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

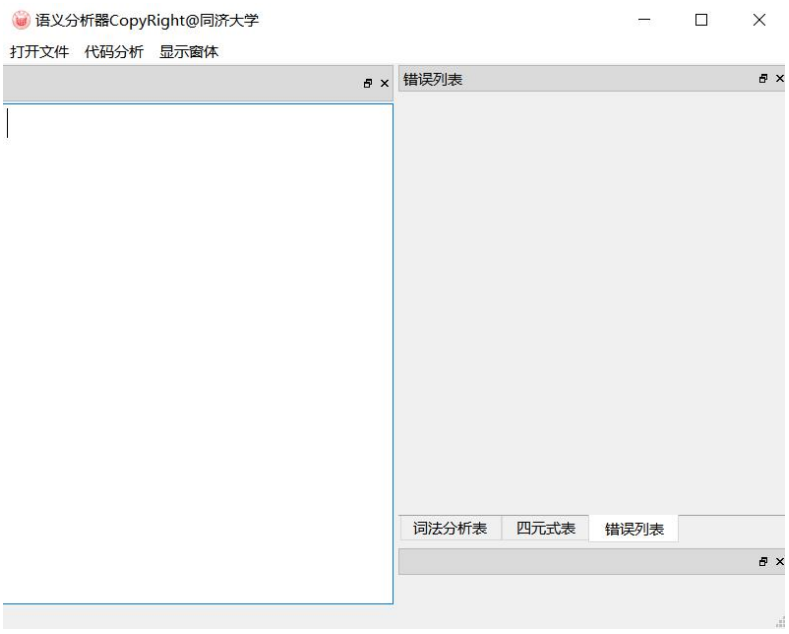
1.语义分析器题目说明

1. 给出源程序的语义分析结果，实现中间代码生成（建议以四元式的形式作为中间代码）；
2. 注意静态语义错误的诊断和处理；
3. 在此基础上，考虑更为通行的高级语言的语义检查和中间代码生成所需要注意的内容，并给出解决方案。

(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

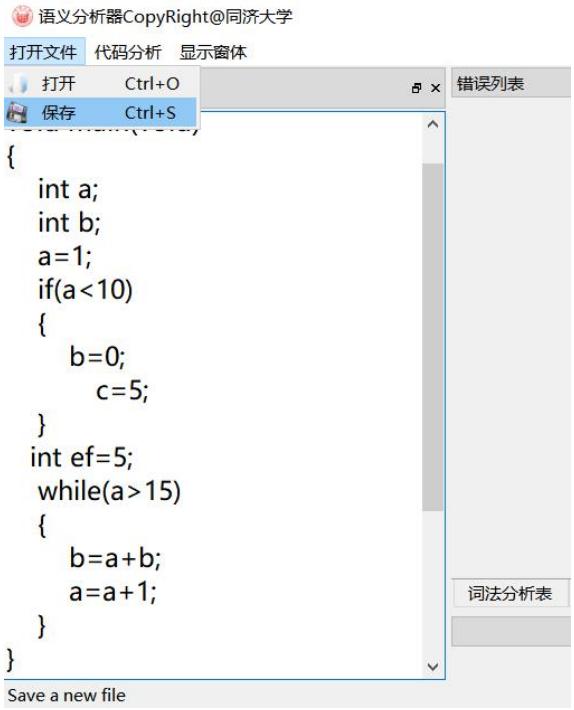
2.1 主界面



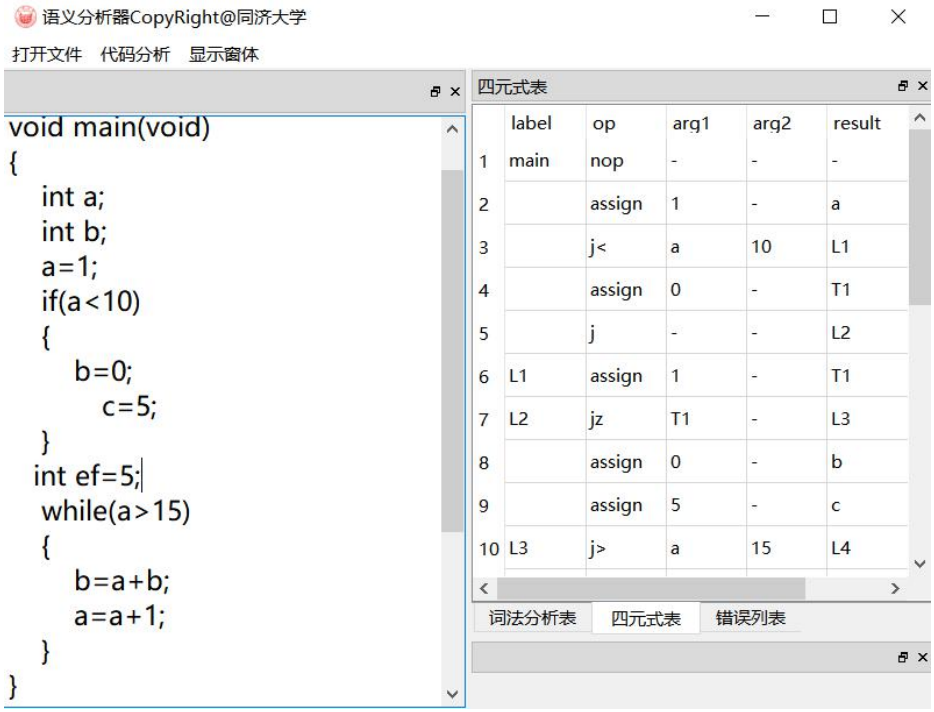
2.2 软件功能

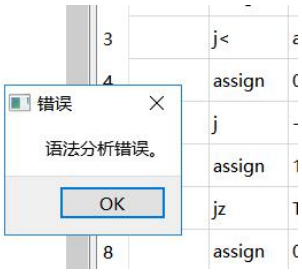
1. 首先打开选择文件 menubar 点击打开文件 action，选择要分析的类 C 语言程序:

2. 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：

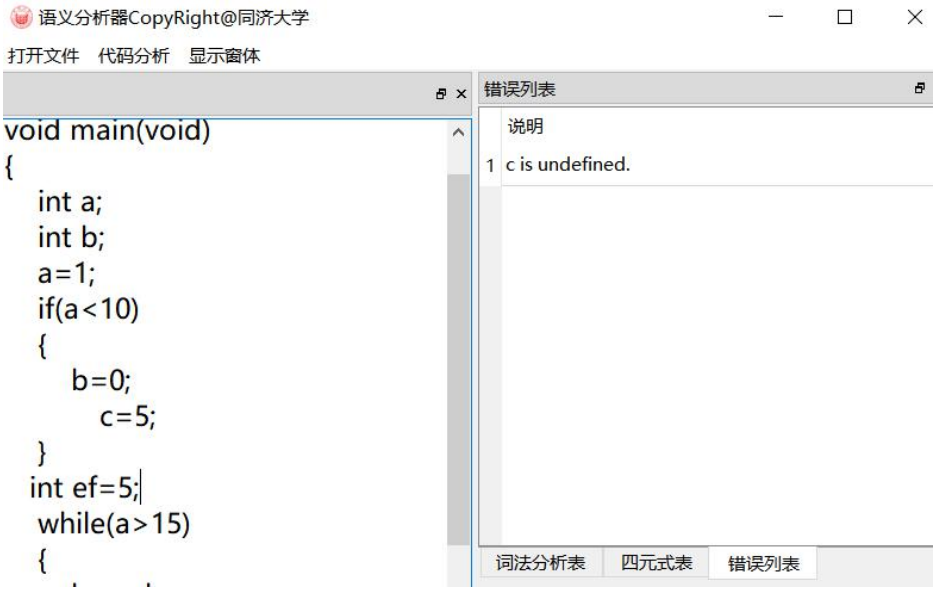


3. 随后点击 menubar 中的语义分析 action，随后点击四元式表便可查看到类 C 程序的语义分析结果，如果语义分析发现语法存在错误，则会弹出语法分析错误窗口：

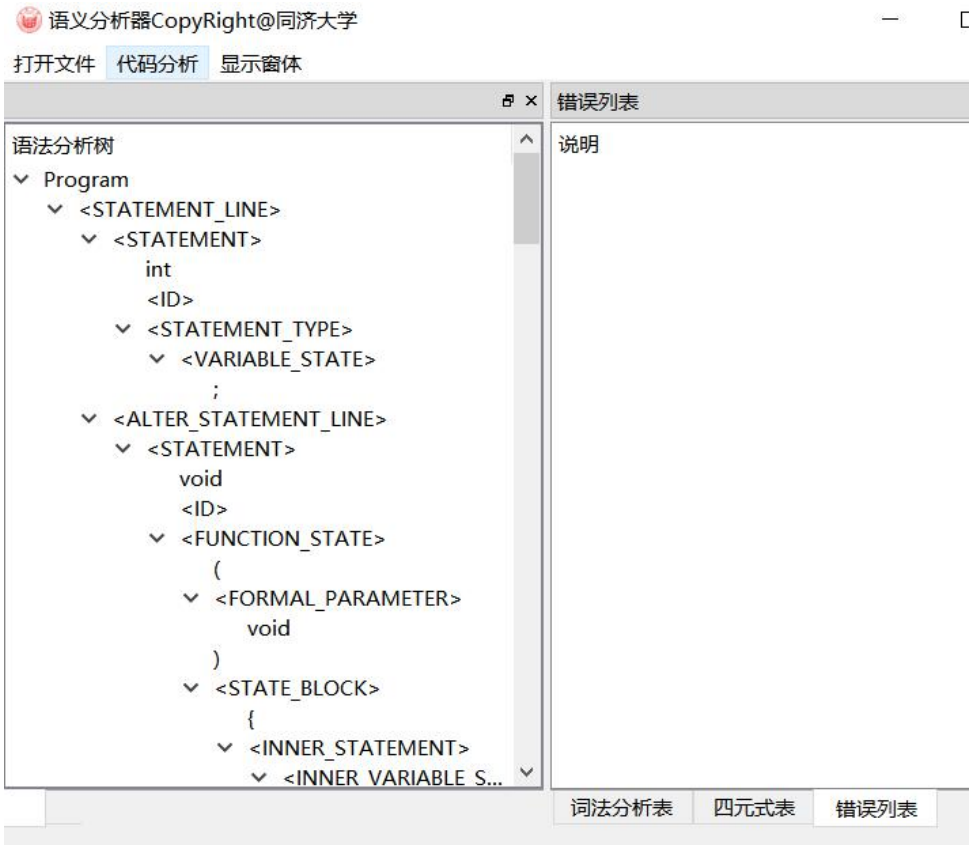




4.如果语法分析正确而语义分析错误，则会在错误列表显示语义分析错误原因：



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：

```
void Semantic::PostOrderTraverse(STree* T)
{
    int i;
    T->flag = 0;
    SemanticFun_Entrance(T);
    for (i = 0; i < T->getchild_num(); i++)
    {
        PostOrderTraverse(T->getChild(i));
    }
    T->flag = 1;
    SemanticFun_Entrance(T);
}
```

(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：

```
Stable.Scurrent = Stable.Scurrent->parent_table;       //一个函数的定义结
Stable.Scurrent->OFFSET += Stable.Scurrent->tail->child_table->OFFSET;

char *FuncLabel = new char[LABEL_LENGTH];
strcpy(FuncLabel,T->child[1]->name);

Quadruples* NewQuad = Generate("nop", "-", "-", "-");
NewQuad->label = FuncLabel;
Quadruples_List* TempList=new Quadruples_List;
TempList->head = NewQuad;
TempList->tail = NewQuad;
QList_Append(T->QLparent, TempList);

QList_Append(T->QLparent, T->QLchild[2]);

if (Define_Check(T->child[1]->name) == 0)
{
    ST_Node* NewSymbol = new ST_Node;
    strcpy(NewSymbol->name, T->child[1]->name);
    strcpy(NewSymbol->type, "int");
    NewSymbol->offset = Stable.Scurrent->OFFSET;
    NewSymbol->child_table = NULL;
    NewSymbol->next = NULL;
    Stable.Scurrent->OFFSET += 4;
    Stable.AddSymbol(NewSymbol);
}
```

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：

```
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
void SemanticFun_36(STree* T);      //36:  <可选else语句块> ::= else <语句块>
```

(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：

```
//将Qlist L2接入到L1的尾部
void Semantic::QList_Append(Quadruples_List* L1, Quadruples_List* L2)
{
    if (L1->head != NULL && L2->head != NULL)
    {
        L1->tail->next = L2->head;
        L1->tail = L2->tail;
    }
    else if (L1->head == NULL && L2->head != NULL)
    {
        L1->head = L2->head;
        L1->tail = L2->tail;
    }
}
```

(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：

```
int num = 1;
int No[MAX_CHILD_NUM] = { 1,-1,-1,-1,-1,-1 };
QList_Connect(num, No, T->QLparent, T->QLchild);
```

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：

```
void Semantic::QList_Connect(int num, int No[MAX_CHILD_NUM], Quadruples_List* QLparent, Quadruples_List* QLchild[MAX_CHILD_NUM])
{
    int i, j;
    //找到第一条非空的四元式链
    for (i = 0; i < num; i++)
    {
        if (QLchild[No[i]]->head != NULL)
            break;
    }
    if (i < num)
    {
        QLparent->head = QLchild[No[i]]->head;
        //往后搜索，如果遇到非空四元式链，进行拼接
        for (j = i + 1; j < num; j++)
        {
            if (QLchild[No[j]]->head != NULL)
            {
                QLchild[No[i]]->tail->next = QLchild[No[j]]->head;
                i = j;
            }
        }
        QLparent->tail = QLchild[No[i]]->tail;
    }

}
```

(6)构造四元式节点的函数如下：

```
Quadruples* Semantic::Generate(const char* op, con
{
    Quadruples* p = new Quadruples;
    strcpy_s(p->op, op);
    strcpy_s(p->arg1, arg1);
    strcpy_s(p->arg2, arg2);
    strcpy_s(p->result, result);
    p->label = NULL;
    p->next = NULL;
    return p;
}
```

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

```
int Semantic::Define_Check(char IDname[NAME_LENGTH])
{
    ST_Node* p = Stable.Scurrent->head;
    while (p)
    {
        if (strcmp(p->name, IDname) == 0)
        {
            ERR_Node* E = new ERR_Node;
            strcpy_s(E->ErrorMessage, IDname);
            strcat(E->ErrorMessage, " redefinition.");
            Etable.AddError(E);
            return -1;
        }
        p = p->next;
    }
    return 0;
}
```

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

```
int Semantic::Use_Check(char IDname[NAME_LENGTH])
{
    SymbolTable* tp= Stable.Scurrent;
    ST_Node *np;
    while (tp)
    {
        np = tp->head;
        while (np)
        {
            if (strcmp(np->name, IDname) == 0)
            {
                return 0;
            }
            np = np->next;
        }
        tp = tp->parent_table;
    }
    ERR_Node* E = new ERR_Node;

    strcpy_s(E->ErrorMessage, IDname);
    strcat(E->ErrorMessage, " is undefined.");
    Etable.AddError(E);

    return -1;
}
```

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，

```
 int row=0;
 for(Quadruples* TempQuad = STREE->QLparent->head; TempQuad != NULL; TempQuad = TempQuad->next,row++)
 {
     QTableWidgetItem *item0, *item1, *item2, *item3, *item4;
     item0 = new QTableWidgetItem;
     item1 = new QTableWidgetItem;
     item2 = new QTableWidgetItem;
     item3 = new QTableWidgetItem;
     item4 = new QTableWidgetItem;
     item0->setText(tr(TempQuad->label));
     table->setItem(row, 0, item0);
     item1->setText(tr(TempQuad->op));
     table->setItem(row, 1, item1);
     item2->setText(tr(TempQuad->arg1));
     table->setItem(row, 2, item2);
     item3->setText(tr(TempQuad->arg2));
     table->setItem(row, 3, item3);
     item4->setText(tr(TempQuad->result));
     table->setItem(row, 4, item4);

 }
  table->setEditTriggers(QAbstractItemView::NoEditTriggers);
```

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：

```
struct Quadruples {
        char op[20];
        char arg1[NAME_LENGTH];
        char arg2[NAME_LENGTH];
        char result[NAME_LENGTH];
        struct Quadruples* next;
        char* label;        //该四元式所在行对应的标号
};
struct Quadruples_List {
        Quadruples* head;
        Quadruples* tail;
};
```

3.4 显示错误列表方式

显示错误列表所用的 Qt 控件为 QtableWidget，

```
int row1=0;
for(ERR_Node* TempErr = SEM.Etable.head; TempErr!= NULL; TempErr = TempErr->next,row1++)
{
    QTableWidgetItem *item0;
    item0 = new QTableWidgetItem;


     QString txt = QString(TempErr->ErrorMessage);
    item0->setText(txt);
    ErrTable->setItem(row1, 0, item0);
   // qDebug(TempQuad->label);

}
ErrTable->setEditTriggers(QAbstractItemView::NoEditTriggers);
```

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。

```
struct ERR_Node {
    char ErrorMessage[100];
    ERR_Node *next;
};
class ErrorTable {
private:
    ERR_Node *head;
    ERR_Node *tail;
public:
    ErrorTable();
    void AddError(ERR_Node* E);
    friend Semantic;
    friend MainWindow;
};
```

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);
    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/

Name	Last modified	Size	Metadata
Parent Directory		-	
submodules/	20-Jan-2017 13:19	-	
single/	20-Jan-2017 13:14	-	
qt-opensource-windows-x86-winrt-msvc2015-5.8.0.exe	20-Jan-2017 12:54	1.2G	Details
qt-opensource-windows-x86-winrt-msvc2013-5.8.0.exe	20-Jan-2017 12:53	1.2G	Details
qt-opensource-windows-x86-msvc2015_64-5.8.0.exe	20-Jan-2017 12:52	1.0G	Details
qt-opensource-windows-x86-msvc2015-5.8.0.exe	20-Jan-2017 12:59	1.0G	Details
qt-opensource-windows-x86-msvc2013_64-5.8.0.exe	20-Jan-2017 12:51	958M	Details
qt-opensource-windows-x86-msvc2013-5.8.0.exe	20-Jan-2017 12:50	947M	Details
qt-opensource-windows-x86-mingw530-5.8.0.exe	20-Jan-2017 12:49	1.2G	Details
qt-opensource-windows-x86-android-5.8.0.exe	20-Jan-2017 12:48	1.3G	Details
qt-opensource-mac-x64-clang-5.8.0.dmg	20-Jan-2017 12:45	1.3G	Details
qt-opensource-mac-x64-android-ios-5.8.0.dmg	20-Jan-2017 12:44	3.4G	Details
qt-opensource-mac-x64-android-5.8.0.dmg	20-Jan-2017 12:40	1.4G	Details
qt-opensource-linux-x64-android-5.8.0.run	20-Jan-2017 12:34	817M	Details
qt-opensource-linux-x64-5.8.0.run	20-Jan-2017 12:34	766M	Details
md5sums.txt	22-Nov-2017 13:16	1.1K	Details
android-patches-5.8-2017_11_16.tar.gz	22-Nov-2017 10:40	4.8K	Details

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