//demo02

//①需要识别B分支下的变量定义为标识符（完成）

//②需要有四元式的语句:N,I,E,E1,T,T1,F

//③每个节点考虑有没有什么属性附加

//使用递归下降分析法

//头文件

#include <iostream>

#include <map>

#include <algorithm>

#include <string>

#include <vector>

using namespace std;

string in;//输入

//词法分析

string instr;//输入符号串

int index;//当前输入符号读入字符的位置

char character;//全局变量字符，存放最新读入的字符

string token;//字符数组，存放已读入的字符序列

map<string, int> Symbol;//未定义标识符表<标识符,map中所在下标>

map<string, int> DefSymbol;//已定义标识符表

map<string, int> Digit;//常数表

map<string, int>::iterator it;

const int len = 50;

string Reserve[len];//保留字表

string Operator[2 \* len];//运算符

string Boundary[3 \* len];//界符

struct Binary {

Binary(int c, int i, string v = "-") {

category = c;

index = i;

value = v;

}

Binary() {}

int category = 0;//类别

int index = 0;//对应标识符在标识符表位置或常数值在常数表的位置

string value = "-";//标识符或常数值

};

struct Binary all[100];//所有单词

int ip;//所有单词集下标

void init\_Reserve() {//构造保留字表的函数

Reserve[1] = "main";

Reserve[2] = "var";

Reserve[3] = "integer";

Reserve[4] = "real";

Reserve[5] = "";

Reserve[6] = "";

Reserve[7] = "while";

Reserve[8] = "do";

Reserve[9] = "if";

Reserve[10] = "then";

Reserve[11] = "else";

Reserve[12] = "or";

Reserve[13] = "and";

Reserve[14] = "not";

}

void init\_Operator() {//初始化运算符表

Operator[50] = ":=";

Operator[51] = "+";

Operator[52] = "-";

Operator[53] = "\*";

Operator[54] = "/";

Operator[55] = "<";

Operator[56] = "<=";

Operator[57] = ">";

Operator[58] = ">=";

Operator[59] = "==";

Operator[60] = "!=";

}

void init\_Boundary() {//界符表初始化

Boundary[100] = "(";

Boundary[101] = ")";

Boundary[102] = ",";

Boundary[103] = ":";

Boundary[104] = ";";

Boundary[105] = "{";

Boundary[106] = "}";

Boundary[107] = "#";

}

void getChar() {//读入一个字符

character = instr[index++];

}

void getnbc() {//读入非空白字符

while (character == ' ') {

getChar();

}

}

void concat() {//连接字符串

token = token + character;

}

bool letter() {//判断是否为字母

if ((character >= 'A' && character <= 'Z') || (character >= 'a' && character <= 'z'))

return true;

return false;

}

bool digit() {//判断是否为数字

if (character >= '0' && character <= '9')

return true;

return false;

}

void retract() {//回退字符的函数

character = ' ';

index--;

}

int reserve() {//匹配保留字符

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

if (Reserve[i] == token)return i;

return -1;

}

string symbol() {

it = Symbol.find(token);//查找表中是否已存在该标识符

if (it != Symbol.end()) {//存在

return it->first;//返回该标识符

}

else {//不存在

Symbol[token] = Symbol.size();

return token;

}

}

bool defSymbol(string defSym,bool push) { //defSym表示符；push控制是否放入已定义标识符表中

it = DefSymbol.find(defSym);//查找表中是否已存在该标识符

if (it != DefSymbol.end()) {//存在

return true;//返回true

}

else {//不存在

if (push) {//放入已定义标识符表中

DefSymbol[defSym] = DefSymbol.size();

}

return false;//返回false

}

}

string constant() {

it = Digit.find(token);

if (it != Digit.end()) {

return it->first;

}

else {

Digit[token] = Digit.size();

return token;

}

}

Binary error() {

cout << token << "\t-->\t该单词不存在" << endl;

return Binary(0, 0);

}

//词法分析函数，逐个识别单词

Binary LexAnalyze() {

token = "";

getChar();

getnbc();

string val;

int num = -1;

switch (character) {

case'a':

case'b':

case'c':

case'd':

case'e':

case'f':

case'g':

case'h':

case'i':

case'j':

case'k':

case'l':

case'm':

case'n':

case'o':

case'p':

case'q':

case'r':

case's':

case't':

case'u':

case'v':

case'w':

case'x':

case'y':

case'z':

case'A':

case'B':

case'C':

case'D':

case'E':

case'F':

case'G':

case'H':

case'I':

case'J':

case'K':

case'L':

case'M':

case'N':

case'O':

case'P':

case'Q':

case'R':

case'S':

case'T':

case'U':

case'V':

case'W':

case'X':

case'Y':

case'Z':

while (letter() || digit()) {//为字母或数字

concat();//追加到token末尾

getChar();//读取下一个字符

}

retract();//回退一个字符

num = reserve();//查看保留字表

if (num != -1) {

return Binary(num, 0,Reserve[num]);

}

else {

val = symbol();//查看标识符表

return Binary(39, Symbol[val], val);//index从1开始

}

break;

case'0':

case'1':

case'2':

case'3':

case'4':

case'5':

case'6':

case'7':

case'8':

case'9':

while (digit()) {//为数字

concat();

getChar();

}

retract();

val = constant();//查看常数表

return Binary(40, Digit[val], val);

break;

case'<':

getChar();

if (character == '=') return Binary(56, 0, Operator[56]);//返回<=符号

else {

retract();

return Binary(55, 0, Operator[55]);//返回<符号

}

break;

case'>':

getChar();

if (character == '=')return Binary(58, 0, Operator[58]);//返回>=符号

else {

retract();

return Binary(57, 0, Operator[57]);//返回>符号

}

break;

case'=':

getChar();

if (character == '=') return Binary(59, 0, Operator[59]);//返回==符号

else {

retract();

return error();// 错误

}

break;

case'!':

getChar();

if (character == '=')return Binary(60, 0, Boundary[60]);

else return error();

break;

case'+':

return Binary(51, 0, Operator[51]);

break;

case'-':

return Binary(52, 0, Operator[52]);

break;

case'\*':

return Binary(53, 0, Operator[53]);

break;

case'/':

return Binary(54, 0, Operator[54]);

break;

case'(':

return Binary(100, 0, Boundary[100]);

break;

case')':

return Binary(101, 0, Boundary[101]);

break;

case',':

return Binary(102, 0, Boundary[102]);

break;

case':':

getChar();

if (character == '=')return Binary(50, 0, Operator[50]);

else {

retract();

return Binary(103, 0, Boundary[103]);

}

break;

case';':

return Binary(104, 0, Boundary[104]);

break;

case'{':

return Binary(105, 0, Boundary[105]);

break;

case'}':

return Binary(106, 0, Boundary[106]);

break;

case'#':

return Binary(107, 0, Boundary[107]);

break;

default:

return error();

}

}

void show\_table() {

cout << "\n==================" << "标识符" << "==================" << endl;

cout << "标识符\t\t类别编码\t表中位置" << endl;

for (it = DefSymbol.begin(); it != DefSymbol.end(); it++) {

if (it->first.size() >= 8)

cout << it->first << "\t39\t\t" << it->second << endl;

else

cout << it->first << "\t\t39\t\t" << it->second << endl;

}

cout << "\n==================" << "常数表" << "==================" << endl;

cout << "常量值\t\t类别编码\t表中位置" << endl;

for (it = Digit.begin(); it != Digit.end(); it++) {

cout << it->first << "\t\t40\t\t" << it->second << endl;

}

}

ostream& operator<<(ostream& output, const Binary& B)//对Binary的输出<<重载

{

output << "category: " << B.category << " index: " << B.index << " value: " << B.value;

return output;

}

//语法分析

//语义分析

//四元式

struct Quadruple {

Quadruple(string o, string s = "\_", string d = "\_", string r = "0") {

opcode = o;

soperand = s;

doperand = d;

result = r;

}

Quadruple(){}

string opcode;//操作码

string soperand;//源操作码

string doperand;//目的操作码

string result;//操作结果

};

map<string, Quadruple> Quadruples;//四元式集

int qid = 150;//四元式集四元式下标,代码编号

map<string, Quadruple>::iterator itQuad;

int mid = 0;//M~

int nid = 0;//N~

ostream& operator<<(ostream& output, const Quadruple& Q)//对Quadruple的输出<<重载

{

output << "opcode: " << Q.opcode << " soperand: " << Q.soperand << " doperand: " << Q.doperand << " result:" << Q.result;

return output;

}

//语法分析树

typedef struct tree {

string data;

tree\* next[10] = {NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL };

map<string, vector<string>> Attribute;

}\*ptree, tree;

int tid = 1;//T变量的下标

vector<string>::iterator itv;//向量迭代器

map<string, vector<string>>::iterator itm;//属性迭代器

int ipp;//单词表当前语法分析单词下标

void Analysis();

void P();

void A(int pipt);

void B(int pipt);

void D(int pipt);

void X(int pipt);

void X1(int pipt);

void Y(int pipt);

void G(int pipt);

void H(int pipt);

void I(int pipt);

void K(int pipt);

void K1(int pipt);

void L(int pipt);

void E(int pipt);

void E1(int pipt);

void T(int pipt);

void T1(int pipt);

void F(int pipt);

void R(int pipt);

void R1(int pipt);

void Z(int pipt);

void Z1(int pipt);

void Q(int pipt);

void W(int pipt);

void traverse(ptree r) {//前序遍历

cout << r->data << "\t\t属性：";

for (itm = r->Attribute.begin(); itm != r->Attribute.end(); itm++) {

cout << itm->first<<": ";

for (itv = itm->second.begin(); itv != itm->second.end(); itv++) {

cout << \*itv << " ";

}

}

cout << endl;

for (int j = 0; j < 10; j++) {

if (r->next[j] != NULL) {

traverse(r->next[j]);

}

}

}

ptree root = new tree;//树根节点

ptree p = root;//创建树用的中间变量

void Analysis() {

root->data = "P";

P();

}

//①所有文法函数表示，递归下降

void P() {

int pip = 0;//本节点下的直接子节点下标,用于添加子节点

if (all[ipp++].category == 1 && all[ipp++].category==100 && all[ipp++].category==101) {//main()

ptree t1 = new tree;

t1->data = "main()";

p->next[pip++] = t1;

cout << "main()匹配成功" << endl;

ptree t2 = new tree;

t2->data = "A";

p->next[pip++] = t2;

A(pip - 1);

}

else {

cout << "Perror2" << endl;

}

cout << "P遍历完成" << endl;

}

void A(int pipt) {

p = p->next[pipt];

ptree t = p;

int pip = 0;//本节点下的直接子节点下标,用于添加子节点

if (all[ipp].category == 105) {//{

ptree t1 = new tree;

t1->data = "{";

p->next[pip++] = t1;

cout << "{匹配成功" << endl;

ipp++;

ptree t2 = new tree;

t2->data = "B";

p->next[pip++] = t2;

B(pip - 1);

p = t;

ptree t3 = new tree;

t3->data = "G";

p->next[pip++] = t3;

G(pip - 1);

p = t;

if (all[ipp].category == 106) {//}

ptree t4 = new tree;

t4->data = "}";

p->next[pip++] = t4;

cout << "}匹配成功" << endl;

ipp++;

}

else {

cout << "Aerror1" << endl;

}

}

else {

cout << "Aerror2" << endl;

}

}

void B(int pipt) {

p = p->next[pipt];

ptree t = p;

int pip = 0;//本节点下的直接子节点下标,用于添加子节点

if (all[ipp].category == 2) {//var

ptree t1 = new tree;

t1->data = "var";

p->next[pip++] = t1;

cout << "var匹配成功" << endl;

ipp++;

ptree t2 = new tree;

t2->data = "D";

p->next[pip++] = t2;

D(pip - 1);

p = t;

}

else {

cout << "Berror1" << endl;

}

}

void D(int pipt) {

p = p->next[pipt];

ptree t = p;

int pip = 0;//本节点下的直接子节点下标,用于添加子节点

ptree t1 = new tree;

t1->data = "X";

p->next[pip++] = t1;

X(pip - 1);

p = t;

if (all[ipp].category == 103) {//:

ptree t2 = new tree;

t2->data = ":";

p->next[pip++] = t2;

cout << ":匹配成功" << endl;

ipp++;

ptree t3 = new tree;

t3->data = "Y";

p->next[pip++] = t3;

Y(pip - 1);

p = t;

if (all[ipp].category == 104) {//;

ptree t4 = new tree;

t4->data = ";";

p->next[pip++] = t4;

cout << ";匹配成功" << endl;

ipp++;

vector<string> v1;

v1.push\_back(p->next[2]->Attribute["type"].back());

p->Attribute["inherit"] = v1;//D.inherit=Y.type

vector<string> v2;

v2.push\_back(p->Attribute["inherit"].back());

p->next[0]->Attribute["type"] = v2;

if (all[ipp].category == 39 && all[ipp + 1].category == 102) {

ptree t5 = new tree;

t5->data = "D";

p->next[pip++] = t5;

D(pip - 1);

p = t;

}else if (all[ipp].category == 39 && all[ipp + 1].category == 103 && (all[ipp + 2].category == 3 || all[ipp + 2].category == 4)) {

ptree t5 = new tree;

t5->data = "D";

p->next[pip++] = t5;

D(pip - 1);

p = t;

}

}

else {

cout << "Derror1" << endl;

exit(-1);

}

}

else {

cout << "Derror2" << endl;

exit(-1);

}

}

void X(int pipt) {

p = p->next[pipt];

ptree t = p;

int pip = 0;//本节点下的直接子节点下标,用于添加子节点

if (all[ipp].category == 39) {

ptree t1 = new tree;

t1->data = all[ipp].value;

p->next[pip++] = t1;

cout <<all[ipp].value<< "匹配成功" << endl;

if (defSymbol(all[ipp].value, true)) {

cout << all[ipp].value << "重复定义了" << endl;

exit(-1);//直接退出程序

}

ipp++;

ptree t2 = new tree;

t2->data = "X1";

p->next[pip++] = t2;

X1(pip - 1);

p = t;

}

else {

cout << "Xerror1" << endl;

}

}

void X1(int pipt) {

p = p->next[pipt];

ptree t = p;

int pip = 0;//本节点下的直接子节点下标,用于添加子节点

if (all[ipp].category == 102) {//,

ptree t1 = new tree;

t1->data = ",";

p->next[pip++] = t1;

cout << ",匹配成功" << endl;

ipp++;

ptree t2 = new tree;

t2->data = "X";

p->next[pip++] = t2;

X(pip - 1);

p = t;

}

}

void Y(int pipt) {

p = p->next[pipt];

int pip = 0;//本节点下的直接子节点下标,用于添加子节点

if (all[ipp].category == 3) {//integer

ptree t1 = new tree;

t1->data = "integer";

p->next[pip++] = t1;

cout <<"integer匹配成功" << endl;

ipp++;

vector<string> v;

v.push\_back("integer");

p->Attribute["type"] = v;

return;

}

if (all[ipp].category == 4) {//real

ptree t1 = new tree;

t1->data = "real";

p->next[pip++] = t1;

cout <<"real匹配成功" << endl;

ipp++;

vector<string> v;

v.push\_back("real");

p->Attribute["type"] = v;

return;

}

else {

cout << "Yerror" << endl;

exit(-1);

}

}

void G(int pipt) {

p = p->next[pipt];

ptree t = p;

int pip = 0;//本节点下的直接子节点下标,用于添加子节点

ptree t1 = new tree;

t1->data = "H";

p->next[pip++] = t1;

H(pip - 1);

p = t;

if (all[ipp].category == 104) {//;

ptree t2 = new tree;

t2->data = ";";

p->next[pip++] = t2;

cout <<";匹配成功" << endl;

ipp++;

ptree t3 = new tree;

t3->data = "M" + to\_string(mid++);

p->next[pip++] = t3;

cout << t3->data << "添加成功" << endl;

vector<string> v1;

v1.push\_back(to\_string(qid));

p->next[2]->Attribute["quad"] = v1;//M.quad为上一代码编号的下一个值

ptree t4 = new tree;

t4->data = "G";

p->next[pip++] = t4;

G(pip - 1);

p = t;

vector<string> v2;

v2.push\_back(p->next[2]->Attribute["quad"].back());

p->next[0]->Attribute["next"] = v2;

p->Attribute["next"] = p->next[3]->Attribute["next"];//上G.next=下G.next

}

else {

p->Attribute["next"] = p->next[0]->Attribute["next"];//G.next=H.next

}

}

void H(int pipt) {

p = p->next[pipt];

ptree t = p;

int pip = 0;//本节点下的直接子节点下标,用于添加子节点

if (all[ipp].category == 39) {

ptree t1 = new tree;

t1->data = "I";

p->next[pip++] = t1;

I(pip - 1);

p = t;

p->Attribute["next"] = p->next[0]->Attribute["next"];

}

else if (all[ipp].category == 9) {//if

ptree t1 = new tree;

t1->data = "K";

p->next[pip++] = t1;

K(pip - 1);

p = t;

p->Attribute["next"] = p->next[0]->Attribute["next"];

}

else if (all[ipp].category == 7) {//while

ptree t1 = new tree;

t1->data = "L";

p->next[pip++] = t1;

L(pip - 1);

p = t;

p->Attribute["next"] = p->next[0]->Attribute["next"];

}

else {

cout << "Herror" << endl;

}

}

void I(int pipt) {

p = p->next[pipt];

ptree t = p;

int pip = 0;//本节点下的直接子节点下标,用于添加子节点

//if (all[ipp].category == 39){

if (defSymbol(all[ipp].value,false)) {

ptree t1 = new tree;

t1->data = all[ipp].value;

p->next[pip++] = t1;

cout << all[ipp].value << "匹配成功" << endl;

ipp++;

if (all[ipp].category == 50) {//:=

ptree t2 = new tree;

t2->data = ":=";

p->next[pip++] = t2;

cout << ":=匹配成功" << endl;

ipp++;

ptree t3 = new tree;

t3->data = "E";

p->next[pip++] = t3;

E(pip-1);

p = t;

}

}

else {

cout << "Ierror:没有定义此标识符" << endl;

return;

}

Quadruple quad( p->next[1]->data,

p->next[2]->Attribute.find("place")->second.back(),

"\_",

p->next[0]->data);

Quadruples[to\_string(qid++)] = quad;

vector<string> v;

p->Attribute["next"] = v;

}

void K(int pipt) {

p = p->next[pipt];

ptree t = p;

int pip = 0;//本节点下的直接子节点下标,用于添加子节点

if (all[ipp].category == 9) {//if

ptree t1 = new tree;

t1->data = "if";

p->next[pip++] = t1;

cout << "if匹配成功" << endl;

ipp++;

ptree t2 = new tree;

t2->data = "R";

p->next[pip++] = t2;

R(pip - 1);

p = t;

if (all[ipp].category == 10) {//then

ptree t3 = new tree;

t3->data = "then";

p->next[pip++] = t3;

cout << "then匹配成功" << endl;

ipp++;

ptree t4 = new tree;

t4->data = "M" + to\_string(mid++);

p->next[pip++] = t4;

cout << t4->data << "添加成功" << endl;

vector<string> v1;

v1.push\_back(to\_string(qid));

p->next[3]->Attribute["quad"] = v1;//M.quad为上一代码编号的下一个值

ptree t5 = new tree;

t5->data = "H";

p->next[pip++] = t5;

H(pip - 1);

p = t;

if (all[ipp].category == 11) {//else

ptree t6 = new tree;

t6->data = "K1";

p->next[pip++] = t6;

K1(pip - 1);

p = t;

//有else的部分

//回填M1.quad

itQuad = Quadruples.find(p->next[1]->Attribute["true"].back());//R的true四元组

itQuad->second.result = p->next[3]->Attribute["quad"].back();//M1.quad回填

//回填M2.quad

itQuad = Quadruples.find(p->next[1]->Attribute["false"].back());//R的false四元式

itQuad->second.result = p->next[5]->next[2]->Attribute["quad"].back();//M2.quad回填

//K.next

vector<string> v;

//H1的next

vector<string> vt1 = p->next[4]->Attribute["next"];

for (itv = vt1.begin(); itv != vt1.end(); itv++) {

v.push\_back(\*itv);

}

//N的next

v.push\_back(p->next[5]->next[0]->Attribute["next"].back());

//H2的next

vector<string> vt2 = p->next[5]->next[3]->Attribute["next"];

for (itv = vt2.begin(); itv != vt2.end(); itv++) {

v.push\_back(\*itv);

}

p->Attribute["next"] = v;

}

else {

//回填

itQuad = Quadruples.find(p->next[1]->Attribute["true"].back());//R为true的

itQuad->second.result = p->next[3]->Attribute["quad"].back();//M.quad回填

vector<string> v;

vector<string> vt1 = p->next[1]->Attribute["false"];//R为false的四元组

for (itv = vt1.begin(); itv != vt1.end(); itv++) {

v.push\_back(\*itv);

}

vector<string> vt2 = p->next[4]->Attribute["next"];//H的next

for (itv = vt2.begin(); itv != vt2.end(); itv++) {

v.push\_back(\*itv);

}

p->Attribute["next"] = v; //K.next属性

}

}

else {

cout << "Kerror1:没有then" << endl;

}

}

else {

cout << "Kerror2" << endl;

}

}

void K1(int pipt) {

p = p->next[pipt];

ptree t = p;

int pip = 0;//本节点下的直接子节点下标,用于添加子节点

if (all[ipp].category == 11) {//else

ptree t1 = new tree;

t1->data = "N" + to\_string(nid++);

p->next[pip++] = t1;

cout << t1->data << "添加成功" << endl;

vector<string> v1;

v1.push\_back(to\_string(qid));

p->next[0]->Attribute["next"] = v1;//N.next，记录无条件跳转，用来直接跳过H2

//无条件跳转语句

Quadruple quad("J");

Quadruples[to\_string(qid++)] = quad;

ptree t2 = new tree;

t2->data = "else";

p->next[pip++] = t2;

cout << "else匹配成功" << endl;

ipp++;

ptree t3 = new tree;

t3->data = "M" + to\_string(mid++);

p->next[pip++] = t3;

cout << t3->data << "添加成功" << endl;

vector<string> v2;

v2.push\_back(to\_string(qid));

p->next[2]->Attribute["quad"] = v2;//M.quad为上一代码编号的下一个值

ptree t4 = new tree;

t4->data = "H";

p->next[pip++] = t4;

H(pip - 1);

p = t;

}

else {

cout << "K1error" << endl;

}

}

void L(int pipt) {

p = p->next[pipt];

ptree t = p;

int pip = 0;//本节点下的直接子节点下标,用于添加子节点

if (all[ipp].category == 7) {//while

ptree t1 = new tree;

t1->data = "while";

p->next[pip++] = t1;

cout <<"while匹配成功" << endl;

ipp++;

ptree t2 = new tree;

t2->data = "M" + to\_string(mid++);

p->next[pip++] = t2;

cout << t2->data << "添加成功" << endl;

vector<string> v1;

v1.push\_back(to\_string(qid));

p->next[1]->Attribute["quad"] = v1;//M1.quad为上一代码编号的下一个值

ptree t3 = new tree;

t3->data = "R";

p->next[pip++] = t3;

R(pip - 1);

p = t;

if (all[ipp].category == 8) {//do

ptree t4 = new tree;

t4->data = "do";

p->next[pip++] = t4;

cout << "do匹配成功" << endl;

ipp++;

ptree t5 = new tree;

t5->data = "M" + to\_string(mid++);

p->next[pip++] = t5;

cout << t5->data << "添加成功" << endl;

vector<string> v2;

v2.push\_back(to\_string(qid));

p->next[4]->Attribute["quad"] = v2;//M2.quad为上一代码编号的下一个值

ptree t6 = new tree;

t6->data = "H";

p->next[pip++] = t6;

H(pip - 1);

p = t;

//R为false时

p->Attribute["next"] = p->next[2]->Attribute["false"];

//回填

itQuad = Quadruples.find(p->next[2]->Attribute["true"].back());//R为true的四元组代码编号

itQuad->second.result = p->next[4]->Attribute["quad"].back();//M2.quad值回填

//添加四元式,无条件跳转

Quadruple quad("J", "\_", "\_", p->next[1]->Attribute["quad"].back());//emit("J","\_","\_",M1.quad)

Quadruples[to\_string(qid++)] = quad;

}

else {

cout << "Lerror:没有do" << endl;

exit(-1);

}

}

}

void E(int pipt) {

p = p->next[pipt];

ptree t = p;

int pip = 0;//本节点下的直接子节点下标,用于添加子节点

ptree t1 = new tree;

t1->data = "T";

p->next[pip++] = t1;

T(pip - 1);

p = t;

ptree t2 = new tree;

t2->data = "E1";

p->next[pip++] = t2;

E1(pip - 1);

p = t;

//E1的子节点为空

if (p->next[1]->next[0] == NULL) {

p->Attribute["place"] = p->next[0]->Attribute["place"];

}

else {//E1的子节点不为空

vector<string> v;

v.push\_back("T" + to\_string(tid++));

p->Attribute["place"] = v;

//四元式

Quadruple quad(p->next[1]->next[0]->data,

p->next[0]->Attribute["place"].back(),

p->next[1]->Attribute["place"].back(),

p->Attribute["place"].back());

Quadruples[to\_string(qid++)] = quad;

}

}

void E1(int pipt) {

p = p->next[pipt];

ptree t = p;

int pip = 0;//本节点下的直接子节点下标,用于添加子节点

if (all[ipp].category == 51) {//+

ptree t1 = new tree;

t1->data = "+";

p->next[pip++] = t1;

cout <<"+匹配成功" << endl;

ipp++;

ptree t2 = new tree;

t2->data = "T";

p->next[pip++] = t2;

T(pip - 1);

p = t;

ptree t3 = new tree;

t3->data = "E1";

p->next[pip++] = t3;

E1(pip - 1);

p = t;

if (p->next[2]->next[0] == NULL) {//E1子节点为空

p->Attribute["place"] = p->next[1]->Attribute["place"];

}

else if(p->next[2]->next[2]->next[0]==NULL) {//此下此下E1子节点为空

vector<string> v;

v.push\_back("T" + to\_string(tid++));

p->Attribute["place"] = v;//设置place属性

//下边搞四元式

//（此下E1下+，此下T的place，此下E1下T的place，中间变量T~）//s:左T,d:右T

Quadruple quad(p->next[2]->next[0]->data,

p->next[1]->Attribute["place"].back(),

p->next[2]->next[1]->Attribute["place"].back(),

p->Attribute["place"].back());

Quadruples[to\_string(qid++)] = quad;

}

else {

vector<string> v;

v.push\_back("T" + to\_string(tid++));

p->Attribute["place"] = v;//设置place属性

//下边搞四元式

//（此下E1下+，此下T的place，此下E1的place，中间变量T~）//s:左T,d:右T

Quadruple quad(p->next[2]->next[0]->data,

p->next[1]->Attribute["place"].back(),

p->next[2]->Attribute["place"].back(),

p->Attribute["place"].back());

Quadruples[to\_string(qid++)] = quad;

}

}

else if (all[ipp].category == 52) {//-

ptree t1 = new tree;

t1->data = "-";

p->next[pip++] = t1;

cout <<"-匹配成功" << endl;

ipp++;

ptree t2 = new tree;

t2->data = "T";

p->next[pip++] = t2;

T(pip - 1);

p = t;

ptree t3 = new tree;

t3->data = "E1";

p->next[pip++] = t3;

E1(pip - 1);

p = t;

if (p->next[2]->next[0] == NULL) {//E1子节点为空

p->Attribute["place"] = p->next[1]->Attribute["place"];

}

else if(p->next[2]->next[2]->next[0]==NULL) {//此下此下E1子节点为空

vector<string> v;

v.push\_back("T" + to\_string(tid++));

p->Attribute["place"] = v ;//设置place属性

//下边搞四元式

//（此下E1下-，此下T的place，此下E1下T的place，中间变量T~）//s:左T,d:右T

Quadruple quad(p->next[2]->next[0]->data,

p->next[1]->Attribute["place"].back(),

p->next[2]->next[1]->Attribute["place"].back(),

p->Attribute["place"].back());

Quadruples[to\_string(qid++)] = quad;

}

else {

vector<string> v;

v.push\_back("T" + to\_string(tid++));

p->Attribute["place"] = v;//设置place属性

//下边搞四元式

//（此下E1下-，此下T的place，此下E1的place，中间变量T~）//s:左T,d:右T

Quadruple quad(p->next[2]->next[0]->data,

p->next[1]->Attribute["place"].back(),

p->next[2]->Attribute["place"].back(),

p->Attribute["place"].back());

Quadruples[to\_string(qid++)] = quad;

}

}

}

void T(int pipt) {

p = p->next[pipt];

ptree t = p;

int pip = 0;//本节点下的直接子节点下标,用于添加子节点

ptree t1 = new tree;

t1->data = "F";

p->next[pip++] = t1;

F(pip - 1);

p = t;

ptree t2 = new tree;

t2->data = "T1";

p->next[pip++] = t2;

T1(pip - 1);

p = t;

//考虑T1的子节点为空的情况

if (p->next[1]->next[0] == NULL) {

p->Attribute["place"] = p->next[0]->Attribute["place"];

}

else {

//palce属性为F左和F右的中间变量T\*

vector<string> v;

v.push\_back("T" + to\_string(tid++));

p->Attribute["place"] = v;

//下边搞四元式

Quadruple quad(p->next[1]->next[0]->data,

p->next[0]->Attribute["place"].back(),

p->next[1]->Attribute["place"].back(), //T1最根处

p->Attribute["place"].back());

Quadruples[to\_string(qid++)] = quad;//添加进四元式集

}

}

void T1(int pipt) {//考虑T1子节点为空，考虑链式乘，place属性化到最根处

p = p->next[pipt];

ptree t = p;

int pip = 0;//本节点下的直接子节点下标,用于添加子节点

if (all[ipp].category == 53) {// \*

ptree t1 = new tree;

t1->data = "\*";

p->next[pip++] = t1;

cout << "\*匹配成功" << endl;

ipp++;

ptree t2 = new tree;

t2->data = "F";

p->next[pip++] = t2;

F(pip - 1);

p = t;

ptree t3 = new tree;

t3->data = "T1";

p->next[pip++] = t3;

T1(pip - 1);

p = t;

if (p->next[2]->next[0] == NULL) {//T1子节点为空

p->Attribute["place"] = p->next[1]->Attribute["place"];

}

else if(p->next[2]->next[2]->next[0] == NULL) {//此下此下T1子节点为空

vector<string> v;

v.push\_back("T" + to\_string(tid++));

p->Attribute["place"] = v;//设置place属性

//下边搞四元式

//（此下T1下\*，此下F的place，此下T1下F的place，中间变量T~）//s:左F,d:右F

Quadruple quad(p->next[2]->next[0]->data,

p->next[1]->Attribute["place"].back(),

p->next[2]->next[1]->Attribute["place"].back(),

p->Attribute["place"].back());

Quadruples[to\_string(qid++)] = quad;

}

else {

vector<string> v;

v.push\_back("T" + to\_string(tid++));

p->Attribute["place"] = v;//设置place属性

//下边搞四元式

//（此下T1下\*，此下F的place，此下T1的place，中间变量T~）//s:左F,d:右F

Quadruple quad(p->next[2]->next[0]->data,

p->next[1]->Attribute["place"].back(),

p->next[2]->Attribute["place"].back(),

p->Attribute["place"].back());

Quadruples[to\_string(qid++)] = quad;

}

}

else if (all[ipp].category == 54) {// /

ptree t1 = new tree;

t1->data = "/";

p->next[pip++] = t1;

cout << "/匹配成功" << endl;

ipp++;

ptree t2 = new tree;

t2->data = "F";

p->next[pip++] = t2;

F(pip - 1);

p = t;

ptree t3 = new tree;

t3->data = "T1";

p->next[pip++] = t3;

T1(pip - 1);

p = t;

if (p->next[2]->next[0] == NULL) {//T1子节点为空

p->Attribute["place"] = p->next[1]->Attribute["place"];

}

else if(p->next[2]->next[2]->next[0] == NULL){//此下此下T1节点为空

vector<string> v;

v.push\_back("T" + to\_string(tid++));

p->Attribute["place"] = v;//设置place属性

//下边搞四元式

//（此下T1下/，此下F的place，此下T1下F的place，中间变量T~）//s:左F,d:右F

Quadruple quad(p->next[2]->next[0]->data,

p->next[1]->Attribute["place"].back(),

p->next[2]->next[1]->Attribute["place"].back(),

p->Attribute["place"].back());

Quadruples[to\_string(qid++)] = quad;

}

else {

vector<string> v;

v.push\_back("T" + to\_string(tid++));

p->Attribute["place"] = v;//设置place属性

//下边搞四元式

//（此下T1下/，此下F的place，此下T1的place，中间变量T~）//s:左F,d:右F

Quadruple quad(p->next[2]->next[0]->data,

p->next[1]->Attribute["place"].back(),

p->next[2]->Attribute["place"].back(),

p->Attribute["place"].back());

Quadruples[to\_string(qid++)] = quad;

}

}

}

void F(int pipt) {

p = p->next[pipt];

ptree t = p;

int pip = 0;//本节点下的直接子节点下标,用于添加子节点

if (defSymbol(all[ipp].value, false) || all[ipp].category == 40) {//匹配标识符或常数

ptree t1 = new tree;//创建节点

t1->data = all[ipp].value;

p->next[pip++] = t1;

cout << all[ipp].value << "匹配成功" << endl;

vector<string> v;

v.push\_back(all[ipp].value);

p->Attribute["place"] = v;//设置place属性

ipp++;

}else if (all[ipp].category == 100) {//(

ptree t1 = new tree;

t1->data = "(";

p->next[pip++] = t1;

cout << "(匹配成功" << endl;

ipp++;

ptree t2 = new tree;

t2->data = "E";

p->next[pip++] = t2;

E(pip - 1);

p = t;

p->Attribute["place"] = p->next[pip - 1]->Attribute["place"];

if (all[ipp].category == 101) {//)

ptree t3 = new tree;

t3->data = ")";

p->next[pip++] = t3;

cout << ")匹配成功" << endl;

ipp++;

}

else {

cout << "Ferror1:没有 ) " << endl;

return;

}

}

else {

cout << "Ferror2：没有定义此标识符" << endl;

}

}

void R(int pipt) {

p = p->next[pipt];

ptree t = p;

int pip = 0;//本节点下的直接子节点下标,用于添加子节点

ptree t1 = new tree;

t1->data = "Z";

p->next[pip++] = t1;

Z(pip - 1);

p = t;

ptree t2 = new tree;

t2->data = "R1";

p->next[pip++] = t2;

R1(pip - 1);

p = t;

if (p->next[1]->next[0] == NULL) {//此节点的R1子节点next为空

p->Attribute["true"] = p->next[0]->Attribute["true"];

p->Attribute["false"] = p->next[0]->Attribute["false"];

}

else if(p->next[1]->next[3]->next[0] == NULL){//此节点此下R1子节点next为空

//R.false的情况

p->Attribute["false"] = p->next[1]->next[2]->Attribute["false"];//R.false=Z2.false

//R.true的情况

vector<string> v2;

vector<string> vt1 = p->next[0]->Attribute["true"];//Z1的true跳转代码编号向量

for (itv = vt1.begin(); itv != vt1.end(); itv++) {

v2.push\_back(\*itv);

}

vector<string> vt2 = p->next[1]->next[2]->Attribute["true"];//Z2的true跳转代码编号向量

for (itv = vt2.begin(); itv != vt2.end(); itv++) {

v2.push\_back(\*itv);

}

p->Attribute["true"] = v2;//链接

//回填

itQuad = Quadruples.find(p->next[0]->Attribute["false"].back());//Z1的false的四元式

itQuad->second.result = p->next[1]->next[1]->Attribute["quad"].back();//M的quad属性值回填到Q1的true的result

}

else {

//R.false的情况

p->Attribute["false"] = p->next[1]->Attribute["false"];//R.false=R1.false

//R.true的情况

vector<string> v2;

vector<string> vt1 = p->next[0]->Attribute["true"]; //Z1的true跳转代码编号向量

for (itv = vt1.begin(); itv != vt1.end(); itv++) {

v2.push\_back(\*itv);

}

vector<string> vt2 = p->next[1]->Attribute["true"]; //R1的true跳转代码编号向量

for (itv = vt2.begin(); itv != vt2.end(); itv++) {

v2.push\_back(\*itv);

}

p->Attribute["true"] = v2;

//回填

itQuad = Quadruples.find(p->next[0]->Attribute["false"].back());//Z1的false的四元式

itQuad->second.result = p->next[1]->next[1]->Attribute["quad"].back();//M的quad属性值回填到Q1的true的result

}

}

void R1(int pipt) {

p = p->next[pipt];

ptree t = p;

int pip = 0;//本节点下的直接子节点下标,用于添加子节点

if (all[ipp].category == 12) {//or

ptree t1 = new tree;

t1->data = "or";

p->next[pip++] = t1;

cout << "or匹配成功" << endl;

ipp++;

ptree t2 = new tree;

t2->data = "M" + to\_string(mid++);

p->next[pip++] = t2;

cout << t2->data << "添加成功" << endl;

vector<string> v1;

v1.push\_back(to\_string(qid));

p->next[1]->Attribute["quad"] = v1;//M.quad为上一代码编号的下一个值

ptree t3 = new tree;

t3->data = "Z";

p->next[pip++] = t3;

Z(pip - 1);

p = t;

ptree t4 = new tree;

t4->data = "R1";

p->next[pip++] = t4;

R1(pip - 1);

p = t;

if (p->next[3]->next[0] == NULL) {}//此节点下R1子节点为空

else if (p->next[3]->next[3]->next[0] == NULL) {//此下此下R1子节点为空

//R1.false的情况

p->Attribute["false"] = p->next[3]->next[2]->Attribute["false"];//R1.false=Z2.false

//R1.true的情况

vector<string> v2;

vector<string> vt1 = p->next[2]->Attribute["true"];//Z1的true跳转代码编号向量

for (itv = vt1.begin(); itv != vt1.end(); itv++) {

v2.push\_back(\*itv);

}

vector<string> vt2 = p->next[3]->next[2]->Attribute["true"];//Z2的true跳转代码编号向量

for (itv = vt2.begin(); itv != vt2.end(); itv++) {

v2.push\_back(\*itv);

}

p->Attribute["true"] = v2;

//回填

itQuad = Quadruples.find(p->next[2]->Attribute["false"].back());//Z1的false的四元式

itQuad->second.result = p->next[3]->next[1]->Attribute["quad"].back();//M的quad属性值回填到Q1的true的result

}

else {//此下此下R1子节点不为空

//R1.false的情况

p->Attribute["false"] = p->next[3]->Attribute["false"];//上R1.false=下R1.false

//R1.true的情况

vector<string> v2;

vector<string> vt1 = p->next[2]->Attribute["true"]; //Z1的true跳转代码编号向量

for (itv = vt1.begin(); itv != vt1.end(); itv++) {

v2.push\_back(\*itv);

}

vector<string> vt2 = p->next[3]->Attribute["true"]; //下R1的true跳转代码编号向量

for (itv = vt2.begin(); itv != vt2.end(); itv++) {

v2.push\_back(\*itv);

}

p->Attribute["true"] = v2;

//回填

itQuad = Quadruples.find(p->next[2]->Attribute["false"].back());//Z1的false的四元式

itQuad->second.result = p->next[3]->next[1]->Attribute["quad"].back();//M的quad属性值回填到Q1的true的result

}

}

}

void Z(int pipt) {//对true和false属性

p = p->next[pipt];

ptree t = p;

int pip = 0;//本节点下的直接子节点下标,用于添加子节点

ptree t1 = new tree;

t1->data = "Q";

p->next[pip++] = t1;

Q(pip - 1);

p = t;

ptree t2 = new tree;

t2->data = "Z1";

p->next[pip++] = t2;

Z1(pip - 1);

p = t;

if (p->next[1]->next[0] == NULL) {//此Z节点下Z1子节点next为空

p->Attribute["true"] = p->next[0]->Attribute["true"];

p->Attribute["false"] = p->next[0]->Attribute["false"];

}

else if(p->next[1]->next[3]->next[0]==NULL){//此Z节点下此下Z1子节点next为空

//Z.true的情况

p->Attribute["true"] = p->next[1]->next[2]->Attribute["true"];//Z.true=Q2.true

//Z.false的情况

vector<string> v1;

v1.push\_back(p->next[0]->Attribute["false"].back());//Q1的false跳转代码编号

v1.push\_back(p->next[1]->next[2]->Attribute["false"].back());//Q2的false跳转代码编号

p->Attribute["false"] = v1;

//回填

itQuad = Quadruples.find(p->next[0]->Attribute["true"].back());//Q1的true的四元式

itQuad->second.result = p->next[1]->next[1]->Attribute["quad"].back();//M的quad属性值回填到Q1的true的result

}

else {

//Z.true的情况

p->Attribute["true"] = p->next[1]->Attribute["true"];//Z.true=Z1.true

//Z.false的情况

vector<string> v2;

v2.push\_back(p->next[0]->Attribute["false"].back());//Q1的false跳转代码编号

vector<string> vt = p->next[1]->Attribute["false"]; //Z1的false跳转代码编号向量

for (itv = vt.begin(); itv != vt.end(); itv++) {

v2.push\_back(\*itv);

}

p->Attribute["false"] = v2;

//回填

itQuad = Quadruples.find(p->next[0]->Attribute["true"].back());//Q1的true的四元式

itQuad->second.result = p->next[1]->next[1]->Attribute["quad"].back();//M的quad属性值回填到Q1的true的result

}

}

void Z1(int pipt) {

p = p->next[pipt];

ptree t = p;

int pip = 0;//本节点下的直接子节点下标,用于添加子节点

if (all[ipp].category == 13) {//and

ptree t1 = new tree;

t1->data = "and";

p->next[pip++] = t1;

cout << "and匹配成功" << endl;

ipp++;

ptree t2 = new tree;

t2->data = "M" + to\_string(mid++);

p->next[pip++] = t2;

cout << t2->data << "添加成功" << endl;

vector<string> v1;

v1.push\_back(to\_string(qid));

p->next[1]->Attribute["quad"] = v1;//M.quad为上一代码编号的下一个值

ptree t3 = new tree;

t3->data = "Q";

p->next[pip++] = t3;

Q(pip - 1);

p = t;

ptree t4 = new tree;

t4->data = "Z1";

p->next[pip++] = t4;

Z1(pip - 1);

p = t;

if (p->next[3]->next[0] == NULL) {}//此节点下Z1子节点为空}

else if(p->next[3]->next[3]->next[0]==NULL){//此下此下Z1子节点为空

//Z1.true的情况

p->Attribute["true"] = p->next[3]->next[2]->Attribute["true"];//Z1.true=Q2.true

//Z1.false的情况

//建链,更改四元式的值（放弃）

//itQuad = Quadruples.find(p->next[3]->next[2]->Attribute["false"].back());//Q2的false的四元式

//itQuad->second.result = p->next[2]->Attribute["false"];//Q1的false的四元式编号与Q2的false链接

vector<string> v2;

v2.push\_back(p->next[2]->Attribute["false"].back());//Q1的false跳转代码编号

v2.push\_back(p->next[3]->next[2]->Attribute["false"].back());//Q2的false跳转代码编号

p->Attribute["false"] = v2;

//回填和建链会让四元式的result值冲突，这里选择只回填到四元式

itQuad = Quadruples.find(p->next[2]->Attribute["true"].back());//Q1的true的四元式

itQuad->second.result = p->next[3]->next[1]->Attribute["quad"].back();//M的quad属性值回填到Q1的true的result

}

else {//此下此下Z1子节点不为空

//Z1.true的情况

p->Attribute["true"] = p->next[3]->Attribute["true"];//上Z1.true=下Z1.true

//Z1.false的情况

vector<string> v2;

v2.push\_back(p->next[2]->Attribute["false"].back());//Q1的false跳转代码编号

vector<string> vt = p->next[3]->Attribute["false"]; //下Z1的false跳转代码编号向量

for (itv = vt.begin(); itv != vt.end(); itv++) {

v2.push\_back(\*itv);

}

p->Attribute["false"] = v2;

//回填

itQuad = Quadruples.find(p->next[2]->Attribute["true"].back());//Q1的true的四元式

itQuad->second.result = p->next[3]->next[1]->Attribute["quad"].back();//M的quad属性值回填到Q1的true的result

}

}

}

void Q(int pipt) {

p = p->next[pipt];

ptree t = p;

int pip = 0;//本节点下的直接子节点下标,用于添加子节点

if (defSymbol(all[ipp].value, false) || all[ipp].category==40) {

ptree t1 = new tree;

t1->data = all[ipp].value;

p->next[pip++] = t1;

cout << all[ipp].value << "匹配成功" << endl;

ipp++;

ptree t2 = new tree;

t2->data = "W";

p->next[pip++] = t2;

W(pip - 1);

p = t;

if (defSymbol(all[ipp].value, false) || all[ipp].category == 40) {

ptree t3 = new tree;

t3->data = all[ipp].value;

p->next[pip++] = t3;

cout << all[ipp].value << "匹配成功" << endl;

ipp++;

}

else {

cout << "Qerror1:没有定义此标识符" << endl;

return;

}

Quadruple quad1("J" + p->next[1]->next[0]->data, p->next[0]->data, p->next[2]->data);//关于比较的布尔因子的四元式

Quadruples[to\_string(qid++)] = quad1;

Quadruple quad2("J");

Quadruples[to\_string(qid++)] = quad2;

vector<string> v1;

v1.push\_back(to\_string(qid - 2));

p->Attribute["true"] = v1;

vector<string> v2;

v2.push\_back(to\_string(qid - 1));

p->Attribute["false"] = v2;

}else

if (all[ipp].category == 14) {//not

ptree t1 = new tree;

t1->data = "not";

p->next[pip++] = t1;

cout << "not匹配成功" << endl;

ipp++;

ptree t2 = new tree;

t2->data = "Q";

p->next[pip++] = t2;

Q(pip - 1);

p = t;

p->Attribute["true"] = p->next[1]->Attribute["false"];

p->Attribute["false"] = p->next[1]->Attribute["true"];

}

else {

cout << "Qerror2:没有定义此标识符" << endl;

return;

}

}

void W(int pipt) {

p = p->next[pipt];

int pip = 0;//本节点下的直接子节点下标,用于添加子节点

if (all[ipp].category == 55) {//<

ptree t1 = new tree;

t1->data = "<";

p->next[pip++] = t1;

cout << "<匹配成功" << endl;

ipp++;

return;

}else

if (all[ipp].category == 56) {//<=

ptree t1 = new tree;

t1->data = "<=";

p->next[pip++] = t1;

cout << "<=匹配成功" << endl;

ipp++;

return;

}else

if (all[ipp].category == 57) {//>

ptree t1 = new tree;

t1->data = ">";

p->next[pip++] = t1;

cout << ">匹配成功" << endl;

ipp++;

return;

}else

if (all[ipp].category == 58) {//>=

ptree t1 = new tree;

t1->data = ">=";

p->next[pip++] = t1;

cout << ">=匹配成功" << endl;

ipp++;

return;

}else

if (all[ipp].category == 59) {//==

ptree t1 = new tree;

t1->data = "==";

p->next[pip++] = t1;

cout << "==匹配成功" << endl;

ipp++;

return;

}else

if (all[ipp].category == 60) {//!=

ptree t1 = new tree;

t1->data = "!=";

p->next[pip++] = t1;

cout << "!=匹配成功" << endl;

ipp++;

return;

}

else {

cout << "Werror"<< endl;

}

}

int main() {

init\_Reserve();//保留字表初始化

init\_Boundary();//界符表初始化

init\_Operator();//运算符表初始化

Symbol.clear();//标识符集初始化

Digit.clear();//常数集初始化

index = 0;

character = ' ';

token = "";

cout << "请输入待词法分析的源程序代码：输入#代表结束输入\n" << endl;

//源程序代码输入处理

while (cin >> in && in[in.size() - 1] != '#') {

instr = instr + " " + in;

}

//识别二元组初始化

Binary word(0, 0, "-");

//循环进行词法分析直到识别所有单词符号

while (index < instr.size()) {

word = LexAnalyze();

all[ip++] = word;

}

cout << "\n==================匹配结果==================\n" << endl;

Analysis();

//展示构造的各种词汇表

cout << "\n==================词汇表展示==================\n" << endl;

show\_table();

cout << endl;

cout << endl;

cout << "\n==================语法树(前序遍历)==================\n" << endl;

traverse(root);

cout << endl;

cout << endl;

cout << "\n==================四元式集==================\n" << endl;

for (itQuad = Quadruples.begin(); itQuad != Quadruples.end(); itQuad++) {

cout << itQuad->first << " " << itQuad->second << endl;

}

cout << endl;

cout << endl;

//cout << "------------试验区--------------" << endl;\*/

/\*int n=10;

cout<<to\_string(n+1)<<endl;\*///int转string

//vector

//vector<string> str;

//str.push\_back("root");

//str.push\_back("doge");

///\*cout << str[0] << endl;

//cout << str[1] << endl;\*/

//vector<string>::iterator itv;

//for (itv = str.begin(); itv != str.end(); itv++) {

// cout << \*itv << endl;

//}

return 0;

}