НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

Кафедра системного програмування і спеціалізованих комп‟ютерних

систем

**КУРСОВА РОБОТА**

з дисципліни «Системне програмування»

на тему: Розробка компілятора програм мовою Асемблера

Студента 2 курсу групи КВ-41

напряму підготовки

6.050102 «Ком`ютерна інженерія»

Горпинича-Радуженка Івана

Керівник\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Національна оцінка \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Кількість балів: \_\_\_\_\_\_\_\_\_Оцінка: ECTS \_\_\_\_\_

Члени комісії \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(підпис) (вчене звання, науковий ступінь, прізвище та ініціали)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(підпис) (вчене звання, науковий ступінь, прізвище та ініціали)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(підпис) (вчене звання, науковий ступінь, прізвище та ініціали)

Київ – 2016 рік

**ТЕХНІЧНЕ ЗАВДАННЯ**

1. Вхідні дані транслятора - текстовий файл з довільною програмою на мові Асемблера, складеною в відповідності з обмеженнями, які задані в варіанті курсової роботи. Для підготовки програми на мові Асемблера використовується, наприклад, стандартний додаток OS Windows Блокнот.

2. На всі синтаксичні конструкції (ідентифікатори, константи, директиви, машинні команди, режими адресації і т.д.), які допускаються в TASM(MASM) і які виходять за рамки обмежень в варіанті курсової роботи повинно видаватись діагностичне повідомлення про синтаксичну помилку.

3. В результаті роботи транслятора повинен бути створений текстовий файл лістінга (розширення .lst). Формат файлу лістінга повинен співпадати з форматом файлу лістінга MASM або TASM. Діагностичні повідомлення формуються на українській мові. Таблиця символів в файлі лістінга може бути в довільному форматі.

4. Транслятор повинен аналізувати командний рядок, в якому задаються імена початкового файлу та файлу лістінга. Всі діагностичні повідомлення, які формуються в файлі лістінга додатково повинні виводитись на екран монітора. Крім того, на екран виводиться загальна кількість помилок, виявлених в початковій програмі.

**ІНДИВІДУАЛЬНЕ ЗАВДАННЯ**

**Варіант 5**

*Ідентифікатори*

Містять великі і малі букви латинского алфавіту та цифри. Починаються з букви. Великі та малі букви не відрізняються. Довжина ідентифікаторів не більше 4 символів

*Константи*

Шістнадцяткові, десяткові, двійкові та текстові константи

*Директиви*

END, SEGMENT - без операндів, ENDS, ASSUME

DB,DW,DD з одним операндом - константою (рядкові константи тільки для DB)

*Розрядність даних та адрес*

32-розрядні дані та зміщення в сегменті, y випадку 16-розрядних даних та 16- розрядних зміщень генеруються відповідні префікси зміни розрядності

*Адресація операндів пам'яті*

Базова індексна адресація ([bx+di],[bp+ei],[edx+esi],[ebx+ecx] і т.п.) з оператором визначення типу (ptr) при необхідності

*Заміна сегментів*

Префікси заміни сегментів можуть задаватись тільки явно

*Машинні команди*

Sti

Mov **reg, mem**

Push **reg**

Div **mem**

Adc **reg, reg**

Sub **reg, mem**

Xor **mem,imm**

Jz

jmp (міжсегментна пряма та внутрішньосегментна відносна адресація)

Де **reg –** 8,16 або 32-розрядні РЗП

**mem –** адреса операнда в пам’яті

**imm -** 8,16 або 32-розрядні безпосередні дані (константи)

**Опис загальної структури програми:**

**Analys\_header.h:** містить оголошення типів та функцій, необхідних для роботи лексичного та синтаксичного аналізатора

**Lexical\_Analys.cpp:** містить функції, що реалізують роботу лексичного аналізатору.

**Syntax\_Analys.cpp:** містить функції, що реалізують визначення структури речення.

**File\_manager.cpp:** містить допоміжні функції, такі як перевід деяких літер українського алфавіту на схожі літери латинського, вивід в файли результату роботи лексичного та синтаксичного аналізатора

**main.cpp:** містить у собі виклики основних функцій.



**Загальний опис модулів програми**

**Lexical\_Analys.cpp**

Містить в собі реалізацію лексичного аналізатора.

Функція lexem\* del\_sentence(string buf\_line)*;* - є лексичним аналізатором, що приймає на вхід строку та розбиває її на список, що складається з структур типу lexem, в якому визначенні поля для типу, довжини та самої лексеми.

Функції UserType analys\_word(char\* out\_string), serType analys\_const(char\* out\_string), призначенні для визначення типу лексеми, використовуючи булеві функції:

bool singlesymb(char ch) – тип односимвольної лексеми;

bool reg32(char\* out\_string) – тип ідентифікатору 32-бітного регістру;

bool reg16(char\* out\_string) – тип ідентифікатору 16-бітного регістру;

bool reg8(char\* out\_string) – тип ідентифікатору 8-бітного регістру;

bool machcommand(char\* out\_string) – тип ідентифікатору мнемокоду машинної інструкції;

bool segreg(char\* out\_string) – тип ідентифікатору сегментного регістру;

bool direct(char\* out\_string) – тип ідентифікатору директиви;

bool dirtype(char\* out\_string) – тип ідентифікатору директиви данних типу 1;

bool deftype(char\* out\_string) – тип ідентифікатору оператора визначення типу;

bool fourtype(char\* out\_string) – тип ідентифікатору типу 4;

bool const8(char\* out\_string) - тип вісімкової константи;

bool const16(char\* out\_string) – тип шістнадцяткової константи;

bool const2(char\* out\_string) – тип двійкової константи;

bool const10(char\* out\_string) – тип десяткової константи;

bool conststr(char\* out\_string) – тип строкової константи;

Після визначення лексеми, її типу, довжини, вони записуються до структури та додаються до списку.

**Syntax\_Analys.cpp**

Містить в собі функцію int\* sentence\_struct(lexem\* Parts\_sent, int flag)яка приймає на вхід вказівник на список лексем та кількість операндів, та повертає вказівник на масив дійсних чисел, у якому записані значення, відповідальні за впорядковане місцезнаходження імен та міток, мнемокодів та операндів.

**File\_manager.cpp**

string string\_correction(string buf\_line, int a) – призначена для коректування строки, тобто переводу деяких українських літер на схожі літери англійського алфавіту, та переведення символів з нижнього регістру у верхній;

int count\_operand(lexem\* buf\_struct) – призначена для підрахування кількості операндів у рядку;

void create\_lex\_analys() – призначена для створення аналізу чергового радка та запису в файл Lexical\_Analys.txt у табличному вигляді;

void create\_sintax\_analys() - призначена для створення структури поточного рядка та запису в файл Sintax\_Analys.txt у табличному вигляді;

**main.cpp**

Головна програма, в якій виконуються виклики функцій

void create\_lex\_analys(), void create\_sintax\_analys(), та коректне закінчення усієї програми.

**Додаток 1**

**Analys\_header.h**

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <string.h>

#include <stdio.h>

#include <ctype.h>

#include <iomanip>

#include <iostream>

#include <fstream>

#include <string>

using namespace std;

typedef enum

{

user\_identifier,

reg\_32,

reg\_16,

reg\_8,

machine\_command,

segment\_reg,

single\_symb,

directive,

dir\_type,

type\_4,

defin\_type,

const\_16,

const\_10,

const\_8,

const\_binary,

const\_string,

eof,

UNDEF

}UserType;

struct lexem

{

UserType TypeUser;

char\* lexem\_str;

int lexem\_length;

struct lexem\* next;

};

lexem\* del\_sentence(string buf\_line);

bool singlesymb(char ch);

bool reg32(char\* out\_string);

bool reg16(char\* out\_string);

bool reg8(char\* out\_string);

bool machcommand(char\* out\_string);

bool segreg(char\* out\_string);

bool direct(char\* out\_string);

bool dirtype(char\* out\_string);

bool deftype(char\* out\_string);

bool fourtype(char\* out\_string);

bool const8(char\* out\_string);

bool const16(char\* out\_string);

bool const2(char\* out\_string);

bool const10(char\* out\_string);

bool conststr(char\* out\_string);

UserType analys\_word(char\* out\_string);

UserType analys\_const(char\* out\_string);

void create\_lex\_analys();

string string\_correction(string buf\_line, int a);

int\* sentence\_struct(lexem\* Parts\_sent, int flag);

int count\_operand(lexem\* buf\_struct);

void create\_sintax\_analys();

**Lexical\_Analys.cpp**

#include "Analys\_header.h"

lexem\* del\_sentence(string buf\_line)

{

lexem \*buf;

lexem \*Parts\_sent;

Parts\_sent = (lexem\*)malloc(sizeof(lexem));

buf = Parts\_sent;

buf->next = NULL;

int a = buf\_line.size();

char in\_string[255];

int g = 0, flag = 0;

strncpy(in\_string, buf\_line.c\_str(), a);

in\_string[a] = '\0';

char ch = in\_string[0];

if (ch == '\0')

{

char\* bufch = (char\*)malloc(sizeof(char));

bufch[0] = '\0';

buf->lexem\_str = bufch;

buf->lexem\_length = 0;

buf->TypeUser = eof;

}

char\* out\_string = (char\*)malloc(sizeof(char));

out\_string[0] = '\0';

int length, i;

length = strlen(in\_string);

for (i = 0; i <= length - 1; i)

{

if (singlesymb(ch) == true)

{

if (ch == ';')

{

i = length;

}

else if (ch == ' ')

{

i++;

ch = in\_string[i];

}

else

{

char\* bufch = (char\*)malloc(sizeof(char));

bufch[0] = ch;

bufch[1] = '\0';

buf->lexem\_str = bufch;

buf->lexem\_length = 1;

buf->TypeUser = single\_symb;

i++;

ch = in\_string[i];

if (in\_string[i] != '\0' && in\_string[i] != ';')

{

buf->next = (lexem\*)malloc(sizeof(lexem));

buf = buf->next;

buf->next = NULL;

}

}

}

else

{

if (ch == '\'')

{

i++;

ch = in\_string[i];

g = 0;

out\_string[g] = ch;

g++;

i++;

ch = in\_string[i];

while (ch != '\'' && ch != '\0')

{

out\_string[g] = ch;

g++;

i++;

ch = in\_string[i];

}

if (ch == '\'')

{

buf->TypeUser = const\_string;

}

else

{

buf->TypeUser = user\_identifier;

}

out\_string[g] = '\0';

buf->lexem\_str = out\_string;

buf->lexem\_length = g;

i++;

ch = in\_string[i];

}

else

{

g = 0;

while (singlesymb(ch) == false)

{

out\_string[g] = ch;

g++;

i++;

ch = in\_string[i];

}

out\_string[g] = '\0';

buf->lexem\_str = out\_string;

buf->lexem\_length = g;

if (isalpha(out\_string[0]) != 0)

{

buf->TypeUser = analys\_word(out\_string);

}

else if (isdigit(out\_string[0]) != 0)

{

buf->TypeUser = analys\_const(out\_string);

}

}

if (in\_string[i] != '\0' && in\_string[i] != ';')

{

out\_string = (char\*)malloc(sizeof(char));

buf->next = (lexem\*)malloc(sizeof(lexem));

buf = buf->next;

buf->next = NULL;

}

}

}

return Parts\_sent;

}

bool singlesymb(char ch)

{

if (ch == ';' || ch == ' ' || ch == ',' || ch == '+' || ch == '-' || ch == ':' || ch == '[' || ch == ']' || ch == '\*' || ch == '(' || ch == ')' || ch == '=' || ch == '\0')

{

return true;

}

else

{

return false;

}

}

bool reg32(char\* out\_string)

{

if (strcmp(out\_string, "EAX") == 0 || strcmp(out\_string, "ECX") == 0 || strcmp(out\_string, "EDX") == 0 || strcmp(out\_string, "EBX") == 0 || strcmp(out\_string, "EBP") == 0 || strcmp(out\_string, "ESI") == 0 || strcmp(out\_string, "EDI") == 0 || strcmp(out\_string, "ESP") == 0)

{

return true;

}

else

{

return false;

}

}

bool reg16(char\* out\_string)

{

if (strcmp(out\_string, "AX") == 0 || strcmp(out\_string, "CX") == 0 || strcmp(out\_string, "DX") == 0 || strcmp(out\_string, "BX") == 0 || strcmp(out\_string, "SP") == 0 || strcmp(out\_string, "BP") == 0 || strcmp(out\_string, "SI") == 0 || strcmp(out\_string, "DI") == 0)

{

return true;

}

else

{

return false;

}

}

bool reg8(char\* out\_string)

{

if (strcmp(out\_string, "AL") == 0 || strcmp(out\_string, "AH") == 0 || strcmp(out\_string, "BL") == 0 || strcmp(out\_string, "BH") == 0 || strcmp(out\_string, "CL") == 0 || strcmp(out\_string, "CH") == 0 || strcmp(out\_string, "DL") == 0 || strcmp(out\_string, "DH") == 0)

{

return true;

}

else

{

return false;

}

}

bool machcommand(char\* out\_string)

{

if (strcmp(out\_string, "STI") == 0 || strcmp(out\_string, "MOV") == 0 || strcmp(out\_string, "PUSH") == 0 || strcmp(out\_string, "DIV") == 0 || strcmp(out\_string, "ADC") == 0 || strcmp(out\_string, "SUB") == 0 || strcmp(out\_string, "XOR") == 0 || strcmp(out\_string, "JZ") == 0 || strcmp(out\_string, "JMP") == 0)

{

return true;

}

else

{

return false;

}

}

bool segreg(char\* out\_string)

{

if (strcmp(out\_string, "DS") == 0 || strcmp(out\_string, "SS") == 0 || strcmp(out\_string, "ES") == 0 || strcmp(out\_string, "CS") == 0)

{

return true;

}

else

{

return false;

}

}

bool direct(char\* out\_string)

{

if (strcmp(out\_string, "SEGMENT") == 0 || strcmp(out\_string, "ENDS") == 0 || strcmp(out\_string, "END") == 0 || strcmp(out\_string, "ASSUME") == 0)

{

return true;

}

else

{

return false;

}

}

bool dirtype(char\* out\_string)

{

if (strcmp(out\_string, "DB") == 0 || strcmp(out\_string, "DW") == 0 || strcmp(out\_string, "EQU") == 0 || strcmp(out\_string, "DD") == 0)

{

return true;

}

else

{

return false;

}

}

bool deftype(char\* out\_string)

{

if (strcmp(out\_string, "PTR") == 0)

{

return true;

}

else

{

return false;

}

}

bool fourtype(char\* out\_string)

{

if (strcmp(out\_string, "BYTE") == 0 || strcmp(out\_string, "WORD") == 0 || strcmp(out\_string, "DWORD") == 0 || strcmp(out\_string, "QWORD") == 0)

{

return true;

}

else

{

return false;

}

}

bool const8(char\* out\_string)

{

if (out\_string[strlen(out\_string) - 1] == 'O' || out\_string[strlen(out\_string) - 1] == 'Q')

{

return true;

}

else

{

return false;

}

}

bool const16(char\* out\_string)

{

if (out\_string[strlen(out\_string) - 1] == 'H')

{

return true;

}

else

{

return false;

}

}

bool const2(char\* out\_string)

{

if (out\_string[strlen(out\_string) - 1] == 'B')

{

return true;

}

else

{

return false;

}

}

bool const10(char\* out\_string)

{

if (out\_string[strlen(out\_string) - 1] == 'D' || isdigit(out\_string[strlen(out\_string) - 1]) != 0)

{

return true;

}

else

{

return false;

}

}

bool conststr(char\* out\_string)

{

if (out\_string[strlen(out\_string) - strlen(out\_string)] == '\'' && out\_string[strlen(out\_string) - 1] == '\'' || out\_string[strlen(out\_string) - strlen(out\_string)] == '\"' && out\_string[strlen(out\_string) - 1] == '\"')

{

return true;

}

else

{

return false;

}

}

UserType analys\_word(char\* out\_string)

{

if (direct(out\_string) == true)

{

return directive;

}

if (reg32(out\_string) == true)

{

return reg\_32;

}

if (reg16(out\_string) == true)

{

return reg\_16;

}

if (reg8(out\_string) == true)

{

return reg\_8;

}

if (reg32(out\_string) == true)

{

return reg\_32;

}

if (machcommand(out\_string) == true)

{

return machine\_command;

}

if (segreg(out\_string) == true)

{

return segment\_reg;

}

if (dirtype(out\_string) == true)

{

return dir\_type;

}

if (deftype(out\_string) == true)

{

return defin\_type;

}

if (fourtype(out\_string) == true)

{

return type\_4;

}

return user\_identifier;

}

UserType analys\_const(char\* out\_string)

{

if (const2(out\_string) == true)

{

return const\_binary;

}

if (const8(out\_string) == true)

{

return const\_8;

}

if (const10(out\_string) == true)

{

return const\_10;

}

if (const16(out\_string) == true)

{

return const\_16;

}

if (conststr(out\_string) == true)

{

return const\_string;

}

return UNDEF;

}

**Sintax\_Analys.cpp**

#include "Analys\_header.h"

int\* sentence\_struct(lexem\* Parts\_sent, int flag)

{

int count = 0, count\_lex = 0, j;;

if (flag == 0)

{

j = 6;

}

else

{

j = 2 + (2 \* flag - (flag - 1)) \* 2;

}

int\* argz = new int[j];

char\* bufch = new char;

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

{

argz[i] = 0;

}

if (Parts\_sent->TypeUser == eof)

{

}

else

{

if (Parts\_sent->TypeUser == user\_identifier)

{

argz[0] = 1; count = 1;

Parts\_sent = Parts\_sent->next;

if (strchr(Parts\_sent->lexem\_str, ':') != NULL)

{

Parts\_sent = Parts\_sent->next;

if (Parts\_sent != NULL)

if (Parts\_sent->TypeUser == directive || Parts\_sent->TypeUser == dir\_type || Parts\_sent->TypeUser == machine\_command)

{

argz[1] = 3; count = 3; argz[2] = 1;

Parts\_sent = Parts\_sent->next;

}

}

if (Parts\_sent != NULL)

if (Parts\_sent->TypeUser == directive || Parts\_sent->TypeUser == dir\_type || Parts\_sent->TypeUser == machine\_command || strchr(Parts\_sent->lexem\_str, '=') != NULL)

{

argz[1] = 2; argz[2] = 1; count = 2;

Parts\_sent = Parts\_sent->next;

}

}

else

{ if (Parts\_sent->TypeUser == directive || Parts\_sent->TypeUser == dir\_type || Parts\_sent->TypeUser == machine\_command)

{

argz[0] = 0; argz[1] = 1; argz[2] = 1; count = 1;

Parts\_sent = Parts\_sent->next;

}

}

if (Parts\_sent != NULL)

{

argz[3] = count + 1;

count\_lex = 0;

while (Parts\_sent != NULL && strchr(Parts\_sent->lexem\_str, ',') == NULL)

{

count\_lex++; count++;

Parts\_sent = Parts\_sent->next;

}

argz[4] = count\_lex;

if (Parts\_sent != NULL)

{

if (strchr(Parts\_sent->lexem\_str, ',') != NULL)

{

argz[5] = count + 2; count\_lex = 0;

Parts\_sent = Parts\_sent->next;

if (flag = 2)

{

while (Parts\_sent != NULL && strchr(Parts\_sent->lexem\_str, ',') == NULL)

{

count\_lex++; count++;

Parts\_sent = Parts\_sent->next;

}

argz[6] = count\_lex;

if (Parts\_sent != NULL)

{

if (strchr(Parts\_sent->lexem\_str, ',') != NULL)

{

argz[7] = count + 3; count\_lex = 0;

Parts\_sent = Parts\_sent->next;

while (Parts\_sent != NULL)

{

count\_lex++; count++;

Parts\_sent = Parts\_sent->next;

}

argz[8] = count\_lex;

}

}

}

while (Parts\_sent != NULL)

{

count\_lex++; count++;

Parts\_sent = Parts\_sent->next;

}

argz[6] = count\_lex;

}

}

}

}

return argz;

}

**File\_manager.cpp**

#include "Analys\_header.h"

void create\_lex\_analys()

{

string buf\_line;

lexem\* buf\_struct;

ifstream file\_test;

ofstream file\_lexem;

int count\_line = 1, count\_lex;

file\_test.open("D:\\Documents\\2 курс\\2 семестр\\СП\\Course\\CourseWork\_Jonh\\test.asm");

file\_lexem.open("D:\\Documents\\2 курс\\2 семестр\\СП\\Course\\CourseWork\_Jonh\\Lexical\_Analys.txt");

if (file\_test.is\_open())

{

while (!file\_test.eof())

{

getline(file\_test, buf\_line);

int a = buf\_line.size();

buf\_struct = del\_sentence(string\_correction(buf\_line, a));

file\_lexem << "РЯДОК №" << count\_line << endl;

file\_lexem << "---------------------------------------------------------------------------------------" << endl;

file\_lexem << "| № | ЛЕКСЕМА | ДОВЖИНА | ТИП ЛЕКСЕМИ |" << endl;

file\_lexem << "---------------------------------------------------------------------------------------" << endl;

count\_lex = 1;

while (buf\_struct != NULL)

{

if (buf\_struct->TypeUser == eof)

{file\_lexem << "ПОРОЖНІЙ РЯДОК" << endl;

}

else

{

file\_lexem << '|' << setw(3) << count\_lex << setw(3) << '|';

file\_lexem << setw(21) << buf\_struct->lexem\_str << '|';

file\_lexem << setw(7) << buf\_struct->lexem\_length << setw(7) << '|';

if (buf\_struct->TypeUser == user\_identifier)

{

file\_lexem << setw(43) << "Iдентифікатор користувача або невизначенний" << '|' << endl;

}

else

if (buf\_struct->TypeUser == reg\_32)

{

file\_lexem << setw(43) << "Iдентифікатор 32-бітного регістру" << '|' << endl;

}

else

if (buf\_struct->TypeUser == reg\_16)

{

file\_lexem << setw(43) << "Iдентифікатор 16-бітного регістру" << '|' << endl;

}

else

if (buf\_struct->TypeUser == reg\_8)

{

file\_lexem << setw(43) << "Iдентифікатор 8-бітного регістру" << '|' << endl;

}

else

if (buf\_struct->TypeUser == machine\_command)

{

file\_lexem << setw(43) << "Ідентифікатор мнемокоду машинної інструкції" << '|' << endl;

}

else

if (buf\_struct->TypeUser == segment\_reg)

{

file\_lexem << setw(43) << "Ідентифікатор сегментного регістру" << '|' << endl;

}

else

if (buf\_struct->TypeUser == single\_symb)

{

file\_lexem << setw(43) << "Односимвольна" << '|' << endl;

}

else

if (buf\_struct->TypeUser == directive)

{

file\_lexem << setw(43) << "Ідентифікатор директиви" << '|' << endl;

}

else

if (buf\_struct->TypeUser == dir\_type)

{

file\_lexem << setw(43) << "Ідентифікатор директиви данних тип 1" << '|' << endl;

}

else

if (buf\_struct->TypeUser == type\_4)

{

file\_lexem << setw(43) << "Ідентифікатор тип 4" << '|' << endl;

}

else

if (buf\_struct->TypeUser == defin\_type)

{

file\_lexem << setw(43) << "Ідентифікатор оператора визначення типу" << '|' << endl;

}

else

if (buf\_struct->TypeUser == const\_16)

{

file\_lexem << setw(43) << "Шіснадцяткова константа" << '|' << endl;

}

else

if (buf\_struct->TypeUser == const\_10)

{

file\_lexem << setw(43) << "Десяткова константа" << '|' << endl;

}

else

if (buf\_struct->TypeUser == const\_8)

{

file\_lexem << setw(43) << "Вісімкова константа" << '|' << endl;

}

else

if (buf\_struct->TypeUser == const\_binary)

{

file\_lexem << setw(43) << "Двiйкова константа" << '|' << endl;

}

else

if (buf\_struct->TypeUser == const\_string)

{

file\_lexem << setw(43) << "Строкова константа" << '|' << endl;

}

file\_lexem << "---------------------------------------------------------------------------------------" << endl;

}

buf\_struct = buf\_struct->next;

count\_lex++;

}

count\_line++;

}

file\_test.close();

}

else cout << "Unable to open file";

return;

}

string string\_correction(string buf\_line, int a)

{

char buf\_ch;

if (buf\_line.empty()) return buf\_line;

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

{

if (buf\_line[i] == 'а') buf\_line[i] = 'a';

else if (buf\_line[i] == 'А') buf\_line[i] = 'A';

else if (buf\_line[i] == 'е') buf\_line[i] = 'e';

else if (buf\_line[i] == 'Е') buf\_line[i] = 'E';

else if (buf\_line[i] == 'і') buf\_line[i] = 'i';

else if (buf\_line[i] == 'І') buf\_line[i] = 'I';

else if (buf\_line[i] == 'у') buf\_line[i] = 'y';

else if (buf\_line[i] == 'о') buf\_line[i] = 'o';

else if (buf\_line[i] == 'О') buf\_line[i] = 'O';

else if (buf\_line[i] == 'р') buf\_line[i] = 'p';

else if (buf\_line[i] == 'Р') buf\_line[i] = 'P';

else if (buf\_line[i] == 'Н') buf\_line[i] = 'H';

else if (buf\_line[i] == 'К') buf\_line[i] = 'K';

else if (buf\_line[i] == 'х') buf\_line[i] = 'x';

else if (buf\_line[i] == 'Х') buf\_line[i] = 'X';

else if (buf\_line[i] == 'с') buf\_line[i] = 'c';

else if (buf\_line[i] == 'С') buf\_line[i] = 'C';

else if (buf\_line[i] == 'В') buf\_line[i] = 'B';

else if (buf\_line[i] == 'М') buf\_line[i] = 'M';

else if (buf\_line[i] == 'Т') buf\_line[i] = 'T';

buf\_ch = buf\_line[i];

buf\_line[i] = toupper(buf\_ch);

} return buf\_line;

}

int count\_operand(lexem\* buf\_struct)

{

int flag = 0;

while (buf\_struct != NULL)

{

if (strchr(buf\_struct->lexem\_str, ','))

flag++;

buf\_struct = buf\_struct->next;

}

return flag;

}

void create\_sintax\_analys()

{

string buf\_line;

lexem\* buf\_struct;

ifstream file\_test;

ofstream file\_sintax;

int count\_line = 1, flag;

int\* buf\_vect;

file\_test.open("D:\\Documents\\2 курс\\2 семестр\\СП\\Course\\CourseWork\_Jonh\\test.asm");

file\_sintax.open("D:\\Documents\\2 курс\\2 семестр\\СП\\Course\\CourseWork\_Jonh\\Sintax\_Analys.txt");

if (file\_test.is\_open())

{

while (!file\_test.eof())

{

getline(file\_test, buf\_line);

int a = buf\_line.size();

buf\_struct = del\_sentence(string\_correction(buf\_line, a));

flag = count\_operand(buf\_struct);

buf\_vect = sentence\_struct(buf\_struct, flag);

if (flag != 2)

{

file\_sintax << "РЯДОК №" << count\_line << endl;

int j = 6, i = 0;

file\_sintax << setw(12) << buf\_vect[i] << '|';

for ( i = 1; i <= j; i++)

{

file\_sintax << setw(11) << buf\_vect[i] << '|' ;

}

file\_sintax << endl;

}

else

{

file\_sintax << "РЯДОК №" << count\_line << endl;

int j = 2 + (2 \* flag - (flag - 1)) \* 2, i = 0;

file\_sintax << setw(12) << buf\_vect[i] << '|';

for (i = 1; i <= j; i++)

{

file\_sintax << setw(11) << buf\_vect[i] << '|';

}

file\_sintax << endl;

}

count\_line++;

}

file\_test.close();

}

else cout << "Unable to open file";

return;

}

**main.cpp**

#include "Analys\_header.h"

int main()

{

create\_lex\_analys();

create\_sintax\_analys();

}

**Тестове завдання без помилок:**

data1 segment

VAL1 db 11010b

VAL2 dw 15

VAL3 dd 0abcdh

STR db 'рядок'

SMTH equ [ebx

NINE = 178

data1 ends

data2 segment

VAL4 db 4h

VAL5 dw 0b9h

data2 ends

code1 segment

assume cs:code1, ds:data1

start:

sti

jz M1

jmp M1

push ecx

div dword ptr SMTH+edi]

xor dword ptr es:[esp+ebp], 0ah

adc eax, ecx

mov ecx, ds:[bx+di]

sub eax, ds:[bp+ di]

xor ds:val1, NINE

jz start

M1:

jmp start

jmp start2

code1 ends

code2 segment

assume cs:code2, ds:data1, es:data2

start2:

push ecx

adc eax, ecx

NINE = 50

xor ds:val1, NINE

jmp start

end start