Приложение.

**“main.cpp”**

#include <iostream>

#include "lexic.h"

#include "syntax.h"

using namespace std;

int main(int argc, char\* argv[])

{

Lexic L;

Syntax S;

int a;

setlocale(0, "rus");

cout << "Какую программу компилировать(1 или 2)?\n";

cin >> a;

if (a == 1)// если передаем аргументы, то argc будет больше 1(в зависимости от кол-ва аргументов)

{

L.openfile("input.mod");// вывод второй строки из массива указателей на строки(нумерация в строках начинается с 0 )

}

else if(a==2)

{

L.openfile("input2.mod");

}

if(L.Lexanalys())

{

if(S.Synanalys())

{

if(S.GenerCode())

{

cout << "Компиляция завершена!!!" << endl;

}

}

}

return 0;

}

**“lexic.h”**

#ifndef LEXIC\_H

#define LEXIC\_H

#include <fstream>

#include <string> //подключение библиотек

#include <iostream>

#include <vector>

using namespace std;

class Lexic

{

ifstream cin\_code; //файл с исходной программой

ifstream cin\_mainwords; //файл с ключевыми словами исходного языка

ofstream out\_code;

vector <string> Lexems; //массив лексем

string mp; //автомат

bool flagS, flagN, flagO, flagL; //флаги накопления слова, числа и оператора соответственно

char ch; //входящий символ

struct list1 //список ключевых слов и т.д.

{

string word; //идентификатор

int num; //номер в списке

list1 \*next; //ссылка на следующий

} \*variables, \*constants, \*keywords; // списки переменных, констант и ключевых слов

struct list2 //список операций

{

string word; //идентификатор

char id; //символ типа операции

int num; //номер операции среди операций того же типа

list2 \*next; //ссылка на следующий элемент

} \*operations; //список операций

int numstroke;

public:

Lexic();

int add0(list1 \*&head, string w); //добавление в список 1 типа при неизвестном номере

void add1(list1 \*&head, string w, int n); //добавление в очередь 1 типа при известном номере

void add2(list2 \*&head, string w, char id, int n); //добавление в список 2 типа

bool sym(); //является ли символ «символом»

string numtostr(int);

void openfile(char \*);

bool num(); //является ли символ цифрой

bool oper(); //является ли символ знаком операции

bool separ(); //является ли символ разделителем

void mpclear(); //очистка магазина

void InsertWord(); //добавление ключевого слова или переменной

void InsertNumber(); //добавление числовой константы

void InsertOper(); //добавление операции

void Insert(); //добавление токена в зависимости от его типа

void Error(); //обработчик ошибок

void automat(char get); //функция, реализующая действие автомата

void printLex();

bool endfile();

char getsym();

int Lexanalys();

void closefiles();

};

#endif

**“lexic.cpp”**

#include "lexic.h"

#include <fstream>

#include <string> //подключение библиотек

#include <iostream>

#include <vector>

using namespace std;

Lexic::Lexic()

{

//cin\_code.open("input.mod");

cin\_mainwords.open("mainwords.txt");

out\_code.open("temp.txt");

variables = NULL;

operations = NULL;

constants = NULL;

numstroke=1;

keywords = NULL; //обнуление указателей на начала списков

flagL=flagN=flagO=flagS=false; //установка начальных состояний автомата

int k;

char q; //символ типа операции

cin\_mainwords >> k >> q;

string s;

for (int i = 0; i < k; i++) //чтение ключевых слов из файла и помещение в список

{

cin\_mainwords >> s;

add1(keywords, s, i);

}

//print2(keywords, "k");

cin\_mainwords >> k;

for (int i = 0; i < k; i++) //чтение операций из файла и помещение в список

{

int n;

cin\_mainwords >> n >> q;

for (int j = 0; j < n; j++) //чтение операций типа ch

{

cin\_mainwords >> s;

add2(operations, s, q, j);

}

}

}

void Lexic::openfile(char \*s)

{

cin\_code.open(s);

}

int Lexic::Lexanalys()

{

while (!endfile()) //пока не достигнут конец файла

{

automat(getsym()/\*читаем символ\*/); //запускаем автомат

}

automat(' '); //запускаем автомат с пробельным символом для обработки содержимого магазина на последнем шаге

printLex(); //вывод списков

closefiles();

return 1;

}

void Lexic::closefiles()

{

cin\_code.close();

cin\_mainwords.close();

out\_code.close();

}

int Lexic::add0(list1 \*&head, string w) //добавление в список 1 типа при неизвестном номере

{

list1 \*temp = new list1; //создание нового элемента

temp -> next = NULL;

temp -> word = w; //присвоение новому элементу идентификатора w

if (head) //если список не пуст

{

list1 \*t;

for (t=head; t -> next != NULL; t = t -> next);

temp -> num = t -> num + 1;

t -> next = temp; //добавление в хвост очереди

}

else

{

temp -> num = 0;

head = temp;

}

return temp -> num; //возвращаем номер добавленного элемента

}

void Lexic::add1(list1 \*&head, string w, int n) //добавление в очередь 1 типа при известном номере

{

list1 \*temp = new list1; //аналогично add0

temp -> word = w;

temp -> next = NULL;

temp -> num = n;

if (head)

{

list1 \*t;

for (t=head; t -> next != NULL; t = t -> next);

t -> next = temp;

}

else

{

head = temp;

}

}

void Lexic::add2(list2 \*&head, string w, char id, int n) //добавление в список 2 типа

{

list2 \*temp = new list2;

temp -> next = NULL;

temp -> id = id;

temp -> word = w;

temp -> num = n;

if (head)

{

list2 \*t;

for (t=head; t -> next != NULL; t = t -> next);

t -> next = temp;

}

else

{

head = temp;

}

}

bool Lexic::sym() //является ли символ «символом»

{

return ((ch >= 'A' && ch <= 'Z') || (ch >= 'a' && ch <= 'z') || (ch == '\_')) ? true : false;

}

bool Lexic::num() //является ли символ цифрой

{

return (ch >= '0' && ch <= '9') ? true : false;

}

bool Lexic::oper() //является ли символ знаком операции

{

return (ch =='"' || ch==':' || ch== ',' || ch=='.' || ch == '+' || ch == '-' || ch == '\*' || ch == '/' || ch == '>' || ch == '<' || ch == '&' || ch == '|' || ch == '=' || ch == ';' || /\*ch == '(' || ch == ')' ||\*/ ch == '{' || ch == '}') ? true : false;

}

bool Lexic::separ() //является ли символ разделителем

{

return (ch == '"' || ch=='[' || ch==']' || ch== ',' || ch=='.' || ch == ' ' || ch == '\n' || ch == ')' || ch == '(' || ch == ';' || ch == '}' || ch == '{' || (!flagO && mp != "" && oper())) ? true : false;

}

void Lexic::mpclear() //очистка магазина

{

mp = "";

}

void Lexic::InsertWord() //добавление ключевого слова или переменной

{

//char ch1;

string w;

list1 \*temp;

for (temp=keywords; temp!=NULL; temp = temp -> next) //поиск в списке констант

{

if (temp -> word == mp)

{

w = "k";

/\*ch1 = '0' + temp -> num;

w += ch1;\*/

w += " " + temp -> word + " " + numtostr(numstroke);

Lexems.push\_back(w);

return;

}

}

for (temp=variables; temp!=NULL; temp = temp -> next) //поиск в списке переменных

{

if (temp -> word == mp)

{

w = "j";

w += " " + temp -> word + " " + numtostr(numstroke);

Lexems.push\_back(w);

return;

}

}

//int nnum = add0(variables, mp); //добавление новой переменной

w = "j";

w += " " + mp + " " + numtostr(numstroke);

Lexems.push\_back(w);

//cout << w << endl;

}

void Lexic::InsertNumber() //добавление числовой константы

{

//char ch1;

string w;

list1 \*temp;

for (temp=constants; temp!=NULL; temp = temp -> next) //поиск в списке констант

if (temp -> word == mp)

{

w = "c";

w += " " + temp -> word + " " + numtostr(numstroke);

Lexems.push\_back(w);

return;

}

//int nnum = add0(constants, mp); //добавление новой константы

w = "c";

w += " " + mp + " " + numtostr(numstroke);

Lexems.push\_back(w);

}

void Lexic::InsertOper() //добавление операции

{

//char ch1;

string w;

list2 \*temp;

for (temp=operations; temp != NULL; temp = temp -> next)

//поиск в списке операций

if (temp -> word == mp)

{

w = temp -> id;

w += " " + temp -> word + " " + numtostr(numstroke);

Lexems.push\_back(w);

return;

}

}

string Lexic::numtostr(int num)

{

char buf[5];

string temp="";

itoa(num, buf, 10);

for(int i=0 ; i<5 && buf[i]!='\0'; i++) temp+=buf[i];

//temp+="\0";

return temp;

}

void Lexic::Insert() //добавление токена в зависимости от его типа

{

string w="";

if (flagS) //если токен – ключевое слово или имя переменной

InsertWord();

if (flagN) //если токен – числовая константа

InsertNumber();

if (flagO) //если токен - операция

InsertOper();

if(flagL)

{

w = "l";

w += " " + mp + " " + numtostr(numstroke);

Lexems.push\_back(w);

}

mpclear(); //очистка магазина

}

void Lexic::Error() //обработчик ошибок

{

Lexems.push\_back("Error"); //вывод сообщения об ошибке

while (!separ() && cin\_code.peek()!=EOF) //пропуск символов до следующего разделителя

cin\_code >> ch;

mpclear();

}

void Lexic::automat(char get) //функция, реализующая действие автомата

{

string w;

static char last=' ';

ch=get;

int h=0;

static int q=0;

if(q==1)

{

numstroke++;

q=0;

}

if(ch=='\n')

{

q=1;

}

if(ch=='"')

{

//Insert();

ch=getsym();

mp="\"";

while(ch!='"')

{

mp+=ch;

ch=getsym();

}

mp+=ch;

flagL=true;

flagS = flagN = flagO = false;

Insert();

flagL=false;

mp=")";

flagO=true;

Insert();

return;

}

if (mp.length() == 0) //если магазин пуст

{

last=ch;

if (!(ch == ' ' || ch == '\n')) //если символ не пробел и не маркер конца строки

{

flagS = sym();

flagN = num();

flagO = oper(); //определяем тип символа

mp += ch; //добавляем символ в магазин

return;

}

else return;

}

if(ch=='\*' && last=='(')

{

while(h!=1)

{

do

{

ch=getsym();

}while(ch!='\*');

if(getsym()!=')') continue;

else h=1;

}

h=0;

return;

}

if (!separ() || (flagN && ch=='.')) //если символ - не разделитель

{

if ((sym() || num()) && flagS)

//если накапливаем «слово» и текущий символ – «символ» или цифра

mp += ch;

else if ((num() || ch=='.') && flagN)

mp += ch;

//если накапливаем число и текущий символ - цифра или точка

else if (oper() && flagO)

//если накапливаем операцию и текущий символ – символ операции

mp += ch;

else if (flagN && sym())

//если накапливаем число, а текущий символ – «символ», то ошибка

Error();

else

{

if(mp=="[")

{

w="i ";

w += mp + " " + numtostr(numstroke);

Lexems.push\_back(w);

}

Insert();

flagS = sym();

flagN = num();

flagO = oper();

mp += ch; //добавляем символ в магазин

}

last = ch;

return;

}

if (separ())

{

if (oper()) //если разделитель – символ операции

{

Insert();

flagS = sym();

flagN = num();

flagO = oper();

mp = ch;

}

else

{

Insert();

if(ch=='(' || ch==')')

{

w="s ";

w += ch;

w+= " " + numtostr(numstroke);

Lexems.push\_back(w);

}

else if(ch=='[' || ch==']')

{

w="i ";

w += ch;

w+= " " + numtostr(numstroke);

Lexems.push\_back(w);

}

}

last = ch;

}

}

void Lexic::printLex()

{

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

out\_code << Lexems[i] << endl;

}

bool Lexic::endfile()

{

return cin\_code.peek()==EOF ? true : false;

}

char Lexic::getsym()

{

char q;

q=cin\_code.get();

return q;

}

**“syntax.h”**

#ifndef SYNTAX\_H

#define SYNTAX\_H

#include <stack>

#include <vector>

#include <iostream>

#include <clocale>

#include <fstream>

#include <string> //подключение библиотек

using namespace std;

class ASTnode

{

int Type;

string Text;

ASTnode \*parent;

vector <ASTnode\*> childs;

public:

ASTnode(int type, string text, ASTnode \*child1, ASTnode \*child2);

void printtext();

int AddChild(ASTnode \*child);

void settext(string qwe);

int curpar();

ASTnode \*retchild(int);

string childtext(int);

void setparent(ASTnode \*qwe);

void settype(int qwe);

int rettype();

string rettext();

int RemoveChild(ASTnode \*child);

void ASTremove(ASTnode \*node);

void printnode(ASTnode \*node);

int childscount();

};

class Syntax

{

ASTnode \*root;

fstream code;

ofstream mod;

string word, t, nam, page, currentproc;

struct tableid

{

string var, type, proc, zn;

int mas, kol;

}rty;

struct forindex

{

string jk1, jk2;

}jok;

vector <forindex> fi;

vector <tableid> TI;

int realvar;

int metka;

int varnum;

int joknum;

int

UNKNOWN,

ADD,

SUB,

MUL,

DIV,

MOD,

STR,

ASSIGN,

KEYWORD,

SEPARATOR,

IDENT,

CONSTANT,

PROGRAMM,

PROCEDURE,

ERROR,

NONTERMINAL,

OR,

AND,

EQ,

NE,

NOT,

SRAV,

OPCIKLE,

OPIF,

OPELSE,

OPELSIF,

TERMINAL,

fparam;

public:

Syntax();

~Syntax();

void scancode();

string idstringtype(string);

void while\_cikle(ASTnode \*);

int GSTPS(string);

string HelpStr(string);

int TIcheck(string);

int Synanalys();

void ifcode(ASTnode \*);

void ProvCikl(ASTnode \*);

void printjok();

int ischislo(string);

//int typeof(string);

ASTnode\* START();

ASTnode\* MODULE();

ASTnode\* NAME();

ASTnode \*O();

ASTnode \*P();

void setindex(ASTnode \*);

string getindex(string);

void printdata();

ASTnode \*PO();

void opproc(ASTnode\*);

ASTnode \*IFELSE();

ASTnode \*CIKL();

ASTnode \*OO();

ASTnode \*A();

void printid();

int check(string w);

ASTnode \*BOOL();

ASTnode \*JOIN();

ASTnode \*EQUA();

ASTnode \*REL();

ASTnode \*ARIFM();

ASTnode \*term();

ASTnode \*unary();

ASTnode \*factor();

ASTnode \*offset(string qwe);

string sidtype(string w);

int idtype(string w);

void FACTPAR(ASTnode \*);

ASTnode \*F();

void prisv(ASTnode\*);

ASTnode \*FORMALPAR();

ASTnode \*C();

ASTnode \*C1();

ASTnode\* CHISLO();

ASTnode\* V();

ASTnode\* TYPE();

int GenerCode();

};

#endif

**“syntax.cpp”**

#include "syntax.h"

#include <stack>

#include <vector>

#include <iostream>

#include <clocale>

#include <fstream>

#include <string> //подключение библиотек

using namespace std;

ASTnode::ASTnode(int type, string text, ASTnode \*child1, ASTnode \*child2)

{

parent = NULL;

Type = type;

Text = text;

if(child1 != NULL) AddChild(child1);

if(child2 != NULL) AddChild(child2);

}

void ASTnode::printtext()

{

cout << this->Text;

}

ASTnode\* ASTnode::retchild(int i)

{

return this->childs[i];

}

string ASTnode::rettext()

{

return this->Text;

}

string ASTnode::childtext(int i)

{

return this->childs[i]->Text;

}

int ASTnode::AddChild(ASTnode \*child)

{

if(child==NULL) return 0;

childs.push\_back(child);

child->parent = this;

return 1;

}

void ASTnode::settext(string qwe)

{

Text=qwe;

}

int ASTnode::curpar()

{

if(this->parent==NULL) return 0;

else return 1;

}

int ASTnode::rettype()

{

return this->Type;

}

void ASTnode::setparent(ASTnode \*qwe)

{

parent=qwe;

}

void ASTnode::settype(int qwe)

{

Type=qwe;

}

int ASTnode::RemoveChild(ASTnode \*child)

{

if (child->parent == this) child->parent = NULL;

return 2;

}

void ASTnode::ASTremove(ASTnode \*node)

{

if(node==NULL) return;

for(int i=0 ; i<(int)node->childs.size() ; i++)

{

ASTremove(node->childs[i]);

}

node->childs.clear();

delete node;

}

void ASTnode::printnode(ASTnode \*node)

{

static int g=0;

if(node==NULL) return;

cout << node->Text <<endl;

for(int i=0 ; i<(int)node->childs.size() ; i++)

{

for(int j=0 ; j<g ; j++) cout << " ";

cout << "|";

g++;

printnode(node->childs[i]);

}

g--;

}

int ASTnode::childscount()

{

return this->childs.size();

}

Syntax::Syntax()

{

mod.open("gencode.asm");

varnum=1;

ERROR = 404;

NONTERMINAL = 101;

UNKNOWN = 0;

KEYWORD=1;

CONSTANT = 2;

IDENT = 3;

ADD=11;

SUB=12;

MUL = 13;

DIV = 14;

MOD = 15;

ASSIGN = 16;

STR=71;

SEPARATOR = 52;

OR = 17;

AND=18;

EQ = 19;

NE=20;

NOT=21;

PROCEDURE=200;

SRAV=22;

joknum=0;

realvar=0;

OPCIKLE=30;

OPIF=31;

OPELSE=32;

OPELSIF=33;

TERMINAL=100;

root = NULL;

fparam=0;

metka=0;

code.open("temp.txt", ios::in);

currentproc="glob";

}

Syntax::~Syntax()

{

mod.close();

code.close();

root->ASTremove(root);

}

void Syntax::printdata()

{

mod << ".386" << endl;

mod << "MODEL FLAT, STDCALL" << endl;

mod << ".STACK 4096" << endl;

mod << "include \\masm32\\include\\masm32.inc" << endl;

mod << "include \\masm32\\include\\user32.inc" << endl;

mod << "include \\masm32\\include\\kernel32.inc" << endl;

mod << "includelib \\masm32\\lib\\masm32.lib" << endl;

mod << "includelib \\masm32\\lib\\user32.lib" << endl;

mod << "includelib \\masm32\\lib\\kernel32.lib" << endl;

mod << ".data" << endl;

mod << "prnt\_scan\_int DB \"%d\",0" << endl;

mod << "prnt\_scan\_real DB \"%f\",0" << endl;

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

{

if(TI[i].proc!="glob" && TI[i].type!="PROCEDURE")

{

mod << "proc\_" << TI[i].proc << " " << TI[i].var << " DD " << TI[i].zn << endl;

}

else if(TI[i].proc=="glob" && TI[i].type!="PROCEDURE")

{

if(TI[i].type=="string")

{

mod << "var\_" << TI[i].kol << " DB " << TI[i].var << ",0" << endl;

}

else

{

switch(idtype(TI[i].var))

{

case 4:

mod << TI[i].var << " DD " << TI[i].zn << endl;

break;

}

}

}

}

mod << ".code" << endl;

mod << "main PROC" << endl;

mod << "FINIT" << endl;

}

void Syntax::printjok()

{

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

{

if(ischislo(fi[i].jk1)==0)

{

mod << "MOV " << fi[i].jk2 << "," << fi[i].jk1 << endl;

}

}

}

void Syntax::ProvCikl(ASTnode \*current)

{

for(int j=0 ; j<current->childscount() ; j++)

{

if(current->retchild(j)->rettext()=="WHILE")

{

while\_cikle(current->retchild(j));

}

else if(current->retchild(j)->rettext()=="IF")

{

ifcode(current->retchild(j));

}

else if(current->retchild(j)->rettext()!=":=")

{

opproc(current->retchild(j));

}

else

{

if(current->retchild(j)->retchild(1)->childscount()==0)

{

mod << "MOV eax," << current->retchild(j)->retchild(1)->rettext() << endl;

mod << "MOV " << current->retchild(j)->retchild(0)->rettext() << ",eax" << endl;

}

else

{

setindex(current->retchild(j)->retchild(1));

printjok();

prisv(current->retchild(j));

fi.clear();

joknum=0;

realvar=0;

}

}

}

}

int Syntax::GenerCode()

{

ASTnode \*current=NULL;

int i=root->childscount();

printdata();

for(int j=2 ; j<i-1 ; j++)

{

if(root->childtext(j)=="NACHINKA")

{

current=root->retchild(j);

ProvCikl(current);

}

}

mod << "main ENDP" << endl;

mod << "END main" << endl;

return 1;

}

void Syntax::while\_cikle(ASTnode \*temp)

{

if(temp->retchild(0)->rettext()=="<")

{

mod << "L" << metka << ": CMP " << temp->retchild(0)->retchild(0)->rettext() << "," << temp->retchild(0)->retchild(1)->rettext()<< endl;

mod << "JGE " << "L" << metka+1 << endl;

}

else if(temp->retchild(0)->rettext()==">")

{

mod << "L" << metka << ": CMP " << temp->retchild(0)->retchild(0)->rettext() << "," << temp->retchild(0)->retchild(1)->rettext()<< endl;

mod << "JLE " << "L" << metka+1 << endl;

}

else if(temp->retchild(0)->rettext()=="<=")

{

mod << "L" << metka << ": CMP " << temp->retchild(0)->retchild(0)->rettext() << "," << temp->retchild(0)->retchild(1)->rettext()<< endl;

mod << "JG " << "L" << metka+1 << endl;

}

else if(temp->retchild(0)->rettext()==">=")

{

mod << "L" << metka << ": CMP " << temp->retchild(0)->retchild(0)->rettext() << "," << temp->retchild(0)->retchild(1)->rettext()<< endl;

mod << "JL " << "L" << metka+1 << endl;

}

else if(temp->retchild(0)->rettext()=="=")

{

mod << "L" << metka << ": CMP " << temp->retchild(0)->retchild(0)->rettext() << "," << temp->retchild(0)->retchild(1)->rettext()<< endl;

mod << "JNE " << "L" << metka+1 << endl;

}

else if(temp->retchild(0)->rettext()=="<>")

{

mod << "L" << metka << ": CMP " << temp->retchild(0)->retchild(0)->rettext() << "," << temp->retchild(0)->retchild(1)->rettext()<< endl;

mod << "JE " << "L" << metka+1 << endl;

}

ProvCikl(temp->retchild(1));

mod << "JMP L" << metka << endl;

metka++;

mod << "L" << metka << ":" << endl;

}

void Syntax::ifcode(ASTnode \*temp)

{

if(temp->retchild(0)->rettext()=="<")

{

mod << "CMP " << temp->retchild(0)->retchild(0)->rettext() << "," << temp->retchild(0)->retchild(1)->rettext()<< endl;

mod << "JGE " << "L" << metka << endl;

}

else if(temp->retchild(0)->rettext()==">")

{

mod << "CMP " << temp->retchild(0)->retchild(0)->rettext() << "," << temp->retchild(0)->retchild(1)->rettext()<< endl;

mod << "JLE " << "L" << metka << endl;

}

else if(temp->retchild(0)->rettext()=="<=")

{

mod << "CMP " << temp->retchild(0)->retchild(0)->rettext() << "," << temp->retchild(0)->retchild(1)->rettext()<< endl;

mod << "JG " << "L" << metka << endl;

}

else if(temp->retchild(0)->rettext()==">=")

{

mod << "CMP " << temp->retchild(0)->retchild(0)->rettext() << "," << temp->retchild(0)->retchild(1)->rettext()<< endl;

mod << "JL " << "L" << metka << endl;

}

else if(temp->retchild(0)->rettext()=="=")

{

mod << "CMP " << temp->retchild(0)->retchild(0)->rettext() << "," << temp->retchild(0)->retchild(1)->rettext()<< endl;

mod << "JNE " << "L" << metka << endl;

}

else if(temp->retchild(0)->rettext()=="<>")

{

mod << "CMP " << temp->retchild(0)->retchild(0)->rettext() << "," << temp->retchild(0)->retchild(1)->rettext()<< endl;

mod << "JE " << "L" << metka << endl;

}

if(temp->childscount()>2 && temp->retchild(2)->rettext()=="ELSE")

{

ProvCikl(temp->retchild(1));

mod << "JMP L" << metka+1 << endl;

mod << "L" << metka << ":" << endl;

ProvCikl(temp->retchild(3));

mod << "L" << ++metka << ":" << endl;

}

else

{

ProvCikl(temp->retchild(1));

mod << "L" << metka << ":" << endl;

metka++;

}

}

void Syntax::setindex(ASTnode \*temp)

{

static int n=1;

if(temp->childscount()==1)

{

jok.jk1=temp->retchild(0)->rettext();

jok.jk2="eax";

fi.push\_back(jok);

}

else

for(int i=0 ; i<temp->childscount() ; i++)

{

setindex(temp->retchild(i));

if(temp->retchild(i)->rettype()==IDENT && idstringtype(temp->retchild(i)->rettext())=="REAL")

{

realvar=1;

}

if(temp->retchild(i)->rettype()==IDENT)

{

jok.jk1=temp->retchild(i)->rettext();

for(int j=0 ; j<(int)fi.size(); j++)

{

if(fi[j].jk1==temp->retchild(i)->rettext())

{

n=0;

break;

}

}

if(n)

{

switch(joknum)

{

case 0:

jok.jk2="eax";

break;

case 1:

jok.jk2="ebx";

break;

case 2:

jok.jk2="ecx";

break;

case 3:

jok.jk2="edx";

break;

}

fi.push\_back(jok);

joknum++;

}

n=1;

}

else if(temp->retchild(i)->rettype()==CONSTANT)

{

jok.jk1=jok.jk2=temp->retchild(i)->rettext();

fi.push\_back(jok);

}

}

}

string Syntax::getindex(string s)

{

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

{

if(fi[i].jk1==s) return fi[i].jk2;

}

return "";

}

void Syntax::opproc(ASTnode \*temp)

{

if(temp->rettext()=="sqrt")

{

if(temp->retchild(0)->rettype()==IDENT && idstringtype(temp->retchild(0)->rettext())=="REAL")

{

realvar=1;

}

if(realvar==1)

{

mod << "MOV eax," << temp->retchild(0)->rettext() << endl;

mod << "FLD eax" << endl;

mod << "FSQRT" << endl;

mod << "FSTP eax" << getindex(temp->retchild(0)->rettext()) << endl;

mod << "MOV " << temp->retchild(0)->rettext() << ","<< "eax" << endl;

}

else

{

mod << "MOV eax," << temp->retchild(0)->rettext() << endl;

mod << "FILD eax" << endl;

mod << "FSQRT" << endl;

mod << "FISTP eax" << getindex(temp->retchild(0)->rettext()) << endl;

mod << "MOV " << temp->retchild(0)->rettext() << ","<< "eax" << endl;

}

realvar=0;

}

else if(temp->rettext()=="printreal")

{

mod << "MOV eax," << temp->retchild(0)->rettext() << endl;

mod << "PUSH eax" << endl;

mod << "MOV eax,offset prnt\_scan\_real" << endl;

mod << "PUSH eax" << endl;

mod << "CALL printf" << endl;

mod << "ADD esp,8" << endl;

}

else if(temp->rettext()=="printint")

{

mod << "MOV eax," << temp->retchild(0)->rettext() << endl;

mod << "PUSH eax" << endl;

mod << "MOV eax,offset prnt\_scan\_int" << endl;

mod << "PUSH eax" << endl;

mod << "CALL printf" << endl;

mod << "ADD esp,8" << endl;

}

else if(temp->rettext()=="scanint")

{

mod << "LEA eax," << temp->retchild(0)->rettext() << endl;

mod << "PUSH eax" << endl;

mod << "MOV eax,offset prnt\_scan\_int" << endl;

mod << "PUSH eax" << endl;

mod << "CALL scanf" << endl;

mod << "ADD esp,8" << endl;

}

else if(temp->rettext()=="scanreal")

{

mod << "LEA eax," << temp->retchild(0)->rettext() << endl;

mod << "PUSH eax" << endl;

mod << "MOV eax,offset prnt\_scan\_real" << endl;

mod << "PUSH eax" << endl;

mod << "CALL scanf" << endl;

mod << "ADD esp,8" << endl;

}

else if(temp->rettext()=="printstring")

{

mod << "MOV eax,offset " << "var\_" << GSTPS(temp->retchild(0)->rettext()) << endl;

mod << "PUSH eax" << endl;

mod << "CALL printf" << endl;

mod << "ADD esp,4" << endl;

}

}

int Syntax::GSTPS(string s)

{

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

{

if(TI[i].var==s)

{

return TI[i].kol;

}

}

return 0;

}

void Syntax::prisv(ASTnode \*temp)

{

if(temp->retchild(0)->rettype()!=IDENT && temp->retchild(0)->rettype()!=CONSTANT)

{

prisv(temp->retchild(0));

}

if(temp->retchild(1) && temp->retchild(1)->rettype()!=IDENT && temp->retchild(1)->rettype()!=CONSTANT)

{

prisv(temp->retchild(1));

}

if(realvar==1)

{

if(temp->rettext()=="-")

{

if(temp->childscount()==1)

{

temp->settext(temp->retchild(0)->rettext());

mod << "NEG " << getindex(temp->retchild(0)->rettext()) << endl;

}

else

{

temp->settext(temp->retchild(0)->rettext());

mod << "FSUB " << getindex(temp->retchild(0)->rettext()) << "," << getindex(temp->retchild(1)->rettext()) << endl;

}

}

else if(temp->rettext()=="+")

{

temp->settext(temp->retchild(0)->rettext());

mod << "FADD " << getindex(temp->retchild(0)->rettext()) << "," << getindex(temp->retchild(1)->rettext()) << endl;

}

else if(temp->rettext()=="\*")

{

temp->settext(temp->retchild(0)->rettext());

mod << "FMUL " << getindex(temp->retchild(0)->rettext()) << "," << getindex(temp->retchild(1)->rettext()) << endl;

}

else if(temp->rettext()=="/")

{

temp->settext(temp->retchild(0)->rettext());

mod << "FDIV " << getindex(temp->retchild(0)->rettext()) << "," << getindex(temp->retchild(1)->rettext()) << endl;

}

else if(temp->rettext()==":=")

{

mod << "MOV " << temp->retchild(0)->rettext() << "," << getindex(temp->retchild(1)->rettext()) << endl;

return;

}

}

else

{

if(temp->rettext()=="-")

{

if(temp->childscount()==1)

{

temp->settext(temp->retchild(0)->rettext());

mod << "NEG " << getindex(temp->retchild(0)->rettext()) << endl;

}

else

{

temp->settext(temp->retchild(0)->rettext());

mod << "FISUB " << getindex(temp->retchild(0)->rettext()) << "," << getindex(temp->retchild(1)->rettext()) << endl;

}

}

else if(temp->rettext()=="+")

{

temp->settext(temp->retchild(0)->rettext());

mod << "FIADD " << getindex(temp->retchild(0)->rettext()) << "," << getindex(temp->retchild(1)->rettext()) << endl;

}

else if(temp->rettext()=="\*")

{

temp->settext(temp->retchild(0)->rettext());

mod << "FIMUL " << getindex(temp->retchild(0)->rettext()) << "," << getindex(temp->retchild(1)->rettext()) << endl;

}

else if(temp->rettext()=="/")

{

temp->settext(temp->retchild(0)->rettext());

mod << "FIDIV " << getindex(temp->retchild(0)->rettext()) << "," << getindex(temp->retchild(1)->rettext()) << endl;

}

else if(temp->rettext()==":=")

{

mod << "MOV " << temp->retchild(0)->rettext() << "," << getindex(temp->retchild(1)->rettext()) << endl;

return;

}

}

}

void Syntax::scancode()

{

code >> t >> word >> page;

}

int Syntax::ischislo(string s)

{

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

{

if(s[i]=='.') continue;

if ((s[i] < '0') || (s[i] > '9')) return 0;

}

return 1;

}

int Syntax::Synanalys()

{

if(START()!=NULL)

{

return 1;

}

else return 0;

}

ASTnode\* Syntax::START()

{

ASTnode \*prog = new ASTnode(NONTERMINAL, "PROGRAMM", NULL, NULL);

root=prog;

ASTnode \*qwe = NULL;

while(!code.eof())

{

try

{

qwe=MODULE();

if(qwe==NULL)

{

//cout << word << endl<< "Syntax Error" << endl;

return NULL;

}

prog->AddChild(qwe);

scancode();

if(word!=";") throw 10;

qwe=O();

prog->AddChild(qwe);

if(word!="BEGIN")

{

throw 20;

root->ASTremove(root);

return NULL;

}

prog->AddChild(P());

if(word!="END")

{

throw 21;

root->ASTremove(root);

return NULL;

}

prog->AddChild(new ASTnode(TERMINAL, word, NAME(), NULL));

scancode();

if(word!=".")

{

throw 4;

root->ASTremove(root);

return NULL;

}

code >> t;

}

catch(int i)

{

switch(i)

{

case 20: cout << "Expected BEGIN"; break;

case 21: cout << "Expected END"; break;

case 4: cout << "Expected ."; break;

case 10: cout << "Expected ;"; break;

case 12: cout << "Undeclared identifier"; break;

case 2: cout << "Syntax error"; break;

case 3: cout << "Expected number"; break;

case 5: cout << "Expected OF"; break;

case 6: cout << "Expected type of identifier"; break;

case 7: cout << "Expected name"; break;

default: break;

}

root->ASTremove(root);

cout <<" String: " << page;

return NULL;

}

}

prog->printnode(prog);

return prog;

}

ASTnode\* Syntax::MODULE()

{

ASTnode \*name=NULL;

scancode();

if(word!="MODULE") return NULL;

else

{

name = NAME();

if(name==NULL) return NULL;

return new ASTnode(KEYWORD, "MODULE", name, NULL);

}

}

ASTnode\* Syntax::NAME()

{

scancode();

nam=word;

if(t!="j") throw 7;

return new ASTnode(IDENT, word, NULL, NULL);

}

ASTnode\* Syntax::O()

{

string temp;

ASTnode \*ob=NULL;

ob=new ASTnode(NONTERMINAL, "DECLARATION", NULL, NULL);

scancode();

while(word=="CONST" || word=="VAR" || word=="PROCEDURE")

{

if(word=="CONST") ob->AddChild(C());

if(word=="VAR") ob->AddChild(V());

if(word=="PROCEDURE") ob->AddChild(F());

scancode();

}

if(ob->childscount()==0)

{

delete ob;

return NULL;

}

return ob;

}

ASTnode\* Syntax::P()

{

ASTnode \*temp=NULL;

temp=new ASTnode(NONTERMINAL, "NACHINKA", NULL, NULL);

scancode();

while(t=="j" || word=="WHILE" || word=="IF")

{

if(t=="j") temp->AddChild(OO());

else if(word=="WHILE" || word=="IF") temp->AddChild(PO());

scancode();

}

if(temp->childscount()==0) return NULL;

return temp;

}

ASTnode\* Syntax::PO()

{

if(word=="IF") return IFELSE();

else if(word=="WHILE") return CIKL();

else return NULL;

}

ASTnode\* Syntax::IFELSE()

{

ASTnode \*temp=NULL;

temp=new ASTnode(OPIF, "IF", NULL, NULL);

scancode();

temp->AddChild(BOOL());

if(word!="THEN")

{

cout << "Error: Expected THEN";

root->ASTremove(root);

return NULL;

}

temp->AddChild(P());

if(word=="ELSE")

{

temp->AddChild(new ASTnode(OPELSE, word, NULL, NULL));

temp->AddChild(P());

}

else if(word=="ELSEIF")

{

while(word=="ELSEIF")

{

temp->AddChild(new ASTnode(OPELSIF, word, NULL, NULL));

scancode();

temp->AddChild(BOOL());

if(word!="THEN")

{

cout << "Error: Expected THEN";

root->ASTremove(root);

return NULL;

}

temp->AddChild(P());

}

if(word=="ELSE")

{

temp->AddChild(new ASTnode(OPELSE, word, NULL, NULL));

temp->AddChild(P());

}

}

if(word!="END")

{

cout << "Error: Expected END";

root->ASTremove(root);

return NULL;

}

return temp;

}

ASTnode\* Syntax::CIKL()

{

ASTnode \*temp=NULL;

temp=new ASTnode(OPCIKLE, "WHILE", NULL, NULL);

scancode();

temp->AddChild(BOOL());

if(word!="DO")

{

cout << "Error: Expected DO";

root->ASTremove(root);

return NULL;

}

temp->AddChild(P());

if(word!="END")

{

cout << "Error: Expected END";

root->ASTremove(root);

return NULL;

}

return temp;

}

ASTnode\* Syntax::OO()

{

ASTnode \*temp=NULL;

string ident;

ident=word;

scancode();

if(word=="(")

{

temp=new ASTnode(PROCEDURE, ident, NULL, NULL);

FACTPAR(temp);

if(word!=")")

{

temp->ASTremove(temp);

cout << "Error";

return NULL;

}

scancode();

if(word!=";")

{

temp->ASTremove(temp);

cout << "Error";

return NULL;

}

}

else if(word==":=") temp=new ASTnode(ASSIGN, word, new ASTnode(IDENT, ident, NULL, NULL), A());

if(word!=";") throw 10;

return temp;

}

ASTnode\* Syntax::A()

{

ASTnode \*temp=NULL;

scancode();

temp=ARIFM();

return temp;

}

void Syntax::printid()

{

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

cout << TI[i].var << " " << TI[i].type << " " << TI[i].kol << " " << TI[i].mas<< endl;

}

int Syntax::check(string w)

{

if(w=="OR") return 17;

if(w=="AND") return 18;

if(w=="+") return 11;

if(w=="-") return 12;

if(w=="\*") return 13;

if(w=="/" || w=="DIV") return 14;

if(w=="MOD") return 15;

if(w=="j") return 3;

if(w=="c") return 2;

return 0;

}

ASTnode\* Syntax::BOOL()

{

ASTnode \*temp=NULL;

string w;

temp=JOIN();

while(word=="OR")

{

w=word;

scancode();

temp=new ASTnode(OR, w, temp, JOIN());

}

return temp;

}

ASTnode\* Syntax::JOIN()

{

ASTnode \*temp=NULL;

string w;

temp=EQUA();

while(word=="AND")

{

w=word;

scancode();

temp=new ASTnode(AND, w, temp, EQUA());

}

return temp;

}

ASTnode\* Syntax::EQUA()

{

ASTnode \*temp=NULL;

string w;

temp=REL();

while(word=="=" || word=="<>")

{

w=word;

scancode();

temp=new ASTnode(AND, w, temp, REL());

}

return temp;

}

ASTnode\* Syntax::REL()

{

ASTnode \*temp=NULL;

string w;

temp=ARIFM();

if(word=="<" || word==">" || word==">=" || word=="<=")

{

w=word;

scancode();

return new ASTnode(SRAV, w, temp, ARIFM());

}

else return temp;

}

ASTnode\* Syntax::ARIFM()

{

ASTnode \*temp=NULL;

string w;

temp=term();

while(word=="+" || word=="-")

{

w=word;

scancode();

temp=new ASTnode(check(w), w, temp, term());

}

return temp;

}

ASTnode\* Syntax::term()

{

ASTnode \*temp=NULL;

string w;

temp=unary();

while(word=="\*" || word=="/" || word=="MOD" || word=="DIV")

{

w=word;

scancode();

temp=new ASTnode(check(w), w, temp, unary());

}

return temp;

}

ASTnode\* Syntax::unary()

{

if(word=="-")

{

scancode();

return new ASTnode(SUB, "-", unary(), NULL);

}

else if(word=="NOT")

{

scancode();

return new ASTnode(NOT, "NOT", unary(), NULL);

}

else return factor();

}

ASTnode\* Syntax::factor()

{

ASTnode \*temp=NULL;

string w;

if(t=="c")

{

temp = new ASTnode(CONSTANT, word, NULL, NULL);

scancode();

return temp;

}

else if(t=="j")

{

w=word;

scancode();

if(idtype(w)==-1) throw 12;

if(idtype(w)!=55) return new ASTnode(IDENT, w, NULL, NULL);

else return offset(w);

}

else if(word=="(")

{

scancode();

temp=BOOL();

if(word==")")

{

scancode();

return temp;

}

}

return NULL;

}

ASTnode\* Syntax::offset(string qwe)

{

ASTnode \*temp=NULL, \*t1=NULL;

//cout << word;

if(word!="[")

{

//cout << "yes";

throw 2;

return NULL;

}

scancode();

t1=new ASTnode(MUL, "\*", new ASTnode(CONSTANT, word, NULL, NULL), new ASTnode(CONSTANT, sidtype(qwe), NULL, NULL));

temp=new ASTnode(ADD, "+", new ASTnode(IDENT, qwe, NULL, NULL), t1);

scancode();

if(word!="]")

{

throw 2;

return NULL;

}

scancode();

return temp;

}

string Syntax::sidtype(string w)

{

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

{

if(TI[i].var==w)

{

if(TI[i].type=="INTEGER" || TI[i].type=="CARDINAL" || TI[i].type=="REAL")

{

return "4";

}

else if(TI[i].type=="CHAR") return "1";

}

}

return " ";

}

int Syntax::idtype(string w)

{

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

{

if(TI[i].var==w)

{

if(TI[i].mas==1) return 55;

else if(TI[i].type=="INTEGER" || TI[i].type=="CARDINAL" || TI[i].type=="REAL" || TI[i].type=="CONST")

{

return 4;

}

else if(TI[i].type=="CHAR") return 1;

}

}

return -1;

}

string Syntax::idstringtype(string w)

{

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

{

if(TI[i].var==w)

{

return TI[i].type;

}

}

return "-1";

}

void Syntax::FACTPAR(ASTnode \*temp)

{

int str=0;

do

{

scancode();

if(t=="l") str=1;

else if(t!="j" && t!="c")

{

temp->ASTremove(temp);

cout << "Error";

return;

}

if(str==1)

{

temp->AddChild(new ASTnode(STR, word, NULL, NULL));

str=0;

if(TIcheck(word))

{

rty.kol=varnum;

varnum++;

rty.mas=0;

rty.proc="glob";

rty.var=HelpStr(word);

rty.type="string";

TI.push\_back(rty);

}

}

else if(t=="j") temp->AddChild(new ASTnode(IDENT, word, NULL, NULL));

else if(t=="c") temp->AddChild(new ASTnode(CONSTANT, word, NULL, NULL));

scancode();

}

while(word==",");

}

int Syntax::TIcheck(string s)

{

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

{

if(TI[i].var==s) return 0;

}

return 1;

}

string Syntax::HelpStr(string s)

{

if(s=="endl")

{

return "\\n";

}

else return s;

}

ASTnode\* Syntax::F()

{

ASTnode \*temp=NULL, \*name=NULL, \*parameters=NULL, \*temp2=NULL, \*endname=NULL;

if(word!="PROCEDURE") return NULL;

temp=new ASTnode(KEYWORD, "PROCEDURE", NULL, NULL);

name=NAME();

rty.proc=currentproc;

currentproc=word;

rty.type="PROCEDURE";

rty.mas=0;

rty.kol=0;

rty.var=nam;

TI.push\_back(rty);

if(name==NULL)

{

delete name;

delete temp;

return NULL;

}

scancode();

if(word=="(")

{

parameters=FORMALPAR();

if(parameters==NULL)

{

delete name;

delete temp;

return NULL;

}

name->AddChild(parameters);

scancode();

}

if(word!=";")

{

delete name;

delete temp;

delete parameters;

return NULL;

}

temp2=O();

name->AddChild(temp2);

if(word!="BEGIN")

{

delete name;

delete temp;

delete parameters;

delete temp2;

return NULL;

}

name->AddChild(P());

if(word!="END")

{

delete name;

delete temp;

delete parameters;

delete temp2;

return NULL;

}

endname=NAME();

if(endname==NULL)

{

delete name;

delete endname;

delete temp;

delete parameters;

delete temp2;

return NULL;

}

delete endname;

/\*scancode();

if(word!=";")

{

delete name;

delete endname;

delete temp;

delete parameters;

delete temp2;

return NULL;

}\*/

temp->AddChild(name);

currentproc="glob";

return temp;

}

ASTnode\* Syntax::FORMALPAR()

{

ASTnode \*temp=NULL, \*temp2=NULL, \*par=NULL;

temp=new ASTnode(NONTERMINAL, "PARAMETERS", NULL, NULL);

fparam=1;

do

{

scancode();

if(word=="VAR")

{

scancode();

par=new ASTnode(KEYWORD, word, NULL, NULL);

temp->AddChild(par);

}

else

{

temp->AddChild(new ASTnode(KEYWORD, word, NULL, NULL));

}

scancode();

if(word!=":")

{

delete temp;

delete temp2;

return NULL;

}

temp2=TYPE();

if(temp2==NULL)

{

delete temp;

delete temp2;

return NULL;

}

par->AddChild(temp2);

scancode();

}

while(word!=")");

fparam=0;

return temp;

}

ASTnode\* Syntax::C()

{

ASTnode \*temp=NULL, \*name=NULL, \*value=NULL, \*temp2=NULL;

if(word!="CONST") return NULL;

temp=new ASTnode(KEYWORD, "CONST", NULL, NULL);

name=NAME();

rty.proc=currentproc;

rty.var=nam;

rty.type="CONST";

rty.mas=0;

rty.kol=1;

if(name==NULL)

{

delete name;

delete temp;

return NULL;

}

scancode();

if(word!="=")

{

delete name;

delete temp;

return NULL;

}

value=C1();

rty.zn=value->rettext();

temp2=new ASTnode(ASSIGN, "=", name, value);

temp->AddChild(temp2);

scancode();

if(word!=";")

{

delete name;

delete temp;

delete temp2;

delete value;

return NULL;

}

TI.push\_back(rty);

return temp;

}

ASTnode\* Syntax::C1()

{

ASTnode \*temp=NULL;

scancode();

if(word=="-")

{

scancode();

temp=new ASTnode(SUB, "-", CHISLO(), NULL);

return temp;

}

else return CHISLO();

}

ASTnode\* Syntax::CHISLO()

{

if(!ischislo(word)) return NULL;

return new ASTnode(CONSTANT, word, NULL, NULL);

}

ASTnode\* Syntax::V()

{

ASTnode \*temp=NULL, \*name=NULL;

string ide;

if(word!="VAR") return NULL;

temp=new ASTnode(KEYWORD, "VAR", NULL, NULL);

name=NAME();

if(name==NULL) return NULL;

temp->AddChild(name);

scancode();

/\*while(word==",")

{

name=NAME();

if(name==NULL)

{

delete temp;

return NULL;

}

temp->AddChild(name);

code >> t >> word;

}\*/

if(word!=":") return NULL;

temp->AddChild(TYPE());

scancode();

if(word!=";")

{

delete name;

delete temp;

return NULL;

}

return temp;

}

ASTnode\* Syntax::TYPE()

{

scancode();

if(word!="INTEGER" && word!="REAL" && word!="CHAR" && word!="BOOLEAN" && word!="ARRAY") throw 6;

if(word=="ARRAY")

{

if(fparam)

{

scancode();

scancode();

word="ARRAY OF " + word;

}

else

{

rty.var=nam;

rty.mas=1;

scancode();

if(word!="[") throw 2;

scancode();

if(word!="0") throw 3;

scancode();

if(word!=".") throw 4;

scancode();

if(word!=".") throw 4;

scancode();

if(!ischislo(word)) throw 3;

else rty.kol=atoi(word.c\_str())+1;

scancode();

if(word!="]") throw 2;

scancode();

if(word!="OF") throw 5;

scancode();

if(t!="k") throw 6;

else rty.type=word;

word="ARRAY OF " + word;

rty.proc=currentproc;

rty.zn="?";

}

}

else

{

rty.proc=currentproc;

rty.var=nam;

rty.type=word;

rty.mas=0;

rty.kol=1;

rty.zn="?";

}

if(fparam==0) TI.push\_back(rty);

return new ASTnode(KEYWORD, word, NULL, NULL);

}