

**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： CS2208**

**学 号： U202215423**

**姓 名： 汤婧羽**

**指导教师：**

**报告日期： 2023.9.8**

任务书

* 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](https://baike.baidu.com/item/%E6%BA%90%E4%BB%A3%E7%A0%81)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。

抽象语法树(Abstract Syntax Tree ,AST)作为程序的一种中间表示形式,在[程序分析](https://baike.baidu.com/item/%E7%A8%8B%E5%BA%8F%E5%88%86%E6%9E%90" \t "_blank)等诸多领域有广泛的应用.利用抽象语法树可以方便地实现多种[源程序](https://baike.baidu.com/item/%E6%BA%90%E7%A8%8B%E5%BA%8F" \t "_blank)处理工具,比如源程序浏览器、智能编辑器、[语言翻译器](https://baike.baidu.com/item/%E8%AF%AD%E8%A8%80%E7%BF%BB%E8%AF%91%E5%99%A8/6119005" \t "_blank)等。

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。

* 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（参见实验指导），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。

* **参考文献**

[1] 严蔚敏等. 数据结构(C 语言版). 清华大学出版社

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

# 目录

[任务书 II](#_Toc145171145)

[目录 IV](#_Toc145171146)

[1引言 6](#_Toc145171147)

[1.1课题背景与意义 6](#_Toc145171148)

[1.2国内外研究现状 6](#_Toc145171149)

[1.3课程设计的主要研究工作 7](#_Toc145171150)

[2系统需求分析与总体设计 8](#_Toc145171151)

[2.1系统需求分析 8](#_Toc145171152)

[2.2系统总体设计 8](#_Toc145171153)

[3系统详细设计 10](#_Toc145171154)

[3.1有关常量、全局变量、数据类型以及数据结构的定义 10](#_Toc145171155)

[3.2主要算法设计 11](#_Toc145171156)

[3.2.1 词法分析函数算法设计 11](#_Toc145171157)

[3.2.2 语法分析算法设计 16](#_Toc145171158)

[3.2.3 树遍历算法设计 36](#_Toc145171159)

[3.2.4 缩进编排算法设计 39](#_Toc145171160)

[4系统实现与测试 43](#_Toc145171161)

[4.1系统实现 43](#_Toc145171162)

[4.2系统测试 44](#_Toc145171163)

[4.2.1词法分析测试 44](#_Toc145171164)

[4.2.2语法分析测试 48](#_Toc145171165)

[4.2.3缩进编排测试 58](#_Toc145171166)

[4.2.4退出系统测试 59](#_Toc145171167)

[5总结与展望 60](#_Toc145171168)

[5.1全文总结 60](#_Toc145171169)

[5.2工作展望 60](#_Toc145171170)

[6 体会 62](#_Toc145171171)

[参考文献 63](#_Toc145171172)

[附录 64](#_Toc145171173)

# 1**引言**

## 1.1课题背景与意义

抽象语法树作为程序的一种中间表示形式,在[程序分析](https://baike.baidu.com/item/%E7%A8%8B%E5%BA%8F%E5%88%86%E6%9E%90)等诸多领域有广泛的应用。利用抽象语法树可以方便地实现多种[源程序](https://baike.baidu.com/item/%E6%BA%90%E7%A8%8B%E5%BA%8F)处理工具,比如源程序浏览器、智能编辑器、[语言翻译器](https://baike.baidu.com/item/%E8%AF%AD%E8%A8%80%E7%BF%BB%E8%AF%91%E5%99%A8/6119005)等。

通过学习对抽象语法树的搭建过程与方法，了解并掌握巴克斯（BNF）范式定义高级语言的词法与语法规则、形式语言与自动机原理以及编译技术中的递归下降语法分析法。

## 1.2国内外研究现状

从20世纪60年代至今，高级语言源程序的格式化处理一直是计算机研究发展和开发领域内的一个活跃课题。这个领域的重要性源于其对软件开发生命周期的关键影响，尤其是在大规模和复杂性不断增加的软件项目中。通过规范和自动化源代码的格式，可以提高代码的可读性、可维护性和可理解性，从而降低了错误率，减少了开发时间，并促进了代码重用。因此，高级语言源程序的格式化处理一直被视为软件工程的重要组成部分。

虽然基于高级语言的源程序处理工具的设计已经相对成熟，例如编译器、解释器和代码编辑器，但随着时间的推移，编程语言的演变和新的编程范例的出现，处理工具需要不断适应新的挑战和需求。这种动态性使得格式化处理工具的研究和开发一直是一个不断演进的领域。

近十年来，国外在高级语言格式处理工具的设计方面取得了显著进展。这些进展主要包括以下方面：

复杂算法的应用：国外研究者越来越倾向于采用更复杂的算法，例如语法分析、抽象语法树（AST）遍历和数据流分析，以推断和简化程序中的信息。这些算法可以帮助处理工具更好地理解代码的结构和语义，从而更精确地进行格式化。

智能化格式化：国外的研究也在朝着智能化方向发展，即根据代码的上下文和编码习惯自动调整格式。这种智能化格式化有助于保持代码的一致性，同时允许开发人员在一些情况下灵活地自定义格式。

相比之下，国内目前的研究和应用在高级语言格式处理方面相对较为保守。国内的重点主要集中在特定处理器的特定部分，例如嵌入式系统开发、GPU编程等领域。这些领域的格式化需求可能与通用的编程语言不同，因此需要专门的工具和规范。

总之，高级语言源程序的格式化处理是一个重要且不断发展的领域，国内外都在不断探索和改进相关技术，以满足不断变化的软件开发需求。在国际上，研究者更多地关注复杂算法和智能化格式化，而国内更专注于特定领域的格式化需求。这个领域仍然具有广阔的研究和应用前景，将继续推动软件开发的进步。

## 1.3课程设计的主要研究工作

本次设计为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。实现词法分析器的相关技术是采用有穷自动机的原理，用 EBNF 表示各类单词，并对应确定有穷自动机 DFA。实现语法分析器的相关技术是采用递归下降子程序法，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，在分析过程的同时生成一棵抽象语法树。最后采用先根遍历，创建风格统一的格式化缩进编排的源程序文件。

# 2系统需求分析与总体设计

## 2.1系统需求分析

本次课程设计要求将源代码的语法结构以树的形式表示，并能够先序遍历以缩进编排的格式输出该抽象语法树。搭建抽象语法树过程包括预处理源码并生成中间文件，以排除头文件、预处理、注释等附加的错误干扰；准确识别语言全部单词及出现的词法错误，并按种类编码进行单词与错误项显示；准确分析语法结构，并以函数调用、递归等方式进行抽象语法树的构建；选择合适的数据结构，方便抽象语法树的遍历输出；进行串行调试，在各节点设置报错功能，优化交互便利性；以风格统一的格式化缩进编排，生成输出文件。

## 2.2系统总体设计

系统包含三个主功能：词法分析、语法分析以及缩进编排并输出文件。

1. 词法分析功能：设计中包括全局变量和头文件引入，定义了各种单词类型的名称，以及一个打印函数。主要函数是lexicalAnalyse，它使用getToken函数从源文件中逐个获取单词，然后根据单词类型调用打印函数输出。getToken函数负责单词的识别，它通过状态机来判断不同类型的单词，包括标识符、关键字、常量、运算符、注释和宏定义
2. 语法分析功能：设w为全局变量，存放当前读入的单词种类编码，token\_text保存单词的自身值。errors 表示错误标记，一旦有错，释放抽象语法树全部结点的空间。 根指针指向的是一个外部定义序列的结点，形成一个二叉树，函数/变量节点的左儿子为它的类型，右儿子为剩下的序列。 syntaxAnalyse函数是整个语法分析的入口点，它调用其他函数来完成语法分析的过程。program函数用于分析整个程序的结构，包括变量定义、函数定义等。其他函数如ExtDefList、ExtDef、ArrayDef、ExtVarDef等则用于处理不同类型的语法结构，如外部定义、数组定义、外部变量定义等。最后前序遍历生成的树并输出
3. 缩进编排功能：通过读取输入文件逐行获取源代码，并在适当的位置插入缩进，以保持代码的可读性。在处理代码块的开头和结尾时，根据大括号的出现和语句关键字（如if、for、while）的位置，动态调整缩进层数。最后，将格式化后的代码写入输出文件。这个设计思路的核心是识别代码块的开始和结束，然后在适当位置插入缩进，以产生格式良好的源代码输出。

系统模块流程图如图2-1所示。

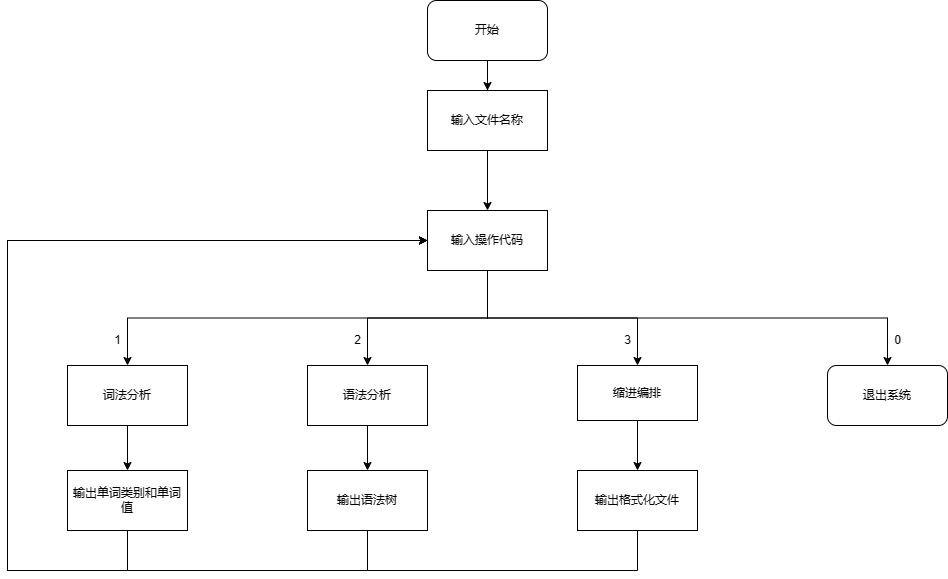


图 2-1 系统模块流程图

# 3系统详细设计

## 3.1有关常量、全局变量、数据类型以及数据结构的定义

为了使得系统便于统一化管理和便捷化浏览，定义常数、全局变量以及数据类型是必不可少的环节，将相关含义与常量、全局变量或数据类型对应起来可方便操作。而采用合适的数据结构将使得抽象语法树的生成和遍历过程更加方便和高效。

1. 常量定义：TokenType 和 DataType 枚举类型是用于在抽象语法树中表示不同类型的节点和标记的关键工具。它们一起构成了编译器或解释器的核心数据结构，用于表示编程语言中的语法元素和语义。TokenType 用于标识词法分析阶段中识别出的不同标记类型，例如标识符、关键字、运算符等。而 DataType 用于表示抽象语法树中不同类型的节点，包括变量、函数、循环、条件语句等。这些标记类型用于识别和区分源代码中的不同部分，有助于编译器和解释器在处理源代码时进行正确的语法分析和语义处理。每个标记类型都与一个唯一的整数值相关联，以便在程序中进行识别和操作。这个枚举类型在编译器和解释器的词法分析阶段发挥关键作用，帮助程序理解源代码的结构和含义。
2. 全局变量定义：token\_text用于保存单词的自身值；cnt\_lines用于检测目前处理到的行数；w用于存放当前读入的单词种类编码；type和token\_text 分别用于存放语法分析提前预估的类型关键字与变量或函数名。
3. 数据类型定义：此处定义了部分函数返回值的类型status为int以及栈数据的类型为树指针类型ASTTree\*。
4. 数据结构定义：对于词法分析处理，生成用于处理识别关键字的数据结构keyword，包含两个元素：对应关键字的字符串以及对应的种类编码。对于抽象语法树的生成和遍历操作，则选择孩子链表表示法构建邻接表数据结构，选择此方法出于两点：1.该数据结构逻辑与此抽象语法树模型贴近，更加直观；2.与数据结构课程对图的邻接表操作相联系，贴近课堂，构建与遍历操作便利。抽象语法树的存储结构如图3-1所示，ASTTree 结构体用于表示抽象语法树的节点，包含左子树指针 l、右子树指针 r 以及一个整数类型字段 type 用于表示节点的类型。此外，它还包含一个嵌套结构 data，其中包含了整数类型字段 type 和字符指针 data，用于存储节点的相关数据。VDN 结构体用于表示变量定义的节点，包括了以下成员：size 整数字段表示参数的个数，variable 字符指针数组，最多可以存储 20 个参数名，next 指向下一个变量定义节点的指针，用于构建链表结构。VDN用于存储变量或函数定义，便于判断变量或函数是否已被定义过，而ASTTree用于存储抽象语法树。

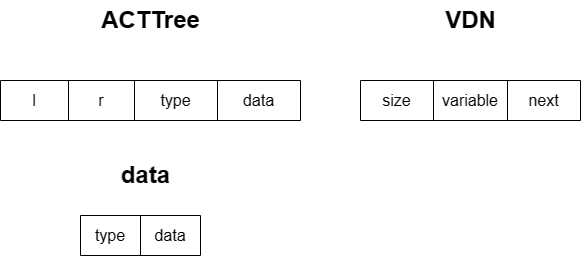


图3-1 抽象语法树的存储结构图

## 3.2主要算法设计

### 3.2.1 词法分析函数算法设计

词法分析有两个函数一个是int lexicalAnalyse()，一个是int getToken(FILE\* fp)。lexicalAnalyse用于从源程序文件中读取字符，把字符传给getToken，根据getToken返回的值打印出单词类型和单词的值（如图3-2-1所示）。

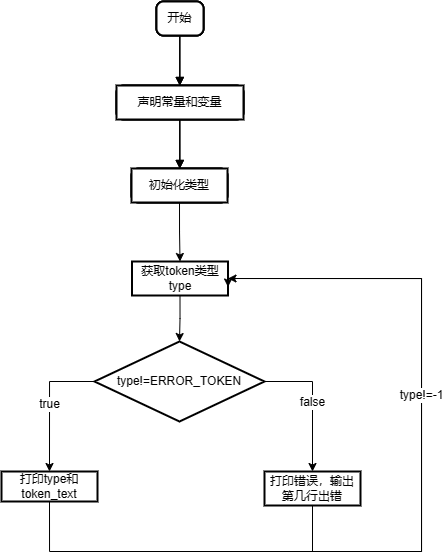


图3-2-1 lexicalAnalyse流程图

词法分析需要识别出规定的C语言单词子集：标识符、关键字、常量、运算符和定界符。为便于返回识别单词类型，需通过枚举类型定义各类单词种类编号：enum TokenType {...};。如IDENT是标识符的种类编码；INT表示关键字int；INT\_CONST表示整型常量等。而全局变量token\_text则用于保存标识符值和常量值，以备后续处理。

getToken函数读取lexicalAnalyse传来的单词，返回单词类型。注意需跳过空白符，当读取回车符时，行数cnt\_lines增一，读取到字母或下划线时，需判断是关键字还是标识符或数组（见图3-2-2）。

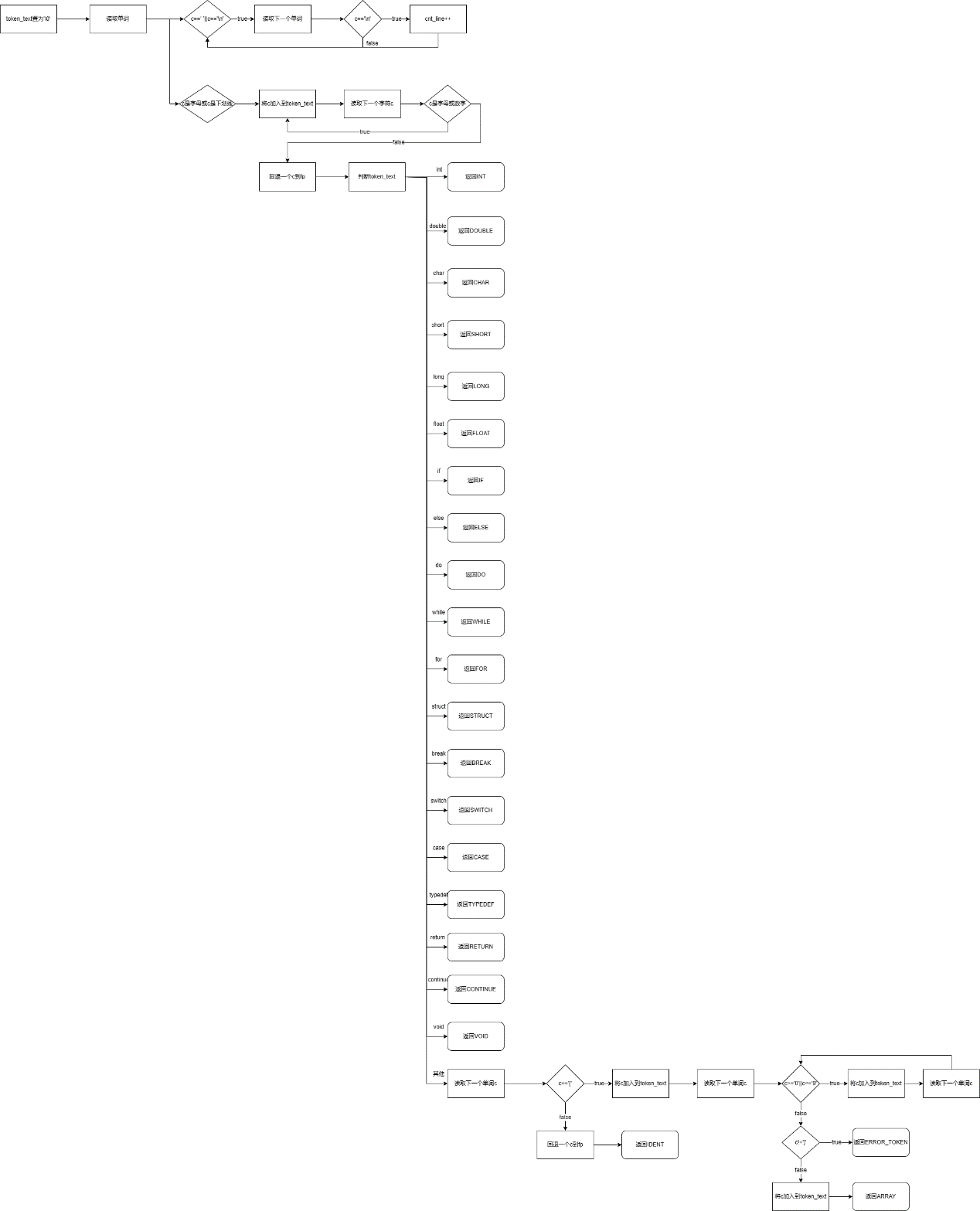


图3-2-2 对空格/换行符/字母/下划线的处理

读取到数字时，需判断是整形常量还是浮点型常量（如图3-2-3）

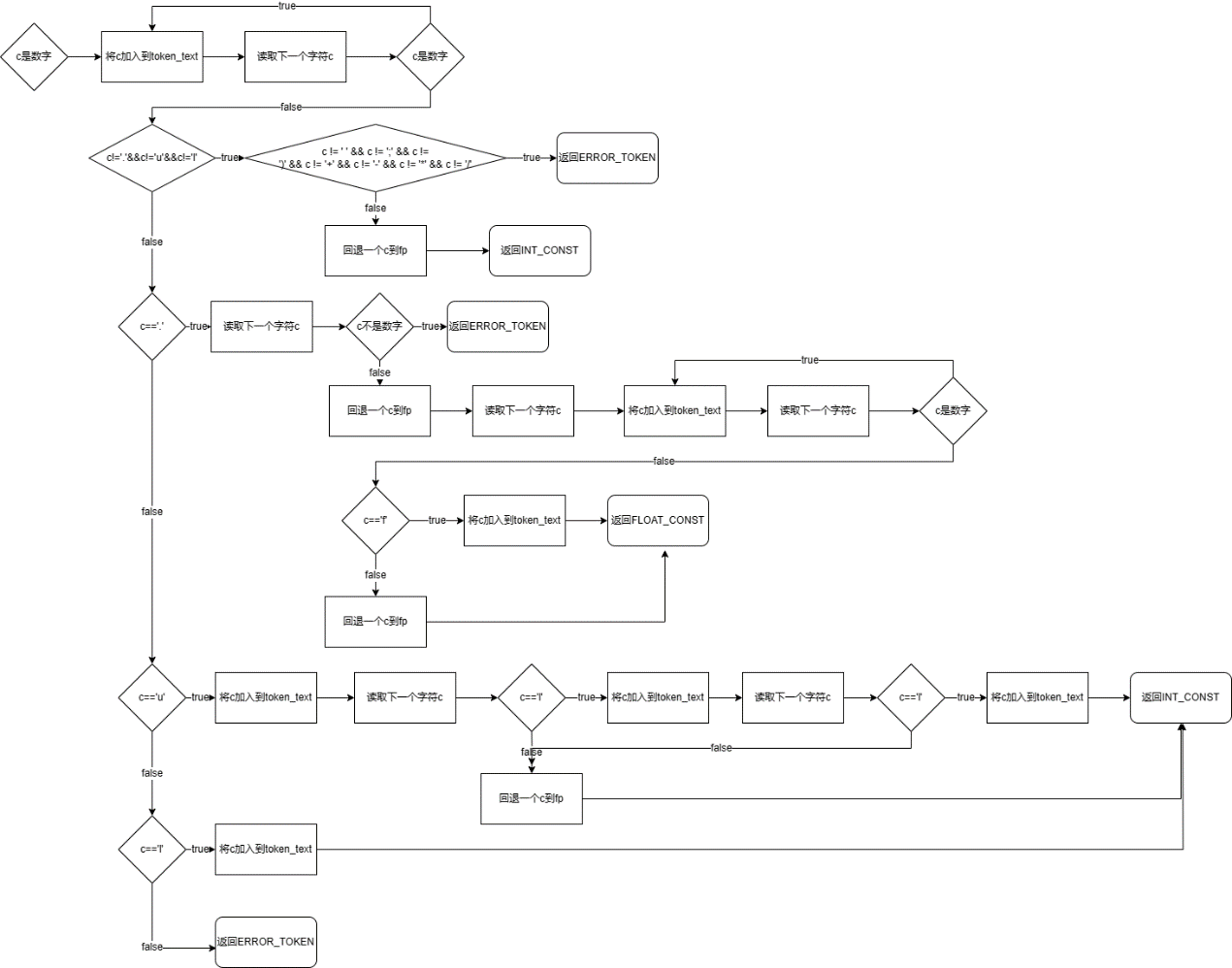


图3-2-3 对数字的处理

以‘.’,‘\’’,‘”’开头的字符的处理见图3-2-4

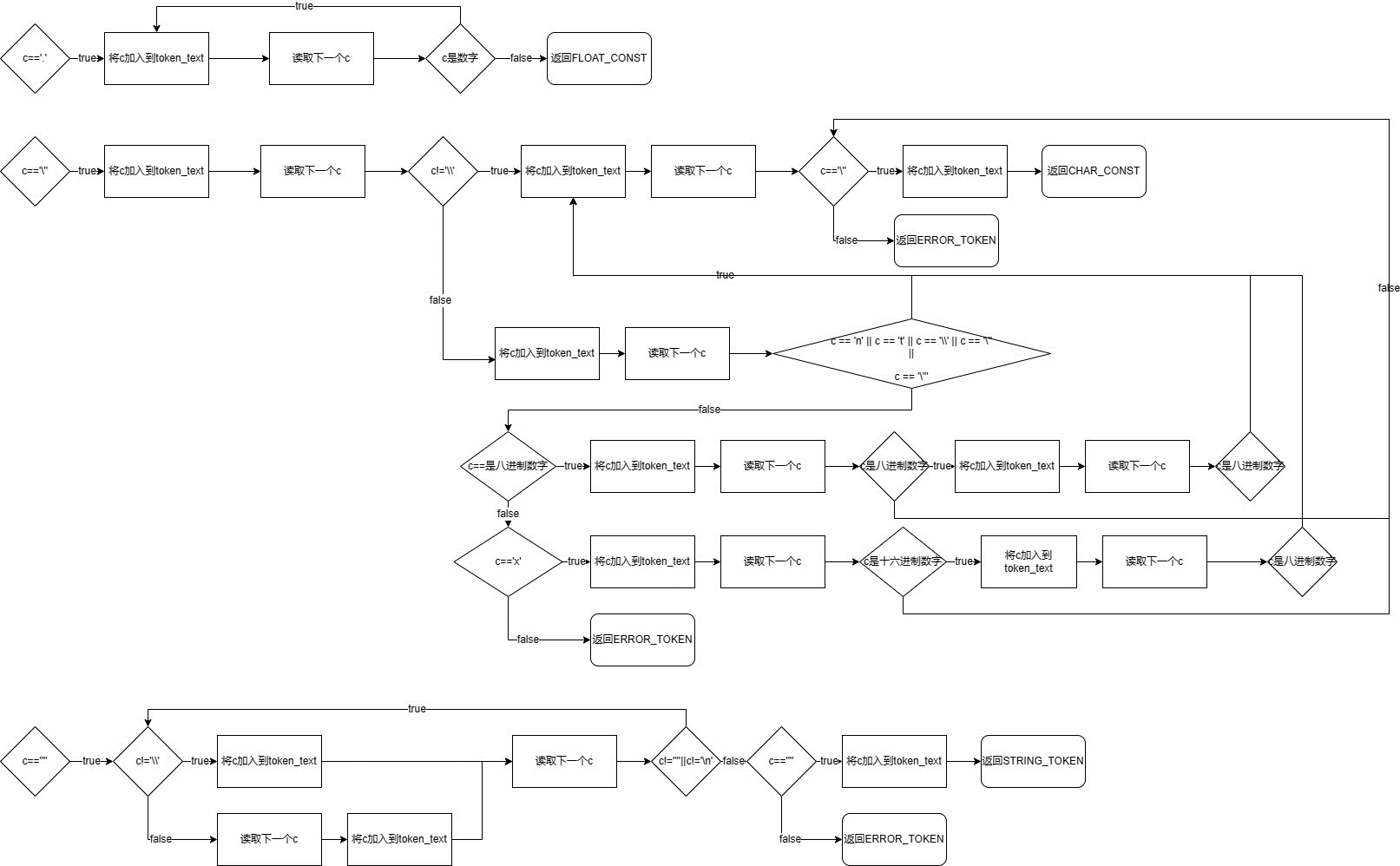


图3-2-4 对以‘.’,‘\’’,‘”’开头的字符的处理

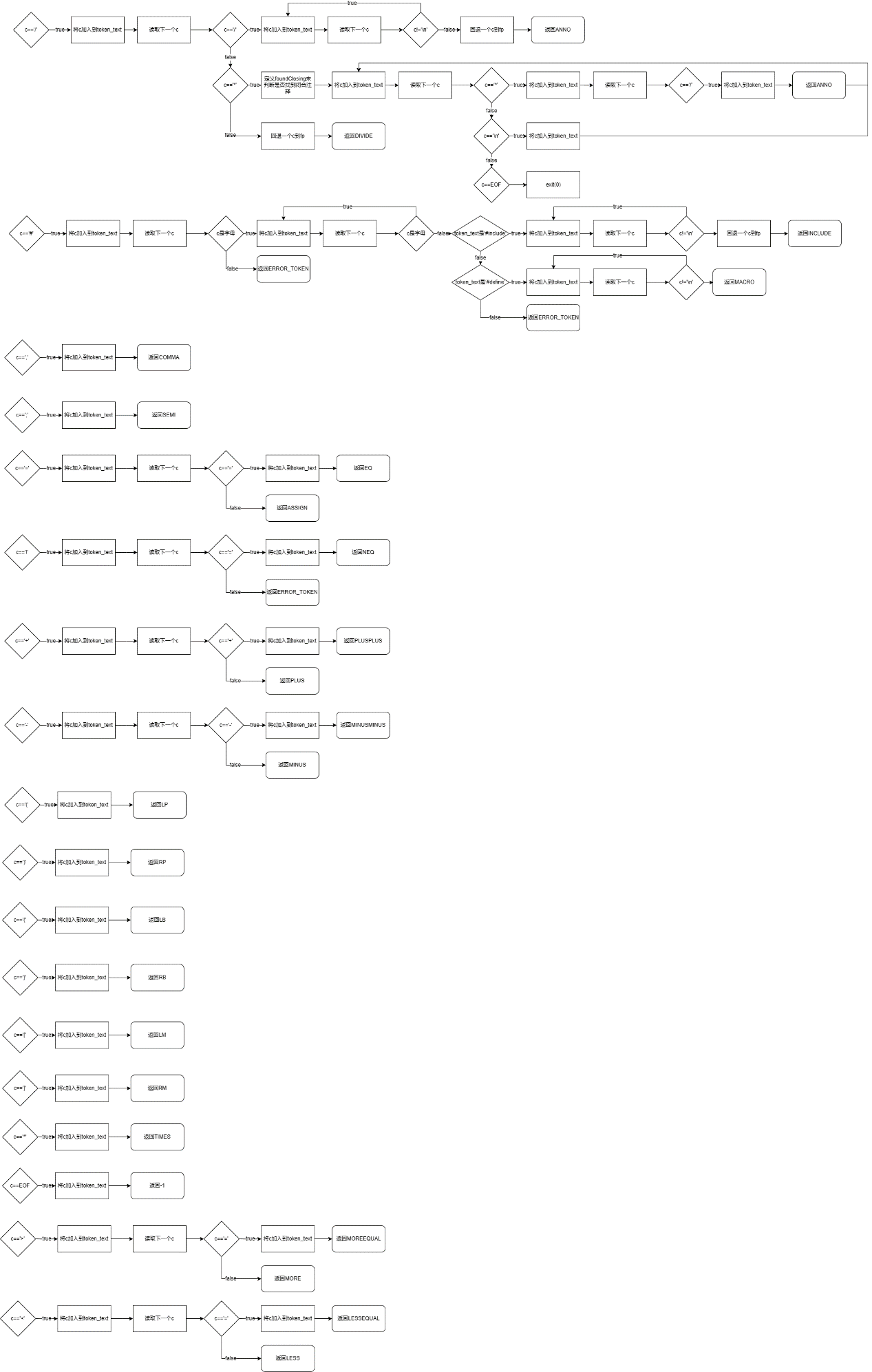
以‘/’，‘#’，‘，’，‘；’，‘=’，‘！’，‘+’，‘-’等其他字符开头的单词处理方式如图3-2-5

图3-2-5 对以‘.’,‘\’’,‘”’开头的字符的处理

值得关注的是对常量的分析，将其划分为四大块，分别为以整型常量、浮点型常量、字符型常量以及字符串常量。整型常量需要关注后缀形式u、l以及ul，以及前缀0x或0X的十六进制数和前缀0的八进制数。而浮点数则需关注后缀f 以及l，除此之外还需关注一些特殊情况，如数字+小数点、小数点+数字、指数型浮点数。解决方法为将情况细分为下划线|字母、数字1~9、数字0以及小数点。将数字细分为0与1~9，可以清晰的处理后续情况。

而对于字符型常量则需要考虑转义字符‘\x’以及普通字符‘x’的情况，同时还需关注一些特殊的错误情况，如‘\’，‘’’，‘x’ 。并且考虑到如若未找寻到第2个单引号，则会一直寻找下去直至文件结尾的问题。故设置换行终止寻找条件，避免错误延伸。

对于字符串型常量，则需判定换行‘\’为一行末尾的情况，如：“xxx\

x”。而相应的会出现一些错误情况，如直接换行情况，以\作为字符串最后一个字符的情况“xxx\”。

除了上述对常量的关注外，对注释处理也十分重要。故将注释细分为行注释

与块注释两部分。其中行注释则以“//”作为开始，不可换行；块注释以“/\*” 作为开始，以“\*/”作结，可换行。

### 3.2.2 语法分析算法设计

语法分析算法主要思想为递归下降子程序法，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语法成分，调用相应子程序进行语法结构分析。特别的，后面需要进行缩进编排处理，故在语法分析环节将会生成一个缩进值结构数组队列，用于存放缩进值及缩进对应行数。详细介绍如下：

算法设计首先需要构建C语言的语法规则，按照BNF范式定义语法规则。syntaxAnalyse为主函数，调用program函数，若返回值为NULL或haveMistake非0，则退出程序并报错，否则，进行先序遍历。如图3-3-1。

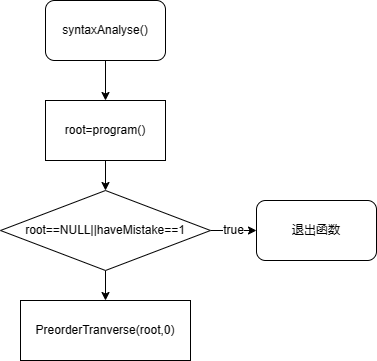


图3-3-1 syntaxAnalyse函数

构建<程序>函数program时，语法规则为**<**程序**>** ：：**=<**外部定义序列**>**；将当前行数和初始化为 0 的缩进值推入对应的处理缩进量的队列中。读取一个单词 w。调用 ExtDefList 函数判定外部定义序列，生成子树 c。如若成功，则将“程序”和 0 分别存入根节点的数据域 root及type中，将子树 c 作为树 T 的第一个子树。若子树 c 未生成成功，则返回NULL。如图3-3-2。

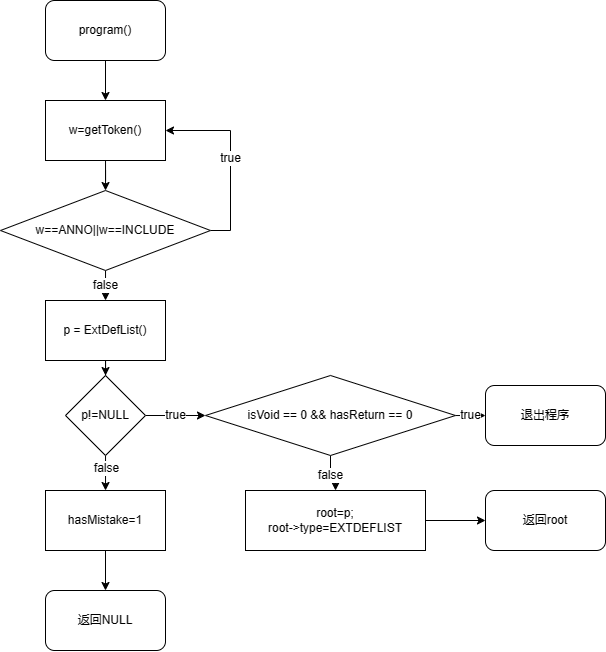


图3-3-2 program函数

构建<外部定义序列>函数时，语法规则为**<**外部定义序列**>**：：**=<**外部定义**>**

**<**外部定义序列**> | <**外部定义**>**；当 w为-1或出现错误时时，返回 NULL。

将EXTDEFLIST和 NULL分别存入函数形参 T 的根数据域中。调用外部定义函数，生成子树 c，若生成成功则将其作为 T 第 1 棵子树。再次递归调用外部定义序列函数，如若调用成功则将其作为 T 的第 2 棵子树。如图3-3-3。

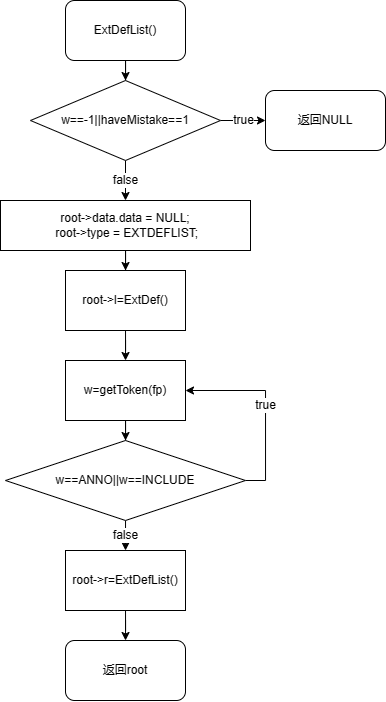


图3-3-3 ExtDefList函数

构建<外部定义>函数时，语法规则为**<**外部定义**>**：：**=<**外部变量定义**>| <**

函数定义**>**；第一个单词必为类型关键字，若是则将其保存至全局变量 type中，否则报错并退出函数。第二个单词必为标识符，若是则保存至 token\_text 中，否则报错并退出函数。再读取一个单词，若为左小括号则进入函数定义函数，否则进入外部变量定义函数。构建树形参 T。如图 3-3-4 所示。

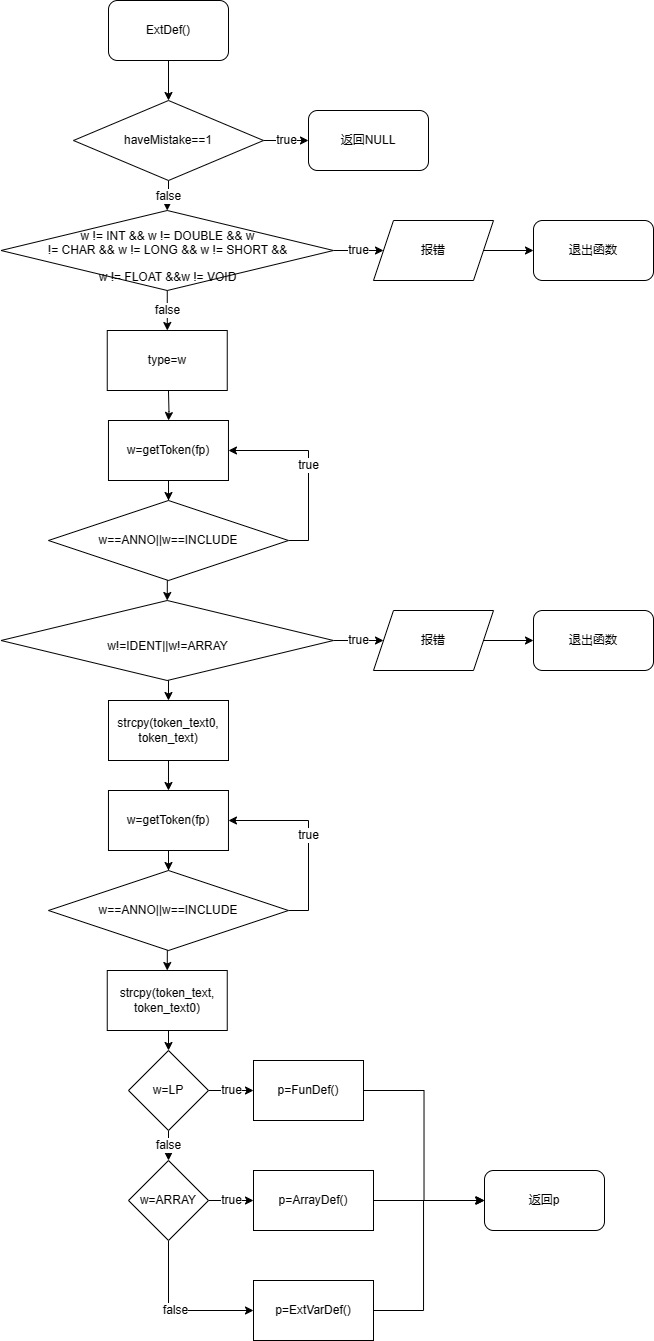


图3-3-4 ExtDef函数

构建<数组定义>函数时，首先，代码检查变量 type 是否为 VOID，如果是，它将输出错误信息并终止程序的执行。这是因为它不允许声明 void 类型的数组。接着，代码分配内存并创建一个名为 root 的 AST 节点，表示数组定义。设置 root 节点的类型为 ARRAYDEF，并初始化其左右子树指针为 NULL，以及数据字段为 NULL。创建一个名为 **p** 的新 AST 节点，用于表示数组的数据类型（**ARRAYTYPE**）。根据变量 **type** 的值，将 **p** 节点的数据字段设置为相应的类型字符串（例如，"int"、"double" 等）。将 **p** 节点作为 **root** 节点的左子树，表示数组类型信息与数组定义相关联。创建另一个名为 **p** 的新 AST 节点，用于表示数组的名称（ARRAYNAME）。为 p 节点分配内存以保存数组名称，并将其数据字段设置为 token\_text 的副本。这里的 token\_text 可能是数组的标识符。

将 **p** 节点作为 root 节点的右子树，表示数组名称与数组定义相关联。创建名为 **q** 的新 AST 节点，用于表示数组的大小（ARRAYSIZE）。将 q 节点的左右子树指针设置为 NULL，并将其数据字段设置为 string\_num。string\_num 可能包含数组的大小信息。将 q 节点作为 p 节点的左子树，表示数组大小信息与数组定义相关联。最后，返回 root 节点，表示整个数组定义的抽象语法树。如图3-3-5。

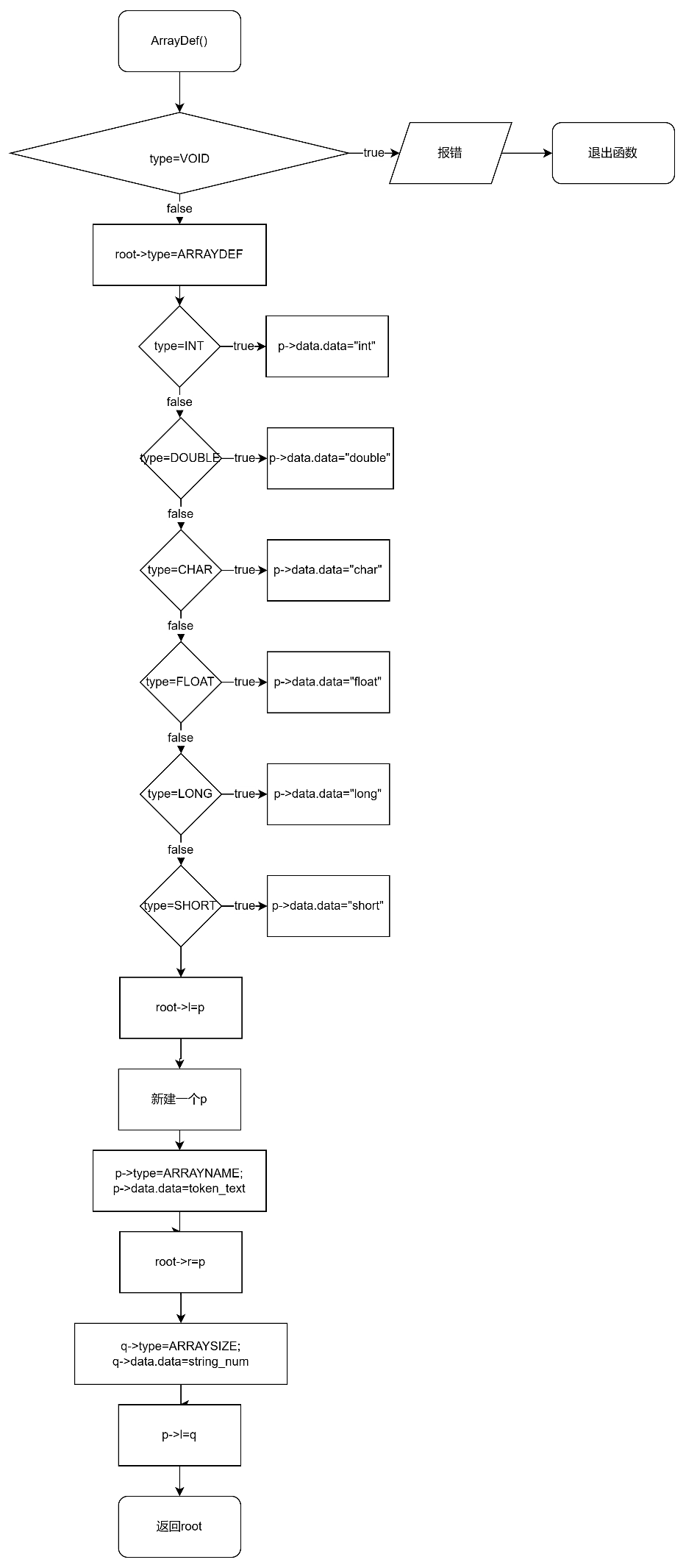


图3-3-5 ArrayDef函数

构建<外部变量定义>函数时，首先，代码检查变量 haveMistake 是否等于1，如果是，它会返回**NULL**，表示已经有错误存在，不再继续处理。接着，代码检查变量 **type** 是否为 **VOID**，如果是，它将输出错误信息并终止程序的执行。这是因为不允许声明 **void** 类型的变量。调用 addName(token\_text) 函数，将变量名 token\_text 添加到某种名称列表中，并检查是否成功。如果 addName 返回1，表示已经有重复的变量名，它将设置 haveMistake 为1，并返回 NULL。然后，代码分配内存并创建一个名为 root 的 AST 节点，表示外部变量定义。设置 root 节点的类型为 EXTVARDEF，并初始化其左右子树指针为 NULL，以及数据字段为 NULL。创建一个名为 p 的新 AST 节点，用于表示外部变量的数据类型（EXTVARTYPE）。根据变量 type 的值，将 p 节点的数据字段设置为相应的类型字符串（例如，"int"、"double" 等）。将 p 节点作为 root 节点的左子树，表示外部变量类型信息与外部变量定义相关联。创建另一个名为 p 的新 AST 节点，用于表示外部变量列表（EXTVARLIST）。将 p 节点的左右子树指针都设置为 NULL，并将其数据字段设置为 NULL。在循环中，代码处理外部变量列表中的每个变量。首先，它创建一个新的 AST 节点 q，表示外部变量列表中的下一个变量。将 q 节点的左右子树指针都设置为 NULL，并将其数据字段设置为 NULL。然后，将 q 节点添加到 p 节点的右子树中，表示外部变量列表的下一个元素。对于每个外部变量，它还会创建一个新的 AST 节点 p，表示变量名（EXTVAR）。分配内存以保存变量名，并将其数据字段设置为 token\_text 的副本。在处理完一个外部变量后，代码会检查 w 是否为 COMMA 或 SEMI，如果不是，它将输出错误信息并终止程序的执行，表示外部变量定义不符合语法规则。如果 w 是 SEMI，则表示外部变量定义结束，代码返回整个外部变量定义的抽象语法树 root。如果 w 是 COMMA，则表示还有更多的外部变量定义，代码会继续处理下一个变量。在处理每个外部变量前，代码会检查 w 是否为 IDENT，如果不是，它将输出错误信息并终止程序的执行，表示外部变量定义不符合语法规则。在处理完每个外部变量后，代码还会再次调用 addName(token\_text)，将变量名添加到名称列表中，并检查是否有重复的变量名。最后，代码会继续读取下一个词法记号，直到不再是注释或包含指令。如图3-3-6所示。

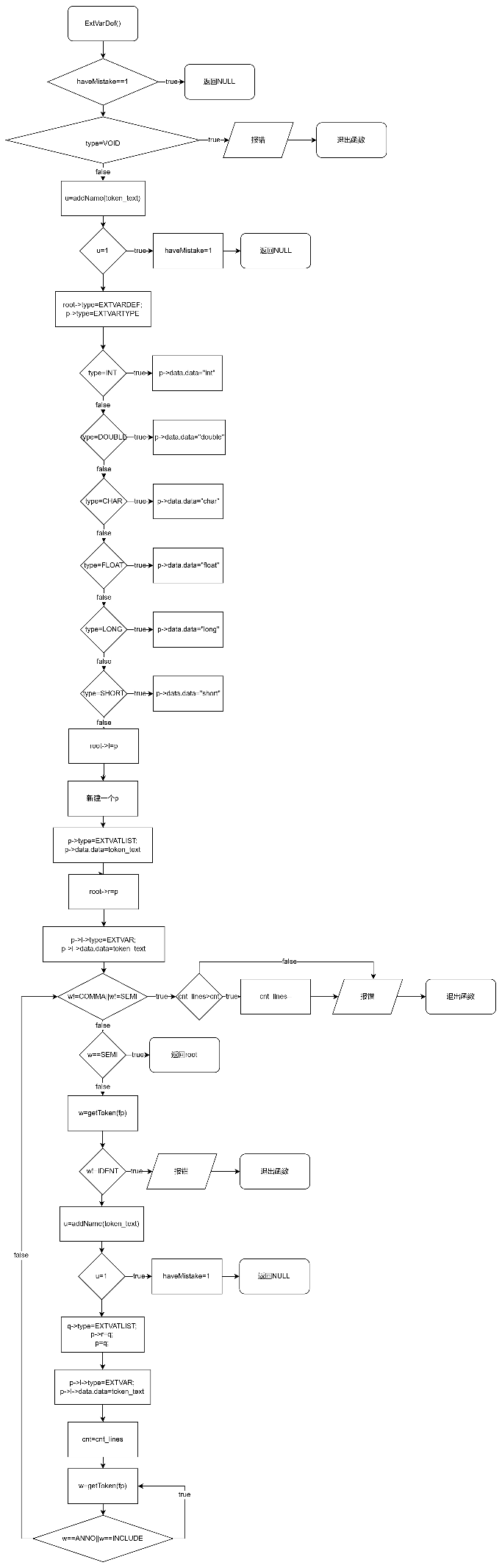


图3-3-6 ExtVarDef函数

构建<函数定义>函数时，首先，代码检查变量 haveMistake 是否等于1，如果是，它会返回NULL，表示已经有错误存在，不再继续处理。接着，代码分配内存并创建一个名为 root 的 AST 节点，表示函数定义。设置 root 节点的类型为 FUNCDEF，并初始化其数据字段为 NULL。创建一个名为 p 的新 AST 节点，用于表示函数的返回类型（FUNCRETURNTYPE）。根据变量 type 的值，将 p 节点的数据字段设置为相应的类型字符串（例如，"int"、"double" 等），并将 isVoid 标志设置为0，表示不是 void 类型的函数。如果 type 为 VOID，则将 p 节点的数据字段设置为 "void"，并将 isVoid 标志设置为1，表示是 void 类型的函数。将 **p** 节点作为 root 节点的左子树，表示函数的返回类型信息与函数定义相关联。创建一个名为 q 的新 AST 节点，用于表示函数名（FUNCNAME）。分配内存以保存函数名，并将其数据字段设置为 token\_text 的副本。在创建 q 节点后，代码将函数名添加到一个名为 VDN 的链表中，用于保存函数名。这个链表可能用于其他用途，但在提供的代码片段中没有显示它的完整定义。将 q 节点作为 root 节点的右子树，表示函数名与函数定义相关联。创建 **q** 节点的左子树，通过调用 FormParaList() 函数，用于处理函数的参数列表。读取下一个词法记号 w，直到不再是注释或包含指令。如果 w 是 **SEMI**，表示这是函数的原型声明，将 root 节点的类型设置为 FUNCCLAIM，并将函数体部分设置为 NULL。如果 w 是 LB，表示这是函数的定义，调用 CompState() 函数来处理函数体，将其设置为 q 节点的右子树，并将函数体类型设置为 FUNCBODY。如果 w 不是 SEMI 也不是 LB，则表示函数定义有错误，输出错误信息并终止程序的执行。最后，返回 root 节点，表示整个函数定义的抽象语法树。如图3-3-7。

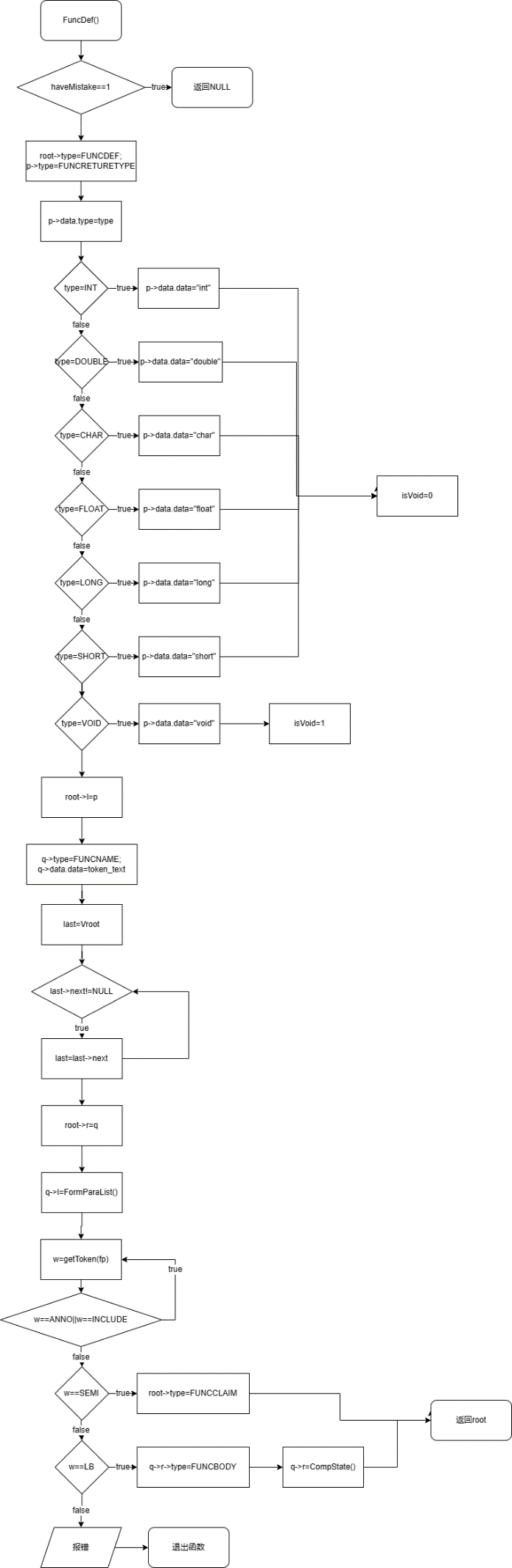


图3-3-7 FuncDef函数

构建<形参列表>函数时，首先，代码检查 haveMistake 是否等于1，如果是的话，就返回一个空指针 NULL，表示存在错误，无法构建参数列表的抽象语法树。接着，代码调用 getToken(fp) 函数来获取下一个标记（token），并将其存储在变量 w 中。然后，通过一个 while 循环来检查 w 是否等于 ANNO 或 INCLUDE，如果是的话，继续调用 getToken(fp) 来获取下一个标记，直到 w 不等于 ANNO 或 INCLUDE 为止。这个部分的目的是跳过一系列注释或包含指令，以便找到实际的参数定义。接下来，代码检查 w 是否等于 RP（右括号），如果是的话，表示参数列表为空，直接返回 NULL。如果 w 不是 RP，那么代码继续检查是否 w 等于 COMMA（逗号），表示参数列表中还有更多的参数。如果是的话，代码再次调用 getToken(fp) 来跳过可能的注释或包含指令。然后，代码分配内存并创建一个新的 ASTTree 节点，将其赋值给 root。这个节点的 data 字段设置为 NULL，表示参数列表节点本身没有数据，而 type 字段设置为 FUNCFORMALPARALIST，表示这是一个函数的形式参数列表节点。接着，代码递归调用 FormParaDef() 函数来构建一个参数定义的子树，并将其赋值给 root->l 字段。最后，代码递归调用 FormParaList() 函数来构建下一个参数列表的子树，并将其赋值给 root->r 字段。最终，函数返回构建好的参数列表的抽象语法树节点 root，其中包含了一个参数定义子树和下一个参数列表子树。如图3-3-8。

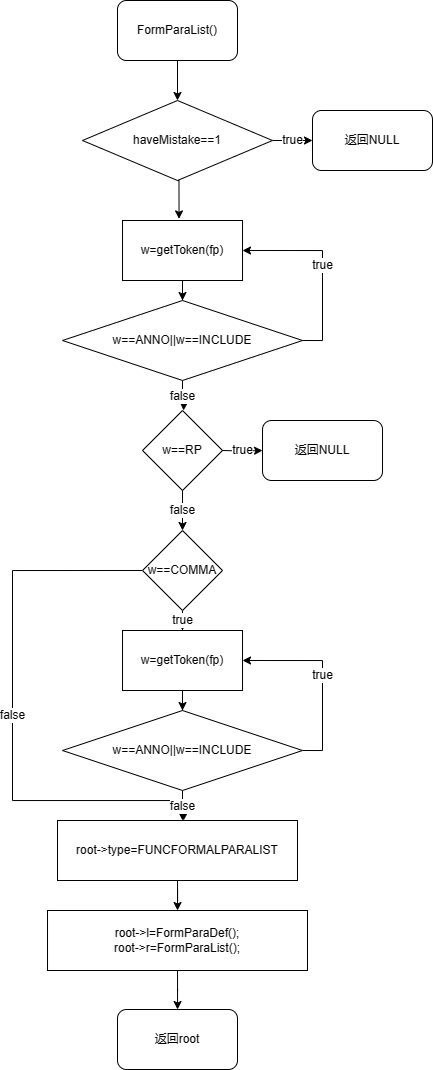


图3-3-8 FormParaList函数

构建<形参定义>函数时，首先，代码检查 haveMistake 是否等于1，如果是的话，就返回一个空指针 NULL，表示存在错误，无法构建参数定义的抽象语法树。接着，代码检查变量 w 是否等于 INT、DOUBLE、CHAR、LONG、SHORT 或 FLOAT 中的任何一个。如果不是其中的任何一个，说明出现了不期望的数据类型，代码会打印错误信息，包括错误发生的行数，然后终止程序运行。如果 w 是一个合法的数据类型，那么将 w 的值赋给变量 type，表示参数的数据类型。接着，调用 getToken(fp) 来获取下一个标记，并通过一个 while 循环来跳过可能的注释或包含指令。然后，代码检查下一个标记 w 是否为 IDENT，即标识符或数组声明。如果不是 IDENT，则打印错误信息，指示在数据类型后面期望一个标识符或数组声明，并终止程序运行。如果 w 是 IDENT，则开始构建参数定义的抽象语法树。首先，分配内存并创建一个新的 ASTTree 节点 root，将其类型设置为 FUNCFORMALPARADEF，表示这是一个函数参数定义节点。接着，分配内存并创建一个新的 ASTTree 节点 p，将其类型设置为 FUNCFORMALPARATYPE，表示这是参数的数据类型节点。将数据类型存储在 p->data.type 字段中，并根据数据类型设置 p->data.data 字段为相应的类型名称。继续设置 p 的左右子节点为 NULL，然后将 p 节点作为 root 节点的左子节点，表示参数定义中的数据类型部分。接着，再次分配内存并创建一个新的 ASTTree 节点 p，将其类型设置为 FUNCFORMALPARA，表示这是参数定义中的标识符部分。代码还尝试将标识符存储在 p->data.data 字段中，然后将 p 节点的左右子节点设置为 NULL。最后，代码调用 addName(token\_text) 来添加标识符到名称表，并检查返回值 u 是否为1，如果是，表示标识符已经存在，将 haveMistake 设置为1，表示存在错误，并返回 NULL。如果标识符添加成功，代码将标识符的内容复制到新分配的内存中，并将其存储在 p->data.data 字段中。然后，将 p 节点作为 root 节点的右子节点，表示参数定义中的标识符部分。最后，返回构建好的参数定义的抽象语法树节点 root，其中包含了数据类型节点和标识符节点。如图3-3-9。

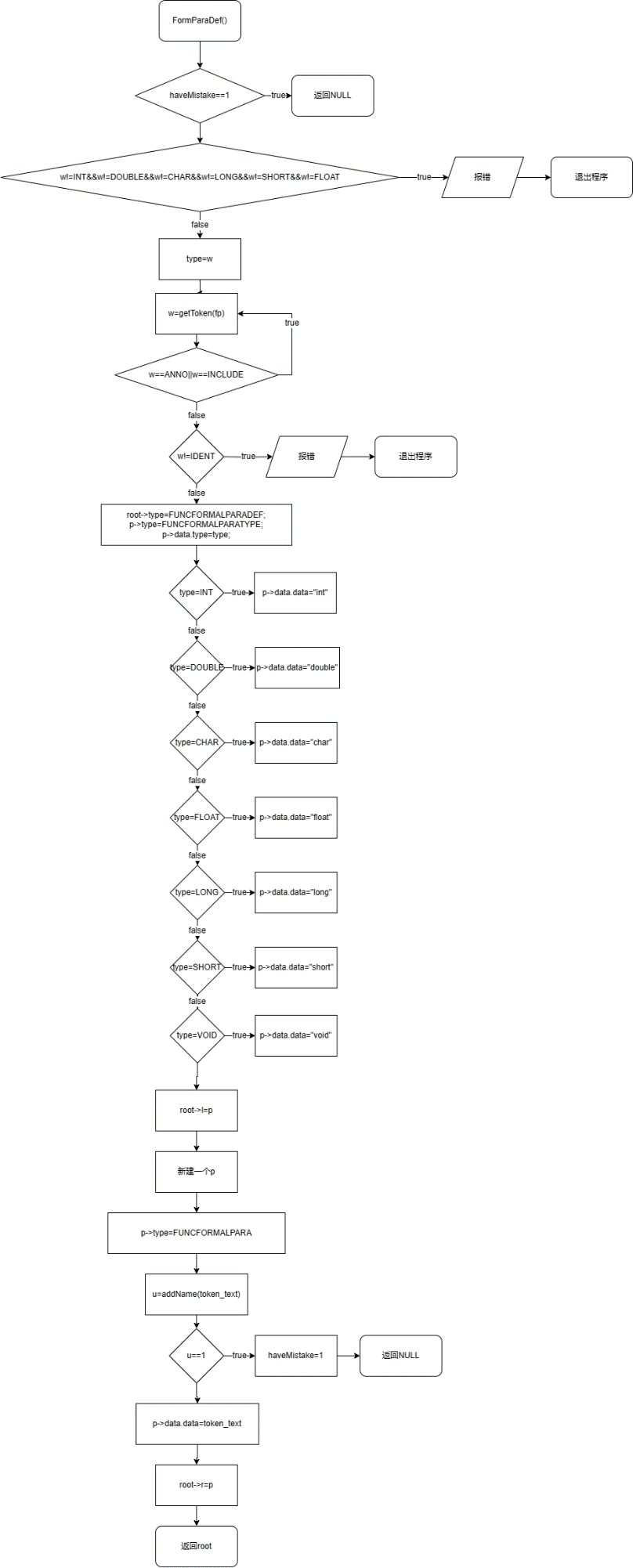


图3-3-9 FormParaDef函数

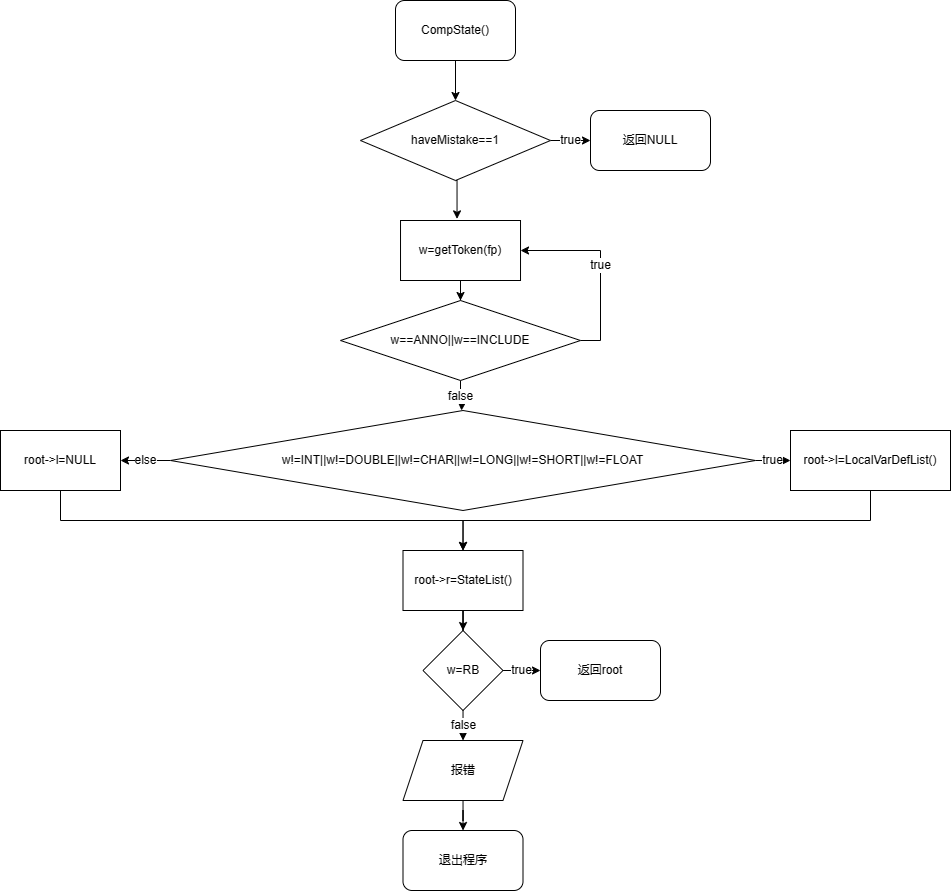
构建<复合语句>函数时，首先，代码检查 haveMistake 是否等于1，如果是的话，就返回一个空指针 NULL，表示存在错误，无法构建复合语句的抽象语法树。接着，代码分配内存并创建一个新的 ASTTree 节点 root，将其类型设置为 COMPOUNDSTATE，表示这是一个复合语句节点。同时，初始化 root 的左子节点和右子节点为 NULL。然后，调用 getToken(fp) 函数来获取下一个标记，并通过一个 while 循环来跳过可能的注释或包含指令。接着，代码检查 w 是否为 INT、DOUBLE、CHAR、LONG、SHORT 或 FLOAT 中的任何一个。如果是的话，表示可能存在局部变量定义，于是调用 LocalVarDefList() 函数来构建局部变量定义列表，并将其作为 root 的左子节点。如果 w 不是上述数据类型之一，说明不是局部变量定义，将 root 的左子节点设置为 NULL。接下来，调用 StateList() 函数来构建语句列表，并将其作为 root 的右子节点。最后，代码检查 w 是否为 RB（右大括号），如果是的话，表示复合语句构建完毕，返回构建好的复合语句的抽象语法树节点 root。如果 w 不是右大括号，表示复合语句没有正确结束，打印错误信息，终止程序运行，将 haveMistake 设置为1，释放 root 节点的内存，并返回 NULL。如图3-3-10。

图3-3-10 CompState函数

构建<表达式>函数时，首先，代码检查 haveMistake 是否等于1，如果是的话，就返回一个空指针 NULL，表示存在错误，无法构建表达式的抽象语法树。接着，代码检查当前的标记 w 是否等于 endsym，如果是的话，说明表达式为空，直接返回 NULL。初始化一个变量 error 为0，用于跟踪错误。创建两个栈 op 和 opn，分别用于操作符和操作数。创建一个根节点 root，并将其类型设置为 EXPRESSION。创建一个操作符节点 p，并将其类型设置为 OPERATOR，数据类型设置为 POUND，然后将其推入操作符栈 op 中。这个节点表示栈底的标志。进入一个循环，条件是当前标记 w 不等于 endsym 或者操作符栈 op 的栈顶元素不是 POUND，并且没有错误发生。在循环中，首先检查操作符栈 op 的栈顶元素，如果是 RP（右括号），则表示可能遇到了一个子表达式的结束。需要检查栈内元素是否足够，如果不足3个则发生错误，否则将两个 RP 出栈，表示子表达式结束。如果当前标记 w 是 IDENT，则检查标识符是否存在，如果不存在则将 haveMistake 设置为1。如果当前标记 w 是合法的操作数（IDENT、INT\_CONST、FLOAT\_CONST、CHAR\_CONST、ARRAY 或 STRING\_CONST），则创建一个操作数节点 p，将数据类型设置为 OPERAND，并将标记内容复制到 p->data.data 中。然后将该节点推入操作数栈 opn 中，并获取下一个标记 w，同时跳过可能的注释或包含指令。如果当前标记 w 是操作符，根据操作符优先级判断如何处理：如果优先级比栈顶操作符低（'<'），则创建一个操作符节点 p，将数据类型设置为当前操作符 w，将标记内容复制到 p->data.data 中，然后推入操作符栈 op 中，并获取下一个标记 w，同时跳过可能的注释或包含指令。如果优先级与栈顶操作符相同（'='），则将栈顶操作符弹出，并获取下一个标记 w，同时跳过可能的注释或包含指令。如果优先级比栈顶操作符高（'>'），则弹出栈顶操作符和操作数，进行操作符和操作数的组合，并将结果操作数推入操作数栈 opn 中，然后创建一个操作符节点 p，将数据类型设置为当前操作符 w，将标记内容复制到 p->data.data 中，然后推入操作符栈 op 中，并获取下一个标记 w，同时跳过可能的注释或包含指令。如果操作符优先级没有定义，发生未知操作符错误，打印错误信息并终止程序。如果循环结束后，操作数栈 opn 中只剩一个元素，操作符栈 op 中只剩一个 POUND 元素，且没有发生错误，则将操作数栈顶元素作为根节点 root 的左子节点，并返回 root。如果不满足上述条件，表示表达式解析出错，打印错误信息并终止程序。如图3-3-11。

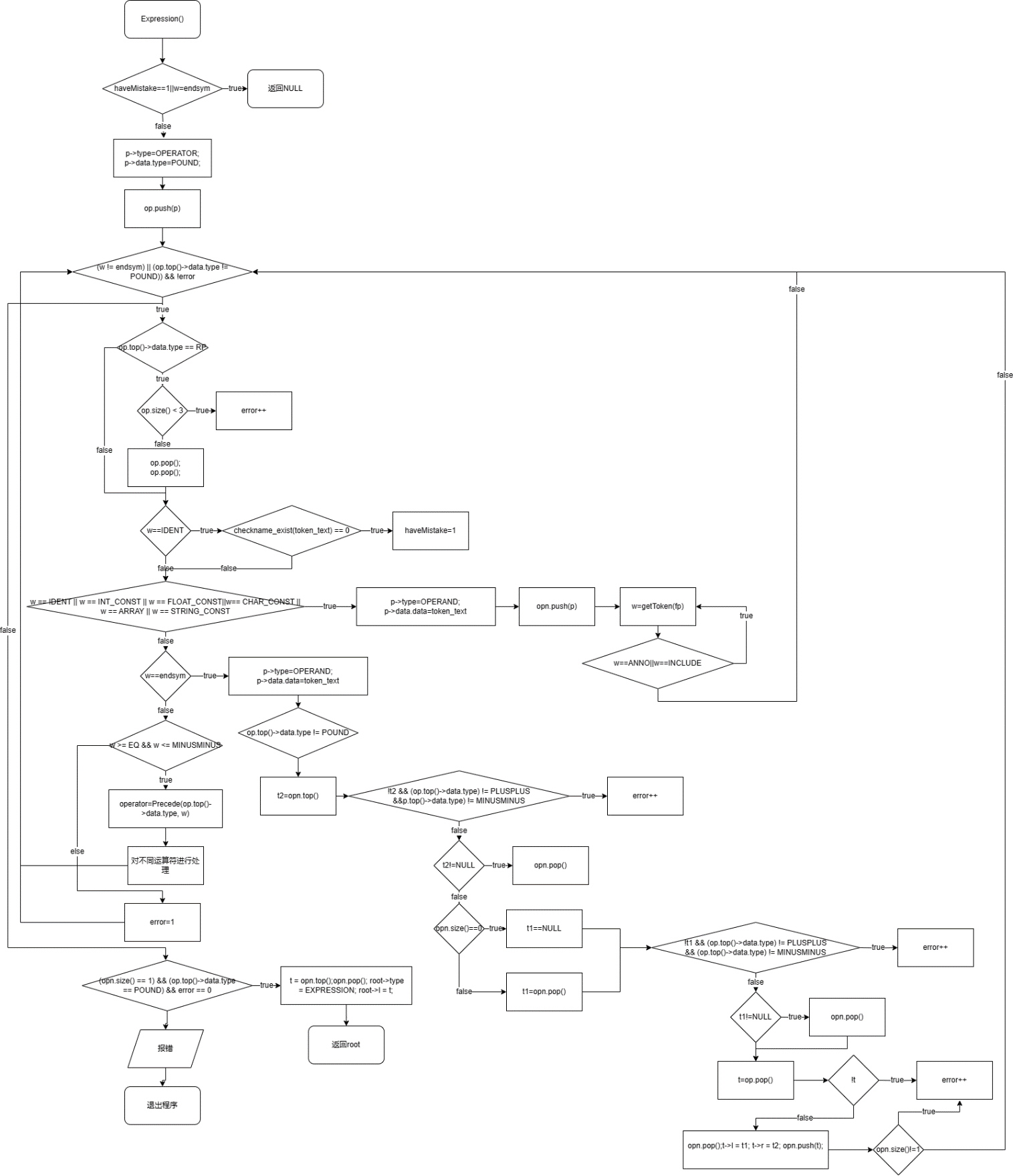


图3-3-11 Expression函数

构建<语句序列>函数时，首先，代码检查 haveMistake 是否等于1，如果是的话，就返回一个空指针 NULL，表示存在错误，无法构建语句列表的抽象语法树。初始化一个根节点 root 为 NULL。调用 Statement() 函数来解析一个语句，并将其结果存储在变量 r1 中。检查 r1 是否为 NULL，如果是的话，表示没有解析到有效的语句，直接返回 NULL。如果 r1 不为 NULL，则创建一个新的节点 root，将其类型设置为 STATELIST，表示这是一个语句列表节点。将 r1 作为左子节点连接到 root，并将 root 的右子节点初始化为 NULL。获取下一个标记 w，并通过一个 while 循环来跳过可能的注释或包含指令。如果 w 不等于右大括号 RB，说明还有更多的语句需要解析，递归调用 StateList() 函数来构建下一个语句列表的抽象语法树，并将其作为 root 的右子节点连接。如果 w 等于右大括号 RB，表示语句列表结束，直接返回 root。如图3-3-12。

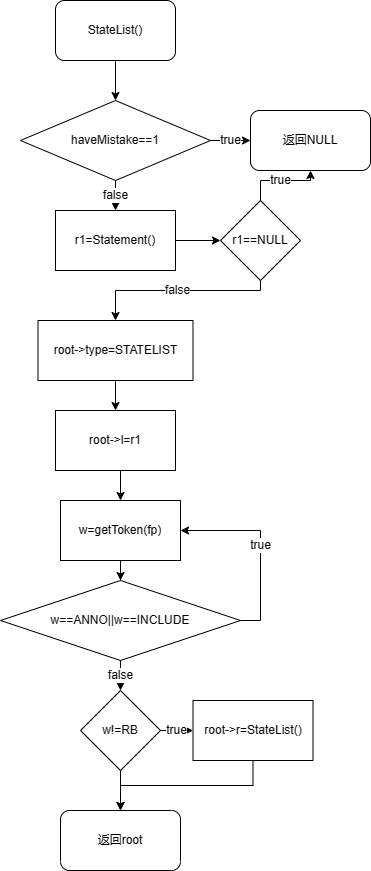


图3-3-12 StateList函数

构建<语句>函数时，首先，代码检查 haveMistake 是否等于1，如果是的话，就返回一个空指针 NULL，表示存在错误，无法构建语句的抽象语法树。初始化一个根节点 root，并设置其左右子节点和数据为空。使用 switch 语句基于当前标记 w 的值来选择不同类型的语句进行解析。如果 w 的值是 IF，则解析条件语句（if语句）。这包括解析条件表达式、if分支的语句块（可能是语句列表或单一语句），以及可选的else分支。解析后将相应的抽象语法树连接到 root 中。如果 w 的值是 WHILE，则解析循环语句（while语句）。这包括解析条件表达式和循环体（可能是语句列表或单一语句），然后将相应的抽象语法树连接到 root 中。如果 w 的值是 FOR，则解析循环语句（for语句）。这包括解析for循环的三个部分：初始化、条件表达式和迭代表达式，以及循环体。然后将相应的抽象语法树连接到 root 中。如果 w 的值是 RETURN，则解析返回语句。这包括解析返回值的表达式，并将其连接到 root 中。如果 w 的值是 DO，则解析do-while循环语句。这包括解析循环体和循环条件，然后将相应的抽象语法树连接到 root 中。如果 w 的值是 BREAK，则表示遇到了break语句。将一个表示break语句的抽象语法树连接到 root 中。如果 w 的值是 CONTINUE，则表示遇到了continue语句。将一个表示continue语句的抽象语法树连接到 root 中。如果 w 的值是某种表达式的起始标记（例如 INT\_CONST、FLOAT\_CONST、CHAR\_CONST、IDENT、ARRAY），则调用 Expression() 函数来解析表达式，并将其连接到 root 中。返回构建好的 root 抽象语法树节点，表示成功解析并构建了一个语句的抽象语法树。

### 3.2.3 树遍历算法设计

通过采用树的先序遍历来对抽象语法树进行显示。在此处，由于在设计最初阶段考虑到遍历算法思想的构成，于是选择逻辑熟悉的邻接表作为树的存储结构。于是联系课堂，思维发散，此处树的先序遍历操作与无向图的深度优先搜索遍历操作几乎完全一致，故减轻了算法架构难度。不同点在于，对语法树显示需要具有一定的缩进处理，以下进行详细说明：

函数PreorderTranverse接受两个参数：ASTTree \*root表示树的根节点指针，int depth表示当前节点的深度，用于控制缩进。首先，函数检查根节点是否为nullptr（即空节点）。如果根节点为空，它输出两个空格，表示缩进，并结束当前函数调用。如果根节点不为空，它执行以下操作：使用for循环生成与当前深度相符数量的空格，用于缩进，以便可视化树的结构。调用showType函数，将当前节点的类型打印出来。这可能是AST节点的一种标识，表示节点的类型。如果当前节点的数据非空（root->data.data不为NULL），则再次使用for循环生成相同数量的空格，以保持与类型相同的缩进，并将节点的数据（root->data.data）打印出来。然后，函数递归地调用自身两次，分别传递左子树（root->l）和右子树（root->r）作为新的根节点，同时递增深度参数（depth + 1）。这样，它将遍历树的左子树和右子树，实现了深度优先的先序遍历。如图3-4-1。

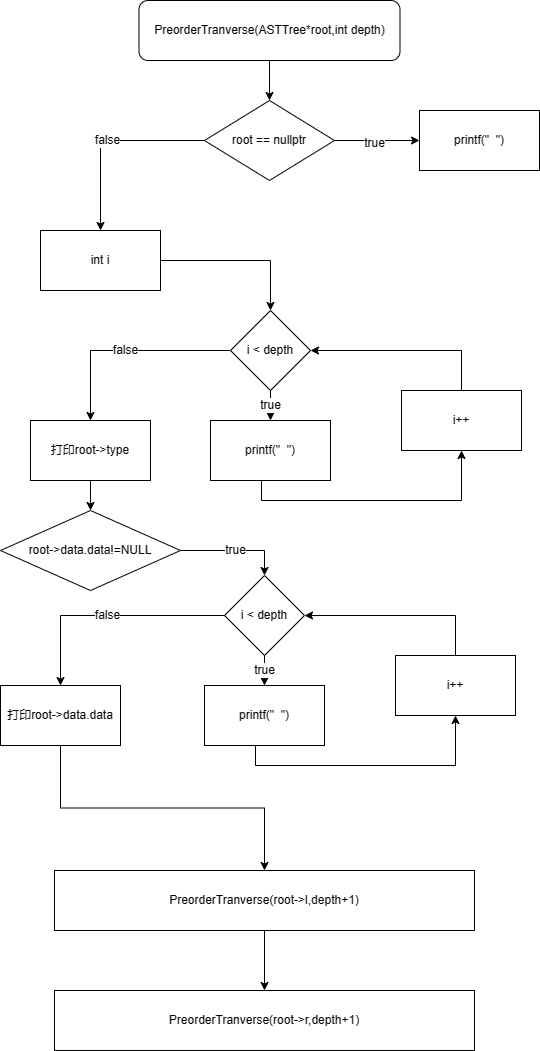


图3-4-1 先序遍历

### 3.2.4 缩进编排算法设计

缩进编排操作则需在语法分析后进行，原因在于在进行语法分析过程的同时，生成了缩进值与对应行数的缩进编排处理队列。此生成过程已在语法分析算法设计中说明。故根据此队列，队列每推出一个元素，则后续行数的缩进值与该元素中的缩进值相等，直到达到队列队首元素指示行数，再推出一个元素，后续如此循环操作。当队列中无元素时，则退出循环，继续读取文件直至文件结尾。该功能包括三个函数，format()函数为主要函数，在此函数中调用readline()函数，计算所需缩进，再将结果传入Output()函数中，使结果写入新的文件中。 format()函数如图3-5-1，readline()函数如图3-5-2，Output()函数如图3-5-3。其中，被readline()调用的getToken()函数在3.2.1已经描述过。

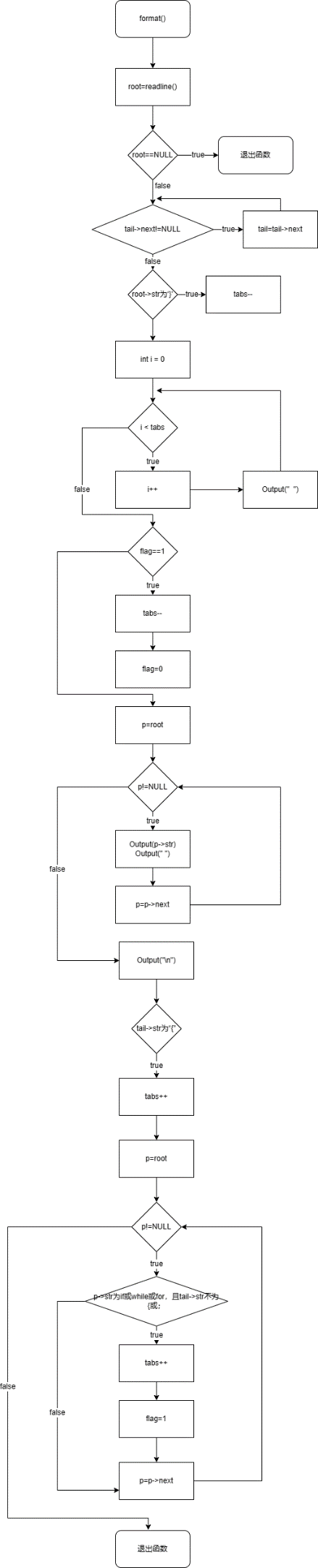


图3-5-1 format()函数

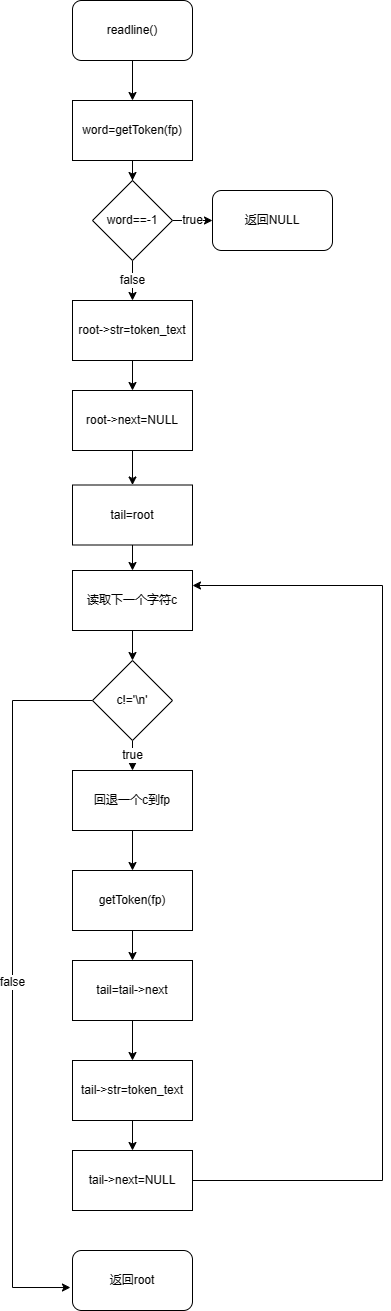


图3-5-2 readline()函数

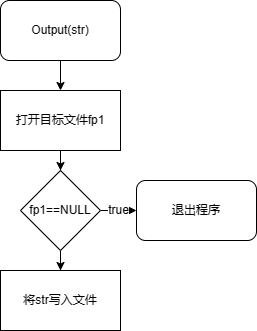


图3-5-3 Output()函数

# 4系统实现与测试

## 4.1系统实现

（1）系统实现环境

软件环境：Visual Studio Code

硬件环境: Intel(R) Core(TM) i5-12500H 2.50 GHz

（2）系统建构组成

在构建完各部件主要函数后，将其构建成系统。整个系统采用操作选择方式进行词法分析、语法分析、 缩进编排和退出程序操作。在进入操作选择前，设置文件选择输入。在输入待操作文件名成功后，即可进入操作选择界面。选择词法分析流程时，将处理行数 cnt\_lines 初始化为 1，表明操作行从第 1 行开始。进行词法分析处理，循环调用 getToken 函数识别各单词， 并进行单词类型与单词值的输出，当读取到错误时，保存至结构数组中，在结尾处按顺序输出。语法分析则调用<程序>函数进行抽象语法树 T 的构建； 若构建失败，则输出错误所在行数；若构建成功，则直接调用树遍历函数先序遍历抽象语法树 T，显示至图像界面后表明输出成功。而缩进编排功能进行抽象语法树生成， 最终使用 C 语言源程序文件， 调用缩进编排Output函数生成格式化文件。 输入0则为退出系统操作。系统界面如图4-1-1所示。具体系统架构如图4-1-2所示。

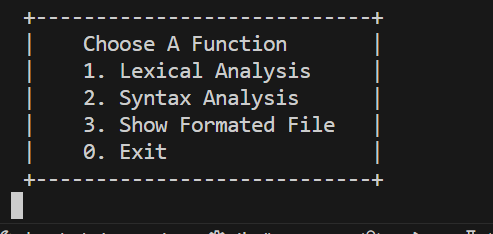


图4 -1-1 系统界面

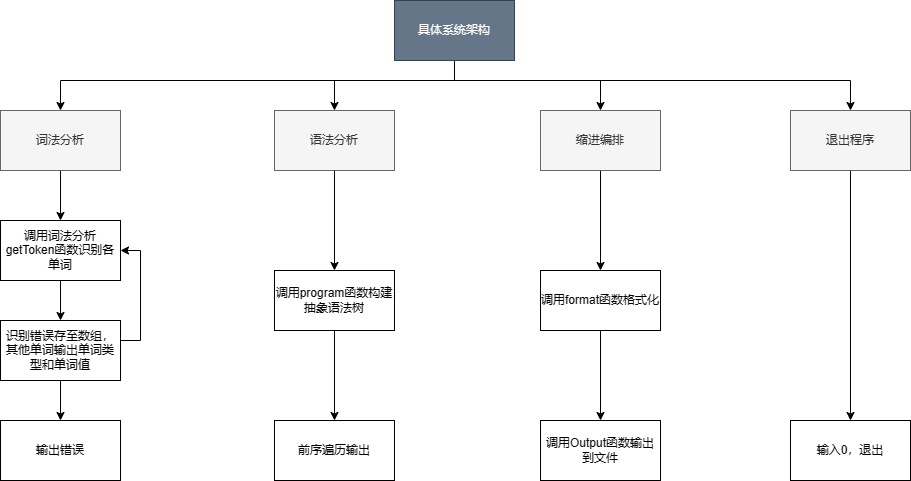


图4 -1-2 系统架构

## 4.2系统测试

对于系统测试环节， 具体包含词法分析功能测试、 语法分析测试、 缩进编排处理测试以及退出系统测试

### 4.2.1词法分析测试

1.正确词法分析测试：

常量、 标识符、 关键字、 定界符、 运算符处理如下。

测试用例包含宏定义、 头文件、 行注释、 块注释， 以及宏定义出现在语句中的情况。 常量测试用例包含 int 类型、 unsigned 类型、 unsigned long类型、 long 类型、 float 类型、 double 类型、 long double 类型、 八进制数、 十六进制数、 字符类型、 字符串常量。 关键字包含各种类型定义。 测试中测试用例如图 4-2-1 所示。 相关测试结果如 4-2-2 和 4-2-3 所示。

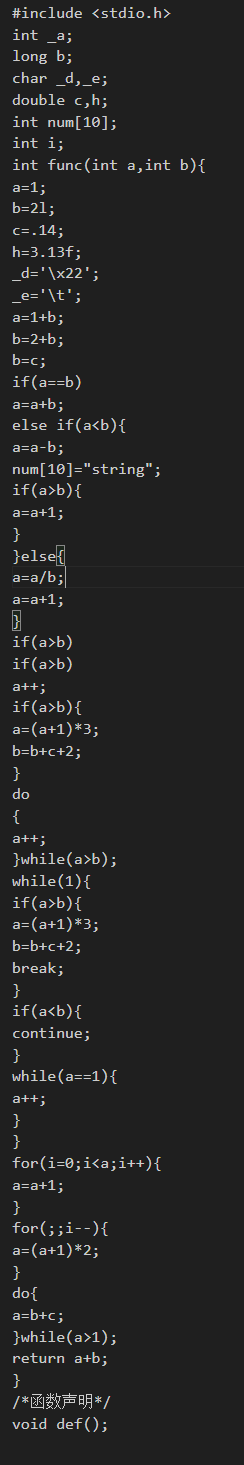


图4-2-1 测试样例

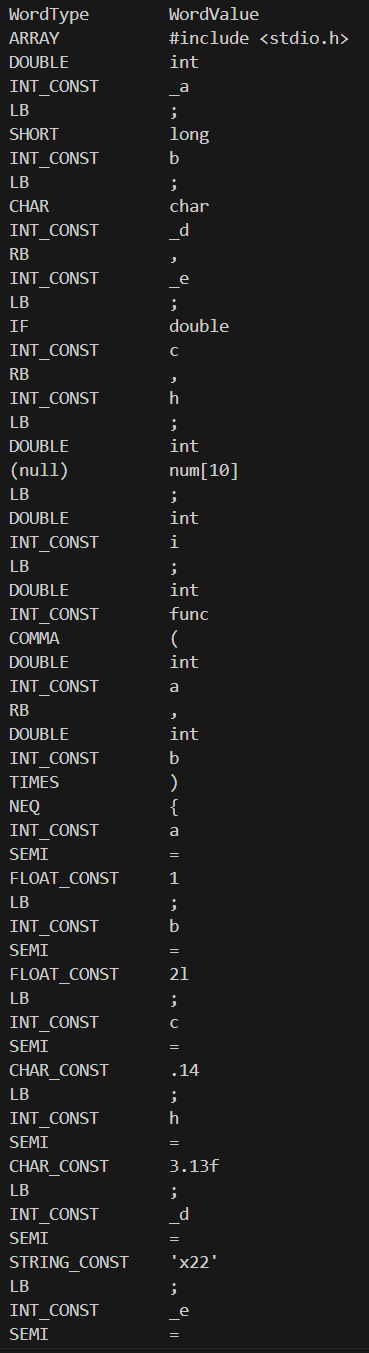
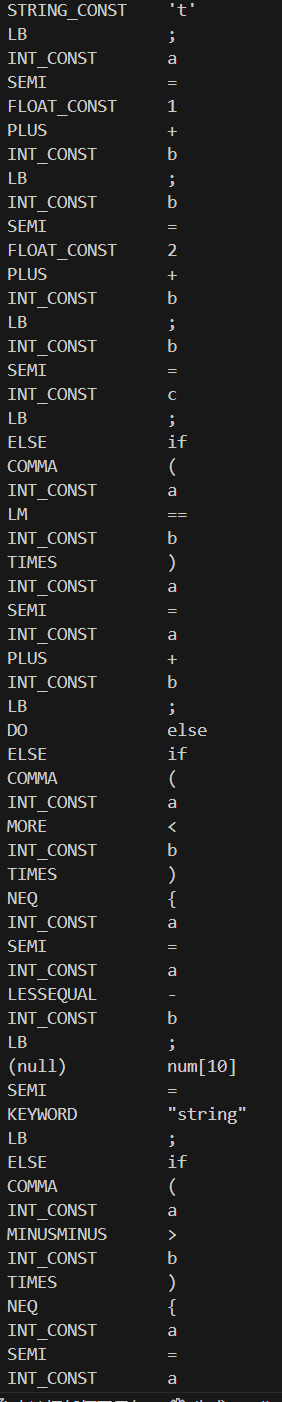
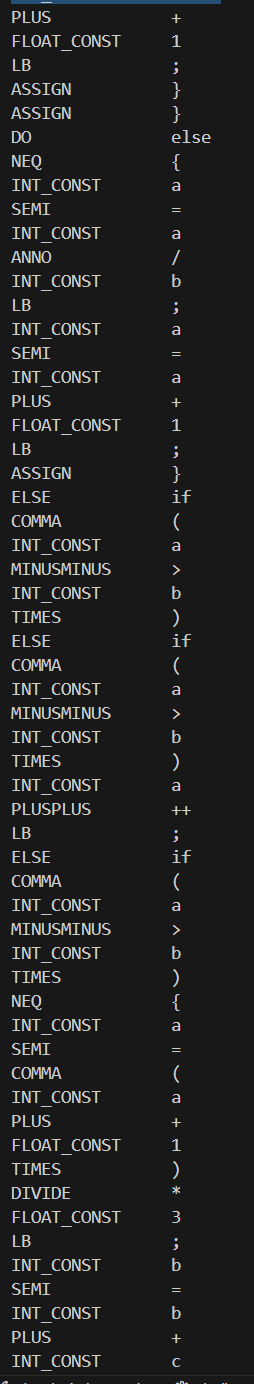
  

图4-2-2 测试结果

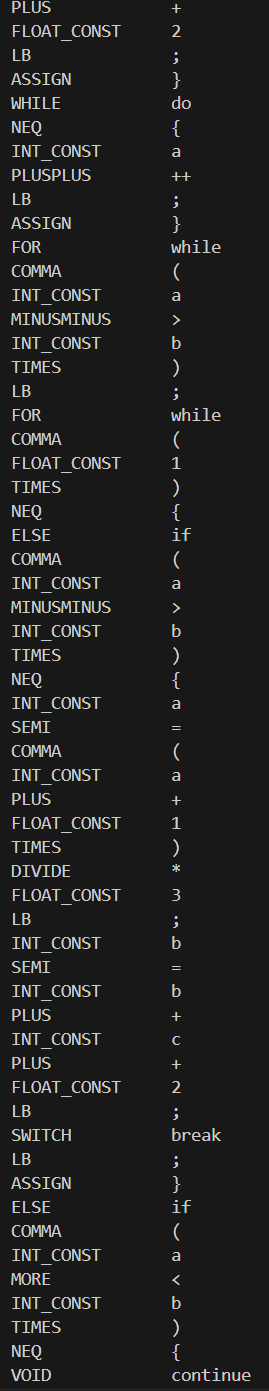
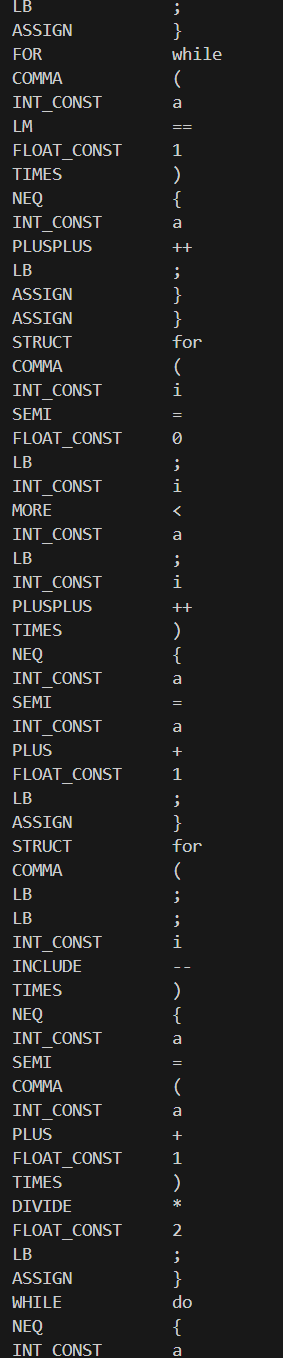
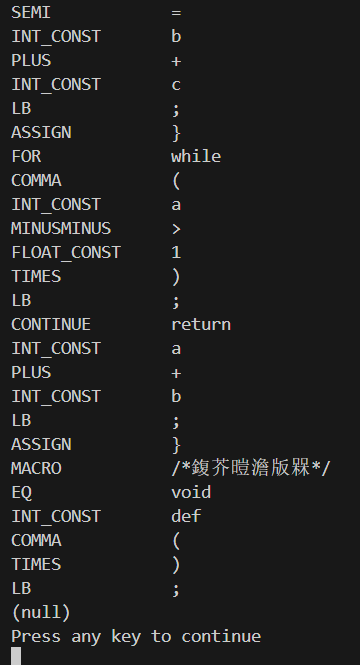
  

图4-2-3 测试结果

2.错误词法分析测试：

插入特殊字符（如@），如图4-2-4。报错，并显示错误行数。

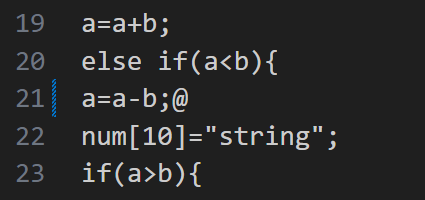
 

图4-2-4 插入特殊字符及报错

### 4.2.2语法分析测试

1.正确语法分析测试：

测试用例中包含外部变量定义、 函数定义、 局部变量定义、 表达式语句、 复合语句、 if 语句、 if-else 语句、 while 语句、 for 语句、 return 语句、 break语句、 continue 语句。如图4-3-1。

测试结果如图4-3-2，图4-3-3，图4-3-4，图4-3-5，图4-3-6，图4-3-7。

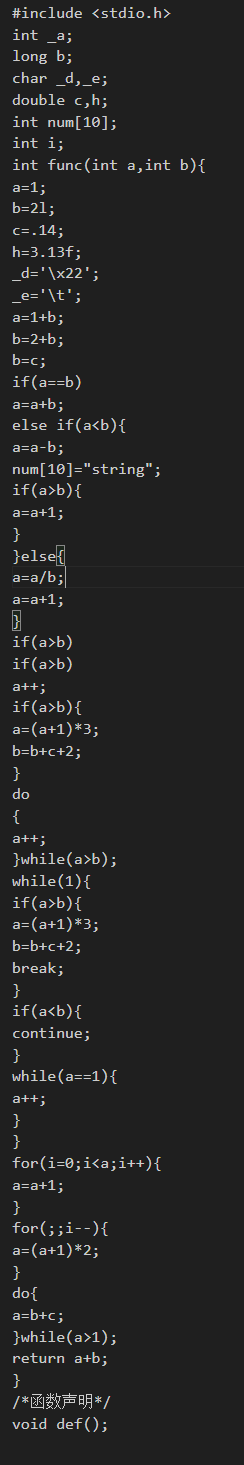


图4-3-1 测试样例



图4-3-2 测试结果

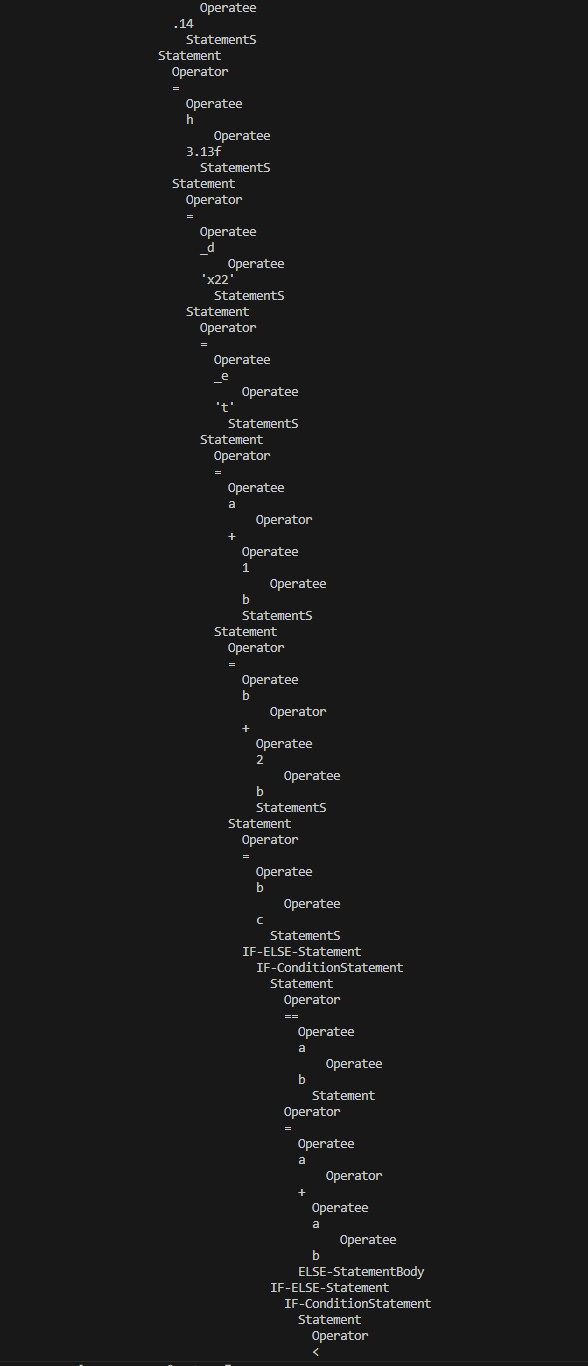


图4-3-3 测试结果

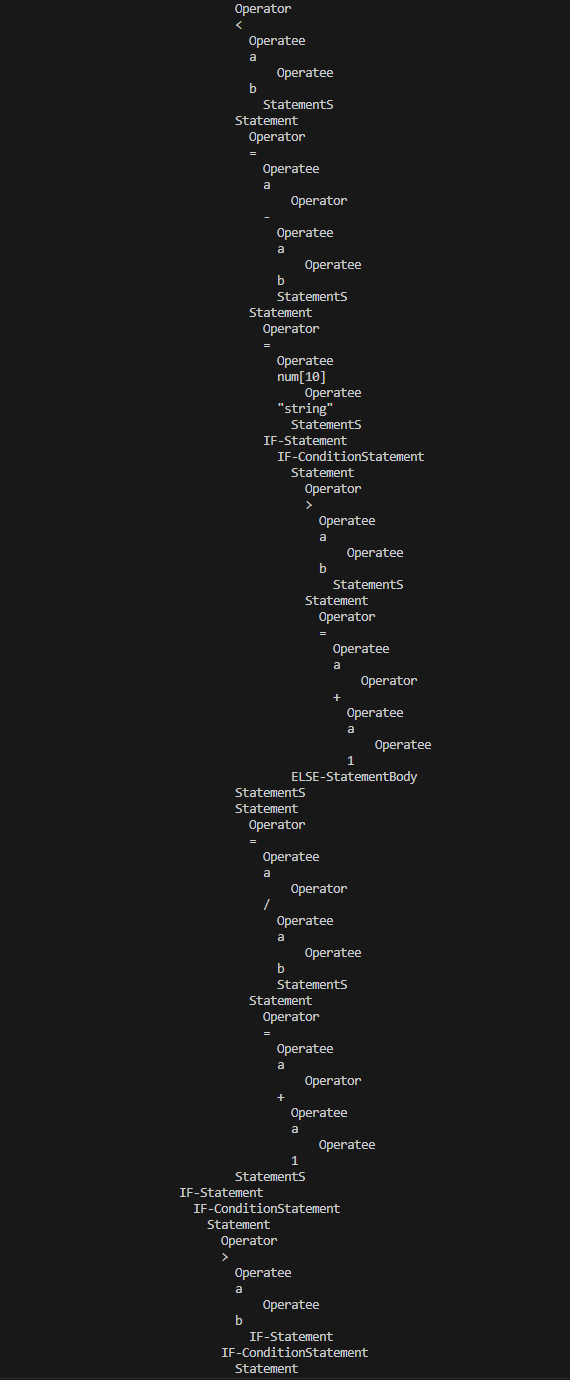


图4-3-4 测试结果

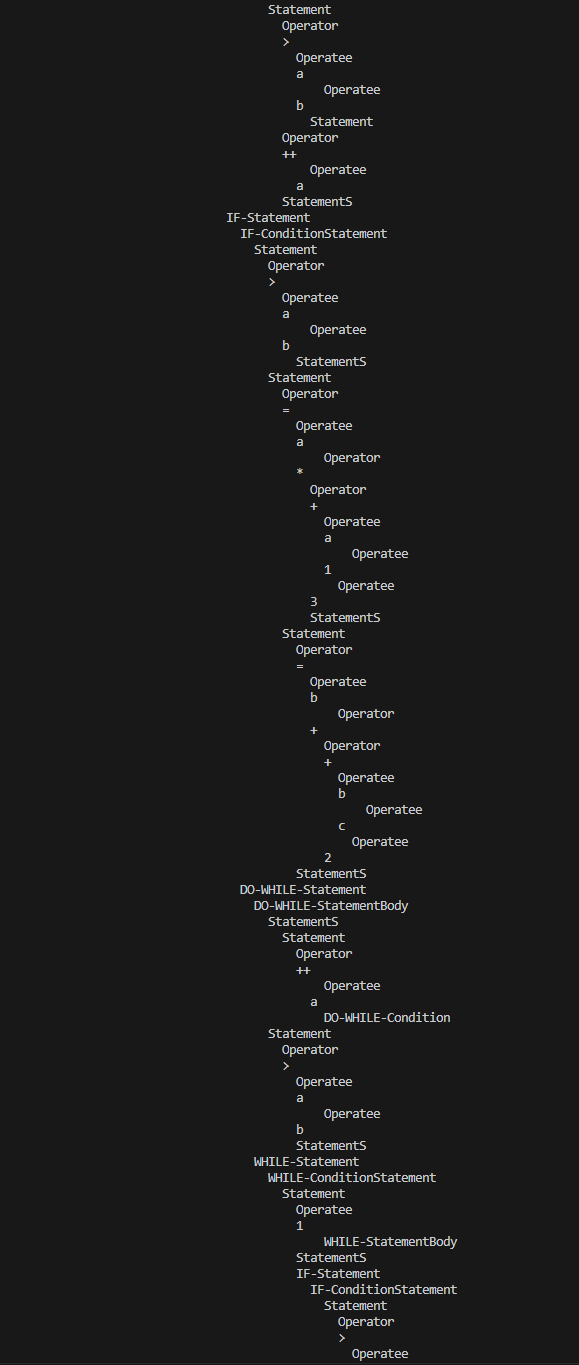


图4-3-5 测试结果



图4-3-6 测试结果

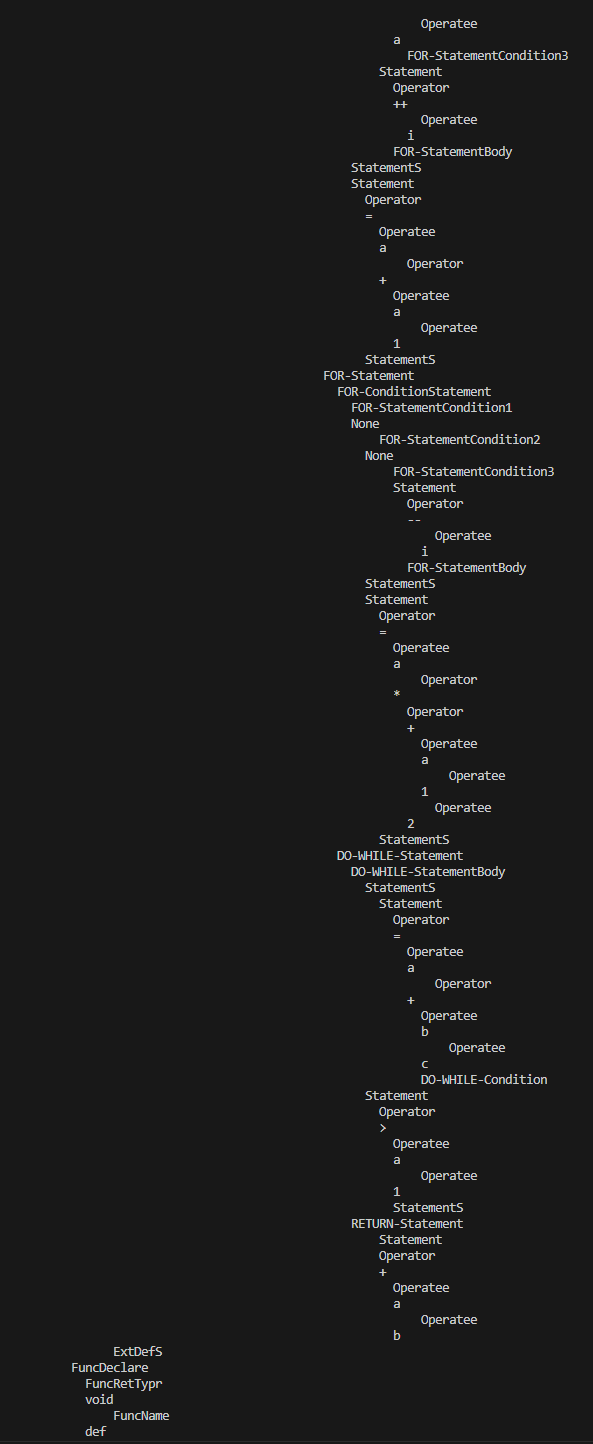


图4-3-7 测试结果

2.错误语法测试分析：

对于错误情况，分为以下情况，如结尾符号出错，定义出错以及各类语句出错

* 1. 结尾符号出错

对于结尾符号出错问题，则是未加分号、右大括号，或多行注释末尾’\*/’缺失。未加分号如图4-2-8所示，多行注释出错如图4-3-9所示。

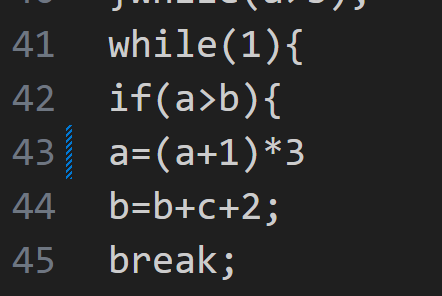
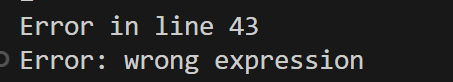
 

图4-3-8 错误语法分析测试用例及结果-未打分号

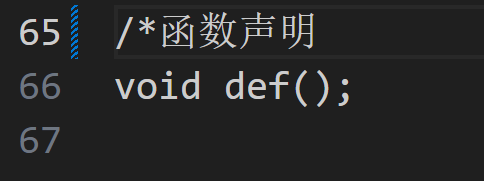
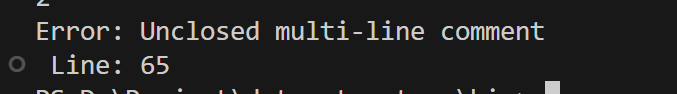
 

图4-3-9错误语法分析测试用例及结果-多行注释

* 1. 定义出错

对于定义出错情况，则主要分为变量定义语句与函数定义语句。

变量定义语句错误设置在变量类型，标识符以及结尾处。如图 4-3-10。而函数定义语句错误则设置在返回值类型，函数名以及形参处。如图 4-3-11

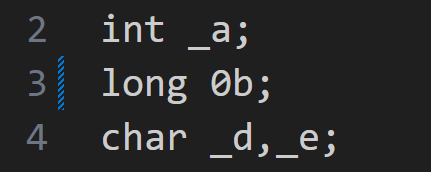
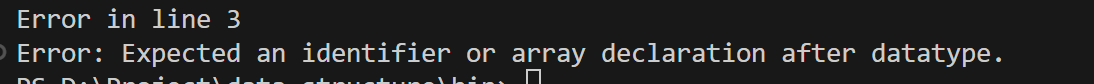
 

图4-3-10 错误语法分析测试用例及结果-变量定义

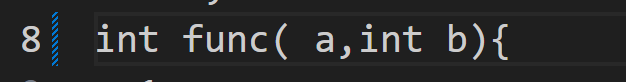
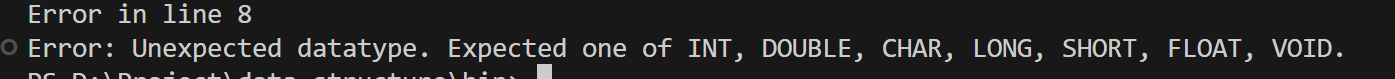
 

图4-3-11 错误语法分析测试用例及结果-函数定义

* 1. 各类语句出错

对于各类语句出错情况，则主要分为 if(-else)语句、while 语句及 for 语句的条件表达式错误，表达式语句错误，各类语句的结尾处理。如括号缺失（如图 4-3-12）等。

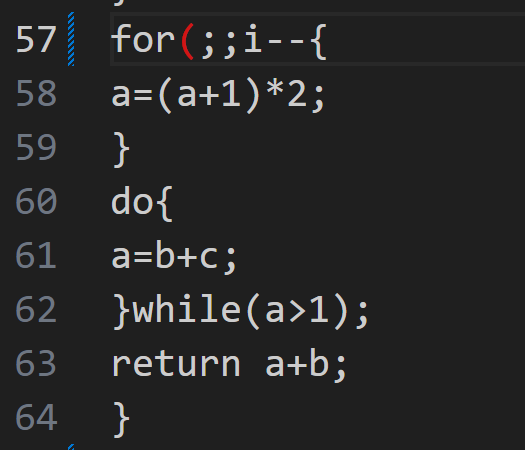
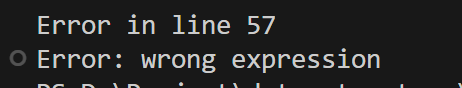
 

图4-3-12 错误语法分析测试用例及结果-括号缺失

### 4.2.3缩进编排测试

对待格式化 C 语言源程序进行缩进编排，并将文件输出到新的cpp文件中。 编排前源文件如图 4-4-1 所示， 编排后格式化文件如图 4-4-2 所示

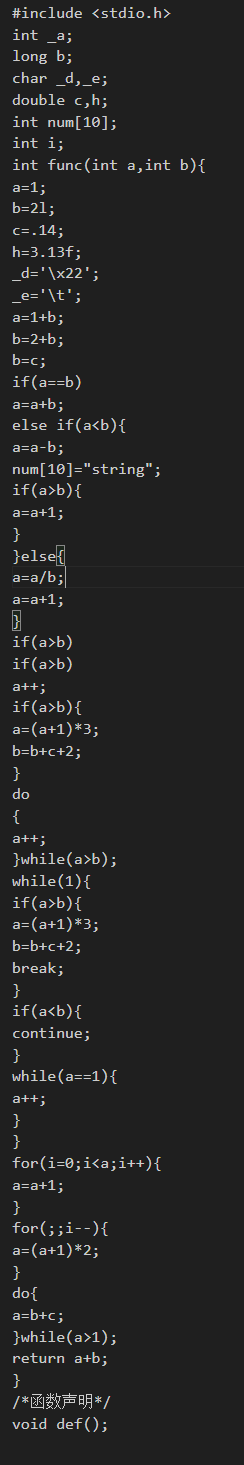
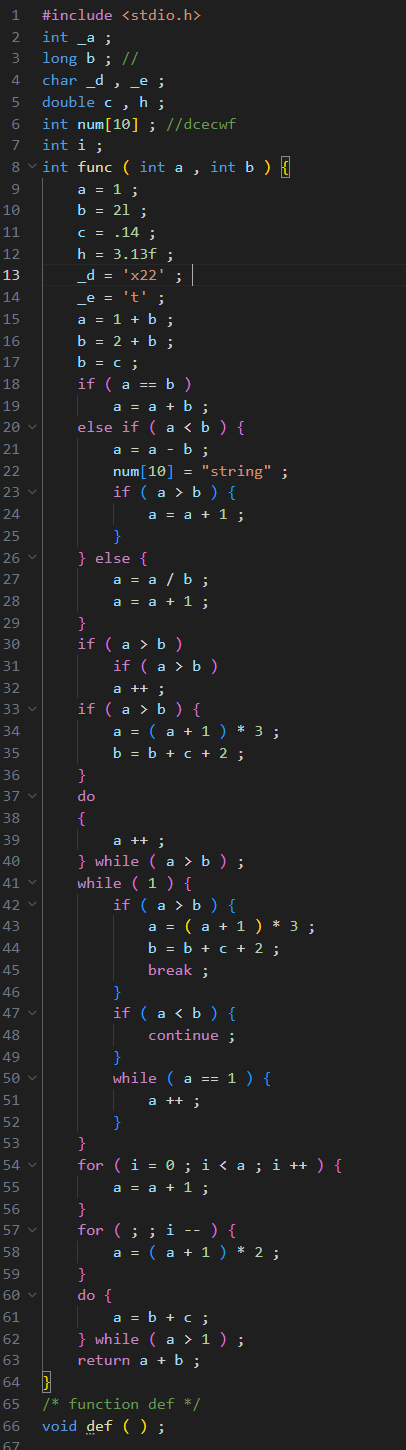
 

图4-4-1 缩进编排前原文件 图4-4-2 缩进编排后cpp文件

### 4.2.4退出系统测试

输入0，即可退出系统

# 5总结与展望

## 5.1全文总结

1. 单词识别（词法分析）：

设计并实现一个确定有穷自动机（DFA）来识别指定C语言子集中的所有单词。

对于每个单词，分配种类编码和单词字符串。

能够处理各种形式的常量和关键字，以及错误处理，即报错功能。

编写测试用例，确保覆盖所有单词种类，并验证报错功能是否正常工作。

2. 语法结构分析：

实现语法分析器，将词法分析的结果组织成抽象语法树（AST）。

支持函数声明、定义、各种语句（if、while、for等）以及嵌套结构。

包括报错功能以指示语法错误的位置。

编写测试用例，确保覆盖所有语法结构，并验证报错功能是否正常工作。

3. 生成抽象语法树：

将语法分析的结果表示为抽象语法树，以便理解源程序的语法结构。

这可以是一个数据结构，每个节点表示一个语法单元（如语句、表达式、函数等）。

可以使用图形方式或其他方式来可视化抽象语法树。

4. 生成源程序文件：

基于生成的抽象语法树，按缩进规则重新生成源程序文件。

对语法树进行遍历，并在输出文件中添加适当的缩进。

验证生成的源程序文件是否符合缩进规则。

## 5.2工作展望

在今后的研究中，围绕着如下几个方面开展工作:

1. 深入研究词法分析和语法分析：

探索更高级的词法分析和语法分析技术，例如使用工具生成器（如Lex和Yacc）来自动生成词法分析器和语法分析器。

研究如何处理更复杂的C语言特性，例如指针、结构、枚举等，以扩展编译器的功能。

1. 优化和代码生成：

着重研究优化技术，以改善生成的目标代码的性能。

探索不同目标架构（如x86、ARM等）上的代码生成，以实现跨平台的编译器。

研究如何进行寄存器分配和指令选择，以提高目标代码的效率。

1. 错误处理和调试支持：

开发更强大的错误检测和报告机制，以帮助程序员识别和解决问题。

实现调试支持，包括生成调试信息、支持源代码级别的断点设置和变量监视等功能。

1. 扩展语言支持：

研究如何添加新的语言特性或扩展现有的C语言子集，以提供更多的编程功能。

支持标准库函数和外部库的链接，以提供更多的功能和便利性。

1. 性能评估和测试：

进行性能评估研究，以了解编译器生成的代码在不同工作负载下的表现。

编写广泛的测试用例，以确保编译器的正确性和稳定性。

研究自动化测试工具和技术，以提高测试效率。

# 6 体会

完成编写一个简化版C语言编译器的工作是一项令人兴奋而具有挑战性的任务。这个过程让我深入了解了编程语言和编译器的内部运作方式，为我的计算机科学知识提供了深刻的增益。通过深入分析C语言的语法和语义规则，我获得了对编程语言设计和实现的更深刻理解。这使我能够更好地理解现有编程语言以及如何更好地使用它们来解决实际问题。

在编写编译器的过程中，我迎来了复杂性的挑战。处理词法分析、语法分析、语义分析和代码生成等多个方面的复杂性要求高度的抽象能力和逻辑思维。这让我深刻认识到在计算机科学领域追求卓越需要不断学习和挑战自己的能力。

另一个重要的方面是错误处理的重要性。编译器的错误处理机制不仅有助于识别和解决程序中的错误，还提供了友好的错误信息，有助于程序员更轻松地调试代码。这个经验教育了我编写可靠和用户友好的错误处理系统的重要性，并加深了我对代码质量和可维护性的理解。

编写编译器还涉及到代码生成的优化。通过研究优化技术，我学会了如何生成高效的目标代码，以提高程序的性能。这不仅有助于提高编译器生成的代码的质量，还有助于我在其他项目中优化代码的能力。

另外，编写编译器需要良好的工程实践。模块化设计、版本控制、文档编写等实践都是不可或缺的。这个过程让我更好地理解了如何组织和管理大型软件项目，这对于今后的工作至关重要。

在这个过程中，我也强调了测试和验证的重要性。编写广泛的测试用例并验证编译器的正确性是至关重要的。通过编写测试用例，我学会了如何检测和修复潜在的问题，这是任何软件工程师必备的技能之一。

总的来说，完成编写一个编译器的工作是一次充实而有益的学习经验。它不仅提高了我的技术水平，还增强了我的问题解决能力和工程实践经验。这个项目让我更深入地了解了计算机科学领域的许多重要概念，并将这些知识和经验应用到今后的职业生涯中。这是一个值得投入时间和精力的学习过程，我期待着在将来的工作中继续应用和发展这些技能和见解

# 参考文献

将报告中引用的参考文献放在此处。

[1] Stephen Prata（著）,姜佑（译).C Primer Plus(第6 版)中文版.人民邮电出版社,2016.4

[2] 严蔚敏等.数据结构(C语言版). 清华大学出版社

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

# 附录

main.cpp

#include <stdio.h>

#include <stdlib.h>

#include "format.h"

#include "getToken.h"

#include "lexicalAnalyse.h"

#include "syntaxAnalyse.h"

char \*filename;

FILE \*fp;

int main(int argc, char \*argv[])

{

    if (argc != 2)

    {

        fprintf(stderr, "%s <filename>", argv[0]);

        exit(-1);

    }

    filename = argv[1];

    int option;

    while (1)

    {

        system("clear");

        printf(" +----------------------------+\n");

        printf(" |    Choose A Function       |\n");

        printf(" |    1. Lexical Analysis     |\n");

        printf(" |    2. Syntax Analysis      |\n");

        printf(" |    3. Show Formated File   |\n");

        printf(" |    0. Exit                 |\n");

        printf(" +----------------------------+\n");

        scanf("%d", &option);

        switch (option)

        {

        case 1:

            fp = fopen(filename, "r");

            lexicalAnalyse();

            printf("Press any key to continue\n");

            fclose(fp);

            getchar();

            getchar();

            break;

        case 2:

            fp = fopen(filename, "r");

            syntaxAnalyse();

            printf("Press any key to continue\n");

            fclose(fp);

            getchar();

            getchar();

            break;

        case 3:

            fp = fopen(filename, "r");

            format();

            printf("Press any key to continue\n");

            fclose(fp);

            getchar();

            getchar();

            break;

        case 0:

            system("clear");

            printf("Thanks for using ;\n");

            return 0;

        default:

            printf("Unknown key, please enter again\n");

            printf("Press any key to continue\n");

            getchar();

            getchar();

            break;

        }

    }

}

getToken.cpp

#include "getToken.h"

char token\_text[20]; // 存放单词自身值

char string\_num[20]; // 存放数字的字符串

int cnt\_lines = 1;

// 判断字符是否是数字

int isNum(char c)

{

    return (c >= 48 && c <= 57);

}

// 判断字符是否是十六进制数字

int isXnum(char c)

{

    return ((isNum(c)) || (c >= 'a' && c <= 'f') || (c >= 'A' && c <= 'F'));

};

// 判断字符是否是字母

int isLetter(char c)

{

    return ((c >= 65 && c <= 90) || (c >= 97 && c <= 122));

}

// 判断字符是否是字母或数字

int isLetterOrNum(char c)

{

    return isLetter(c) || isNum(c);

}

// 添加字符到token\_text

int add2token(char \*token, char c)

{

    int i = 0;

    while (\*(token + i) != '\0')

    {

        i++;

    }

    /\* max len : 20 \*/

    if (i >= 19)

    {

        return -1; // 添加失败

    }

    \*(token + i) = c;

    \*(token + i + 1) = '\0';

    return 1;

}

// 从文件中读取一个单词，返回单词的类型

int getToken(FILE \*fp)

{

    char c;

    \*(token\_text) = '\0';

    do

    {

        c = fgetc(fp);

        if (c == '\n')

        {

            cnt\_lines++;

        }

    } while (c == ' ' || c == '\n');

    if (isLetter(c) || c == '\_')

    {

        do

        {

            add2token(token\_text, c);

        } while (isLetterOrNum(c = fgetc(fp)));

        ungetc(c, fp);

        if (strcmp(token\_text, "int") == 0)

        {

            return INT;

        }

        if (strcmp(token\_text, "double") == 0)

        {

            return DOUBLE;

        }

        if (strcmp(token\_text, "char") == 0)

        {

            return CHAR;

        }

        if (strcmp(token\_text, "short") == 0)

        {

            return SHORT;

        }

        if (strcmp(token\_text, "long") == 0)

        {

            return LONG;

        }

        if (strcmp(token\_text, "float") == 0)

        {

            return FLOAT;

        }

        /\* is keyword \*/

        if (strcmp(token\_text, "if") == 0)

        {

            return IF;

        }

        if (strcmp(token\_text, "else") == 0)

        {

            return ELSE;

        }

        if (strcmp(token\_text, "do") == 0)

        {

            return DO;

        }

        if (strcmp(token\_text, "while") == 0)

        {

            return WHILE;

        }

        if (strcmp(token\_text, "for") == 0)

        {

            return FOR;

        }

        if (strcmp(token\_text, "struct") == 0)

        {

            return STRUCT;

        }

        if (strcmp(token\_text, "break") == 0)

        {

            return BREAK;

        }

        if (strcmp(token\_text, "switch") == 0)

        {

            return SWITCH;

        }

        if (strcmp(token\_text, "case") == 0)

        {

            return CASE;

        }

        if (strcmp(token\_text, "typedef") == 0)

        {

            return TYPEDEF;

        }

        if (strcmp(token\_text, "return") == 0)

        {

            return RETURN;

        }

        if (strcmp(token\_text, "continue") == 0)

        {

            return CONTINUE;

        }

        if (strcmp(token\_text, "void") == 0)

        {

            return VOID;

        }

        for (int i = 0; i < KEYWORD\_LEN; i++)

        {

            if (strcmp(token\_text, KeyWords[i]) == 0)

            {

                return KEYWORD;

            }

            else

            {

                c = fgetc(fp);

                if (c == '[')

                {

                    // 识别数组

                    add2token(token\_text, c);

                    c = fgetc(fp);

                    while (c >= '0' && c <= '9')

                    {

                        add2token(token\_text, c);

                        add2token(string\_num, c);

                        c = fgetc(fp);

                    }

                    if (c != ']')

                    {

                        return ERROR\_TOKEN;

                    }

                    add2token(token\_text, c);

                    return ARRAY;

                }

                else

                {

                    ungetc(c, fp);

                    return IDENT;

                }

            }

        }

    }

    if (isNum(c))

    {

        do

        {

            add2token(token\_text, c);

        } while (isNum(c = fgetc(fp)));

        if (c != '.' && c != 'u' && c != 'l')

        {

            if (c != ' ' && c != ';' && c != ')' && c != '+' && c != '-' &&

                c != '\*' && c != '/')

            {

                return ERROR\_TOKEN;

            }

            ungetc(c, fp);

            return INT\_CONST;

        }

        else if (c == '.')

        {

            c = fgetc(fp);

            if (!isNum(c))

            {

                return ERROR\_TOKEN;

            }

            else

            {

                ungetc(c, fp);

                c = '.';

                add2token(token\_text, c);

                c = fgetc(fp);

                do

                {

                    add2token(token\_text, c);

                } while (isNum(c = fgetc(fp)));

                if (c == 'f')

                {

                    add2token(token\_text, c);

                    return FLOAT\_CONST;

                }

                else

                {

                    ungetc(c, fp);

                }

                return FLOAT\_CONST;

            }

        }

        else if (c == 'u')

        {

            add2token(token\_text, c);

            c = fgetc(fp);

            if (c == 'l')

            {

                add2token(token\_text, c);

                c = fgetc(fp);

                if (c == 'l')

                {

                    add2token(token\_text, c);

                    return INT\_CONST;

                }

                else

                {

                    ungetc(c, fp);

                    return INT\_CONST;

                }

            }

            else

            {

                ungetc(c, fp);

                return INT\_CONST;

            }

        }

        else if (c == 'l')

        {

            add2token(token\_text, c);

            return INT\_CONST;

        }

        else

        {

            return ERROR\_TOKEN;

        }

    }

    switch (c)

    {

    // 以.开头的浮点数

    case '.':

        do

        {

            add2token(token\_text, c);

        } while (isNum(c = fgetc(fp)));

        ungetc(c, fp);

        return FLOAT\_CONST;

    // 字符

    case '\'':

        add2token(token\_text, '\'');

        if ((c = fgetc(fp)) != '\\')

        {

            add2token(token\_text, c);

            if ((c = fgetc(fp)) == '\'')

            {

                add2token(token\_text, c);

                return CHAR\_CONST;

            }

            else

            {

                return ERROR\_TOKEN;

            }

        }

        else

        {

            //  \\

            add2token(token\_text, '\\');

            c = fgetc(fp);

            if (c == 'n' || c == 't' || c == '\\' || c == '\'' ||

                c == '\"')

            {

                add2token(token\_text, c);

                if ((c = fgetc(fp)) == '\'')

                {

                    add2token(token\_text, c);

                    return CHAR\_CONST;

                }

                else

                {

                    return ERROR\_TOKEN;

                }

            }

            else if (c == 'x')

            {

                // 十六进制

                add2token(token\_text, c);

                if (isXnum((c = fgetc(fp))))

                {

                    add2token(token\_text, c);

                    if (isXnum((c = fgetc(fp))))

                    {

                        add2token(token\_text, c);

                    }

                    else

                    {

                        ungetc(c, fp);

                    }

                    if ((c = fgetc(fp)) == '\'')

                    {

                        add2token(token\_text, '\'');

                        return CHAR\_CONST;

                    }

                    else

                    {

                        return ERROR\_TOKEN;

                    }

                }

                else

                {

                    return ERROR\_TOKEN;

                }

            }

            else if (c >= '0' && c <= '7')

            {

                // 八进制

                add2token(token\_text, c);

                if ((c = fgetc(fp)) >= '0' && c <= '7')

                {

                    add2token(token\_text, c);

                    if ((c = fgetc(fp)) >= '0' && c <= '7')

                    {

                        add2token(token\_text, c);

                        if ((c = fgetc(fp)) == '\'')

                        {

                            add2token(token\_text, '\'');

                            return CHAR\_CONST;

                        }

                        else

                        {

                            return ERROR\_TOKEN;

                        }

                    }

                    else if (c == '\'')

                    {

                        add2token(token\_text, '\'');

                        return CHAR\_CONST;

                    }

                    else

                    {

                        return ERROR\_TOKEN;

                    }

                }

                else

                {

                    if (c == '\'')

                    {

                        add2token(token\_text, c);

                        return CHAR\_CONST;

                    }

                    else

                    {

                        ungetc(c, fp);

                        return ERROR\_TOKEN;

                    }

                }

            }

            else

            {

                return ERROR\_TOKEN;

            }

        }

        break;

    // 字符串

    case '"':

        do

        {

            if (c != '\\')

                add2token(token\_text, c);

            if (c == '\\')

            {

                c = fgetc(fp);

                add2token(token\_text, c);

            }

        } while ((c = fgetc(fp)) != '"' && c != '\n');

        if (c == '"')

        {

            add2token(token\_text, '"');

            return STRING\_CONST;

        }

        else

        {

            return ERROR\_TOKEN;

        }

        break;

    // 注释/除号

    case '/':

        add2token(token\_text, c);

        if ((c = fgetc(fp)) == '/')

        {

            do

            {

                add2token(token\_text, c);

            } while ((c = fgetc(fp)) != '\n');

            ungetc(c, fp);

            return ANNO;

        }

        else if (c == '\*')

        {

            int foundClosing = 0; // 添加一个标志来检测是否找到了闭合的注释

            while (1)

            {

                add2token(token\_text, c);

                c = fgetc(fp);

                if (c == '\*')

                {

                    add2token(token\_text, c);

                    if ((c = fgetc(fp)) == '/')

                    {

                        add2token(token\_text, c);

                        return ANNO;

                    }

                }

                if (c == '\n')

                {

                    add2token(token\_text, c);

                    c = '\t';

                    add2token(token\_text, c);

                    add2token(token\_text, c);

                }

                if (c == EOF)

                {

                    // 如果文件结束而没有找到闭合的注释，报错

                    printf("Error: Unclosed multi-line comment\n Line: %d\n", cnt\_lines);

                    exit(0);

                }

            }

            if (!foundClosing)

            {

                // 如果没有找到闭合的注释，报错

                printf("Error: Unclosed multi-line comment Line: %d\n", cnt\_lines);

                exit(0);

            }

        }

        else

        {

            ungetc(c, fp);

            return DIVIDE;

        }

        break;

    // 宏定义/包含

    case '#':

        add2token(token\_text, c);

        if (isLetter(c = fgetc(fp)))

        {

            do

            {

                add2token(token\_text, c);

            } while (isLetter(c = fgetc(fp)));

            if (strcmp(token\_text, "#include") == 0)

            {

                do

                {

                    add2token(token\_text, c);

                } while ((c = fgetc(fp)) != '\n');

                ungetc(c, fp);

                return INCLUDE;

            }

            else if (strcmp(token\_text, "#define") == 0)

            {

                do

                {

                    add2token(token\_text, c);

                } while ((c = fgetc(fp)) != '\n');

                return MACRO;

            }

            else

            {

                return ERROR\_TOKEN;

            }

        }

        else

        {

            return ERROR\_TOKEN;

        }

        break;

    case ',':

        add2token(token\_text, c);

        return COMMA;

    case ';':

        add2token(token\_text, c);

        return SEMI;

    case '=':

        c = fgetc(fp);

        if (c == '=')

        {

            add2token(token\_text, c);

            add2token(token\_text, c);

            return EQ;

        }

        ungetc(c, fp);

        add2token(token\_text, '=');

        return ASSIGN;

    case '!':

        c = fgetc(fp);

        if (c == '=')

        {

            add2token(token\_text, '!');

            add2token(token\_text, '=');

            return NEQ;

        }

        else

        {

            return ERROR\_TOKEN;

        }

    case '+':

        c = fgetc(fp);

        if (c == '+')

        {

            add2token(token\_text, c);

            add2token(token\_text, c);

            return PLUSPLUS;

        }

        ungetc(c, fp);

        add2token(token\_text, '+');

        return PLUS;

    case '-':

        c = fgetc(fp);

        if (c == '-')

        {

            add2token(token\_text, c);

            add2token(token\_text, c);

            return MINUSMINUS;

        }

        ungetc(c, fp);

        add2token(token\_text, '-');

        return MINUS;

    case '(':

        add2token(token\_text, c);

        return LP;

    case ')':

        add2token(token\_text, c);

        return RP;

    case '{':

        add2token(token\_text, c);

        return LB;

    case '}':

        add2token(token\_text, c);

        return RB;

    case '[':

        add2token(token\_text, c);

        return LM;

    case ']':

        add2token(token\_text, c);

        return RM;

    case '\*':

        add2token(token\_text, c);

        return TIMES;

    case '>':

        add2token(token\_text, c);

        if ((c = fgetc(fp)) == '=')

        {

            add2token(token\_text, c);

            return MOREEQUAL;

        }

        else

        {

            ungetc(c, fp);

            return MORE;

        }

    case '<':

        add2token(token\_text, c);

        if ((c = fgetc(fp)) == '=')

        {

            add2token(token\_text, c);

            return LESSEQUAL;

        }

        else

        {

            ungetc(c, fp);

            return LESS;

        }

    case EOF:

        return -1;

    default:

        return ERROR\_TOKEN;

    }

}

getToken.h

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#ifndef LANGUAGE\_FORMAT\_PROCESSING\_TOOL\_GETTOKEN\_H

#define LANGUAGE\_FORMAT\_PROCESSING\_TOOL\_GETTOKEN\_H

typedef enum TokenType {

    ERROR\_TOKEN = 1,

    IDENT,      //标识符

    INT\_CONST,  //整形常量

    FLOAT\_CONST,

    CHAR\_CONST,

    STRING\_CONST,

    KEYWORD,

    INT,

    FLOAT,

    CHAR,

    LONG,

    SHORT,

    DOUBLE,

    IF,

    ELSE,

    DO,

    WHILE,

    FOR,

    STRUCT,

    BREAK,

    SWITCH,

    CASE,

    TYPEDEF,

    RETURN,

    CONTINUE,

    VOID,   // 26

    LB,     //左大括号

    RB,     //右大括号

    LM,     //左中括号

    RM,     //右中括号

    SEMI,   //分号31

    COMMA,  //逗号

    /\*EQ到MINUSMINUS为运算符，必须连在一起\*/

    EQ,          //'=='

    NEQ,         //‘!=’

    ASSIGN,      //'='35

    LP,          //左括号

    RP,          //右括号

    TIMES,       //乘法

    DIVIDE,      //除法

    PLUS,        //加法40

    PLUSPLUS,    //自增运算

    POUND,       //井号42

    MORE,        //大于号

    LESS,        //小于号

    MOREEQUAL,   //大于等于

    LESSEQUAL,   //小于等于

    MINUS,       //减法

    MINUSMINUS,  //自减运算

    ANNO,        //注释

    INCLUDE,     //头文件引用

    MACRO,       //宏定义

    ARRAY,       //数组

} TokenType;

const int KEYWORD\_LEN = 13;

static char\* KeyWords[KEYWORD\_LEN] = {

    "struct",   "break", "else", "switch", "case", "typedef", "return",

    "continue", "for",   "void", "do",     "if",   "while"};

int isNum(char c);

int isLetter(char c);

int isLetterOrNum(char c);

int getToken(FILE\*);

#endif  // LANGUAGE\_FORMAT\_PROCESSING\_TOOL\_GETTOKEN\_H

lexicalAnalyse.cpp

#include "lexicalAnalyse.h"

#include "getToken.h"

extern char token\_text[20];

extern int cnt\_lines;

extern FILE \*fp;

const char \*token\_names[] = {

    "ERROR\_TOKEN",

    "IDENT",

    "INT\_CONST",

    "FLOAT\_CONST",

    "CHAR\_CONST",

    "STRING\_CONST",

    "KEYWORD",

    "INT",

    "DOUBLE",

    "FLOAT",

    "CHAR",

    "SHORT",

    "LONG",

    "IF",

    "ELSE",

    "DO",

    "WHILE",

    "FOR",

    "STRUCT",

    "BREAK",

    "SWITCH",

    "CASE",

    "TYPEDEF",

    "RETURN",

    "CONTINUE",

    "VOID",

    "EQ",

    "NEQ",

    "ASSIGN",

    "LP",

    "RP",

    "LB",

    "RB",

    "LM",

    "RM",

    "SEMI",

    "COMMA",

    "TIMES",

    "DIVIDE",

    "ANNO",

    "PLUS",

    "PLUSPLUS",

    "MINUS",

    "MINUSMINUS",

    "MORE",

    "MOREEQUAL",

    "LESS",

    "LESSEQUAL",

    "INCLUDE",

    "MACRO",

    "ARRAY"};

void printToken(int type, const char \*value)

{

    printf("%-14s\t%s\n", token\_names[type], value);

}

int lexicalAnalyse()

{

    int type;

    printf("\n");

    printf("WordType\tWordValue\n");

    do

    {

        type = getToken(fp);

        if (type != ERROR\_TOKEN)

        {

            printToken(type, token\_text);

        }

        else

        {

            printf("\t\x1b[31mError in line %d\x1b[0m\n", cnt\_lines);

        }

    } while (type != -1);

    return 0;

}

lexicalAnalyse.h

#include <stdio.h>

#include <stdlib.h>

#ifndef LANUAGE\_FORMAT\_PROCESSING\_lexicalAnalyse\_H

#define LANUAGE\_FORMAT\_PROCESSING\_lexicalAnalyse\_H

int lexicalAnalyse();

#endif  // LANUAGE\_FORMAT\_PROCESSING\_lexicalAnalyse\_H

syntaxAnalyse.cpp

#include "syntaxAnalyse.h"

#include "getToken.h"

using namespace std;

extern char token\_text[20];

extern char string\_num[20];

extern int cnt\_lines;

int w, type;

int haveMistake = 0;

extern FILE \*fp;

VDN \*Vroot; // VarName链表根节点

int isVoid, hasReturn, isInRecycle = 0;

// 开始语法分析

void syntaxAnalyse()

{

    ASTTree \*root = program();

    if (root == NULL || haveMistake == 1) // 若有错或root为空则输出语法错误

    {

        printf("Syntax Error!\n");

        return;

    }

    else

    {

        PreorderTranverse(root, 0); // 否则进行先序遍历

    }

}

// 程序入口

ASTTree \*program()

{

    w = getToken(fp);

    while (w == ANNO || w == INCLUDE)

    {

        w = getToken(fp);

    }

    Vroot = (VDN \*)malloc(sizeof(VDN));

    Vroot->size = 0;

    Vroot->next = NULL;

    ASTTree \*p = ExtDefList();

    if (p != NULL)

    {

        if (isVoid == 0 && hasReturn == 0)

        {

            printf("Error: Mising return value\n");

            exit(0);

        }

        ASTTree \*root = p;

        root->type = EXTDEFLIST;

        return root;

    }

    else

    {

        haveMistake = 1;

        return NULL;

    }

}

// 释放抽象语法树

void freeTree(ASTTree \*root)

{

    if (root)

    {

        freeTree(root->l);

        freeTree(root->r);

        free(root);

    }

}

// 形成最表层的节点

ASTTree \*ExtDefList()

{

    if (haveMistake == 1)

    {

        return NULL;

    }

    if (w == -1)

    {

        return NULL;

    }

    ASTTree \*root = (ASTTree \*)malloc(sizeof(ASTTree));

    root->data.data = NULL;

    root->type = EXTDEFLIST;

    root->l = ExtDef();

    w = getToken(fp);

    while (w == ANNO || w == INCLUDE)

    {

        w = getToken(fp);

    }

    root->r = ExtDefList(); // 下一个节点

    return root;

}

// 判断节点是函数还是数组还是变量

ASTTree \*ExtDef()

{

    if (haveMistake == 1)

    {

        return NULL;

    }

    if (w != INT && w != DOUBLE && w != CHAR && w != LONG && w != SHORT &&

        w != FLOAT && w != VOID)

    {

        printf("Mistake in line %d\n", cnt\_lines);

        printf("Error: Unexpected datatype. Expected one of INT, DOUBLE, CHAR, LONG, SHORT, FLOAT, VOID.\n");

        exit(0);

    }

    type = w;

    w = getToken(fp);

    while (w == ANNO || w == INCLUDE)

    {

        w = getToken(fp);

    }

    if (w != IDENT && w != ARRAY)

    {

        printf("Error in line %d\n", cnt\_lines);

        printf("Error: Expected an identifier or array declaration after datatype.\n");

        exit(0);

    }

    char token\_text0[20];

    strcpy(token\_text0, token\_text);

    ASTTree \*p;

    w = getToken(fp);

    while (w == ANNO || w == INCLUDE)

    {

        w = getToken(fp);

    }

    strcpy(token\_text, token\_text0);

    if (w == LP)

    {

        p = FuncDef();

    }

    else if (w == ARRAY)

    {

        p = ArrayDef();

    }

    else

    {

        p = ExtVarDef();

    }

    return p;

}

// 解析数组

ASTTree \*ArrayDef()

{

    if (type == VOID)

    {

        printf("Error in line %d\n", cnt\_lines);

        printf("Error: Can't declare a void type array\n");

        exit(0);

    }

    ASTTree \*root = (ASTTree \*)malloc(sizeof(ASTTree));

    root->type = ARRAYDEF;

    root->l = NULL;

    root->r = NULL;

    root->data.data = NULL;

    ASTTree \*p = (ASTTree \*)malloc(sizeof(ASTTree));

    p->type = ARRAYTYPE;

    p->l = NULL;

    p->r = NULL;

    if (type == INT)

    {

        p->data.data = "int";

    }

    if (type == DOUBLE)

    {

        p->data.data = "double";

    }

    if (type == CHAR)

    {

        p->data.data = "char";

    }

    if (type == FLOAT)

    {

        p->data.data = "float";

    }

    if (type == LONG)

    {

        p->data.data = "long";

    }

    if (type == SHORT)

    {

        p->data.data = "short";

    }

    root->l = p;

    p = (ASTTree \*)malloc(sizeof(ASTTree));

    p->type = ARRAYNAME;

    p->l = NULL;

    p->r = NULL;

    char \*token\_text3 = (char \*)malloc(20 \* sizeof(char));

    strcpy(token\_text3, token\_text);

    p->data.data = token\_text3;

    root->r = p;

    ASTTree \*q = (ASTTree \*)malloc(sizeof(ASTTree));

    q->type = ARRAYSIZE;

    q->l = q->r = NULL;

    q->data.data = string\_num;

    p->l = q;

    return root;

}

// 解析变量

ASTTree \*ExtVarDef()

{

    if (haveMistake == 1)

    {

        return NULL;

    }

    int cnt;

    if (type == VOID)

    {

        printf("Error in line %d\n", cnt\_lines);

        printf("Error: Can't declare a void type variable\n");

        exit(0);

    }

    int u;

    u = addName(token\_text);

    if (u == 1)

    {

        haveMistake = 1;

        return NULL;

    }

    ASTTree \*root = (ASTTree \*)malloc(sizeof(ASTTree));

    root->l = NULL;

    root->r = NULL;

    root->data.data = NULL;

    root->type = EXTVARDEF;

    ASTTree \*p = (ASTTree \*)malloc(sizeof(ASTTree));

    p->l = p->r = NULL;

    p->data.data = NULL;

    p->type = EXTVARTYPE;

    p->data.type = type;

    if (type == INT)

    {

        p->data.data = "int";

    }

    if (type == DOUBLE)

    {

        p->data.data = "double";

    }

    if (type == CHAR)

    {

        p->data.data = "char";

    }

    if (type == FLOAT)

    {

        p->data.data = "float";

    }

    if (type == LONG)

    {

        p->data.data = "long";

    }

    if (type == SHORT)

    {

        p->data.data = "short";

    }

    root->l = p;

    p = (ASTTree \*)malloc(sizeof(ASTTree));

    p->l = p->r = NULL;

    p->data.data = NULL;

    p->type = EXTVARLIST;

    root->r = p;

    p->l = (ASTTree \*)malloc(sizeof(ASTTree));

    p->l->l = p->l->r = NULL;

    p->data.data = NULL;

    p->l->type = EXTVAR;

    char \*token\_text1 = (char \*)malloc(25 \* sizeof(char)); //@

    strcpy(token\_text1, token\_text);

    p->l->data.data = token\_text1;

    while (1)

    {

        if (w != COMMA && w != SEMI)

        {

            if (cnt\_lines > cnt)

            {

                cnt\_lines--;

            }

            printf("Error in line %d\n", cnt\_lines); //%

            printf("Error: Wrong external define\n");

            exit(0);

        }

        if (w == SEMI)

        {

            return root;

        }

        w = getToken(fp);

        if (w != IDENT)

        {

            printf("Error in line %d\n", cnt\_lines); //%

            printf("Error: Wrong external define\n");

            exit(0);

        }

        int u;

        u = addName(token\_text);

        if (u == 1)

        {

            haveMistake = 1;

            return NULL;

        }

        ASTTree \*q = (ASTTree \*)malloc(sizeof(ASTTree));

        q->l = q->r = NULL;

        q->data.data = NULL;

        q->type = EXTVARLIST;

        p->r = q;

        p = q;

        p->l = (ASTTree \*)malloc(sizeof(ASTTree));

        p->l->l = p->l->r = NULL;

        p->l->data.data = NULL;

        p->l->type = EXTVAR;

        char \*token\_text1 = (char \*)malloc(25 \* sizeof(char));

        strcpy(token\_text1, token\_text);

        p->l->data.data = token\_text1;

        cnt = cnt\_lines;

        w = getToken(fp);

        while (w == ANNO || w == INCLUDE)

        {

            w = getToken(fp);

        }

    }

}

// 解析函数

ASTTree \*FuncDef()

{

    if (haveMistake == 1)

    {

        return NULL;

    }

    ASTTree \*root = (ASTTree \*)malloc(sizeof(ASTTree));

    root->data.data = NULL;

    root->type = FUNCDEF;

    ASTTree \*p = (ASTTree \*)malloc(sizeof(ASTTree));

    p->data.data = NULL;

    p->type = FUNCRETURNTYPE;

    p->data.type = type;

    if (type == INT)

    {

        p->data.data = "int";

        isVoid = 0;

    }

    if (type == DOUBLE)

    {

        p->data.data = "double";

        isVoid = 0;

    }

    if (type == CHAR)

    {

        p->data.data = "char";

        isVoid = 0;

    }

    if (type == FLOAT)

    {

        p->data.data = "float";

        isVoid = 0;

    }

    if (type == LONG)

    {

        p->data.data = "long";

        isVoid = 0;

    }

    if (type == SHORT)

    {

        p->data.data = "short";

        isVoid = 0;

    }

    if (type == VOID)

    {

        p->data.data = "void";

        isVoid = 1;

    }

    p->l = NULL;

    p->r = NULL;

    root->l = p;

    // 参数

    ASTTree \*q = (ASTTree \*)malloc(sizeof(ASTTree));

    q->data.data = NULL;

    q->type = FUNCNAME;

    char \*token\_text1 = (char \*)malloc(25 \* sizeof(char));

    strcpy(token\_text1, token\_text);

    q->data.data = token\_text1;

    // 创建一个新的VarName链表节点，用于保存函数名

    VDN \*last = Vroot;

    while (last->next != NULL)

    {

        last = last->next;

    }

    last->next = (VDN \*)malloc(sizeof(VDN));

    last = last->next;

    last->next = NULL;

    last->size = 0;

    root->r = q;

    q->l = FormParaList();

    w = getToken(fp);

    while (w == ANNO || w == INCLUDE)

    {

        w = getToken(fp);

    }

    if (w == SEMI)

    {

        // prototype declare

        root->r->r = NULL;

        root->type = FUNCCLAIM;

    }

    else if (w == LB)

    {

        q->r = CompState();

        q->r->type = FUNCBODY;

    }

    else

    {

        printf("Error in line %d\n", cnt\_lines);

        printf("Error: Error in function declaration\n");

        exit(0);

    }

    return root;

}

// 构造表示函数的形式参数列表的抽象语法树

ASTTree \*FormParaList()

{

    if (haveMistake == 1)

    {

        return NULL;

    }

    w = getToken(fp);

    while (w == ANNO || w == INCLUDE)

    {

        w = getToken(fp);

    }

    if (w == RP)

    {

        return NULL;

    }

    if (w == COMMA)

    {

        w = getToken(fp);

        while (w == ANNO || w == INCLUDE)

        {

            w = getToken(fp);

        }

    }

    ASTTree \*root = (ASTTree \*)malloc(sizeof(ASTTree));

    root->data.data = NULL;

    root->type = FUNCFORMALPARALIST;

    root->l = FormParaDef();

    root->r = FormParaList();

    return root;

}

// 构造表示函数形式参数定义的抽象语法树

ASTTree \*FormParaDef()

{

    if (haveMistake == 1)

    {

        return NULL;

    }

    if (w != INT && w != DOUBLE && w != CHAR && w != LONG && w != SHORT &&

        w != FLOAT)

    {

        printf("Error in line %d\n", cnt\_lines);

        printf("Error: Unexpected datatype. Expected one of INT, DOUBLE, CHAR, LONG, SHORT, FLOAT, VOID.\n");

        exit(0);

    }

    type = w;

    w = getToken(fp);

    while (w == ANNO || w == INCLUDE)

    {

        w = getToken(fp);

    }

    if (w != IDENT)

    {

        printf("Error in line %d\n", cnt\_lines);

        printf("Error: Expected an identifier or array declaration after datatype.\n");

        exit(0);

    }

    ASTTree \*root = (ASTTree \*)malloc(sizeof(ASTTree));

    root->data.data = NULL;

    root->type = FUNCFORMALPARADEF;

    ASTTree \*p = (ASTTree \*)malloc(sizeof(ASTTree));

    p->data.data = NULL;

    p->type = FUNCFORMALPARATYPE;

    p->data.type = type;

    if (type == INT)

    {

        p->data.data = "int";

    }

    if (type == DOUBLE)

    {

        p->data.data = "double";

    }

    if (type == CHAR)

    {

        p->data.data = "char";

    }

    if (type == FLOAT)

    {

        p->data.data = "float";

    }

    if (type == LONG)

    {

        p->data.data = "long";

    }

    if (type == SHORT)

    {

        p->data.data = "short";

    }

    if (type == VOID)

    {

        p->data.data = "void";

    }

    p->l = p->r = NULL;

    root->l = p;

    p = (ASTTree \*)malloc(sizeof(ASTTree));

    p->data.data = NULL;

    p->type = FUNCFORMALPARA;

    int u;

    u = addName(token\_text);

    if (u == 1)

    {

        haveMistake = 1;

        return NULL;

    }

    char \*token\_text1 = (char \*)malloc(25 \* sizeof(char));

    strcpy(token\_text1, token\_text);

    p->data.data = token\_text1;

    p->l = p->r = NULL;

    root->r = p;

    return root;

}

// 构造复合语句的抽象语法树

ASTTree \*CompState()

{

    if (haveMistake == 1)

    {

        return NULL;

    }

    ASTTree \*root = (ASTTree \*)malloc(sizeof(ASTTree));

    root->data.data = NULL;

    root->l = NULL;

    root->r = NULL;

    w = getToken(fp);

    while (w == ANNO || w == INCLUDE)

    {

        w = getToken(fp);

    }

    if (w == INT || w == DOUBLE || w == CHAR || w == LONG || w == SHORT ||

        w == FLOAT)

    {

        root->l = LocalVarDefList();

    }

    else

    {

        root->l = NULL;

    }

    root->r = StateList();

    if (w == RB)

    {

        return root;

    }

    else

    {

        printf("Error: error in compound statement\n");

        exit(0);

        haveMistake = 1;

        freeTree(root);

        return NULL;

    }

}

// 构造局部变量定义列表的抽象语法树

ASTTree \*LocalVarDefList()

{

    if (haveMistake == 1)

    {

        return NULL;

    }

    ASTTree \*root = (ASTTree \*)malloc(sizeof(ASTTree));

    root->data.data = NULL;

    root->type = LOCALVARDEFLIST;

    root->l = NULL;

    root->r = NULL;

    ASTTree \*p = (ASTTree \*)malloc(sizeof(ASTTree));

    p->data.data = NULL;

    p->type = LOCALVARDEF;

    p->l = p->r = NULL;

    root->l = p;

    p->l = (ASTTree \*)malloc(sizeof(ASTTree));

    p->l->data.data = NULL;

    p->l->type = LOCALVARTYPE;

    char \*token\_text1 = (char \*)malloc(25 \* sizeof(char));

    strcpy(token\_text1, token\_text);

    p->l->data.data = token\_text1;

    p->l->l = p->l->r = NULL;

    w = getToken(fp);

    while (w == ANNO || w == INCLUDE)

    {

        w = getToken(fp);

    }

    ASTTree \*q = (ASTTree \*)malloc(sizeof(ASTTree)); // VarNameS结点

    q->data.data = NULL;

    q->l = q->r = NULL;

    p->r = q;

    q->type = LOCALVARNAMELIST;

    q->l = (ASTTree \*)malloc(sizeof(ASTTree)); // LocalVarName结点

    q->l->data.data = NULL;

    q->l->type = LOCALVARNAME;

    char \*token\_text2 = (char \*)malloc(25 \* sizeof(char));

    strcpy(token\_text2, token\_text);

    q->l->data.data = token\_text2;

    q->l->l = q->l->r = NULL;

    int u;

    u = addName(token\_text);

    if (u == 1)

    {

        haveMistake = 1;

        return NULL;

    }

    while (1)

    {

        w = getToken(fp);

        while (w == ANNO || w == INCLUDE)

        {

            w = getToken(fp);

        }

        if (w == SEMI)

        {

            q->r = NULL;

            w = getToken(fp);

            while (w == ANNO || w == INCLUDE)

            {

                w = getToken(fp);

            }

            break;

        }

        else if (w == COMMA)

        {

            w = getToken(fp);

            while (w == ANNO || w == INCLUDE)

            {

                w = getToken(fp);

            }

            ASTTree \*s = (ASTTree \*)malloc(sizeof(ASTTree));

            s->data.data = NULL;

            q->r = s;

            q = q->r;

            q->type = LOCALVARNAMELIST;

            q->l = (ASTTree \*)malloc(sizeof(ASTTree));

            q->l->data.data = NULL;

            q->l->type = LOCALVARNAME;

            char \*token\_text1 = (char \*)malloc(25 \* sizeof(char));

            strcpy(token\_text1, token\_text);

            q->l->data.data = token\_text1;

            int u;

            u = addName(token\_text);

            if (u == 1)

            {

                freeTree(root);

                haveMistake = 1;

                return NULL;

            }

        }

        else

        {

            printf("Error in line %d\n", cnt\_lines); //%

            printf("Error: Wrong definition of variable\n");

            exit(0);

        }

    }

    if (w == INT || w == DOUBLE || w == CHAR || w == LONG || w == SHORT ||

        w == FLOAT)

    {

        root->r = LocalVarDefList();

    }

    else

    {

        root->r = NULL;

    }

    root->r = NULL;

    return root;

}

// 解析语句块

ASTTree \*StateList()

{

    if (haveMistake == 1)

    {

        return NULL;

    }

    ASTTree \*root = NULL;

    ASTTree \*r1 = Statement();

    if (r1 == NULL)

    {

        return NULL;

    }

    else

    {

        // 创建一个新节点来保存状态

        root = (ASTTree \*)malloc(sizeof(ASTTree));

        root->data.data = NULL;

        root->type = STATELIST;

        root->l = r1;

        root->r = NULL;

        w = getToken(fp);

        while (w == ANNO || w == INCLUDE)

        {

            w = getToken(fp);

        }

        if (w != RB)

        {

            root->r = StateList();

            return root;

        }

        else

        {

            return root;

        }

    }

}

// 解析单个语句

ASTTree \*Statement()

{

    if (haveMistake == 1)

    {

        return NULL;

    }

    ASTTree \*root = (ASTTree \*)malloc(sizeof(ASTTree));

    root->l = NULL;

    root->r = NULL;

    root->data.data = NULL;

    switch (w)

    {

    case IF:

    {

        w = getToken(fp);

        while (w == ANNO || w == INCLUDE)

        {

            w = getToken(fp);

        }

        if (w != LP)

        {

            printf("Error in line %d\n", cnt\_lines);

            printf("Error: Wrong IF statement\n");

            exit(0);

        }

        w = getToken(fp);

        while (w == ANNO || w == INCLUDE)

        {

            w = getToken(fp);

        }

        ASTTree \*p1 = (ASTTree \*)malloc(sizeof(ASTTree));

        p1->data.data = NULL;

        p1->type = IFPART;

        p1->l = Expression(RP);

        if (p1->l == NULL)

        {

            printf("Error in line %d\n", cnt\_lines);

            printf("Error: ifStatementCondition部分出错\n");

            exit(0);

        }

        w = getToken(fp);

        while (w == ANNO || w == INCLUDE)

        {

            w = getToken(fp);

        }

        if (w == LB)

        {

            w = getToken(fp);

            while (w == ANNO || w == INCLUDE)

            {

                w = getToken(fp);

            }

            p1->r = StateList();

        }

        else if (w == INT\_CONST || w == FLOAT\_CONST || w == CHAR\_CONST ||

                 w == IDENT || w == KEYWORD || w == IF || w == WHILE ||

                 w == ELSE || w == FOR || w == DO)

        {

            p1->r = Statement();

            p1->r->r = NULL;

        }

        else

        {

            printf("Error: Error within IF\n");

            haveMistake = 1;

            return NULL;

        }

        root->l = p1;

        w = getToken(fp);

        while (w == ANNO || w == INCLUDE)

        {

            w = getToken(fp);

        }

        if (w == ELSE)

        {

            root->type = IFELSESTATEMENT;

            ASTTree \*p2 = (ASTTree \*)malloc(sizeof(ASTTree));

            p2->data.data = NULL;

            p2->type = ELSEPART;

            root->r = p2;

            w = getToken(fp);

            while (w == ANNO || w == INCLUDE)

            {

                w = getToken(fp);

            }

            if (w == LB)

            {

                w = getToken(fp);

                while (w == ANNO || w == INCLUDE)

                {

                    w = getToken(fp);

                }

                p2->r = StateList();

            }

            else if (w == INT\_CONST || w == FLOAT\_CONST ||

                     w == CHAR\_CONST || w == IDENT || w == KEYWORD)

            {

                p2->r = Statement();

                p2->r->r = NULL;

            }

            else if (w == IF)

            {

                p2->l = Statement();

            }

            else

            {

                printf("Error: Error in ELSE\n");

                haveMistake = 1;

                return NULL;

            }

        }

        else

        {

            root->type = IFSTATEMENT;

            returnToken(w, fp);

        }

        return root;

    }

    case WHILE:

    {

        isInRecycle = 1;

        w = getToken(fp);

        while (w == ANNO || w == INCLUDE)

        {

            w = getToken(fp);

        }

        if (w != LP)

        {

            printf("Error in line %d\n", cnt\_lines);

            printf("Error: Error in WHILE\n");

            exit(0);

            haveMistake = 1;

            return NULL;

        }

        w = getToken(fp);

        while (w == ANNO || w == INCLUDE)

        {

            w = getToken(fp);

        }

        ASTTree \*p1 = (ASTTree \*)malloc(sizeof(ASTTree));

        p1->data.data = NULL;

        p1->type = WHILEPART;

        p1->l = NULL;

        p1->r = NULL;

        p1->l = Expression(RP);

        if (p1->l == NULL)

        {

            printf("Error in line %d\n", cnt\_lines);

            printf("Error: whileStatementCondition部分出错\n");

            exit(0);

            haveMistake = 1;

            return NULL;

        }

        ASTTree \*p2 = (ASTTree \*)malloc(sizeof(ASTTree));

        p2->data.data = NULL;

        p2->type = WHILEBODY;

        p2->l = NULL;

        p2->r = NULL;

        w = getToken(fp);

        while (w == ANNO || w == INCLUDE)

        {

            w = getToken(fp);

        }

        if (w == LB)

        {

            w = getToken(fp);

            while (w == ANNO || w == INCLUDE)

            {

                w = getToken(fp);

            }

            p2->r = StateList();

        }

        else if (w == INT\_CONST || w == FLOAT\_CONST || w == CHAR\_CONST ||

                 w == IDENT || w == KEYWORD)

        {

            p2->r = Statement();

            p2->r->r = NULL;

        }

        else

        {

            printf("Error in line %d\n", cnt\_lines);

            printf("Error: Error in WHILE\n");

            exit(0);

            haveMistake = 1;

            return NULL;

        }

        root->type = WHILESTATEMENT;

        root->l = p1;

        root->r = p2;

        isInRecycle = 0;

        return root;

    }

    case FOR:

    {

        isInRecycle = 1;

        w = getToken(fp);

        while (w == ANNO || w == INCLUDE)

        {

            w = getToken(fp);

        }

        if (w != LP)

        {

            printf("Error in line %d\n", cnt\_lines);

            printf("Error: Error in FOR\n");

            exit(0);

            haveMistake = 1;

            return NULL;

        }

        w = getToken(fp);

        while (w == ANNO || w == INCLUDE)

        {

            w = getToken(fp);

        }

        ASTTree \*p1 = (ASTTree \*)malloc(sizeof(ASTTree));

        p1->data.data = NULL;

        p1->type = FORPART;

        ASTTree \*q = (ASTTree \*)malloc(sizeof(ASTTree)); // FOR part 1

        p1->l = q;

        q->type = FORPART1;

        q->data.data = NULL;

        q->l = Expression(SEMI);

        if (q->l == NULL)

        {

            q->data.data = "None";

        }

        w = getToken(fp);

        while (w == ANNO || w == INCLUDE)

        {

            w = getToken(fp);

        }

        q->r = (ASTTree \*)malloc(sizeof(ASTTree)); // FOR part 2

        q = q->r;

        q->type = FORPART2;

        q->data.data = NULL;

        q->l = Expression(SEMI);

        if (q->l == NULL)

        {

            q->data.data = "None";

        }

        w = getToken(fp);

        while (w == ANNO || w == INCLUDE)

        {

            w = getToken(fp);

        }

        q->r = (ASTTree \*)malloc(sizeof(ASTTree)); // FOR part 3

        q = q->r;

        q->r = NULL;

        q->type = FORPART3;

        q->data.data = NULL;

        q->l = Expression(RP);

        if (q->l == NULL)

        {

            q->data.data = "None";

        }

        ASTTree \*p2 = (ASTTree \*)malloc(sizeof(ASTTree)); // FOR body

        p2->l = NULL;

        p2->r = NULL;

        p2->type = FORBODY;

        p2->data.data = NULL;

        w = getToken(fp);

        while (w == ANNO || w == INCLUDE)

        {

            w = getToken(fp);

        }

        if (w == LB)

        {

            w = getToken(fp);

            while (w == ANNO || w == INCLUDE)

            {

                w = getToken(fp);

            }

            p2->r = StateList();

        }

        else if (w == INT\_CONST || w == FLOAT\_CONST || w == CHAR\_CONST ||

                 w == IDENT || w == KEYWORD)

        {

            p2->r = Statement();

            p2->r->r = NULL;

        }

        else

        {

            printf("Error: Error in FOR\n");

            exit(0);

        }

        root->type = FORSTATEMENT;

        root->l = p1;

        root->r = p2;

        isInRecycle = 0;

        return root;

    }

    case RETURN:

    {

        hasReturn = 1;

        if (isVoid == 1)

        {

            printf("Error in line %d\n", cnt\_lines);

            printf("Error: There should be no return statement\n");

            exit(0);

        }

        root->type = RETURNSTATEMENT;

        root->l = NULL;

        root->r = NULL;

        w = getToken(fp);

        while (w == ANNO || w == INCLUDE)

        {

            w = getToken(fp);

        }

        root->r = Expression(SEMI);

        return root;

    }

    case DO:

    {

        isInRecycle = 1;

        w = getToken(fp);

        while (w == ANNO || w == INCLUDE)

        {

            w = getToken(fp);

        }

        if (w != LB)

        {

            printf("Error in line %d\n", cnt\_lines);

            printf("Error: mising bracket in do-while\n");

            exit(0);

        }

        ASTTree \*p1 = (ASTTree \*)malloc(sizeof(ASTTree));

        p1->type = DOWHILEBODY;

        p1->l = p1->r = NULL;

        ASTTree \*p2 = (ASTTree \*)malloc(sizeof(ASTTree));

        p2->type = DOWHILECONDITION;

        p2->l = p2->r = NULL;

        root->l = p1;

        root->r = p2;

        root->data.data = p1->data.data = p2->data.data = NULL;

        w = getToken(fp);

        while (w == ANNO || w == INCLUDE)

        {

            w = getToken(fp);

        }

        p1->l = StateList();

        w = getToken(fp);

        while (w == ANNO || w == INCLUDE)

        {

            w = getToken(fp);

        }

        if (w != WHILE)

        {

            printf("Error in line %d\n", cnt\_lines);

            printf("Error: missing WHILE in do-while\n");

            freeTree(root);

            exit(0);

        }

        root->type = DOWHILESTATEMENT;

        w = getToken(fp);

        while (w == ANNO || w == INCLUDE)

        {

            w = getToken(fp);

        }

        if (w != LP)

        {

            printf("Error in line %d\n", cnt\_lines);

            printf("Error: missing condition in do-while\n");

            freeTree(root);

            exit(0);

        }

        w = getToken(fp);

        while (w == ANNO || w == INCLUDE)

        {

            w = getToken(fp);

        }

        p2->l = Expression(RP);

        if (p2->l == NULL)

        {

            printf("Error in line %d\n", cnt\_lines);

            printf("Error: missing condition in do-while\n");

            exit(0);

        }

        w = getToken(fp);

        while (w == ANNO || w == INCLUDE)

        {

            w = getToken(fp);

        }

        if (w != SEMI)

        {

            printf("Error in line %d\n", cnt\_lines);

            printf("Error: mising semicolon in do-while\n");

            freeTree(root);

            exit(0);

        }

        isInRecycle = 0;

        return root;

    }

    case BREAK:

    {

        w = getToken(fp);

        while (w == ANNO || w == INCLUDE)

        {

            w = getToken(fp);

        }

        if (w != SEMI)

        {

            printf("Error in line %d\n", cnt\_lines);

            printf("Error: mising semicolon in BREAK\n");

            exit(0);

        }

        if (isInRecycle == 0)

        {

            printf("Error in line %d\n", cnt\_lines);

            printf("Error: unexpected BREAK\n");

            exit(0);

        }

        root->type = BREAKSTATEMENT;

        return root;

    }

    case CONTINUE:

    {

        w = getToken(fp);

        while (w == ANNO || w == INCLUDE)

        {

            w = getToken(fp);

        }

        if (w != SEMI)

        {

            printf("Error in line %d\n", cnt\_lines);

            printf("Error: mising semicolon in continue\n");

            exit(0);

        }

        if (isInRecycle == 0)

        {

            printf("Error in line %d\n", cnt\_lines);

            printf("Error: unexpected continue\n");

            exit(0);

        }

        root->type = CONTINUESTATEMENT;

        return root;

    }

    case INT\_CONST:

    case FLOAT\_CONST:

    case CHAR\_CONST:

    case IDENT:

    case ARRAY:

        return Expression(SEMI);

    }

    return root;

}

// 表达式解析函数，endsym为结束符

ASTTree \*Expression(int endsym)

{

    if (haveMistake == 1)

    {

        return NULL;

    }

    if (w == endsym)

    {

        return NULL;

    }

    int error = 0;

    stack<ASTTree \*> op;

    ASTTree \*p = (ASTTree \*)malloc(sizeof(ASTTree));

    p->data.data = NULL;

    p->type = OPERATOR;

    p->data.type = POUND;

    op.push(p);

    stack<ASTTree \*> opn;

    ASTTree \*t, \*t1, \*t2, \*root;

    while (((w != endsym) || (op.top()->data.type != POUND)) && !error)

    {

        if (op.top()->data.type == RP)

        {

            if (op.size() < 3)

            {

                error++;

                break;

            }

            op.pop();

            op.pop();

        }

        if (w == IDENT)

        {

            if (checkname\_exist(token\_text) == 0)

            {

                haveMistake = 1;

            }

        }

        if (w == IDENT || w == INT\_CONST || w == FLOAT\_CONST ||

            w == CHAR\_CONST || w == ARRAY || w == STRING\_CONST)

        {

            p = (ASTTree \*)malloc(sizeof(ASTTree));

            p->data.data = NULL;

            p->type = OPERAND;

            char \*token\_text1 = (char \*)malloc(25 \* sizeof(char));

            strcpy(token\_text1, token\_text);

            p->data.data = token\_text1;

            opn.push(p);

            w = getToken(fp);

            while (w == ANNO || w == INCLUDE)

            {

                w = getToken(fp);

            }

        }

        else if (w == endsym)

        {

            ASTTree \*p = (ASTTree \*)malloc(sizeof(ASTTree));

            p->data.data = NULL;

            p->type = OPERATOR;

            p->data.type = POUND;

            while (op.top()->data.type != POUND)

            {

                t2 = opn.top();

                if (!t2 && (op.top()->data.type) != PLUSPLUS &&

                    (op.top()->data.type) != MINUSMINUS)

                {

                    error++;

                    break;

                }

                if (t2 != NULL)

                    opn.pop();

                if (opn.size() == 0)

                {

                    t1 = NULL;

                }

                else

                {

                    t1 = opn.top();

                }

                if (!t1 && (op.top()->data.type) != PLUSPLUS &&

                    (op.top()->data.type) != MINUSMINUS)

                {

                    error++;

                    break;

                }

                if (t1 != NULL)

                    opn.pop();

                t = op.top();

                if (!t)

                {

                    error++;

                    break;

                }

                op.pop();

                t->l = t1;

                t->r = t2;

                opn.push(t);

            }

            if (opn.size() != 1)

            {

                error++;

            }

        }

        else if (w >= EQ && w <= MINUSMINUS)

        {

            char \*token\_text1 = (char \*)malloc(25 \* sizeof(char)); //@

            switch (Precede(op.top()->data.type, w))

            {

            case '<':

                p = (ASTTree \*)malloc(sizeof(ASTTree));

                p->data.data = NULL;

                p->type = OPERATOR;

                p->data.type = w;

                strcpy(token\_text1, token\_text);

                p->data.data = token\_text1;

                op.push(p);

                w = getToken(fp);

                while (w == ANNO || w == INCLUDE)

                {

                    w = getToken(fp);

                }

                break;

            case '=':

                t = op.top();

                if (!t)

                {

                    error++;

                    op.pop();

                }

                w = getToken(fp);

                while (w == ANNO || w == INCLUDE)

                {

                    w = getToken(fp);

                }

                break;

            case '>':

                t2 = opn.top();

                if (!t2 && (op.top()->data.type) != PLUSPLUS &&

                    (op.top()->data.type) != MINUSMINUS)

                {

                    error++;

                    break;

                }

                if (t2 != NULL)

                    opn.pop();

                if (opn.size() == 0)

                {

                    t1 = NULL;

                }

                else

                {

                    t1 = opn.top();

                }

                if (!t1 && (op.top()->data.type) != PLUSPLUS &&

                    (op.top()->data.type) != MINUSMINUS)

                {

                    error++;

                    break;

                }

                if (t1 != NULL)

                    opn.pop();

                t = op.top();

                if (!t)

                {

                    error++;

                    break;

                }

                op.pop();

                t->l = t1;

                t->r = t2;

                opn.push(t);

                p = (ASTTree \*)malloc(sizeof(ASTTree));

                p->data.data = NULL;

                p->type = OPERATOR;

                p->data.type = w;

                strcpy(token\_text1, token\_text);

                p->data.data = token\_text1;

                op.push(p);

                w = getToken(fp);

                while (w == ANNO || w == INCLUDE)

                {

                    w = getToken(fp);

                }

                break;

            case '\0':

                printf("Error in line %d\n", cnt\_lines);

                printf("Error: unknown operator\n");

                exit(0);

            }

        }

        else

        {

            error = 1;

        }

    };

    if ((opn.size() == 1) && (op.top()->data.type == POUND) && error == 0)

    {

        t = opn.top();

        opn.pop();

        root = (ASTTree \*)malloc(sizeof(ASTTree));

        root->data.data = NULL;

        root->l = NULL;

        root->r = NULL;

        root->type = EXPRESSION;

        root->l = t;

        return root;

    }

    else

    {

        printf("Error in line %d\n", cnt\_lines);

        printf("Error: wrong expression\n");

        exit(0);

    }

}

// 判断运算符优先级

char Precede(int c1, int c2)

{

    if (haveMistake == 1)

    {

        return NULL;

    }

    if (c1 == PLUS || c1 == MINUS)

    {

        switch (c2)

        {

        case PLUS:

        case MINUS:

        case RP:

        case POUND:

        case MORE:

        case LESS:

        case MOREEQUAL:

        case LESSEQUAL:

        case EQ:

        case NEQ:

        case ASSIGN:

            return '>';

        case TIMES:

        case DIVIDE:

        case LP:

        case PLUSPLUS:

        case MINUSMINUS:

            return '<';

        default:

            return '\0';

            break;

        }

    }

    else if (c1 == TIMES || c1 == DIVIDE)

    {

        switch (c2)

        {

        case PLUS:

        case MINUS:

        case RP:

        case POUND:

        case TIMES:

        case DIVIDE:

        case MORE:

        case LESS:

        case MOREEQUAL:

        case LESSEQUAL:

        case EQ:

        case NEQ:

            return '>';

        case LP:

        case PLUSPLUS:

        case MINUSMINUS:

            return '<';

        default:

            return '\0';

        }

    }

    else if (c1 == LP)

    {

        switch (c2)

        {

        case PLUS:

        case MINUS:

        case TIMES:

        case DIVIDE:

        case LP:

        case PLUSPLUS:

        case MINUSMINUS:

            return '<';

        case RP:

            return '=';

        case MORE:

        case LESS:

        case MOREEQUAL:

        case LESSEQUAL:

        case EQ:

        case NEQ:

        case POUND:

            return '>';

        default:

            return '\0';

        }

    }

    else if (c1 == RP)

    {

        switch (c2)

        {

        case PLUS:

        case MINUS:

        case TIMES:

        case DIVIDE:

        case LP:

        case MORE:

        case LESS:

        case MOREEQUAL:

        case LESSEQUAL:

        case EQ:

        case NEQ:

        case PLUSPLUS:

        case MINUSMINUS:

        case POUND:

            return '>';

        default:

            return '\0';

        }

    }

    else if (c1 == ASSIGN)

    {

        switch (c2)

        {

        case PLUS:

        case MINUS:

        case TIMES:

        case DIVIDE:

        case LP:

        case MORE:

        case LESS:

        case MOREEQUAL:

        case LESSEQUAL:

        case EQ:

        case NEQ:

        case PLUSPLUS:

        case MINUSMINUS:

            return '<';

        case POUND:

            return '>';

        default:

            return '\0';

        }

    }

    else if (c1 == MORE || c1 == LESS || c1 == MOREEQUAL || c1 == LESSEQUAL)

    {

        switch (c2)

        {

        case PLUS:

        case MINUS:

        case TIMES:

        case DIVIDE:

        case LP:

        case PLUSPLUS:

        case MINUSMINUS:

            return '<';

        case RP:

        case MORE:

        case LESS:

        case MOREEQUAL:

        case LESSEQUAL:

        case EQ:

        case NEQ:

        case POUND:

            return '>';

        default:

            return '\0';

        }

    }

    else if (c1 == EQ || c1 == NEQ)

    {

        switch (c2)

        {

        case PLUS:

        case MINUS:

        case TIMES:

        case DIVIDE:

        case LP:

        case MORE:

        case LESS:

        case MOREEQUAL:

        case LESSEQUAL:

        case PLUSPLUS:

        case MINUSMINUS:

            return '<';

        case RP:

        case EQ:

        case NEQ:

        case POUND:

            return '>';

        default:

            return '\0';

        }

    }

    else if (c1 == POUND)

    {

        switch (c2)

        {

        case PLUS:

        case MINUS:

        case TIMES:

        case DIVIDE:

        case LP:

        case MORE:

        case LESS:

        case MOREEQUAL:

        case LESSEQUAL:

        case RP:

        case EQ:

        case NEQ:

        case ASSIGN:

        case PLUSPLUS:

        case MINUSMINUS:

            return '<';

        case POUND:

            return '=';

        default:

            return '\0';

        }

    }

    else if (c1 == PLUSPLUS || c1 == MINUSMINUS)

    {

        switch (c2)

        {

        case PLUS:

        case MINUS:

        case TIMES:

        case DIVIDE:

        case LP:

        case MORE:

        case LESS:

        case MOREEQUAL:

        case LESSEQUAL:

        case EQ:

        case NEQ:

        case ASSIGN:

        case POUND:

            return '>';

        case RP:

            return '<';

        default:

            return '\0';

        }

    }

}

// 将token返回到文件流

void returnToken(int w, FILE \*fp)

{

    int digit = strlen(token\_text);

    int i;

    for (i = 0; i < digit; i++)

    {

        ungetc(token\_text[digit - 1 - i], fp);

    }

}

// 显示节点类型

void showType(int type)

{

    switch (type)

    {

    case 1:

        printf("ExtDefS\n");

        break;

    case 2:

        printf("ExtDef\n");

        break;

    case 3:

        printf("ExtVarType\n");

        break;

    case 4:

        printf("ExtVarNameS\n");

        break;

    case 5:

        printf("ExtVarName\n");

        break;

    case 6:

        printf("FuncDef\n");

        break;

    case 7:

        printf("FuncRetTypr\n");

        break;

    case 8:

        printf("FuncName\n");

        break;

    case 9:

        printf("FuncParamS\n");

        break;

    case 10:

        printf("FuncPramaDef\n");

        break;

    case 11:

        printf("FuncPramaType\n");

        break;

    case 12:

        printf("FuncPramaName\n");

        break;

    case 13:

        printf("FuncBody\n");

        break;

    case 14:

        printf("LocalVarDefS\n");

        break;

    case 15:

        printf("LocalVarDef\n");

        break;

    case 16:

        printf("LocalVarType\n");

        break;

    case 17:

        printf("LocalVarNameS\n");

        break;

    case 18:

        printf("LocalVarName\n");

        break;

    case 19:

        printf("StatementS\n");

        break;

    case 20:

        printf("Operatee\n");

        break;

    case 21:

        printf("Operator\n");

        break;

    case 22:

        printf("Statement\n");

        break;

    case 23:

        printf("IF-ConditionStatement\n");

        break;

    case 24:

        printf("ELSE-StatementBody\n");

        break;

    case 25:

        printf("IF-Statement\n");

        break;

    case 26:

        printf("IF-ELSE-Statement\n");

        break;

    case 27:

        printf("WHILE-Statement\n");

        break;

    case 28:

        printf("WHILE-ConditionStatement\n");

        break;

    case 29:

        printf("WHILE-StatementBody\n");

        break;

    case 30:

        printf("FOR-Statement\n");

        break;

    case 31:

        printf("FOR-ConditionStatement\n");

        break;

    case 32:

        printf("FOR-StatementCondition1\n");

        break;

    case 33:

        printf("FOR-StatementCondition2\n");

        break;

    case 34:

        printf("FOR-StatementCondition3\n");

        break;

    case 35:

        printf("FOR-StatementBody\n");

        break;

    case 36:

        printf("RETURN-Statement\n");

        break;

    case 37:

        printf("BREAK-Statement\n");

        break;

    case 38:

        printf("DO-WHILE-Statement\n");

        break;

    case 39:

        printf("DO-WHILE-StatementBody\n");

        break;

    case 40:

        printf("DO-WHILE-Condition\n");

        break;

    case 41:

        printf("continueStatement\n");

        break;

    case 42:

        printf("FuncDeclare\n");

        break;

    case 43:

        printf("ArrDef\n");

        break;

    case 44:

        printf("ArrType\n");

        break;

    case 45:

        printf("ArrName\n");

        break;

    case 46:

        printf("ArrSize\n");

        break;

    }

}

// 前序遍历抽象语法树并输出

void PreorderTranverse(ASTTree \*root, int depth)

{

    if (root == nullptr)

        printf("  ");

    else

    {

        int i;

        for (i = 0; i < depth; i++)

        {

            printf("  ");

        }

        showType(root->type);

        if (root->data.data != NULL)

        {

            for (i = 0; i < depth; i++)

            {

                printf("  ");

            }

            printf("%s\n", root->data.data);

        }

        PreorderTranverse(root->l, depth + 1);

        PreorderTranverse(root->r, depth + 1);

    }

}

// 添加名称到变量名列表中

int addName(char \*token\_text)

{

    if (haveMistake == 1)

    {

        return NULL;

    }

    int i, flag = 0;

    VDN \*p = Vroot;

    while (p->next != NULL)

    {

        p = p->next; // 找到最后一个节点

    }

    for (i = 0; i < (p->size); i++)

    {

        if (strcmp(token\_text, p->variable[i]) == 0) // 如果变量名已经存在

        {

            flag = 1;

            break;

        }

    }

    if (flag == 1) // 如果变量名已经存在

    {

        printf("Error in line %d\n", cnt\_lines);

        printf("Error: Duplicate variable define\n");

        exit(0);

        haveMistake = 1;

        return 1;

    }

    char \*savename = (char \*)malloc(25 \* sizeof(char));

    strcpy(savename, token\_text);

    p->variable[p->size] = savename;

    p->size++;

    return flag;

}

// 检查变量名是否已在列表中声明

int checkname\_exist(char \*token\_text)

{

    if (haveMistake == 1)

    {

        return NULL;

    }

    int i;

    int flag = 0;

    VDN \*p = Vroot;

    while (p->next != NULL)

    {

        p = p->next;

    }

    for (i = 0; i < (p->size); i++)

    {

        if (strcmp(token\_text, p->variable[i]) == 0)

        {

            flag = 1;

            break;

        }

    }

    for (i = 0; i < (Vroot->size); i++)

    {

        if (strcmp(token\_text, Vroot->variable[i]) == 0)

        {

            flag = 1;

            break;

        }

    }

    if (flag == 0)

    {

        printf("Error in line %d\n", cnt\_lines);

        printf("Error: Using undeclared variable\n");

        exit(0);

        haveMistake = 1;

    }

    return flag;

}

syntaxAnalyse.h

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <iostream>

#include <stack>

#ifndef LANUAGE\_FORMAT\_PROCESSING\_GRAMMAR\_ANALYSE\_H

#define LANUAGE\_FORMAT\_PROCESSING\_GRAMMAR\_ANALYSE\_H

typedef struct ASTTree

{

    struct ASTTree \*l;

    struct ASTTree \*r;

    int type;

    struct data

    {

        int type;

        char \*data;

    } data;

} ASTTree;

typedef struct VariableDefNode

{

    int size;           // 参数个数

    char \*variable[20]; // 参数名

    struct VariableDefNode \*next;

} VDN;

typedef enum DataType

{

    EXTDEFLIST = 1,

    EXTVARDEF,

    EXTVARTYPE,

    EXTVARLIST,         // 外部变量名序列结点

    EXTVAR,             // 外部变量名结点

    FUNCDEF,            // 函数定义结点

    FUNCRETURNTYPE,     // 函数返回值类型结点

    FUNCNAME,           // 函数名结点

    FUNCFORMALPARALIST, // 函数形式参数序列结点

    FUNCFORMALPARADEF,  // 函数形式参数结点10

    FUNCFORMALPARATYPE, // 函数形参类型结点

    FUNCFORMALPARA,     // 函数形参名结点

    FUNCBODY,           // 函数体结点

    LOCALVARDEFLIST,    // 局部变量定义序列结点

    LOCALVARDEF,        // 局部变量定义结点15

    LOCALVARTYPE,       // 局部变量类型结点

    LOCALVARNAMELIST,   // 局部变量名序列

    LOCALVARNAME,       // 局部变量名

    STATELIST,          // 语句序列结点

    OPERAND,            // 操作数结点20

    OPERATOR,           // 运算符结点

    EXPRESSION,         // 表达式

    IFPART,             // if语句部分

    ELSEPART,           // else部分

    IFSTATEMENT,        // if语句25

    IFELSESTATEMENT,    // if-else语句

    WHILESTATEMENT,     // while语句结点

    WHILEPART,          // while条件语句结点

    WHILEBODY,          // while语句体

    FORSTATEMENT,       // for语句结点30

    FORPART,            // for条件语句

    FORPART1,           // for语句条件一

    FORPART2,           // for语句条件二

    FORPART3,           // for语句条件三

    FORBODY,            // for语句体35

    RETURNSTATEMENT,    // return语句

    BREAKSTATEMENT,     // break语句

    DOWHILESTATEMENT,   // do-while循环语句

    DOWHILEBODY,        // do-while语句体

    DOWHILECONDITION,   // do-while条件40

    CONTINUESTATEMENT,  // continue语句

    FUNCCLAIM,          // 函数声明

    ARRAYDEF,           // 数组定义

    ARRAYTYPE,          // 数组类型

    ARRAYNAME,          // 数组名45

    ARRAYSIZE,          // 数组大小

};

void syntaxAnalyse();

void freeTree(ASTTree \*root);

ASTTree \*program();

ASTTree \*ExtDefList();

ASTTree \*ExtDef();

ASTTree \*ExtVarDef();

ASTTree \*FuncDef();

ASTTree \*FormParaList();

ASTTree \*FormParaDef();

ASTTree \*CompState();

ASTTree \*LocalVarDefList();

ASTTree \*StateList();

ASTTree \*Statement();

ASTTree \*Expression(int endsym);

char Precede(int c1, int c2);

void returnToken(int w, FILE \*fp);

void showType(int type);

void PreorderTranverse(ASTTree \*root, int depth);

int addName(char \*token\_text);

int checkname\_exist(char \*token\_text);

ASTTree \*ArrayDef();

#endif // LANUAGE\_FORMAT\_PROCESSING\_GRAMMAR\_ANALYSE\_H

format.cpp

#include "format.h"

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include "getToken.h"

extern FILE \*fp;

extern char token\_text[20];

FILE \*fp1 = fopen("D:\\Project\\data\_structure\\src\\output.c", "a");

// 输出到output.c

void Output(char \*res)

{

    if (fp1 == NULL)

    {

        printf("Cannot open file strike any key exit!");

        getchar();

        exit(1);

    }

    fprintf(fp1, "%s", res);

}

// 格式化

void format()

{

    int tabs = 0; // 控制缩进

    int flag = 0; // 用于标记无大括号但需要缩进的情况

    token \*root, \*tail, \*p;

    while (1)

    {

        root = readline();

        if (root == NULL)

        {

            break;

        }

        tail = root;

        while (tail->next != NULL)

        {

            tail = tail->next;

        }

        if ((root == tail) && (strcmp(tail->str, "}") == 0))

        {

            tabs--;

        }

        if ((root != tail) && (strcmp(root->str, "}") == 0))

        {

            tabs--;

        }

        for (int i = 0; i < tabs; i++)

        { // 输出缩进

            printf("    ");

            Output("    ");

        }

        if (flag == 1)

        {

            tabs--;

            flag = 0;

        }

        p = root;

        while (p != NULL)

        {

            printf("%s ", p->str);

            Output(p->str);

            Output(" ");

            p = p->next;

        }

        printf("\n");

        Output("\n");

        if (strcmp(tail->str, "{") == 0)

        {

            tabs++;

        }

        p = root;

        while (p != NULL)

        {

            if ((strcmp(p->str, "if") == 0) || (strcmp(p->str, "for") == 0) ||

                (strcmp(p->str, "while") == 0))

            {

                if (strcmp(tail->str, "{") != 0 &&

                    strcmp(tail->str, ";") != 0)

                {

                    tabs++;

                    flag = 1;

                }

            }

            p = p->next;

        }

    }

    fclose(fp1);

}

// 读取一行

token \*readline()

{

    int word = getToken(fp);

    if (word == -1)

    {

        return NULL;

    }

    token \*root = NULL, \*tail = NULL;

    root = (token \*)malloc(sizeof(token));

    char \*token\_text1 = (char \*)malloc(25 \* sizeof(char));

    strcpy(token\_text1, token\_text);

    root->str = token\_text1;

    root->next = NULL;

    tail = root;

    char c;

    while ((c = fgetc(fp)) != '\n')

    {

        ungetc(c, fp);

        getToken(fp);

        tail->next = (token \*)malloc(sizeof(token));

        tail = tail->next;

        token\_text1 = (char \*)malloc(25 \* sizeof(char));

        strcpy(token\_text1, token\_text);

        tail->str = token\_text1;

        tail->next = NULL;

    }

    return root;

}

format.h

#ifndef LANUAGE\_FORMAT\_PROCESSING\_FORMAT\_H

#define LANUAGE\_FORMAT\_PROCESSING\_FORMAT\_H

typedef struct token {

    char\* str;

    struct token\* next;

} token;

void format();

token\* readline();

#endif  // LANUAGE\_FORMAT\_PROCESSING\_FORMAT\_H