

**课程设计报告**

**题目：基于高级语言源程序格式处理工具**

**课程名称：程序设计综合课程设计**

**专业班级：**

**学 号：**

**姓 名：**

**指导教师： 李 开**

**报告日期： 2022-09-01**

**计算机科学与技术学院**

**目录**

[1.引言 3](#_Toc115256700)

[1.1问题描述 3](#_Toc115256701)

[1.2需求与技术现状分析 3](#_Toc115256702)

[2.程序总体设计 4](#_Toc115256703)

[2.1设计目的 4](#_Toc115256704)

[2.2设计要求 4](#_Toc115256705)

[2.3总体程序设计思路 4](#_Toc115256706)

[2.4流程图 5](#_Toc115256707)

[3.数据结构和算法详细设计 6](#_Toc115256708)

[3.1词法分析设计 6](#_Toc115256709)

[3.2语法分析设计 8](#_Toc115256710)

[3.3格式处理设计 9](#_Toc115256711)

[3.4保存文件设计 9](#_Toc115256712)

[4. 系统实现和测试 9](#_Toc115256713)

[4.1词法分析 9](#_Toc115256714)

[4.2语法分析 14](#_Toc115256715)

[4.3格式处理与文件保存 16](#_Toc115256716)

[4.4.系统测试 17](#_Toc115256717)

[5. 总结与展望 36](#_Toc115256718)

[5.1全文总结 36](#_Toc115256719)

[5.2工作展望 37](#_Toc115256720)

[6.体会 37](#_Toc115256721)

[6.1总结 37](#_Toc115256722)

[6.2特色 38](#_Toc115256723)

[6.3不足 38](#_Toc115256724)

[参考文献 38](#_Toc115256725)

[附录 39](#_Toc115256726)

[附录1 AST.cpp： 39](#_Toc115256727)

[附录2 AST.h： 40](#_Toc115256728)

[附录3 lexer.cpp： 41](#_Toc115256729)

[附录4 lexer.h： 54](#_Toc115256730)

[附录5 main.cpp： 58](#_Toc115256731)

[附录6 parsing.cpp： 59](#_Toc115256732)

[附录7 parsing.h： 92](#_Toc115256733)

[附录8 pre\_process.cpp： 93](#_Toc115256734)

[附录9 pre\_process.h： 96](#_Toc115256735)

[附录10 print\_file.cpp： 97](#_Toc115256736)

[附录11 print\_file.h： 102](#_Toc115256737)

[附录12 Queue.cpp： 103](#_Toc115256738)

[附录13 Queue.h： 105](#_Toc115256739)

[附录14 Stack.cpp： 106](#_Toc115256740)

[附录15 Stack.h： 108](#_Toc115256741)

[附录16 Traverse.cpp： 110](#_Toc115256742)

[附录17 Traverse.h： 115](#_Toc115256743)

## 1.引言

### 1.1问题描述

抽象语法树(abstract syntax code, AST) 是源代码的抽象语法结构的树状表示，树上的每个节点都表示源代码中的一种结构， 这所以说是抽象的，是因为抽象语法树并不会表示出真实语法出现的每一个细节， 比如说，嵌套括号被隐含在树的结构中，并没有以节点的形式呈现。抽象语法树并不依赖于源语言的语法，也就是说语法分析阶段所采用的上下文无文文法，因为在写文法时，经常会对文法进行等价的转换(消除左递归，回溯，二义性等) , 这样会给文法分析引入一些多余的成分，对后续阶段造成不利影响，甚至会使合个阶段变得混乱。因些，很多编译器经常要独立地构造语法分析树为前端，后端建立一个清晰的接口。

### 1.2需求与技术现状分析

抽象语法树(Abstract Syntax Tree ,AST)作为程序的一种中间表示形式,在[程序分析](https://baike.baidu.com/item/%E7%A8%8B%E5%BA%8F%E5%88%86%E6%9E%90)等诸多领域有广泛的应用.利用抽象语法树可以方便地实现多种[源程序](https://baike.baidu.com/item/%E6%BA%90%E7%A8%8B%E5%BA%8F)处理工具,比如源程序浏览器、智能编辑器、[语言翻译器](https://baike.baidu.com/item/%E8%AF%AD%E8%A8%80%E7%BF%BB%E8%AF%91%E5%99%A8/6119005)等。通常是作为[编译器](https://baike.baidu.com/item/%E7%BC%96%E8%AF%91%E5%99%A8)或[解释器](https://baike.baidu.com/item/%E8%A7%A3%E9%87%8A%E5%99%A8)的组件出现的，它的作用是进行语法检查、并构建由输入的单词组成的数据结构（一般是[语法分析树](https://baike.baidu.com/item/%E8%AF%AD%E6%B3%95%E5%88%86%E6%9E%90%E6%A0%91)、[抽象语法树](https://baike.baidu.com/item/%E6%8A%BD%E8%B1%A1%E8%AF%AD%E6%B3%95%E6%A0%91)等层次化的数据结构）。语法分析器通常使用一个独立的[词法分析器](https://baike.baidu.com/item/%E8%AF%8D%E6%B3%95%E5%88%86%E6%9E%90%E5%99%A8)从输入字符流中分离出一个个的“单词”，并将单词流作为其输入。实际开发中，语法分析器可以手工编写，也可以使用工具（半）自动生成。

个人通过对源程序的实现，发现各种编译器的中间表示就是语法分析树，AST，感觉本次实验的最终目的就是编译出来一个简单的语言编译器，所以我认为语法分析树目前的现状就是编译器的内部原理，现在实现的程序比较简单，词法识别和语法识别还不够全面，只能够识别比较简单的较为基本的源程序。

## 2.程序总体设计

### 2.1设计目的

编写一个具有词法分析（能够识别出各种单词）和语法分析（生成语法分析树AST）和能够将输入的文本程序以一定的格式输出并且保存在文件中。

### 2.2设计要求

要求具有如下功能：

**总体设计功能：**由源程序到抽象语法树的过程，逻辑上包含2个重要的阶段，一是词法分析，识别出所有按词法规则定义的单词；二是语法分析，根据定义的语法规则，分析单词序列是否满足语法规则，同时生成抽象语法树。

**输入输出功能：**本程序以读取.c文件的形式进行输入，输出时可以进行保存文件，并且保存的文件中的源代码有一定的规范。

**（1）词法分析过程：**词法分析需要识别出五类单词，标识符、关键字、常量、运算符和定界符，词法分析每识别出一个单词，就可返回单词的编码。该系统可以有效地识别关键字，字符，整形数字，长整型数字，变量id，数组，单目操作符号，和双目操作符号，以及宏定义和行注释与段注释

**（2）语法分析生成AST语法树的过程：**通过识别出来的各种字符，采用的实现方法是编译技术中的递归下降子程序法，生成一个识别各种结构的语法树，例如复合语句，函数参数，函数参数列表，关键字int和id的赋值，if—else语句，while循环语句等

### 2.3总体程序设计思路

本程序使用C/C++语言实现，程序分为五部分：词法分析（lexer），语法分析（parsing）, 格式化处理(traverse)，保存文件(print\_file)以及其他数据结构等，分别将4个处理阶段写在4个cpp文件中，分别是lexer.cpp，parsing.cpp,traverse.cpp和printf\_file.cpp,四个文件分别完成词法分析，语法分析，格式化处理和保存文件的任务。

Lexer.cpp的主要工作是检查注释是否合法、词法分析获取token\_text。

parsing.cpp的主要工作是根据词法分析之后的token\_text进行语法分析，生成语法树。

Traverse.cpp是根据生成的语法分析树，采用先根遍历的方式将程序代码进行格式化处理，最后输出格式化处理后的语法分析树。

Print\_file.cpp负责将格式化后的程序保存在指定文件里面，

Pre\_process.cpp对预处理部分进行解析，

此外还有stack.cpp,queue.cpp,AST.cpp来提供项目所需要的数据结构。

表2.1代码块及功能简要概述

|  |  |
| --- | --- |
| 模块 | 功能 |
| main | 程序入口 |
| Lexer | 词法分析 |
| Parsing | 语法分析 |
| Traverse | 格式化处理，并输出语法分析树 |
| Print\_file | 保存文件 |
| Pre\_process | 对预处理部分进行解析 |
| Stack | 创建数据结构栈 |
| Queue | 创建数据结构队列 |
| AST | 创建抽象语法树 |

### 2.4流程图

如图2.1

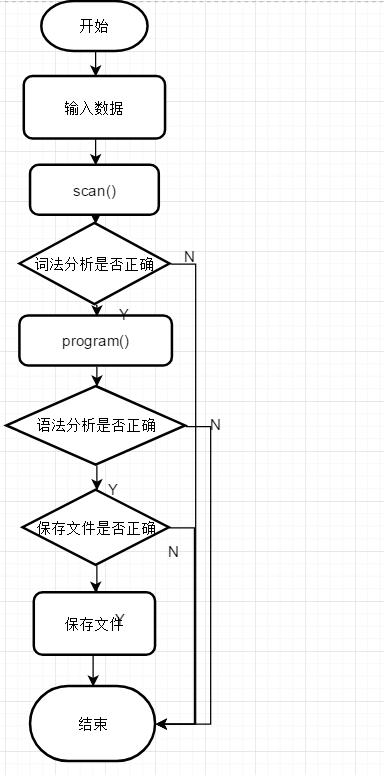


图2.1系统总体流程图

## 3.数据结构和算法详细设计

### 3.1词法分析设计

词法分析包含的数据元素：

表3.1词法分析数据

|  |  |
| --- | --- |
| 程序数据项定义 | 输入数据 |
| 字符串 | 字符串常量 |
| 数字 | 整型，长整型，无符号整型，无符号长整型，浮点型，双精度浮点型 |
| 预处理 | 1. #include <xxx.h> 2. #include ”xxx.h” 3. #define xxx xxx |
| 关键字，保留字 | AUTO,BREAK,CONTINUE等 |
| 定界符 | 括号，逗号，分号，引号 |
| 运算符 | 赋值运算符，逻辑运算符，关系运算符，单目算术运算符 |

词法分析思路：将关键字和枚举变量值编成哈希表(rule：第一个字符对应的ASCII码对10求模，值对应第一个下标，每行最后设置哨兵none)，存入KeyWordHashTable中，然后在代码块lexer.cpp里面在筛去空格变成一个个单词统计输入代码段总体的行数，统计每一行除去空格的字符串的长度，然后通过循环和条件判断语句识别出单词，关键字，符号，数字等等。

词法分析的DFA如下所示，一共分为5中判断：分别能够识别出字母，整型数字，长整型数字，关键字，变量id，符号，注释，宏定义等等。每次从状态0开始，从源程序文件中读取一个字符，可以到达下一个状态，当到达环形的状态（结束状态）时，表示成功的读取到了一个单词，返回单词的编码，保存在全局变量token\_text中。而单词的自身值放在token\_textstring中。

如图3.1

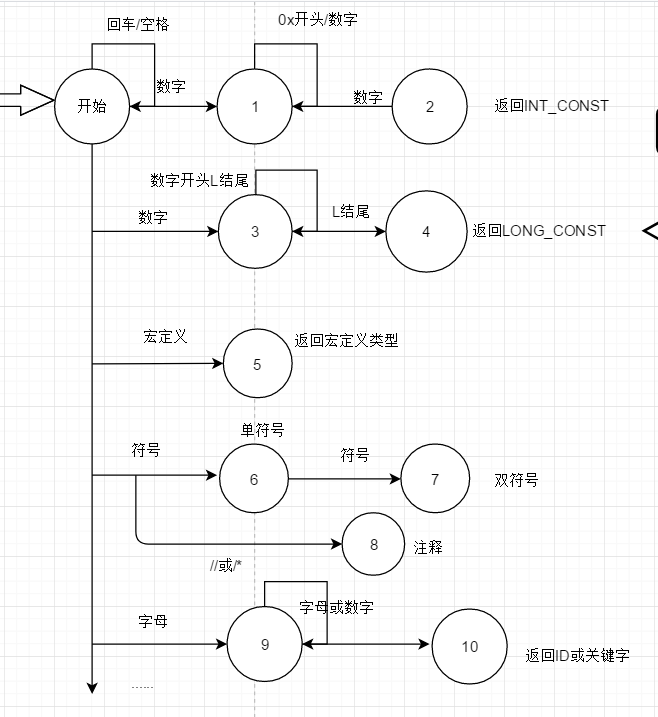


图3.1DFA转换图

### 3.2语法分析设计

语法分析采用递归下降方法的程序结构：在词法分析当中识别出来的单词都通过数组传递到getton\_tokentext()函数里面返回单词相应的编码，并生成一个结构数组，将单词的原有值保存起来，首先进入program()子程序块，这是递归下降程序的开始，通过program程序递归调用其他程序，然后再其他的程序里面也有相应的递归调用，到达功能相符的程序，因为一个源代码开头一定是宏定义，之后或者是外部变量的定义或者是函数类型的定义，可以依照这个规律进行规划子程序的顺序。

如图3.2

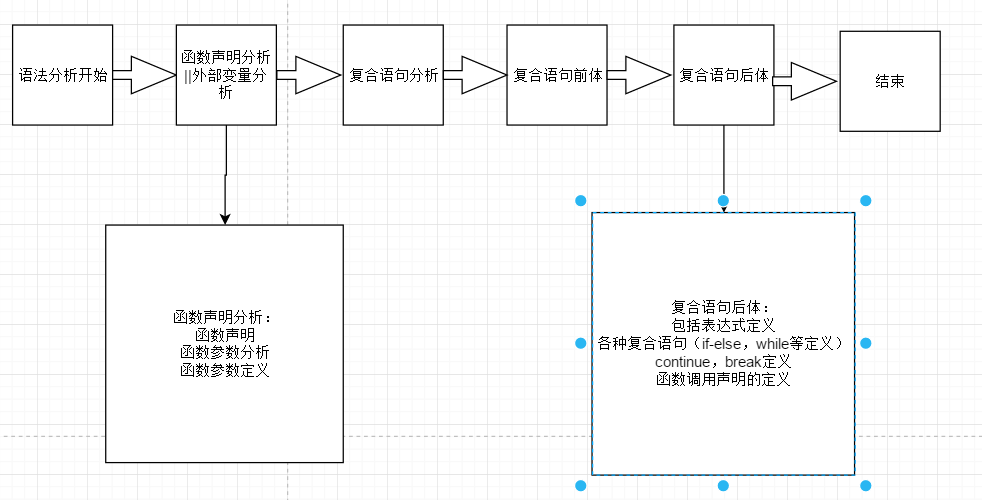


图3.2语法分析设计图

图3.3 AST语法树

### 3.3格式处理设计

依照上述program()代码块生成的分析树先根遍历，其中不同的结点为标记点对后续的符号进行判断，因为一些符号例如’(‘,’{‘等等一些符号并不会写入语法分析书里面，所以要在保存文件的过程中加上。这是一个比较难判断的点，但是在进行词法分析的过程中，将单词的原值保存在了一个结构数组里面，子要将分析树的结点和数组元素对应上，这个功能还是可以实现的。

### 3.4保存文件设计

设计同格式处理，省略了对语句的描述信息

## 系统实现和测试

### 4.1词法分析

#### 4.1.1词法分析数据定义

enum token\_kind

{

    /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*标识符\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

    IDENT,

    /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*保留字\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

    AUTO,

    BREAK,

    CASE,

    CONTINUE,

    DEFAULT,

    DO,

    RETURN,

    SIZEOF,

    SWITCH,

    TYPEDEF,

    VOID,

    VOLATILE,

    WHILE,

    ELSE,

    EXTERN,

    FOR,

    GOTO,

    IF,

    CONST,

    STATIC,

    REGISTER,

    CHAR,

    DOUBLE,

    ENUM,

    FLOAT,

    INT,

    LONG,

    SIGNED,

    SHORT,

    STRUCT,

    UNION,

    UNSIGNED,

    /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*定界符\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

    //起止符号

    BEGIN\_END, //#

    COMMA,           // ,

    SEMICOLON,       // ;

    LPARENTHESE,     // (

    RPARENTHESE,     // )

    LSUBSCRIPT,      // [

    RSUBSCRIPT,      // ]

    LCURLYBRACE,     // {

    RCURLYBRACE,     // }

    SINGLE\_QUO\_MARK, // '

    DOUBLE\_QUO\_MARK, // "

    /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*运算符\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

    //赋值运算符

    ASSIGN,      // =

    PLUS\_EQ,     // +=

    MINUS\_EQ,    // -=

    MULTIPLY\_EQ, // \*=

    DIVIDE\_EQ,   // /=

    MOD\_EQ,      // %=

    //逻辑运算符（双目）

    AND, // &&

    OR,  // ||

    //关系运算符

    EQUAL,         // ==

    UNEQUAL,       // !=

    LESS,          // <

    MORE,          // >

    LESS\_OR\_EQUAL, // <=

    MORE\_OR\_EQUAL, // >=

    //单目算术运算符

    PLUS,     // +

    MINUS,    // -

    MULTIPLY, // \*

    DIVIDE,   // /

    MOD,      // %

    /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

     \* 暂不支持:

    //算术运算符（单目）

    PLUSPLUS,   // ++

    MINUSMINUS, // --

    NON,        // !单目逻辑运算符

    //位运算符

    BIT\_AND,   // 位运算与&

    BIT\_OR,    // |

    BIT\_XOR,   // ^

    BIT\_NON,   // ~

    BIT\_LEFT,  // <<

    BIT\_RIGHT, // >>

    //赋值运算符（位运算）

    BIT\_LEFT\_EQ,  // <<=

    BIT\_RIGHT\_EQ, // >>=

    BIT\_AND\_EQ,   // &=

    BIT\_XOR\_EQ,   // ^=

    BIT\_OR\_EQ,    // |=

    GET\_ADDRESS,  // 取地址符&

    POINTER,      // 取变量符\*

    CONDITION,    // 条件表达式？：

    \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

    /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*常量\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

    INT\_CONST,

    UNSIGNED\_CONST,

    LONG\_CONST,

    UNSIGNED\_LONG\_CONST,

    FLOAT\_CONST,

    DOUBLE\_CONST,

    LONG\_DOUBLE\_CONST,

    CHAR\_CONST,

    STRING\_CONST,

    /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*预处理标识符\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

    DEFINE,

    INCLUDE,

    /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*报错符\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

    ERROR\_TOKEN,

}; //end of enum token\_kind

包含了大部分的定界符，运算符，常量，以及所有的关键字，保留字，同时还有预处理标识符，此外还定义了个人需要使用的报错符，标识符。

#### 4.1.2 词法分析结构定义

关键字字符串值与枚举常量值对应结构（用于构建哈希表）

typedef struct KeyWord\_name\_value

{

    char name[10];

    int value;

} KeyWord\_name\_value;

关键字和枚举常量的哈希表：

KeyWord\_name\_value KeyWordHashTable[10][10] = {

    // ascii码 mod 10 ==0:d,n,x

    {{"double", DOUBLE}, {"do", DO}, {"default", DEFAULT}, {"none", IDENT}},

    // ascii码 mod 10 ==1:e,o,y

    {{"enum", ENUM}, {"extern", EXTERN}, {"else", ELSE}, {"none", IDENT}},

    // ascii码 mod 10 ==2:f,p,z

    {{"float", FLOAT}, {"for", FOR}, {"none", IDENT}},

    // ascii码 mod 10 ==3:g,q

    {{"goto", GOTO}, {"none", IDENT}},

    // ascii码 mod 10 ==4:h,r

    {{"register", REGISTER}, {"return", RETURN}, {"none", IDENT}},

    // ascii码 mod 10 ==5:i,s

    {{"short", SHORT},

     {"int", INT},

     {"struct", STRUCT},

     {"sighed", SIGNED},

     {"static", STATIC},

     {"if", IF},

     {"switch", SWITCH},

     {"sizeof", SIZEOF},

     {"none", IDENT}},

    // ascii码 mod 10 ==6:j,t

    {{"typedef", TYPEDEF}, {"none", IDENT}},

    // ascii码 mod 10 ==7:a,k,u

    {{"auto", AUTO}, {"union", UNION}, {"unsigned", UNSIGNED}, {"none", IDENT}},

    // ascii码 mod 10 ==8:b,l,v

    {{"long", LONG},

     {"volatile", VOLATILE},

     {"void", VOID},

     {"break", BREAK},

     {"none", IDENT}},

    // ascii码 mod 10 ==9:c,m,w

    {{"char", CHAR},

     {"const", CONST},

     {"case", CASE},

     {"while", WHILE},

     {"continue", CONTINUE},

     {"none", IDENT}},

}; // end of KeyWordHashTable

哈希表说明：第一个字符的ASCII码对10求模，值对应数组第一个分量下标

 每行最后设置哨兵none，枚举类型值对应为IDENT

Token结构体的定义：

struct token\_info

{

    int token\_kind;      //种类码

    int line;            //行数

    char token\_text[50]; //自身值

};

#### 4.1.3 词法分析函数定义

表4.1词法分析函数及其功能

|  |  |
| --- | --- |
| 函数 | 功能 |
| int isKeyWord(char str[]); | 返回单词关键字的种类编码 |
| bool isNum(char c); | 鉴定字符是否为数字 |
| bool isLetter(char c); | 鉴定字符是否为字母 |
| token\_info get\_token(FILE \*p, int &line\_num, char token\_text[]); | 获取单词，并返回单词的信息 |

### 4.2语法分析

#### 4.2.1语法分析树结点定义

实现语法分析树的方法是，把每一个语句都当作一个节点来做，同一层级的语句用兄弟结点连接，不同层级的附属语句用子结点连接，代码的编写和理解阅读的过程中相对来说比较容易，例如，一个函数声明语句，首先生成一个函数声明的结点，将返回类型作为子结点，而返回类型，函数语句，参数列表以及函数内容用兄弟结点相连。

typedef struct treeNode {

  int node\_type; //-1:该节点为语法单元 其他:该节点token\_kind值

  char self\_value[50];  //语句类型或token自身值

  treeNode \*nextBro;    //下一个兄弟结点

  treeNode \*firstChild; //第一个子结点

} \* p\_treeNode;

#### 4.2.2语法分析函数定义

bool is\_Type\_Specifier(int token\_type); // 判断是否是类型说明符

bool is\_Operator(int token\_type); // 判断是否是运算符

bool is\_CONST\_value(int token\_type); // 判断是否是常量

p\_treeNode Program(Queue &Q, FILE \*p, int &line, char token\_text[]); // 创建程序结点

p\_treeNode ex\_Declaration\_List(Queue &Q, p\_treeNode ex\_declaration\_list, FILE \*p, int &line, char token\_text[]); // 创建外部定义序列

p\_treeNode ex\_Val\_Declaration(Queue &Q, p\_treeNode val\_dec, FILE \*p, int &line, char token\_text[]); // 生成外部变量声明

p\_treeNode Fun\_Declaration(Queue &Q, p\_treeNode fun\_dec, FILE \*p, int &line, char token\_text[]); // 生成函数定义或函数声明结点

p\_treeNode Type\_Specifier(token\_info w, p\_treeNode type\_specifier); // 生成种类识别符结点

p\_treeNode Val\_List(Queue &Q, p\_treeNode val\_list, FILE \*p, int &line, char token\_text[]); // 生成变量序列结点

p\_treeNode Params(Queue &Q, p\_treeNode params, FILE \*p, int &line, char token\_text[]); // 生成参数集结点

p\_treeNode Param\_List(Queue &Q, p\_treeNode param\_list, FILE \*p, int &line, char token\_text[]); // 创建参数列表结点

p\_treeNode Param(Queue &Q, p\_treeNode param, FILE \*p, int &line, char token\_text[]); // 生成参数结点

p\_treeNode Statement(Queue &Q, p\_treeNode statement, FILE \*p, int &line, char token\_text[]); // 生成语句结点

p\_treeNode Compound\_Statement(Queue &Q, p\_treeNode compound\_statement, FILE \*p, int &line, char token\_text[]); // 创建复合语句结点

p\_treeNode Selection\_Statement(Queue &Q, p\_treeNode selection\_statement, FILE \*p, int &line, char token\_text[]); // 生成选择语句结点

p\_treeNode Condition\_Expression(Queue &Q, p\_treeNode condition\_expression, FILE \*p, int &line, char token\_text[]); // 创建条件表达式结点

p\_treeNode IF\_Statement(Queue &Q, p\_treeNode if\_statement, FILE \*p, int &line, char token\_text[]); // 创建IF\_STATEMENT结点

p\_treeNode IF\_ELSE\_Statement(Queue &Q, p\_treeNode if\_else\_statement, p\_treeNode if\_statement, FILE \*p, int &line, char token\_text[]); // 创建IF\_ELSE\_STATEMENT语句的结点

p\_treeNode ELSE\_Statement(Queue &Q, p\_treeNode else\_statement, FILE \*p, int &line, char token\_text[]); // 创建ELSE\_STATEMENT的结点

p\_treeNode While\_Statement(Queue &Q, p\_treeNode while\_statement, FILE \*p, int &line, char token\_text[]); // 创建WHILE\_STATEMENT的结点

p\_treeNode For\_Statement(Queue &Q, p\_treeNode for\_statement, FILE \*p, int &line, char token\_text[]); // 创建FOR\_STATEMENT的结点

p\_treeNode Local\_Val\_Declaration(Queue &Q, p\_treeNode local\_val\_declaration, FILE \*p, int &line, char token\_text[]); // 创建局部变量定义结点

p\_treeNode Fun\_Call(Queue &Q, p\_treeNode fun\_Call, FILE \*p, int &line, char token\_text[]); // 创建函数调用结点

p\_treeNode Params\_Call(Queue &Q, p\_treeNode params\_call, FILE \*p, int &line, char token\_text[]); // 创建函数调用的参数语句结点

p\_treeNode Return\_Statement(Queue &Q, p\_treeNode return\_statement, FILE \*p, int &line, char token\_text[]); // 创建返回语句的结点

p\_treeNode Break\_Statement(Queue &Q, p\_treeNode break\_statement, FILE \*p, int &line, char token\_text[]); // 创建break语句的结点

p\_treeNode Continue\_Statement(Queue &Q, p\_treeNode continue\_statement, FILE \*p, int &line, char token\_text[]); // 创建continue语句的结点

p\_treeNode Expression(Queue &Q, p\_treeNode expression, FILE \*p, int &line, char token\_text[], int endsym); // 创建表达式结点

p\_treeNode Const\_Value(token\_info w, p\_treeNode const\_value); //创建常量语句结点

### 4.3格式处理与文件保存

#### 4.3.1格式处理函数定义

void printTabs(int tabs); // 输出TAB

void printFunCall(p\_treeNode fun\_call); // 对函数调用语句进行分析处理

void traverseExp(p\_treeNode p); // 对结点进行分析处理

void traverse(p\_treeNode p, int tabs); // 递归处理抽象语法树，并打印

#### 4.3.2文件保存函数定义

void f\_printTabs(FILE \*fp, int tabs); // 输出TAB

void f\_printFunCall(FILE \*fp, p\_treeNode fun\_call); // 对函数调用语句进行分析处理

void f\_traverseExp(FILE \*fp, p\_treeNode p, bool newRoll); // 对结点进行分析处理

void f\_traverse(FILE \*fp, p\_treeNode p, int tabs, bool newRoll); // 递归将代码保存到文件

### 4.4.系统测试

实验环境条件：Linux version 3.10.0-1160.el7.x86\_64

编译工具：cmake version 3.24.0

#### 4.3.1功能测试

**测试样例1：**

**输入代码：**

#include<stdio.h>

#include<stdlib.h>

#define OK 1

int main()

{

// declare variable

int a = 100L;

double b = .37L;

long c = 57u;

int arr[10];

char d = 'c';

a = 3;

d = 'd';

// if

if (a > 0)

a+=1;

// if else if

if ( a > 0 || b <= 0 + 2)

{

a-=1;

}

else if ( c != 2 )

{

c /= a;

}

else

{

c \*= b;

}

// while

while( a < 2 ){

a+=1;

if ( a > 9 ){

break;

}else{

continue;

}

}

// for

for (int i = 0 ; i < 2 ; i+=1)

{

}

return 0;

}

**运行结果：**

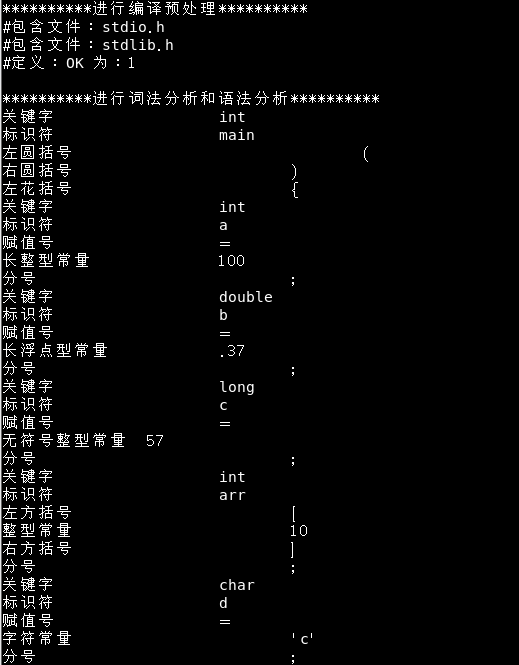
****

图4.1词法分析和语法分析1-1

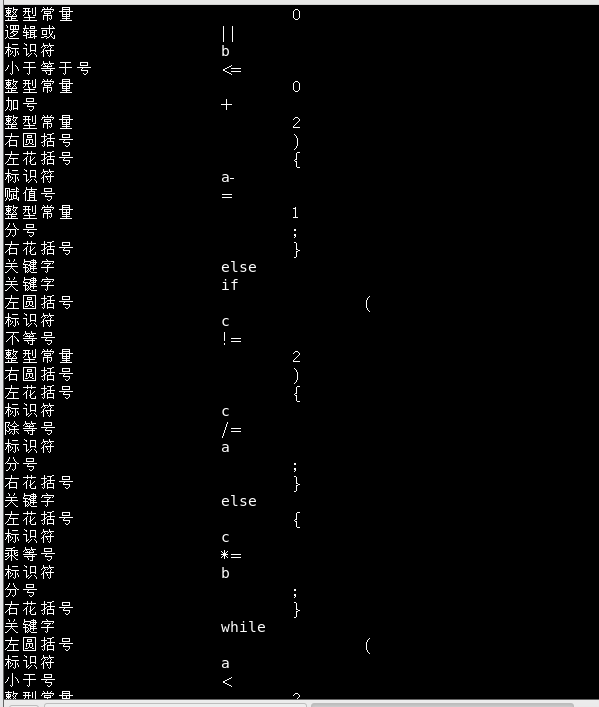
****

图4.2词法分析和语法分析1-2

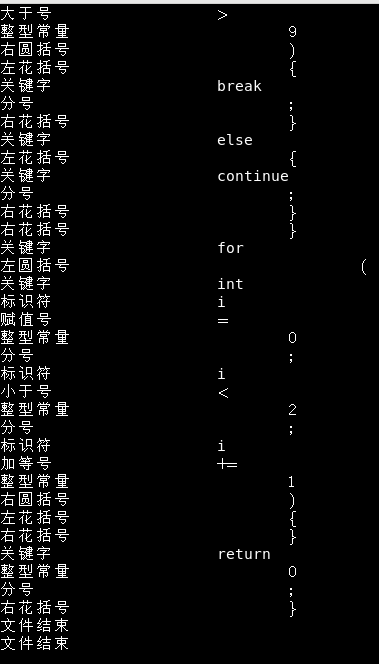
****

图4.3词法分析和语法分析1-3

****

图4.4格式化处理1-1

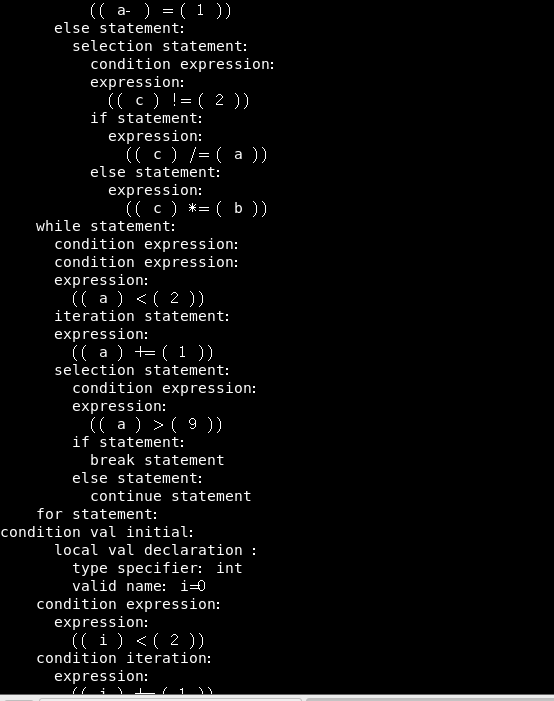
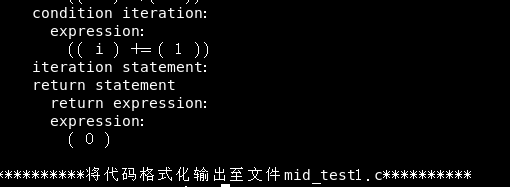
****

图4.5格式化处理1-2

****

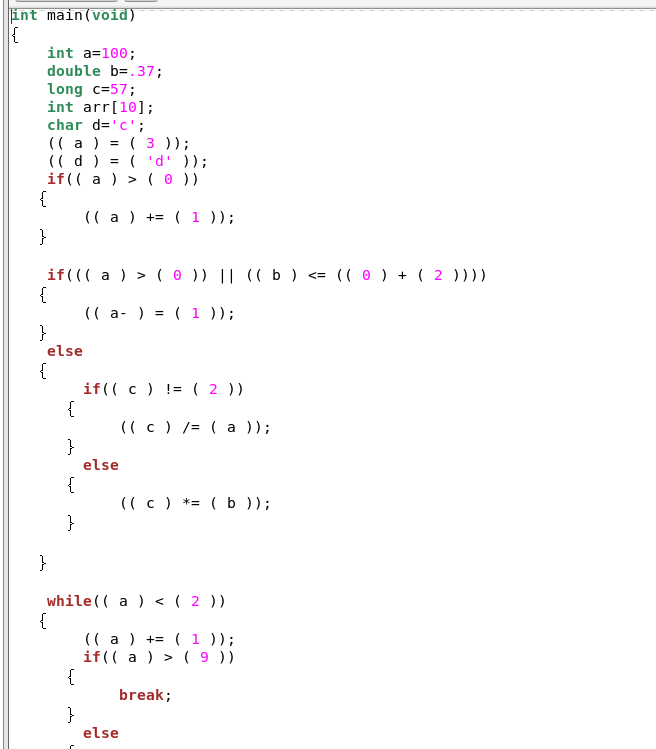
****

图4.7文件保存1-1

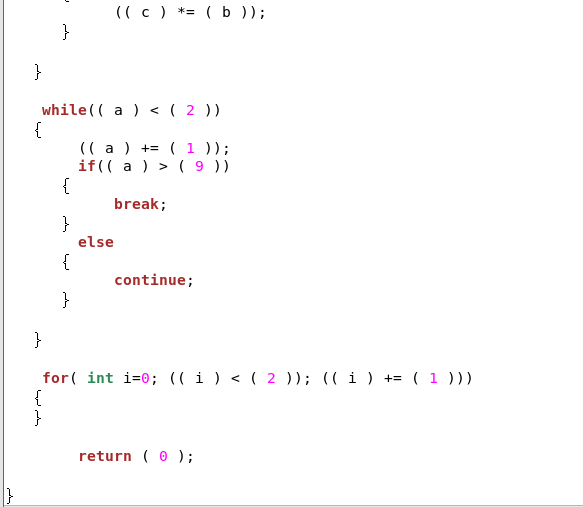
****

图4.8文件保存1-2

**测试样例2：**

**输入代码：**

#include <stdlib.h>

#include "test.h"

#define DEF def

//行注释

//line comment

// declare func

void funcDeclare(int i, float f);

void funcVoid();

double globVar = 1, globarr[9];

/\*

\*block comment

\*

\*/

int main()

{

// 10 16 8 l u

int i\_1 = 123ul, i\_2 = 0x9b7c, i\_3 = 0125;

float f\_1 = .1l, f\_2 = .3e4;

double d = .314e3;

char c\_1 = '\0x8';

char s[0x19] = "May there be no course design in heaven";

if (c\_1 == 's' || i\_1 != i\_2)

{

funcDeclare(i, 0.1);

if (c\_1 == 9)

{

globVar \*= 2;

}

}

else if (d == .27 || c\_1 > '9')

{

funcDeclare(i, 0.1);

}

else { i\_1 += 0;}

while (i\_1 >= 0 && f\_1 < 1 + 3)

{

for (i\_1 = 0; i\_1 <= c\_1; i\_1 += 1)

{

i\_1 += 0;

continue;

}

break;

}

// i\_1=f\_1%98\*((8 - 2)+(9-i\_3)/4);

return i\_1;

}

**运行结果：**

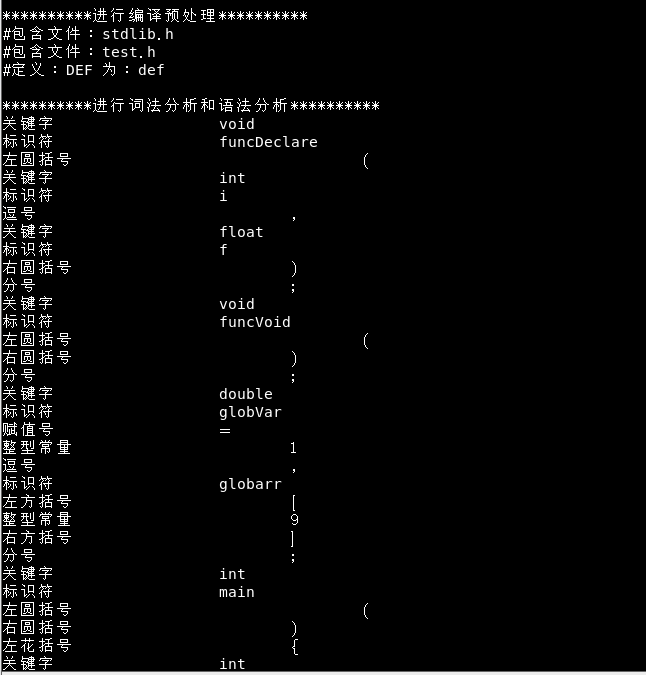
****

图4.9词法分析和语法分析2-1

****

图4.10词法分析和语法分析2-2

****

图4.11词法分析和语法分析2-3

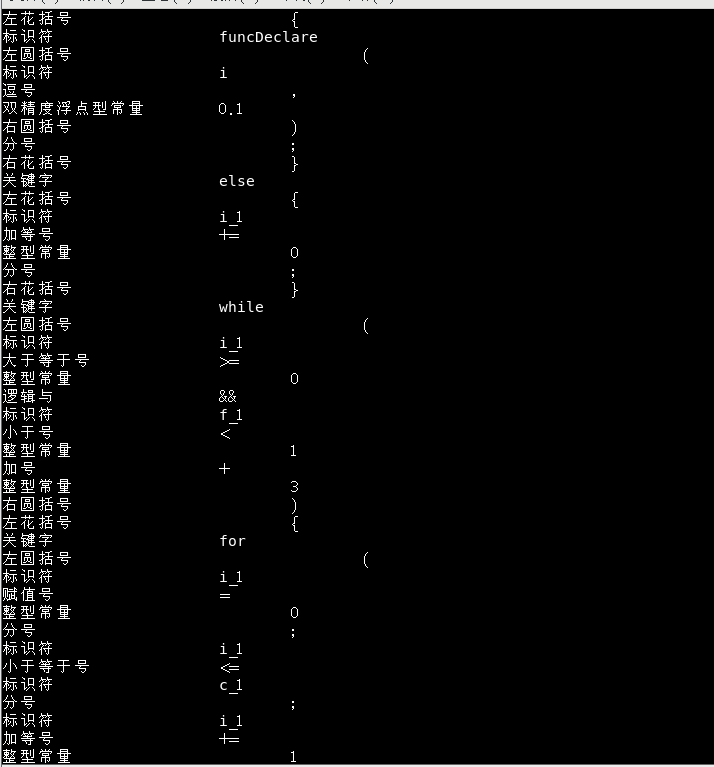
****

图4.12词法分析和语法分析2-4

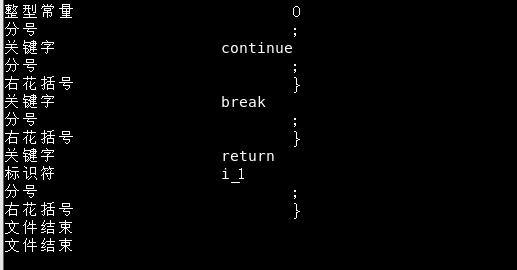
****

图4.13词法分析和语法分析2-5

****

图4.14格式化处理2-1

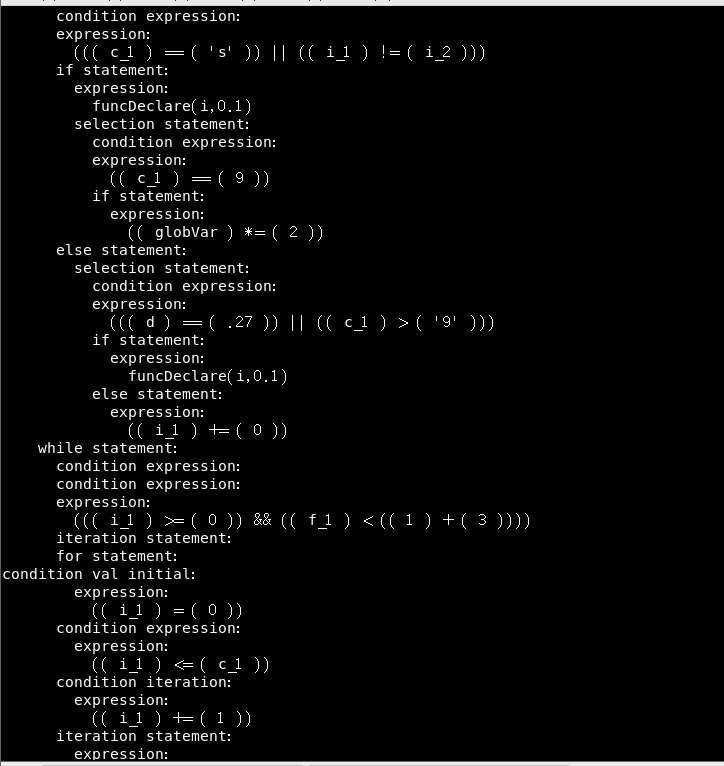
****

图4.15格式化处理2-2

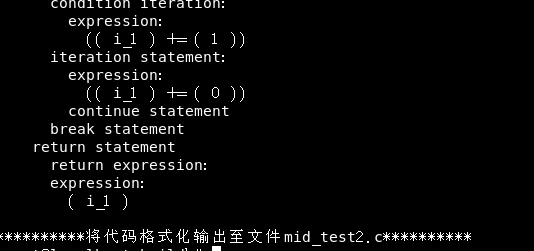
****

图4.16格式化处理2-3****

图4.17文件保存2-1

#### 4.3.2异常处理

**测试样例1：**

**输入代码：**

int main()

{

int fun()

int a = 1;

}

**运行结果：**

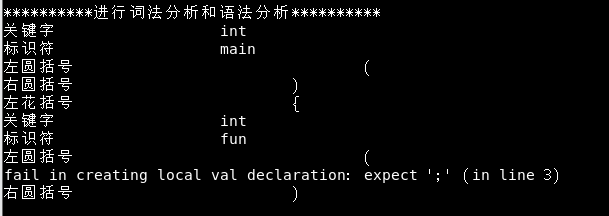


图4.18异常处理1

**测试样例2：**

**输入代码：**

int main()

{

int a = 1;

int b = 3;

for (int i = 0;i < a;i+=1){

123 = 1

}

}

运行结果：

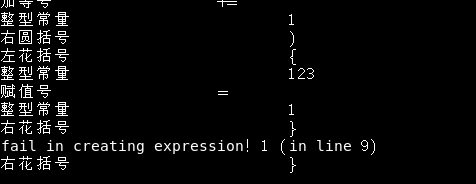


图4.19异常处理2

**测试样例3：**

**输入代码：**

#include <stdlib.h>

#include "test.h"

#define DEF def

void funcDeclare(int i, float f);

void funcVoid();

double globVar = 1, globarr[9];

int main()

{

int i\_1 = 123ul, i\_2 = 0x9b7c, i\_3 = 0125;

float f\_1 = .1l, f\_2 = .3e4;

double d = .314e3;

char c\_1 = '\0x8';

char s[0x19] = "May there be no course design in heaven";

if (c\_1 == 's' || i\_1 != i\_2)

{

funcDeclare(i, 0.1);

// error missing ()

if c\_1 == 9

{

globVar \*= 2;

}

}

else if (d == .27 || c\_1 > '9')

{

funcDeclare(i, 0.1);

}

else

{

i\_1 += 0;

}

while (i\_1 >= 0 && f\_1 < 1 + 3)

{

for (i\_1 = 0; i\_1 <= c\_1; i\_1 += 1)

{

i\_1 += 0;

continue;

}

break;

}

// i\_1=f\_1%98\*((8 - 2)+(9-i\_3)/4);

return i\_1;

}

**运行结果：**

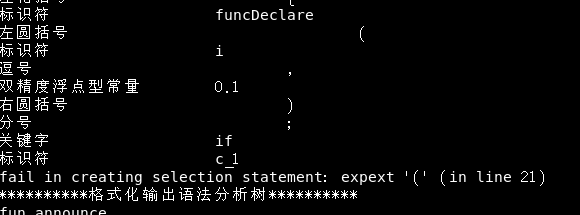


图4.19异常处理3

## 总结与展望

### 5.1全文总结

（1）题目的选择

数独题目的核心主要是算法的设计，对于这个题目，我本人并没有很好的思路，而且网上的大多资料针对的是数独而非双数独，针对设计双数独的过程中可能出现的种种问题，我并没有十足的把握，因此选择了个人觉得难度稍低一些的源程序处理，这个题目的核心主要是对数据结构课程中所学的栈，队列，树等数据结构的具体应用，而且对于其中可能出现的问题有着更高的把握来解决，从而坚定了我对题目的选择。

（2）总体设计

在词法分析部分还算比较正常和简单，只需要识别单词和各种符号即可，但是在语法分析构造语法分析树的过程中，遇到了很多的障碍，这里一方面要考虑语法分析树结构的改变，另一方面要考虑到各种函数功能的关联问题，与梯度下降算法的理解有很大的关联，这需要很强的逻辑思维，耗时时间最长没花费精力也最多。

（3）文件保存

程序最后一部分时根据生成的语法分析树保存文件，这里有种让我从头开始的感觉，因为在生成语法分析树的过程中，并没有将一些特殊的符号写在里面，在节点遍历的过程中并不能找到这些符号，所以又要用到词法分析时保存过的单词的原值，所以各种循环和判断比较繁琐。

### 5.2工作展望

在今后的研究中，围绕着如下几个方面开展工作

（1）对程序的优化，我目前所编写的这个程序有很大的局限性，对一些多义词只能识别出一种意思，比如位运算符号(只能识别逻辑运算)等，而且代码里面也有很多的bug，不能够适用于所有的代码，一些错误会时常出现，而且我是根据任务书先进行编写的测试代码，这样有一些对照的作用，然后根据测试的代码修改完善程序，所以对测试代码改变太大会出现很多无法识别的状况。

（2）程序中出现了很多不必要的循环，下一步要实现的是减少这个程序的时间复杂度和空间复杂度，正如我所了解到的，梯度下降算法是一些编译器内部的核心算法，如果对程序加以改进和完善，并且缩短识别需要的时间，那么就可以构建出来一个简单的智能编译器了。

## 6.体会

### 6.1总结

本次课程设计是我第一次系统的进行一个比较复杂的程序的编写。当然作为一个正在学习的程序员，这可能仅仅是个开始，但是代码的确是我编写的最长的代码了，可能可以优化的地方还很多。

我最有收获的一点就是对C语言以及C语言的数据结构有了更深的体会，并且学习并且了解熟悉了梯度下降算法的主要思想，使我的思维变得更加的严谨，而且在本次课程设计中增加了自己的动手能力，锻炼了构造一个项目的框架方法，能够很好的给出系统的框架，并且能够按照程序流程进行程序的编写。能够运用递归下降的方法进行分析问题，在词法分析中掌握了状态机的转变，同时能够熟练运用状态机进行字符的匹配。在语法分析中，学习到了怎么样通过文法构建代码，以及根据文法编写代码带来的相关问题，如左递归，节点的定义等。

最后，在完成代码的时刻，心里有一种成就感，不枉这么多天的废寝忘食，并且最后复盘一下，自己也确实发现了很多的不足，时间浪费太严重，而且任务完成比较拖沓，通常一个任务要2天才能完成，这也是时间给的过于充分的副作用吧。并且一边写代码一边发现问题并且解决问题的过程还是很值得人来回忆的，希望能够通过这次经历，为以后编写大型项目打下基础。

### 6.2特色

1. 通过构造哈希表完成了对单词的存储以及提取。
2. 宏定义结点的增加和注释的判别，程序能够判别行注释和段注释。
3. 使测试代码每一行的语句都生成一棵子树，这样能够统一树的结构，并且能给编写程序的过程带来很大的方便。
4. 保存文件时的单词原值遍历和语法树先根遍历的共同使用。
5. 使用CMake对项目进行编译，对于跨平台有更高的支持性。

### 6.3不足

1. 循环结构体运用的过多，使时间复杂度上升。
2. 静态分配数组来存储测试代码的数据，空间复杂度上升。
3. 不能识别很多复杂语句生成语法分析树。
4. 程序还存在一些bug不时弹出，还未找到原因进行改正。

## 参考文献

[1] 王生原，董渊，张素琴，吕映芝等. 编译原理（第3版）. 北京：清华大学出版社. 前4章

[2] 严蔚敏等. 数据结构(C语言版). 清华大学出版社

[3] 百度百科. 抽象语法树

## 附录

### 附录1 AST.cpp：

#include "AST.h"

/\*\*

 \* @description: 释放抽象语法树全部空间

 \* @param {p\_treeNode} root 抽象语法树根结点

 \* @return {p\_treeNode} 空指针

 \* @call: destoryAST

 \*/

p\_treeNode destoryAST(p\_treeNode root) {

  if (!root)

    return NULL;

  if (root->firstChild)

    destoryAST(root->firstChild);

  if (root->nextBro)

    destoryAST(root->nextBro);

  free(root);

  return NULL;

}

/\*\*

 \* @description: 初始化树结点

 \* @param {p\_treeNode} newTree 树结点指针

 \* @param {int} token -1:该节点为语法单元 其他:该节点token\_kind值

 \* @param {char} \*grammar\_type 语法单元类型(结点为token时可传入空字符串)

 \* @return {status} OK：初始化成功 ERROR：初始化失败

 \* @call: (none)

 \* @called\_by: insertChild

 \*/

status iniTreeNode(p\_treeNode &newTree, int token, char \*grammar\_type) {

  if (!newTree) {

    newTree = destoryAST(newTree);

  }

  newTree = (p\_treeNode)malloc(sizeof(treeNode));

  if (!newTree)

    return ERROR;

  newTree->firstChild = NULL;

  newTree->nextBro = NULL;

  newTree->node\_type = token;

  strcpy(newTree->self\_value, grammar\_type);

  return OK;

}

/\*\*

 \* @description: 插入子结点

 \* @param {p\_treeNode} parent 父结点指针

 \* @param {int} token: -1:该节点为语法单元 其他:该节点token\_kind值

 \* @param {char} \*childGrammarType

 \* 子结点语法单元类型(结点为token时可传入空字符串)

 \* @return {status} OK：插入成功 ERROR：插入失败

 \* @call: iniTreeNode

 \*/

p\_treeNode insertChild(p\_treeNode &parent, int token, char \*childGrammarType) {

  p\_treeNode newChild = (p\_treeNode)malloc(sizeof(treeNode));

  // int childTabs = parent->tabs;

  if (!iniTreeNode(newChild, token, childGrammarType))

    return NULL;

  if (!parent->firstChild) { //当前结点无子结点

    parent->firstChild = newChild;

  } else { //当前结点有子结点

    p\_treeNode pointer = parent->firstChild;

    for (; pointer->nextBro; pointer = pointer->nextBro)

      ; //前往最后一个子结点

    pointer->nextBro = newChild;

  }

  return newChild;

}

### 附录2 AST.h：

#ifndef COURSE\_DESIGN\_AST\_H

#define COURSE\_DESIGN\_AST\_H

#include "lexer.h"

#include <cstdio>

#include <cstring>

#include <mm\_malloc.h>

#define OK 1

#define ERROR 0

typedef int status;

typedef struct treeNode {

  int node\_type; //-1:该节点为语法单元 其他:该节点token\_kind值

  char self\_value[50];  //语句类型或token自身值

  treeNode \*nextBro;    //下一个兄弟结点

  treeNode \*firstChild; //第一个子结点

} \* p\_treeNode;

//抽象语法树函数

p\_treeNode destoryAST(p\_treeNode root);

status iniTreeNode(p\_treeNode &newTree, int token, char \*grammar\_type);

p\_treeNode insertChild(p\_treeNode &parent, int token, char \*childGrammarType);

#endif // COURSE\_DESIGN\_AST\_H

### 附录3 lexer.cpp：

#include "lexer.h"

/\*\*

 \* @description:将关键字和枚举类型常量值编成哈希表，用于查找对应

 \* @rule: 第一个字符的ASCII码对10求模，值对应数组第一个分量下标

 \* 每行最后设置哨兵none，枚举类型值对应为IDENT

 \*/

KeyWord\_name\_value KeyWordHashTable[10][10] = {

    // ascii码 mod 10 ==0:d,n,x

    {{"double", DOUBLE}, {"do", DO}, {"default", DEFAULT}, {"none", IDENT}},

    // ascii码 mod 10 ==1:e,o,y

    {{"enum", ENUM}, {"extern", EXTERN}, {"else", ELSE}, {"none", IDENT}},

    // ascii码 mod 10 ==2:f,p,z

    {{"float", FLOAT}, {"for", FOR}, {"none", IDENT}},

    // ascii码 mod 10 ==3:g,q

    {{"goto", GOTO}, {"none", IDENT}},

    // ascii码 mod 10 ==4:h,r

    {{"register", REGISTER}, {"return", RETURN}, {"none", IDENT}},

    // ascii码 mod 10 ==5:i,s

    {{"short", SHORT},

     {"int", INT},

     {"struct", STRUCT},

     {"sighed", SIGNED},

     {"static", STATIC},

     {"if", IF},

     {"switch", SWITCH},

     {"sizeof", SIZEOF},

     {"none", IDENT}},

    // ascii码 mod 10 ==6:j,t

    {{"typedef", TYPEDEF}, {"none", IDENT}},

    // ascii码 mod 10 ==7:a,k,u

    {{"auto", AUTO}, {"union", UNION}, {"unsigned", UNSIGNED}, {"none", IDENT}},

    // ascii码 mod 10 ==8:b,l,v

    {{"long", LONG},

     {"volatile", VOLATILE},

     {"void", VOID},

     {"break", BREAK},

     {"none", IDENT}},

    // ascii码 mod 10 ==9:c,m,w

    {{"char", CHAR},

     {"const", CONST},

     {"case", CASE},

     {"while", WHILE},

     {"continue", CONTINUE},

     {"none", IDENT}},

}; // end of KeyWordHashTable

/\*\*

 \* @description: 检查字符串是否为关键字

 \* @param {string}字符串

 \* @return {enum token\_kind}对应枚举类型常量值

 \* @call: (none)

 \* @called\_by: get\_token

 \*/

int isKeyWord(char str[]) {

  int line = str[0] % 10;

  for (int column = 0;; column++) {

    if (strcmp(str, KeyWordHashTable[line][column].name) == 0 ||

        strcmp("none", KeyWordHashTable[line][column].name) == 0)

      return KeyWordHashTable[line][column].value;

  }

} // end of isKeyWord

/\*\*

 \* @description: 判断字符是否为数字

 \* @param {char} c

 \* @return {bool}是数字返回true，否则返回false

 \* @call: (none)

 \* @called\_by: get\_token

 \*/

bool isNum(char c) {

  if ((c >= '0' && c <= '9') || c == '.' || c == '-')

    return true;

  else

    return false;

} // end of isNum

/\*\*

 \* @description: 判断字符是否为字母

 \* @param {char} c

 \* @return {bool}是字母返回true，不是返回false

 \* @call: (none)

 \* @called\_by:get\_token

 \*/

bool isLetter(char c) {

  if ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') || c == '\_')

    return true;

  else

    return false;

} // end of isLetter

/\*\*

 \* @description: 对源程序进行词法分析,并打印种类码和自身值

 \* @param {FILE} \*p 文件指针

 \* @param {int} &line\_num 当前行数，用于报错

 \* @param {char} token\_text 单词的自身值

 \* @return {int} 单词的种类码(包括报错符)，或EOF

 \* @call: enum token\_kind isKeyWord;

 \*        bool isNum;

 \*        bool isLetter

 \* @called\_by: functions in file "parser.cpp"

 \*/

token\_info get\_token(FILE \*p, int &line\_num, char token\_text[]) {

  char c;

  token\_info this\_token;

  token\_text[0] = '\0'; //重置token\_text变量

  while ((c = fgetc(p)) == ' ' || c == '\t' || c == '\n')

  //过滤空白符

  {

    if (c == '\n') //遇到换行符，行数自增

      line\_num += 1;

  }

  // printf("\*\*\*\*\*ch:%c\n", c);

  if (isLetter(c))

  //以字母开头，应该为关键字或标识符

  {

    do { //读取当前标识符串并存入字符串变量token\_text中

      token\_text[strlen(token\_text) + 1] = '\0';

      token\_text[strlen(token\_text)] = c;

    } while (isLetter(c = fgetc(p)) || isNum(c));

    ungetc(c, p); //将多读的字符退回缓冲区

    if (isKeyWord(token\_text) != IDENT) {

      printf("关键字\t\t\t%s\n", token\_text);

    } else {

      printf("标识符\t\t\t%s\n", token\_text);

    }

    this\_token.line = line\_num;

    this\_token.token\_kind = isKeyWord(token\_text);

    strcpy(this\_token.token\_text, token\_text);

    return this\_token;

  } // end of if(isLetter)

  if (isNum(c))

  //以数字开头，应该是整型、浮点型常量

  {

    bool isFloat = false;

    bool isHex = false; //十六进制

    if (c == '.')

      isFloat = true;

    do {

      token\_text[strlen(token\_text) + 1] = '\0';

      token\_text[strlen(token\_text)] = c;

      if (c == '.' || (!isHex && (c == 'e' || c == 'E')))

        isFloat = true;

      if (c == 'x' || c == 'X')

        isHex = true;

      if ((c >= 'a' && c <= 'e') || (c >= 'A' && c <= 'E')) {

        if (!isHex)

          break;

      }

    } while (isNum(c = fgetc(p)) || (c >= 'a' && c <= 'e') ||

             (c >= 'A' && c <= 'E') || c == 'x' || c == 'X');

    ungetc(c, p);

    c = fgetc(p);

    /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*处理常量后缀\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

    if (c == 'u' || c == 'U')

    //可能为UNSIGNED\_CONST、UNSIGNED\_LONG\_CONST

    {

      //浮点型变量不能有'u'、'U'后缀

      if (isFloat) {

        printf("错误的浮点型常量\t%s(in line %d)\n", token\_text, line\_num);

        this\_token.line = line\_num;

        this\_token.token\_kind = ERROR\_TOKEN;

        strcpy(this\_token.token\_text, token\_text);

        return this\_token;

      }

      if ((c = fgetc(p)) == 'l' || c == 'L') {

        printf("无符号长整型常量\t%s\n", token\_text);

        this\_token.line = line\_num;

        this\_token.token\_kind = UNSIGNED\_LONG\_CONST;

        strcpy(this\_token.token\_text, token\_text);

        return this\_token;

      } else {

        ungetc(c, p);

        printf("无符号整型常量\t%s\n", token\_text);

        this\_token.line = line\_num;

        this\_token.token\_kind = UNSIGNED\_CONST;

        strcpy(this\_token.token\_text, token\_text);

        return this\_token;

      }

    } // end of if

    else if (c == 'l' || c == 'L')

    //可能为LONG\_CONST、UNSIGNED\_LONG\_CONST、LONG\_DOUBLE\_CONST

    {

      if ((c = fgetc(p)) == 'u' || c == 'U') {

        //浮点型变量不能有'u'、'U'后缀

        if (isFloat) {

          printf("错误的浮点型常量\t%s(in line %d)\n", token\_text, line\_num);

          this\_token.line = line\_num;

          this\_token.token\_kind = ERROR\_TOKEN;

          strcpy(this\_token.token\_text, token\_text);

          return this\_token;

        } else {

          printf("无符号整型常量\t%s\n", token\_text);

          this\_token.line = line\_num;

          this\_token.token\_kind = UNSIGNED\_LONG\_CONST;

          strcpy(this\_token.token\_text, token\_text);

          return this\_token;

        }

      } // end of if

      else {

        ungetc(c, p);

        if (isFloat) {

          printf("长浮点型常量\t\t%s\n", token\_text);

          this\_token.line = line\_num;

          this\_token.token\_kind = LONG\_DOUBLE\_CONST;

          strcpy(this\_token.token\_text, token\_text);

          return this\_token;

        } else {

          printf("长整型常量\t\t%s\n", token\_text);

          this\_token.line = line\_num;

          this\_token.token\_kind = LONG\_CONST;

          strcpy(this\_token.token\_text, token\_text);

          return this\_token;

        }

      } // end of else

    }   // end of if (c == 'l' || c == 'L')

    else if (c == 'f' || c == 'F')

    //为单精度浮点型常量

    {

      printf("浮点型常量\t\t%s\n", token\_text);

      this\_token.line = line\_num;

      this\_token.token\_kind = FLOAT\_CONST;

      strcpy(this\_token.token\_text, token\_text);

      return this\_token;

    } // end of if

    //若常量无后缀

    else {

      ungetc(c, p);

      if (isFloat) {

        printf("双精度浮点型常量\t%s\n", token\_text);

        this\_token.line = line\_num;

        this\_token.token\_kind = DOUBLE\_CONST;

        strcpy(this\_token.token\_text, token\_text);

        return this\_token;

      } else {

        printf("整型常量\t\t\t%s\n", token\_text);

        this\_token.line = line\_num;

        this\_token.token\_kind = INT\_CONST;

        strcpy(this\_token.token\_text, token\_text);

        return this\_token;

      }

    } // end of else

  }   // end of if(isNum)

  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*判断特殊符号\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

  token\_text[strlen(token\_text) + 1] = '\0';

  token\_text[strlen(token\_text)] = c;

  switch (c) {

  case '#':

    token\_text[strlen(token\_text) + 1] = '\0';

    token\_text[strlen(token\_text)] = c;

    printf("未知的标识符\t\t%s(in line %d)\n", token\_text, line\_num);

    this\_token.line = line\_num;

    this\_token.token\_kind = ERROR\_TOKEN;

    strcpy(this\_token.token\_text, token\_text);

    return this\_token;

  case '+':

    if ((c = fgetc(p)) == '=') {

      token\_text[strlen(token\_text) + 1] = '\0';

      token\_text[strlen(token\_text)] = c;

      printf("加等号\t\t\t%s\n", token\_text);

      this\_token.line = line\_num;

      this\_token.token\_kind = PLUS\_EQ;

      strcpy(this\_token.token\_text, token\_text);

      return this\_token;

    } else {

      ungetc(c, p);

      printf("加号\t\t\t%s\n", token\_text);

      this\_token.line = line\_num;

      this\_token.token\_kind = PLUS;

      strcpy(this\_token.token\_text, token\_text);

      return this\_token;

    }

    break;

  case '-':

    if ((c = fgetc(p)) == '=') {

      token\_text[strlen(token\_text) + 1] = '\0';

      token\_text[strlen(token\_text)] = c;

      printf("减等号\t\t\t%s\n", token\_text);

      this\_token.line = line\_num;

      this\_token.token\_kind = MINUS\_EQ;

      strcpy(this\_token.token\_text, token\_text);

      return this\_token;

    } else {

      ungetc(c, p);

      printf("减号\t\t\t%s\n", token\_text);

      this\_token.line = line\_num;

      this\_token.token\_kind = MINUS;

      strcpy(this\_token.token\_text, token\_text);

      return this\_token;

    }

    break;

  case '\*':

    if ((c = fgetc(p)) == '=') {

      token\_text[strlen(token\_text) + 1] = '\0';

      token\_text[strlen(token\_text)] = c;

      printf("乘等号\t\t\t%s\n", token\_text);

      this\_token.line = line\_num;

      this\_token.token\_kind = MULTIPLY\_EQ;

      strcpy(this\_token.token\_text, token\_text);

      return this\_token;

    } else {

      ungetc(c, p);

      printf("乘号\t\t\t%s\n", token\_text);

      this\_token.line = line\_num;

      this\_token.token\_kind = MULTIPLY;

      strcpy(this\_token.token\_text, token\_text);

      return this\_token;

    }

    break;

  case '/':

    //不考虑转义

    if ((c = fgetc(p)) == '=') {

      token\_text[strlen(token\_text) + 1] = '\0';

      token\_text[strlen(token\_text)] = c;

      printf("除等号\t\t\t%s\n", token\_text);

      this\_token.line = line\_num;

      this\_token.token\_kind = DIVIDE\_EQ;

      strcpy(this\_token.token\_text, token\_text);

      return this\_token;

    } else {

      ungetc(c, p);

      printf("除等号\t\t\t%s\n", token\_text);

      this\_token.line = line\_num;

      this\_token.token\_kind = DIVIDE;

      strcpy(this\_token.token\_text, token\_text);

      return this\_token;

    }

  case '%':

    //未考虑类似：printf("%d",number)的情形

    if ((c = fgetc(p)) == '=') {

      token\_text[strlen(token\_text) + 1] = '\0';

      token\_text[strlen(token\_text)] = c;

      printf("模等号\t\t\t%s\n", token\_text);

      this\_token.line = line\_num;

      this\_token.token\_kind = MOD\_EQ;

      strcpy(this\_token.token\_text, token\_text);

      return this\_token;

    } else {

      ungetc(c, p);

      printf("求模号\t\t\t%s\n", token\_text);

      this\_token.line = line\_num;

      this\_token.token\_kind = MOD;

      strcpy(this\_token.token\_text, token\_text);

      return this\_token;

    }

    break;

  case '=':

    if ((c = fgetc(p)) == '=') {

      token\_text[strlen(token\_text) + 1] = '\0';

      token\_text[strlen(token\_text)] = c;

      printf("相等号\t\t\t%s\n", token\_text);

      this\_token.line = line\_num;

      this\_token.token\_kind = EQUAL;

      strcpy(this\_token.token\_text, token\_text);

      return this\_token;

    } else {

      ungetc(c, p);

      printf("赋值号\t\t\t%s\n", token\_text);

      this\_token.line = line\_num;

      this\_token.token\_kind = ASSIGN;

      strcpy(this\_token.token\_text, token\_text);

      return this\_token;

    }

    break;

  case '!':

    //未考虑逻辑非运算

    if ((c = fgetc(p)) == '=') {

      token\_text[strlen(token\_text) + 1] = '\0';

      token\_text[strlen(token\_text)] = c;

      printf("不等号\t\t\t%s\n", token\_text);

      this\_token.line = line\_num;

      this\_token.token\_kind = UNEQUAL;

      strcpy(this\_token.token\_text, token\_text);

      return this\_token;

    } else {

      ungetc(c, p);

      printf("错误的标识符\t\t%s(in line %d)\n", token\_text, line\_num);

      this\_token.line = line\_num;

      this\_token.token\_kind = ERROR\_TOKEN;

      strcpy(this\_token.token\_text, token\_text);

      return this\_token;

    }

    break;

  case '<':

    if ((c = fgetc(p)) == '=') {

      token\_text[strlen(token\_text) + 1] = '\0';

      token\_text[strlen(token\_text)] = c;

      printf("小于等于号\t\t%s\n", token\_text);

      this\_token.line = line\_num;

      this\_token.token\_kind = LESS\_OR\_EQUAL;

      strcpy(this\_token.token\_text, token\_text);

      return this\_token;

    } else {

      ungetc(c, p);

      printf("小于号\t\t\t%s\n", token\_text);

      this\_token.line = line\_num;

      this\_token.token\_kind = LESS;

      strcpy(this\_token.token\_text, token\_text);

      return this\_token;

    }

    break;

  case '>':

    if ((c = fgetc(p)) == '=') {

      token\_text[strlen(token\_text) + 1] = '\0';

      token\_text[strlen(token\_text)] = c;

      printf("大于等于号\t\t%s\n", token\_text);

      this\_token.line = line\_num;

      this\_token.token\_kind = MORE\_OR\_EQUAL;

      strcpy(this\_token.token\_text, token\_text);

      return this\_token;

    } else {

      ungetc(c, p);

      printf("大于号\t\t\t%s\n", token\_text);

      this\_token.line = line\_num;

      this\_token.token\_kind = MORE;

      strcpy(this\_token.token\_text, token\_text);

      return this\_token;

    }

    break;

  case '&':

    //未考虑按位与运算

    if ((c = fgetc(p)) == '&') {

      token\_text[strlen(token\_text) + 1] = '\0';

      token\_text[strlen(token\_text)] = c;

      printf("逻辑与\t\t\t%s\n", token\_text);

      this\_token.line = line\_num;

      this\_token.token\_kind = AND;

      strcpy(this\_token.token\_text, token\_text);

      return this\_token;

    } else {

      ungetc(c, p);

      printf("错误的标识符\t\t%s(in line %d)\n", token\_text, line\_num);

      this\_token.line = line\_num;

      this\_token.token\_kind = ERROR\_TOKEN;

      strcpy(this\_token.token\_text, token\_text);

      return this\_token;

    }

    break;

  case '|':

    //未考虑按位或运算

    if ((c = fgetc(p)) == '|') {

      token\_text[strlen(token\_text) + 1] = '\0';

      token\_text[strlen(token\_text)] = c;

      printf("逻辑或\t\t\t%s\n", token\_text);

      this\_token.line = line\_num;

      this\_token.token\_kind = OR;

      strcpy(this\_token.token\_text, token\_text);

      return this\_token;

    } else {

      ungetc(c, p);

      printf("错误的标识符\t\t%s(in line %d)\n", token\_text, line\_num);

      this\_token.line = line\_num;

      this\_token.token\_kind = ERROR\_TOKEN;

      strcpy(this\_token.token\_text, token\_text);

      return this\_token;

    }

    break;

  case ',':

    printf("逗号\t\t\t\t%s\n", token\_text);

    this\_token.line = line\_num;

    this\_token.token\_kind = COMMA;

    strcpy(this\_token.token\_text, token\_text);

    return this\_token;

    break;

  case ';':

    printf("分号\t\t\t\t%s\n", token\_text);

    this\_token.line = line\_num;

    this\_token.token\_kind = SEMICOLON;

    strcpy(this\_token.token\_text, token\_text);

    return this\_token;

    break;

  case '(':

    printf("左圆括号\t\t\t\t%s\n", token\_text);

    this\_token.line = line\_num;

    this\_token.token\_kind = LPARENTHESE;

    strcpy(this\_token.token\_text, token\_text);

    return this\_token;

    break;

  case ')':

    printf("右圆括号\t\t\t%s\n", token\_text);

    this\_token.line = line\_num;

    this\_token.token\_kind = RPARENTHESE;

    strcpy(this\_token.token\_text, token\_text);

    return this\_token;

    break;

  case '[':

    printf("左方括号\t\t\t%s\n", token\_text);

    this\_token.line = line\_num;

    this\_token.token\_kind = LSUBSCRIPT;

    strcpy(this\_token.token\_text, token\_text);

    return this\_token;

    break;

  case ']':

    printf("右方括号\t\t\t%s\n", token\_text);

    this\_token.line = line\_num;

    this\_token.token\_kind = RSUBSCRIPT;

    strcpy(this\_token.token\_text, token\_text);

    return this\_token;

    break;

  case '{':

    printf("左花括号\t\t\t%s\n", token\_text);

    this\_token.line = line\_num;

    this\_token.token\_kind = LCURLYBRACE;

    strcpy(this\_token.token\_text, token\_text);

    return this\_token;

    break;

  case '}':

    printf("右花括号\t\t\t%s\n", token\_text);

    this\_token.line = line\_num;

    this\_token.token\_kind = RCURLYBRACE;

    strcpy(this\_token.token\_text, token\_text);

    return this\_token;

    break;

  case '\'':

    do {

      c = fgetc(p);

      token\_text[strlen(token\_text) + 1] = '\0';

      token\_text[strlen(token\_text)] = c;

    } while (c != '\'');

    if (strlen(token\_text) == 3 ||

        (strlen(token\_text) == 4 && token\_text[1] == '\\')) {

      printf("字符常量\t\t\t%s\n", token\_text);

      this\_token.line = line\_num;

      this\_token.token\_kind = CHAR\_CONST;

      strcpy(this\_token.token\_text, token\_text);

      return this\_token;

    } else {

      printf("错误的字符常量\t\t%s(in line %d)\n", token\_text, line\_num);

      this\_token.line = line\_num;

      this\_token.token\_kind = ERROR\_TOKEN;

      strcpy(this\_token.token\_text, token\_text);

      return this\_token;

    }

    break;

  case '\"':

    do {

      c = fgetc(p);

      if (c == '\n')

        line\_num++;

      token\_text[strlen(token\_text) + 1] = '\0';

      token\_text[strlen(token\_text)] = c;

    } while (c != '\"');

    printf("字符串常量\t\t%s\n", token\_text);

    this\_token.line = line\_num;

    this\_token.token\_kind = STRING\_CONST;

    strcpy(this\_token.token\_text, token\_text);

    return this\_token;

  default:

    if (feof(p)) {

      printf("文件结束\n");

      this\_token.line = line\_num;

      this\_token.token\_kind = EOF;

      strcpy(this\_token.token\_text, token\_text);

      return this\_token;

    } else {

      printf("未知的标识符\t\t%s (in line %d)\n", token\_text, line\_num);

      this\_token.line = line\_num;

      this\_token.token\_kind = ERROR\_TOKEN;

      strcpy(this\_token.token\_text, token\_text);

      return this\_token; //报错符号

    }

    break;

  } // end of switch(c)（特殊符号判断）

} // end of fun: int get\_token;

### 附录4 lexer.h：

#ifndef COURSE\_DESIGN\_LEXER\_H

#define COURSE\_DESIGN\_LEXER\_H

#include <cstdio>

#include <cstring>

/\*\*

 \* 定义单词种类

 \* 关键字:  AUTO-UNSIGNED

 \* 类型说明符:  CHAR-UNSIGNED

 \* 运算符：ASSIGN-MOD

 \* 单目算术运算符:  PLUS-MOD

 \* 关系运算符:  EQUAL-MORE\_OR\_EQUAL

 \* 逻辑运算符（双目）:  AND、OR

 \* 赋值运算符:  ASSIGN-MODE\_EQ

 \* 常量:  INT\_CONST-STRING\_CONST

 \* 运算符顺序中越靠后优先级越高

 \*/

enum token\_kind

{

    /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*标识符\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

    IDENT,

    /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*保留字\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

    AUTO,

    BREAK,

    CASE,

    CONTINUE,

    DEFAULT,

    DO,

    RETURN,

    SIZEOF,

    SWITCH,

    TYPEDEF,

    VOID,

    VOLATILE,

    WHILE,

    ELSE,

    EXTERN,

    FOR,

    GOTO,

    IF,

    CONST,

    STATIC,

    REGISTER,

    CHAR,

    DOUBLE,

    ENUM,

    FLOAT,

    INT,

    LONG,

    SIGNED,

    SHORT,

    STRUCT,

    UNION,

    UNSIGNED,

    /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*定界符\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

    //起止符号

    BEGIN\_END, //#

    COMMA,           // ,

    SEMICOLON,       // ;

    LPARENTHESE,     // (

    RPARENTHESE,     // )

    LSUBSCRIPT,      // [

    RSUBSCRIPT,      // ]

    LCURLYBRACE,     // {

    RCURLYBRACE,     // }

    SINGLE\_QUO\_MARK, // '

    DOUBLE\_QUO\_MARK, // "

    /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*运算符\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

    //赋值运算符

    ASSIGN,      // =

    PLUS\_EQ,     // +=

    MINUS\_EQ,    // -=

    MULTIPLY\_EQ, // \*=

    DIVIDE\_EQ,   // /=

    MOD\_EQ,      // %=

    //逻辑运算符（双目）

    AND, // &&

    OR,  // ||

    //关系运算符

    EQUAL,         // ==

    UNEQUAL,       // !=

    LESS,          // <

    MORE,          // >

    LESS\_OR\_EQUAL, // <=

    MORE\_OR\_EQUAL, // >=

    //单目算术运算符

    PLUS,     // +

    MINUS,    // -

    MULTIPLY, // \*

    DIVIDE,   // /

    MOD,      // %

    /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

     \* 暂不支持:

    //算术运算符（单目）

    PLUSPLUS,   // ++

    MINUSMINUS, // --

    NON,        // !单目逻辑运算符

    //位运算符

    BIT\_AND,   // 位运算与&

    BIT\_OR,    // |

    BIT\_XOR,   // ^

    BIT\_NON,   // ~

    BIT\_LEFT,  // <<

    BIT\_RIGHT, // >>

    //赋值运算符（位运算）

    BIT\_LEFT\_EQ,  // <<=

    BIT\_RIGHT\_EQ, // >>=

    BIT\_AND\_EQ,   // &=

    BIT\_XOR\_EQ,   // ^=

    BIT\_OR\_EQ,    // |=

    GET\_ADDRESS,  // 取地址符&

    POINTER,      // 取变量符\*

    CONDITION,    // 条件表达式？：

    \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

    /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*常量\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

    INT\_CONST,

    UNSIGNED\_CONST,

    LONG\_CONST,

    UNSIGNED\_LONG\_CONST,

    FLOAT\_CONST,

    DOUBLE\_CONST,

    LONG\_DOUBLE\_CONST,

    CHAR\_CONST,

    STRING\_CONST,

    /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*预处理标识符\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

    DEFINE,

    INCLUDE,

    /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*报错符\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

    ERROR\_TOKEN,

}; //end of enum token\_kind

//关键字字符串值与枚举常量值对应结构（用于构建哈希表）

typedef struct KeyWord\_name\_value

{

    char name[10];

    int value;

} KeyWord\_name\_value;

struct token\_info

{

    int token\_kind;      //种类码

    int line;            //行数

    char token\_text[50]; //自身值

};

//词法分析函数

int isKeyWord(char str[]);

bool isNum(char c);

bool isLetter(char c);

token\_info get\_token(FILE \*p, int &line\_num, char token\_text[]);

#endif //COURSE\_DESIGN\_LEXER\_H

### 附录5 main.cpp：

#pragma once

#include "Traverse.h"

#include "lexer.h"

#include "parsing.h"

#include "pre\_process.h"

#include "print\_file.h"

int main() {

  char source\_file\_name[100], mid\_file\_name[110];

  printf("源程序文件名称(不含文件后缀名.c)：\n");

  scanf("%s", source\_file\_name);

  sprintf(mid\_file\_name, "mid\_%s.c", source\_file\_name);

  sprintf(source\_file\_name, "%s.c", source\_file\_name);

  if (!pre\_process(source\_file\_name, mid\_file\_name))

    printf("文件打开失败！\n");

  FILE \*mid\_file = fopen(mid\_file\_name, "r");

  int line = 1;

  char token\_text[50];

  Queue Q;

  iniQueue(Q);

  printf("\n\*\*\*\*\*\*\*\*\*\*进行词法分析和语法分析\*\*\*\*\*\*\*\*\*\*\n");

  p\_treeNode program = Program(Q, mid\_file, line, token\_text);

  printf("\n\*\*\*\*\*\*\*\*\*\*格式化输出语法分析树\*\*\*\*\*\*\*\*\*\*\n");

  traverse(program, 0);

  printf("\n\*\*\*\*\*\*\*\*\*\*将代码格式化输出至文件%s\*\*\*\*\*\*\*\*\*\*\n", mid\_file\_name);

  fclose(mid\_file);

  mid\_file = fopen(mid\_file\_name, "w");

  if (!mid\_file) {

    printf("文件打开失败\n");

    return 0;

  }

  f\_traverse(mid\_file, program, 0, true);

  return 0;

}

### 附录6 parsing.cpp：

#include "parsing.h"

int error = 0;

/\*\*

 \* @description: 判断token是否为类型说明符

 \* @param {int} token\_type 标识符种类

 \* @return {bool} true/false

 \* @call\_by ex\_Declaration\_List; Param\_List; Statement; Local\_Val\_Declaration

 \*/

bool is\_Type\_Specifier(int token\_type) {

  if (token\_type >= CHAR && token\_type <= UNSIGNED)

    return true;

  else

    return false;

}

/\*\*

 \* @description: 判断token是否为运算符

 \* @param {int} token\_type 标识符种类

 \* @return {bool} true/false

 \* @called\_by Expression

 \*/

bool is\_Operator(int token\_type) {

  if (token\_type >= ASSIGN && token\_type <= MOD)

    return true;

  else

    return false;

}

/\*\*

 \* @description: 判断token是否为常量值

 \* @param {int} token\_type 标识符种类

 \* @return {bool} true/false

 \* @called\_by: Statement; Params\_Call

 \*/

bool is\_CONST\_value(int token\_type) {

  if (token\_type >= INT\_CONST && token\_type <= STRING\_CONST)

    return true;

  else

    return false;

}

/\*\*

 \* @description: 创建程序结点

 \* @param {Queue} &Q 读取的token队列

 \* @param {FILE} \*p 读取token的文件指针

 \* @param {int} &line token的行号

 \* @param {char} token\_text token自身值

 \* @return {p\_treeNode} 结点指针

 \* @call ex\_Declaration\_List

 \*/

p\_treeNode Program(Queue &Q, FILE \*p, int &line, char token\_text[]) {

  p\_treeNode program = NULL;

  char grammar\_type[] = "program";

  if (!iniTreeNode(program, -1, grammar\_type)) { //初始化根结点失败

    printf("%s\n", "unknown error in creating program!\n");

    return NULL;

  }

  char son\_grammar\_type[] = "ex declaration list";

  p\_treeNode ex\_declaration\_list = insertChild(program, -1, son\_grammar\_type);

  ex\_declaration\_list =

      ex\_Declaration\_List(Q, ex\_declaration\_list, p, line, token\_text);

  if (!ex\_declaration\_list) { //外部定义序列为空

    printf("fail in creating ex declaration list!(in line %d)\n", line);

    error = 1;

    destoryAST(ex\_declaration\_list);

    return program;

  }

  return program;

} // end of Program

/\*\*

 \* @description: 创建外部定义序列

 \* @param {Queue} &Q 读取的token队列

 \* @param {p\_treeNode} ex\_declaration\_list根结点

 \* @param {FILE} \*p 读取token的文件指针

 \* @param {int} &line token的行号

 \* @param {char} token\_text token自身值

 \* @return {p\_treeNode} 结点指针

 \* @call: is\_Type\_Specifier; Fun\_Declaration; ex\_Val\_Declaration

 \* @called\_by: program; ex\_Declaration\_List

 \*/

p\_treeNode ex\_Declaration\_List(Queue &Q, p\_treeNode ex\_declaration\_list,

                               FILE \*p, int &line, char token\_text[]) {

  token\_info w = get\_token(p, line, token\_text);

  if (w.token\_kind == EOF || feof(p))

    return NULL;

  if ((!is\_Type\_Specifier(w.token\_kind)) && w.token\_kind != VOID) {

    printf(

        "fail in creating ex declaration: expect type specifier!(in line %d)\n",

        line);

    error = 1;

    destoryAST(ex\_declaration\_list);

    return NULL;

  }

  enQueue(Q, w);

  w = get\_token(p, line, token\_text);

  if (w.token\_kind != IDENT) {

    printf("fail in creating ex declaration: expect IDENT!(in line %d)\n",

           line);

    error = 1;

    destoryAST(ex\_declaration\_list);

    return NULL;

  }

  enQueue(Q, w);

  w = get\_token(p, line, token\_text);

  switch (w.token\_kind) {

  case LPARENTHESE: {

    char grammar\_type[] = "fun declaration";

    p\_treeNode fun\_dec = insertChild(ex\_declaration\_list, -1, grammar\_type);

    fun\_dec = Fun\_Declaration(Q, fun\_dec, p, line, token\_text);

    break;

  }

  case COMMA:

  case SEMICOLON:

  case LSUBSCRIPT:

  case ASSIGN: {

    enQueue(Q, w);

    if (error)

      return ex\_declaration\_list;

    char grammar\_type1[] = "ex val declaration";

    p\_treeNode ex\_val\_dec = insertChild(ex\_declaration\_list, -1, grammar\_type1);

    ex\_val\_dec = ex\_Val\_Declaration(Q, ex\_val\_dec, p, line, token\_text);

    break;

  }

  default: {

    printf("fail in creating ex declaration: expect ; or param(in line %d)\n",

           line);

    error = 1;

    destoryAST(ex\_declaration\_list);

    return NULL;

    break;

  }

  } // end of switch

  if (error)

    return ex\_declaration\_list;

  char grammar\_type2[] = "ex declaration list";

  p\_treeNode son\_ex\_declaration\_list =

      insertChild(ex\_declaration\_list, -1, grammar\_type2);

  son\_ex\_declaration\_list =

      ex\_Declaration\_List(Q, son\_ex\_declaration\_list, p, line, token\_text);

  return ex\_declaration\_list;

} // end of ex\_Declaration\_List

/\*\*

 \* @description: 生成外部变量声明

 \* @param {Queue} &Q 读取的token队列

 \* @param {p\_treeNode} dec\_list根结点

 \* @param {FILE} \*p 读取token的文件指针

 \* @param {int} &line token的行号

 \* @param {char} token\_text token自身值

 \* @return {p\_treeNode} 结点指针

 \* @call: Type\_Specifier; Val\_List

 \* @called\_by: ex\_Declaration\_List

 \*/

p\_treeNode ex\_Val\_Declaration(Queue &Q, p\_treeNode val\_dec, FILE \*p, int &line,

                              char token\_text[]) {

  token\_info w;

  //将类型识别符放入结点

  deQueue(Q, w);

  char son0\_gt[] = "type specifier"; //第一个子结点的语法单元类型

  p\_treeNode type\_spe = insertChild(val\_dec, -1, son0\_gt);

  type\_spe = Type\_Specifier(w, type\_spe);

  if (error)

    return val\_dec;

  //创建标识符序列结点

  char son1\_gt[] = "val list"; //第二个子结点的语法单元类型

  p\_treeNode ex\_val\_list = insertChild(val\_dec, -1, son1\_gt);

  ex\_val\_list = Val\_List(Q, ex\_val\_list, p, line, token\_text);

  return val\_dec;

} // end of ex\_Val\_Declaration

/\*\*

 \* @description: 生成函数定义或函数声明结点

 \* @param {Queue} &Q 读取的token队列

 \* @param {p\_treeNode} fun\_dec根结点

 \* @param {FILE} \*p 读取token的文件指针

 \* @param {int} &line token的行号

 \* @param {char} token\_text token自身值

 \* @return {p\_treeNode} 结点指针

 \* @call: Type\_Specifier; Params; Compound\_Statement

 \* @called\_by: ex\_Declaration\_List

 \*/

p\_treeNode Fun\_Declaration(Queue &Q, p\_treeNode fun\_dec, FILE \*p, int &line,

                           char token\_text[]) {

  token\_info w;

  //生成函数返回类型结点

  deQueue(Q, w);

  char son0\_gt[] = "return type specifier";

  p\_treeNode type\_specifier = insertChild(fun\_dec, -1, son0\_gt);

  type\_specifier = Type\_Specifier(w, type\_specifier);

  if (error)

    return fun\_dec;

  //生成函数名结点

  deQueue(Q, w);

  char son1\_gt[] = "fun name";

  p\_treeNode fun\_name = insertChild(fun\_dec, IDENT, w.token\_text);

  if (error)

    return fun\_dec;

  //生成参数序列

  char son2\_gt[] = "params";

  p\_treeNode params = insertChild(fun\_dec, -1, son2\_gt);

  params = Params(Q, params, p, line, token\_text);

  if (error)

    return fun\_dec;

  //判断是否为函数声明，若不是则生成函数体

  w = get\_token(p, line, token\_text);

  if (w.token\_kind == SEMICOLON) {

    fun\_dec->self\_value[0] = '\0';

    strcpy(fun\_dec->self\_value, "fun announce");

    return fun\_dec;

  } else if (w.token\_kind == LCURLYBRACE) { //生成函数体复合语句结点

    // enQueue(Q, w);

    enQueue(Q, w);

    char son3\_gt[] = "fun compound statement";

    p\_treeNode fun\_body = insertChild(fun\_dec, -1, son3\_gt);

    fun\_body = Compound\_Statement(Q, fun\_body, p, line, token\_text);

    return fun\_dec;

  } else {

    printf("fail in creating fun declaration: expect \';\' (in line %d)\n",

           line);

    error = 1;

    destoryAST(fun\_dec);

    return NULL;

  }

} // end of Fun\_Declaration

/\*\*

 \* @description: 生成种类识别符结点

 \* @param {token\_info} w token信息结构

 \* @param {p\_treeNode} type\_specifier 种类识别符结点指针

 \* @return {p\_treeNode} 种类识别符结点指针

 \* @called\_by ex\_Val\_Declaration; Fun\_Declaration; Param\_List; Param; Statement;

 \*/

p\_treeNode Type\_Specifier(token\_info w, p\_treeNode type\_specifier) {

  p\_treeNode type = insertChild(type\_specifier, w.token\_kind, w.token\_text);

  return type\_specifier;

} // end of Type\_Specifier

/\*\*

 \* @description: 生成变量序列结点

 \* @param {Queue} &Q 读取的token队列

 \* @param {p\_treeNode} val\_list根结点

 \* @param {FILE} \*p 读取token的文件指针

 \* @param {int} &line token的行号

 \* @param {char} token\_text token自身值

 \* @return {p\_treeNode} 结点指针

 \* @call: Val\_List;

 \* @called\_by: ex\_val\_declaration; Val\_List

 \*/

p\_treeNode Val\_List(Queue &Q, p\_treeNode val\_list, FILE \*p, int &line,

                    char token\_text[]) {

  //进入该函数时栈中应有两个元素

  //栈中第一个元素应为变量标识符

  token\_info w;

  deQueue(Q, w);

  if (w.token\_kind != IDENT) {

    printf("fail in creating valid list: expect ident(in line %d)\n", line);

    error = 1;

    destoryAST(val\_list);

    return NULL;

  } // end of if

  if (Q\_get\_last(Q).token\_kind == LSUBSCRIPT) {

    //数组

    string list\_subscript;

    list\_subscript = list\_subscript + w.token\_text;

    deQueue(Q, w);

    while (w.token\_kind != RSUBSCRIPT) {

      list\_subscript = list\_subscript + w.token\_text;

      w = get\_token(p, line, token\_text);

    }

    list\_subscript = list\_subscript + w.token\_text;

    w = get\_token(p, line, token\_text);

    if (w.token\_kind == COMMA) {

      p\_treeNode val\_name =

          insertChild(val\_list, IDENT, (char \*)list\_subscript.data());

      w = get\_token(p, line, token\_text);

      enQueue(Q, w);

      w = get\_token(p, line, token\_text);

      enQueue(Q, w);

      char son1\_gt[] = "val list";

      p\_treeNode son\_val\_list = insertChild(val\_list, -1, son1\_gt);

      son\_val\_list = Val\_List(Q, son\_val\_list, p, line, token\_text);

      if (error)

        return val\_list;

    } // end of if

    else if (w.token\_kind == ASSIGN) {

      list\_subscript = list\_subscript + w.token\_text;

      w = get\_token(p, line, token\_text);

      list\_subscript = list\_subscript + w.token\_text;

      p\_treeNode val\_name =

          insertChild(val\_list, IDENT, (char \*)list\_subscript.data());

    } else if (w.token\_kind == SEMICOLON) {

      p\_treeNode val\_name =

          insertChild(val\_list, IDENT, (char \*)list\_subscript.data());

      return val\_list;

    }

  } // end of if

  else if (Q\_get\_last(Q).token\_kind == ASSIGN) {

    //定义时赋值

    string val\_assign;

    val\_assign = val\_assign + w.token\_text;

    deQueue(Q, w);

    val\_assign = val\_assign + w.token\_text;

    w = get\_token(p, line, token\_text);

    val\_assign = val\_assign + w.token\_text;

    p\_treeNode val\_name =

        insertChild(val\_list, IDENT, (char \*)val\_assign.data());

    w = get\_token(p, line, token\_text);

    if (w.token\_kind == COMMA) {

      w = get\_token(p, line, token\_text);

      enQueue(Q, w);

      w = get\_token(p, line, token\_text);

      enQueue(Q, w);

      char son1\_gt[] = "val list";

      p\_treeNode son\_val\_list = insertChild(val\_list, -1, son1\_gt);

      son\_val\_list = Val\_List(Q, son\_val\_list, p, line, token\_text);

      if (error)

        return val\_list;

    } // end of if

    else if (w.token\_kind == SEMICOLON) {

      return val\_list;

    }

  } // end of else if

  else if (Q\_get\_last(Q).token\_kind == COMMA) {

    //变量序列

    p\_treeNode val\_name = insertChild(val\_list, IDENT, w.token\_text);

    deQueue(Q, w);

    w = get\_token(p, line, token\_text);

    enQueue(Q, w);

    w = get\_token(p, line, token\_text);

    enQueue(Q, w);

    char son1\_gt[] = "val list";

    p\_treeNode son\_val\_list = insertChild(val\_list, -1, son1\_gt);

    son\_val\_list = Val\_List(Q, son\_val\_list, p, line, token\_text);

    if (error)

      return val\_list;

  }                                                 // end of else if

  else if (Q\_get\_last(Q).token\_kind == SEMICOLON) { //变量序列结束

    p\_treeNode val\_name = insertChild(val\_list, IDENT, w.token\_text);

    return val\_list;

  } else {

    printf("fail in creating valid list: expect \';\'(in line %d)\n", line);

    error = 1;

    destoryAST(val\_list);

    return NULL;

  }

  return val\_list;

} // end of Val\_List

/\*\*

 \* @description: 生成参数集结点

 \* @param {Queue} &Q 读取的token队列

 \* @param {p\_treeNode} param\_list结点

 \* @param {FILE} \*p 读取token的文件指针

 \* @param {int} &line token的行号

 \* @param {char} token\_text token自身值

 \* @return {p\_treeNode} 结点指针

 \* @call: Param\_List

 \* @called\_by: Fun\_Declaration; Fun\_call

 \*/

p\_treeNode Params(Queue &Q, p\_treeNode params, FILE \*p, int &line,

                  char token\_text[]) {

  token\_info w = get\_token(p, line, token\_text);

  if (w.token\_kind == RPARENTHESE || w.token\_kind == VOID) { //函数无参数

    return params;

  } else {

    enQueue(Q, w);

    char son\_gt[] = "param list";

    p\_treeNode param\_list = insertChild(params, -1, son\_gt);

    param\_list = Param\_List(Q, param\_list, p, line, token\_text);

    return params;

  }

} // end of Params

/\*\*

 \* @description: 创建参数列表结点

 \* @param {Queue} &Q 读取的token队列

 \* @param {p\_treeNode} param\_list结点

 \* @param {FILE} \*p 读取token的文件指针

 \* @param {int} &line token的行号

 \* @param {char} token\_text token自身值

 \* @return {p\_treeNode} 结点指针

 \* @call: is\_Type\_Specifier; Param; Param\_List

 \* @called\_by: Params

 \*/

p\_treeNode Param\_List(Queue &Q, p\_treeNode param\_list, FILE \*p, int &line,

                      char token\_text[]) {

  token\_info w;

  deQueue(Q, w);

  if (w.token\_kind == RPARENTHESE) { //参数序列结束

    return NULL;

  }

  if (!is\_Type\_Specifier(w.token\_kind)) {

    printf("fail in creating param list: expect type specifier (in line %d)\n",

           line);

    error = 1;

    destoryAST(param\_list);

    return NULL;

  }

  enQueue(Q, w);

  w = get\_token(p, line, token\_text);

  if (w.token\_kind != IDENT) {

    printf("fail in creating param list: expect ident (in line %d)\n", line);

    error = 1;

    destoryAST(param\_list);

    return NULL;

  }

  enQueue(Q, w);

  //创建参数结点

  char son0\_gt[] = "param";

  p\_treeNode param = insertChild(param\_list, -1, son0\_gt);

  param = Param(Q, param, p, line, token\_text);

  if (error)

    return param\_list;

  //创建参数序列结点

  if (Q\_num(Q) == 0) {

    w = get\_token(p, line, token\_text);

    enQueue(Q, w);

  }

  char son1\_gt[] = "param list";

  p\_treeNode son\_param\_list = insertChild(param\_list, -1, son1\_gt);

  son\_param\_list = Param\_List(Q, son\_param\_list, p, line, token\_text);

  if (error)

    return param\_list;

  return param\_list;

} // end of Param\_List

/\*\*

 \* @description: 生成参数结点

 \* @param {Queue} &Q 读取的token队列

 \* @param {p\_treeNode} param结点

 \* @param {FILE} \*p 读取token的文件指针

 \* @param {int} &line token的行号

 \* @param {char} token\_text token自身值

 \* @return {p\_treeNode} 结点指针

 \* @call: Type\_Specifier

 \* @called\_by: Param\_List

 \*/

p\_treeNode Param(Queue &Q, p\_treeNode param, FILE \*p, int &line,

                 char token\_text[]) {

  token\_info w;

  //创建类型识别符结点

  deQueue(Q, w);

  char son0\_gt[] = "type specifier";

  p\_treeNode type\_specifier = insertChild(param, -1, son0\_gt);

  type\_specifier = Type\_Specifier(w, type\_specifier);

  //创建参数名结点

  deQueue(Q, w);

  token\_info nw = get\_token(p, line, token\_text);

  if (nw.token\_kind == COMMA) {

    ;

  } else if (nw.token\_kind == RPARENTHESE) {

    enQueue(Q, nw);

  } else if (nw.token\_kind == LSUBSCRIPT) {

    //数组参数

    string list;

    list = list + w.token\_text + '[';

    nw = get\_token(p, line, token\_text);

    while (nw.token\_kind != RSUBSCRIPT) {

      list = list + nw.token\_text;

      nw = get\_token(p, line, token\_text);

    }

    list = list + nw.token\_text;

    nw = get\_token(p, line, token\_text);

    if (nw.token\_kind == COMMA) {

      ;

    } else if (nw.token\_kind == RPARENTHESE) {

      enQueue(Q, nw);

    }

    p\_treeNode param\_name = insertChild(param, IDENT, (char \*)list.data());

    return param;

  } // end of else if(数组参数)

  p\_treeNode param\_name = insertChild(param, IDENT, w.token\_text);

  return param;

} // end of Param

/\*\*

 \* @description: 生成语句结点

 \* @param {Queue} &Q 读取的token队列

 \* @param {p\_treeNode} statement结点

 \* @param {FILE} \*p 读取token的文件指针

 \* @param {int} &line token的行号

 \* @param {char} token\_text token自身值

 \* @return {p\_treeNode} 结点指针

 \* @call: is\_Type\_Specifier; is\_CONST\_value; Local\_Val\_Declaration;

 \*        Expression; Selection\_Statement; While\_Statement; For\_Statement;

 \*        Compound\_Statement; Return\_Statement; Break\_Statement;

 \*        Continue\_Statement

 \* @called\_by: Compound\_Statement

 \*/

p\_treeNode Statement(Queue &Q, p\_treeNode statement, FILE \*p, int &line,

                     char token\_text[]) {

  //调用此函数时，Q中有语句的第一个单词

  //此函数结束时，Q中有下一语句的第一个单词

  token\_info w;

  if (is\_Type\_Specifier(Q\_get\_first(Q).token\_kind)) {

    char son0\_gt[] = "local val declaration";

    p\_treeNode local\_val\_declaration = insertChild(statement, -1, son0\_gt);

    local\_val\_declaration =

        Local\_Val\_Declaration(Q, local\_val\_declaration, p, line, token\_text);

    w = get\_token(p, line, token\_text);

    enQueue(Q, w);

    if (error)

      return statement;

  } else if (is\_CONST\_value(Q\_get\_first(Q).token\_kind) ||

             Q\_get\_first(Q).token\_kind == IDENT) {

    char son0\_gt[] = "expression";

    p\_treeNode expression = insertChild(statement, -1, son0\_gt);

    expression = Expression(Q, expression, p, line, token\_text, SEMICOLON);

    w = get\_token(p, line, token\_text);

    enQueue(Q, w);

    if (error)

      return statement;

  } else {

    deQueue(Q, w);

    switch (w.token\_kind) {

    case IF: {

      char son0\_gt[] = "selection statement";

      p\_treeNode selection\_statement = insertChild(statement, -1, son0\_gt);

      selection\_statement =

          Selection\_Statement(Q, selection\_statement, p, line, token\_text);

      if (error)

        return statement;

      break;

    }

    case WHILE: {

      char son0\_gt[] = "while statement";

      p\_treeNode while\_statement = insertChild(statement, -1, son0\_gt);

      while\_statement =

          While\_Statement(Q, while\_statement, p, line, token\_text);

      if (error)

        return statement;

      break;

    }

    case FOR: {

      char son0\_gt[] = "for statement";

      p\_treeNode for\_statement = insertChild(statement, -1, son0\_gt);

      for\_statement = For\_Statement(Q, for\_statement, p, line, token\_text);

      if (error)

        return statement;

      break;

    }

    case LCURLYBRACE: {

      enQueue(Q, w);

      char son0\_gt[] = "compound statement";

      p\_treeNode compound\_statement = insertChild(statement, -1, son0\_gt);

      compound\_statement =

          Compound\_Statement(Q, compound\_statement, p, line, token\_text);

      if (error)

        return statement;

      break;

    }

    case RCURLYBRACE: {

      enQueue(Q, w);

      return NULL;

    }

    case RETURN: {

      char son0\_gt[] = "return statement";

      p\_treeNode return\_statement = insertChild(statement, -1, son0\_gt);

      return\_statement =

          Return\_Statement(Q, return\_statement, p, line, token\_text);

      w = get\_token(p, line, token\_text);

      enQueue(Q, w);

      if (error)

        return statement;

      break;

    }

    case BREAK: {

      char son0\_gt[] = "break statement";

      p\_treeNode break\_statement = insertChild(statement, -1, son0\_gt);

      break\_statement =

          Break\_Statement(Q, break\_statement, p, line, token\_text);

      w = get\_token(p, line, token\_text);

      w = get\_token(p, line, token\_text);

      enQueue(Q, w);

      if (error)

        return statement;

      break;

    }

    case CONTINUE: {

      char son0\_gt[] = "continue statement";

      p\_treeNode continue\_statement = insertChild(statement, -1, son0\_gt);

      continue\_statement =

          Continue\_Statement(Q, continue\_statement, p, line, token\_text);

      w = get\_token(p, line, token\_text);

      w = get\_token(p, line, token\_text);

      enQueue(Q, w);

      if (error)

        return statement;

      break;

    }

    case COMMA:

    case SEMICOLON: {

      w = get\_token(p, line, token\_text);

      enQueue(Q, w);

      return statement;

      break;

    }

    default: {

      printf("fail in creating statement: (in line %d)\n", line);

      error = 1;

      destoryAST(statement);

      return NULL;

      break;

    }

    } // end of switch

  }   // end of else

  return statement;

} // end of Statement

/\*\*

 \* @description: 创建复合语句结点

 \* @param {Queue} &Q 读取的token队列

 \* @param {p\_treeNode} compound\_statement结点

 \* @param {FILE} \*p 读取token的文件指针

 \* @param {int} &line token的行号

 \* @param {char} token\_text token自身值

 \* @return {p\_treeNode} 结点指针

 \* @call: Statement; Compound\_Statement

 \* @called\_by: Statement; Fun\_Declaration

 \*/

p\_treeNode Compound\_Statement(Queue &Q, p\_treeNode compound\_statement, FILE \*p,

                              int &line, char token\_text[]) {

  //调用此函数时，Q中应只有大括号

  token\_info w;

  deQueue(Q, w);

  if (w.token\_kind == RCURLYBRACE) {

    //复合语句结束

    w = get\_token(p, line, token\_text);

    enQueue(Q, w);

    return NULL;

    //结束时Q中应有一个元素

  } else {

    if (w.token\_kind == LCURLYBRACE) {

      w = get\_token(p, line, token\_text);

    }

    enQueue(Q, w);

    //创建语句结点

    char son0\_gt[] = "statement";

    p\_treeNode statement = insertChild(compound\_statement, -1, son0\_gt);

    statement = Statement(Q, statement, p, line, token\_text);

    if (error)

      return compound\_statement;

    //创建复合语句结点

    char son1\_gt[] = "compound statement";

    p\_treeNode son\_compound\_statement =

        insertChild(compound\_statement, -1, son1\_gt);

    son\_compound\_statement =

        Compound\_Statement(Q, son\_compound\_statement, p, line, token\_text);

    return compound\_statement;

  } // end of else

  //结束时Q中应有一个元素

} // end of Compound\_Statement

/\*\*

 \* @description: 生成选择语句结点

 \* @param {Queue} &Q 读取的token队列

 \* @param {p\_treeNode} selection\_statement结点

 \* @param {FILE} \*p 读取token的文件指针

 \* @param {int} &line token的行号

 \* @param {char} token\_text token自身值

 \* @return {p\_treeNode} 结点指针

 \* @call: Condition\_Expression; IF\_Statement; IF\_ELSE\_Statement

 \* @called\_by: Statement

 \*/

p\_treeNode Selection\_Statement(Queue &Q, p\_treeNode selection\_statement,

                               FILE \*p, int &line, char token\_text[]) {

  //调用此函数时Q中无元素

  token\_info w = get\_token(p, line, token\_text);

  if (w.token\_kind != LPARENTHESE) {

    printf("fail in creating selection statement: expext \'(\' (in line %d)",

           line);

    error = 1;

    destoryAST(selection\_statement);

    return NULL;

  }

  char son0\_gt[] = "condition expression";

  p\_treeNode condition\_expression =

      insertChild(selection\_statement, -1, son0\_gt);

  condition\_expression =

      Condition\_Expression(Q, condition\_expression, p, line, token\_text);

  if (error)

    return selection\_statement;

  //创建IF\_Statement语句

  //此时Q中无元素

  p\_treeNode par\_if\_statm;

  iniTreeNode(par\_if\_statm, -1, "par if statm");

  char son1\_gt[] = "if statement";

  p\_treeNode if\_statement = insertChild(par\_if\_statm, -1, son1\_gt);

  if\_statement = IF\_Statement(Q, if\_statement, p, line, token\_text);

  if (error)

    return selection\_statement;

  //此时Q中有下一语句的第一个单词

  //判断是否有else语句

  deQueue(Q, w);

  if (w.token\_kind == ELSE) { //如果为if\_else语句

    char son2\_gt[] = "if else statement";

    p\_treeNode if\_else\_statement =

        insertChild(selection\_statement, -1, son2\_gt);

    if\_else\_statement = IF\_ELSE\_Statement(Q, if\_else\_statement, if\_statement, p,

                                          line, token\_text);

    if (error)

      return selection\_statement;

  } else { //如果为if语句

    enQueue(Q, w);

    selection\_statement->firstChild->nextBro = if\_statement;

  }

  // destoryAST(par\_if\_statm);

  //结束时Q中有下一语句的第一个单词

} // end of Selection\_Statement

/\*\*

 \* @description: 创建条件表达式结点

 \* @param {Queue} &Q 读取的token队列

 \* @param {p\_treeNode} condition\_expression结点

 \* @param {FILE} \*p 读取token的文件指针

 \* @param {int} &line token的行号

 \* @param {char} token\_text token自身值

 \* @return {p\_treeNode} 结点指针

 \* @call: Expression

 \* @called\_by: Selection\_Statement; While\_Statement; For\_Statement

 \*/

p\_treeNode Condition\_Expression(Queue &Q, p\_treeNode condition\_expression,

                                FILE \*p, int &line, char token\_text[]) {

  enQueue(Q, get\_token(p, line, token\_text));

  char son0\_gt[] = "expression";

  p\_treeNode expression = insertChild(condition\_expression, -1, son0\_gt);

  expression = Expression(Q, expression, p, line, token\_text, RPARENTHESE);

  if (error)

    return condition\_expression;

  return condition\_expression;

} // end of Condition\_Expression

/\*\*

 \* @description: 创建IF\_Statement语句结点

 \* @param {Queue} &Q 读取的token队列

 \* @param {p\_treeNode} if\_statement结点

 \* @param {FILE} \*p 读取token的文件指针

 \* @param {int} &line token的行号

 \* @param {char} token\_text token自身值

 \* @return {p\_treeNode} 结点指针

 \* @call: Compound\_Statement; Statement

 \* @called\_by: Selection\_Statement

 \*/

p\_treeNode IF\_Statement(Queue &Q, p\_treeNode if\_statement, FILE \*p, int &line,

                        char token\_text[]) {

  //调用此函数时Q中无元素

  token\_info w = get\_token(p, line, token\_text);

  enQueue(Q, w);

  if (w.token\_kind == LCURLYBRACE) {

    char son0\_gt[] = "compound statement";

    p\_treeNode compound\_statement = insertChild(if\_statement, -1, son0\_gt);

    compound\_statement =

        Compound\_Statement(Q, compound\_statement, p, line, token\_text);

  } else {

    char son0\_gt[] = "statement";

    p\_treeNode statement = insertChild(if\_statement, -1, son0\_gt);

    statement = Statement(Q, statement, p, line, token\_text);

  }

  //结束时Q中有下一语句的第一个单词

  return if\_statement;

} // end of IF\_Statement

/\*\*

 \* @description: 创建IF\_ELSE\_Statement语句结点

 \* @param {Queue} &Q 读取的token队列

 \* @param {p\_treeNode} if\_else\_statement结点

 \* @param {FILE} \*p 读取token的文件指针

 \* @param {int} &line token的行号

 \* @param {char} token\_text token自身值

 \* @return {p\_treeNode} 结点指针

 \* @call: ELSE\_Statement

 \* @called\_by: Selection\_Statement

 \*/

p\_treeNode IF\_ELSE\_Statement(Queue &Q, p\_treeNode if\_else\_statement,

                             p\_treeNode if\_statement, FILE \*p, int &line,

                             char token\_text[]) {

  //调用此函数时Q中无元素，应读取else下一个单词

  enQueue(Q, get\_token(p, line, token\_text));

  if\_else\_statement->firstChild = if\_statement;

  char son1\_gt[] = "else statement";

  p\_treeNode else\_statement = insertChild(if\_else\_statement, -1, son1\_gt);

  else\_statement = ELSE\_Statement(Q, else\_statement, p, line, token\_text);

  return if\_else\_statement;

  //结束时Q中有下一语句的第一个单词

} // end of IF\_ELSE\_Statement

/\*\*

 \* @description: 创建ELSE\_Statement语句结点

 \* @param {Queue} &Q 读取的token队列

 \* @param {p\_treeNode} else\_statement

 \* @param {FILE} \*p 读取token的文件指针

 \* @param {int} &line token的行号

 \* @param {char} token\_text token自身值

 \* @return {p\_treeNode} 结点指针

 \* @call: Compound\_Statement; Statement

 \* @called\_by: Selection\_Statement

 \*/

p\_treeNode ELSE\_Statement(Queue &Q, p\_treeNode else\_statement, FILE \*p,

                          int &line, char token\_text[]) {

  //调用此函数时，Q中有else后的第一个单词

  if (Q\_get\_first(Q).token\_kind == LCURLYBRACE) {

    char son0\_gt[] = "compound statement";

    p\_treeNode statement = insertChild(else\_statement, -1, son0\_gt);

    statement = Compound\_Statement(Q, statement, p, line, token\_text);

  } else {

    char son0\_gt[] = "statement";

    p\_treeNode statement = insertChild(else\_statement, -1, son0\_gt);

    statement = Statement(Q, statement, p, line, token\_text);

  }

  return else\_statement;

  //结束时Q中有下一语句的第一个单词

} // end of ELSE\_Statement

/\*\*

 \* @description: 创建While\_Statement语句结点

 \* @param {Queue} &Q 读取的token队列

 \* @param {p\_treeNode} while\_statement结点

 \* @param {FILE} \*p 读取token的文件指针

 \* @param {int} &line token的行号

 \* @param {char} token\_text token自身值

 \* @return {p\_treeNode} 结点指针

 \* @call: Condition\_Expression; Compound\_Statement; Statement

 \* @called\_by: Statement

 \*/

p\_treeNode While\_Statement(Queue &Q, p\_treeNode while\_statement, FILE \*p,

                           int &line, char token\_text[]) {

  //调用此函数时Q中无元素

  token\_info w = get\_token(p, line, token\_text);

  if (w.token\_kind != LPARENTHESE) {

    printf("fail in creating while statement: expect \'(\' (in line %d)\n",

           line);

    error = 1;

    destoryAST(while\_statement);

    return NULL;

  }

  //创建条件表达式语句

  char son0\_gt[] = "condition expression";

  p\_treeNode condition\_expression = insertChild(while\_statement, -1, son0\_gt);

  condition\_expression =

      Condition\_Expression(Q, condition\_expression, p, line, token\_text);

  if (error)

    return while\_statement;

  //创建循环执行语句

  w = get\_token(p, line, token\_text);

  enQueue(Q, w);

  if (w.token\_kind == LCURLYBRACE) {

    char son1\_gt[] = "compound statement";

    p\_treeNode iteration\_statement = insertChild(while\_statement, -1, son1\_gt);

    iteration\_statement =

        Compound\_Statement(Q, iteration\_statement, p, line, token\_text);

  } else {

    char son1\_gt[] = "statement";

    p\_treeNode iteration\_statement = insertChild(while\_statement, -1, son1\_gt);

    iteration\_statement =

        Statement(Q, iteration\_statement, p, line, token\_text);

  }

  return while\_statement;

} // end of While\_Statement

/\*\*

 \* @description: 创建For\_Statement语句结点

 \* @param {Queue} &Q 读取的token队列

 \* @param {p\_treeNode} for\_statement结点

 \* @param {FILE} \*p 读取token的文件指针

 \* @param {int} &line token的行号

 \* @param {char} token\_text token自身值

 \* @return {p\_treeNode} 结点指针

 \* @call: Statement; Expression; Compound\_Statement

 \* @called\_by: Statement

 \*/

p\_treeNode For\_Statement(Queue &Q, p\_treeNode for\_statement, FILE \*p, int &line,

                         char token\_text[]) {

  //调用此函数时Q中无元素

  token\_info w = get\_token(p, line, token\_text);

  if (w.token\_kind != LPARENTHESE) {

    printf("fail in creating for statement: expect \'(\' (in line %d)\n", line);

    error = 1;

    destoryAST(for\_statement);

    return NULL;

  }

  w = get\_token(p, line, token\_text);

  //创建循环条件变量初始化语句结点

  enQueue(Q, w);

  char son0\_gt[] = "condition val initial";

  p\_treeNode condition\_val\_init = insertChild(for\_statement, -1, son0\_gt);

  condition\_val\_init = Statement(Q, condition\_val\_init, p, line, token\_text);

  if (error)

    return for\_statement;

  //创建循环条件表达式结点

  char son1\_gt[] = "expression";

  p\_treeNode condition\_expression = insertChild(for\_statement, -1, son1\_gt);

  condition\_expression =

      Expression(Q, condition\_expression, p, line, token\_text, SEMICOLON);

  if (error)

    return for\_statement;

  //创建循环变量迭代表达式结点

  w = get\_token(p, line, token\_text);

  enQueue(Q, w);

  char son2\_gt[] = "expression";

  p\_treeNode condition\_iteration = insertChild(for\_statement, -1, son2\_gt);

  condition\_iteration =

      Expression(Q, condition\_iteration, p, line, token\_text, RPARENTHESE);

  if (error)

    return for\_statement;

  //创建循环执行语句部分

  w = get\_token(p, line, token\_text);

  enQueue(Q, w);

  if (w.token\_kind == LCURLYBRACE) {

    char son3\_gt[] = "compound statement";

    p\_treeNode iteration\_statement = insertChild(for\_statement, -1, son3\_gt);

    iteration\_statement =

        Compound\_Statement(Q, iteration\_statement, p, line, token\_text);

  } else {

    char son3\_gt[] = "statement";

    p\_treeNode iteration\_statement = insertChild(for\_statement, -1, son3\_gt);

    iteration\_statement =

        Statement(Q, iteration\_statement, p, line, token\_text);

  }

  return for\_statement;

} // end of For\_Statement

/\*\*

 \* @description: 创建局部变量定义结点

 \* @param {Queue} &Q 读取的token队列

 \* @param {p\_treeNode} local\_val\_declaration结点

 \* @param {FILE} \*p 读取token的文件指针

 \* @param {int} &line token的行号

 \* @param {char} token\_text token自身值

 \* @return {p\_treeNode} 结点指针

 \* @call: Type\_Specifier; Val\_List

 \* @called\_by: Statement

 \*/

p\_treeNode Local\_Val\_Declaration(Queue &Q, p\_treeNode local\_val\_declaration,

                                 FILE \*p, int &line, char token\_text[]) {

  //创建类型识别符结点

  token\_info w;

  deQueue(Q, w);

  char son\_0gt[] = "type specifier";

  p\_treeNode type\_specifier = insertChild(local\_val\_declaration, -1, son\_0gt);

  type\_specifier = Type\_Specifier(w, type\_specifier);

  //创建变量序列结点

  w = get\_token(p, line, token\_text);

  if (w.token\_kind != IDENT) {

    printf(

        "fail in creating local val declaration: expect IDENT (in line %d)\n",

        line);

    error = 1;

    destoryAST(local\_val\_declaration);

    return NULL;

  }

  enQueue(Q, w);

  w = get\_token(p, line, token\_text);

  if (w.token\_kind != COMMA && w.token\_kind != SEMICOLON &&

      w.token\_kind != LSUBSCRIPT && w.token\_kind != ASSIGN) {

    printf(

        "fail in creating local val declaration: expect \';\' (in line %d)\n",

        line);

    error = 1;

    destoryAST(local\_val\_declaration);

    return NULL;

  }

  enQueue(Q, w);

  char son1\_gt[] = "val list";

  p\_treeNode val\_list = insertChild(local\_val\_declaration, -1, son1\_gt);

  val\_list = Val\_List(Q, val\_list, p, line, token\_text);

  return local\_val\_declaration;

} // end of Local\_Val\_List

/\*\*

 \* @description: 创建函数调用结点

 \* @param {Queue} &Q 读取的token队列

 \* @param {p\_treeNode} Fun\_Call结点

 \* @param {FILE} \*p 读取token的文件指针

 \* @param {int} &line token的行号

 \* @param {char} token\_text token自身值

 \* @return {p\_treeNode} 结点指针

 \* @call: Params\_Call

 \* @called\_by: Expression

 \*/

p\_treeNode Fun\_Call(Queue &Q, p\_treeNode fun\_Call, FILE \*p, int &line,

                    char token\_text[]) {

  //调用此函数时Q中应有一个ident元素

  //创建函数名结点

  token\_info w;

  deQueue(Q, w);

  p\_treeNode fun\_name = insertChild(fun\_Call, IDENT, w.token\_text);

  //创建函数参数集结点

  char son1\_gt[] = "params";

  p\_treeNode params = insertChild(fun\_Call, -1, son1\_gt);

  params = Params\_Call(Q, params, p, line, token\_text);

  return fun\_Call;

} // end of Fun\_Call

/\*\*

 \* @description: 创建函数调用的参数语句结点

 \* @param {Queue} &Q 读取的token队列

 \* @param {p\_treeNode} params\_call结点

 \* @param {FILE} \*p 读取token的文件指针

 \* @param {int} &line token的行号

 \* @param {char} token\_text token自身值

 \* @return {p\_treeNode} 结点指针

 \* @call:

 \* @called\_by: Fun\_Call

 \*/

p\_treeNode Params\_Call(Queue &Q, p\_treeNode params\_call, FILE \*p, int &line,

                       char token\_text[]) {

  //调用此函数时Q中应无元素

  token\_info w = get\_token(p, line, token\_text);

  while (w.token\_kind != RPARENTHESE) {

    if (w.token\_kind == IDENT || is\_CONST\_value(w.token\_kind)) {

      p\_treeNode param = insertChild(params\_call, w.token\_kind, w.token\_text);

      w = get\_token(p, line, token\_text);

    } else if (w.token\_kind == COMMA) {

      w = get\_token(p, line, token\_text);

    } else {

      printf("fail in creating fun call params: expect \')\' (in line %d)\n",

             line);

      error = 1;

      destoryAST(params\_call);

      return NULL;

    }

  }

  return params\_call;

}

/\*\*

 \* @description: 创建返回语句结点

 \* @param {Queue} &Q 读取的token队列

 \* @param {p\_treeNode} return\_statement结点

 \* @param {FILE} \*p 读取token的文件指针

 \* @param {int} &line token的行号

 \* @param {char} token\_text token自身值

 \* @return {p\_treeNode} 结点指针

 \* @call: Expression

 \* @called\_by: Statement

 \*/

p\_treeNode Return\_Statement(Queue &Q, p\_treeNode return\_statement, FILE \*p,

                            int &line, char token\_text[]) {

  token\_info w = get\_token(p, line, token\_text);

  enQueue(Q, w);

  char son0\_gt[] = "expression";

  p\_treeNode return\_expression = insertChild(return\_statement, -1, son0\_gt);

  return\_expression =

      Expression(Q, return\_expression, p, line, token\_text, SEMICOLON);

  return return\_statement;

} // end of Return\_Statement

/\*\*

 \* @description: 创建break语句结点

 \* @param {Queue} &Q 读取的token队列

 \* @param {p\_treeNode} break结点

 \* @param {FILE} \*p 读取token的文件指针

 \* @param {int} &line token的行号

 \* @param {char} token\_text token自身值

 \* @return {p\_treeNode} 结点指针

 \* @call:

 \* @called\_by: Statement

 \*/

p\_treeNode Break\_Statement(Queue &Q, p\_treeNode break\_statement, FILE \*p,

                           int &line, char token\_text[]) {

  return break\_statement;

}

/\*\*

 \* @description: 创建continue语句结点

 \* @param {Queue} &Q 读取的token队列

 \* @param {p\_treeNode} continue结点

 \* @param {FILE} \*p 读取token的文件指针

 \* @param {int} &line token的行号

 \* @param {char} token\_text token自身值

 \* @return {p\_treeNode} 结点指针

 \* @call:

 \* @called\_by: Statement

 \*/

p\_treeNode Continue\_Statement(Queue &Q, p\_treeNode continue\_statement, FILE \*p,

                              int &line, char token\_text[]) {

  return continue\_statement;

}

/\*\*

 \* @description: 创建表达式结点

 \* @param {Queue} &Q 读取的token队列

 \* @param {p\_treeNode} expression结点

 \* @param {FILE} \*p 读取token的文件指针

 \* @param {int} &line token的行号

 \* @param {char} token\_text token自身值

 \* @param {int} endsym 标记结束符号

 \* @return {p\_treeNode} 结点指针

 \* @call:

 \* @called\_by: Statement; Condition\_Expression; For\_Statement; is\_Operator

 \*/

p\_treeNode Expression(Queue &Q, p\_treeNode expression, FILE \*p, int &line,

                      char token\_text[], int endsym) {

  //调用该算法时，Q中有表达式的第一个单词

  Stack op;  //运算符栈

  Stack opn; //操作数栈

  iniStack(op);

  iniStack(opn);

  bool expError = false; //错误标记

  token\_info w;

  deQueue(Q, w);

  p\_treeNode pw = NULL;

  iniTreeNode(pw, BEGIN\_END, "#");

  S\_push(op, pw);

  pw = NULL;

  while (!isEmptyStack(op) && w.token\_kind != BEGIN\_END && !expError) {

    if (is\_CONST\_value(w.token\_kind)) {

      //常数直接进操作数栈，并读取下一个单词

      iniTreeNode(pw, w.token\_kind, w.token\_text);

      S\_push(opn, pw);

      pw = NULL;

      w = get\_token(p, line, token\_text);

      continue;

    } // end of if(w is const value)

    else if (w.token\_kind == IDENT) {

      //读入下一个单词检查是否为函数调用

      token\_info nw = get\_token(p, line, token\_text);

      if (nw.token\_kind == LPARENTHESE) {

        // w为函数调用

        enQueue(Q, w);

        char is\_fun\_call[] = "fun call";

        iniTreeNode(pw, -1, is\_fun\_call);

        pw = Fun\_Call(Q, pw, p, line, token\_text);

        S\_push(opn, pw);

        pw = NULL;

        w = get\_token(p, line, token\_text);

        continue;

      } else {

        // w不为函数调用

        iniTreeNode(pw, IDENT, w.token\_text);

        S\_push(opn, pw);

        pw = NULL;

        w = nw;

        continue;

      }

    } // end of else if(w is ident)

    else if (w.token\_kind == LPARENTHESE) {

      //左小括号直接进栈

      S\_push(op, pw);

      pw = NULL;

      w = get\_token(p, line, token\_text);

    } else if (is\_Operator(w.token\_kind) || w.token\_kind == RPARENTHESE) {

      iniTreeNode(pw, w.token\_kind, w.token\_text);

      p\_treeNode top = NULL;

      S\_pop(op, top);

      if (top->node\_type == BEGIN\_END) {

        if (w.token\_kind == RPARENTHESE) {

          if (w.token\_kind == endsym) {

            w.token\_kind = BEGIN\_END;

            continue;

          } else {

            error = 2;

            continue;

          }

        }

        S\_push(op, top);

        S\_push(op, pw);

        w = get\_token(p, line, token\_text);

        continue;

      } else if (top->node\_type == LPARENTHESE) {

        if (w.token\_kind == RPARENTHESE) {

          //去括号

          pw = NULL;

          w = get\_token(p, line, token\_text);

        } else {

          S\_push(op, top);

          S\_push(op, pw);

          pw = NULL;

          w = get\_token(p, line, token\_text);

        }

      } else if (top->node\_type < w.token\_kind) {

        // w的运算优先级高

        S\_push(op, top);

        S\_push(op, pw);

        pw = NULL;

        w = get\_token(p, line, token\_text);

      } else if (top->node\_type >= w.token\_kind) {

        // w的运算优先级低

        //取出栈顶运算符执行操作

        p\_treeNode t1, t2; // t1:前一个操作数; t2:后一个操作数

        if (!S\_pop(opn, t2)) {

          expError = 3;

        }

        if (!S\_pop(opn, t1)) {

          expError = 4;

        }

        top->firstChild = t1;

        t1->nextBro = t2;

        S\_push(opn, top);

        continue;

      } else if (w.token\_kind == endsym) {

        w.token\_kind = BEGIN\_END;

      } else {

        expError = 5;

      }

    } // end of if(w is operator)

    else if (w.token\_kind == endsym) {

      w.token\_kind = BEGIN\_END;

    } else {

      expError = 6;

    }

  } // end of while

  while (!isEmptyStack(op) && S\_get\_top(op)->node\_type != BEGIN\_END) {

    //最后对栈中所有运算符执行操作

    p\_treeNode top;

    S\_pop(op, top);

    p\_treeNode t1, t2; // t1:前一个操作数; t2:后一个操作数

    if (!S\_pop(opn, t2))

      expError = 7;

    if (!S\_pop(opn, t1))

      expError = 8;

    top->firstChild = t1;

    t1->nextBro = t2;

    S\_push(opn, top);

  } // end of while

  if (S\_num(opn) == 1 && !expError) {

    S\_pop(opn, expression->firstChild);

    return expression;

  } else {

    printf("fail in creating expression! %d (in line %d)\n", expError, line);

    error = 1;

    destoryAST(expression);

    return NULL;

  }

}

/\*\*

 \* @description: 创建常量语句结点

 \* @param {token\_info} w token种类信息

 \* @param {p\_treeNode} const\_value结点

 \* @return {p\_treeNode} 结点指针

 \* @call:

 \* @called\_by: Expression

 \*/

p\_treeNode Const\_Value(token\_info w, p\_treeNode const\_value) {

  iniTreeNode(const\_value, w.token\_kind, w.token\_text);

  return const\_value;

}

### 附录7 parsing.h：

#ifndef COURSE\_DESIGN\_PARSER\_H

#define COURSE\_DESIGN\_PARSER\_H

#include "lexer.h"

#include "AST.h"

#include "Queue.h"

#include "Stack.h"

#include <iostream>

#include <string>

using namespace std;

bool is\_Type\_Specifier(int token\_type);

bool is\_Operator(int token\_type);

bool is\_CONST\_value(int token\_type);

p\_treeNode Program(Queue &Q, FILE \*p, int &line, char token\_text[]);

p\_treeNode ex\_Declaration\_List(Queue &Q, p\_treeNode ex\_declaration\_list, FILE \*p, int &line, char token\_text[]);

p\_treeNode ex\_Val\_Declaration(Queue &Q, p\_treeNode val\_dec, FILE \*p, int &line, char token\_text[]);

p\_treeNode Fun\_Declaration(Queue &Q, p\_treeNode fun\_dec, FILE \*p, int &line, char token\_text[]);

p\_treeNode Type\_Specifier(token\_info w, p\_treeNode type\_specifier);

p\_treeNode Val\_List(Queue &Q, p\_treeNode val\_list, FILE \*p, int &line, char token\_text[]);

p\_treeNode Params(Queue &Q, p\_treeNode params, FILE \*p, int &line, char token\_text[]);

p\_treeNode Param\_List(Queue &Q, p\_treeNode param\_list, FILE \*p, int &line, char token\_text[]);

p\_treeNode Param(Queue &Q, p\_treeNode param, FILE \*p, int &line, char token\_text[]);

p\_treeNode Statement(Queue &Q, p\_treeNode statement, FILE \*p, int &line, char token\_text[]);

p\_treeNode Compound\_Statement(Queue &Q, p\_treeNode compound\_statement, FILE \*p, int &line, char token\_text[]);

p\_treeNode Selection\_Statement(Queue &Q, p\_treeNode selection\_statement, FILE \*p, int &line, char token\_text[]);

p\_treeNode Condition\_Expression(Queue &Q, p\_treeNode condition\_expression, FILE \*p, int &line, char token\_text[]);

p\_treeNode IF\_Statement(Queue &Q, p\_treeNode if\_statement, FILE \*p, int &line, char token\_text[]);

p\_treeNode IF\_ELSE\_Statement(Queue &Q, p\_treeNode if\_else\_statement, p\_treeNode if\_statement, FILE \*p, int &line, char token\_text[]);

p\_treeNode ELSE\_Statement(Queue &Q, p\_treeNode else\_statement, FILE \*p, int &line, char token\_text[]);

p\_treeNode While\_Statement(Queue &Q, p\_treeNode while\_statement, FILE \*p, int &line, char token\_text[]);

p\_treeNode For\_Statement(Queue &Q, p\_treeNode for\_statement, FILE \*p, int &line, char token\_text[]);

p\_treeNode Local\_Val\_Declaration(Queue &Q, p\_treeNode local\_val\_declaration, FILE \*p, int &line, char token\_text[]);

p\_treeNode Fun\_Call(Queue &Q, p\_treeNode fun\_Call, FILE \*p, int &line, char token\_text[]);

p\_treeNode Params\_Call(Queue &Q, p\_treeNode params\_call, FILE \*p, int &line, char token\_text[]);

p\_treeNode Return\_Statement(Queue &Q, p\_treeNode return\_statement, FILE \*p, int &line, char token\_text[]);

p\_treeNode Break\_Statement(Queue &Q, p\_treeNode break\_statement, FILE \*p, int &line, char token\_text[]);

p\_treeNode Continue\_Statement(Queue &Q, p\_treeNode continue\_statement, FILE \*p, int &line, char token\_text[]);

p\_treeNode Expression(Queue &Q, p\_treeNode expression, FILE \*p, int &line, char token\_text[], int endsym);

p\_treeNode Const\_Value(token\_info w, p\_treeNode const\_value);

#endif //COURSE\_DESIGN\_PARSER\_H

### 附录8 pre\_process.cpp：

#include "pre\_process.h"

#include "pre\_process.h"

#include <iostream>

#include <string>

using namespace std;

/\*\*

 \* @description: 删除注释、执行包含文件与宏定义，然后将剩余部分保存入新文件

 \* @param {char} originFileName源文件名

 \* @param {char} processedFileName新文件名

 \* @return {int} 0：文件打开失败；1：处理成功

 \* @call: (none)

 \* @called\_by: main(in file "main.cpp")

 \*/

int pre\_process(char originFileName[], char processedFileName[]) {

  FILE \*origin = fopen(originFileName, "r");

  FILE \*processed = fopen(processedFileName, "w");

  if (!origin || !processed)

    return 0; //文件打开失败

  else

    printf("打开文件成功\n");

  printf("\n\*\*\*\*\*\*\*\*\*\*进行编译预处理\*\*\*\*\*\*\*\*\*\*\n");

  char ch;

  string command, file\_name, word\_defined, define\_as;

  while (!feof(origin)) {

    if ((ch = fgetc(origin)) == '#') {

      while ((ch = fgetc(origin)) == ' ') { //清除空白符

        ;

      }

      ungetc(ch, origin);

      while ((ch = fgetc(origin)) != ' ' && ch != '<' && ch != '\"' &&

             ch != '\n' && ch != '\t') { //获得预处理命令

        if (ch == '\n')

          putc('\n', processed);

        command = command + ch;

      }

      ungetc(ch, origin);

      if (command == "include") { //处理文件包含

        while ((ch = fgetc(origin)) == ' ' || ch == '<' ||

               ch == '\"') { //清除空白符

          ;

        }

        ungetc(ch, origin);

        while ((ch = fgetc(origin)) != '>' && ch != '\"') {

          file\_name = file\_name + ch;

        }

        cout << "#包含文件：" << file\_name << "\n";

      } else if (command == "define") {                     //处理宏定义

        while ((ch = fgetc(origin)) == ' ' || ch == '\t') { //清除空白符

          ;

        }

        ungetc(ch, origin);

        while ((ch = fgetc(origin)) != ' ' && ch != '\t' && ch != '\n') {

          word\_defined = word\_defined + ch;

        }

        ungetc(ch, origin);

        while ((ch = fgetc(origin)) == ' ' || ch == '\t') {

          ;

        }

        if (ch == '\n') {

          cout << "#定义：" << word\_defined << "\n";

        } else {

          do {

            define\_as = define\_as + ch;

          } while ((ch = fgetc(origin)) != '\n');

          cout << "#定义：" << word\_defined << " 为：" << define\_as << "\n";

          fgetc(origin);

        }

        ungetc(ch, origin);

      } // end of else if

      else {

        printf("未知的预处理指令\n");

        while ((ch = fgetc(origin)) != '\n') {

          ;

        }

      }

      command.clear();

      file\_name.clear();

      word\_defined.clear();

      define\_as.clear(); //重置字符串

      continue;

    } // end of if ((ch = fgetc(origin)) == '#')

    else if (ch == '/') {

      switch ((ch = fgetc(origin))) {

      case '/':

        //行注释

        do {

          ch = fgetc(origin);

        } while (ch != '\n');

        putc('\n', processed);

        break;

      case '\*':

        //块注释

        ch = getc(origin);

        do {

          if (ch != '\*') {

            if (ch == '\n')

              putc('\n', processed);

            ch = fgetc(origin);

          }

        } while (!(ch == '\*' && (ch = fgetc(origin)) == '/'));

        break;

      default:

        //非注释

        fputc('/', processed);

        fputc(ch, processed);

        break;

      } // end of switch

      if ((ch = fgetc(origin)) == EOF)

        return 1;

      ungetc(ch, origin);

    } // end of else if ((ch = fgetc(origin)) == '/')

    else if (ch != EOF) {

      fputc(ch, processed);

    }

  } // end of while

  fclose(origin);

  fclose(processed);

  return 1;

} // end of pre\_process

### 附录9 pre\_process.h：

#ifndef COURSE\_DESIGN\_PROCESS\_H

#define COURSE\_DESIGN\_PROCESS\_H

int pre\_process(char originFileName[], char processedFileName[]);

#endif //COURSE\_DESIGN\_PROCESS\_H

### 附录10 print\_file.cpp：

#include "print\_file.h"

void f\_printTabs(FILE \*fp, int tabs) {

  for (int i = 0; i < tabs; i++)

    fprintf(fp, "    ");

}

void f\_printFunCall(FILE \*fp, p\_treeNode fun\_call) {

  fprintf(fp, "%s", fun\_call->firstChild->self\_value);

  fprintf(fp, "(");

  if (fun\_call->firstChild->nextBro) {

    for (p\_treeNode p = fun\_call->firstChild->nextBro->firstChild; p;) {

      fprintf(fp, "%s", p->self\_value);

      p = p->nextBro;

      if (p) {

        fprintf(fp, ",");

      }

    } // end of for

  }   // end of if

  fprintf(fp, ")");

} // end of f\_printFunCall

void f\_printExp(FILE \*fp, p\_treeNode p) {

  if (!p)

    return;

  if (strcmp(p->self\_value, "fun call") == 0) {

    f\_printFunCall(fp, p);

  } else {

    fprintf(fp, "(");

    f\_printExp(fp, p->firstChild);

    fprintf(fp, " %s ", p->self\_value);

    if (p->firstChild) {

      f\_printExp(fp, p->firstChild->nextBro);

    }

    fprintf(fp, ")");

    return;

  } // end of else

} // end of f\_printExp

void f\_traverseExp(FILE \*fp, p\_treeNode p, bool newRoll) {

  f\_printExp(fp, p);

  if (newRoll)

    fprintf(fp, ";");

} // end of f\_traverseExp

void f\_traverse(FILE \*fp, p\_treeNode p, int tabs, bool newRoll) {

  if (!p)

    return;

  if (strcmp(p->self\_value, "fun declaration") == 0 ||

      strcmp(p->self\_value, "fun announce") == 0) {

    fprintf(fp, "%s ", p->firstChild->firstChild->self\_value);

    fprintf(fp, "%s", p->firstChild->nextBro->self\_value);

    if (!p->firstChild->nextBro->nextBro->firstChild) {

      // 无参数

      fprintf(fp, "(void)");

    } else {

      // 有参数

      // param list

      fprintf(fp, "(");

      f\_traverse(fp, p->firstChild->nextBro->nextBro->firstChild, tabs + 1,

                 true);

      fprintf(fp, ")");

    }

    if (strcmp(p->self\_value, "fun declaration") == 0) {

      // fun compound statement

      fprintf(fp, "\n{\n");

      tabs++;

      f\_traverse(fp, p->firstChild->nextBro->nextBro->nextBro, tabs, true);

      tabs--;

      fprintf(fp, "\n}");

      return;

    } else {

      fprintf(fp, ";\n");

      return;

    }

  } // end of if(fun\_dec or fun\_ann)

  else if (strcmp(p->self\_value, "param list") == 0) {

    if (!p->firstChild)

      return;

    //函数声明有参数的param list结点

    fprintf(fp, "%s %s", p->firstChild->firstChild->firstChild->self\_value,

            p->firstChild->firstChild->nextBro->self\_value);

    if (p->firstChild->nextBro->firstChild) {

      fprintf(fp, ", ");

    }

    f\_traverse(fp, p->firstChild->nextBro, tabs, 1);

    return;

  } // end of else if(param\_list)

  else if (strcmp(p->self\_value, "ex val declaration") == 0 ||

           strcmp(p->self\_value, "local val declaration") == 0) {

    //变量声明

    fprintf(fp, "%s ", p->firstChild->firstChild->self\_value);

    f\_traverse(fp, p->firstChild->nextBro, tabs, 0); // val list结点

    fprintf(fp, ";");

    if (strcmp(p->self\_value, "ex val declaration") == 0)

      fprintf(fp, "\n");

    return;

  } // end of else if(val\_dec)

  else if (strcmp(p->self\_value, "val list") == 0) {

    if (!p->firstChild) {

      return;

    }

    fprintf(fp, "%s", p->firstChild->self\_value);

    if (p->firstChild->nextBro)

      if (p->firstChild->nextBro->firstChild) {

        fprintf(fp, ", ");

      }

    f\_traverse(fp, p->firstChild->nextBro, tabs, 1);

    return;

  } // end of else if(val\_list)

  else if (strcmp(p->self\_value, "fun compound statement") == 0) {

    //函数语句结点fun compound statement

    f\_traverse(fp, p->firstChild, tabs, 0);

    f\_traverse(fp, p->firstChild->nextBro, tabs, 0);

    return;

  } else if (strcmp(p->self\_value, "statement") == 0) {

    // statement结点

    if (!p->firstChild) {

      return;

    }

    f\_printTabs(fp, tabs);

    f\_traverse(fp, p->firstChild, tabs, 1);

    fprintf(fp, "\n");

    return;

  } else if (strcmp(p->self\_value, "compound statement") == 0) {

    // compound statement结点

    if (!p->firstChild) {

      return;

    }

    f\_traverse(fp, p->firstChild, tabs, 0);

    f\_traverse(fp, p->firstChild->nextBro, tabs, 0);

    return;

  } else if (strcmp(p->self\_value, "selection statement") == 0) {

    fprintf(fp, "if");

    f\_traverse(fp, p->firstChild, tabs, 1);

    f\_traverse(fp, p->firstChild->nextBro, tabs, 0);

    return;

  } else if (strcmp(p->self\_value, "condition expression") == 0) {

    f\_traverse(fp, p->firstChild, tabs, 0);

    fprintf(fp, "\n");

    return;

  } else if (strcmp(p->self\_value, "if else statement") == 0) {

    f\_traverse(fp, p->firstChild, tabs, 1);

    f\_traverse(fp, p->firstChild->nextBro, tabs, 1);

    return;

  } else if (strcmp(p->self\_value, "if statement") == 0) {

    f\_printTabs(fp, tabs);

    fprintf(fp, "{\n");

    tabs++;

    f\_traverse(fp, p->firstChild, tabs, 1);

    tabs--;

    f\_printTabs(fp, tabs);

    fprintf(fp, "}\n");

    return;

  } else if (strcmp(p->self\_value, "else statement") == 0) {

    f\_printTabs(fp, tabs);

    fprintf(fp, "else\n");

    f\_printTabs(fp, tabs);

    fprintf(fp, "{\n");

    tabs++;

    f\_traverse(fp, p->firstChild, tabs, 1);

    tabs--;

    f\_printTabs(fp, tabs);

    fprintf(fp, "}\n");

    return;

  } else if (strcmp(p->self\_value, "while statement") == 0) {

    fprintf(fp, "while");

    f\_traverse(fp, p->firstChild, tabs, 1);

    f\_printTabs(fp, tabs);

    fprintf(fp, "{\n");

    tabs++;

    f\_traverse(fp, p->firstChild->nextBro, tabs, 0);

    tabs--;

    f\_printTabs(fp, tabs);

    fprintf(fp, "}\n");

    return;

  } else if (strcmp(p->self\_value, "for statement") == 0) {

    fprintf(fp, "for");

    fprintf(fp, "( ");

    f\_traverse(fp, p->firstChild->firstChild, tabs, 0);

    if (strcmp(p->firstChild->firstChild->self\_value, "expression") == 0)

      fprintf(fp, ";");

    fprintf(fp, " ");

    f\_traverse(fp, p->firstChild->nextBro, tabs, 0);

    fprintf(fp, "; ");

    f\_traverse(fp, p->firstChild->nextBro->nextBro, tabs, 0);

    fprintf(fp, ")\n");

    f\_printTabs(fp, tabs);

    fprintf(fp, "{\n");

    tabs++;

    f\_traverse(fp, p->firstChild->nextBro->nextBro->nextBro, tabs, 0);

    tabs--;

    f\_printTabs(fp, tabs);

    fprintf(fp, "}\n");

    return;

  } else if (strcmp(p->self\_value, "break statement") == 0) {

    fprintf(fp, "break;");

    return;

  } else if (strcmp(p->self\_value, "continue statement") == 0) {

    fprintf(fp, "continue;");

    return;

  } else if (strcmp(p->self\_value, "return statement") == 0) {

    f\_printTabs(fp, tabs);

    fprintf(fp, "return ");

    f\_traverse(fp, p->firstChild, tabs, 1);

    return;

  } else if (strcmp(p->self\_value, "expression") == 0) {

    f\_traverseExp(fp, p->firstChild, newRoll);

    return;

  } else {

    f\_traverse(fp, p->firstChild, tabs, newRoll);

    if (p->firstChild)

      f\_traverse(fp, p->firstChild->nextBro, tabs, newRoll);

    return;

  }

} // end of f\_traverse

### 附录11 print\_file.h：

#ifndef COURSE\_DESIGN\_PRINT\_FILE\_H

#define COURSE\_DESIGN\_PRINT\_FILE\_H

#include "parsing.h"

void f\_printTabs(FILE \*fp, int tabs);

void f\_printFunCall(FILE \*fp, p\_treeNode fun\_call);

void f\_printExp(FILE \*fp, p\_treeNode p);

void f\_traverseExp(FILE \*fp, p\_treeNode p, bool newRoll);

void f\_traverse(FILE \*fp, p\_treeNode p, int tabs, bool newRoll);

#endif // COURSE\_DESIGN\_PRINT\_FILE\_H

### 附录12 Queue.cpp：

#include "Queue.h"

//队列函数

/\*\*

 \* @description: 初始化队列

 \* @param {Queue} Q

 \* @return

 \*/

void iniQueue(Queue &Q)

{

    Q.begin = (QueueNode \*)malloc(sizeof(QueueNode));

    Q.end = Q.begin;

    Q.begin->next = NULL;

    Q.num = 0;

    return;

}

/\*\*

 \* @description: 将元素e入队列

 \* @param {Queue} &Q

 \* @param {Q\_ElemType} e

 \* @return 成功返回OK，失败返回ERROR

 \*/

status enQueue(Queue &Q, Q\_ElemType e)

{

    QueueNode \*new\_node = (QueueNode \*)malloc(sizeof(QueueNode));

    if (!new\_node)

        return ERROR;

    new\_node->next = NULL;

    new\_node->data = e;

    Q.end->next = new\_node;

    Q.end = Q.end->next;

    Q.num += 1;

    return OK;

}

/\*\*

 \* @description: 将队列首元素出队，赋值给e

 \* @param {Queue} &Q

 \* @param {Q\_ElemType} &e

 \* @return 成功返回OK，失败返回ERROR

 \*/

status deQueue(Queue &Q, Q\_ElemType &e)

{

    if (Q.num == 0)

        return ERROR;

    e = Q.begin->next->data;

    if (Q.num == 1)

    {

        free(Q.begin->next);

        Q.end = Q.begin;

        Q.num = 0;

        return OK;

    }

    else

    {

        QueueNode \*to\_delete = Q.begin->next;

        Q.begin->next = Q.begin->next->next;

        free(to\_delete);

        Q.num -= 1;

        return OK;

    }

}

/\*\*

 \* @description: 查找队列中元素个数

 \* @param {Queue} Q

 \* @return {int} 元素个数

 \*/

int Q\_num(Queue Q)

{

    return Q.num;

}

/\*\*

 \* @description: 获得队列第一个元素的种类

 \* @param {Queue} Q

 \* @return {Q\_ElemType}

 \*/

Q\_ElemType Q\_get\_first(Queue Q)

{

    if (Q.num != 0)

        return Q.begin->next->data;

}

/\*\*

 \* @description: 获得队列最后一个元素的种类

 \* @param {Queue} Q

 \* @return {Q\_ElemType}

 \*/

Q\_ElemType Q\_get\_last(Queue Q)

{

    return Q.end->data;

}

### 附录13 Queue.h：

#ifndef COURSE\_DESIGN\_QUEUE\_H

#define COURSE\_DESIGN\_QUEUE\_H

#include <mm\_malloc.h>

#include "lexer.h"

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASIBLE -1

typedef token\_info Q\_ElemType;

typedef int status;

//队列定义（有头结点）

typedef struct QueueNode

{

    Q\_ElemType data;

    QueueNode \*next;

} QueueNode; //队列链表结点

typedef struct Queue

{

    QueueNode \*begin;

    QueueNode \*end;

    int num; //队列元素个数

} Queue;     //队列

void iniQueue(Queue &Q);

status enQueue(Queue &Q, Q\_ElemType e);

status deQueue(Queue &Q, Q\_ElemType &e);

int Q\_num(Queue Q);

Q\_ElemType Q\_get\_first(Queue Q);

Q\_ElemType Q\_get\_last(Queue Q);

#endif //COURSE\_DESIGN\_QUEUE\_H

### 附录14 Stack.cpp：

#include "Stack.h"

//栈函数

/\*\*

 \* @description: 初始化栈

 \* @param {Stack} &S

 \* @return

 \*/

void iniStack(Stack &S) {

  S.begin = (StackNode \*)malloc(sizeof(StackNode));

  S.begin->next = NULL;

  S.num = 0;

  return;

}

/\*\*

 \* @description: 将元素e入栈

 \* @param {Stack} &S

 \* @param {S\_ElemType} e

 \* @return 成功返回OK，失败返回ERROR

 \*/

status S\_push(Stack &S, S\_ElemType e) {

  StackNode \*new\_node = (StackNode \*)malloc(sizeof(StackNode));

  if (!new\_node)

    return ERROR;

  new\_node->next = S.begin;

  new\_node->data = e;

  S.begin = new\_node;

  S.num += 1;

  return OK;

}

/\*\*

 \* @description: 将栈顶元素出栈并赋值给e

 \* @param {Stack} &S

 \* @param {S\_ElemType} &e

 \* @return 成功返回OK，失败返回ERROR

 \*/

status S\_pop(Stack &S, S\_ElemType &e) {

  if (S.num == 0)

    return ERROR;

  e = S.begin->data;

  StackNode \*new\_begin = S.begin->next;

  free(S.begin);

  S.begin = new\_begin;

  S.num -= 1;

  return OK;

}

/\*\*

 \* @description: 查询栈中元素个数

 \* @param {Stack} S

 \* @return {int} 元素个数

 \*/

int S\_num(Stack S) { return S.num; }

/\*\*

 \* @description: 判断栈是否为空

 \* @param {Stack} S

 \* @return 空则返回1，非空返回0

 \*/

int isEmptyStack(Stack S) {

  if (S.num == 0)

    return 1;

  else

    return 0;

}

/\*\*

 \* @description:获取栈顶元素

 \* @param {Stack} S

 \* @return {\*}

 \*/

S\_ElemType S\_get\_top(Stack S) {

  if (S.num != 0)

    return S.begin->data;

}

/\*\*

 \* @description: 清空栈

 \* @param {Stack} S

 \* @return

 \*/

void destroyStack(Stack S) {

  while (S\_num(S)) {

    S\_ElemType e;

    S\_pop(S, e);

  }

  return;

}

### 附录15 Stack.h：

#ifndef COURSE\_DESIGN\_STACK\_H

#define COURSE\_DESIGN\_STACK\_H

#include <mm\_malloc.h>

#include "AST.h"

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASIBLE -1

typedef p\_treeNode S\_ElemType;

typedef int status;

//栈定义（有头结点）

typedef struct StackNode

{

    S\_ElemType data;

    StackNode \*next;

} StackNode; //栈链表结点

typedef struct Stack

{

    StackNode \*begin;

    int num; //栈元素个数

} Stack;     //栈

void iniStack(Stack &S);

status S\_push(Stack &S, S\_ElemType e);

status S\_pop(Stack &S, S\_ElemType &e);

int S\_num(Stack S);

int isEmptyStack(Stack S);

S\_ElemType S\_get\_top(Stack S);

void destroyStack(Stack S);

#endif //COURSE\_DESIGN\_STACK\_H

### 附录16 Traverse.cpp：

#include "Traverse.h"

/\*\*

 \*

 \* @param tabs

 \*/

void printTabs(int tabs) {

  for (int i = 0; i < tabs; i++)

    printf("  ");

}

void printFunCall(p\_treeNode fun\_call) {

  printf("%s", fun\_call->firstChild->self\_value);

  printf("(");

  if (fun\_call->firstChild->nextBro) {

    for (p\_treeNode p = fun\_call->firstChild->nextBro->firstChild; p;) {

      printf("%s", p->self\_value);

      p = p->nextBro;

      if (p) {

        printf(",");

      }

    } // end of for

  }   // end of if

  printf(")");

} // end of printFunCall

void traverseExp(p\_treeNode p) {

  if (!p)

    return;

  else {

    if (strcmp(p->self\_value, "fun call") == 0) {

      printFunCall(p);

    } else {

      printf("(");

      traverseExp(p->firstChild);

      printf(" %s ", p->self\_value);

      if (p->firstChild)

        traverseExp(p->firstChild->nextBro);

      printf(")");

    } // end of else

    return;

  } // end of else

} // end of traverseExp

void traverse(p\_treeNode p, int tabs) {

  if (!p)

    return;

  if (strcmp(p->self\_value, "fun declaration") == 0 ||

      strcmp(p->self\_value, "fun announce") == 0) {

    //函数定义与声明

    printTabs(tabs);

    tabs++;

    printf("%s\n", p->self\_value);

    printTabs(tabs);

    printf("return type: %s\n", p->firstChild->firstChild->self\_value);

    printTabs(tabs);

    printf("fun name: %s\n", p->firstChild->nextBro->self\_value);

    printTabs(tabs);

    printf("params: \n");

    //        printTabs(++tabs);

    tabs++;

    if (!p->firstChild->nextBro->nextBro) {

      //无参数

      printTabs(++tabs);

      printf("void\n");

    } else {

      //有参数

      // param list

      traverse(p->firstChild->nextBro->nextBro->firstChild, tabs);

    }

    tabs--;

    if (strcmp(p->self\_value, "fun announce") == 0)

      return;

    // fun compound statement

    printTabs(tabs);

    printf("fun compound statement:\n");

    tabs++;

    traverse(p->firstChild->nextBro->nextBro->nextBro, tabs);

    return;

  } // end of if(fun\_dec or fun\_ann)

  else if (strcmp(p->self\_value, "param list") == 0) {

    //函数声明有参数的param list结点

    if (!p->firstChild)

      return;

    printTabs(tabs);

    printf("type specifier: %s ; param name:%s \n",

           p->firstChild->firstChild->firstChild->self\_value,

           p->firstChild->firstChild->nextBro->self\_value);

    if (p->firstChild)

      traverse(p->firstChild->nextBro, tabs);

    return;

  } // end of else if(param\_list)

  else if (strcmp(p->self\_value, "ex val declaration") == 0 ||

           strcmp(p->self\_value, "local val declaration") == 0) {

    //变量声明

    printTabs(tabs);

    printf("%s :\n", p->self\_value);

    tabs++;

    printTabs(tabs);

    printf("type specifier: %s\n", p->firstChild->firstChild->self\_value);

    traverse(p->firstChild->nextBro, tabs); // val list结点

    tabs--;

    return;

  } // end of else if(val\_dec)

  else if (strcmp(p->self\_value, "val list") == 0) {

    printTabs(tabs);

    printf("valid name: %s\n", p->firstChild->self\_value);

    traverse(p->firstChild->nextBro, tabs);

    return;

  } // end of else if(val\_list)

  else if (strcmp(p->self\_value, "fun compound statement") == 0) {

    //函数语句结点fun compound statement

    traverse(p->firstChild, tabs);

    if (p->firstChild)

      traverse(p->firstChild->nextBro, tabs);

    return;

  } // end of else if(fun\_cmpd\_statmt)

  else if (strcmp(p->self\_value, "statement") == 0) {

    // statement结点

    traverse(p->firstChild, tabs);

    return;

  } else if (strcmp(p->self\_value, "compound statement") == 0) {

    // compound statement结点

    traverse(p->firstChild, tabs);

    if (p->firstChild)

      traverse(p->firstChild->nextBro, tabs);

    return;

  } else if (strcmp(p->self\_value, "selection statement") == 0) {

    printTabs(tabs);

    printf("selection statement: \n");

    tabs++;

    traverse(p->firstChild, tabs);

    traverse(p->firstChild->nextBro, tabs);

    tabs--;

    return;

  } else if (strcmp(p->self\_value, "condition expression") == 0) {

    printTabs(tabs);

    printf("condition expression: \n");

    traverse(p->firstChild, tabs);

    return;

  } else if (strcmp(p->self\_value, "if else statement") == 0) {

    // tabs++;

    traverse(p->firstChild, tabs);

    traverse(p->firstChild->nextBro, tabs);

    // tabs--;

    return;

  } else if (strcmp(p->self\_value, "if statement") == 0) {

    printTabs(tabs);

    printf("if statement:\n");

    tabs++;

    traverse(p->firstChild, tabs);

    tabs--;

    return;

  } else if (strcmp(p->self\_value, "else statement") == 0) {

    printTabs(tabs);

    printf("else statement:\n");

    tabs++;

    traverse(p->firstChild, tabs);

    tabs--;

    return;

  } else if (strcmp(p->self\_value, "while statement") == 0) {

    printTabs(tabs);

    printf("while statement:\n");

    tabs++;

    printTabs(tabs);

    printf("condition expression:\n");

    traverse(p->firstChild, tabs);

    printTabs(tabs);

    printf("iteration statement:\n");

    traverse(p->firstChild->nextBro, tabs);

    tabs--;

    return;

  } // end of else if(while\_statmt)

  else if (strcmp(p->self\_value, "for statement") == 0) {

    printTabs(tabs);

    printf("for statement:\n");

    printf("condition val initial:\n");

    traverse(p->firstChild->firstChild, tabs + 1);

    printTabs(tabs);

    printf("condition expression:\n");

    traverse(p->firstChild->nextBro, tabs + 1);

    printTabs(tabs);

    printf("condition iteration:\n");

    traverse(p->firstChild->nextBro->nextBro, tabs + 1);

    printTabs(tabs);

    printf("iteration statement:\n");

    traverse(p->firstChild->nextBro->nextBro->nextBro, tabs + 1);

    return;

  } // end of else if(for\_statmt)

  else if (strcmp(p->self\_value, "break statement") == 0) {

    printTabs(tabs);

    printf("break statement\n");

    return;

  } else if (strcmp(p->self\_value, "continue statement") == 0) {

    printTabs(tabs);

    printf("continue statement\n");

    return;

  } else if (strcmp(p->self\_value, "return statement") == 0) {

    printTabs(tabs);

    printf("return statement\n");

    printTabs(tabs + 1);

    printf("return expression:\n");

    traverse(p->firstChild, tabs + 1);

    return;

  } else if (strcmp(p->self\_value, "expression") == 0) {

    printTabs(tabs);

    printf("expression:\n");

    printTabs(tabs + 1);

    traverseExp(p->firstChild);

    printf("\n");

    return;

  } else {

    traverse(p->firstChild, tabs);

    if (p->firstChild)

      traverse(p->firstChild->nextBro, tabs);

    return;

  }

} // end of traverse

### 附录17 Traverse.h：

#ifndef COURSE\_DESIGN\_TRAVERSE\_H

#define COURSE\_DESIGN\_TRAVERSE\_H

#include "parsing.h"

void printTabs(int tabs);

void printFunCall(p\_treeNode fun\_call);

void traverseExp(p\_treeNode p);

void traverse(p\_treeNode p, int tabs);

#endif //COURSE\_DESIGN\_TRAVERSE\_H