華中科技大學

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

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处理工具,比如源程序浏览器、智能编辑器、语言翻译器等。通常是作为编译器或解释器的组件出现的，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、抽象语法树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

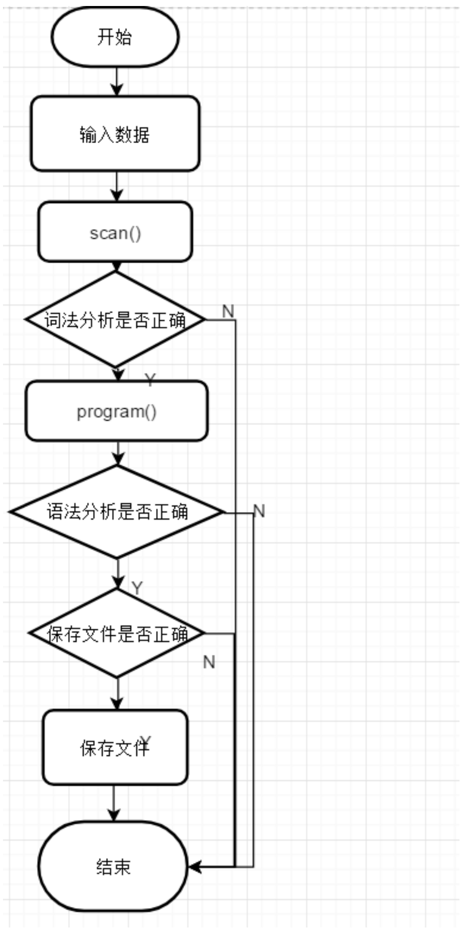


图 2.1 系统总体流程图

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

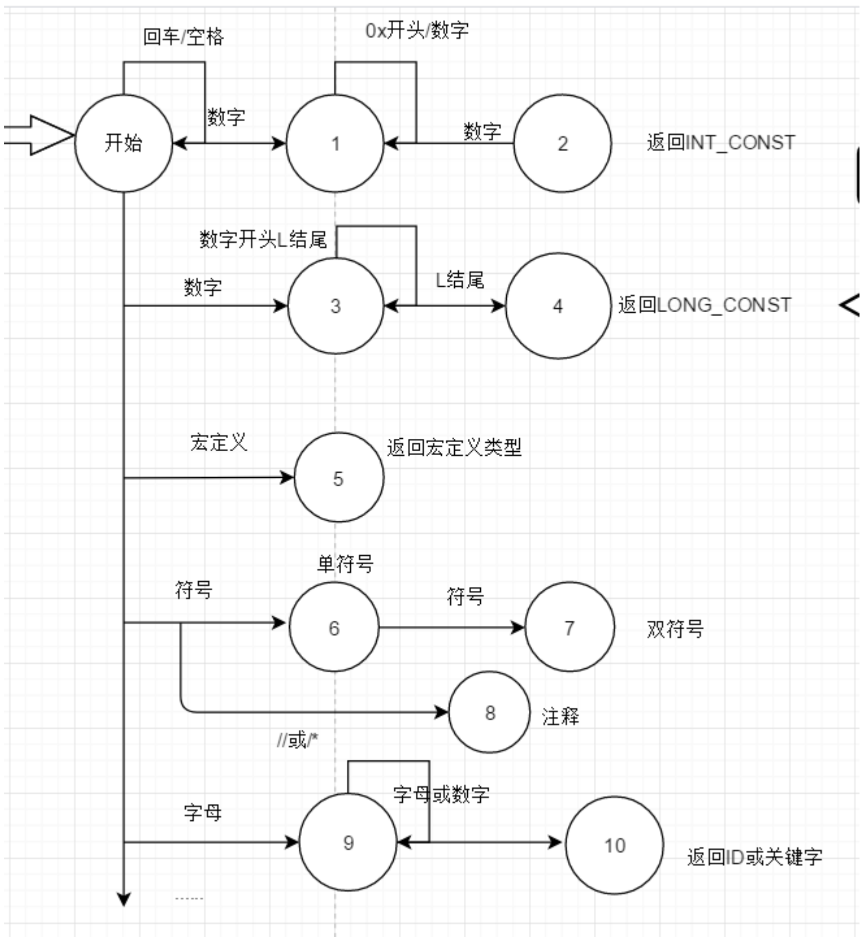


图 3.1DFA 转换图

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

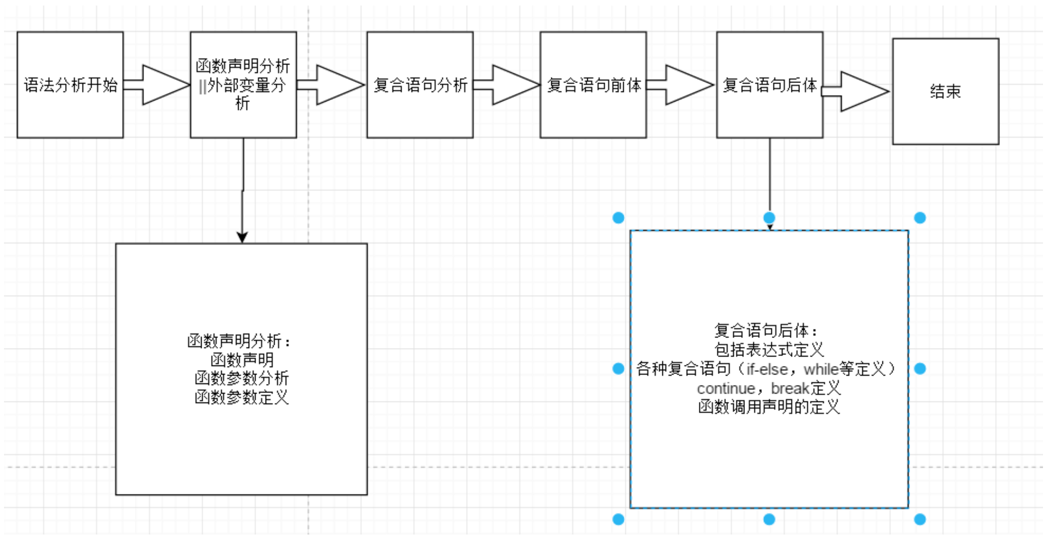


图 3.2 语法分析设计图

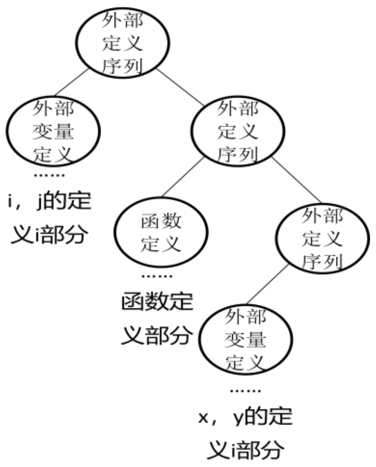


图 3.3 AST 语法树

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

4.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

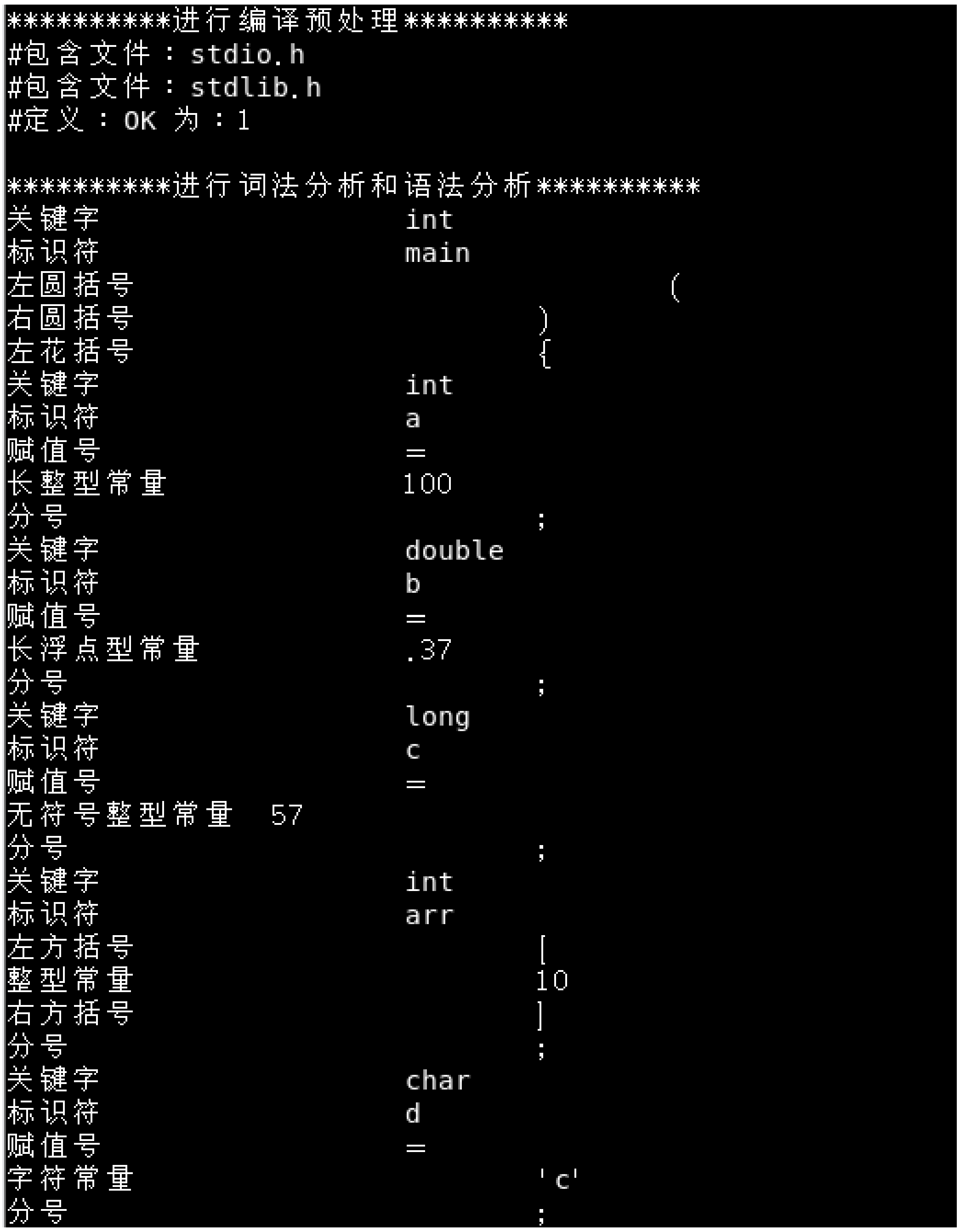
4.3.1 功能测试

测试样例 1：

输入代码：

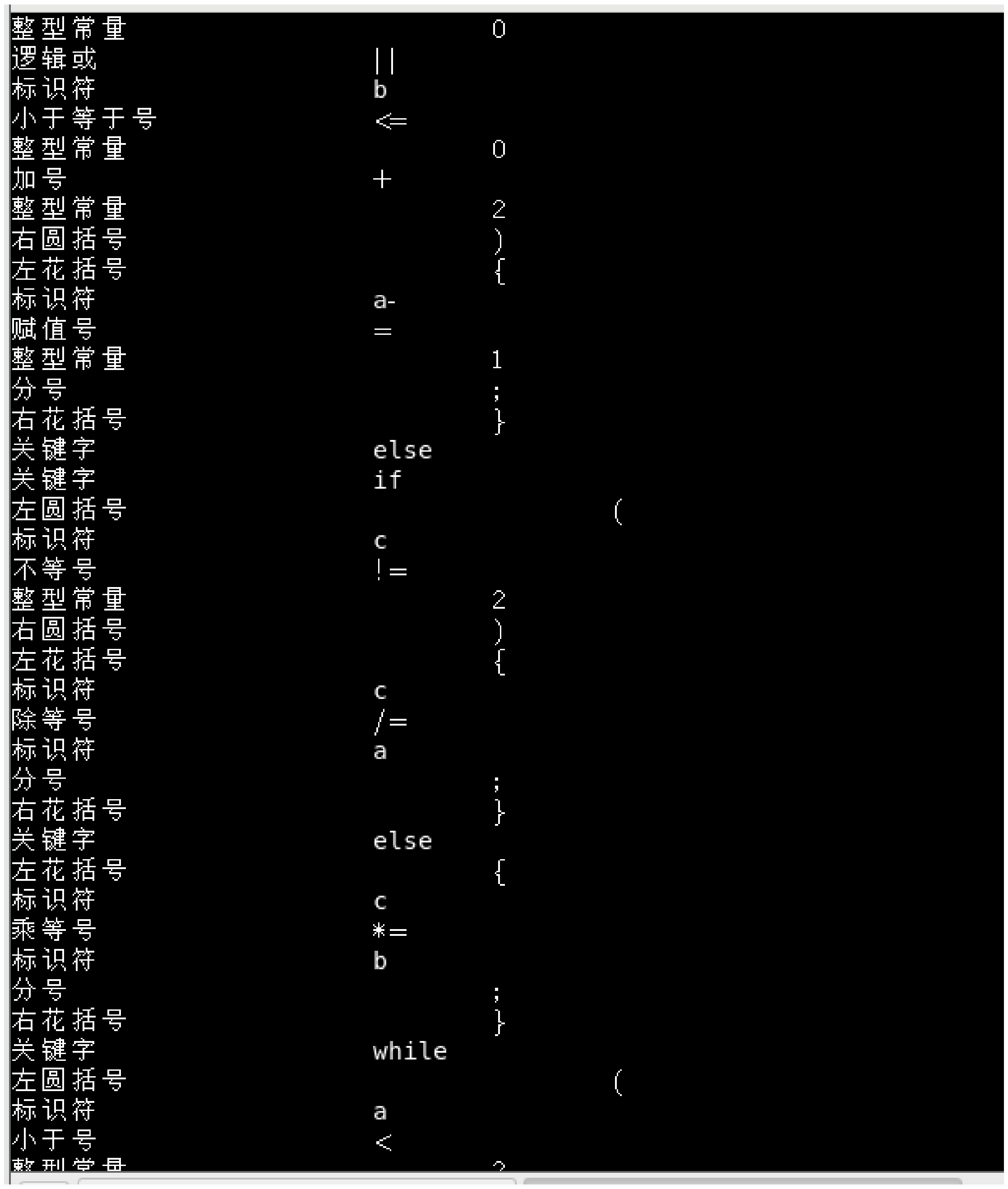
```
#include<stdio.h>
#include<stdlib.h>
#define OK 1

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：



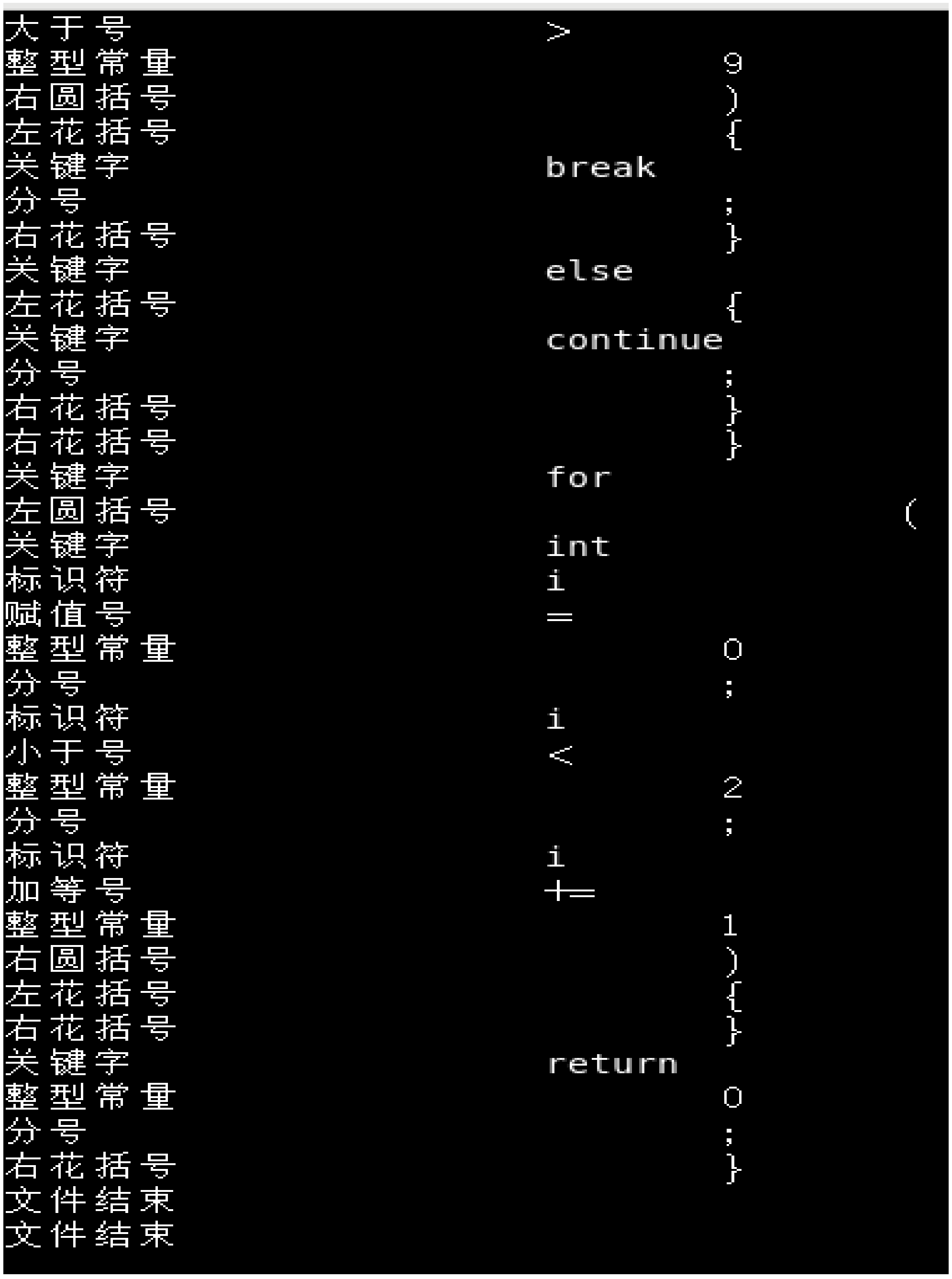
```
**********进行编译预处理**********
#包含文件：stdio.h
#包含文件：stdlib.h
#定义：OK 为：1

**********进行词法分析和语法分析**********
关键字                  int
标识符                  main
左圆括号                                ( 
右圆括号                        )
左花括号                        {
关键字                  int
标识符                  a
赋值号                  =
长整型常量              100
分号                            ;
关键字                  double
标识符                  b
赋值号                  =
长浮点型常量            .37
分号                            ;
关键字                  long
标识符                  c
赋值号                  =
无符号整型常量   57
分号                            ;
关键字                  int
标识符                  arr
左方括号                        [
整型常量                        10
右方括号                        ]
分号                            ;
关键字                  char
标识符                  d
赋值号                  =
字符常量                        'c'
分号                            ;
```

图 4.1 词法分析和语法分析 1-1



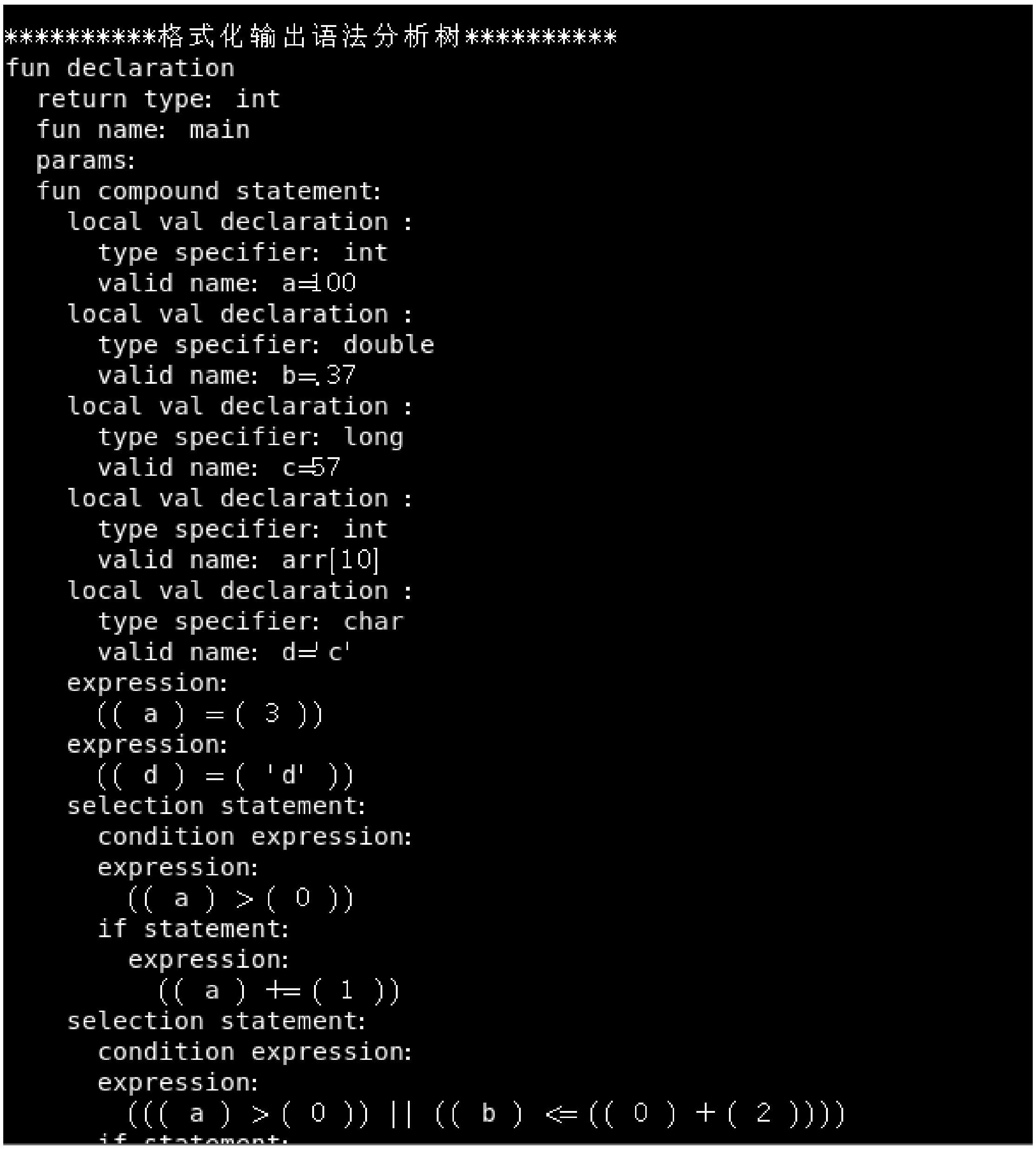
```
整型常量                          0
逻辑或                    ||
标识符                    b
小于等于号                <=
整型常量                          0
加号                      +
整型常量                          2
右圆括号                          )
左花括号                          {
标识符                    a-
赋值号                    =
整型常量                          1
分号                              ;
右花括号                          }
关键字                    else
关键字                    if
左圆括号                                  (
标识符                    c
不等号                    !=
整型常量                          2
右圆括号                          )
左花括号                          {
标识符                    c
除等号                    /=
标识符                    a
分号                              ;
右花括号                          }
关键字                    else
左花括号                          {
标识符                    c
乘等号                    *=
标识符                    b
分号                              ;
右花括号                          }
关键字                    while
左圆括号                                  (
标识符                    a
小于号                    <
整型常量                          2
```

图 4.2 词法分析和语法分析 1-2



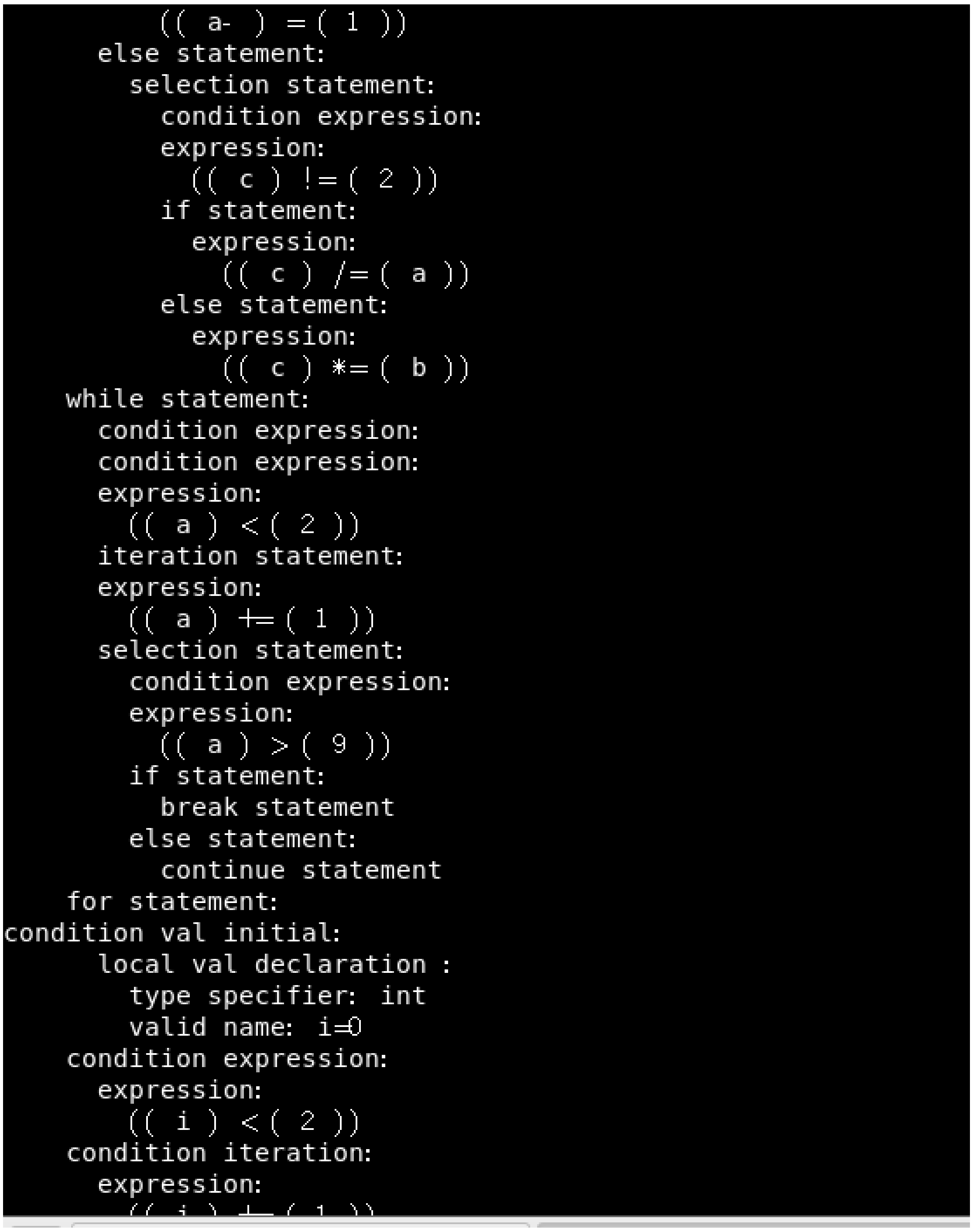
```
大于号                  >
整型常量                        9
右圆括号                        )
左花括号                        {
关键字                  break
分号                            ;
右花括号                        }
关键字                  else
左花括号                        {
关键字                  continue
分号                            ;
右花括号                        }
右花括号                        }
关键字                  for
左圆括号                                        (
关键字                  int
标识符                  i
赋值号                  =
整型常量                        0
分号                            ;
标识符                  i
小于号                  <
整型常量                        2
分号                            ;
标识符                  i
加等号                  +=
整型常量                        1
右圆括号                        )
左花括号                        {
右花括号                        }
关键字                  return
整型常量                        0
分号                            ;
右花括号                        }
文件结束
文件结束
```

图 4.3 词法分析和语法分析 1-3



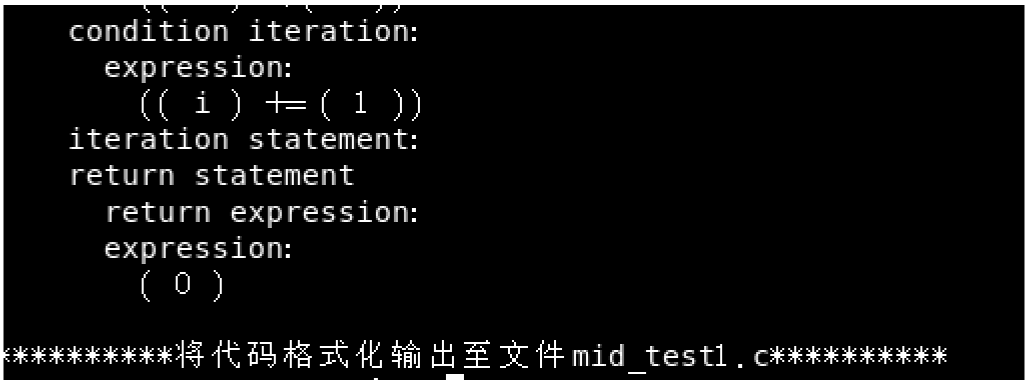
```
**********格式化输出语法分析树**********
fun declaration
  return type: int
  fun name: main
  params:
  fun compound statement:
    local val declaration :
      type specifier: int
      valid name: a=100
    local val declaration :
      type specifier: double
      valid name: b=.37
    local val declaration :
      type specifier: long
      valid name: c=57
    local val declaration :
      type specifier: int
      valid name: arr[10]
    local val declaration :
      type specifier: char
      valid name: d=' c'
    expression:
      (( a ) =( 3 ))
    expression:
      (( d ) =( 'd' ))
    selection statement:
      condition expression:
      expression:
        (( a ) >( 0 ))
      if statement:
        expression:
          (( a ) +=( 1 ))
    selection statement:
      condition expression:
      expression:
        ((( a ) >( 0 )) || (( b ) <=(( 0 ) +( 2 ))))
```

图 4. 4 格式化处理 1-1

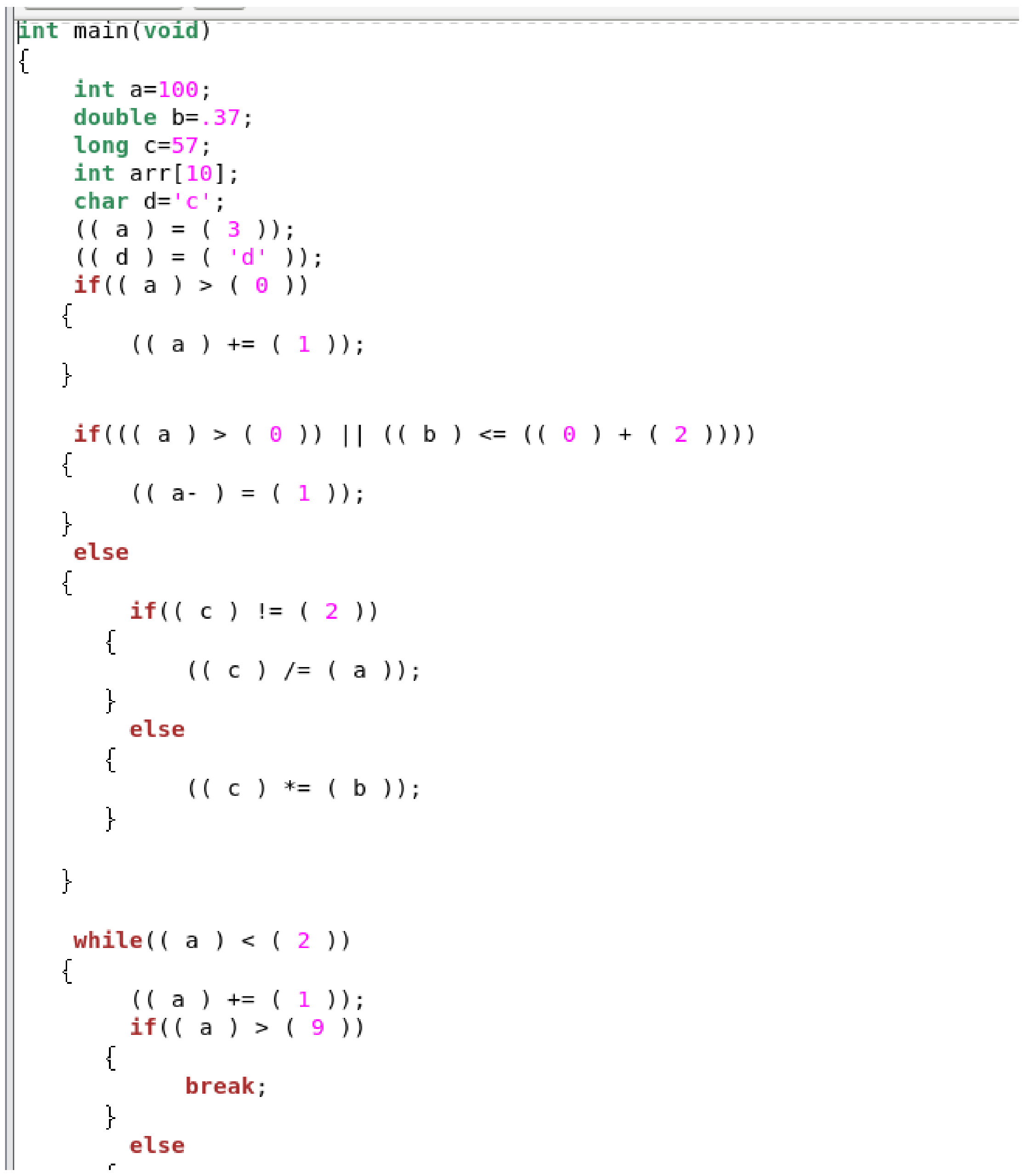


```
          (( a- ) =( 1 ))
      else statement:
        selection statement:
          condition expression:
          expression:
            (( c ) !=( 2 ))
          if statement:
            expression:
              (( c ) /=( a ))
          else statement:
            expression:
              (( c ) *=( b ))
    while statement:
      condition expression:
      condition expression:
      expression:
        (( a ) <( 2 ))
      iteration statement:
      expression:
        (( a ) +=( 1 ))
      selection statement:
        condition expression:
        expression:
          (( a ) >( 9 ))
        if statement:
          break statement
        else statement:
          continue statement
    for statement:
condition val initial:
      local val declaration :
        type specifier: int
        valid name: i=0
    condition expression:
      expression:
        (( i ) <( 2 ))
    condition iteration:
      expression:
```

图 4.5 格式化处理 1-2

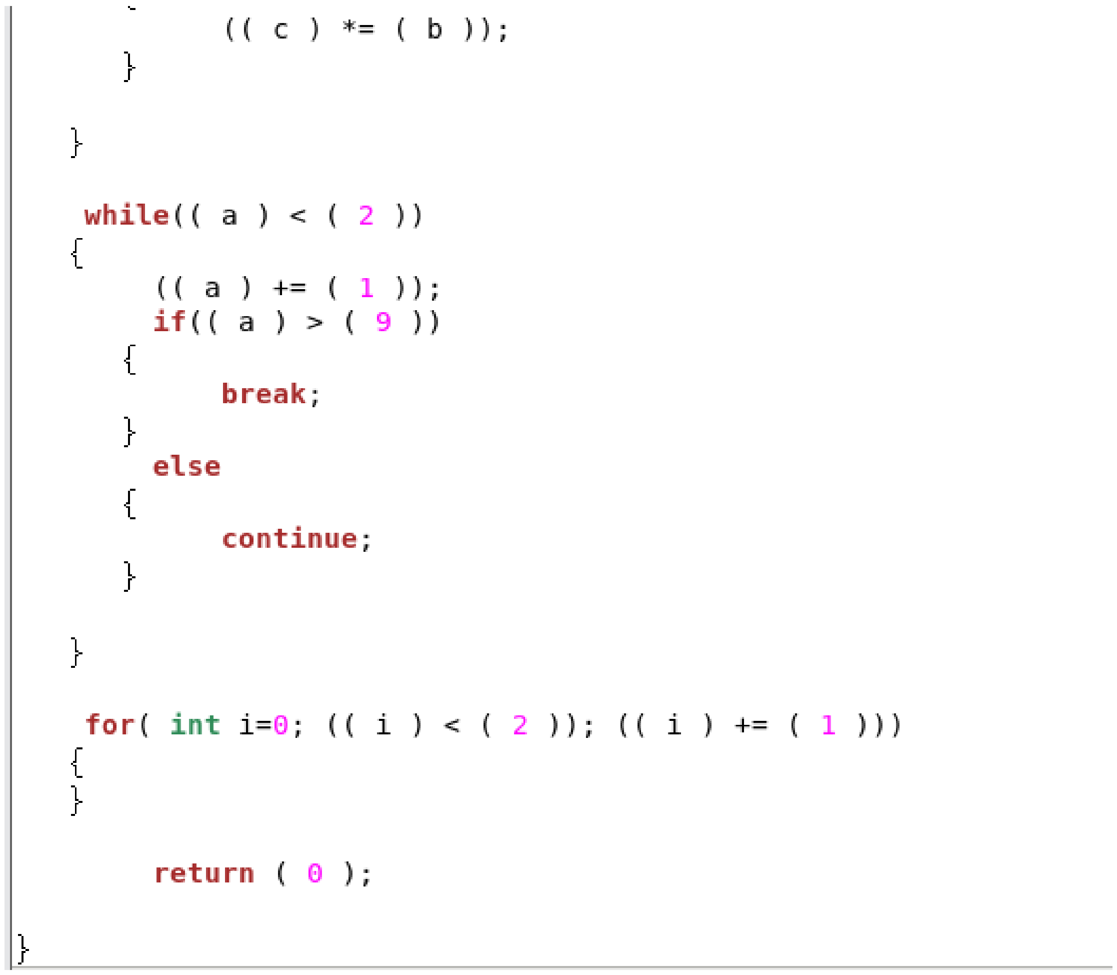


```
    condition iteration:
      expression:
        (( i ) +=( 1 ))
    iteration statement:
    return statement
      return expression:
      expression:
        ( 0 )

**********将代码格式化输出至文件mid_test1.c**********
```



```
int main(void)
{
    int a=100;
    double b=.37;
    long c=57;
    int arr[10];
    char d='c';
    (( a ) = ( 3 ));
    (( d ) = ( 'd' ));
    if(( a ) > ( 0 ))
   {
        (( a ) += ( 1 ));
   }

    if((( a ) > ( 0 )) || (( b ) <= (( 0 ) + ( 2 ))))
   {
        (( a- ) = ( 1 ));
   }
    else
   {
        if(( c ) != ( 2 ))
      {
            (( c ) /= ( a ));
      }
        else
      {
            (( c ) *= ( b ));
      }

   }

    while(( a ) < ( 2 ))
   {
        (( a ) += ( 1 ));
        if(( a ) > ( 9 ))
      {
            break;
      }
        else
```

图 4. 7 文件保存 1-1



```
            (( c ) *= ( b ));
        }

    }

     while(( a ) < ( 2 ))
    {
          (( a ) += ( 1 ));
          if(( a ) > ( 9 ))
        {
            break;
        }
          else
        {
            continue;
        }

    }

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

          return ( 0 );

}
```

图 4.8 文件保存 1-2

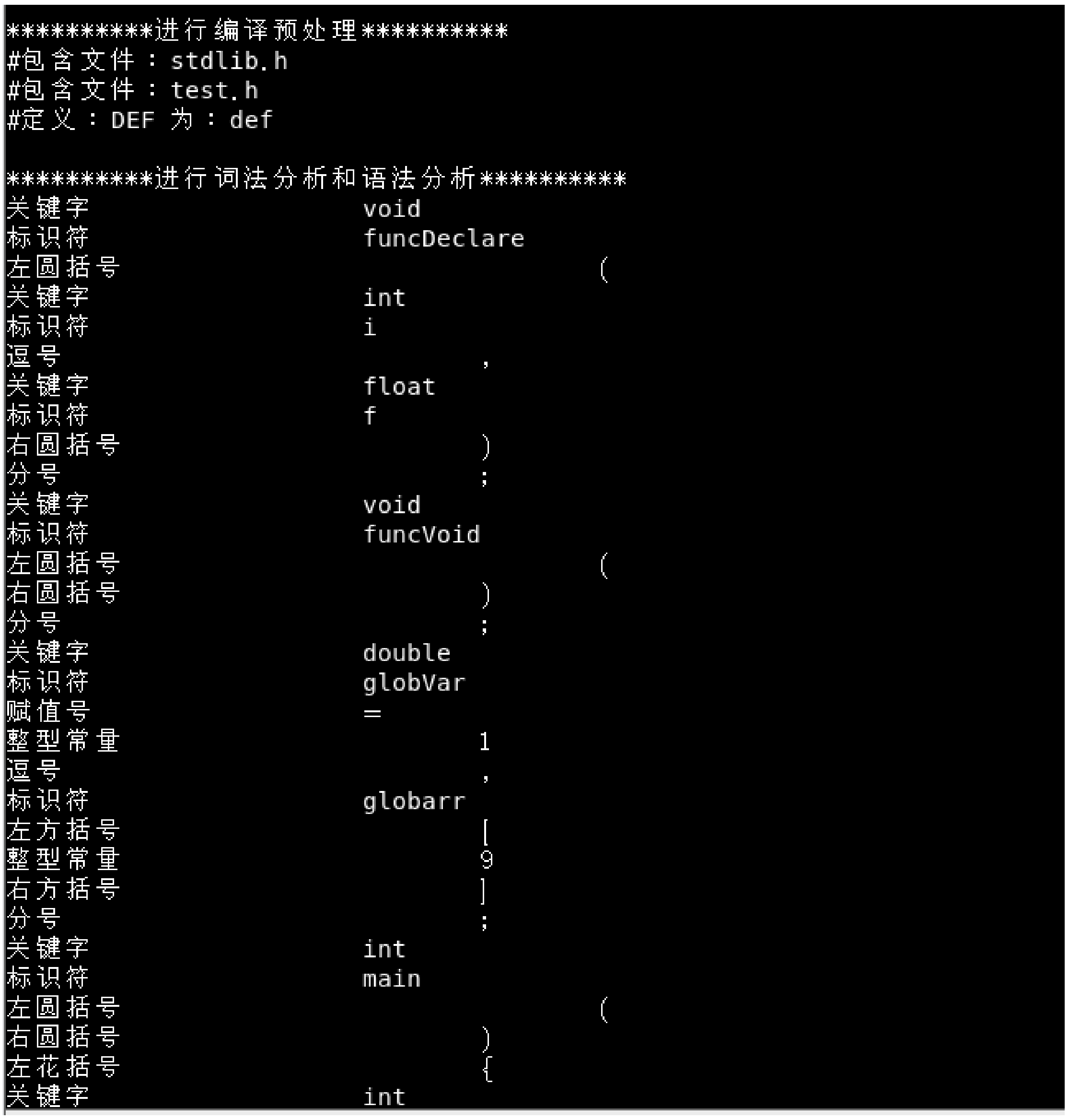
测试样例 2：

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
 *
 */
```

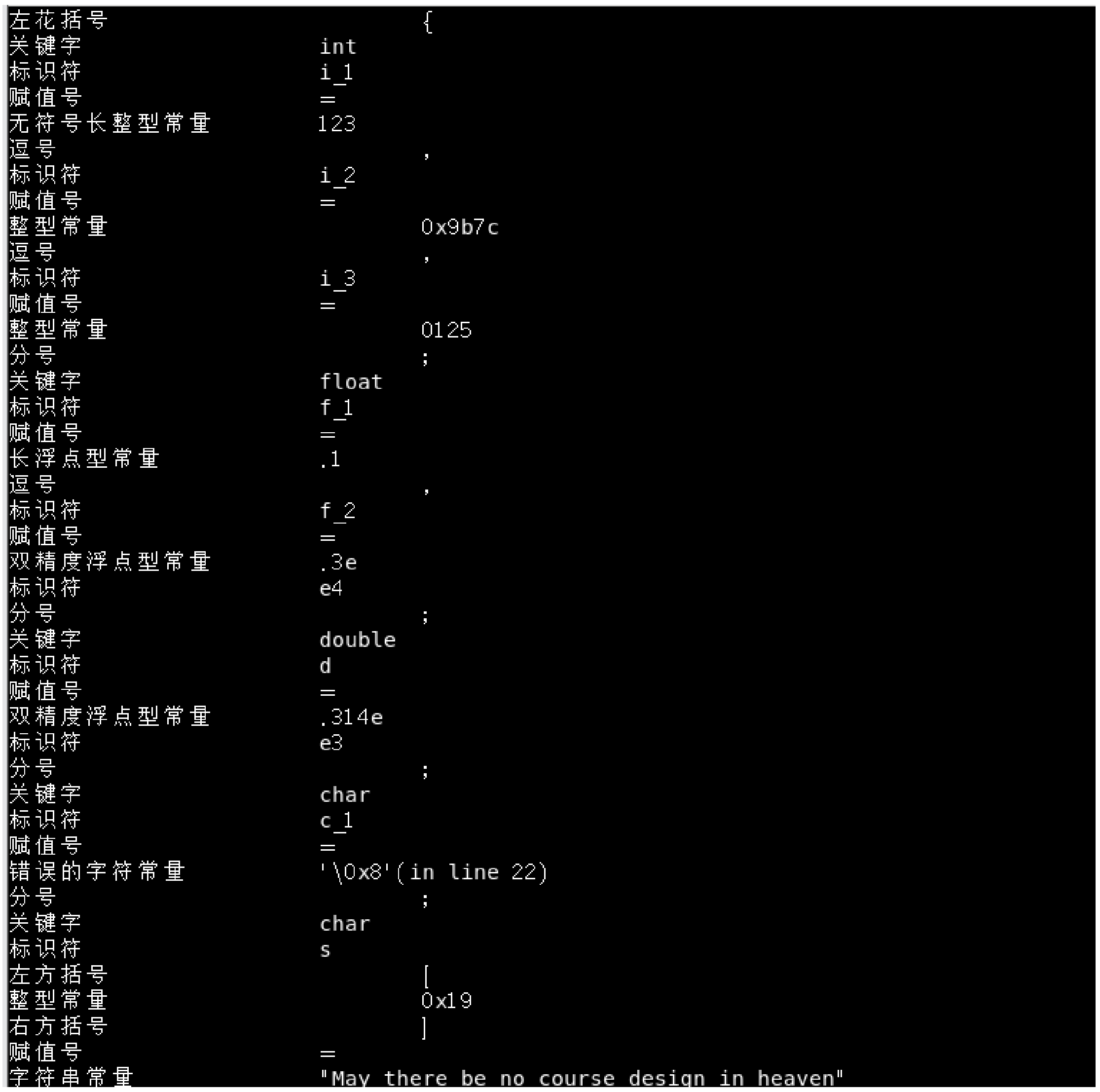
```
int main()
{
      // 10 16 8 l u
       int i_1 = 123ul, i_2 = 0x9b7c, i_3 = 0125;
       float f_1 = .1l, f_2 = .3e4;

       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：



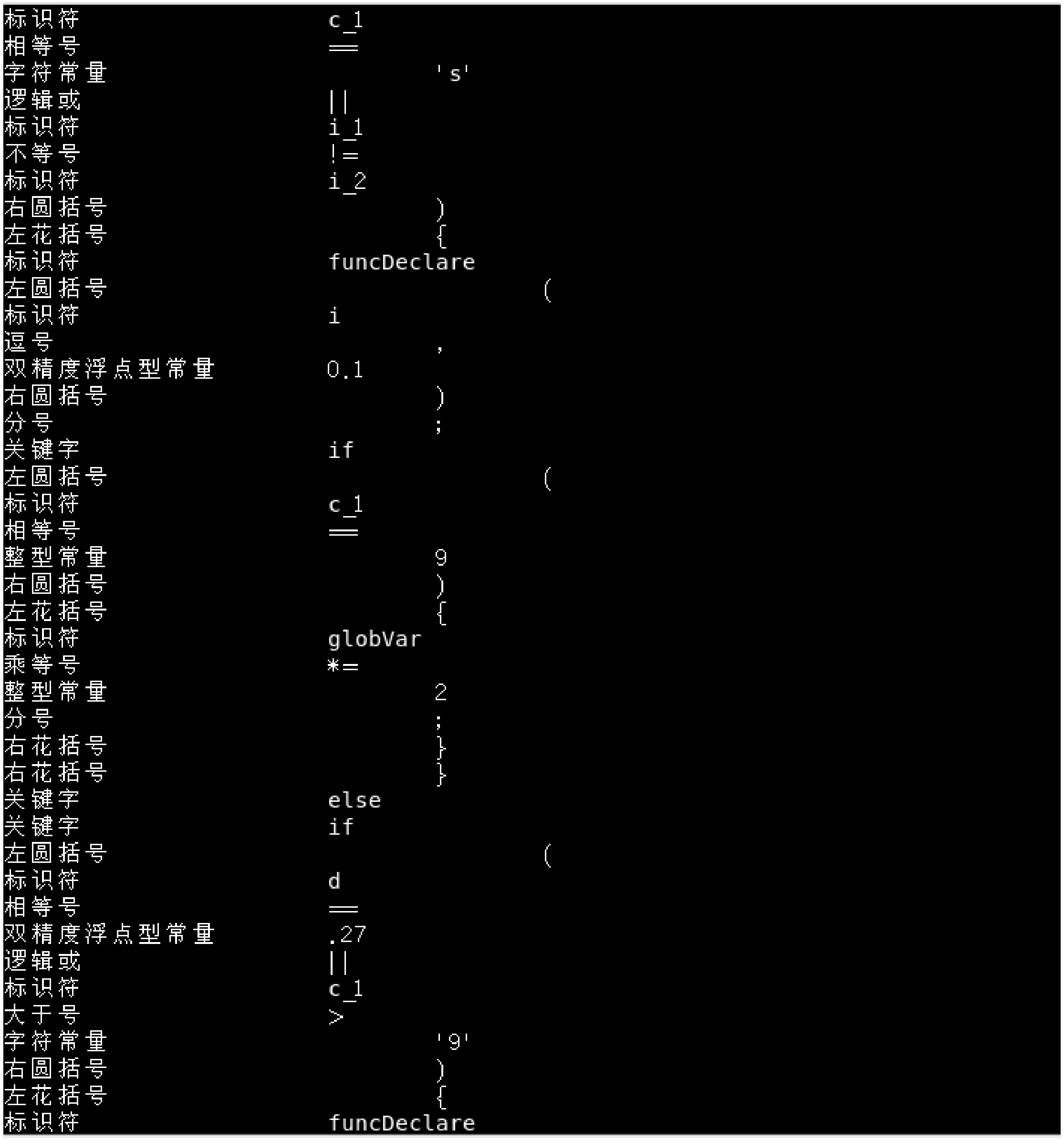
```
**********进行编译预处理**********
#包含文件：stdlib.h
#包含文件：test.h
#定义：DEF 为：def

**********进行词法分析和语法分析**********
关键字                  void
标识符                  funcDeclare
左圆括号                               (
关键字                  int
标识符                  i
逗号                            ,
关键字                  float
标识符                  f
右圆括号                        )
分号                            ;
关键字                  void
标识符                  funcVoid
左圆括号                               (
右圆括号                        )
分号                            ;
关键字                  double
标识符                  globVar
赋值号                  =
整型常量                        1
逗号                            ,
标识符                  globarr
左方括号                        [
整型常量                        9
右方括号                        ]
分号                            ;
关键字                  int
标识符                  main
左圆括号                               (
右圆括号                        )
左花括号                        {
关键字                  int
```

图 4.9 词法分析和语法分析 2-1



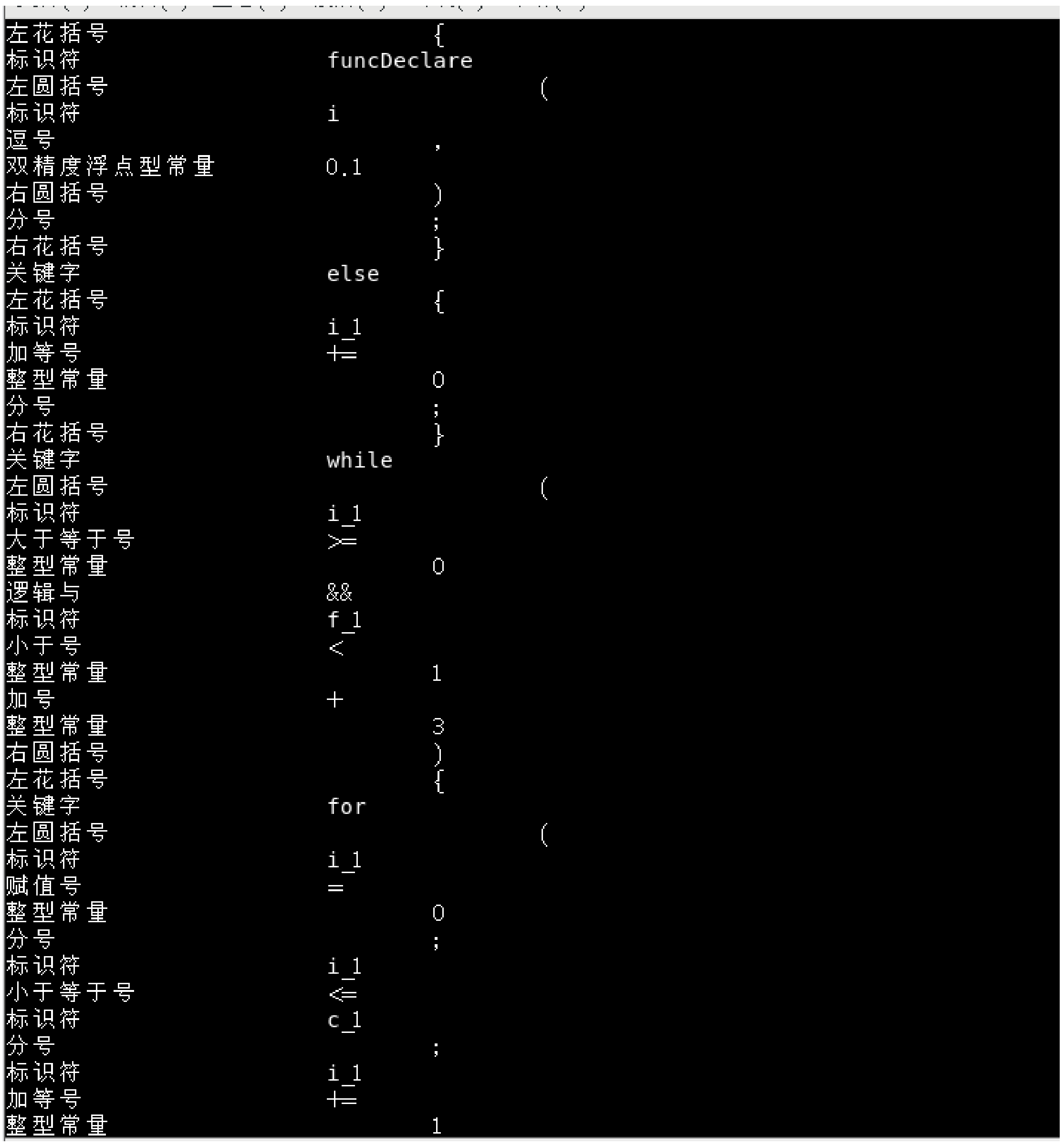
```
左花括号                        {
关键字                  int
标识符                  i_1
赋值号                  =
无符号长整型常量        123
逗号                            ,
标识符                  i_2
赋值号                  =
整型常量                        0x9b7c
逗号                            ,
标识符                  i_3
赋值号                  =
整型常量                        0125
分号                            ;
关键字                  float
标识符                  f_1
赋值号                  =
长浮点型常量            .1
逗号                            ,
标识符                  f_2
赋值号                  =
双精度浮点型常量        .3e
标识符                  e4
分号                            ;
关键字                  double
标识符                  d
赋值号                  =
双精度浮点型常量        .314e
标识符                  e3
分号                            ;
关键字                  char
标识符                  c_1
赋值号                  =
错误的字符常量          '\0x8'(in line 22)
分号                            ;
关键字                  char
标识符                  s
左方括号                        [
整型常量                        0x19
右方括号                        ]
赋值号                  =
字符串常量              "May there be no course design in heaven"
```

图 4.10 词法分析和语法分析 2-2



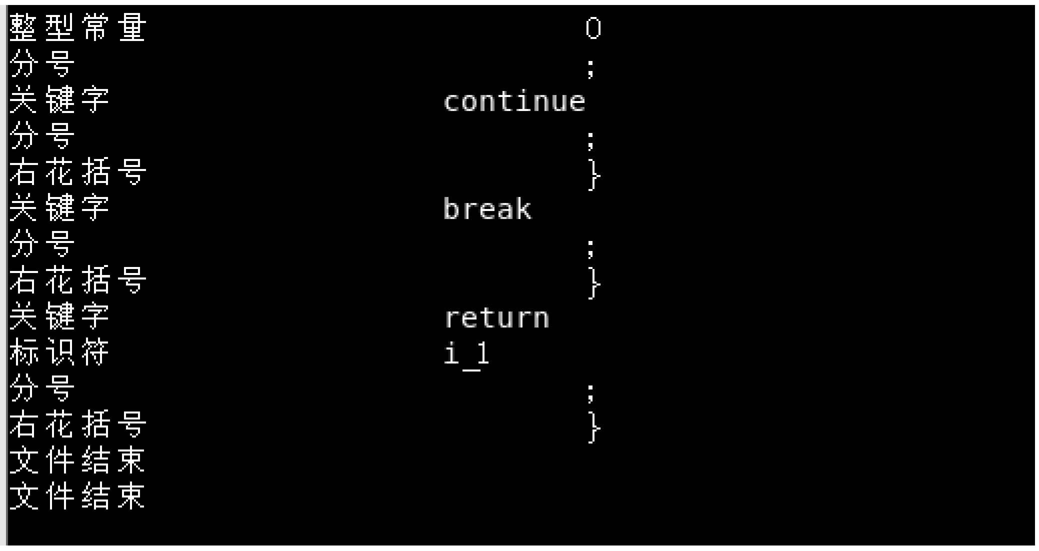
```
标识符                  c_1
相等号                  ==
字符常量                        's'
逻辑或                  ||
标识符                  i_1
不等号                  !=
标识符                  i_2
右圆括号                        )
左花括号                        {
标识符                  funcDeclare
左圆括号                                (
标识符                  i
逗号                            ,
双精度浮点型常量        0.1
右圆括号                        )
分号                            ;
关键字                  if
左圆括号                                (
标识符                  c_1
相等号                  ==
整型常量                        9
右圆括号                        )
左花括号                        {
标识符                  globVar
乘等号                  *=
整型常量                        2
分号                            ;
右花括号                        }
右花括号                        }
关键字                  else
关键字                  if
左圆括号                                (
标识符                  d
相等号                  ==
双精度浮点型常量        .27
逻辑或                  ||
标识符                  c_1
大于号                  >
字符常量                        '9'
右圆括号                        )
左花括号                        {
标识符                  funcDeclare
```

图 4.11 词法分析和语法分析 2-3



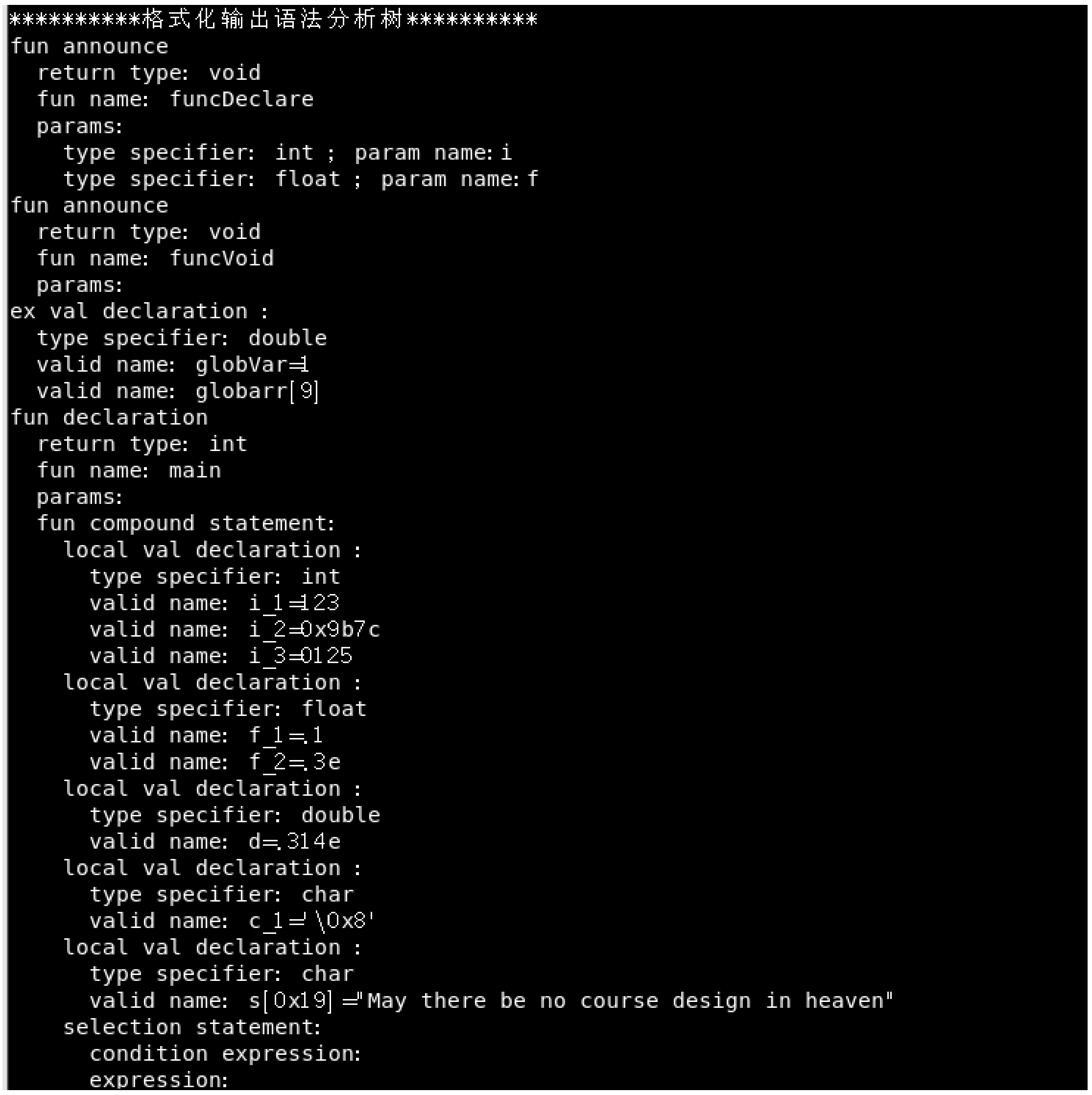
```
左花括号                        {
标识符                  funcDeclare
左圆括号                                ( 
标识符                  i
逗号                            ,
双精度浮点型常量        0.1
右圆括号                        )
分号                            ;
右花括号                        }
关键字                  else
左花括号                        {
标识符                  i_1
加等号                  +=
整型常量                        0
分号                            ;
右花括号                        }
关键字                  while
左圆括号                                (
标识符                  i_1
大于等于号              >=
整型常量                        0
逻辑与                  &&
标识符                  f_1
小于号                  <
整型常量                        1
加号                    +
整型常量                        3
右圆括号                        )
左花括号                        {
关键字                  for
左圆括号                                (
标识符                  i_1
赋值号                  =
整型常量                        0
分号                            ;
标识符                  i_1
小于等于号              <=
标识符                  c_1
分号                            ;
标识符                  i_1
加等号                  +=
整型常量                        1
```

图 4.12 词法分析和语法分析 2-4



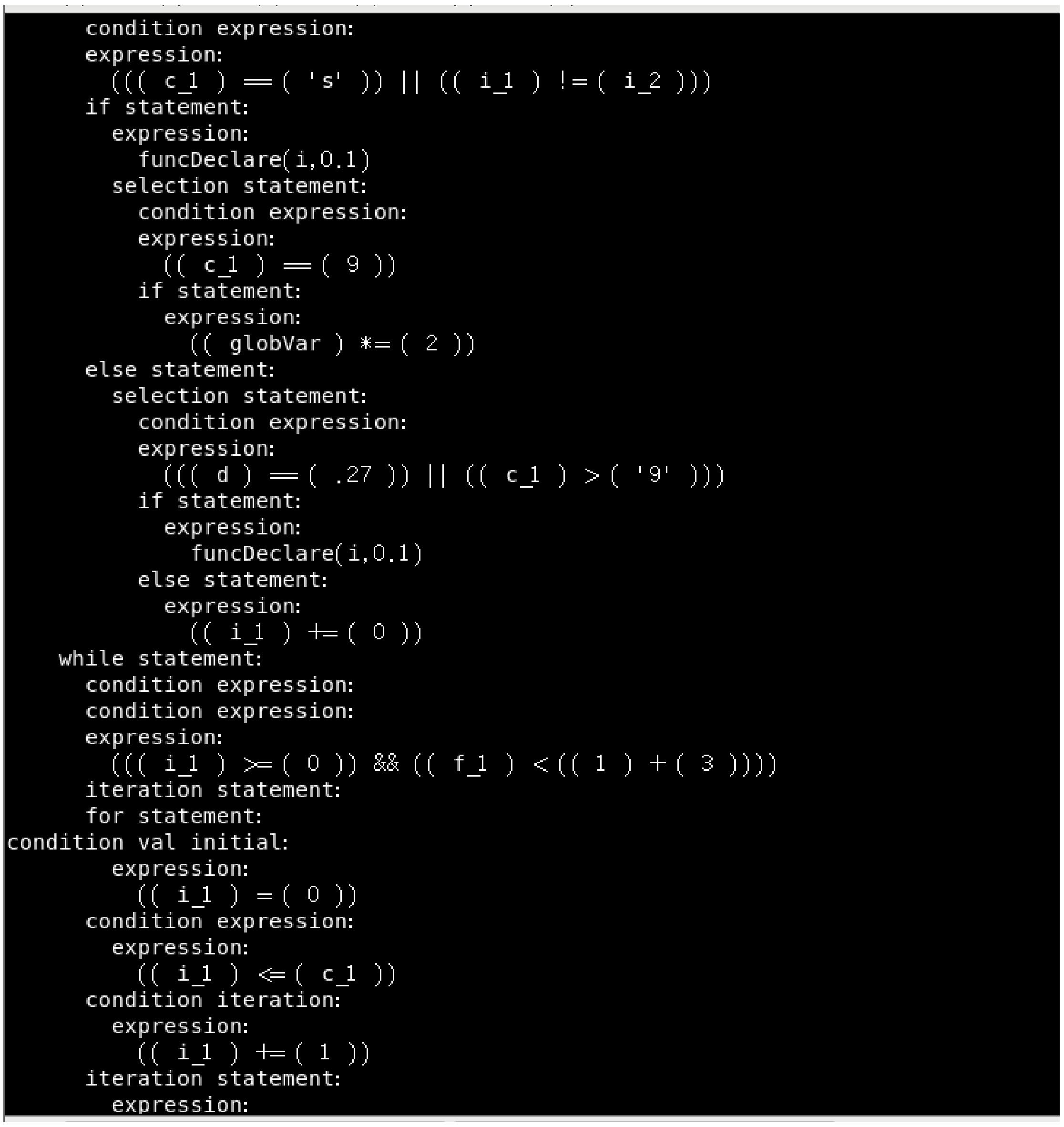
```
整型常量                        0
分号                            ;
关键字                  continue
分号                            ;
右花括号                        }
关键字                  break
分号                            ;
右花括号                        }
关键字                  return
标识符                  i_1
分号                            ;
右花括号                        }
文件结束
文件结束
```

图 4.13 词法分析和语法分析 2-5



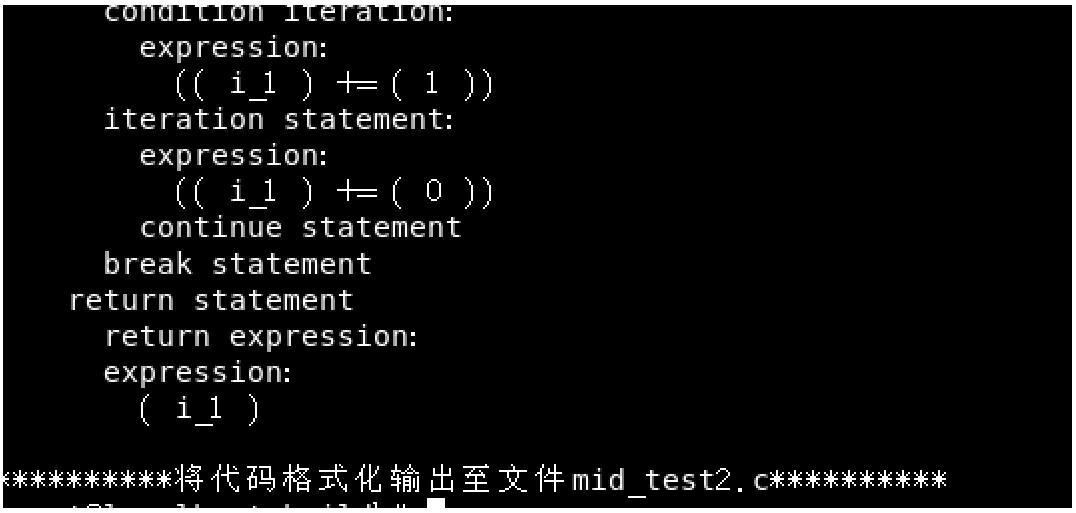
```
**********格式化输出语法分析树**********
fun announce
  return type: void
  fun name: funcDeclare
  params:
    type specifier: int ; param name:i
    type specifier: float ; param name:f
fun announce
  return type: void
  fun name: funcVoid
  params:
ex val declaration :
  type specifier: double
  valid name: globVar=1
  valid name: globarr[9]
fun declaration
  return type: int
  fun name: main
  params:
  fun compound statement:
    local val declaration :
      type specifier: int
      valid name: i_1=123
      valid name: i_2=0x9b7c
      valid name: i_3=0125
    local val declaration :
      type specifier: float
      valid name: f_1=.1
      valid name: f_2=.3e
    local val declaration :
      type specifier: double
      valid name: d=.314e
    local val declaration :
      type specifier: char
      valid name: c_1='\0x8'
    local val declaration :
      type specifier: char
      valid name: s[0x19]="May there be no course design in heaven"
    selection statement:
      condition expression:
      expression:
```

图 4.14 格式化处理 2-1



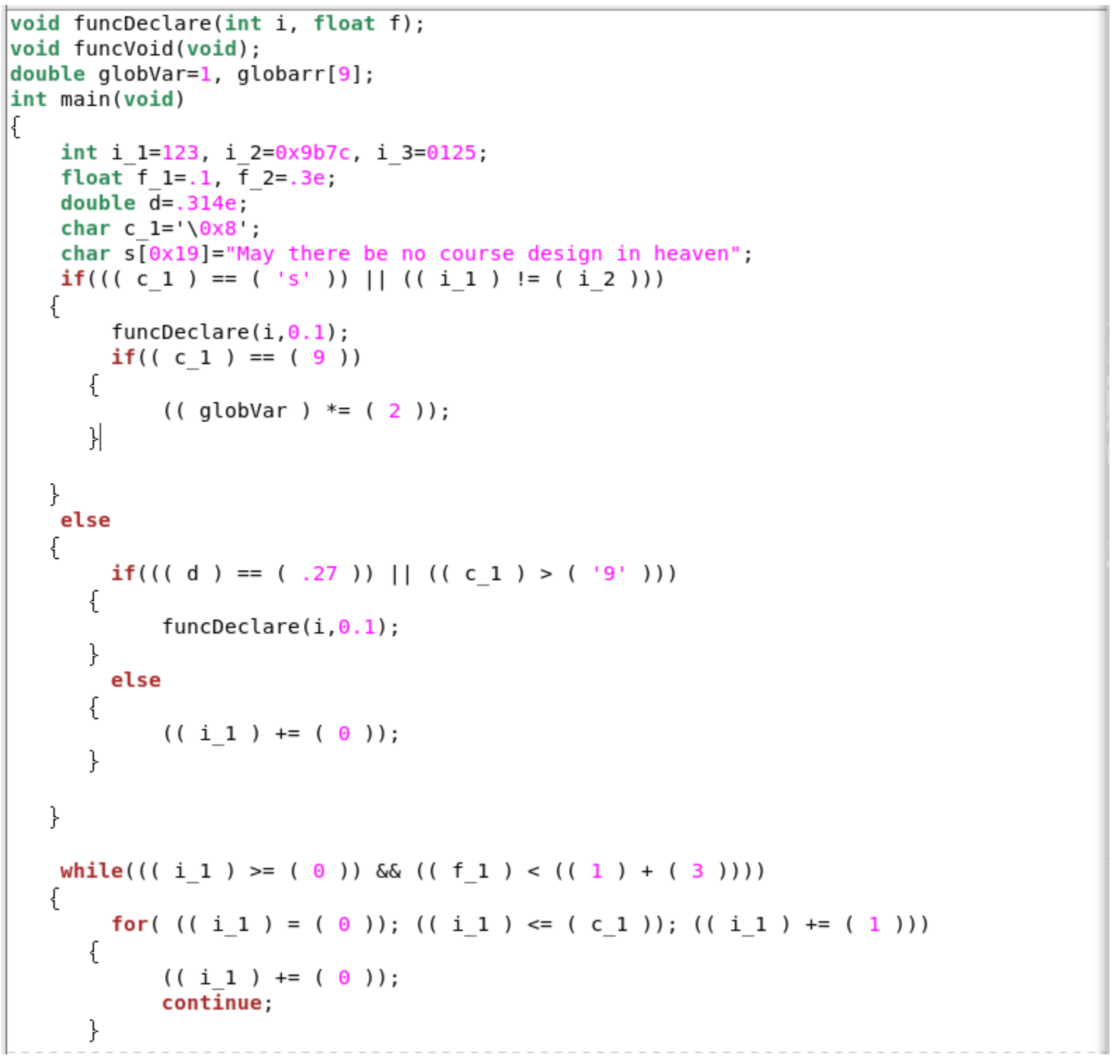
```
      condition expression:
      expression:
        ((( c_1 ) ==( 's' )) || (( i_1 ) !=( i_2 )))
      if statement:
        expression:
          funcDeclare(i,0.1)
        selection statement:
          condition expression:
          expression:
            (( c_1 ) ==( 9 ))
          if statement:
            expression:
              (( globVar ) *=( 2 ))
      else statement:
        selection statement:
          condition expression:
          expression:
            ((( d ) ==( .27 )) || (( c_1 ) >( '9' )))
          if statement:
            expression:
              funcDeclare(i,0.1)
          else statement:
            expression:
              (( i_1 ) +=( 0 ))
    while statement:
      condition expression:
      condition expression:
      expression:
        ((( i_1 ) >=( 0 )) && (( f_1 ) <(( 1 ) +( 3 ))))
      iteration statement:
      for statement:
condition val initial:
        expression:
          (( i_1 ) =( 0 ))
      condition expression:
        expression:
          (( i_1 ) <=( c_1 ))
      condition iteration:
        expression:
          (( i_1 ) +=( 1 ))
      iteration statement:
        expression:
```

图 4.15 格式化处理 2-2



```
      condition iteration:
        expression:
          (( i_1 ) +=( 1 ))
      iteration statement:
        expression:
          (( i_1 ) +=( 0 ))
        continue statement
      break statement
    return statement
      return expression:
      expression:
        ( i_1 )

**********将代码格式化输出至文件mid_test2.c**********
```

图 4.16 格式化处理 2-3



```
void funcDeclare(int i, float f);
void funcVoid(void);
double globVar=1, globarr[9];
int main(void)
{
    int i_1=123, i_2=0x9b7c, i_3=0125;
    float f_1=.1, f_2=.3e;
    double d=.314e;
    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;
      }
```

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()

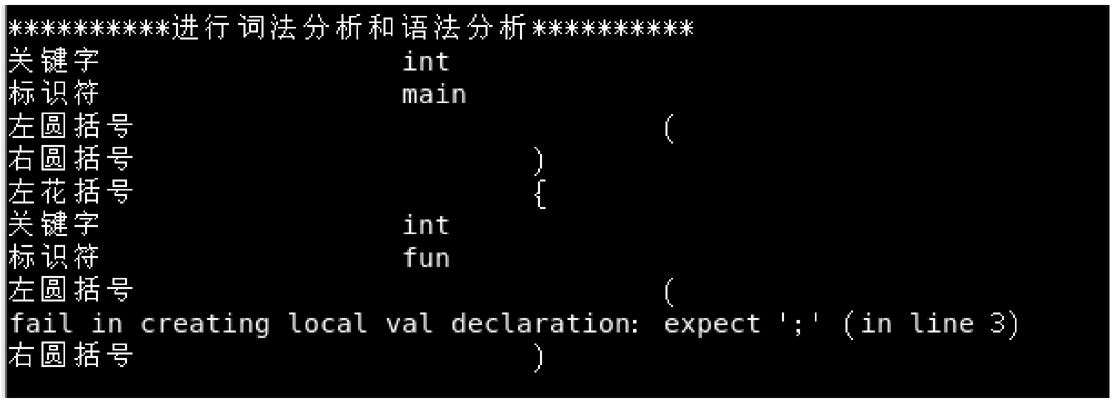
{

int fun()

int a = 1;

}

运行结果：



```
**********进行词法分析和语法分析**********
关键字                  int
标识符                  main
左圆括号                                (
右圆括号                        )
左花括号                        {
关键字                  int
标识符                  fun
左圆括号                                (
fail in creating local val declaration: expect ';' (in line 3)
右圆括号                        )
```

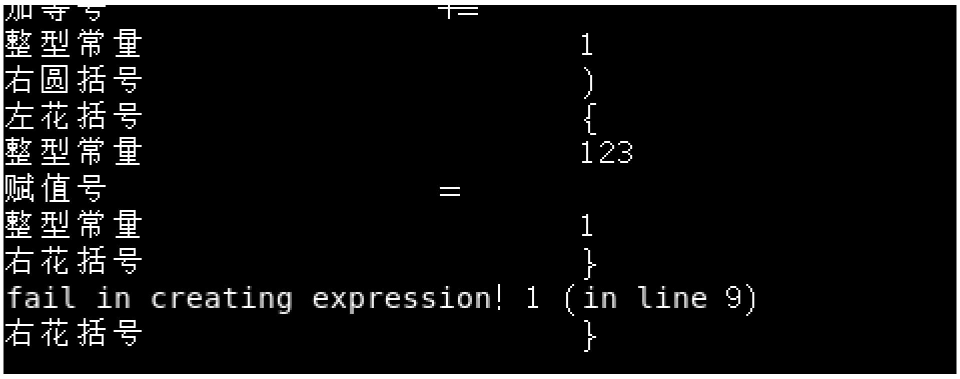
图 4. 18 异常处理 1

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：



```
加等号                  +=
整型常量                        1
右圆括号                        )
左花括号                        {
整型常量                        123
赋值号                  =
整型常量                        1
右花括号                        }
fail in creating expression! 1 (in line 9)
右花括号                        }
```

图 4. 19 异常处理 2

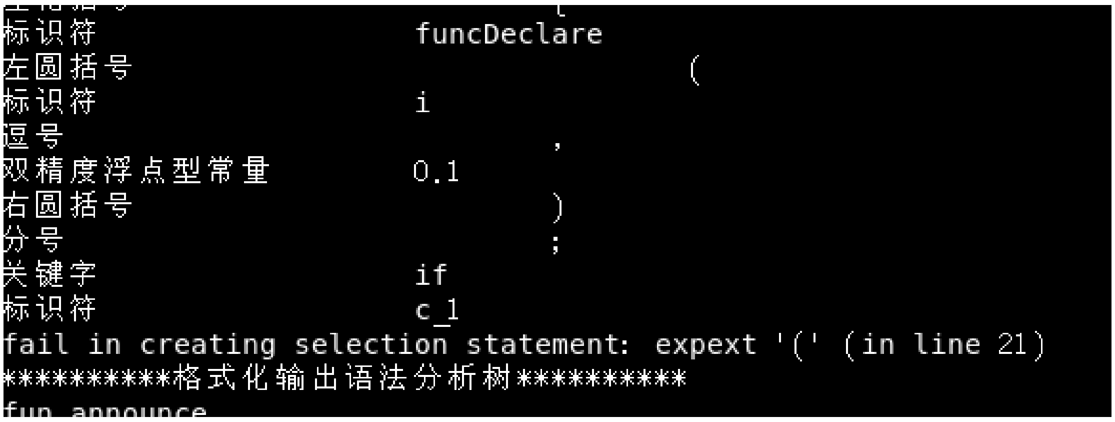
测试样例 3：

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：



```
标识符                 funcDeclare
左圆括号                                (
标识符                 i
逗号                           ,
双精度浮点型常量       0.1
右圆括号                       )
分号                           ;
关键字                 if
标识符                 c_1
fail in creating selection statement: expext '(' (in line 21)
**********格式化输出语法分析树**********
fun announce
```

图 4.19 异常处理 3

5.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
```