой работе, дают представление о технике и методах, лежащих в основе построения компиляторов.

# Список использованных источников

1. **Юров В.** Assembler: специальный справочник– СПб.: Питер, 2000. – 496 с.
2. **Ахо А., Сети Р., Ульман, Джеффри Д.** Компиляторы: принципы, технологии и инструменты.: Пер. с англ – М.: “Вильямс”, 2003. – 768 с.
3. **ИшаковаЕ.Н.**Разработка компиляторов: Методические указания к курсовой работе – Оренбург: ГОУ ОГУ, 2005. – 50с.
4. Лекции по системному программному обеспечению.