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：**CS2010**

学　　号：**U202015591**

姓　　名：郭子杰

指导教师：郑渤龙

报告日期：**2021.10.7**

计算机科学与技术学院

目录

0 任务书

0.1 设计内容

在计算机科学中，抽象语法树（abstract syntax tree 或者缩写为AST），是将源代码的语法结构的用树的形式表示，树上的每个结点都表示源程序代码中的一种语法成分。之所以说是“抽象”，是因为在抽象语法树中，忽略了源程序中语法成分的一些细节，突出了其主要语法特征。

在《高级语言源程序格式处理工具》这个题目中，首先需要采用形式化的方式，使用巴克斯（BNF）范式定义高级语言的词法规则（字符组成单词的规则）、语法规则（单词组成语句、程序等的规则）。再利用形式语言自动机的的原理，对源程序的文件进行词法分析，识别出所有单词；使用编译技术中的递归下降语法分析法，分析源程序的语法结构，并生成抽象语法树，最后可由抽象语法树生成格式化的源程序。

0.2 设计要求

要求具有如下功能：

1. **语言定义**：选定 C 语言的一个子集，要求包含：

   （1）基本数据类型的变量、常量，以及数组。不包含指针、结构，枚举等。

   （2）双目算术运算符（+-*/%），关系运算符、逻辑与（&&）、逻辑或（||）、赋值运算符。不包含逗号运算符、位运算符、各种单目运算符等等。

   （3）函数定义、声明与调用。

   （4）表达式语句、复合语句、if 语句的 2 种形式、while 语句、for 语句，return 语句、break 语句、continue 语句、外部变量说明语句、局部变量说明语句。

   （5）编译预处理（宏定义，文件包含）

（6）注释（块注释与行注释）

2. **单词识别：**设计 DFA 的状态转换图，实验时给出 DFA，并解释如何在状态迁移中完成单词识别（每个单词都有一个种类编号和单词的字符串这 2 个特征值），最终生成单词识别（词法分析）子程序。

   注：含后缀常量，以类型不同作为划分标准种类编码值，例如 123 类型为 int，123L 类型为 long，单词识别时，种类编码应该不同；但 0x123 和 123 类型都是 int，种类编码应该相同。

3. **语法结构分析：**

   （1）外部变量的声明；

   （2）函数声明与定义；

   （3）局部变量的声明；

   （4）语句及表达式；

   （5）生成（1)-(4)（包含编译预处理和注释）的抽象语法树并显示。

4. **按缩进编排生成源程序文件：**

   对测试用例生成的抽象语法树进行先根遍历，按缩进编排的方式写到.c 文件中，查看文件验证是否满足任务要求。

1 引言

1.1 课题背景与意义

1.1.1 编译原理

1.1.1.1 简介

编译原理即是对高级程序语言进行翻译的一门科学技术，我们都知道计算机程序由程序语言编写而成，在早期计算机程序语言发展较为缓慢，因为计算机存储的数据和执行的程序都是由 0、1 代码组合而成的，那么在早期程序员编写计算机程序时必须十分了解计算机的底层指令代码通过将这些微程序指令组合排列从而完成一个特定功能的程序，这就对程序员的要求非常高了。人们一直在研究如何如何高效的开发计算机程序，使编程的门槛降低。

编译就是将简单的类似自然语言的程序语言转换成计算机理解的机器语言使计算机高效地执行。从本质上来说编译就是一个算法问题，但由于从高级语言到低级语言的转换问题相当复杂，也就导致了编译的复杂性，在早期人们认为构造一个编译器是非常让人望而生畏的事。

1.1.1.2 编译技术的发展

在早期冯诺依曼计算机时期（20 世纪 40 年代）程序都是以机器 语言编写，机器语言就是实际存储的 01 代码，编写程序是十分枯燥 乏味的。后来汇编语言代替机器语言一符号形式该处操作指令和地 址编码。但汇编语言仍有许多缺点，阅读理解起来很难，而且必须依 赖于特定的机器，如果想使编写好的程序再另一台计算机上运行必 须重写。在 20 实际 50 年代 IBM 的 John Backus 带领一个研究小组 对 FORTRAN 高级语言及其编译器进行开发。编译程序的自动生成 工具初现端倪，现在很多自动生成工具已经广泛使用例如语法分析 工具 LEX，语言分析程序 YACC 等。在 20 世纪 60 年代人们不断的 用自编译技术构造编译程序，即用被编译的语言本身来实现该语言 的编译程序，但其基本原理和结构大体相同。经过不断发展现代编译 技术已经较为成熟，多种高级语言发展迅速都离不开编译技术的进步。

1.1.1.3 编译的基本流程

编译可以分为五个基本步骤：词法分析、语法分析、语义分析及 中间代码的生成、优化、目标代码的生成。这是每个编译器都必须的 基本步骤和流程，从源头输入高级语言源程序输出目标语言代码。

1 词法分析

词法分析器是通过词法分析程序对构成源程序的字符串从左到 右的扫描,逐个字符地读，识别出每个单词符号，识别出的符号一般 以二元式形式输出，即包含符号种类的编码和该符号的值。词法分析 器一般以函数的形式存在，供语法分析器调用。当然也可以一个独立 的词法分析器程序存在。完成词法分析任务的程序称为词法分析程 序或词法分析器或扫描器。

2 语法分析

语法分析是编译过程的第二个阶段。这阶段的任务是在词法分 析的基础上将识别出的单词符号序列组合成各类语法短语，如“语 句”，“表达式”等.语法分析程序的主要步骤是判断源程序语句是否 符合定义的语法规则，在语法结构上是否正确。而一个语法规则又称 为文法，乔姆斯基将文法根据施加不同的限制分为 0 型、1 型、2 型、3 型文法，0 型文法又称短语文法，1 型称为上下文有关文法，2 型称为 上下文无关文法，3 型文法称为正规文法，限制条件依次递增。

3 语义分析

词法分析注重的是每个单词是否合法，以及这个单词属于语言 中的哪些部分。语法分析的上下文无关文法注重的是输入语句是否 可以依据文法匹配产生式。那么，语义分析就是要了解各个语法单位 之间的关系是否合法。实际应用中就是对结构上正确的源程序进行 上下文有关性质的审查, 进行类型审查等。

4 中间代码生成与优化

在进行了语法分析和语义分析阶段的工作之后，有的编译程序 将源程序变成一种内部表示形式，这种内部表示形式叫做中间语言 或中间表示或中间代码。所谓“中间代码”是一种结构简单、含义明确 的记号系统，这种记号系统复杂性介于源程序语言和机器语言之间， 容易将它翻译成目标代码。另外，还可以在中间代码一级进行与机器 无关的优化。

5 目标代码的生成

根据优化后的中间代码，可生成有效的目标代码。而通常编译器 将其翻译为汇编代码，此时还需要将汇编代码经汇编器汇编为目标 机器的机器语言。

6 出错处理

编译的各个阶段都有可能发现源码中的错误，尤其是语法分析 阶段可能会发现大量的错误，因此编译器需要做出错处理，报告错误 类型及错误位置等信息。

1.1.1.4 编译用到的工具和方法

（1）有限自动机（FA）: 有限离散数字系统的抽象数学模型, 它由 一个有限的内部状态集和一组控制规则组成，这些规则是用来控制 在当前状态下读入输入符号后应转向什么状态。根据输入符号的不 同转向下一个状态是否唯一可以分为确定有限自动机（DFA）和不确 定有限自动机（NFA）。利用 DFA 可以初步构造实现词法分析器。

（2）自上而下分析方法：这时一个语法分析方法，给定文法 G 和 源程序串 r。从 G 的开始符号 S 出发，通过反复使用产生式对句型中 的非终结符进行替换(推导)，逐步推导出 r。分析的主旨是选择文法的 产生式的合适的侯选式进行推导，逐步使推导结果与 r 匹配。自上而 下方法面临两大问题：左递归和回溯，诸如 A- >Abc 产生式，因为分 析的处理过程实质上是树的深度遍历，左递归会使分析陷入死循环。 因为必须消除左递归才可以使用自上而下分析法来分析输入符号 串。回溯会造成虚假匹配，可能在某点虚假匹配后很长一段才出现匹 配失败这就需要匹配另一个候选式，必须逐层回溯到虚假匹配的节 点，这时很耗费时间和精力的一件事，那么如何从选取候选式阶段就 采用正确的候选式，一般采用求解非终结符的 FIRST 集合 FOLLOW 集来确定。

（3）自下而上分析方法：从给定的输入串 r 开始，不断寻找子串 与文法 G 中某个产生式 P 的候选式进行匹配，并用 P 的左部代替(归 约)之，逐步归约到开始符号 S。分析的主旨是寻找合适的子串与 P 的 侯选式进行匹配，直到归约到 G 的 S 为止。自下而上分析法中包含较 多分析方法常用的有算符优先分析法、规范规约法、LR 分析法等。

（4）语法制导翻译：是在解析输入的字符串时，在特定位置执行 指定的动作；换言之，根据语法把输入的字符串“翻译”为一串动作， 故名“语法制导翻译”。“特定位置”是通过把“指定动作”（称为语义动 作，semantic action）嵌入到语法规则中指定的。

1.1.1.5 高校教学

掌握编译器的原理对于计算机学习人员是最重要的基础知识。在高校教学中,编译原理主要是为了使学生了解高级语言源程序翻译成计算机能处理的目标

代码的整个过程。通过对编译原理的学习,学生可以系统掌握编译的基本原理和基本技术,对之前学习过的程序语言的设计与和实现有更好的理解，提高实践能力。

编译原理课程是计算机专业最难的课程之一，原因如下:(1)理论性很强，基本原理比较抽象，算法描述主要使用形式化语言，比如，有穷自动机、上下文无关文法等，与以往的学习有很大的差别,学生很难理解算法背后的精髓。(2)教学方式较为单一，因为理论知识过多,大部分学校为了保证教学进度,时常以灌输式为主，课堂上主要是编译系统原理为主，忽视了具体实例的讲解。(3)在实践方面,编译的算法较为复杂,对学生的编程能力以及逻辑思维能力都有很高的要求。

1.2 国内外研究现状

国外大学从 20 世纪 60 年代开始开设编译课程,较为经典的编译原理教程有 Aho 等编著的《Compilers: Principles,Techniques and Tools》(中文名:《编译:原理,技术与工具》)与 Appel 等 P 编著的《Modern Compiler ImplementationIn Java/C++/ML》(中文名:《现代编译原理——C 语言描述》)，这两本书在编译原理领域分别被称为“龙书”“虎书”。“龙书”出现于 1986 年,第 2 版删除了语法分析的算符优先分析法、语法翻译中递归计算方法等过时的技术,增加了面向对象的编译、类型检查等比较新新技术。“虎书”中包含 C 版本、Java 版本，在“龙书”知识点基础之上，增加了循环优化等内容。

近年来,国内学校在编译原理教学方面也积累了大量的经验，上海交通大学张冬茉等 3 提出了现阶段进行大型编译原理课程设计的教学模式和方法，使学生能够通过实现一个实用的编译系统,提高对编译原理的认识。合肥师范学院祖弦等°针对核心知识点,设计实验教学案例,探讨在实验案例驱动下的编译原理课程教学创新方案。

1.3 课程设计的主要研究工作

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。

词法分析的过程，就是在读取源程序的文本文件的过程中，识别出一个个的单词。在词法分析前，需要先给每一类单词定义一个类别码（用枚举常量形式），识别出一个单词后，即可得到该单词的类别码和单词自身值（对应的符

号串）。实现词法分析器的相关技术是采用有穷自动机的原理。

语法分析的过程，建议采用的实现方法是编译技术中的递归下降子程序法，递归下降子程序法是一种非常简洁的语法结构分析算法，基本上是每个语法成分对应一个子程序，每次根据识别出的前几个单词，明确对应的语法成分，调用相应子程序进行语法结构分析。例如在分析语句的语法结构时，当识别出单词 if 后，进行条件语句的处理，同时生成的子树根结点对应条件语句。处理时，首先调用表达式的子程序，得到表达式子树的根指针 T1；再递归调用语句处理部分，得到 if 子句的子树根指针 T2；再看随后的单词，如果不是 else，就表示是一个 if 语句，条件语句子树的根结点标记为“IF 语句”，有 2 棵子树，对应 T1 和 T2；如果随后的单词是 else，就再递归调用语句处理部分，得到 else 子句的子树根指针 T3；最后分析出的是一个 if-else 语句，条件语句子树的根结点标记为“IF_ELSE 语句”，有 3 棵子树，对应 T1、T2 和 T3。

2 系统需求分析与总体设计

2.1 系统需求分析

给定一个测试文件，要求：

（1）识别语言的全部单词

注意相同种类编码的多种形式，都应该包含在测试用例中，例如类型为 int 的常量，有三种形式 0123、123、0x123。

报错功能，指出不符合单词定义的符号位置。

（2）语法结构分析与生成抽象语法树

包含函数声明，定义、表达式（各种运算符均在某个表达式中出现）、所有的语句，以及 if 语句的嵌套，循环语句的嵌套。

常见的语句包括：1.表达式语句； 2.if 语句； 3.if else 语句； 4.while 语句; 5.for 语句； 6. return 语句； 7.break 语句； 8.continue 语句； 9.复合语句； 10.函数定义； 11.函数声明； 12.函数调用； 13. if 语句嵌套； 14.循环语句嵌套；15.外部变量说明语句; 16. 局部变量说明语句。

报错功能，指出不符合语法规则的错误位置。

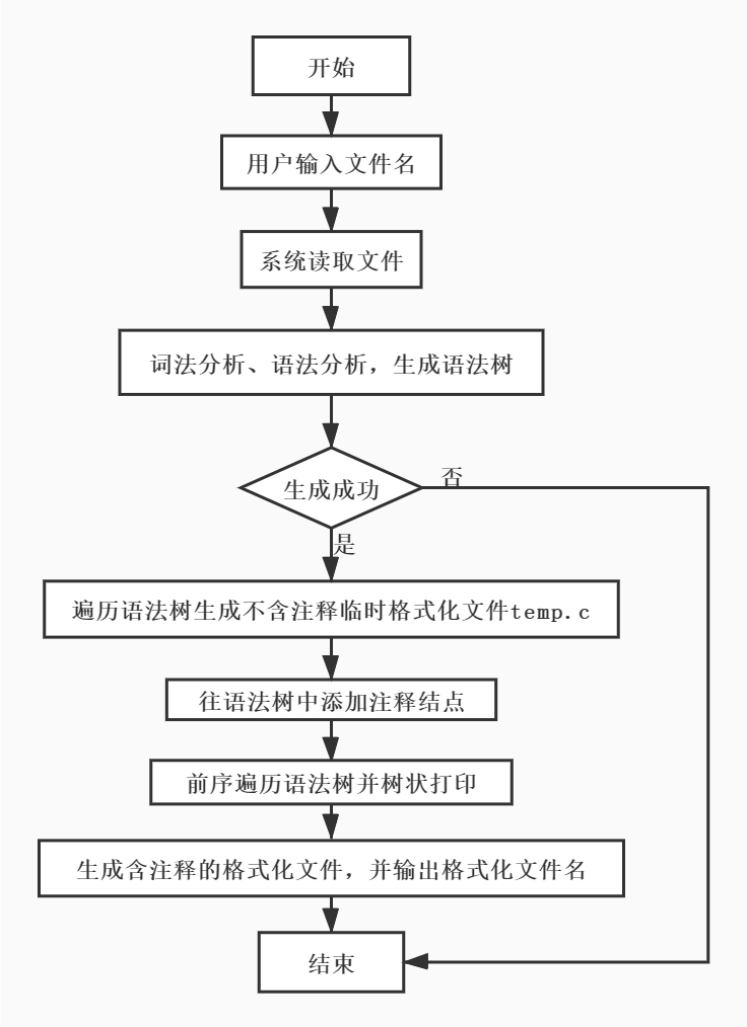
显示抽象语法树，能由抽象语法树说明源程序的语法结构

（3）缩进编排重新生成源程序文件

对生成的抽象语法树进行先根遍历，按缩进编排的方式写到.c 文件中。

2.2 系统总体设计

系统包括两个两个大模块，分别是根据源文件构建抽象语法树和生成格式化文件，其中第二个模块又包括预生成（不含注释）和生成（包含注释）。在程序开始时，由用户提供源文件路径，程序输出由词法分析得到的单词，遍历语法分析得到的抽象语法树并打印其树状结构，最后生成格式化文件，将生成文件的文件名输出。在语法分析过程中，如果发现语法错误，则报错并结束程序。


开始
用户输入文件名
系统读取文件
词法分析、语法分析，生成语法树
生成成功
否
是
遍历语法树生成不含注释临时格式化文件temp.c
往语法树中添加注释结点
前序遍历语法树并树状打印
生成含注释的格式化文件，并输出格式化文件名
结束

3 系统详细设计

3.1 有关数据结构的定义

3.1.1 数据类型

1. Bool

   枚举类型，成员包含 FALSE、TRUE。因为该系统使用纯 c 语言编写，所以自定义了一个 bool 类型，便于操作。

2. Token_kind

   枚举类型，成员包含 INT, FLOAT, DOUBLE, CHAR, LONG, VOID, IF, ELSE, WHILE, FOR, RETURN, BREAK, CONTINUE, DEFINE, INCLUDE, CONST, ARRAY, INT_ARRAY, FLOAT_ARRAY, STRING, STRING_CONST, ERROR_TOKEN, IDENT, INT_CONST, FLOAT_CONST, LONG_CONST, DOUBLE_CONST, EQ, UEQ, LH, RH, PLUS, SUB, MUL, DIV, ASSIGN, LP, RP, AND, OR, NOT, WELL, LSB, RSB, LCB, RCB, SEMI, COMMA, MOD, PLUSPLUS, SUBSUB, SINGLE_QUOTE, DOUBLE_QUOTE, LAB, RAB, NOTE1, NOTE2, NOTE3, NOTETEXT。

   用于定义单词种类，枚举成员按照类别分别对应于：

   关键字：int、float、double、char、long、void、if、else、while、for、return、break、continue、define、include、const。

   类型和标识符：数组、整形数组、浮点型数组、字符串、字符串常量、错误符号、标识符、整形常量、浮点型常量、长整形常量、双精度浮点常量。

   运算符：==、!=、>、<、+、-、*、/、=、(、)、&&、||、!、#、[、]、{、}、;、，、%、++、--、’、”、<，>。

   注释：//、/*、*/、注释文本

3. Node_type

   枚举类型，成员包含 _PROGRAMMAR_, _EXTDEFLIST_,_EXTVARDEF_, _EXTVARLIST_, _FUNCDEF_, _FORMALPARLIST_,_COMPOUNDSTATEMENT_,_LOCVARDEFLIST_, _LOCVARDEF_, _LOCVARLIST_, _ACTUALPARLIST_,

STATEMENTLIST, IFCLAUSE, ELSECLAUSE, LOOPBODY, _FOR_ONE_, _FOR_TWO_, _RETURNVAR_, _EXPRESSION_, _FUNCCALL_, FUNC, FORPARA, EXTVAR, LOCVAR, ACTPARA, TYPE, VAR,OPERATOR, IFELSE, _ASSIGN_, _WHILE_, _FOR_, _IF_, CONDITION, _RETURN_, _BREAK_, _CONTINUE_, _INCLUDE_, _DEFINE_, _FILENAME_, _DEFINE_ONE_, _DEFINE_TWO_, NOTE, _NOTE_。

用于标识语法树结点的类型，枚举类型分别对应于：程序、外部定义序列、外部变量定义、外部变量序列、函数定义、形参序列、复合语句、局部变量定义序列、局部变量定义、局部变量序列、实参序列、语句序列、if 子句、else 子句、循环体、for 初始语句、for 更新语句、返回值、表达式、函数调用、函数、形参、外部变量、局部变量、实参、类型、变量、操作符、ifelse 语句、赋值语句、while 语句、for 语句、if 语句、条件、return 语句、break 语句、continue 语句、include、define、文件名、宏名、宏名替换词、注释、注释序列。

4. Func

结构体类型，定义源文件函数序列，数据项：

```
char** funclist;      //函数名列表
int num, size;        //数量和大小
```

5. Note

结构体类型，定义一条注释，数据项：

```
char* data;      //注释内容
int row, type;   //注释出现的行、注释类型
```

6. NoteList

结构体类型，定义注释序列，数据项：

```
struct Note* notelist;     //注释列表
int num, size;             //数量和大小
```

7. ASTNode

结构体类型，定义抽象语法树的结点，系统最关键的类型，其中用 tag 标记结点结点类型，联合体 data 保存多种类型数据，孩子兄弟表示法连接其他结点，数据项：

```
int tag;  //结点数据类型标记
union   //结点数据
```

```
{
    char ele_name[50];  //语法成分
    char var_name[50];  //变量名
    char func_name[50]; //函数名
    int op;             //运算符
    int var_type;       //变量类型
    struct Note note;   //注释
}data;
struct ASTNode* child, * sibling;    //孩子兄弟表示法
```

8. Stack

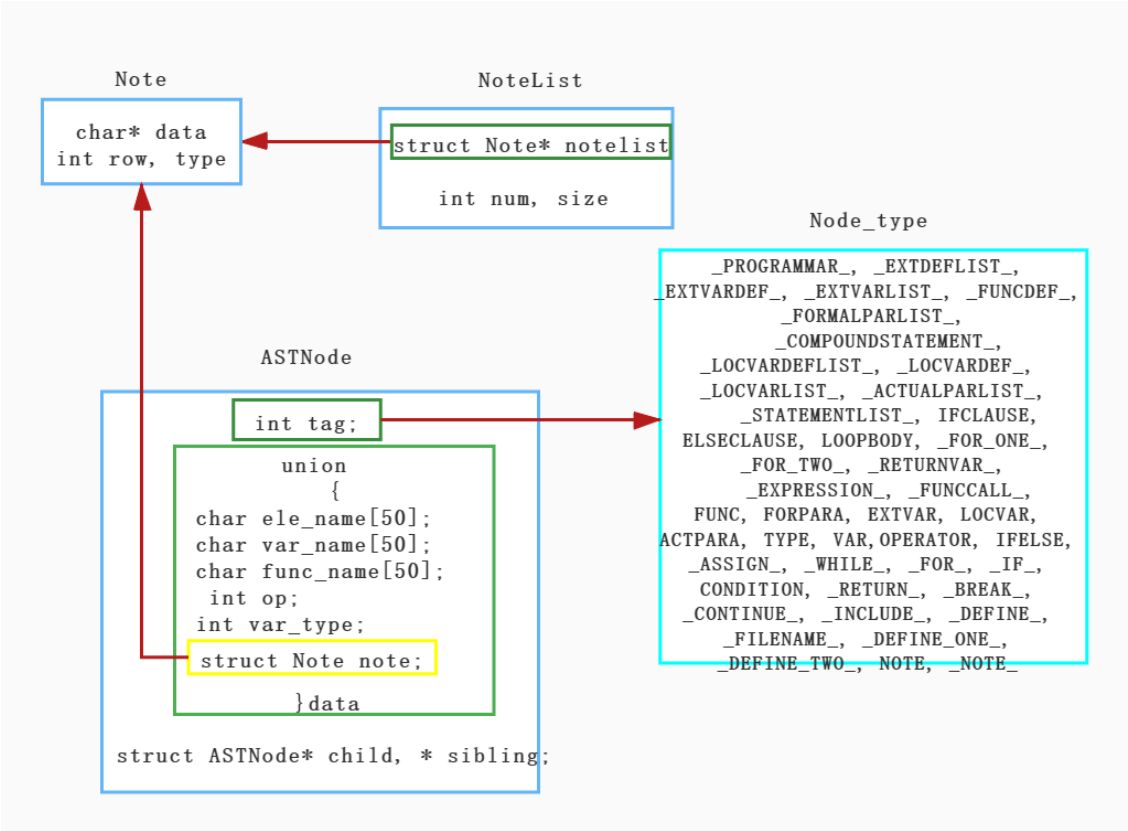
结构体类型，自定义栈类型，用于处理表达式，数据项：

```
ASTree* ptr;      //存储元素
int top, size;    //栈顶、栈容量
```

名称	类型	数据项或成员
Bool	枚举类型	FALSE、True
Token_kind	枚举类型	INT, FLOAT, DOUBLE, CHAR, LONG, VOID, IF, ELSE, WHILE, FOR, RETURN,BREAK, CONTINUE,DEFINE,INCLUDE,CONST, ARRAY, INT_ARRAY, FLOAT_ARRAY, STRING,STRING_CONST,ERROR_TOKEN,IDENT,INT_CONST, FLOAT_CONST, LONG_CONST,DOUBLE_CONST, EQ, UEQ, LH, RH, PLUS, SUB, MUL, DIV, ASSIGN, LP, RP, AND, OR, NOT, WELL, LSB, RSB, LCB, RCB, SEMI, COMMA, MOD,PLUSPLUS, SUBSUB, SINGLE_QUOTE, DOUBLE_QUOTE, LAB, RAB,NOTE1, NOTE2, NOTE3, NOTETEXT
Node_type	枚举类型	_PROGRAMMAR_, _EXTDEFLIST_, _EXTVARDEF_, _EXTVARLIST_, _FUNCDEF_, _FORMALPARLIST_, _COMPOUNDSTATEMENT_, _LOCVARDEFLIST_, _LOCVARDEF_, _LOCVARLIST_, _ACTUALPARLIST_, _STATEMENTLIST_, IFCLAUSE, ELSECLAUSE, LOOPBODY, _FOR_ONE_, _FOR_TWO_, _RETURNVAR_, _EXPRESSION_, _FUNCCALL_, FUNC, FORPARA, EXTVAR, LOCVAR, ACTPARA, TYPE, VAR,OPERATOR, IFELSE, _ASSIGN_, _WHILE_, _FOR_, _IF_, CONDITION, _RETURN_, _BREAK_, _CONTINUE_, _INCLUDE_, _DEFINE_, _FILENAME_, _DEFINE_ONE_, _DEFINE_TWO_, NOTE, _NOTE_
Func	结构体类型	char** funclist int num, size
Note	结构体类型	char* data int row, type

NoteList	结构体类型	struct Note* notelist int num, size
ASTNode	结构体类型	int tag union { char ele_name[50]; char var_name[50]; char func_name[50]; int op; int var_type; struct Note note; }data; struct ASTNode* child, * sibling
Stack	结构体类型	ASTree* ptr int top, size

3.1.2 关联



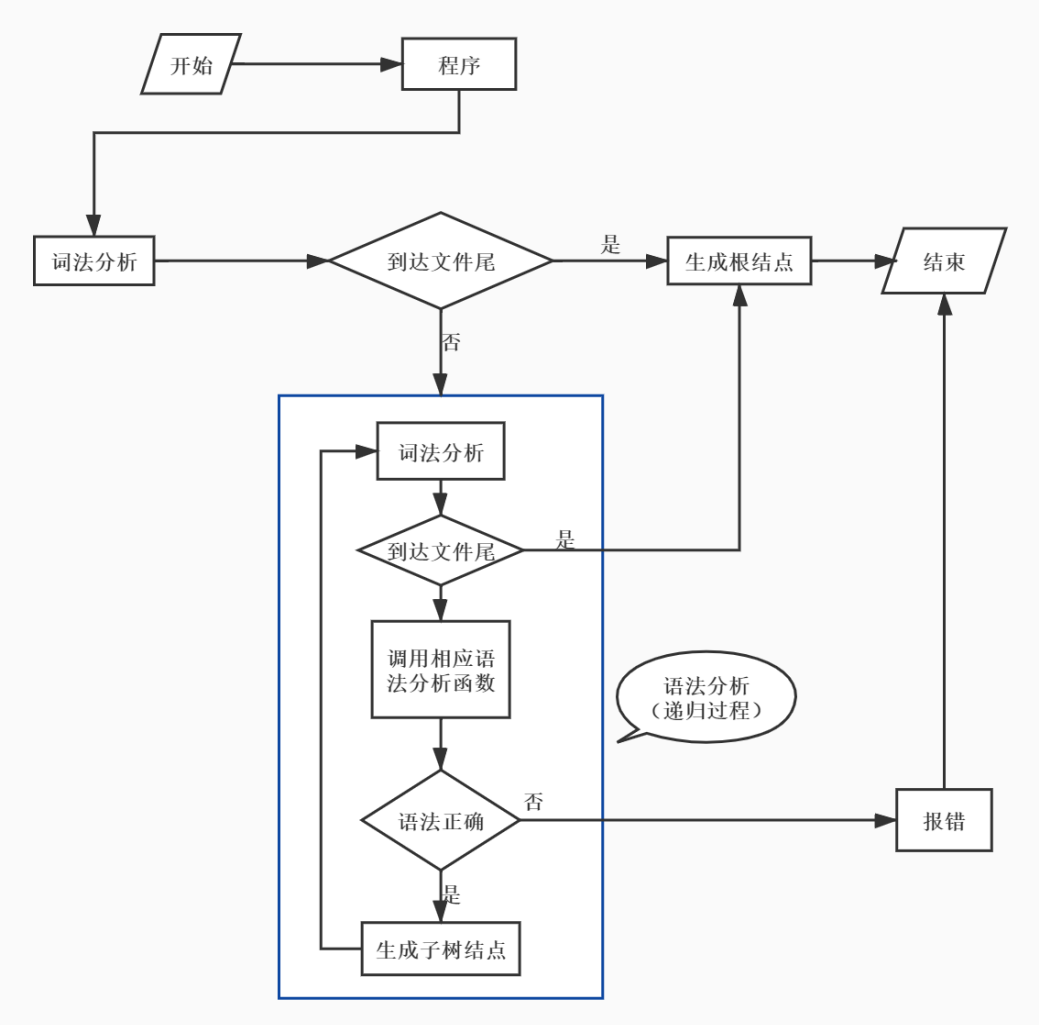
3.2 主要算法设计

3.2.1 构建抽象语法树

在词法分析上，运用有穷自动机原理，从源文件中逐个读取字符进行分析，达到终态后给出识别的单词，并打印。每个单词都对应了一个种类编码，由之前介绍的枚举类型 Token_kind 确定，具体流程将在 4.1.5 函数实现中给出。

在语法分析上，采用递归下降分析法，将各种语法分析的任务拆分为了多个函数，每个函数执行相应的语法分析任务，并生成抽象语法树的结点。在调用函数之前，会由前一个函数进行分析，递归调用其他语法分析函数。语法分析的过程中，需要单词时则调用一次词法分析程序，以识别语法，直至读取至源文件末。在语法分析过程中，若发现语法错误，则报错并结束分析。

流程图如下：



3.2.2 生成格式文件

该模块分为两阶段，预生成和生成。

在预生成阶段，前序递归遍历生成的抽象语法树，根据树结点的标记和数据将相应内容按照设定的格式写入临时文件 temp.c。该文件不含注释。

在生成阶段，以 temp.c 为读入文件，逐行读取，并根据语法分析过程得到的注释列表，将相应内容及注释写入最终的格式化文件。

具体实现见 4.1.5 函数实现。

4 系统实现与测试

4.1 系统实现

4.1.1 宏定义

```
#define INIT_SIZE 20                //初始化大小
#define INCREASEMENT 10             //每次增加的大小
```

4.1.2 自定义数据类型

解释详见 3.3.1 数据类型。

```
typedef enum Bool { FALSE, TRUE } Bool;     //自定义布尔类型

typedef enum Token_kind        //各类单词种类的枚举类型
{
    INT, FLOAT, DOUBLE, CHAR, LONG, VOID, IF, ELSE, WHILE, FOR,
    RETURN,
    BREAK, CONTINUE, DEFINE, INCLUDE, CONST,                    //部
    分关键字
    ARRAY, INT_ARRAY, FLOAT_ARRAY, STRING, STRING_CONST,
            //          数          组          类          型
        ERROR_TOKEN, IDENT, INT_CONST, FLOAT_CONST, LONG_C
    ONST, DOUBLE_CONST, //标识符和各类型常量
      EQ, UEQ, LH, RH, PLUS, SUB, MUL, DIV, ASSIGN, LP, RP,          //
    各种符号
    AND, OR, NOT, WELL, LSB, RSB, LCB, RCB, SEMI, COMMA, MOD,
```

```
    PLUSPLUS, SUBSUB, SINGLE_QUOTE, DOUBLE_QUOTE, LAB, RAB,
    NOTE1, NOTE2, NOTE3, NOTETEXT
}Token_kind;

typedef enum Node_type        //结点类型
{
        _PROGRAMMAR_, _EXTDEFLIST_, _EXTVARDEF_, _EXTVARLI
    ST_, _FUNCDEF_, _FORMALPARLIST_,
        _COMPOUNDSTATEMENT_, _LOCVARDEFLIST_, _LOCVARDF
    _, _LOCVARLIST_, _ACTUALPARLIST_,
    _STATEMENTLIST_, IFCLAUSE, ELSECLAUSE, LOOPBODY, _FOR_
    ONE_, _FOR_TWO_, _RETURNVAR_,
    _EXPRESSION_, _FUNCCALL_, FUNC, FORPARA, EXTVAR, LOCVA
    R, ACTPARA, TYPE, VAR,
    OPERATOR, IFELSE, _ASSIGN_, _WHILE_, _FOR_, _IF_, CONDITION,
     _RETURN_, _BREAK_,
    _CONTINUE_, _INCLUDE_, _DEFINE_, _FILENAME_, _DEFINE_ONE
    _, _DEFINE_TWO_, NOTE, _NOTE_
}Node_type;

struct Func
{
    char** funclist;       //函数名列表
    int num, size;         //数量和大小
} ;

struct Note
{
    char* data;       //注释内容
    int row, type;    //注释出现的行、注释类型
};    //注释结构体

struct NoteList
{
struct Note* notelist;      //注释列表
int num, size;              //数量和大小
};

typedef struct  ASTNode      //抽象语法树结点类型定义
```

```
{
   int tag;  //结点数据类型标记
   union    //结点数据
   {
      char ele_name[50];   //语法成分
      char var_name[50];   //变量名
      char func_name[50];  //函数名
      int op;              //运算符
      int var_type;        //变量类型
      struct Note note;    //注释
   }data;
   struct ASTNode* child, * sibling;    //孩子兄弟表示法
}ASTNode, * ASTree;

typedef struct Stack
{
   ASTree* ptr;       //存储元素
   int top, size;     //栈顶、栈容量
}Stack;               //自定义栈
```

4.1.3 全局变量

```
struct Func func = { NULL,0,0 };              //保存源文件出现的函数
struct NoteList notes = { NULL,0,0 };         //保存源文件出现的注释
FILE* fin, * fout;                            //指向读入文件和输出文件的指针
ASTree Root = NULL;                           //语法树的根结点
Bool error = FALSE;                           //错误标记
char sourcefile[100];                         //源文件路径
char* token_text = NULL, * token_text0 = NULL;  //当前和前一单词的字面值
int token_size = 0;                           //当前 token_text 的大小
int row = 1, col = 1;                         //记录当前读取行列号
int row0 = 1;                                 //格式化文件行数
int type, type0, type1 = -1;                  //暂时记录某一单词
const char* keyword[] =
{
"int","float","double","char","long","void","if","else",
```

```
"while", "for","return","break","continue","define",
"include","const",NULL
};                                  //根据枚举顺序初始化关键字查找表
```

4.1.4 函数声明

```
//辅助函数
Bool IsType(int t);                 //判断是否是类型关键字
Bool IsOperater(int t);             //判断是否是运算符
Bool IsFunc(char* s);               //判断是否是函数名
Bool Pop(Stack* p, ASTree* t);      //出栈
ASTree New();                       //创建新结点并初始化
ASTree Gettop(const Stack s);       //获取栈顶元素
int Find_keyword();                 //查找关键字
void InitStack(Stack* p);           //初始化栈
void Push(Stack* p, ASTree q);      //入栈
void Add_char(const char c);        //拼接字符
void Copy();                        //复制单词
void Info(int type);                //输出 token_text 信息
void Traverse(ASTree p, int dep);   //遍历语法树
char Precede(int c1, int c2);       //求运算符之间的优先级
void traverse(ASTree root);         //遍历表达式树
void ftraverse(ASTree root);        //遍历表达式树输入到文件中
void Addfunc(char* s);              //添加一个函数
void Addnote(char* s, int r, int t);//添加一个注释
void Warning(const char* s);        //报错
void End();                         //程序结束释放空间并关闭文件
void Destroy(ASTree root);          //销毁语法树

//词法分析
int Get_token();                    //词法分析

//语法分析
ASTree Program();                   //程序
ASTree FileInclude();               //文件包含
```

```
ASTree MacroDefine();                    //宏定义
ASTree ExtDefList();                     //外部定义序列
ASTree ExtVarDef();                      //外部变量定义
ASTree ExtVarList();                     //外部变量序列
ASTree ExtVar();                         //外部变量
ASTree ExtDef();                         //外部定义
ASTree LocVarDefList();                  //局部变量定义序列
ASTree LocVarDef();                      //局部变量定义
ASTree LocVarList();                     //局部变量序列
ASTree LocVar();                         //局部变量
ASTree FuncDef();                        //函数定义
ASTree FormalParList();                  //形参序列
ASTree FormalPara();                     //形参
ASTree ActualParList();                  //实参序列
ASTree CompoundStatement();              //复合语句
ASTree StatementList();                  //语句序列
ASTree Statement();                      //语句
ASTree Expression(Token_kind endsym);    //表达式

//格式化源文件
void Format(ASTree root, int dep);       //格式化源文件
void Writenote();                        //将注释写入格式化文件
```

4.1.5 函数功能及实现

(1) 辅助函数

1. void Add_char(const char c)　　//拼接字符

   当 token_text 为 NULL 时，初始化 token_text，当 token_text 的长度达到 token_size 时，重新为 token_text 分配空间。将字符 c 添加至 token_text 末尾。

2. void Addfunc(char* s)　　//添加一个函数

   当识别到源文件中出现函数时，将函数名用字符串 s 添加进入 func.funclist。当 fun.funclist 为空时初始化，当 func.num==func.size 时，重新分配空间并更新 func.size。每次添加，更新 func.num。

3. void Addnote(char* s, int r, int t)　　//添加一个注释

当识别到源文件中出现注释时，将注释内容和注释出现的行号即注释的类型添加进入 notes.notelist，当 notes.notelist 为空时初始化，当 notes.size==notes.num 时，重新分配空间并更新 notes.size。每次添加，更新 notes.num。

4. void Copy()　　//复制单词,将 token_text 的内容复制到 token_text0

当 token_text0 不为空时，释放 token_text0。为 token_text0 分配与 token_text 相同大小的空间，将 token_text 内容复制到 token_text0。

5. int Find_keyword()　　//查找关键字

根据关键字查找表，按顺序比对 token_text，比对成功则返回当先查找表的序号，否则返回-1。

6. Bool IsType(int t)　　//判断是否是类型关键字

根据枚举类型 Token_kind 中成员的值，判断 t 是否是类型关键字的编码。是，则返回 TRUE，否则返回 FALSE。

7. Bool IsOperater(int t)　　//判断是否是运算符

据枚举类型 Token_kind 中成员的值，判断 t 是否是运算符的编码。是，则返回 TRUE，否则返回 FALSE。

8. Bool IsFunc(char* s)　　//判断是否是函数名

在 func.funclist 中查找函数名字符串 s，查找成功返回 TRUE，否则，返回 FALSE。

9. void InitStack(Stack* p)　　//初始化栈

为栈分配存储空间，并置栈顶为 0，栈容量为 INIT_SIZE。

10. ASTree Gettop(const Stack s)　　//获取栈顶元素

若 s.top 为 0，即栈为空，则返回 NULL，否则返回栈顶元素，即 s.ptr[s.top - 1]。

11. Bool Pop(Stack* p, ASTree* t)　　//出栈

若 s.top 为 0，即栈为空，则返回 FALSE，否则，用 t 记录栈顶元素，并使 p->top 自减，返回 TRUE。

12. void Push(Stack* p, ASTree q)　//入栈

如果栈已满，即 p->top==p->size，则为栈重新分配空间，并更新栈容量。将元素 q 入栈，并更新 p->top。

13. char Precede(int t1, int t2)　//求运算符之间的优先级

根据下表，比较 t1、t2 对应的运算符，返回'>'、'<'、或'='，如不可比较则返回' '。

	+	-	*	/	(	)	=赋值	大小于	==和!=	#	&&	||
+	>	>	<	<	<	>		>	>	>	>	>
-	>	>	<	<	<	>		>	>	>	>	>
*	>	>	>	>	<	>		>	>	>	>	>
/	>	>	>	>	<	>		>	>	>	>	>
(	<	<	<	<	<	=		>	>	>	<	<
)	>	>	>	>	>			>	>	>	>	>
=赋值	<	<	<	<	<		<	<	<	>	<	<
大小于	<	<	<	<	<	>		>	>	>	>	>
==和!=	<	<	<	<	<	>		<	>	>	>	>
#	<	<	<	<	<		<	<	<	=	<	<
&&	<	<	<	<	<	>	<	<	<	>	>	>
||	<	<	<	<	<	>	<	<	<	>	<	>

14. void Info(int t)　//输出 token_text 信息

根据 t 对应的编码，打印 token_text 的信息。

15. ASTree New()　//创建新结点并初始化

创建新结点，并令其孩子和兄弟为 NULL。

16. void Traverse(ASTree p, int dep)　//遍历语法树

前序递归遍历语法树，根据结点的标记，按其树状结构以一定格式打印相应内容。

17. void traverse(ASTree root)　//先序遍历表达式树

中序递归遍历表达式树并输出，并适当添加括号保持运算优先级。

18. void ftraverse(ASTree root)　//遍历表达式树输入到文件中

中序递归遍历表达式树并输出到文件中，并适当添加括号保持运算

优先级。

19. void Warning(const char* s)　//报错，根据形参内容报出相应错误

令全局变量 error=TRUE，输出错误的行列号和错误类型。错误类型由字符串 s 给出。

20. void End()　//程序结束释放空间

释放系统分配的所有空间，销毁语法树。

21. void Destroy(ASTree root)　　//销毁语法树

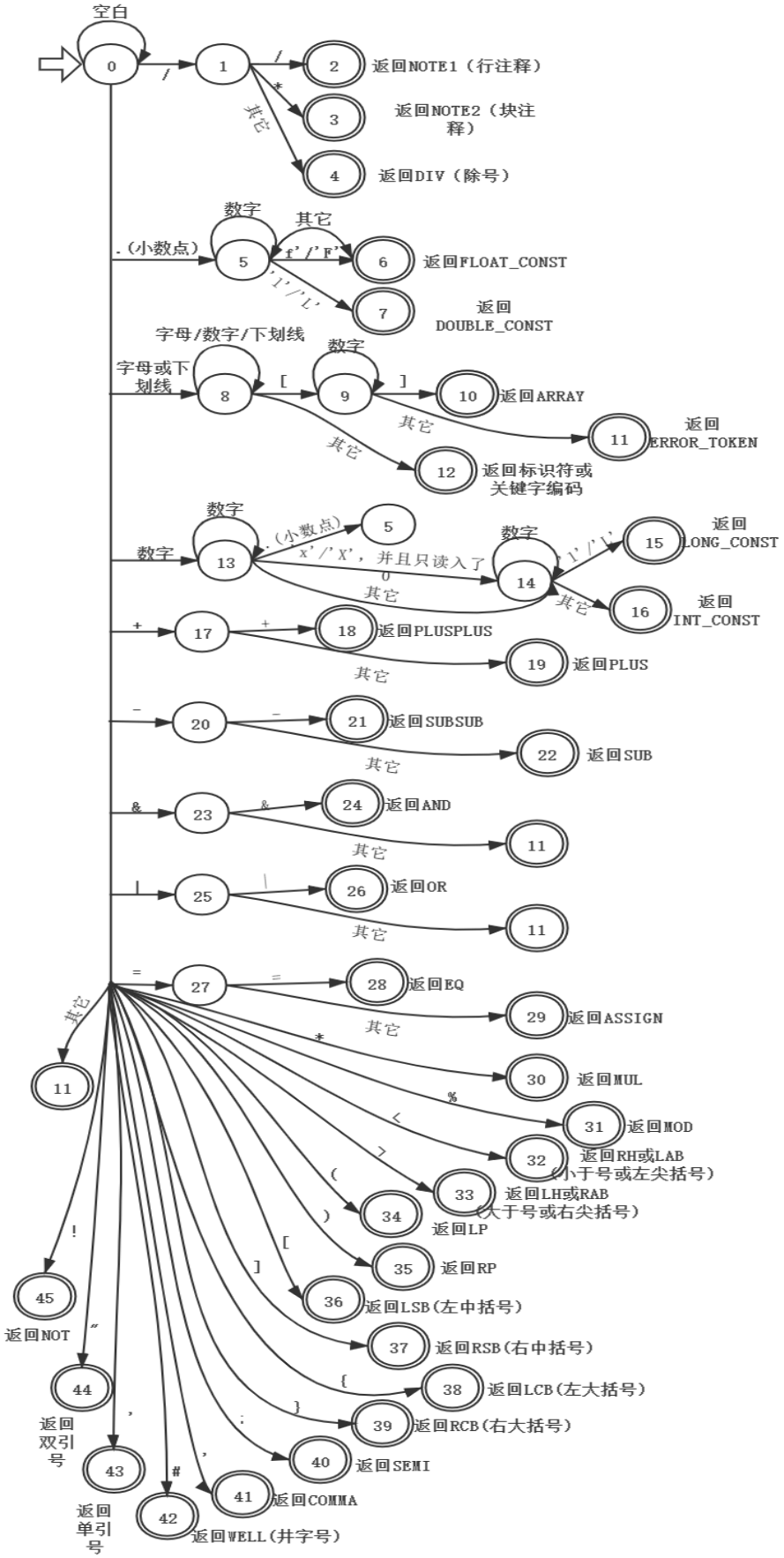
中序递归遍历语法树，释放结点空间。

(2) 主要函数

1. int Get_token()　//词法分析

运用有穷自动机的原理，对源文件进行逐字符读取，并进行字符拼接，根据已读取的字符进行状态转换和终态判断，到达终态后给出识别出的单，返回对应编码并打印，单词编码有自定义枚举类型 Token_kind 唯一确定。

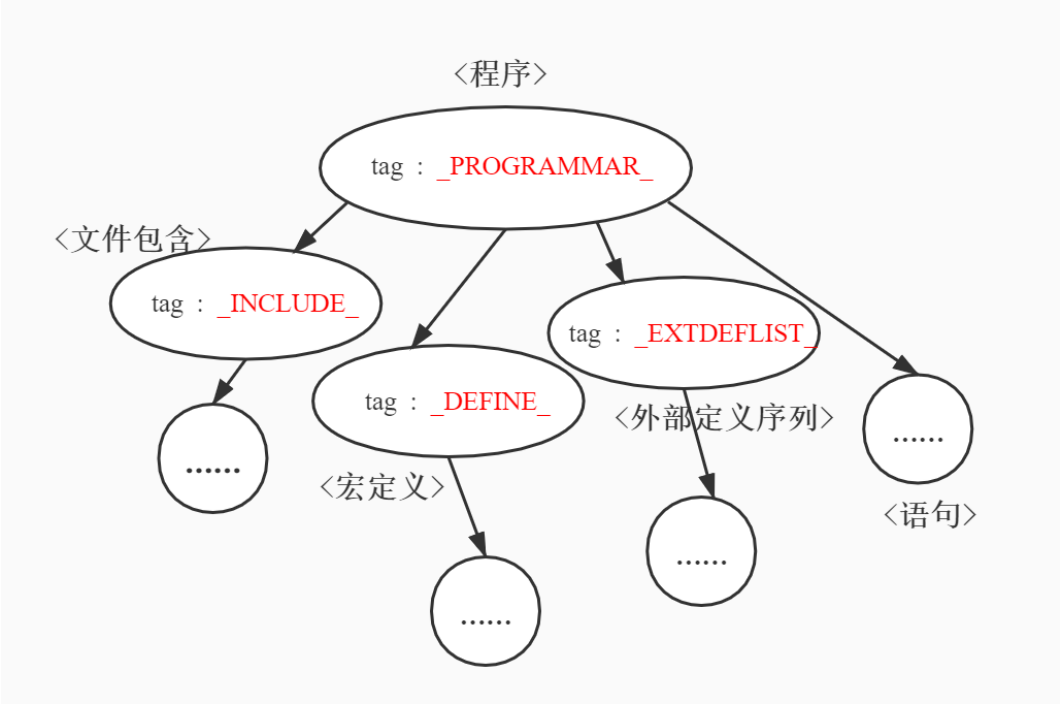
DFA 的状态转换图如下


空白
0
1
/
2
返回NOTE1（行注释）
*
3
返回NOTE2（块注释）
其它
4
返回DIV（除号）
数字
其它
.（小数点）
5
f’/’F’
6
返回FLOAT_CONST
’l’/’L’
7
返回DOUBLE_CONST
字母/数字/下划线
字母或下划线
8
[
9
数字
]
10
返回ARRAY
其它
11
返回ERROR_TOKEN
其它
12
返回标识符或关键字编码
数字
数字
13
.（小数点）
5
x’/’X’，并且只读入了0
14
数字
’l’/’L’
15
返回LONG_CONST
其它
16
返回INT_CONST
+
17
+
18
返回PLUSPLUS
其它
19
返回PLUS
-
20
-
21
返回SUBSUB
其它
22
返回SUB
&
23
&
24
返回AND
其它
11
|
25
|
26
返回OR
其它
11
=
27
=
28
返回EQ
其它
29
返回ASSIGN
其它
11
*
30
返回MUL
%
31
返回MOD
<
32
返回RH或LAB（小于号或左尖括号）
>
33
返回LH或RAB（大于号或右尖括号）
(
34
返回LP
)
35
返回RP
[
36
返回LSB（左中括号）
]
37
返回RSB（右中括号）
{
38
返回LCB（左大括号）
}
39
返回RCB（右大括号）
:
40
返回SEMI
,
41
返回COMMA
#
42
返回WELL（井字号）
’
43
返回单引号
”
44
返回双引号
!
45
返回NOT

2. ASTree Program()

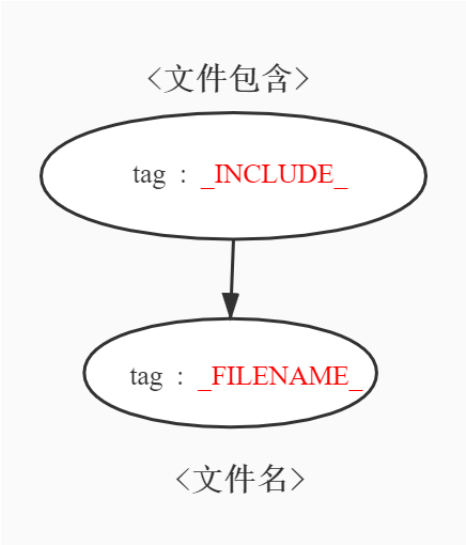
语法单位<程序> ,识别源程序文件中的文件包含、宏定义、外部定义序列、语句，调用相应的函数生成其子树。

生成整个源程序抽象语法树的根结点，整个程序执行过程只会执行一次。如果没分析到错误则返回根结点，否则，返回 NULL。



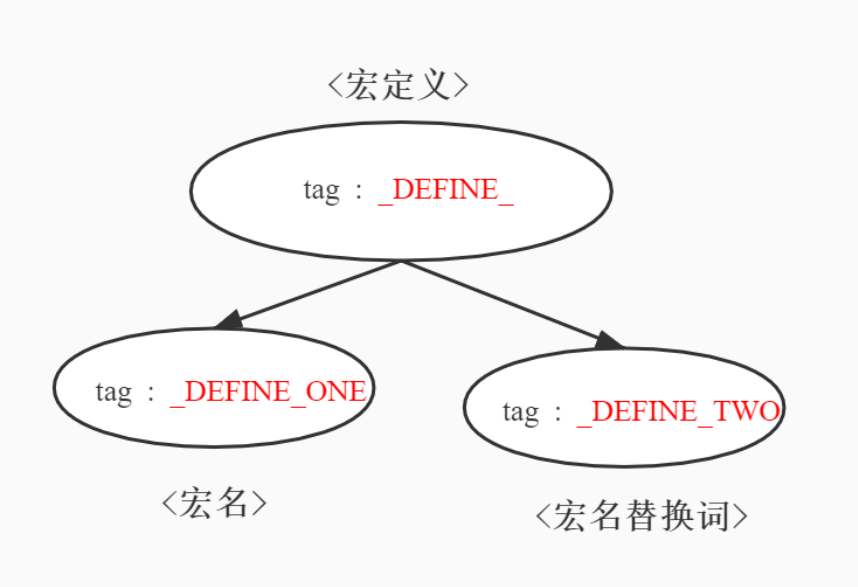
3. ASTree FileInclude()

处理源程序文件中的文件包含，运用自动机逐字符分析，处理空格、拼接文件名。子结点存储文件名。如果没分析到错误则返回根结点，否则，返回 NULL。



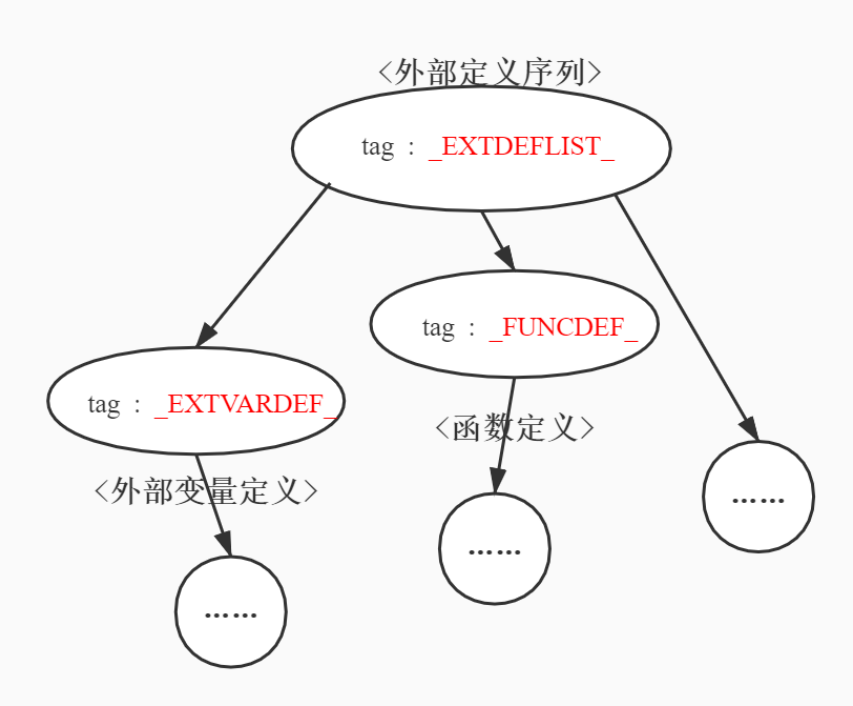
4. ASTree MacroDefine()

处理源程序文件中的宏定义，运用自动机逐字符分析，处理空格、拼接宏名、宏名替换词。生成的根结点有两个子结点，第一个子结点存宏名，第二个子结点存宏名替换词。如果没分析到错误则返回根结点，否则，返回 NULL。



5. ASTree ExtDefList()

处理源程序文件中的外部定义序列，其子孩子为一系列外部定义，包括外部变量定义和函数定义。如果没分析到错误则返回根结点，否则，返回 NULL。

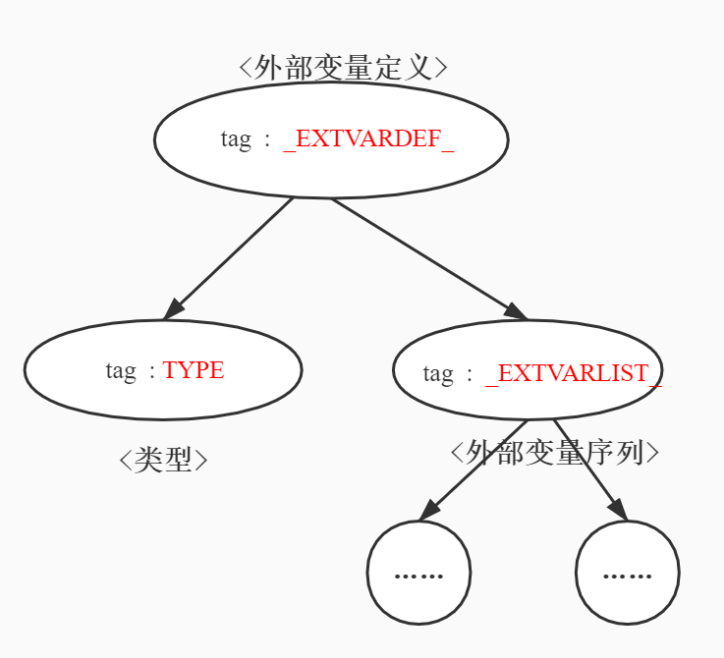


6. ASTree ExtDef()

处理源程序文件中的一个外部定义，由外部定义序列处理函数调用。该函数调用外部变量定义处理函数或函数定义处理函数，返回得到的根结点。

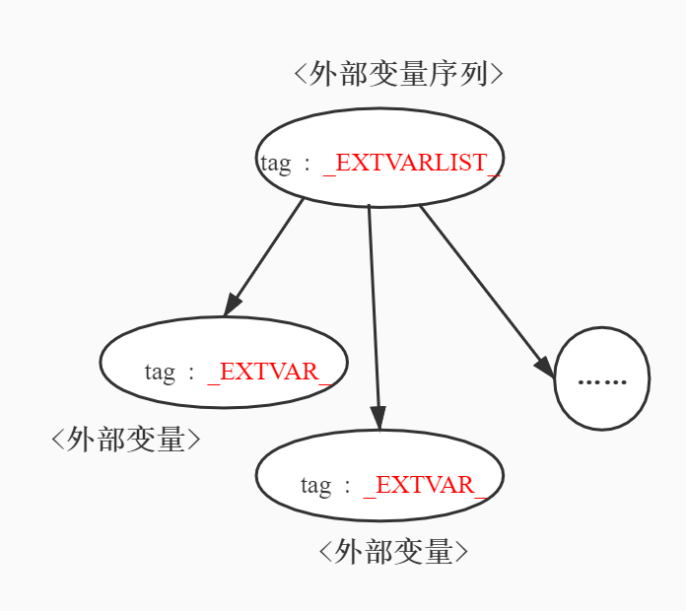
7. ASTree ExtVarDef()

处理源程序文件中的外部变量定义。生成的根结点有两个子结点。第一个子结点存变量的类型，第二的子结点为变量名序列。如果没分析到错误则返回根结点，否则，返回 NULL。

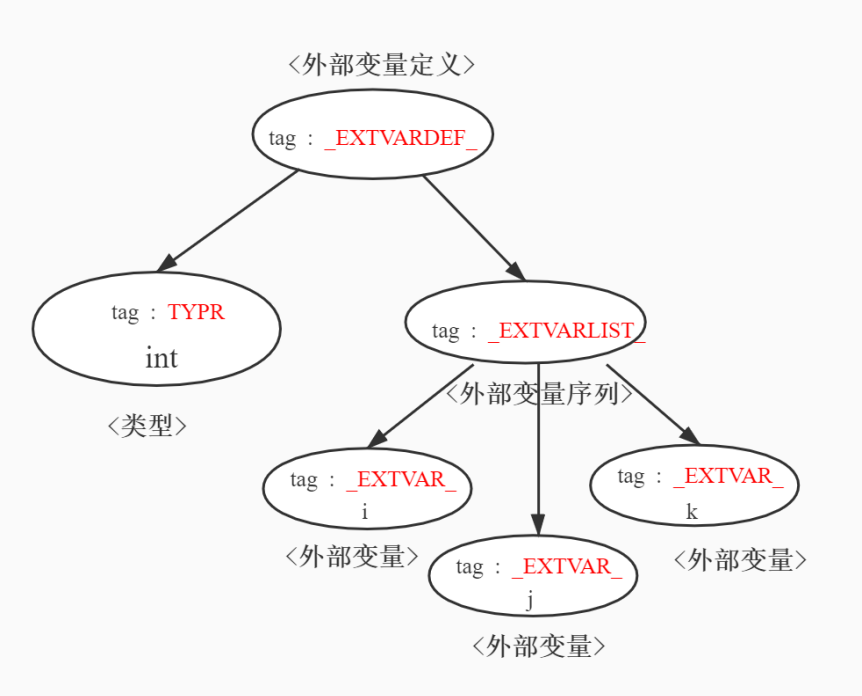


8. ASTree ExtVarList()

处理源程序文件中的外部变量名序列，子孩子个数为变量个数，一个子孩子存一个变量名。如果没分析到错误则返回根结点，否则，返回 NULL。



例如，对于外部变量定义 int i,j,k; 可以得到如下子树。



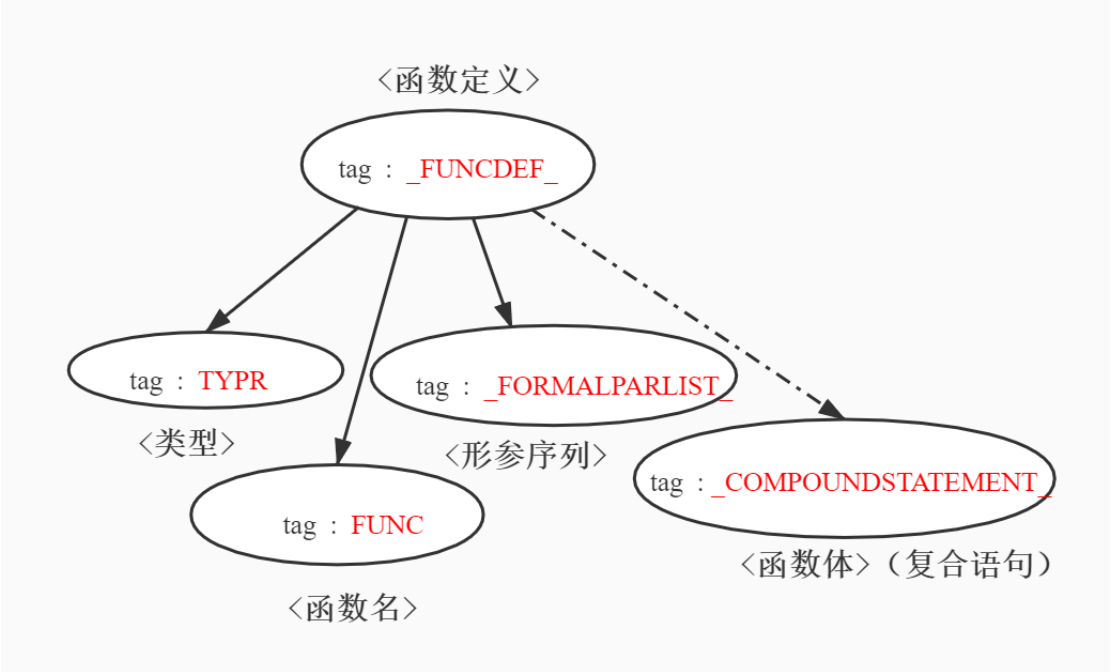
9. ASTree ExtVar()

处理一个外部变量，生成的结点存储变量名。返回该结点。

10. ASTree FuncDef()

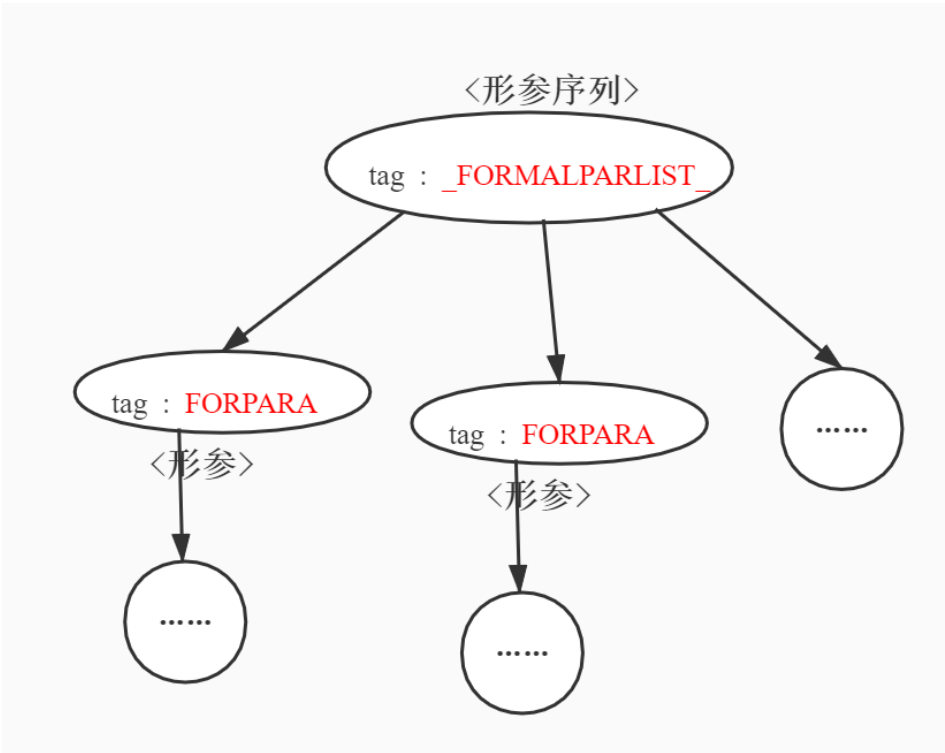
处理源程序文件中的函数定义。若为函数声明则有三个孩子结点，若为函数定义则有四个孩子结点。第一个子结点存函数返回类型，第二个子结点存函数名，第三个子结点存形参序列，若为函数定义则第四个

子结点存函数体。如果没分析到错误则返回根结点，否则，返回 NULL。



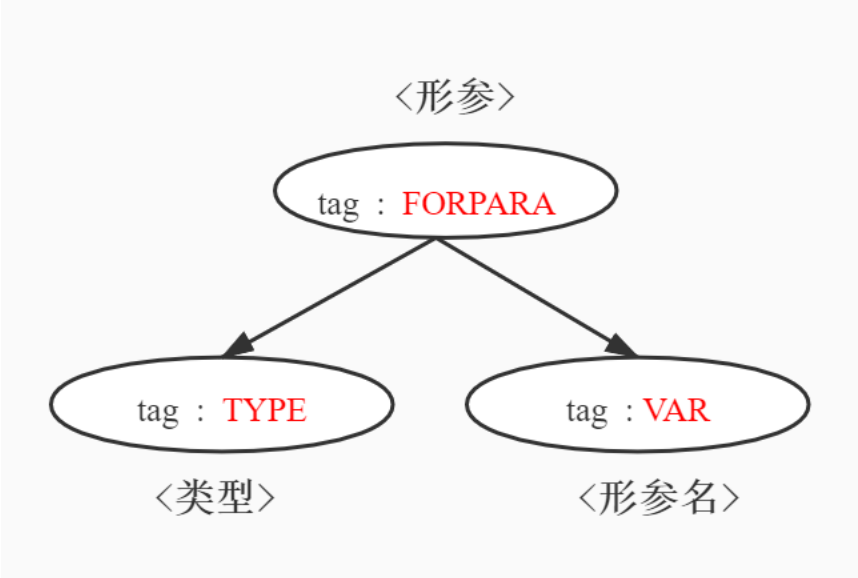
11. ASTree FormalParList()

处理源程序文件中函数声明或定义中的形参序列，孩子结点的个数为形参个数，每个孩子结点存一个形参，包括类型和形参名。如果没分析到错误则返回根结点，否则，返回 NULL。

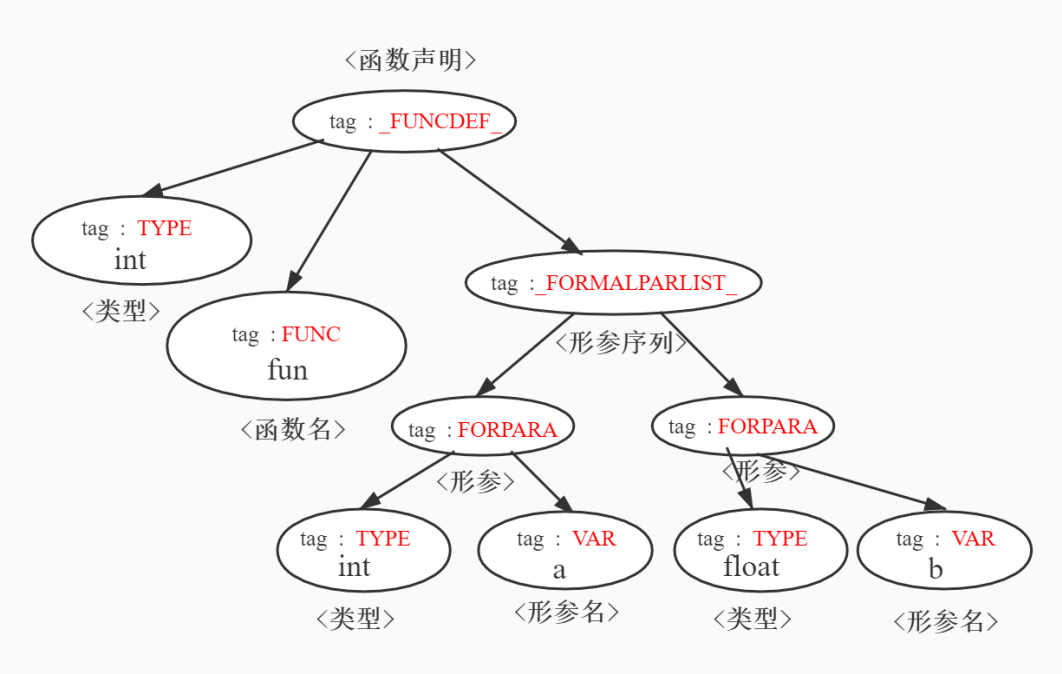


12. ASTree FormalPara()

处理源程序文件中函数声明或定义中的一个形参，该函数形参序列处理函数调用。生成的根结点有两个孩子结点，第一个孩子存形参类型，第二个孩子存形参名。如果没分析到错误则返回根结点，否则，返回NULL。

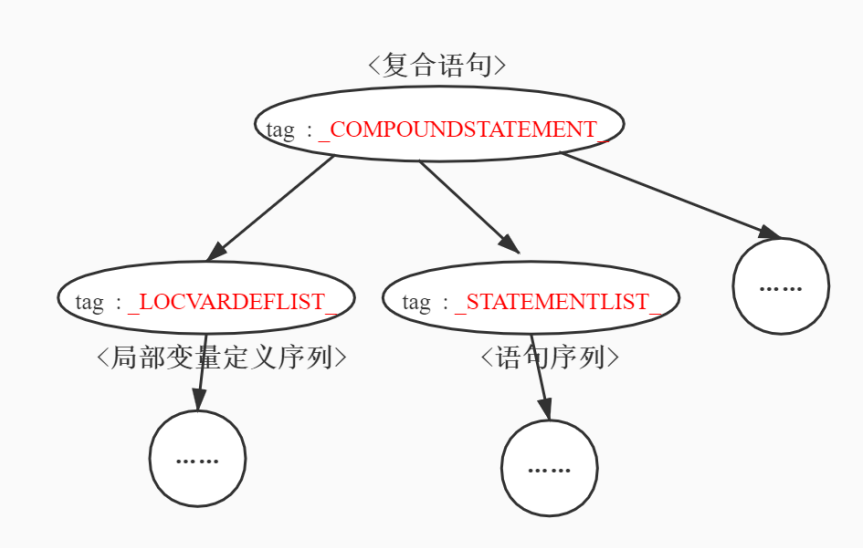


例如，对于函数声明　int fun(int a,float b); 可以得到如下子树。



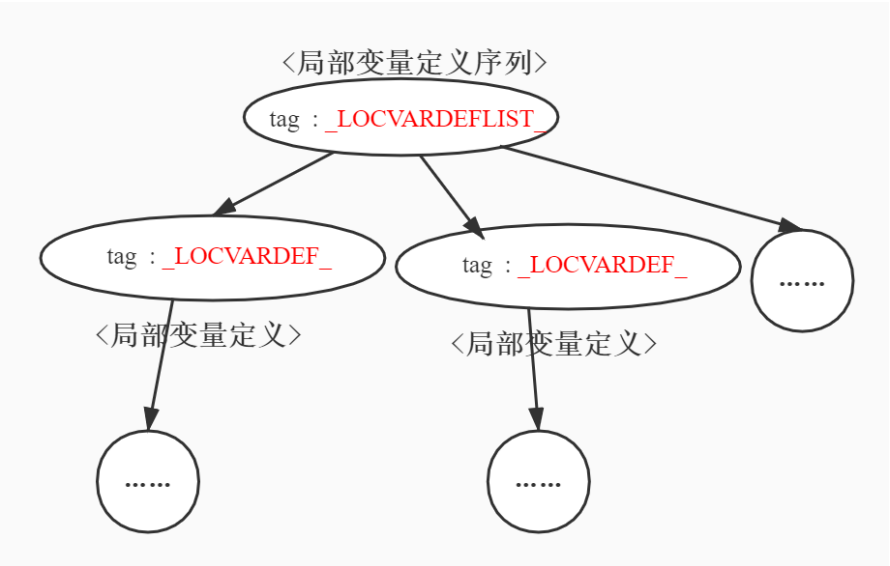
13. ASTree CompoundStatement()

处理源程序文件中的复合语句，也即一个代码块。孩子结点的个数为局部变量定义序列和语句序列的个数。如果没分析到错误则返回根结点，否则，返回 NULL。



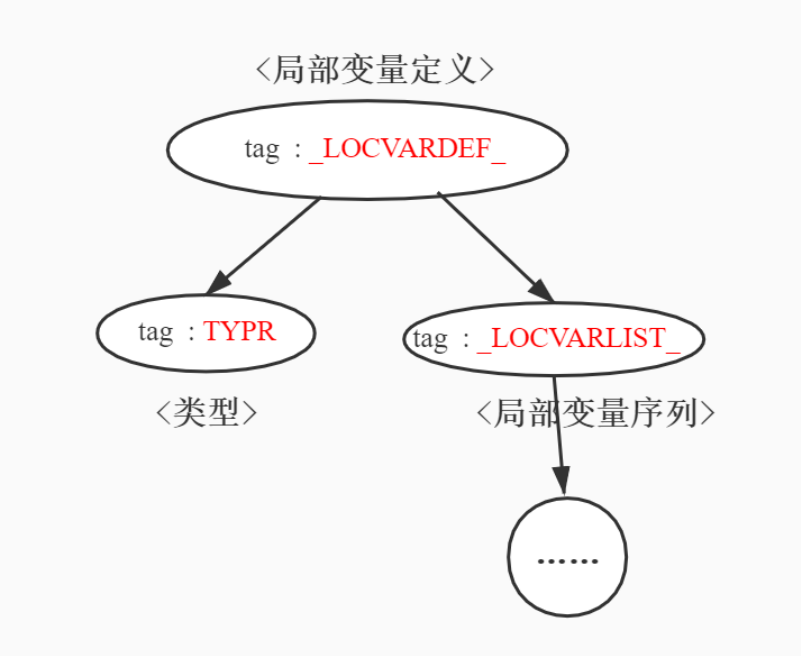
14. ASTree LocVarDefList()

处理源程序文件中的局部变量定义序列，孩子结点个数为局部变量定义的个数。每个孩子结点处理一个局部变量定义。如果没分析到错误则返回根结点，否则，返回 NULL。



15. ASTree LocVarDef()

处理源程序文件中的局部变量定义，类似于 ExtVarDef()函数，生成的根结点有两个孩子结点。第一个孩子结点存类型，第二个孩子结点出列变量序列。该函数由局部变量定义序列函数调用，如果没分析到错误则返回根结点，否则，返回 NULL。



16. ASTree LocVarList()

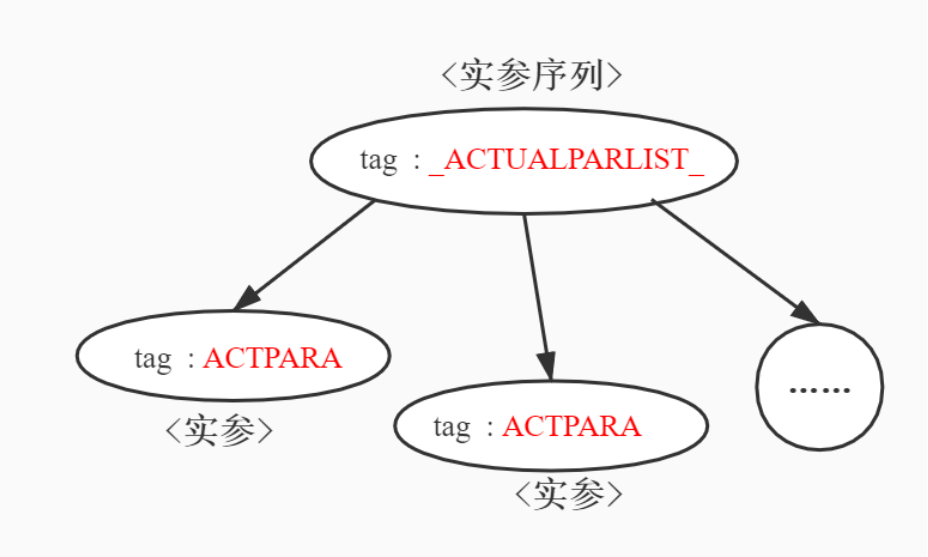
处理源程序文件中的局部变量序列，类似于 ExtVarList()函数。

17. ASTree LocVar()

处理源程序文件中的一个局部变量，类似于 ExtVar()函数。

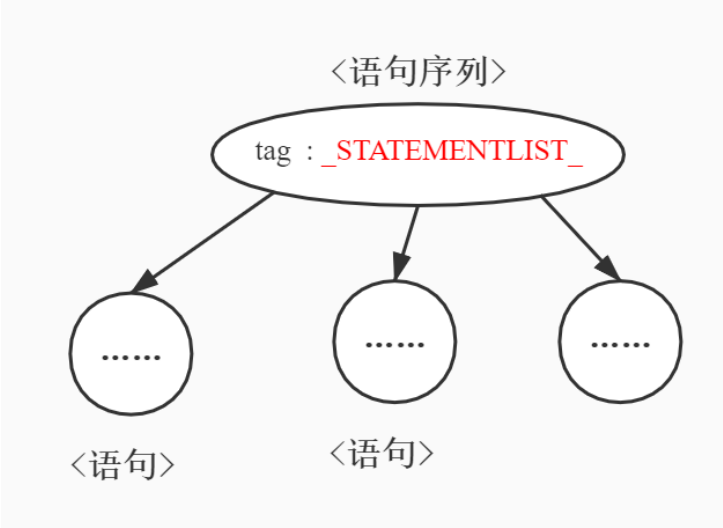
18. ASTree ActualParList()

处理源程序文件中的实参序列，孩子结点个数为实参个数，每个孩子结点存储一个实参。如果没分析到错误则返回根结点，否则，返回 NULL。



19. ASTree StatementList()

处理源程序文件中的语句序列，孩子结点个数为语句个数，每个孩子结点处理一个语句。如果没分析到错误则返回根结点，否则，返回NULL。

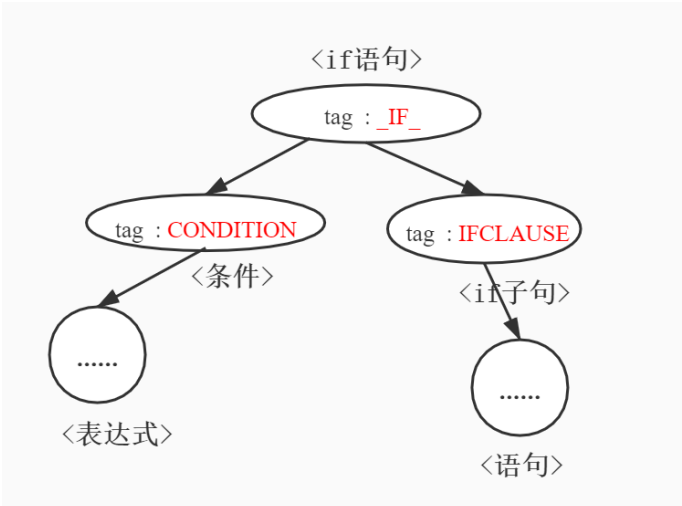


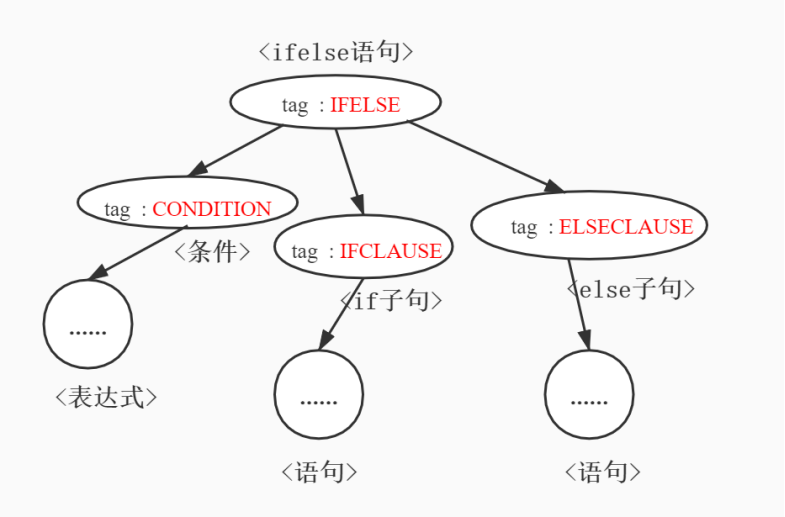
20. ASTree Statement()

处理源程序文件中的各种语句，包括 if 语句、ifelse 语句、while 语句、for 语句、break 语句、continue 语句、return 语句、表达式语句、函数调用语句。由 switch 语句识别应处理哪个语句，如果没分析到错误则返回对应根结点，否则，返回 NULL。

① if 语句 / ifelse 语句：

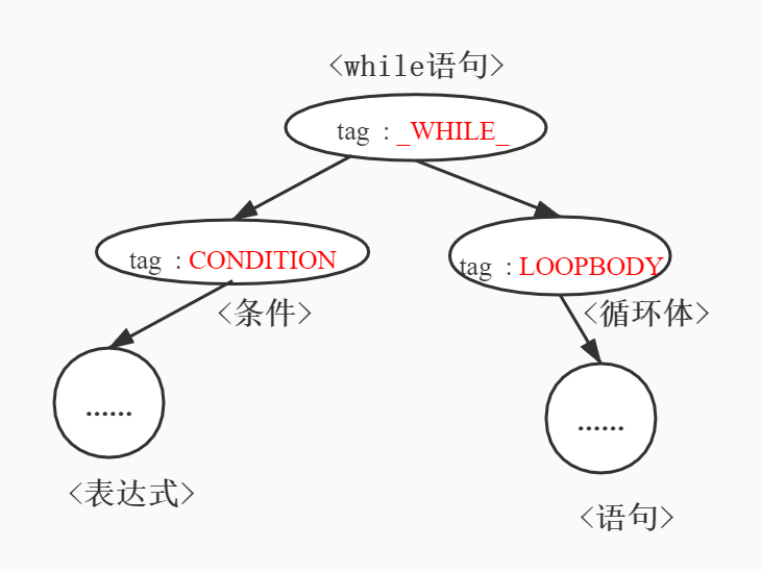
若为 ifelse 语句，则根结点有三个子孩子，否则有两个子孩子。第一个子结点存条件，第二个子结点存 if 子句，如果为 ifelse 语句，则第三个子结点存 else 子句





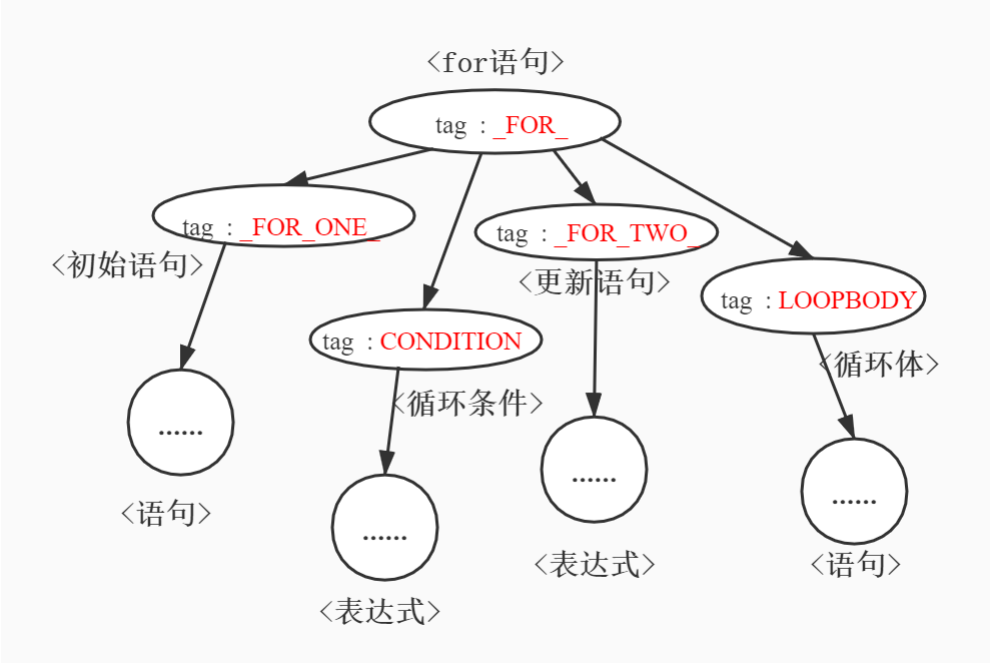
② while 语句：

有两个子孩子，第一个子结点存循环条件，第二个子结点存循环体。



③ for 语句：

有四个子结点。第一个子结点存初始语句，第二个子结点存循环条件，第三个子结点存更新语句，第四个子结点存循环体。

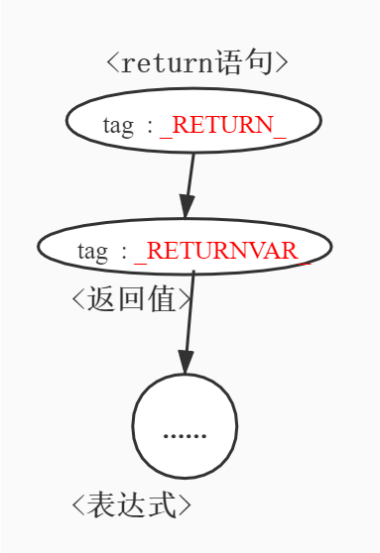


④ break / continue 语句：

无孩子结点，由根结点标记 break 或 continue。

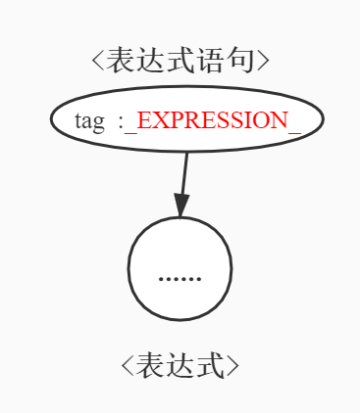
⑤ return 语句：

有一个子结点，用来存返回值。



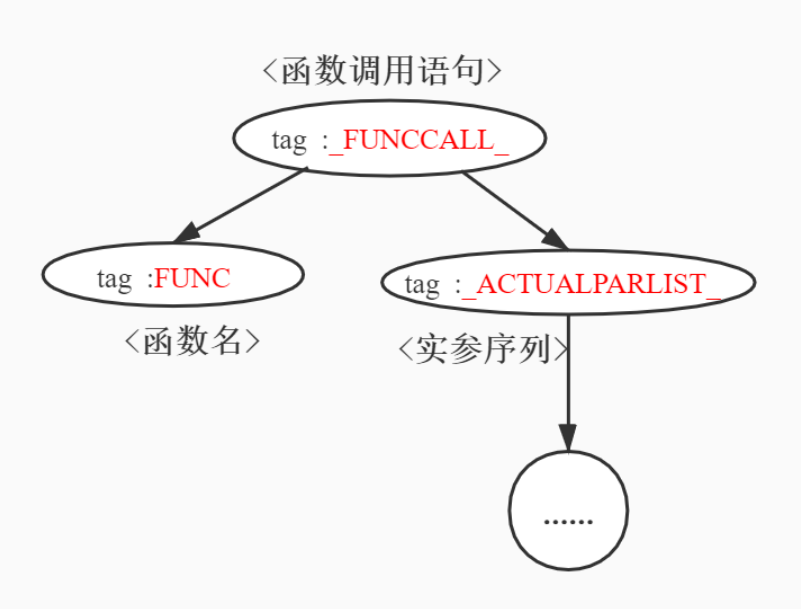
⑥ 表达式语句：

有一个子结点，为表达式树的根结点。



⑦ 函数调用语句：

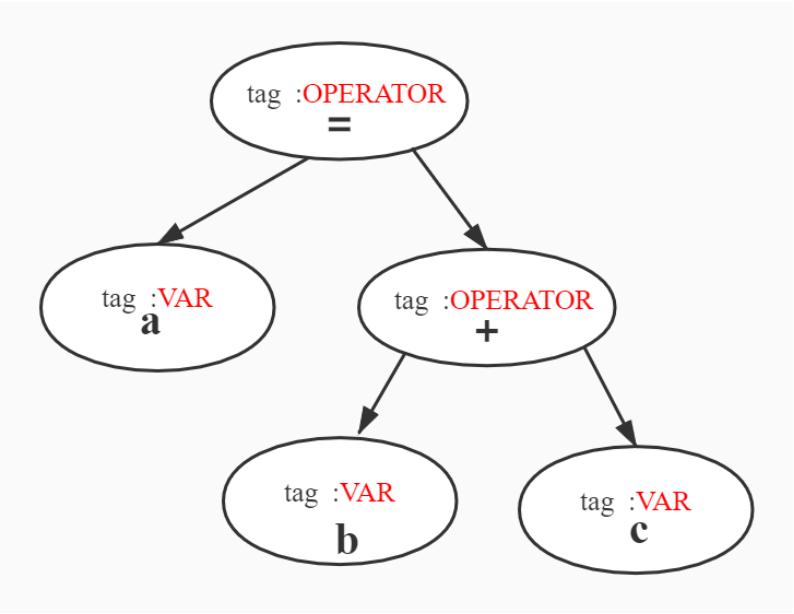
有两个子结点，第一个子结点存函数名，第二个子结点存实参序列。



21. ASTree Expression(Token_kind endsym)

处理表达式语句时以分号结束，作为条件表达式时以反小括号结束。采用栈的数据结构构造表达式树，定义操作数栈和运算符栈，每当遇到一个操作数，生成一个结点，将结点指针进操作数栈；每当遇到一个运算符，生成一个结点，比较其余运算符栈顶元素的优先级，优先级更低则结点入栈，更高则从操作数栈依次退栈两个元素作为该结点的左右孩子；相等则操作符栈退栈，去括号。最终生成一个根结点，如果没分析到错误则返回根结点，否则，返回 NULL。

例如，对于 a=b+c; 可得到如下子树。



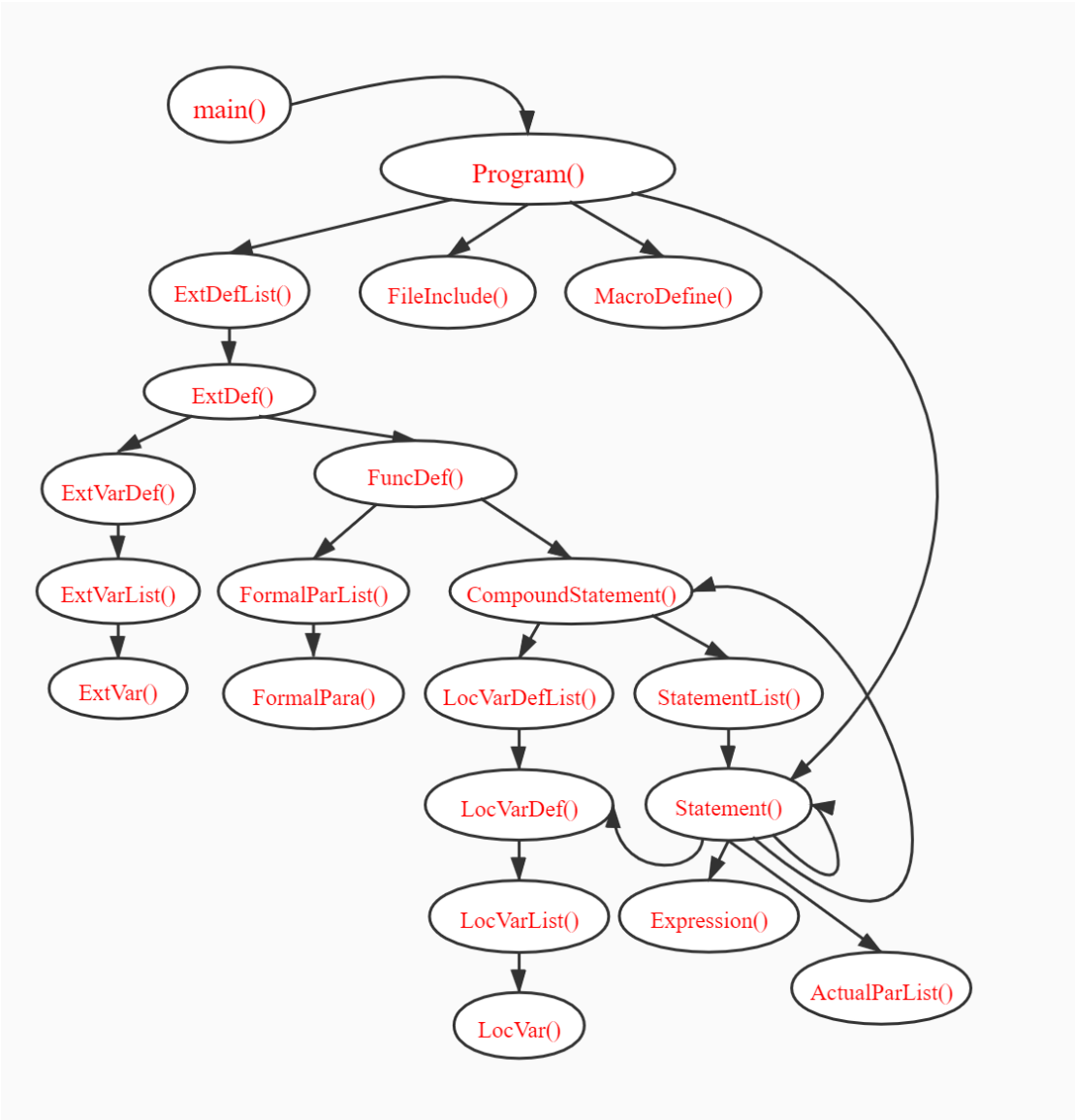
22. void Format(ASTree root, int dep)

先序遍历生成的语法树，根据语法树结点的标记，在输出文件按一定格式输出相应的内容，从而生成风格统一的格式化缩进编排的源程序文件。生成的文件中不包含注释。

23. void Writenote()

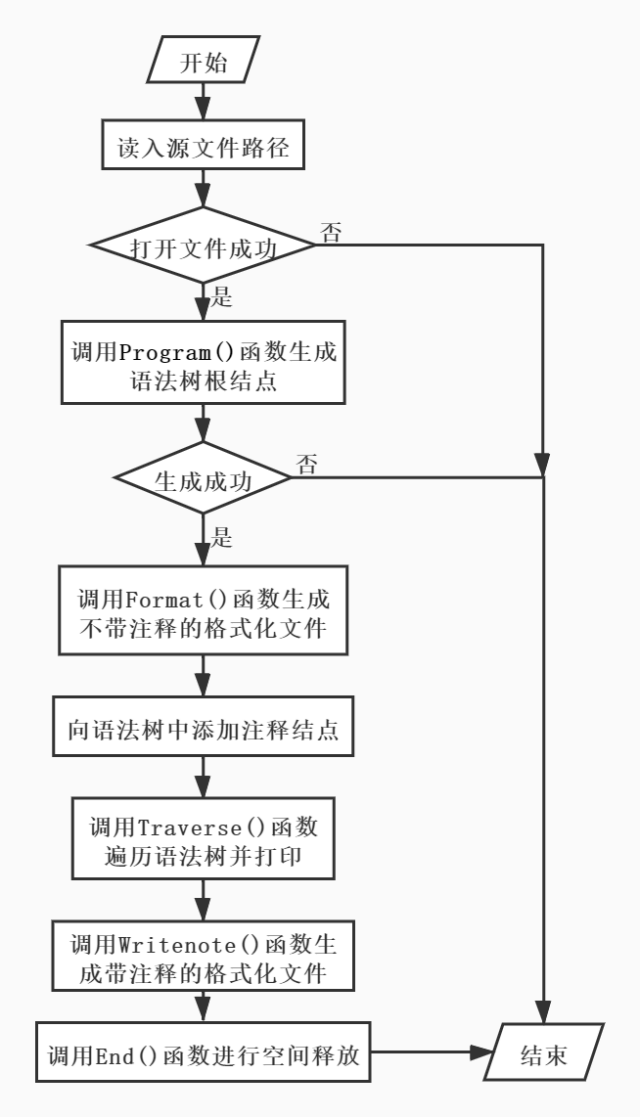
根据词法分析、语法分析过程中得到的注释列表 notes.notelist，从 Format 函数生成的文件中逐行读取内容，并记录读取行号、将内容写入新的输出文件，当行号与注释列表中的注释行号相同时，将注释写入输出文件。

4.1.6 函数调用关系



注：已省略辅助函数的调用。

4.1.7 main 函数



4.2 系统测试

注：

① 以下所有测试均使用同一测试文件 test.c，内容见附录 B。

② 测试方法采用断点调试、单步跳过（逐过程）执行。

③ 编译器采用 vscode。

4.2.1 词法分析

(1) 目标：分析出测试文件中所有单词并打印。

(2) 测试过程

由于词法分析函数是在语法分析过程中调用的，即在语法分析时，需要分析一个单词就调用一次词法分析函数。 词法分析函数在成功分析出一个单词后会调用 Info()函数进行打印。所以，只需在语法分析后，比对程序输出内容和测试文件即可判断是否有误。若有误，则进行单步调试。

①

```
17      Root = Program();
18      if (Root == NULL)      //生成失败
19          return 0;
```

在语法分析之后添加断点。

②

单词类别	单词值
井号	#
关键字	include
左尖括号	<
文件名	stdio.h
右尖括号	>
井号	#
关键字	include
左尖括号	<
文件名	stdlib.
右尖括号	>
井号	#
关键字	define
宏名	INF
宏名替换词	10000
井号	#
关键字	define
宏名	MAXN
宏名替换词	100
关键字	int
标识符	a
逗号	,
标识符	b
逗号	,
数组	g[10]
分号	;
注释符	/*

注释文本	test example
注释符	*/
关键字	long
标识符	c
逗号	,
数组	h[10]
分号	;
关键字	float
标识符	d
逗号	,
数组	i[10]
分号	;
关键字	double
标识符	f
逗号	,
数组	j[10]
分号	;
关键字	char
标识符	k
分号	;
标识符	a
赋值	=
整型常量	123
分号	;
标识符	b
赋值	=
整型常量	0x123
分号	;
标识符	c
赋值	=

长整型常量	123L
分号	;
标识符	f
赋值	=
浮点型常量	0.123F
分号	;
关键字	int
标识符	func_dec
左小括号	(
关键字	int
标识符	a
逗号	,
关键字	float
标识符	b
右小括号	)
分号	;
注释符	//
注释文本	function declaration
关键字	void
标识符	func_def
左小括号	(
右小括号	)
注释符	//
注释文本	function definition
左大括号	{
关键字	int
标识符	i
逗号	,
标识符	j
逗号	,

```
标识符      k
分号        ;
关键字      float
标识符      p
逗号        ,
标识符      q
逗号        ,
标识符      r
分号        ;
右大括号    }
关键字      int
标识符      main
左小括号    (
右小括号    )
左大括号    {
标识符      func_dec
左小括号    (
标识符      a
逗号        ,
标识符      d
右小括号    )
分号        ;
标识符      c
赋值        =
标识符      a
加号        +
标识符      b
分号        ;
关键字      if
左小括号    (
```

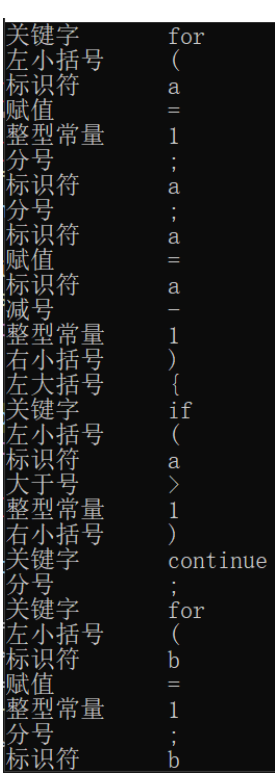
```
标识符      c
大于号      >
整型常量    1
右小括号    )
标识符      a
赋值        =
标识符      a
加号        +
整型常量    1
分号        ;
关键字      if
左小括号    (
标识符      a
大于号      >
标识符      b
与          &&
标识符      c
或          ||
标识符      d
小于号      <
标识符      f
右小括号    )
左大括号    {
标识符      a
赋值        =
标识符      a
大于号      >
标识符      b
与          &&
标识符      c
```

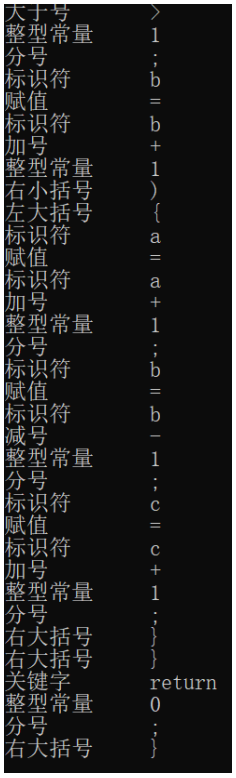
```
或          ||
标识符      d
小于号      <
标识符      f
与          &&
左小括号    (
标识符      a
乘号        *
标识符      b
除号        /
标识符      c
减号        -
整型常量    1
右小括号    )
分号        ;
标识符      b
赋值        =
标识符      b
加号        +
整型常量    1
分号        ;
右大括号    }
关键字      else
标识符      d
赋值        =
标识符      d
减号        -
整型常量    1
分号        ;
关键字      if
```

```
左小括号    (
标识符      a
大于号      >
整型常量    1
右小括号    )
左大括号    {
关键字      int
标识符      i
分号        ;
关键字      if
左小括号    (
标识符      b
大于号      >
整型常量    1
右小括号    )
左大括号    {
关键字      if
左小括号    (
标识符      c
大于号      >
整型常量    1
右小括号    )
标识符      a
赋值        =
标识符      a
加号        +
整型常量    1
分号        ;
关键字      else
标识符      b
```

```
赋值        =
标识符      b
加号        +
整型常量    1
分号        ;
右大括号    }
关键字      else
标识符      c
赋值        =
标识符      c
加号        +
整型常量    1
分号        ;
右大括号    }
关键字      while
左小括号    (
标识符      a
右小括号    )
左大括号    {
关键字      int
标识符      i
逗号        ,
标识符      j
分号        ;
关键字      for
左小括号    (
标识符      i
赋值        =
整型常量    1
分号        ;
```

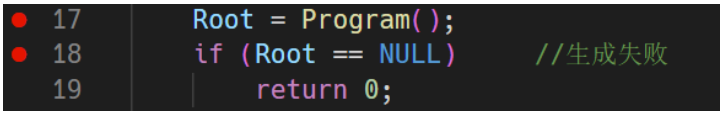
```
标识符      i
小于号      <
整型常量    10
分号        ;
标识符      i
赋值        =
标识符      i
加号        +
整型常量    1
右小括号    )
左大括号    {
标识符      a
赋值        =
标识符      a
加号        +
整型常量    1
分号        ;
右大括号    }
关键字      if
左小括号    (
标识符      a
大于号      >
整型常量    1
右小括号    )
关键字      break
分号        ;
关键字      else
关键字      continue
分号        ;
右大括号    }
```





运行程序，比对输出与测试文件内容。

③ 若无误，则结束。否则，在语法分析之前添加断点，进入分析程序进行单步调试。



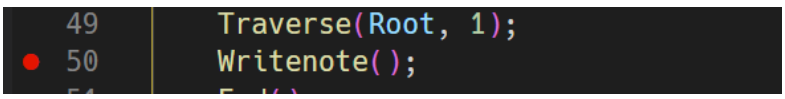
4.2.2 语法分析

(1) 目标

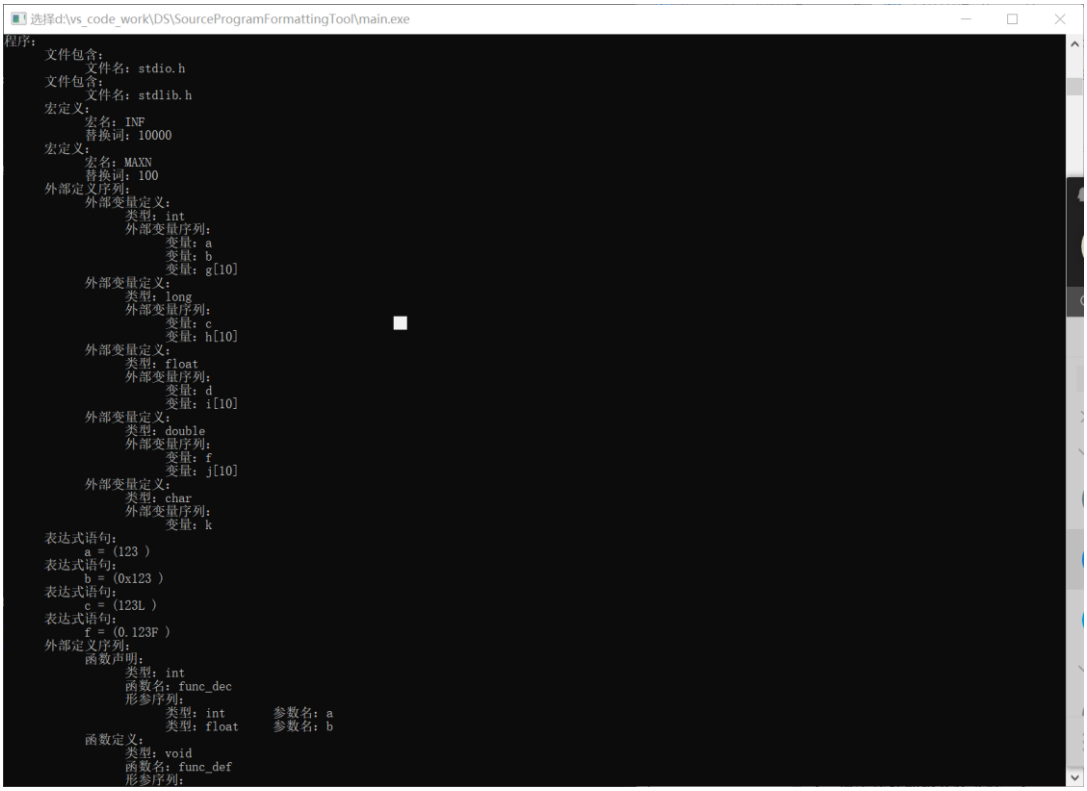
分析测试文件是否有语法错误，若有，则进行报错；若无，生成语法树，并打印树状结构。

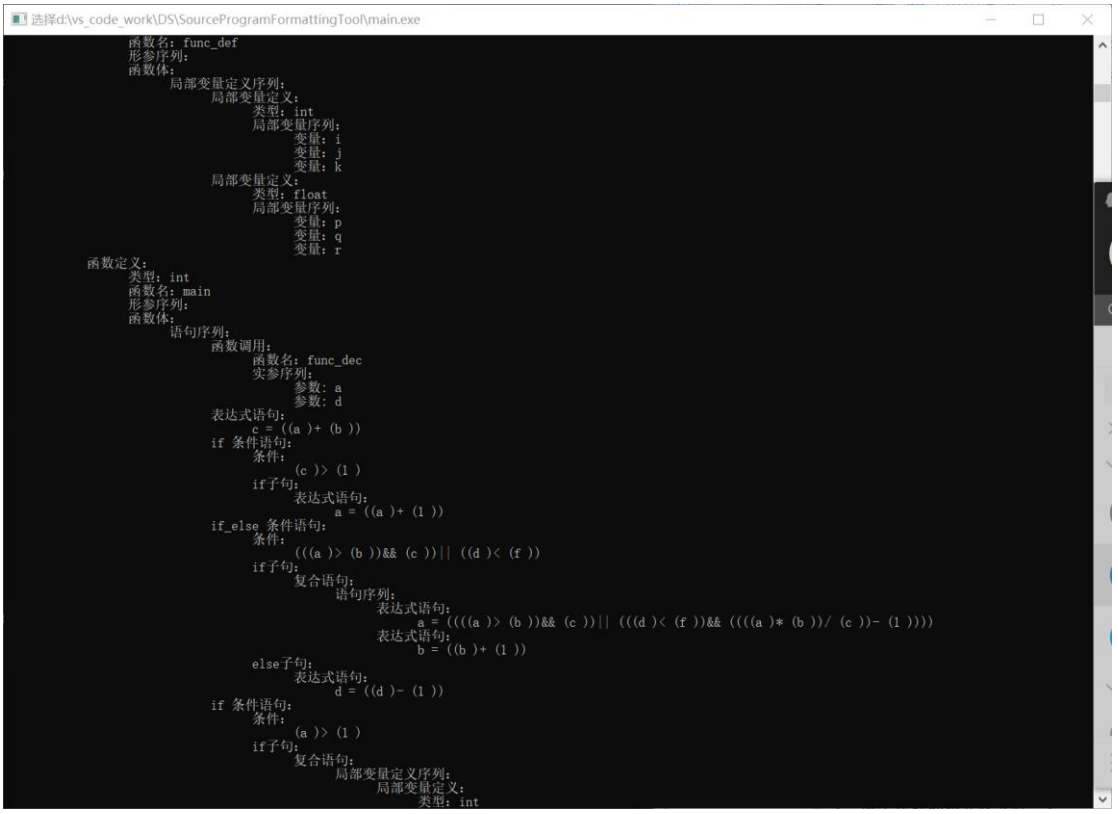
(2) 测试过程

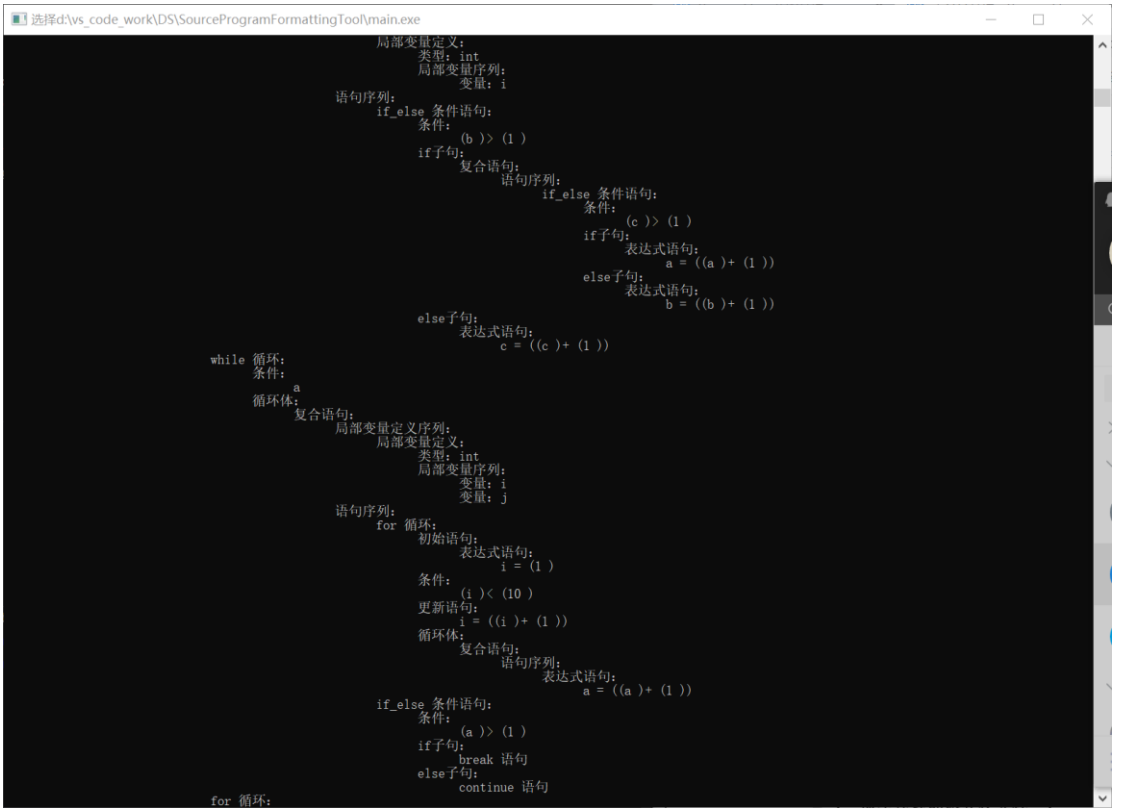
① 在遍历之后添加断点

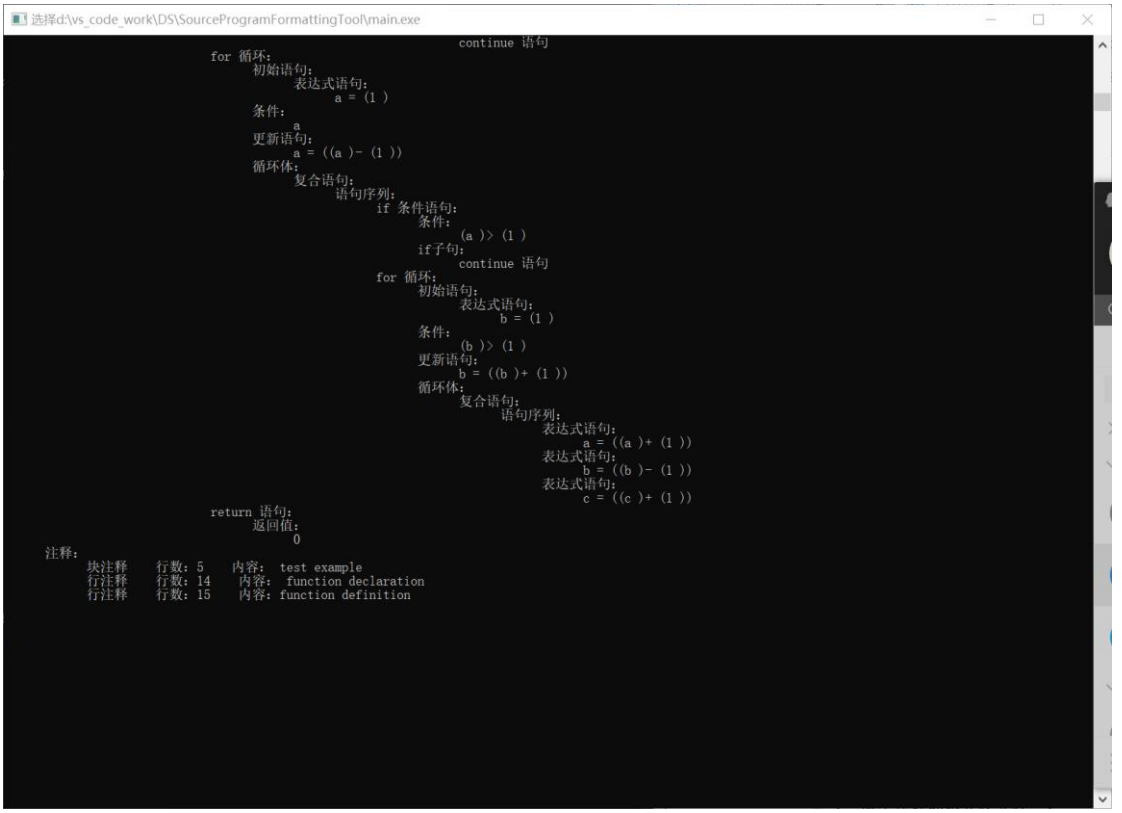


② 运行程序，比对程序输出与测试文件


选择d:\vs_code_work\DS\SourceProgramFormattingTool\main.exe
程序:
文件包含:
文件名: stdio.h
文件包含:
文件名: stdlib.h
宏定义:
宏名: INF
替换词: 10000
宏定义:
宏名: MAXN
替换词: 100
外部定义序列:
外部变量定义:
类型: int
外部变量序列:
变量: a
变量: b
变量: g[10]
外部变量定义:
类型: long
外部变量序列:
变量: c
变量: h[10]
外部变量定义:
类型: float
外部变量序列:
变量: d
变量: i[10]
外部变量定义:
类型: double
外部变量序列:
变量: f
变量: j[10]
外部变量定义:
类型: char
外部变量序列:
变量: k
表达式语句:
a = (123)
表达式语句:
b = (0x123)
表达式语句:
c = (123L)
表达式语句:
f = (0.123F)
外部定义序列:
函数声明:
类型: int
函数名: func_dec
形参序列:
类型: int 参数名: a
类型: float 参数名: b
函数定义:
类型: void
函数名: func_def
形参序列:


选择d:\vs_code_work\DS\SourceProgramFormattingTool\main.exe
函数名: func_def
形参序列:
函数体:
局部变量定义序列:
局部变量定义:
类型: int
局部变量序列:
变量: i
变量: j
变量: k
局部变量定义:
类型: float
局部变量序列:
变量: p
变量: q
变量: r
函数定义:
类型: int
函数名: main
形参序列:
函数体:
语句序列:
函数调用:
函数名: func_dec
实参序列:
参数: a
参数: d
表达式语句:
c = ((a)+ (b))
if 条件语句:
条件:
(c)> (1)
if子句:
表达式语句:
a = ((a)+ (1))
if_else 条件语句:
条件:
(((a)> (b))&& (c))|| ((d)< (f))
if子句:
复合语句:
语句序列:
表达式语句:
a = ((((a)> (b))&& (c))|| (((d)< (f))&& ((((a)* (b))/ (c))- (1))))
表达式语句:
b = ((b)+ (1))
else子句:
表达式语句:
d = ((d)- (1))
if 条件语句:
条件:
(a)> (1)
if子句:
复合语句:
局部变量定义序列:
局部变量定义:
类型: int





③ 若无误，则结束。否则，进行在有误部分进行单步测试。由于本程序将语法分析过程不同部分分配给了不同的函数，只需对相应函数进行测试。

(3) 报错测试

该部分通过修改测试文件后，运行程序，查看结果。

注：每次仅测试一个错误，第一张图为测试文件修改部分，第二张图为程序运行结果，测试完成后将测试文件改回。

①

```
2   #include <stdlib.h
```

```
ERROR!     第 2 行 第 20 列
缺少右尖括号!

请按任意键继续. . .
```

②

```
3   #deine INF 10000
```

```
ERROR!     第 3 行 第 7 列
语法错误! 应输入'define'或'include'!

请按任意键继续. . .
```

③

```
5   int a, b, g[10]
6   long c, h[10];
```

```
ERROR!     第 6 行 第 5 列
应输入逗号或分号!

请按任意键继续. . .
```

④

```
14   int func_dec(int a, float b;
```

```
ERROR!     第 14 行 第 29 列
应输入类型!

请按任意键继续. . .
```

⑤

```
14   int func_dec(int a, float b)
15   void func_def()
```

```
E R R O R !      第 15 行 第 5 列
应输入分号!

请按任意键继续. . .
```

⑥

```
22      func_dec(a, d;
```

```
E R R O R !      第 22 行 第 19 列
语法错误！应输入右括号或逗号或标识符或常量！

请按任意键继续. . .
```

⑦

```
24      if (c > )
```

```
E R R O R !      第 24 行 第 14 列
表达式不合法!

请按任意键继续. . .
```

⑧

```
39          els
40              b = b + 1;
```

```
E R R O R !      第 40 行 第 27 列
表达式不合法!

请按任意键继续. . .
```

⑨

```
5   int a, b, g[10];        /* test example
6   long c, h[10];
```

```
E R R O R !      第 69 行 第 1 列
应输入'*/'!

请按任意键继续. . .
```

⑩

```
3   #define 1 10000
```

```
E R R O R !      第 3 行 第 10 列
宏名不合法，应以字母或下划线开头!

请按任意键继续. . .
```

4.2.3 格式化文件生成

(1) 目标：根据语法树生成风格统一的格式化缩进编排的源程序文件。

(2) 测试过程

① 修改测试文件

```
1   #include <stdio.h    >
2   #include <    stdlib.h>
3   #define A                   10000
4   #define     MAXN 100
5   int a,    b, g[10];        /* test example */
6   long     c, h[10];
7   float d,     i[10];
8   double f, j[10];
9   char k;
10  a = 123    ;
```

```
15  void      func_def(        )
```

```
20  int
21  main( )
```

```
22  {
23  func_dec(a, d);
```

```
27      if     (a > b &&
28          c || d < f)
29      {
```

② 运行程序，查看生成的格式化文件

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #define A 10000     /* test example */
4   #define MAXN 100
5   int a, b, g[10];
6   long c, h[10];
7   float d, i[10];
8   double f, j[10];
9   char k;
10  a = (123 );
```

```
15  void func_def( )
```

```
20  int main()
21  {
```

```
21  {
22      func_dec(a, d);
```

```
26      if ((((a )> (b ))&& (c ))|| ((d )< (f )))
```

③ 无误则结束测试，否则，对相应函数进行单步调试。

5 总结与展望

5.1 全文总结

本次课程设计主要完成了基于抽象语法树、有穷自动机原理、递归下降分析法的高级语言源程序格式处理工具，主要工作如下：

（1）查阅了相关资料，了解了编译原理的一些基本知识，对递归下降分析法、有穷自动机原理等相关算法进行了实现。

（2）完成了系统的整体设计，将系统分模块处理，分块实现。

（3）设计了相关的数据类型，例如单词种类的枚举类型、结点标记的枚举类型、语法树结点类型、栈类型等。

（4）设计了语法分析过程中，相关语法的存储结构。

（4）声明和定义了许多函数，包括 21 个辅助函数和 23 个主要函数。

（5）设计了测试文件，便于系统的测试。

5.2 工作展望

本程序只初步实现了 C 语言的格式化处理，还有许多地方可以改进：

（1）在对 C 语言的程序处理中，只实现了基本的一些语句，一些高级的成分，如结构体、枚举类型、指针等，以及其它一些语句，如 switch_case 语句等待实现。

（2）本程序以编写者本身规定的语法规则对C语言源程序进行格式化处理，可以改进为用户输入其定义的语法规则并选择高级语言，程序进行解析并根据该语法规则对源程序进行语法分析。

（3）程序采用递归下降分析法，当源文件较大时，会调用大量的堆栈，可能会引起栈溢出，可以尝试非递归算法。

6 体会

这一次的课程设计较以前的 C 语言程序设计实验和数据结构实验相比，难度更大，工程量更大，体会如下：

第一，对于所有的程序设计，我认为前期的设计和准备是非常重要的。在本次实验中，我花费了较多的时间在数据类型的定义和程序框架的设计上。在真正开始编写程序之前，我反复阅读了任务书，写出了较为完备的初版头文件，后期也没有因为理解错了题意而 remake。

第二，在程序正式开始编写的前期，我发现自己最初的设计有许多的缺陷，因此，我通过不断的修改，不断的调试，给出了最终的相关定义，之后的修改都只是在此基础上进行增删。

第三，在之后的程序编写过程中，因为有了明确的相关定义和设计，函数的实现比较快，这部分推进迅速。

第四，在最后的程序完善阶段，我发现了程序中的许多错误，因为程序大量使用了申请的内存空间，所以在 debug 的过程中，经常会有程序直接崩溃的情况，也是在众多函数中进行逐个断点调试，才得出了最终的程序（也可能还有没发现的 bug）。

总的来说，对于这次的程序设计，我的收获还是蛮大的：

首先，巩固了 C 语言和数据结构方面的知识。对结构体、指针、多文件编译有了更深入的了解。第二，程序设计能力得以锻炼，本次实验不像之前的实验都给好了框架，需要自己设计相关的定义和数据结构，所以较大的提升了程序设计能力。第三，锻炼了自己的学习能力，在实验中遇到了许多不懂的问题，需要自己去查阅相关的文献资料。

最后，非常感谢老师、助教的辛勤付、付出和同学的指导交流。

参考文献

[1]刘雪飞. 浅谈编译原理. 2018.03 中外企业家.

[2]崔光宇.编译原理教学现状与创新研究. 2017.12 Wireless Internet Technology

[3] 严蔚敏等.数据结构（C 语言版）.清华大学出版社

[4] 高级语言源程序格式处理工具实验指导

附录 A 源代码

A.1 def.h

```
/* 本文件内容为程序相关定义以及函数声明 */

#ifndef _DEF_H_
#define _DEF_H_

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#define INIT_SIZE 20        //初始化大小
#define INCREASEMENT 10     //每次增加的大小

/* 程序相关定义 */

typedef enum Bool { FALSE, TRUE } Bool;     //自定义布尔类型

typedef enum Token_kind        //各类单词种类的枚举类型
/*
   关键字：int、float、double、char、long、void、if、else、while、for、return、
break、continue、define、include、const。
   类型和标识符：数组、整形数组、浮点型数组、字符串、字符串常量、错误
符号、标识符、整形常量、浮点型常量、长整形常量、双精度浮点常量。
   运算符：==、!=、>、<、+、-、*、/、=、(、)、&&、||、!、#、[、]、
{、}、;、，、%、++、--、’、”、<，>。
   注释：//、块注释符 1、快注释符 2、注释文本
*/
{
   INT, FLOAT, DOUBLE, CHAR, LONG, VOID, IF, ELSE, WHILE, FOR,
RETURN,
   BREAK, CONTINUE, DEFINE, INCLUDE, CONST,                    //部分关
键字
```

```
    ARRAY, INT_ARRAY, FLOAT_ARRAY, STRING, STRING_CONST,          //
数组类型
    ERROR_TOKEN, IDENT, INT_CONST, FLOAT_CONST, LONG_CONST,
DOUBLE_CONST, //标识符和各类型常量
    EQ, UEQ, LH, RH, PLUS, SUB, MUL, DIV, ASSIGN, LP, RP,          //各种符
号
    AND, OR, NOT, WELL, LSB, RSB, LCB, RCB, SEMI, COMMA, MOD,
    PLUSPLUS, SUBSUB, SINGLE_QUOTE, DOUBLE_QUOTE, LAB, RAB,
    NOTE1, NOTE2, NOTE3, NOTETEXT
}Token_kind;

typedef enum Node_type        //结点类型
/*
  程序、外部定义序列、外部变量定义、外部变量序列、函数定义、形参序列、
  复合语句、局部变量定义序列、局部变量定义、局部变量序列、实参序列、
  语句序列、if 子句、else 子句、循环体、for 初始语句、for 更新语句、返回值、
  表达式、函数调用、函数、形参、外部变量、局部变量、实参、类型、变量、
  操作符、ifelse 语句、赋值、while 语句、for 语句、if 语句、条件、return 语句、
break 语句、
  continue 语句、include 语句、define 语句、文件名、宏名、宏名替换词、注释
内容、注释
*/
{
  _PROGRAMMAR_, _EXTDEFLIST_, _EXTVARDEF_, _EXTVARLIST_,
_FUNCDEF_, _FORMALPARLIST_,
  _COMPOUNDSTATEMENT_, _LOCVARDEFLIST_, _LOCVARDEF_,
_LOCVARLIST_, _ACTUALPARLIST_,
  _STATEMENTLIST_, IFCLAUSE, ELSECLAUSE, LOOPBODY, _FOR_ONE_,
_FOR_TWO_, _RETURNVAR_,
  _EXPRESSION_, _FUNCCALL_, FUNC, FORPARA, EXTVAR, LOCVAR,
ACTPARA, TYPE, VAR,
  OPERATOR, IFELSE, _ASSIGN_, _WHILE_, _FOR_, _IF_, CONDITION,
_RETURN_, _BREAK_,
  _CONTINUE_, _INCLUDE_, _DEFINE_, _FILENAME_, _DEFINE_ONE_,
_DEFINE_TWO_, NOTE, _NOTE_
}Node_type;

struct Func
```

```
{
   char** funclist;       //函数名列表
   int num, size;         //数量和大小
} func = { NULL,0,0 };     //全局变量，保存源文件出现的函数

struct Note
{
   char* data;       //注释内容
   int row, type;    //注释出现的行、注释类型
};     //注释结构体

struct NoteList
{
   struct Note* notelist;      //注释列表
   int num, size;              //数量和大小
}notes = { NULL,0,0 };         //全局变量，保存源文件出现的注释

typedef struct  ASTNode      //抽象语法树结点类型定义
{
   int tag;  //结点数据类型标记
   union     //结点数据
   {
      char ele_name[50];   //语法成分
      char var_name[50];   //变量名
      char func_name[50];  //函数名
      int op;              //运算符
      int var_type;        //变量类型
      struct Note note;    //注释
   }data;
   struct ASTNode* child, * sibling;     //孩子兄弟表示法
}ASTNode, * ASTree;

typedef struct Stack
{
   ASTree* ptr;       //存储元素
   int top, size;     //栈顶、栈容量
}Stack;               //自定义栈
```

```
//全局变量
FILE* fin, * fout;                          //指向读入文件和输出文件的指针
ASTree Root = NULL;                         //语法树的根结点
Bool error = FALSE;                         //错误标记
char sourcefile[100];                       //源文件路径
char* token_text = NULL, * token_text0 = NULL;    //当前和前一单词的字面值
int token_size = 0;                         //当前 token_text 的大小
int row = 1, col = 1;                       //记录当前读取行列号
int row0 = 1;                               //格式化文件行数
int type, type0, type1 = -1;                //暂时记录某一单词

const char* keyword[] =
{
   "int","float","double","char","long","void","if","else",
   "while", "for","return","break","continue","define",
   "include","const",NULL
};          //根据枚举顺序初始化关键字查找表

/* 函数声明 */

//辅助函数
Bool IsType(int t);                         //判断是否是类型关键字
Bool IsOperater(int t);                     //判断是否是运算符
Bool IsFunc(char* s);                       //判断是否是函数名
Bool Pop(Stack* p, ASTree* t);              //出栈
ASTree New();                               //创建新结点并初始化
ASTree Gettop(const Stack s);               //获取栈顶元素
int Find_keyword();                         //查找关键字
void InitStack(Stack* p);                   //初始化栈
void Push(Stack* p, ASTree q);              //入栈
void Add_char(const char c);                //拼接字符
void Copy();                                //复制单词
void Info(int type);                        //输出 token_text 信息
void Traverse(ASTree p, int dep);           //遍历语法树
char Precede(int c1, int c2);               //求运算符之间的优先级
void traverse(ASTree root);                 //遍历表达式树
```

```
void ftraverse(ASTree root);              //遍历表达式树输入到文件中
void Addfunc(char* s);                    //添加一个函数
void Addnote(char* s, int r, int t);      //添加一个注释
void Warning(const char* s);              //报错
void End();                               //程序结束释放空间并关闭文件
void Destroy(ASTree root);                //销毁语法树


//词法分析
int Get_token();                          //词法分析

//语法分析
ASTree Program();                         //程序
ASTree FileInclude();                     //文件包含
ASTree MacroDefine();                     //宏定义
ASTree ExtDefList();                      //外部定义序列
ASTree ExtVarDef();                       //外部变量定义
ASTree ExtVarList();                      //外部变量序列
ASTree ExtVar();                          //外部变量
ASTree ExtDef();                          //外部定义
ASTree LocVarDefList();                   //局部变量定义序列
ASTree LocVarDef();                       //局部变量定义
ASTree LocVarList();                      //局部变量序列
ASTree LocVar();                          //局部变量
ASTree FuncDef();                         //函数定义
ASTree FormalParList();                   //形参序列
ASTree FormalPara();                      //形参
ASTree ActualParList();                   //实参序列
ASTree CompoundStatement();               //复合语句
ASTree StatementList();                   //语句序列
ASTree Statement();                       //语句
ASTree Expression(Token_kind endsym);     //表达式

//格式化源文件
void Format(ASTree root, int dep);        //格式化源文件
void Writenote();                         //将注释写入格式化文件
```

```
#endif
```

A.2 assist.h

```
/* 本文件内容为程序所需辅助函数的定义 */

#ifndef _ASSIST_H_
#define _ASSIST_H_
#include "def.h"

void Add_char(const char c)
//拼接字符
{
   if (token_text == NULL)                          //token_text 为空则进行初始化
   {
      token_text = (char*)malloc(sizeof(char) * INIT_SIZE);
      token_size = INIT_SIZE;
      token_text[0] = c;                            //添加字符
      token_text[1] = '\0';                         //形成字符串
      return;
   }
   int len;
   if ((len = strlen(token_text)) == token_size - 1)      //token_text 满了
   {
      token_text = (char*)realloc(token_text,           //重新分配空间
         sizeof(char) * (INCREASEMENT + token_size));
      token_size += INCREASEMENT;
   }
   token_text[len] = c;
   token_text[len + 1] = '\0';
}

void Addfunc(char* s)
//添加一个函数
{
   if (func.funclist == NULL)         //函数列表为空则进行初始化
   {
```

```
        func.funclist = (char**)malloc(sizeof(char*) * INIT_SIZE);
        func.size = INIT_SIZE;
    }
    if (func.num == func.size)     //列表满了则重新分配空间
    {
        func.funclist = (char**)realloc(func.funclist, sizeof(char*)
            * (func.size + INCREASEMENT));
        func.size += INCREASEMENT;
    }
    func.funclist[func.num] = (char*)malloc(sizeof(char)
        * (strlen(s) + 1));
    strcpy(func.funclist[func.num], s);
    func.num++;
}

void Addnote(char* s, int r, int t)
//添加一个注释
{
    if (notes.notelist == NULL)    //注释列表为空则进行初始化
    {
        notes.notelist = (struct Note*)malloc(
            sizeof(struct Note) * INIT_SIZE);
        notes.size = INIT_SIZE;
    }
    if (notes.size == notes.num)   //列表满了则重新分配空间
    {
        notes.notelist = (struct Note*)realloc(notes.notelist,
            sizeof(struct Note) * (notes.size + INCREASEMENT));
        notes.size += INCREASEMENT;
    }
    notes.notelist[notes.num].data = (char*)malloc(sizeof(char) * (strlen(s) + 1));
    notes.notelist[notes.num].row = r, notes.notelist[notes.num].type = t;
    strcpy(notes.notelist[notes.num].data, s);
    notes.num++;
}

void Copy()
//复制单词,将 token_text 的内容复制到 token_text0
{
    if (token_text0)    free(token_text0);
```

```
    token_text0 = (char*)malloc(
        sizeof(char) * (strlen(token_text) + 1));
    strcpy(token_text0, token_text);
}

int Find_keyword()
//查找关键字
{
    int i = 0;
    while (keyword[i] && strcmp(keyword[i], token_text))
        i++;
    if (keyword[i] == NULL)    return -1;    //查找失败
    return i;                              //查找成功
}

Bool IsType(int t)
//判断是否是类型关键字
{
    if (t >= INT && t <= VOID)    return TRUE;
    return FALSE;
}

Bool IsOperater(int t)
//判断是否是运算符
{
    if (t >= EQ && t <= WELL)   return TRUE;
    return FALSE;
}

Bool IsFunc(char* s)
//判断是否是函数名
{
    for (int i = 0;i < func.num;i++)
        if (strcmp(func.funclist[i], s) == 0)
            return TRUE;
    return FALSE;
}

void InitStack(Stack* p)
```

```
//初始化栈
{
   p->ptr = (ASTree*)malloc(sizeof(ASTree) * INIT_SIZE);
   p->top = 0;   p->size = INIT_SIZE;
}

ASTree Gettop(const Stack s)
//获取栈顶元素
{
   if (s.top == 0)  return NULL;
   return s.ptr[s.top - 1];
}

Bool Pop(Stack* p, ASTree* t)
//出栈
{
   if (p->top == 0)   return FALSE;
   (*t) = p->ptr[--(p->top)];
   return TRUE;
}

void Push(Stack* p, ASTree q)
//入栈
{
   if (p->top == p->size)
      p->ptr = (ASTree*)realloc(p->ptr, sizeof(ASTree) *
         (p->size + INCREASEMENT)),
      p->size += INCREASEMENT;
   p->ptr[p->top++] = q;
}

char Precede(int t1, int t2)
//求运算符之间的优先级
{
   switch (t1)
   {
   case PLUS:case SUB:
      switch (t2)
      {
      case PLUS:case SUB:case RP:case LH:case RH:
```

```
    case EQ:case UEQ:case WELL:case AND:case OR:
        return '>';
    case MUL:case DIV:case LP: return '<';
    default:return ' ';
    }
case MUL:case DIV:
    switch (t2)
    {
    case PLUS:case SUB:case RP:case LH:case RH:
    case EQ:case UEQ:case WELL:case MUL:case DIV:
    case AND:case OR:    return '>';
    case LP: return '<';  default:return ' ';
    }
case LP:
    switch (t2)
    {
    case PLUS:case SUB:case MUL:case DIV:case LP:
    case AND:case OR:   return '<';
    case RH:case LH:case EQ:case UEQ:case WELL:
        return '>';
    case RP: return '=';
    default:return ' ';
    }
case RP:
    switch (t2)
    {
    case PLUS:case SUB:case DIV:case LH:case RH:
    case EQ:case UEQ:case WELL:case MUL:case LP:
    case AND:case OR:    return '>';
    default:return ' ';
    }
case ASSIGN:
    switch (t2)
    {
    case PLUS:case SUB:case DIV:case LH:case RH:
    case EQ:case UEQ:case ASSIGN:case MUL:case LP:
    case AND:case OR:    return '<';
    case WELL:case RP:return '>';
    default:return ' ';
    }
```

```
case LH:case RH:
   switch (t2)
   {
   case PLUS:case SUB:case MUL:case DIV:case LP:
      return '<';
   case RH:case LH:case EQ:case UEQ:case WELL:case RP:
   case AND:case OR:    return '>';
   default:return ' ';
   }
case EQ:case UEQ:
   switch (t2)
   {
   case PLUS:case SUB:case MUL:case DIV:case LP:
   case RH:case LH:   return '<';
   case EQ:case UEQ:case WELL:case RP:
   case AND:case OR:    return '>';
   default:return ' ';
   }
case WELL:
   switch (t2)
   {
   case PLUS:case SUB:case DIV:case LH:case RH:
   case EQ:case UEQ:case ASSIGN:case MUL:case LP:
   case AND:case OR:    return '<';
   case WELL:return '=';
   default:return ' ';
   }
case AND:
   switch (t2)
   {
   case PLUS:case SUB:case MUL:case DIV:case LP:
   case ASSIGN:case LH:case RH:case EQ:case UEQ:
      return '<';
   case RP:case WELL:case AND:case OR:
      return '>';
   }
case OR:
   switch (t2)
   {
   case PLUS:case SUB:case MUL:case DIV:case LP:
```

```
        case ASSIGN:case LH:case RH:case EQ:case UEQ:
        case AND:    return '<';
        case RP:case WELL:case OR:
            return '>';
        }
    default:return ' ';
    }
}

void Info(int t)
//输出 token_text 信息
{
    static Bool begin = FALSE;
    if (begin == FALSE)
    {
        begin = TRUE;
        printf("%-12s 单词值\n\n", "单词类别");
    }
    switch (t)
    {
    case INT: printf("%-13sint\n", "关键字");break;
    case FLOAT: printf("%-13sfloat\n", "关键字");break;
    case DOUBLE: printf("%-13sdouble\n", "关键字");break;
    case CHAR: printf("%-13schar\n", "关键字");break;
    case LONG: printf("%-13slong\n", "关键字");break;
    case VOID:printf("%-13svoid\n", "关键字");break;
    case IF: printf("%-13sif\n", "关键字");break;
    case ELSE: printf("%-13selse\n", "关键字");break;
    case WHILE: printf("%-13swhile\n", "关键字");break;
    case FOR: printf("%-13sfor\n", "关键字");break;
    case RETURN: printf("%-13sreturn\n", "关键字");break;
    case BREAK: printf("%-13sbreak\n", "关键字");break;
    case CONTINUE: printf("%-13scontinue\n", "关键字");break;
    case DEFINE:printf("%-13sdefine\n", "关键字");break;
    case INCLUDE:printf("%-13sinclude\n", "关键字");break;
    case CONST:printf("%-13sconst\n", "关键字");break;
    case ARRAY: printf("%-13s%s\n", "数组", token_text);break;
    case IDENT:printf("%-13s%s\n", "标识符", token_text);break;
```

```
    case INT_CONST:printf("%-13s%s\n", "整型常量", token_text);break;
    case LONG_CONST:printf("%-13s%s\n", "长整型常量", token_text);break;
    case FLOAT_CONST:printf("%-13s%s\n", "浮点型常量", token_text);break;
    case DOUBLE_CONST:printf("%-13s%s\n", "长双精度浮点型常量", token_text);break;
    case EQ:printf("%-13s==\n", "等号");break;
    case LH:printf("%-13s>\n", "大于号");break;
    case RH:printf("%-13s<\n", "小于号");break;
    case PLUS:printf("%-13s+\n", "加号");break;
    case PLUSPLUS:printf("%-13s++\n", "自增");break;
    case SUB:printf("%-13s-\n", "减号");break;
    case SUBSUB:printf("%-13s--\n", "自减");break;
    case MUL:printf("%-13s*\n", "乘号");break;
    case DIV:printf("%-13s/\n", "除号");break;
    case MOD:printf("%-13s%%\n", "取余");break;
    case ASSIGN:printf("%-13s=\n", "赋值");break;
    case LP:printf("%-13s(\n", "左小括号");break;
    case RP:printf("%-13s)\n", "右小括号");break;
    case LSB:printf("%-13s[\n", "左中括号");break;
    case RSB:printf("%-13s]\n", "右中括号");break;
    case LCB:printf("%-13s{\n", "左大括号");break;
    case RCB:printf("%-13s}\n", "右大括号");break;
    case LAB:printf("%-13s<\n", "左尖括号");break;
    case RAB:printf("%-13s>\n", "右尖括号");break;
    case SEMI:printf("%-13s;\n", "分号");break;
    case COMMA:printf("%-13s,\n", "逗号");break;
    case WELL:printf("%-13s#\n", "井号");break;
    case AND:printf("%-13s&&\n", "与");break;
    case OR:printf("%-13s||\n", "或");break;
    case NOT:printf("%-13s!\n", "非");break;
    case SINGLE_QUOTE:printf("%-13s\'\n", "单引号");break;
    case DOUBLE_QUOTE:printf("%-13s\"\n", "双引号");break;
    case NOTE1:printf("%-13s//\n", "注释符");break;
    case NOTE2:printf("%-13s/*\n", "注释符");break;
    case NOTE3:printf("%-13s*/\n", "注释符");break;
    case NOTETEXT:printf("%-13s%s\n", "注释文本", token_text);break;
```

```
    }
}

ASTree New()
//创建新结点并初始化
{
    ASTree p = (ASTree)malloc(sizeof(ASTNode));
    p->child = p->sibling = NULL;
    return p;
}

void Traverse(ASTree p, int dep)
//遍历语法树
{
    if (p == NULL)  return;
    for (int i = 1;i < dep;i++)
        printf("      ");      //输出前置空格
    switch (p->tag)
    //根据结点类型输出相应内容
    {
    case _WHILE_:case _FOR_:case _ASSIGN_:case _IF_:case IFELSE:
    case     _EXTDEFLIST_:case     _EXTVARDEF_:case     _EXTVARLIST_:case
_FUNCDEF_:
    case          _FORMALPARLIST_:case          _COMPOUNDSTATEMENT_:case
_LOCVARDEFLIST_:
    case _LOCVARDEF_:case _LOCVARLIST_:case _ACTUALPARLIST_:
    case     _STATEMENTLIST_:case     IFCLAUSE:case     ELSECLAUSE:case
LOOPBODY:
    case _FOR_ONE_:case _FOR_TWO_:case _RETURNVAR_:
    case     _FUNCCALL_:case     _PROGRAMMAR_:case     _FILENAME_:case
_DEFINE_ONE_:
    case _DEFINE_TWO_:case _NOTE_:  printf("%s：\n", p->data.ele_name);break;
    case CONDITION:case _EXPRESSION_:
        printf("%s：\n", p->data.ele_name);
        for (int i = 1;i < dep + 1;i++)
            printf("      ");
        traverse(p->child); putchar('\n'); return;
    case _BREAK_:case _CONTINUE_:
        printf("%s\n", p->data.ele_name);break;
```

```
case _RETURN_:
    printf("%s：\n", p->data.ele_name);
    for (int i = 1;i < dep + 1;i++)
        printf("      ");
    printf("%s：\n", p->child->data.ele_name);
    for (int i = 1;i < dep + 2;i++)
        printf("      ");
    traverse(p->child->child);
    putchar('\n'); return;
case _INCLUDE_:
    printf("%s：\n", p->data.ele_name);
    for (int i = 1;i < dep + 1;i++)
        printf("      ");
    printf("%s：%s\n", "文件名", p->child->data.ele_name);
    return;
case _DEFINE_:
    printf("%s：\n", p->data.ele_name);
    for (int i = 1;i < dep + 1;i++)
        printf("      ");
    printf("%s：%s\n", "宏名", p->child->data.ele_name);
    for (int i = 1;i < dep + 1;i++)
        printf("      ");
    printf("%s：%s\n", "替换词", p->child->sibling->data.ele_name);
    return;
case TYPE:
    printf("%s：", "类型");
    switch (p->data.var_type)
    {
    case INT:printf("%s\n", "int");break;
    case FLOAT:printf("%s\n", "float");break;
    case DOUBLE:printf("%s\n", "double");break;
    case CHAR:printf("%s\n", "char");break;
    case LONG:printf("%s\n", "long");break;
    case VOID:printf("%s\n", "void");break;
    }
    break;
case FORPARA:
    printf("%s：", "类型");
    switch (p->child->data.var_type)
```

```
        {
        case INT:printf("%s\t", "int");break;
        case FLOAT:printf("%s\t", "float");break;
        case DOUBLE:printf("%s\n", "double");break;
        case CHAR:printf("%s\t", "char");break;
        case LONG:printf("%s\n", "long");break;
        case VOID:printf("%s\t", "void");return;
        }
        printf("%s：%s\n", "参数名", p->child->sibling->data.var_name);
        return;
    case ACTPARA:printf("%s: %s\n", "参数", p->data.var_name);break;
    case VAR:printf("%s：\n", p->data.var_name);break;
    case LOCVAR:
    case EXTVAR:
        printf("%s：%s\n", "变量", p->data.var_name);break;
    case FUNC:printf("%s：%s\n", "函数名", p->data.func_name);break;
    case OPERATOR:traverse(p);putchar('\n');return;
    case NOTE:
        if (p->data.note.type == 1)
            printf("行注释　行数：%d　内容：%s\n",
                p->data.note.row, p->data.note.data);
        else
            printf("块注释　行数：%d　内容：%s\n",
                p->data.note.row, p->data.note.data);
        break;
    }
    Traverse(p->child, dep + 1);      //遍历第一个孩子结点
    ASTree sib = NULL;
    if (p->child)
        sib = p->child->sibling;
    while (sib)    //遍历剩余孩子结点
    {
        Traverse(sib, dep + 1);
        sib = sib->sibling;
    }
}

void traverse(ASTree root)
//先序遍历表达式树
```

```
{
   if (root == NULL)  return;
   if (root->child)
   {
      if (root->data.op != ASSIGN)    //变量之间加括号,防止优先级出错
         putchar('(');
      traverse(root->child);
      if (root->data.op != ASSIGN)
         putchar(')');
   }
   if (root->tag == VAR)
      printf("%s ", root->data.var_name);
   else
   {
      switch (root->data.op)
      {
      case EQ:printf("==");break;case UEQ:printf("!=");break;
      case LH:putchar('>');break;case RH:putchar('<');break;
      case PLUS:putchar('+');break;case SUB:putchar('-');break;
      case MUL:putchar('*');break;case DIV:putchar('/');break;
      case AND:printf("&&");break;case OR:printf("||");break;
      case ASSIGN:putchar('=');break;
      }
      putchar(' ');
   }
   if (root->child)
   {
      putchar('(');
      traverse(root->child->sibling);    //遍历剩余孩子
      putchar(')');
   }
}

void ftraverse(ASTree root)
//遍历表达式树输入到文件中（实现同上）
{
   if (root == NULL)  return;
   if (root->child)
   {
      if (root->data.op != ASSIGN)
```

```
            fputc('(', fout);
        ftraverse(root->child);
        if (root->data.op != ASSIGN)
            fputc(')', fout);
    }
    if (root->tag == VAR)
        fprintf(fout, "%s ", root->data.var_name);
    else
    {
        switch (root->data.op)
        {
        case EQ:fprintf(fout, "==");break;case UEQ:fprintf(fout, "!=");break;
        case LH:fputc('>', fout);break;case RH:fputc('<', fout);break;
        case PLUS:fputc('+', fout);break;case SUB:fputc('-', fout);break;
        case MUL:fputc('*', fout);break;case DIV:fputc('/', fout);break;
        case AND:fprintf(fout, "&&");break;case OR:fprintf(fout, "||");break;
        case ASSIGN:fputc('=', fout);break;
        }
        fputc(' ', fout);
    }
    if (root->child)
    {
        fputc('(', fout);
        ftraverse(root->child->sibling);
        fputc(')', fout);
    }
}

void Warning(const char* s)
//报错，根据形参内容报出相应错误
{
    error = TRUE;
    printf("\n\nE R R O R !\t 第 %d 行 第 %d 列\n%s\n\n", row, col, s);
    system("pause");
    putchar('\n');
}

void End()
//程序结束释放空间
{
```

```
    free(notes.notelist);
    for (int i = 0;i < func.num;i++)
        free(func.funclist[i]);
    if (func.num)     free(func.funclist);
    if (token_text)   free(token_text);
    if (token_text0)  free(token_text0);
    Destroy(Root);
}

void Destroy(ASTree root)
//销毁语法树
{
    if (root == NULL)  return;
    ASTree p = NULL, q = NULL;
    if (root->child)  p = root->child->sibling;
    Destroy(root->child);
    free(root);
    while (p)
    {
        q = p->sibling;
        Destroy(p);
        p = q;
    }
}

#endif
```

A.3 func.h

```
/* 本文件内容为程序所需主要函数的定义 */

#ifndef _FUNC_H_
#define _FUNC_H_
#include "def.h"
#include "assist.h"

int Get_token()
```

```
//词法分析
{
   if (token_text)  free(token_text);
   token_text = NULL;                    //初始化 token_text 为空
   int c;
   while ((c = fgetc(fin)) != EOF)
   //过滤空白符号
   {
      col++;                             //更新列数
      if (c == '\n')    col = 1, row++;   //更新列数行数
      if (!isspace(c))    break;
   }
   if (c == '/')
   {
      c = fgetc(fin); col++;
      if (c != '*' && c != '/')
      {
         ungetc(c, fin);col--;
         Info(DIV);     //不是注释，则返回除号
         return DIV;
      }
      if (c == '/')  //行注释
      {
         Info(NOTE1);
         int r0 = row;
         c = fgetc(fin);
         while (c != '\n' && c != EOF)
         {
            Add_char(c);
            c = fgetc(fin);
         }
         if (c == '\n') { row++, col = 1; }
         Addnote(token_text, r0, 1);
         Info(NOTETEXT);
         return Get_token();
      }
      else    //块注释
      {
         Info(NOTE2);
```

```
        int c0, r0 = row;
        c0 = c = fgetc(fin);col++;
        while (c != EOF)
        {

            if (c0 == '*' && c == '/')
            {
                token_text[strlen(token_text) - 1] = '\0';
                Addnote(token_text, r0, 2);
                Info(NOTETEXT);
                Info(NOTE3);
                return Get_token();
            }
            if (c == '\n')     col = 1, row++;
            Add_char(c);
            c0 = c;
            c = fgetc(fin);
        }
        Warning("应输入'*/'！");
        return ERROR_TOKEN;
    }
}
if (c == '.')
//处理小数点开头
{
    Add_char(c);
    c = fgetc(fin);
    col++;
    if (!isdigit(c))
    {
        Warning("应输入数字！");
        return ERROR_TOKEN;
    }
    while (isdigit(c))
    {
        Add_char(c);
        c = fgetc(fin);
        col++;
    }
    if (c == 'F' || c == 'f')
```

```
    {
        Add_char(c);
        Info(FLOAT_CONST);              //返回浮点常量
        return FLOAT_CONST;
    }
    else if (c == 'L' || c == 'l')
    {
        Add_char(c);
        Info(DOUBLE_CONST);         //长双精度浮点常量
        return DOUBLE_CONST;
    }
    else
    {
        ungetc(c, fin);
        col--;
        Info(FLOAT_CONST);
        return FLOAT_CONST;
    }
}
if (isalpha(c) || c == '_')
//处理字母开头
{
    do
    {
        Add_char(c);                    //拼接 token_text
        c = fgetc(fin);
        col++;
    } while (isalpha(c) || isdigit(c) || c == '_');
    if (c == '[')  //数组类型
    {
        Add_char(c);
        c = fgetc(fin);
        col++;
        while (isdigit(c))
        {
            Add_char(c);
            c = fgetc(fin);
            col++;
        }
        if (c != ']')
```

```
        {
            Warning("缺少右中括号！");
            return ERROR_TOKEN;
        }
        Add_char(c);
        Info(ARRAY);
        return ARRAY;
    }
    ungetc(c, fin);   col--;          //退回多读取的字符
    if ((type = Find_keyword()) == -1)
    {
        Info(IDENT);
        return IDENT;              //非关键字则返回标识符
    }
    Info(type);
    return type;                  //返回关键字种类
}
if (isdigit(c))
//处理数字开头
{
    do
    {
        Add_char(c);              //拼接数字串
        c = fgetc(fin);
        col++;
    } while (isdigit(c));
    if (c == '.')
    {
        do
        {
            Add_char(c);
            c = fgetc(fin);
            col++;
        } while (isdigit(c));
        if (c == 'F' || c == 'f')
        {
            Add_char(c);
            Info(FLOAT_CONST);          //返回浮点常量
            return FLOAT_CONST;
```

```
        }
        else if (c == 'L' || c == 'l')
        {
            Add_char(c);
            Info(DOUBLE_CONST);
            return DOUBLE_CONST;
        }
        else
        {
            ungetc(c, fin);
            col--;
            Info(FLOAT_CONST);
            return FLOAT_CONST;
        }
    }
    if ((c == 'x' || c == 'X') && strcmp(token_text, "0") == 0)
    {   //16 进制数
        do
        {
            Add_char(c);                    //拼接数字串
            c = fgetc(fin);
            col++;
        } while (isdigit(c) || (c >= 'a' && c <= 'f') || (c >= 'A' && c <= 'F'));
    }
    if (c == 'l' || c == 'L')
    {
        Add_char(c);
        Info(LONG_CONST);        //长整型常量
        return LONG_CONST;
    }
    ungetc(c, fin);  col--;             //退回多读取的字符
    Info(INT_CONST);
    return INT_CONST;                   //返回整型常量
}
switch (c)
{
case '+': c = fgetc(fin); col++;
    if (c == '+') { Info(PLUSPLUS); return PLUSPLUS; }  //返回++
    ungetc(c, fin); col--; Info(PLUS);  return PLUS;    //返回+
```

```
    case '-': c = fgetc(fin); col++;
      if (c == '-') { Info(SUBSUB); return SUBSUB; }      //返回--
      ungetc(c, fin); col--; Info(SUB); return SUB;       //返回-
    case '&': c = fgetc(fin); col++;
      if (c != '&')  return ERROR_TOKEN;
      Info(AND);     return AND;                          //返回与
    case '|': c = fgetc(fin); col++;
      if (c != '|')  return ERROR_TOKEN;
      Info(OR);      return OR;                           //返回或
    case '=': c = fgetc(fin); col++;
      if (c == '=') { Info(EQ); return EQ; }              //返回==
      ungetc(c, fin); col--; Info(ASSIGN); return ASSIGN; //返回=
    //返回其他符号
    case '*':Info(MUL); return MUL;  case '/':Info(DIV); return DIV;
    case '%':Info(MOD); return MOD;
    case '<':
      if (type0 == INCLUDE) { Info(LAB); return LAB; }
      else { Info(RH);  return RH; }
    case '>':Info(LH);  return LH;   case '(':Info(LP);  return LP;
    case ')':Info(RP);  return RP;   case '[':Info(LSB); return LSB;
    case ']':Info(RSB); return RSB;  case '{':Info(LCB); return LCB;
    case '}':Info(RCB); return RCB;  case ';':Info(SEMI);return SEMI;
    case ',':Info(COMMA); return COMMA;  case '#':Info(WELL); return WELL;
    case '\'':Info(SINGLE_QUOTE); return SINGLE_QUOTE;
    case '\"':Info(DOUBLE_QUOTE); return DOUBLE_QUOTE;
    case '!':Info(NOT); return NOT;
    default:if (c == EOF)  return EOF;
          else    return ERROR_TOKEN;     //返回错误符号
    }

}

ASTree Program()
//语法单位<程序>,生成整个程序语法树的根结点,子树为外部定义序列或文件包含或宏定义或语句
{
    ASTree root = New();    root->tag = _PROGRAMMAR_;
    strcpy(root->data.ele_name, "程序");
    type = Get_token();
```

```
if (type == EOF)   return root;
if (IsType(type))
   root->child = ExtDefList();    //外部定义序列
else if (type == WELL)
{
   type = Get_token();
   if (type == INCLUDE)
      root->child = FileInclude();   //文件包含
   else if (type == DEFINE)
      root->child = MacroDefine();   //宏定义
   else
   {
      Destroy(root);
      Warning("语法错误！应输入\'define\'或\'include\'！");
      return NULL;
   }
   type = Get_token();
}
else
{
   root->child = Statement();    //语句
   type = Get_token();
}
if (error == TRUE)
{
   Destroy(root);
   return NULL;
}
ASTree p = root->child;
while (p)
{//生成其它子树
   if (type == EOF)   break;
   if (IsType(type))
      p->sibling = ExtDefList();
   else if (type == WELL)
   {
      type = Get_token();
      if (type == INCLUDE)
         p->sibling = FileInclude();
```

```
            else if (type == DEFINE)
               p->sibling = MacroDefine();
            else
            {
               Destroy(root);
               Warning("语法错误！应输入\'define\'或\'include\'！");
               return NULL;
            }
            type = Get_token();
         }
         else
         {
            p->sibling = Statement();
            type = Get_token();
         }
         p = p->sibling;
      }
      if (error == TRUE)
      {
         Destroy(root);
         return NULL;
      }
      return root;
}

ASTree FileInclude()
//处理文件包含
{
      type0 = INCLUDE;
      type = Get_token();
      if (type != LAB)
      {
         Warning("应输入左尖括号！");
         return NULL;
      }
      free(token_text); token_text = NULL;
      int c = fgetc(fin); col++;
      while (c != EOF && c != '\n')   //处理空格
      {
         if (c != ' ')    break;
```

```
        c = fgetc(fin);
        col++;
    }
    while (c != EOF && c != '\n')   //拼接文件名
    {
        if (c == '>' || c == ' ') break;
        Add_char(c);
        c = fgetc(fin);
        col++;
    }
    if (c == EOF || c == '\n')
    {
        Warning("缺少右尖括号！ ");
        return NULL;
    }
    if (c == ' ')   //处理空格
    {
        c = fgetc(fin);col++;
        while (c == ' ')
        {
            if (c == '>')   break;
            c = fgetc(fin);
            col++;
        }
        if (c != '>')
        {
            Warning("缺少右尖括号！ ");
            return NULL;
        }
    }
    printf("%-13s%s\n", "文件名", token_text);
    Info(RAB);
    ASTree root = New(); root->tag = _INCLUDE_;
    strcpy(root->data.ele_name, "文件包含");
    root->child = New(); root->child->tag = _FILENAME_;   //子孩子存文件名
    strcpy(root->child->data.ele_name, token_text);
    return root;
}
```

```
ASTree MacroDefine()
//处理宏定义
{
   ASTree root = New(); root->tag = _DEFINE_;
   strcpy(root->data.ele_name, "宏定义");
   ASTree p = root->child = New(); p->tag = _DEFINE_ONE_; //第一子孩子存宏名
   ASTree q = p->sibling = New(); q->tag = _DEFINE_TWO_;  //第二子孩子存宏名
替换词
   free(token_text); token_text = NULL;
   int c = fgetc(fin); col++;
   while (c != EOF && c != '\n')  //处理空格
   {
      if (c != ' ')    break;
      c = fgetc(fin);
      col++;
   }
   if (c == EOF || c == '\n')
   {
      Destroy(root);
      Warning("应输入宏名！");
      return NULL;
   }
   if (!isalpha(c) && c != '_')
   {
      Destroy(root);
      Warning("宏名不合法，应以字母或下划线开头！");
      return NULL;
   }
   while (c != EOF)  //拼接宏名
   {
      if (c == ' ' || c == '\n') break;
      Add_char(c);
      c = fgetc(fin);
      col++;
   }
   if (c == '\n' || c == EOF)
   {
      if (c == '\n')     col = 1, row++;
      printf("%-13s%s\n", "宏名", token_text);
```

```
        strcpy(p->data.ele_name, token_text);
        strcpy(q->data.ele_name, "");
        return root;
    }
    Copy(); free(token_text); token_text = NULL;
    while (c != EOF && c != '\n')   //处理空格
    {
        if (c != ' ')    break;
        c = fgetc(fin);
        col++;
    }
    if (c == '\n' || c == EOF)    //替换词为空
    {
        if (c == '\n')    col = 1, row++;
        printf("%-13s%s\n", "宏名", token_text0);
        strcpy(p->data.ele_name, token_text0);
        strcpy(q->data.ele_name, "");
        return root;
    }
    while (c != EOF)    //拼接宏名替换词
    {
        if (c == ' ' || c == '\n') break;
        Add_char(c);
        c = fgetc(fin);
        col++;
    }
    if (c == '\n')    col = 1, row++;
    printf("%-13s%s\n", "宏名", token_text0);
    printf("%-13s%s\n", "宏名替换词", token_text);
    strcpy(p->data.ele_name, token_text0);
    strcpy(q->data.ele_name, token_text);
    return root;
}

ASTree ExtDefList()
//语法单位<外部定义序列>,处理一系列外部定义,返回所有子树均为外部定义结点
{
    if (type == EOF)    return NULL;
```

```
    ASTree root = New();    root->tag = _EXTDEFLIST_;
    strcpy(root->data.ele_name, "外部定义序列");
    ASTree p = root->child = ExtDef();          //第一棵子树
    while (p)
    {
        p->sibling = ExtDef();
        p = p->sibling;
    }
    if (error == TRUE)
    {
        Destroy(root);
        return NULL;
    }
    return root;
}

ASTree ExtDef()
//语法单位<外部定义>,处理一个外部定义
{
    if (type == EOF)     return NULL;
    if (!(IsType(type))) return NULL;
    type0 = type;
    type = Get_token();
    if (type != IDENT && type != ARRAY)
    {
        Warning("应输入标识符！");
        return NULL;
    }
    Copy();
    type = Get_token();
    if (type != LP)    return ExtVarDef();   //外部变量定义
    else       return FuncDef();          //函数定义
}

ASTree ExtVarDef()
//语法成分<外部变量定义>,处理外部变量定义
{
    ASTree root = New(); root->tag = _EXTVARDEF_;
    root->child = New(); root->child->tag = TYPE;
```

```
    strcpy(root->data.ele_name, "外部变量定义");
    root->child->data.var_type = type0;    //第一子孩子存类型
    root->child->sibling = ExtVarList();   //第二子孩子存变量序列
    if (error == TRUE)
    {
        Destroy(root);
        return NULL;
    }
    return root;
}

ASTree ExtVarList()
//语法成分<变量序列>,处理变量序列,每个孩子均为一个外部变量
{
    ASTree root = New(); root->tag = _EXTVARLIST_;
    strcpy(root->data.ele_name, "外部变量序列");
    root->child = ExtVar();
    ASTree p = root->child;
    while (p)
    {
        if (type != COMMA && type != SEMI)
        {
            Destroy(root);
            Warning("应输入逗号或分号！");
            return NULL;
        }
        if (type == SEMI)
        {
            type = Get_token();
            return root;
        }
        type = Get_token();
        if (type != IDENT && type != ARRAY)
        {
            Destroy(root);
            Warning("应输入标识符！");
            return NULL;
        }
        Copy();
```

```
        p->sibling = ExtVar();
        type = Get_token();
        p = p->sibling;
    }
    if (error == TRUE)
    {
        Destroy(root);
        return NULL;
    }
    return root;
}

ASTree ExtVar()
//语法成分<外部变量>,处理一个外部变量
{
    ASTree root;
    root = New(); root->tag = EXTVAR;
    strcpy(root->data.var_name, token_text0);
    return root;
}

ASTree FuncDef()
//语法单位<函数定义>,处理函数定义
{
    ASTree root = New(); root->tag = _FUNCDEF_;
    root->child = New(); root->child->tag = TYPE;   //第一子孩子存类型
    root->child->data.var_type = type0;
    ASTree sec = root->child->sibling = New();
    sec->tag = FUNC;
    strcpy(sec->data.func_name, token_text0);   //第二子孩子存函数名
    Addfunc(token_text0);
    type = Get_token();
    ASTree third = sec->sibling = FormalParList();  //第三子孩子存形参序列
    if (error == TRUE)
    {
        Destroy(root);
        return NULL;
    }
    type = Get_token();
    if (type == LCB)
```

```
    {
        strcpy(root->data.ele_name, "函数定义");
        third->sibling = CompoundStatement();  //第四子孩子存函数体
        if (error == TRUE)
        {
            Destroy(root);
            return NULL;
        }
        strcpy(third->sibling->data.ele_name, "函数体");
    }
    else if (type == SEMI)
        strcpy(root->data.ele_name, "函数声明");
    else if (type != EOF)
    {
        Destroy(root);
        Warning("应输入分号！ ");
        return NULL;
    }
    type = Get_token();
    return root;
}

ASTree FormalParList()
//语法成分<形参序列>,处理形参序列,所有子孩子均为一个形参
{
    ASTree root = New(); root->tag = _FORMALPARLIST_;
    strcpy(root->data.ele_name, "形参序列");
    ASTree p = root->child = FormalPara();
    while (p)
    {
        type = Get_token();
        p->sibling = FormalPara();
        p = p->sibling;
    }
    if (error == TRUE)
    {
        Destroy(root);
        return NULL;
    }
```

```
    return root;
}

ASTree FormalPara()
//语法成分<形参>,处理一个形参
{
    if (type == RP || type == VOID)    return NULL;
    if (type == COMMA)      type = Get_token();
    if (!IsType(type)) { Warning("应输入类型！");return NULL; }
    ASTree p = New(); p->tag = FORPARA;
    strcpy(p->data.ele_name, "形参");
    ASTree fir = p->child = New(); fir->tag = TYPE;   //第一子孩子存类型
    fir->data.var_type = type;
    type = Get_token();
    if (type == IDENT || type == ARRAY)
    {
        fir->sibling = New();
        fir->sibling->tag = VAR;    //第二子孩子存变量名
        strcpy(fir->sibling->data.var_name, token_text);
    }
    else if (type != COMMA && type != RP)
    {
        Destroy(p);
        Warning("应输入逗号或右括号！");
        return NULL;
    }
    return p;
}

ASTree CompoundStatement()
//语法单位<复合语句>,处理复合语句,子孩子为局部变量定义序列或语句序列
{
    ASTree root = New(); root->tag = _COMPOUNDSTATEMENT_;
    strcpy(root->data.ele_name, "复合语句");
    type = Get_token();
    if (IsType(type))
        root->child = LocVarDefList();
    else
        root->child = StatementList();
```

```
    ASTree p = root->child;
    while (p)
    {

        if (IsType(type))
            p->sibling = LocVarDefList();
        else
            p->sibling = StatementList();
        p = p->sibling;
    }
    if (error == TRUE)
    {
        Destroy(root);
        return NULL;
    }
    if (type != RCB)
    {
        Destroy(root);
        Warning("应输入右大括号！ ");
        return NULL;
    }
    return root;
}

ASTree LocVarDefList()
//语法单位<局部变量定义序列>,处理一系列局部变量定义序列,每个孩子均为局部变量定义
{
    ASTree root = New(); root->tag = _LOCVARDEFLIST_;
    strcpy(root->data.ele_name, "局部变量定义序列");
    ASTree p = root->child = LocVarDef();
    while (p)
    {
        if (!IsType(type))
            break;
        p->sibling = LocVarDef();
        p = p->sibling;
    }
    if (error == TRUE)
    {
```

```
        Destroy(root);
        return NULL;
    }
    return root;
}

ASTree LocVarDef()
//语法单位<局部变量定义>,处理一个局部变量定义
{
    ASTree root = New(); root->tag = _LOCVARDEF_;
    strcpy(root->data.ele_name, "局部变量定义");
    root->child = New(); root->child->tag = TYPE;  //第一子孩子存类型
    root->child->data.var_type = type;
    type = Get_token();
    if (type != IDENT && type != ARRAY)
    {
        Destroy(root);
        Warning("应输入标识符！ ");
        return NULL;
    }
    root->child->sibling = LocVarList();   //第二子孩子存变量序列
    if (error == TRUE)
    {
        Destroy(root);
        return NULL;
    }
    return root;
}

ASTree LocVarList()
//语法成分<局部变量序列>,处理局部变量序列,每个子孩子均为一个局部变量
{
    ASTree root = New(); root->tag = _LOCVARLIST_;
    strcpy(root->data.ele_name, "局部变量序列");
    ASTree p = root->child = LocVar();
    while (p)
    {
        type = Get_token();
        if (type != COMMA && type != SEMI)
```

```
        {
            Destroy(root);
            Warning("应输入逗号或分号！");
            return NULL;
        }
        if (type == SEMI)
        {
            type = Get_token();
            return root;
        }
        type = Get_token();
        if (type != IDENT && type != ARRAY)
        {
            Destroy(root);
            Warning("应输入标识符！");
            return NULL;
        }
        p->sibling = LocVar();
        p = p->sibling;
    }
    if (error == TRUE)
    {
        Destroy(root);
        return NULL;
    }
    return root;
}

ASTree LocVar()
//语法成分<局部变量>,处理一个局部变量
{
    ASTree root = New(); root->tag = LOCVAR;
    strcpy(root->data.var_name, token_text);
    return root;
}

ASTree ActualParList()
//处理实参序列,每个子孩子均为一个实参
{
    ASTree root = New(); root->tag = _ACTUALPARLIST_;
```

```
strcpy(root->data.ele_name, "实参序列");
if (type == RP)
{
   type = Get_token();
   if (type != SEMI)
   {
      Destroy(root);
      Warning("应输入分号！");
      return NULL;
   }
   return root;
}
if (type != IDENT && type != INT_CONST && type != FLOAT_CONST)
{
   Destroy(root);
   Warning("应输入标识符或常量！");
   return NULL;
}
ASTree p = root->child = New(); p->tag = ACTPARA;
strcpy(p->data.var_name, token_text);
while (TRUE)
{
   type = Get_token();
   if (type == RP)       break;
   else if (type == COMMA)    continue;
   else if (type == IDENT || type == INT_CONST || type == FLOAT_CONST)
   {
      p->sibling = New();
      p->sibling->tag = ACTPARA;
      strcpy(p->sibling->data.var_name, token_text);
   }
   else
   {
      Destroy(root);
      Warning("语法错误！应输入右括号或逗号或标识符或常量！");
      return NULL;
   }
   p = p->sibling;
}
type = Get_token();
```

```
    if (type != SEMI)
    {
        Destroy(root);
        Warning("应输入分号！ ");
        return NULL;
    }
    if (error == TRUE)
    {
        Destroy(root);
        return NULL;
    }
    return root;
}

ASTree StatementList()
//语法单位<语句序列>,处理一系列语句,每个子孩子均为一个语句
{
    if (type == RCB)   return NULL;
    ASTree root = New(); root->tag = _STATEMENTLIST_;
    strcpy(root->data.ele_name, "语句序列");
    if (type1 == IF)   type1 = -1;
    ASTree p = root->child = Statement();
    while (p)
    {
        if (type1 == IF)    type1 = -1;
        else   type = Get_token();
        if (IsType(type))    return root;
        p->sibling = Statement();
        p = p->sibling;
    }
    if (error == TRUE)
    {
        Destroy(root);
        return NULL;
    }
    return root;
}

ASTree Statement()
//语法单位<语句>,处理一个语句
```

```
{
   ASTree root, p, q, r;
   int t;
   if (IsType(type))    return LocVarDef();  //局部变量定义
   switch (type)
   {
   case IF:   //if 语句
      root = New();
      type = Get_token();
      if (type != LP)
      {
         free(root);
         Warning("应输入左小括号！ ");
         return NULL;
      }
      type = Get_token();
      p = root->child = New();p->tag = CONDITION;  //第一子孩子存条件
      strcpy(p->data.ele_name, "条件");
      p->child = Expression(RP);
      if (error == TRUE)
      {
         Destroy(root);
         return NULL;
      }
      q = p->sibling = New(); q->tag = IFCLAUSE;  //第二子孩子存 if 子句
      strcpy(q->data.ele_name, "if 子句");
      type = Get_token();t = type;
      q->child = Statement();
      if (error == TRUE)
      {
         Destroy(root);
         return NULL;
      }
      if (q->child == NULL && t != SEMI)
      {
         Destroy(root);
         Warning("应输入一个语句！ ");
         return NULL;
      }
```

```
    type = Get_token();
    type1 = IF;
    if (type == ELSE)
    {
        type1 = -1;
        q->sibling = New();
        q->sibling->tag = ELSECLAUSE;     //第三子孩子存 else 子句
        strcpy(q->sibling->data.ele_name, "else 子句");
        type = Get_token(); t = type;
        q->sibling->child = Statement();
        if (error == TRUE)
        {
            Destroy(root);
            return NULL;
        }
        if (q->sibling->child == NULL && t != SEMI)
        {
            Destroy(root);
            Warning("应输入一个语句！ ");
            return NULL;
        }
        root->tag = IFELSE;
        strcpy(root->data.ele_name, "if_else 条件语句");
    }
    else
    {
        root->tag = _IF_;
        strcpy(root->data.ele_name, "if 条件语句");
    }
    break;
case LCB:
    root = CompoundStatement(); break;
case WHILE:    //while 语句
    type0 = WHILE;
    root = New(); root->tag = _WHILE_;
    strcpy(root->data.ele_name, "while 循环");
    type = Get_token();
    if (type != LP)
    {
```

```
        Destroy(root);
        Warning("应输入左小括号！");
        return NULL;
    }
    type = Get_token();
    p = root->child = New(); p->tag = CONDITION;  //第一子孩子存条件
    strcpy(p->data.ele_name, "条件");
    p->child = Expression(RP);
    if (error == TRUE)
    {
        Destroy(root);
        return NULL;
    }
    q = p->sibling = New(); q->tag = LOOPBODY;  //第二子孩子存循环体
    strcpy(q->data.ele_name, "循环体");
    type = Get_token();t = type;
    q->child = Statement();
    if (error == TRUE)
    {
        Destroy(root);
        return NULL;
    }
    if (q->child == NULL && t != SEMI)
    {
        Destroy(root);
        Warning("应输入一个语句！");
        return NULL;
    }
    break;
case FOR:   //for 语句
    type0 = FOR;
    root = New(); root->tag = _FOR_;
    strcpy(root->data.ele_name, "for 循环");
    type = Get_token();
    if (type != LP)
    {
        Destroy(root);
        Warning("应输入左小括号！");
        return NULL;
```

```
}
type = Get_token();
p = root->child = New(); p->tag = _FOR_ONE_;  //第一子孩子存初始语句
strcpy(p->data.ele_name, "初始语句");
p->child = Statement();
if (error == TRUE)
{
   Destroy(root);
   return NULL;
}
if (p->child->tag != _LOCVARDEF_)
   type = Get_token();
q = p->sibling = New(); q->tag = CONDITION;   //第二子孩子存循环条件
strcpy(q->data.ele_name, "条件");
q->child = Expression(SEMI);
if (error == TRUE)
{
   Destroy(root);
   return NULL;
}
r = q->sibling = New(); r->tag = _FOR_TWO_;   //第三子孩子存更新语句
strcpy(r->data.ele_name, "更新语句");
type = Get_token();
r->child = Expression(RP);
if (error == TRUE)
{
   Destroy(root);
   return NULL;
}
r->sibling = New(); r->sibling->tag = LOOPBODY;  //第四子孩子存循环体
strcpy(r->sibling->data.ele_name, "循环体");
type = Get_token(); t = type;
r->sibling->child = Statement();
if (error == TRUE)
{
   Destroy(root);
   return NULL;
}
if (r->sibling->child == NULL && t != SEMI)
```

```
    {
        Destroy(root);
        Warning("应输入一个语句！");
        return NULL;
    }
    break;
case BREAK:   //break 语句
    if (type0 != FOR && type0 != WHILE)
    {
        Warning("break 语句只能出现在 for 循环或 while 循环中！");
        return NULL;
    }
    root = New(); root->tag = _BREAK_;
    strcpy(root->data.ele_name, "break 语句");
    type = Get_token();
    if (type != SEMI)
    {
        Destroy(root);
        Warning("应输入分号！");
        return NULL;
    }
    break;
case CONTINUE:   //continue 语句
    if (type0 != FOR && type0 != WHILE)
    {
        Warning("continue 语句只能出现在 for 循环或 while 循环中！");
        return NULL;
    }
    root = New(); root->tag = _CONTINUE_;
    strcpy(root->data.ele_name, "continue 语句");
    type = Get_token();
    if (type != SEMI)
    {
        Destroy(root);
        Warning("应输入分号！");
        return NULL;
    }
    break;
case RETURN:    //return 语句
```

```
    root = New(); root->tag = _RETURN_;
    strcpy(root->data.ele_name, "return 语句");
    root->child = New(); root->child->tag = _RETURNVAR_;
    strcpy(root->child->data.ele_name, "返回值");  //子孩子存返回值
    type = Get_token();
    root->child->child = Expression(SEMI);
    break;
case LP:
    type = Get_token();
    root = Expression(SEMI); break;
case INT_CONST:case FLOAT_CONST:
    root = New(); root->tag = _EXPRESSION_;
    strcpy(root->data.ele_name, "表达式语句");
    root->child = Expression(SEMI);
    break;
case IDENT:
    root = New();
    if (IsFunc(token_text))
    {  //函数调用语句
       Copy();
       type = Get_token();
       if (type != LP)
       {
          Destroy(root);
          Warning("应输入左括号！");
          return NULL;
       }
       root->tag = _FUNCCALL_;
       strcpy(root->data.ele_name, "函数调用");
       p = root->child = New(); p->tag = FUNC;
       strcpy(p->data.func_name, token_text0);  //第一子孩子存函数名
       type = Get_token();
       p->sibling = ActualParList();   //第二子孩子存实参序列
    }
    else
    {
       root->tag = _EXPRESSION_;
       strcpy(root->data.ele_name, "表达式语句");
       root->child = Expression(SEMI);
```

```
        }
        break;
    case RCB: root = NULL;break;
    case SEMI: root = NULL;break;
    case EOF:root = NULL;break;
    default: { Warning("非法语句！");return NULL;}
    }
    if (error == TRUE)
    {
        Destroy(root);
        return NULL;
    }
    return root;
}

ASTree Expression(Token_kind endsym)
//语法单位<表达式>,处理一个表达式
{
    if (type == endsym)    return NULL;
    Stack ops, ods;
    InitStack(&ops);InitStack(&ods);    //初始化操作符栈和操作数栈
    ASTree q = New();q->tag = OPERATOR;
    q->data.op = WELL; Push(&ops, q);
    ASTree top = Gettop(ops), t, t1, t2;
    while ((type != WELL || top->data.op != WELL) && error == FALSE)
    {
        if (type >= IDENT && type <= DOUBLE_CONST)
        {   //操作数入栈
            q = New(); q->tag = VAR;
            strcpy(q->data.var_name, token_text);
            Push(&ods, q);
            type = Get_token();
        }
        else if (IsOperater(type))
        {
            top = Gettop(ops);
            switch (Precede(top->data.op, type))
            {
            case '<':q = New(); q->tag = OPERATOR;q->data.op = type;
                Push(&ops, q); type = Get_token(); break;  //优先级较小则入栈
```

```
        case '=':if (!Pop(&ops, &t)) error = TRUE;
           type = Get_token();break;      //去括号
        case '>':if (!Pop(&ods, &t2)) error = TRUE;
           if (!Pop(&ods, &t1))  error = TRUE;
           if (!Pop(&ops, &t))   error = TRUE;
           t->child = t1;t1->sibling = t2;
           Push(&ods, t);break;     //优先级较大则出栈两个操作数一个操作符并将运算结果入栈
        default:
           if (endsym == RP && type == RP)  break;
           error = TRUE;break;
        }
        if (type == endsym && ops.top == 1)    type = WELL;
     }
     else if (type == endsym)
     {
        type = WELL;
        if (endsym == SEMI && ops.top != 1)
        {
           if (!Pop(&ods, &t2)) error = TRUE;
           if (!Pop(&ods, &t1))  error = TRUE;
           if (!Pop(&ops, &t))   error = TRUE;
           t->child = t1;t1->sibling = t2;
           Push(&ods, t);
        }
     }
     else error = TRUE;
     top = Gettop(ops);
  }
  top = Gettop(ops);
  if (ods.top == 1 && top->data.op == WELL && error == FALSE)
  {
     free(top);  top = Gettop(ods);
     free(ops.ptr); free(ods.ptr);
     return top;
  }
  else  //运算错误则销毁两个栈并报错
  {
     for (int i = 0;i < ods.top;i++)
        free(ods.ptr[i]);
```

```
        for (int i = 0;i < ops.top;i++)
            free(ops.ptr[i]);
        free(ops.ptr);free(ods.ptr);
        Warning("表达式不合法！");
        return NULL;
    }
    return NULL;
}

void Format(ASTree root, int dep)
//先序遍历语法树
{
    ASTree p, q, r;
    if (root == NULL)  return;
    switch (root->tag)
    {//根据结点类型输出相应内容
    case _PROGRAMMAR_:case _EXTDEFLIST_:case _LOCVARDEFLIST_:
    case _STATEMENTLIST_:
        break;
    case _INCLUDE_:
    {
        fprintf(fout, "#include <%s>\n", root->child->data.ele_name);
        row0++;return;
    }
    case _DEFINE_:
    {
        fprintf(fout, "#define %s %s\n",
            root->child->data.ele_name, root->child->sibling->data.ele_name);
        row0++;return;
    }
    case _EXTVARDEF_:
    {//外部变量定义
        p = root->child;
        fprintf(fout, "%s ", keyword[p->data.var_type]);
        p = p->sibling->child;
        while (p)
        {
            fprintf(fout, "%s", p->data.var_name);
            p = p->sibling;
            if (p == NULL) { fprintf(fout, ";\n"); row0++; }
```

```
        else fprintf(fout, ", ");
    }
    return;
}
case _FUNCDEF_:
{//函数定义
    p = root->child;
    fprintf(fout, "%s ", keyword[p->data.var_type]);
    p = p->sibling;
    fprintf(fout, "%s(", p->data.func_name);
    p = p->sibling; q = p->child;
    if (q == NULL) fprintf(fout, ")");
    while (q)
    {
        fprintf(fout, "%s %s", keyword[q->child->data.var_type]
            , q->child->sibling->data.var_name);
        q = q->sibling;
        if (q == NULL) fprintf(fout, ")");
        else fprintf(fout, ", ");
    }
    p = p->sibling;
    if (p == NULL) { fprintf(fout, ";\n");row0++; }
    else
    {
        fprintf(fout, "\n");
        row0++;
        Format(p, dep);
    }
    return;
}
case _FUNCCALL_:
{//函数调用
    for (int i = 0;i < dep;i++)
        fprintf(fout, "    ");
    p = root->child;
    fprintf(fout, "%s(", p->data.func_name);
    p = p->sibling->child;
    if (p == NULL)
    {
        fprintf(fout, ");\n");
```

```
        row0++;
    }
    while (p)
    {
        fprintf(fout, "%s", p->data.var_name);
        p = p->sibling;
        if (p)  fprintf(fout, ", ");
        else { fprintf(fout, ");\n");row0++; }
    }
    return;
}
case _COMPOUNDSTATEMENT_:
{//复合语句
    for (int i = 0;i < dep;i++)
        fprintf(fout, "    ");
    fprintf(fout, "{\n");row0++;
    p = root->child;
    if (p == NULL)
    {
        for (int i = 0;i < dep;i++)
            fprintf(fout, "    ");
        fprintf(fout, "}\n");row0++;
    }
    while (p)
    {
        Format(p, dep + 1);
        p = p->sibling;
        if (p == NULL)
        {
            for (int i = 0;i < dep;i++)
                fprintf(fout, "    ");
            fprintf(fout, "}\n");row0++;
        }
    }
    return;
}
case _LOCVARDEF_:
{//局部变量定义
    for (int i = 0;i < dep;i++)
        fprintf(fout, "    ");
```

```
    p = root->child;
    fprintf(fout, "%s ", keyword[p->data.var_type]);
    p = p->sibling->child;
    while (p)
    {
        fprintf(fout, "%s", p->data.var_name);
        p = p->sibling;
        if (p == NULL) { fprintf(fout, ";\n");row0++; }
        else fprintf(fout, ", ");
    }
    return;
}
case _IF_:
{//if 语句
    for (int i = 0;i < dep;i++)
        fprintf(fout, "    ");
    fprintf(fout, "if (");
    p = root->child;
    ftraverse(p->child);
    fprintf(fout, ")\n");row0++;
    p = p->sibling;
    if (p->child->tag == _COMPOUNDSTATEMENT_)
        Format(p->child, dep);
    else
        Format(p->child, dep + 1);
    return;
}
case IFELSE:
{//ifelse 语句
    for (int i = 0;i < dep;i++)
        fprintf(fout, "    ");
    fprintf(fout, "if (");
    p = root->child;
    ftraverse(p->child);
    fprintf(fout, ")\n");row0++;
    p = p->sibling;
    if (p->child->tag == _COMPOUNDSTATEMENT_)
        Format(p->child, dep);
    else
        Format(p->child, dep + 1);
```

```
    p = p->sibling;
    for (int i = 0;i < dep;i++)
        fprintf(fout, "    ");
    fprintf(fout, "else\n");row0++;
    if (p->child->tag == _COMPOUNDSTATEMENT_)
        Format(p->child, dep);
    else
        Format(p->child, dep + 1);
    return;
}
case _WHILE_:
{//while 语句
    for (int i = 0;i < dep;i++)
        fprintf(fout, "    ");
    fprintf(fout, "while (");
    p = root->child;
    ftraverse(p->child);
    fprintf(fout, ")\n");row0++;
    p = p->sibling;
    if (p->child->tag == _COMPOUNDSTATEMENT_)
        Format(p->child, dep);
    else
        Format(p->child, dep + 1);
    return;
}
case _FOR_:
{//for 语句
    for (int i = 0;i < dep;i++)
        fprintf(fout, "    ");
    fprintf(fout, "for (");
    p = root->child;q = p->child;
    if (q->tag == _LOCVARDEF_)
    {
        r = q->child;
        fprintf(fout, "%s ", keyword[r->data.var_type]);
        r = r->sibling->child;
        while (r)
        {
            fprintf(fout, "%s", r->data.var_name);
            r = r->sibling;
```

```
            if (r == NULL) fprintf(fout, ";");
            else fprintf(fout, ", ");
        }
    }
    else
    {
        ftraverse(q->child);
        fprintf(fout, ";");
    }
    p = p->sibling;
    ftraverse(p->child); fprintf(fout, ";");
    p = p->sibling;
    ftraverse(p->child); fprintf(fout, ")\n"); row0++;
    p = p->sibling;
    if (p->child->tag == _COMPOUNDSTATEMENT_)
        Format(p->child, dep);
    else
        Format(p->child, dep + 1);
    return;
}
case _BREAK_:
{//break 语句
    for (int i = 0;i < dep;i++)
        fprintf(fout, "    ");
    fprintf(fout, "break;\n"); row0++;
    return;
}
case _CONTINUE_:
{//continue 语句
    for (int i = 0;i < dep;i++)
        fprintf(fout, "    ");
    fprintf(fout, "continue;\n"); row0++;
    return;
}
case _RETURN_:
{//return 语句
    for (int i = 0;i < dep;i++)
        fprintf(fout, "    ");
    fprintf(fout, "return ");
    ftraverse(root->child->child);
```

```
        fprintf(fout, ";\n"); row0++;
        return;
    }
    case OPERATOR:case VAR:
    {//变量或操作符
        for (int i = 0;i < dep;i++)
            fprintf(fout, "    ");
        ftraverse(root);
        fprintf(fout, ";\n"); row0++;
        return;
    }
    case _EXPRESSION_:
    {//表达式
        for (int i = 0;i < dep;i++)
            fprintf(fout, "    ");
        ftraverse(root->child);
        fprintf(fout, ";\n"); row0++;
        return;
    }
    }
    p = root->child;
    while (p)
    {//遍历输出子孩子
        Format(p, dep);
        p = p->sibling;
    }
}

void Writenote()
//将注释写入格式化文件
{
    char filename[100];
    strncpy(filename, sourcefile, strlen(sourcefile) - 2);
    strcat(filename, "_formated.c");    //生成格式化文件名
    if (notes.num == 0)
    {
        char command[100] = "rename temp.c ";
        strcat(command, filename);
        fclose(fin); fclose(fout);
        system(command);
```

```
        printf("\n 格式化源程序名：%s\n\n", filename);
        return;
    }
    row0--;
    fclose(fin);fclose(fout);
    fin = fopen("temp.c", "r");
    fout = fopen(filename, "w");
    char s[100];
    int r = 1, num = 0, d = row - row0;
    for (int i = 0;i < notes.num; i++)
        notes.notelist[i].row -= d;
    while (fgets(s, 100, fin) != NULL)
    {
        s[strlen(s) - 1] = '\0';
        fputs(s, fout);
        if (notes.notelist[num].row == r)  //在对应行输出注释
        {
            if (notes.notelist[num].type == 1)
                fprintf(fout, "\t\t//%s\n", notes.notelist[num].data);
            else
                fprintf(fout, "\t\t/*%s*/\n", notes.notelist[num].data);
            num++;
        }
        else fputc('\n', fout);
        r++;
    }
    fclose(fin);    fclose(fout);
    system("del temp.c");
    printf("\n 格式化源程序名：%s\n\n", filename);
}

#endif
```

A.4 main.c

```
#include "def.h"
#include "assist.h"
#include "func.h"
```

```
int main()
{
   printf("请输入.c 源文件名： ");
   scanf("%s", sourcefile);
   printf("\n");
   fin = fopen(sourcefile, "r");
   if (fin == NULL)
   {
      printf("打开文件失败！\n");
      system("pause");
      return 0;
   }
   Root = Program();
   if (Root == NULL)    //生成失败
      return 0;
   fout = fopen("temp.c", "w");
   Format(Root, 0);                  //生成格式化文件
   ASTree p, q, r;
   if (notes.num > 0)   //生成注释结点
   {
      q = New(); q->tag = NOTE;
      q->data.note = notes.notelist[0];
      r = New(); r->tag = _NOTE_;
      strcpy(r->data.ele_name, "注释");
      r->child = p = q;
   }
   for (int i = 1;i < notes.num;i++)
   {
      p->sibling = New();
      p->sibling->tag = NOTE;
      p->sibling->data.note = notes.notelist[i];
      p = p->sibling;
   }
   if (notes.num > 0)
   {//将注释结点加入语法树
      if (Root->child == NULL) Root->child = r;
      else
      {
         p = Root->child;
```

```
            while (p->sibling)   p = p->sibling;
            p->sibling = r;
        }
    }
    printf("\n\n");
    Traverse(Root, 1);              //遍历语法树
    Writenote();                    //写入注释
    End();                          //结束操作
    system("pause");
    return 0;
}
```

附录 B 测试文件

test.c

```
#include <stdio.h>
#include <stdlib.h>
#define A 10000
#define MAXN 100
int a, b, g[10];        /* test example */
long c, h[10];
float d, i[10];
double f, j[10];
char k;
a = 123;
b = 0x123;
c = 123L;
f = 0.123F;
int func_dec(int a, float b); // function declaration
void func_def()             //function definition
{
   int i, j, k;
   float p, q, r;
}
int main()
{
   func_dec(a, d);
   c = a + b;
   if (c > 1)
      a = a + 1;
   if (a > b && c || d < f)
   {
      a = a > b && c || d < f && (a* b / c - 1);
      b = b + 1;
   }
   d = d - 1;
   if (a > 1)
   {
      int i;
      if (b > 1)
```

```
        {
            if (c > 1)
                a = a + 1;
            else
                b = b + 1;
        }
        else
            c = c + 1;
    }
    while (a)
    {
        int i, j;
        for (i = 1;i < 10;i = i + 1)
        {
            a = a + 1;
        }
        if (a > 1)
            break;
        else
            continue;
    }
    for (a = 1;a;a = a - 1)
    {
        if (a > 1)
            continue;
        for (b = 1;b > 1;b = b + 1)
        {
            a = a + 1;
            b = b - 1;
            c = c + 1;
        }
    }
    return 0;
}
```