**编译原理课程大作业报告二**

**语义分析器说明文档**



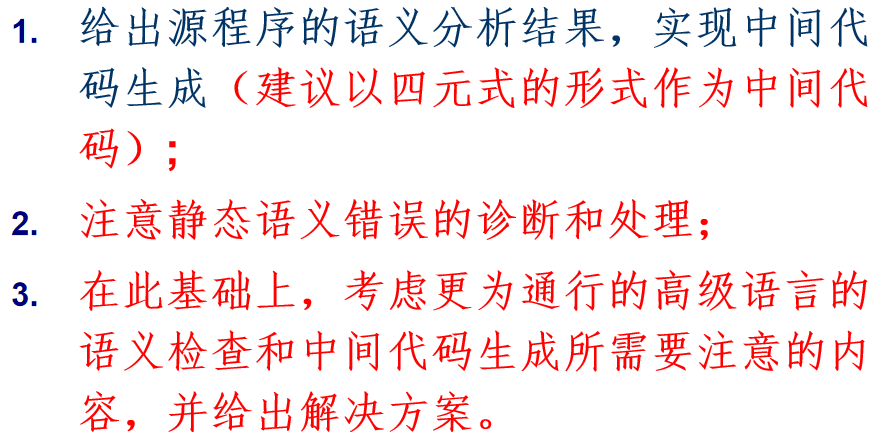
**成员： 涂远鹏-1652262**

**成员： 黎盛烜-1652130**

**指导老师： 丁志军**

**日期： 2018年12月10日**

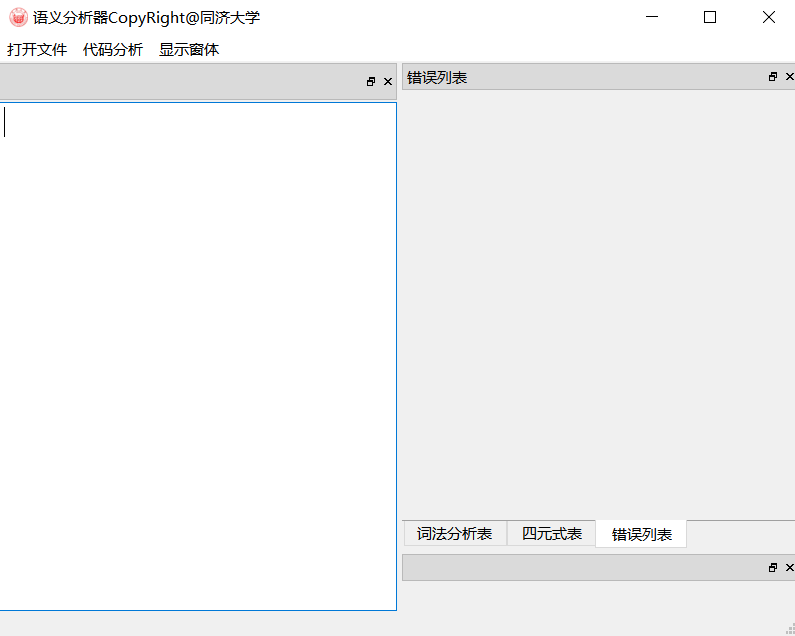
**1.语义分析器题目说明**



1. 输出语义分析结果——四元式
2. 输出语义分析结果，如果语义错误输出错误原因
3. 程序具有通用性，即所编制的语义分析程序能够适用于各类包含过程调用的类C程序。

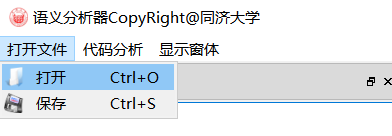
**2.程序使用说明**

**2.1主界面**

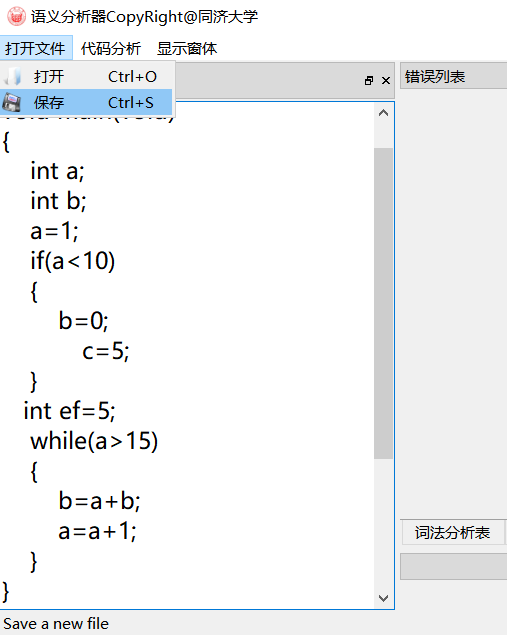


**2.2软件功能**

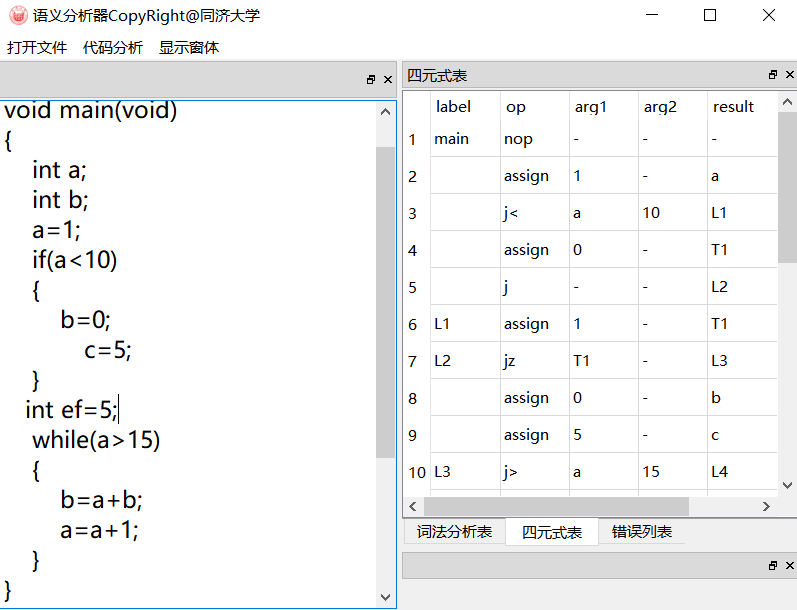
1. 首先打开选择文件menubar点击打开文件action，选择要分析的类C语言程序：

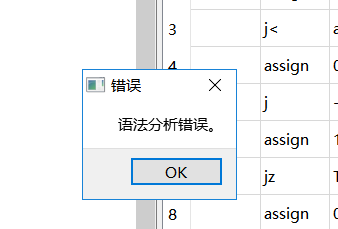


1. 随后在textedit控件中便会显示文件内容，并可以通过直接在textedit控件中编辑文件，随后点击保存对类C文件进行修改：

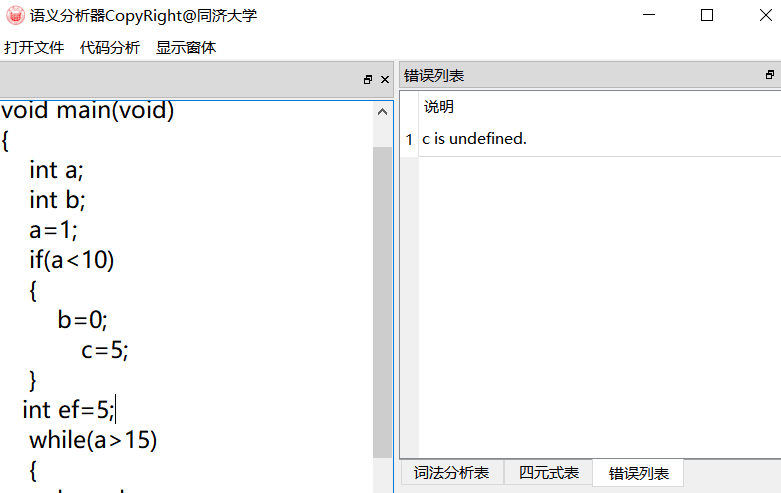


1. 随后点击menubar中的语义分析action，随后点击四元式表便可查看到类C程序的语义分析结果，如果语义分析发现语法存在错误，则会弹出语法分析错误窗口：

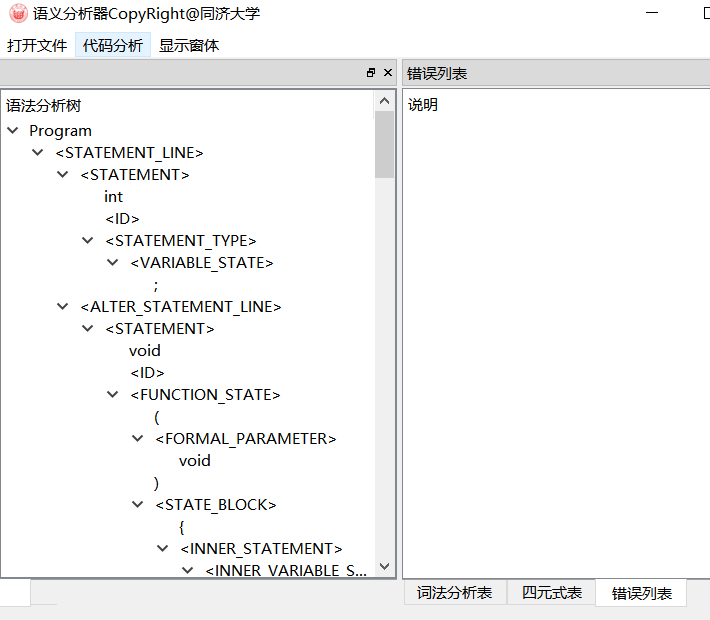




4.如果语法分析正确而语义分析错误，则会在错误列表显示语义分析错误原因：



5.语义分析由于条件控制语句翻译需要语法分析的语法分析树，所以输出语法分析树显示如下：



软件实现方法：

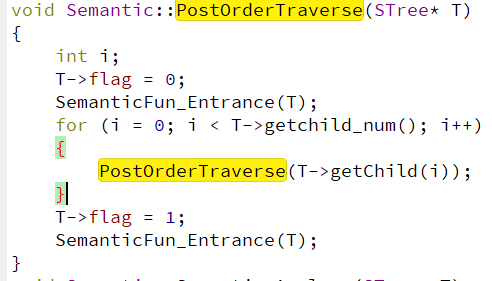
1. 类C程序的编辑与保存是使用textedit控件，通过getText()读取当前textedit中内容保存到指定文件中。
2. 使用QTableView控件显示语义分析结果——四元式表
3. 使用两次扫描的方法得到整个语义分析的结果

**3.语义分析实现算法**

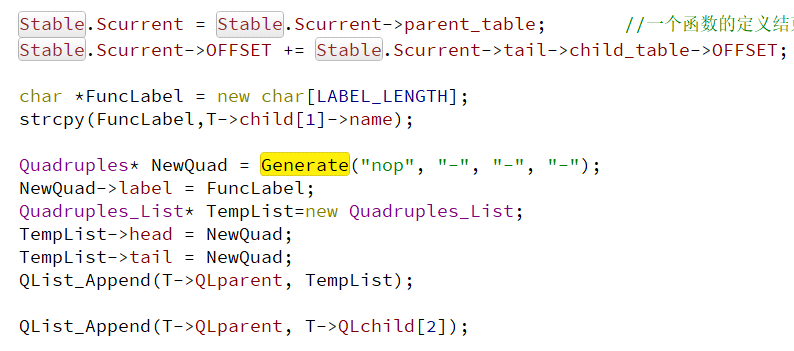
3.1四元式生成算法

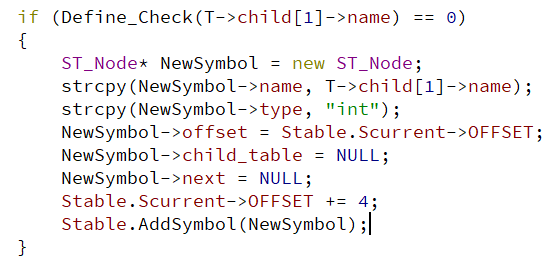
1. 首先进行语法分析生成一颗语法分析树

(2)从根结点开始取语法分析树的节点，对该节点取出当前节点的内容进行语义分析匹配对应的生成式进行移进规约操作，随后将当前节点的儿子节点也进行递归遍历进行语义分析操作，取完儿子节点之后再次对当前节点进行语义匹配：

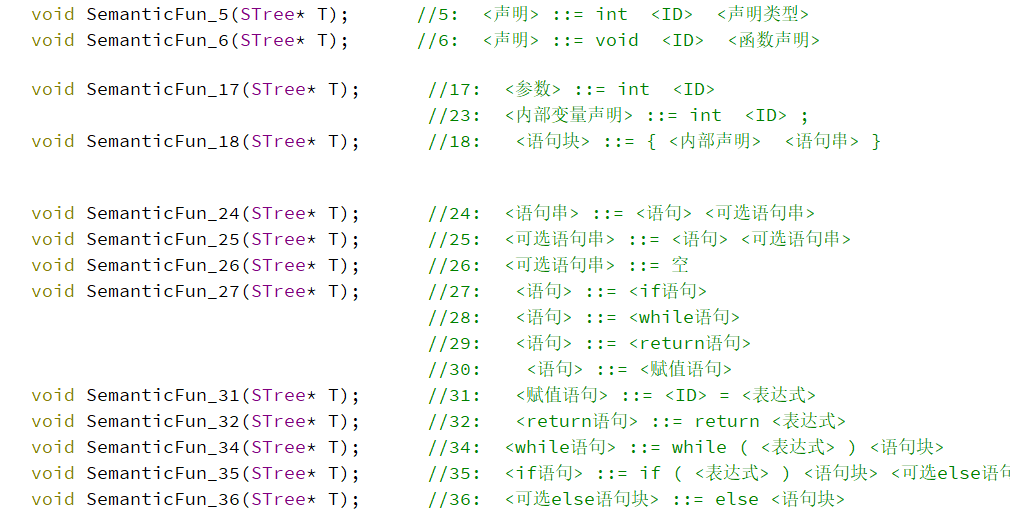


(3)对当前节点中的语句判断为说明语句/赋值语句/布尔表达式，对条件控制及其语句的的翻译采用两遍扫描的方式根据对应的翻译规则翻译为不同的四元式，在每一个语义匹配函数中，如果匹配成功，则新建一个Quadruples节点并用generate函数生成对应的四元式，对于匹配函数的节点label赋值为FuncLable进行标记，对于符号定义语句，新建ST\_Node并压入到符号栈中，修改符号栈的栈顶指针位置：

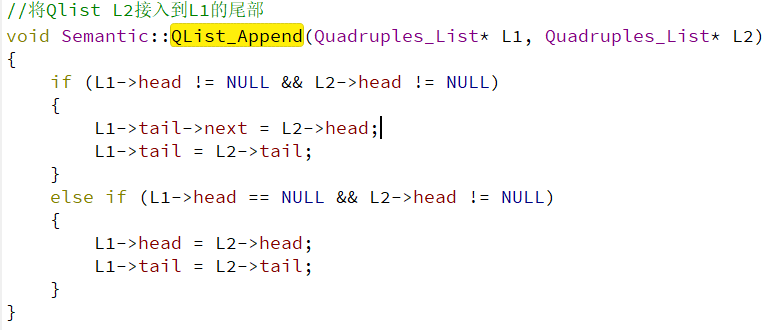




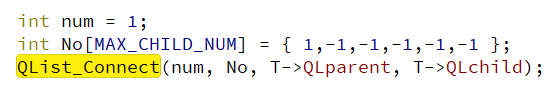
根据所有的语义分析翻译规则可写出的语义分析函数共有65个，此处列出函数声明如下：



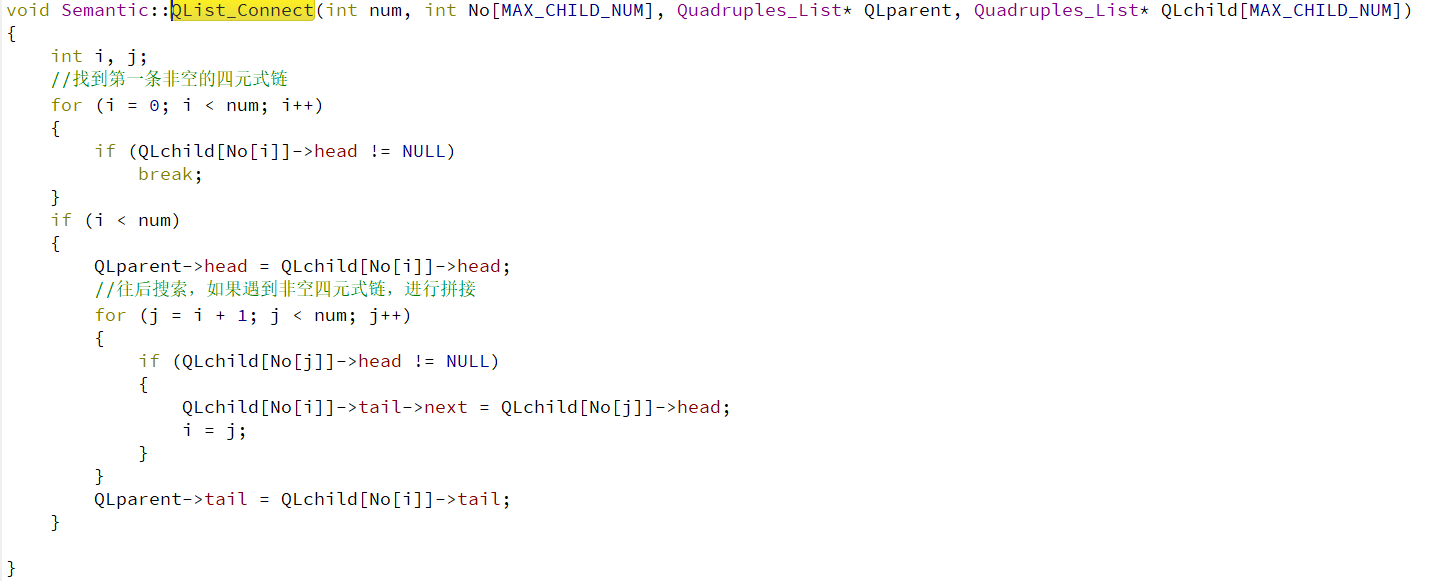
(4)在上述操作中，翻译语句为四元式之后，将该节点拼接入四元式链的尾部：



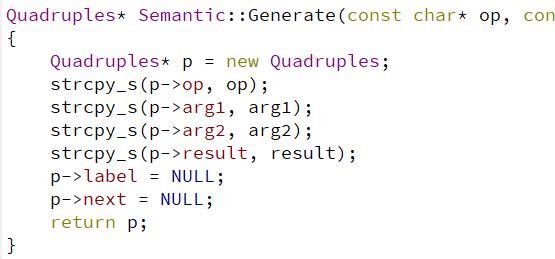
(5)如果子树的节点已完全遍历，则将子树生成的四元式链与父节点的四元式链尾部进行连接：



首先找到第一条非空的四元式链，在该四元式链中往后搜索，如果遇到非空的四元式链，进行拼接：

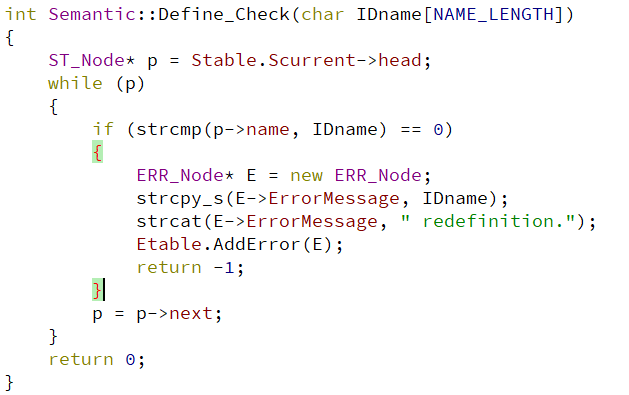


(6)构造四元式节点的函数如下：



3.2错误分析算法

变量定义错误检查：取出当前变量/函数名，取出符号表的头结点从头到尾在符号表中进行匹配，如果该变量已在之前定义过，那么新建一个ERR\_Node，将错误信息存入ERR\_Node节点并加入到错误列表Etable中

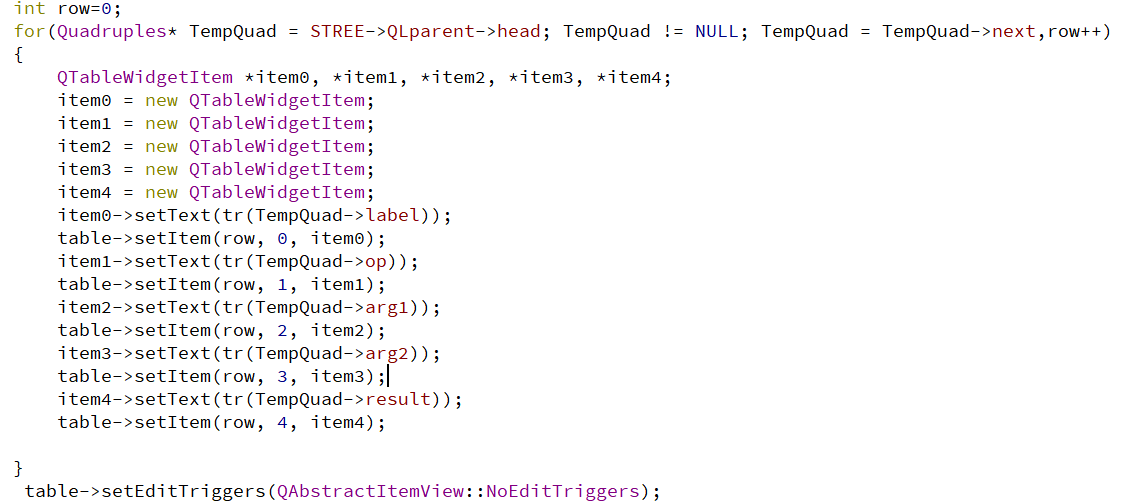


变量及函数使用错误检查：取出当前变量/函数名，取出符号表的头结点从头到尾在符号表中进行匹配，如果没有匹配到，说明该变量/函数在当前使用之前未被定义过，那么新建一个ERR\_Node，将错误信息存入ERR\_Node节点并加入到错误列表Etable中

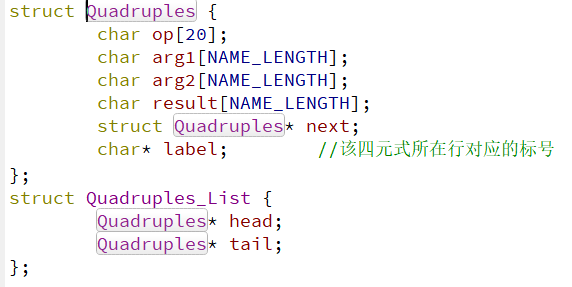


3.3显示四元式表的方式

显示四元式表所用的Qt控件为QtableWidget，

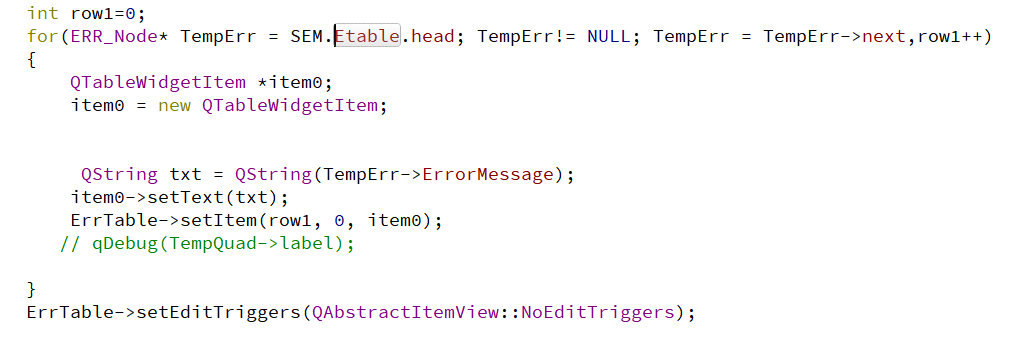


由于四元式表是利用链表方式存储的，首先获取STREE类中的链表头节点然后依次往下取节点，每次取出一个Quadruples节点，按照Quadruples中存储的五个对象内容分别赋给五个QTableWidgetItem对象并将该对象加入到table中，从而达到显示表结构目的，Quadruples与Quadruples List类内容显示如下：

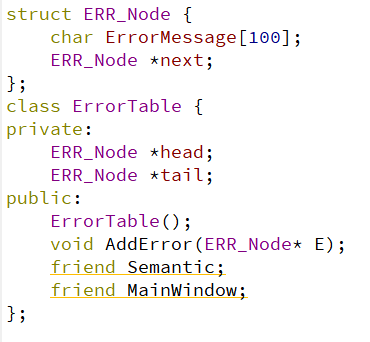


3.4显示错误列表方式

显示错误列表所用的Qt控件为QtableWidget，



使用自定义类Etable存储错误信息，自定义类中存储有链表的头尾节点，每个节点的类为ERR\_Node存储有错误信息，在输出到表格时，每一行均分别新建QTableWidgetItem成员，并给该成员赋值为SEM.Etable链表中节点的Errormessage内容并添加该成员到表结构中。



3.5语义栈

语法分析在语法分析的基础上，增加一个语义栈，栈内元素为语义结点。结点类是S属性文法的表示，判别每次语法分析所使用的产生式，实现不同的语义动作，每当规约到特定非终结符时，即可产生中间代码。

3.6符号表和函数表

每次规约识别出一个新的标识符，都会将其加入符号表中，符号的信息包括标识符、中间变量名、类型、占用空间、内存偏移量、作用的函数等。而当规约到函数定义的时候，则将函数名、形参列表、代号加入函数表。此处是通过添加newlabel标志设置为Func区分符号还是函数。

**4.逻辑结构与物理结构**

class Semantic;

struct LT\_Node {//标签节点

char label[LABEL\_LENGTH];

int line;

LT\_Node\* next;

};

class LabelTable{//标签表类

private:

LT\_Node\* head;

LT\_Node\* tail;

public:

LabelTable();

void AddLabel(char label[LABEL\_LENGTH]);

friend Semantic;

};

struct SymbolTable;

struct ST\_Node {//符号表节点

char name[NAME\_LENGTH];

char type[20];

int offset;

SymbolTable\* child\_table;

ST\_Node\* next;

};

struct SymbolTable {//符号表

ST\_Node\* head;

ST\_Node\* tail;

int OFFSET;

SymbolTable\* parent\_table;

};

class SymbolTableS {

private:

SymbolTable\* Sroot;

SymbolTable\* Scurrent;

public:

SymbolTableS();

void AddSymbol(ST\_Node \*S);

friend Semantic;

};

struct ERR\_Node {//错误列表节点

char ErrorMessage[100];

ERR\_Node \*next;

};

class ErrorTable {//错误列表存储类

private:

ERR\_Node \*head;

ERR\_Node \*tail;

public:

ErrorTable();

void AddError(ERR\_Node\* E);

friend Semantic;

friend MainWindow;

};

class Semantic {//语义分析类，包含所有语义分析函数以及存储有标签表、符号表以及错误列表类

private:

int line;

int LabelNo;

int TempNo;

LabelTable Ltable;

SymbolTableS Stable;

ErrorTable Etable;

stack <char\*> Stemp;

public:

Semantic();

friend MainWindow;

void Num\_to\_Str(int num, char str[10]);

void NewTemp(char Temp[NAME\_LENGTH]);

void NewLabel(char Label[LABEL\_LENGTH]);

Quadruples\* Generate(const char\* op, const char\* arg1, const char\* arg2, const char\* result); //生成一条四元式

void QList\_Connect(int num, int No[MAX\_CHILD\_NUM], Quadruples\_List\* QLparent, Quadruples\_List\* QLchild[MAX\_CHILD\_NUM]); //用于拼接四元式链

void QList\_Append(Quadruples\_List\* L1, Quadruples\_List\* L2); // 将Qlist L2接入到L1的尾部

void PostOrderTraverse(STree\* T);

void Semantic\_Analyze(STree\* T);

void SemanticFun\_Entrance(STree\* T);

void SemanticFun\_5(STree\* T); //5: <声明> ::= int <ID> <声明类型>

void SemanticFun\_6(STree\* T); //6: <声明> ::= void <ID> <函数声明>

void SemanticFun\_17(STree\* T); //17: <参数> ::= int <ID> //23: <内部变量声明> ::= int <ID> ;

void SemanticFun\_18(STree\* T); //18: <语句块> ::= { <内部声明> <语句串> }

void SemanticFun\_24(STree\* T); //24: <语句串> ::= <语句> <可选语句串>

void SemanticFun\_25(STree\* T); //25: <可选语句串> ::= <语句> <可选语句串>

void SemanticFun\_26(STree\* T); //26: <可选语句串> ::= 空

void SemanticFun\_27(STree\* T); //27: <语句> ::= <if语句>

//28: <语句> ::= <while语句>

//29: <语句> ::= <return语句>

//30: <语句> ::= <赋值语句>

void SemanticFun\_31(STree\* T); //31: <赋值语句> ::= <ID> = <表达式>

void SemanticFun\_32(STree\* T); //32: <return语句> ::= return <表达式>

void SemanticFun\_34(STree\* T); //34: <while语句> ::= while ( <表达式> ) <语句块>

void SemanticFun\_35(STree\* T); //35: <if语句> ::= if ( <表达式> ) <语句块> <可选else语句块>

void SemanticFun\_36(STree\* T); //36: <可选else语句块> ::= else <语句块>

void SemanticFun\_39(STree\* T); //39: <可选表达式> :: = <relop> <加法表达式> <可选表达式>

void SemanticFun\_41(STree\* T); //41: <relop> ::= <

//42: <relop> ::= <=

//43: <relop> ::= >

//44: <relop> ::= >=

//45: <relop> ::= ==

//46: <relop> ::= !=

void SemanticFun\_47(STree\* T); //47: <加法表达式> ::= <项> <可选加法表达式>

//51: <项> ::= <因子> <可选项>

void SemanticFun\_48(STree\* T); //48: <可选加法表达式> ::= + <项> <可选加法表达式>

//49: <可选加法表达式> ::= - <项> <可选加法表达式>

//52 : <可选项> :: = \* <因子> <可选项>

//53 : <可选项> :: = / <因子> <可选项>

void SemanticFun\_55(STree\* T); //<因子> ::= num

void SemanticFun\_56(STree\* T); //<因子> ::= ( <表达式> )

void SemanticFun\_57(STree\* T); //<因子> ::= <ID> FTYPE

void SemanticFun\_63(STree\* T); //<实参列表> ::= <表达式> <可选实参列表>

void SemanticFun\_64(STree\* T); //<可选实参列表> ::= , <表达式> <可选实参列表>

void SemanticFun\_Other(STree\* T); //只传递四元式链

int Define\_Check(char IDname[NAME\_LENGTH]);

int Use\_Check(char IDname[NAME\_LENGTH]);

void StableOut();

};

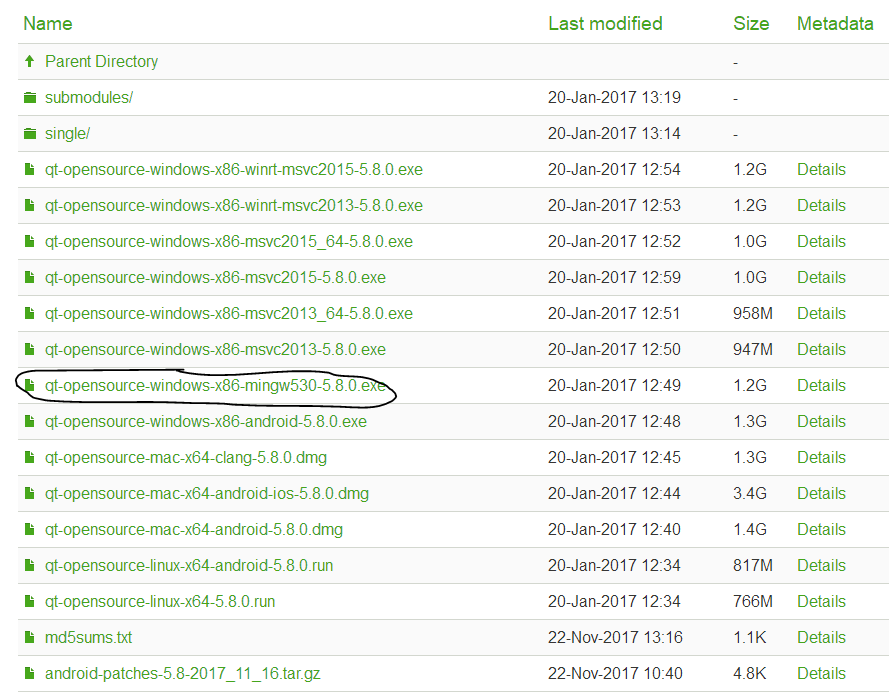
#endif // SEMANTIC\_H

**5.调试运行环境**

编译环境：

Qt5+MingGW32位版本(需要MingGW编译器编译，假如使用MSVC2015编译器进行编译运行出来的exe会无法生成语法分析树中途crash，这是因为MSVC2015不支持中文编码的问题，MSVC2015编译器中,中文输出会提示”常量中有换行符错误”)

QT5+MingGW32下载64位版本地址为：http://download.qt.io/archive/qt/5.8/5.8.0/



运行环境：

Microsoft windows 10 专业版(64位)

内存 8GB（1600 Mhz）

注：由于文法规则grammar-en.txt是利用本地绝对路径读入的，所以移植到其他机器运行时需要修改syntax.cpp中init\_grammar()函数中grammar-en.txt的绝对路径才能正确运行生成语法分析树否则会提示语法分析错误

**6.参考文献**

(1) Term-weighting approaches in automatic text retrieval，Gerard Salton et.

(2) New term weighting formulas for the vector space method in information retrieval

(3) A neural probabilistic language model 2003

(4) Deep Learning in NLP-词向量和语言模型

(5) Recurrent neural network based language models

(6) Statistical Language Models based on Neural Networks，mikolov博士论文

(7) Rnnlm library

(8) A survey of named entity recognition and classification

(9) Deep learning for Chinese word segmentation and POS tagging

(10) Max-margin tensor neural network for chinese word segmentation