

华中科技大学

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：BD2202

学 号：U202215641

姓 名：曲睿

指导教师：李开

报告日期：2023 年 10 月 6 日

计算机科学与技术学院

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。

在本题中，我们的目标是设计实现抽象语法树，并借此进行源程序格式化。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] 王生原，董渊，张素琴，吕映芝等. 编译原理（第 3 版）. 北京：清华大学出版社. 前 4 章
- [2] 严蔚敏等. 数据结构（C 语言版）. 北京：清华大学出版社

目录

任务书	I
1 引言	1
1.1 课题背景与意义.....	1
1.2 国内外研究现状.....	1
1.3 课程设计的主要研究工作.....	1
2 系统需求分析与总体设计	3
2.1 课题背景与意义.....	3
2.2 系统总体设计.....	3
3 系统详细设计	5
3.1 常量、全局变量、数据类型的定义.....	5
3.2 有关数据结构的定义.....	6
3.3 主要算法设计.....	8
4 系统实现与测试	21
4.1 系统实现.....	21
4.2 系统测试.....	22
5 总结与展望	29
5.1 全文总结.....	29
5.2 工作展望.....	29
6 体会	30
参考文献	31
附录	32

1 引言

1.1 课题背景与意义

抽象语法树是一种用于表示程序源代码的树形结构，它可以反映出代码的语法和语义信息，从而方便对代码进行分析、优化、转换等操作。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。其作为程序的一种中间表示形式，在程序分析等诸多领域有广泛的应用。利用抽象语法树可以方便地实现多种源程序处理工具，比如源程序浏览器、智能编辑器、语言翻译器等。

本课题旨在通过学习并实现建立抽象语法树的过程，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、形式语言自动机以及编译技术中的递归下降语法分析法，对编译原理有一定的初步认知。

1.2 国内外研究现状

自 20 世纪 60 年代高级语言兴起以来，高级语言源程序的格式化处理便一直是计算机研究与开发领域的一个活跃课题。如今，虽然基于高级语言的源程序处理工具的设计已经是一门相对成熟的计算机技术，几乎所有的语言都内置有格式化处理工具，但随着算法的不断更新，代码规模的不断增大，处理工具的效率依然是核心研究项目之一。

近十年来，国外在关于高级语言格式处理工具的设计上逐渐采用大量更加复杂的算法，主要用于推断和简化程序中的信息；而国内现阶段仍主要着眼于特定处理器的特定部分，仍具有一定局限性。

1.3 课程设计的主要研究工作

本次课程设计的主要工作是由源程序生成其抽象语法树，在逻辑上，仅包含 2 个重要的阶段，一是词法分析，识别词法规则中已定义的单词；二是语法分析，在词法分析已识别程序中所有单词的含义的基础上，判断单词序列是否满足已定义的语法规则，同时生成抽象语法树。

虽然目标是对源程序进行格式处理，但如果我们可以得到抽象语法树，则只

需对抽象语法树进行先根遍历，为不同层级的结点赋予不同的缩进格数，即可完成缩进编排，因此格式处理也应建立在上述 2 个阶段的基础之上。

实现词法分析所需的相关技术是有穷自动机，用扩展巴克斯（EBNF）范式定义各类单词，包括标识符、整型常量、字符串常量等，并对应确定有穷自动机 DFA；实现语法分析所需的相关技术是递归下降子程序法，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，从而调用相应子程序进行语法结构分析，在分析的同时生成一棵抽象语法树，并记录各成分的缩进量；最后根据语法分析时获得的抽象语法树与各成分缩进量，创建风格统一的格式化缩进编排源程序文件。

2 系统需求分析与总体设计

2.1 系统需求分析

本次课程设计要求所做系统可以将源程序的语法结构用树的形式表示，即生成抽象语法树，并能够以缩进编排的方式生成格式化文件。

系统应当能对源程序进行预编译，对包含头文件、宏定义、行注释、块注释进行提前处理；应当能准确识别预编译后的源程序中的所有单词及其字面值，并存储词法错误以便展示；应当能准确分析源程序中的所有语法结构，构建抽象语法树，识别语法错误；应当能对源程序进行风格统一的缩进编排，输出格式化文件。

2.2 系统总体设计

系统包含4个主要功能模块：词法分析、语法分析与语法树生成、源程序格式化文件生成、选择源文件。

（1）词法分析：包含预编译、词法分析两个过程。由于包含头文件、宏定义、注释等做法在代码编写中不可或缺，但又不是词法分析与语法分析的对象，于是考虑引入预编译，对源程序的预处理部分与注释部分提前调用Pre_Process函数进行处理，若该部分有格式错误，则返回预编译错误；若格式正确，则生成已进行头文件、注释去除与宏替换的中间文件mid_file。对mid_file文件逐词调用get_token函数，识别单词，返回单词类型、单词字面值或错误行数。

（2）语法分析与语法树生成：包含预编译、语法树生成、语法树遍历三个过程。调用Pre_Process函数生成中间文件mid_file，若生成成功，则对mid_file文件调用program函数逐成分生成抽象语法树结点，若出现语法错误，则返回错误所在行；若语法正确，则调用PostRootTraverse函数，进行语法树遍历。

（3）源程序格式化文件生成：包含预编译、语法树生成、格式化三个过程。调用Pre_Process函数生成中间文件mid_file，若生成成功，则对mid_file文件调用program函数生成语法树，若生成成功，则调用FormatCFile函数生成格式化文件formatted_file。

(4) 选择源文件：进入系统时要求输入初始文件名，在运行中亦可随时选择其他源文件进行操作。

功能(1) — (3)均为从预编译开始的完整过程，都可以独立地完成其功能，而无需按顺序进行操作；词法分析与语法分析时的报错功能包含行号，方便对源程序进行调试。

系统模块结构图如图2-1所示。

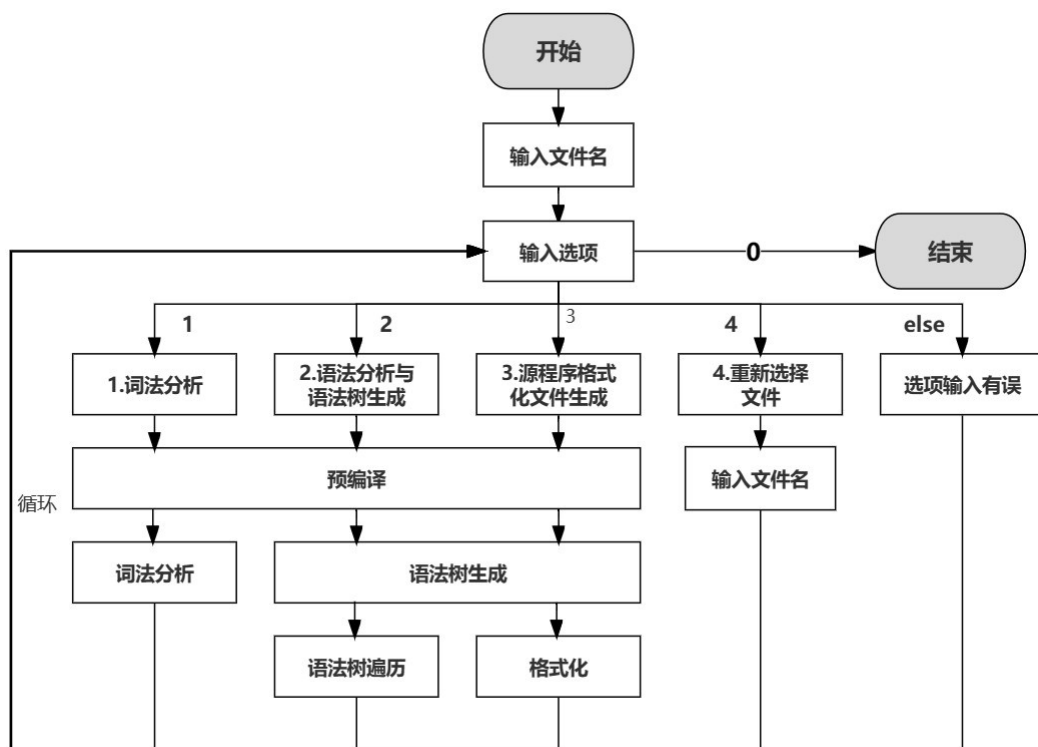


图 2-1 系统模块结构图

3 系统详细设计

3.1 常量、全局变量、数据类型的定义

为使系统便于管理与修改，集中定义常数、全局变量、数据类型并统一各常量、变量的含义是必不可少的一步。

(1) 常量定义：在代码中以`#define`定义。一部分常量用于作为函数返回值，指示函数运行状态；一部分常量作为树、栈等数据结构的容量等无需改变的数据；最后一部分常量作为语法分析中表达式运算符的标志。所有常量定义如表3-1所示。

表 3-1 常量定义

常量	值	含义
OK	1	函数返回值，运行成功
ERROR	0	函数返回值，运行出错
INFEASIBLE	-1	函数返回值，运行中止
YES	1	函数返回值，结果为真
NO	0	函数返回值，结果为假
MAXLEN	100	单词最大长度
MAX_NODE_NUM	1000	抽象语法树结点最大容量
STACK_INIT_SIZE	100	顺序栈初始容量
STACK_INCREMENT	10	顺序栈扩容步长
MAX_OPERATOR_NUM	11	表达式中操作符总数
BEGIN_END	-1	表达式结束标志

(2) 全局变量定义：在代码中以`extern`定义。一部分全局变量用于词法分析时记录当前字符、当前单词的字面值、当前行数；一部分全局变量用于语法分析时记录当前单词类型及其字面值、当前成分缩进量；最后一部分全局变量以哈希表的形式存储关键字。所有全局变量定义如表3-2所示。

表 3-2 全局变量定义

全局变量	含义
NowCh	当前读取的字符
token_text	当前单词字面值
LineNum	当前行数
PKind	当前单词种类值
PTokenKind	当前关键字种类值
PTokenText	当前关键字字面值
PIndentNum	当前成分缩进量
KeyHashCode	关键字哈希表

(3)数据类型定义:在代码中以typedef定义。函数运行状态Status定义为int,树结点元素类型TElemType定义为char*,顺序栈元素类型SElemType定义为树指针CTree*。

3.2 有关数据结构的定义

为使系统在进行预编译、词法分析、语法树生成等过程时更加高效,为需要处理的数据定义合适的数据结构十分重要。

(1)预编译:在此过程中我们需要进行宏替换,因此我们需要存储每个宏的自身值与替换值,从而在源程序中搜索自身值并用替换值进行替换。依此思路,为宏定义Macro结构,包含2个数据项,自身值ident与替换值str,均为char*类型。

(2)词法分析:在此过程中我们需要进行单词类别的判断,对于除关键字与一些特别的标识符外的类型,诸如常量、运算符,均定义enum枚举类型token_kind;而对于关键字,采用字符串匹配进行判断,若采用朴素循环匹配,关键字数量较多,且与某个关键字具有相同前缀的标识符也需要完全遍历,效率较低,因此考虑采用字典序哈希表存储关键字,关键字哈希表存储结构如图3-1所示。

对于关键字,我们既需要其种类码,也需要其字面值,因此为关键字定义

Keyword结构，包含2个数据项，字面值key，类型为char数组，种类码kind，类型为enum token_kind。根据关键字首字符范围与个数，定义全局变量KeyHashCode，类型为Keyword二维数组。

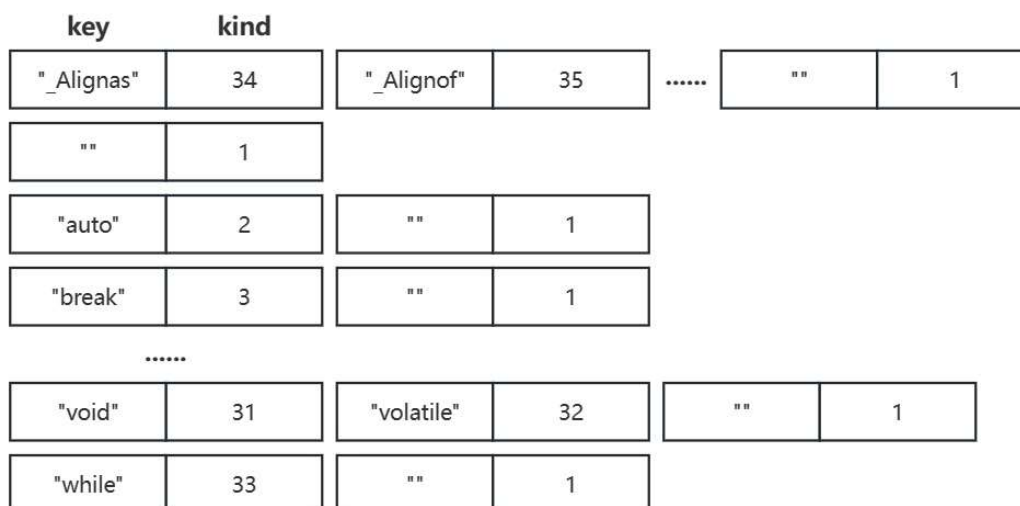


图3-1 关键字哈希表存储结构

词法分析时，若有词法错误，应能在分析结束后统一输出，因此需要存储错误行数。行数输出的顺序与记录顺序相同，符合“先进先出”，因此考虑使用队列存储行数。由于源程序长度未知，错误行数可能很大，也可能很小，不宜用顺序队列直接分配空间，会造成浪费，考虑使用链式队列。链式队列的存储结构如图3-2所示，队列结点QNode包含2个数据项，数据域data，类型为QElemType，即行数，指针域next；链式队列LinkQueue包含2个数据项，front与rear，分别指向队首与队尾，均为QPtr类型，即结点指针。

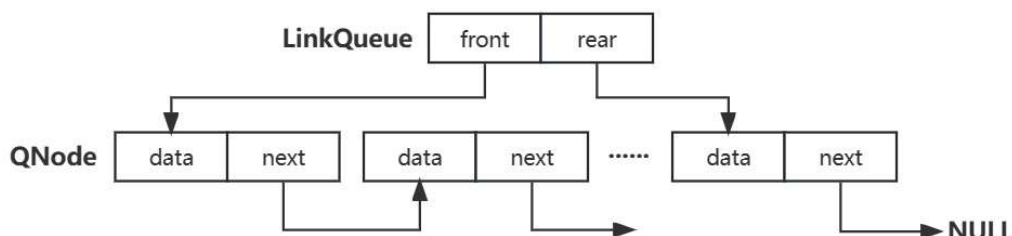


图3-2 链式队列存储结构

（3）语法树生成：对于语法树，考虑到格式化代码的缩进要求，对于每个成分，我们需要其代码块内的子成分以及与其并列地位的其他成分，即对于每个结点，我们更关心其孩子信息与兄弟信息，而非其他的如父结点信息，因此选择

用孩子链表表示法构建邻接表实现。语法树的存储结构如图3-3所示，CTree包含结点数n、根结点下标root与头结点数数组nodes；CTListNode包含数据项data、缩进量IndentNum与首孩子指针FirstChild；CTNode包含该结点下标pos与右兄弟指针next。

通过root可直接得到根结点nodes[root]，对其FirstChild所指向的单链表进行遍历，即可访问其所有孩子。

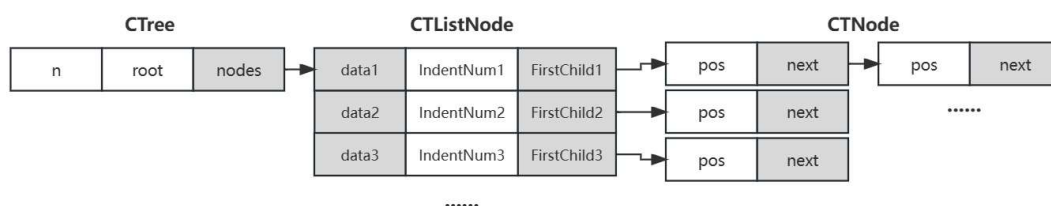


图3-3 语法树存储结构

在生成表达式子树的过程中，需要用到栈结构，对运算符进行基于优先度比较的进出栈操作，考虑到本系统所处理的表达式一般较短，可以直接定义顺序栈结构SqStack，包含3个数据项，当前容量StackSize，类型为int，栈底元素base与栈顶元素top，类型均为SElemType*。

3.3 主要算法设计

系统共有词法分析、预编译、语法树生成、语法树遍历、文件格式化五个模块算法，设计思路如下。

3.3.1 词法分析函数算法设计

词法分析函数声明为 `int GetToken(FILE* fp)`，接收一个文件指针作为参数，返回指针当前所指向的单词的种类编码。由于词法分析需要识别出规则中定义的C语言单词子集，包括标识符、关键字、常量、运算符与定界符，及其字面值。为便于确定单词类型，我们定义enum枚举类型，以表示各类单词种类编码，如IDENT表示标识符，INT表示关键字int，INT_CONST表示整型常量。同时定义全局变量token_text以临时存储单词字面值，以便后续处理。

词法分析算法的主要思想是有穷自动机DFA，每次从状态0开始，通过当

前值与之后几个字符值确定其最终状态。白色状态表示过程，尚未达到最终状态；浅灰色状态表示已达到最终状态，需返回单词编码，且无需退回字符；深灰色状态表示已达到最终状态，需返回单词编码，但多读取了一个字符，需退回。

在状态 0 读取空白符与制表符时，应忽略，继续读取；读取换行符时，行数自增 1，继续读取。

标识符与关键字的过程状态转换图如图 3-4 所示。当前字符为字母或下划线时，继续读取直至当前字符既非字母也非下划线，将当前 `token_text` 在关键字哈希表中进行查找，若查找成功，返回关键字；否则返回标识符。

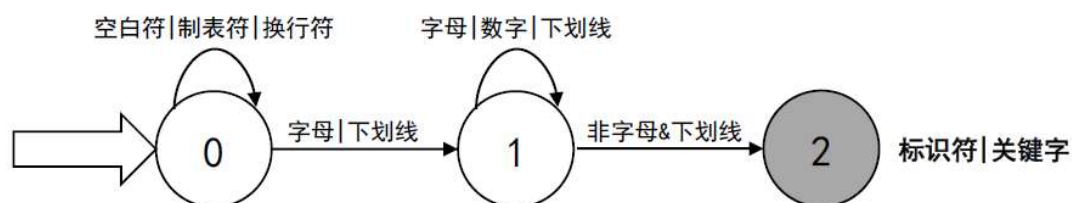


图3-4 标识符与关键字的过程状态转换图

常量的分析较为复杂，故将其分为三大类，整型与浮点型常量、字符型常量与字符串常量。对于整型与浮点型常量，考虑其以十进制表示，即首位数字不为 0 时，过程状态转换图如图 3-5 所示，需注意其后缀 `u`、`U` 表示无符号整型，`l`、`L` 表示长整型，`ul`、`UL` 表示无符号长整型，如无后缀，则默认为整型；若当前字符为小数点，则进行浮点数分析，若后缀为 `e`、`E`，则为指数表示形式，需继续读取一个整型数字，若后缀为 `f`、`F`，则为单精度浮点数，否则默认为双精度浮点数。

若首位数字为 0，可能是以八进制或十六进制表示的整型或浮点型常量，也可能是单独的 0 整型常量，因此读取后面一位。如果为 `x`、`X`，则为十六进制数，向后读取的字符必须为十六进制中可能出现的字符，否则应报错；如果为 0-7 中的某个数，则为八进制数，向后读取的字符必须为八进制中可能出现的字符，否则应报错。除此之外，在十进制常量中考虑过的小数点、指数表示问题，在八进制、十六进制中也应进行讨论。此情况的过程状态转换图如图 3-6 所示。

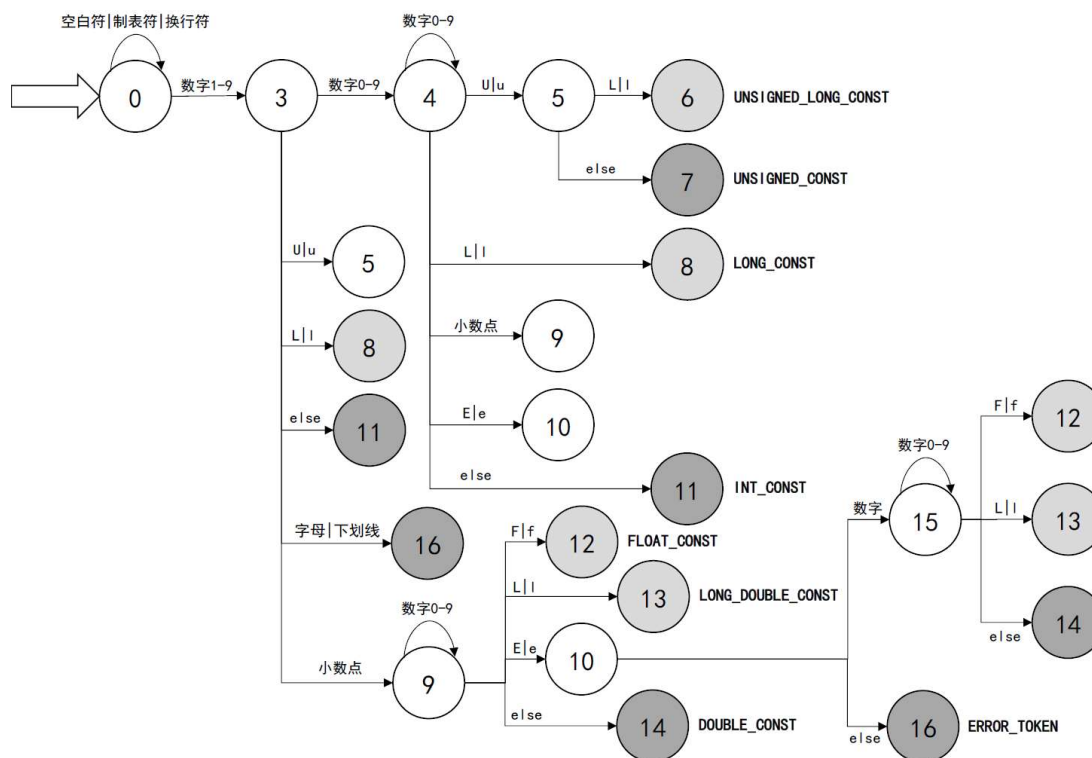


图3-5 十进制整型与浮点型常量的过程状态转换图

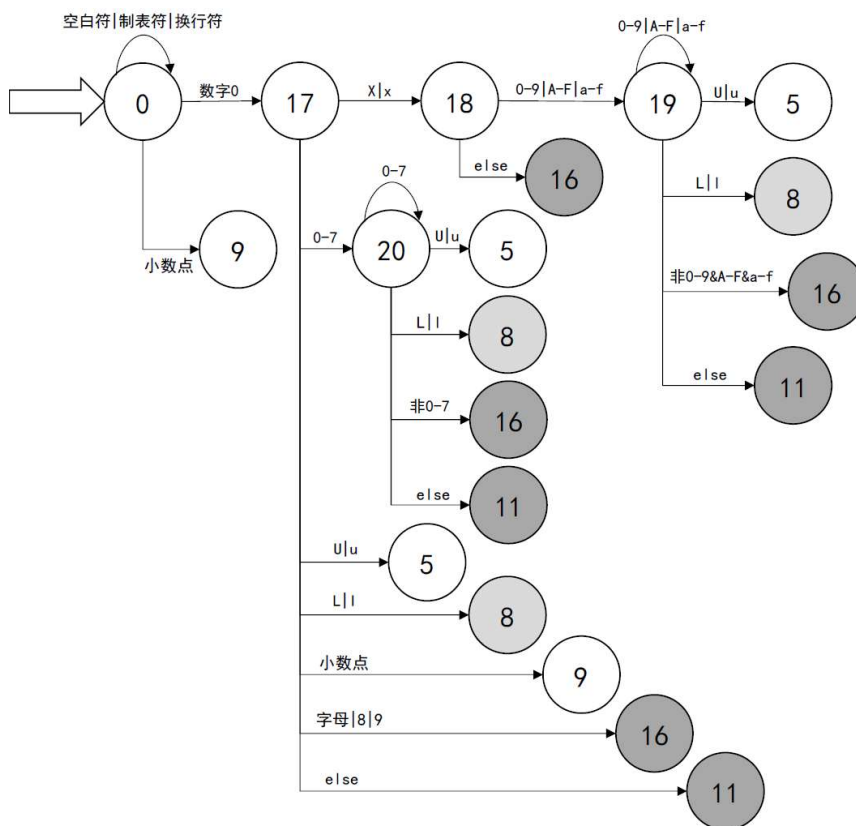


图3-6 八、十六进制整型与浮点型常量的过程状态转换图

对于字符型常量，需考虑普通字符与转义字符，其中转义字符可以用数字进行表示。除此之外，需要考虑非法的字符型常量，包括引号不对应、特殊符号未使用转义字符表示的情况。同时，如果未找到与左单引号对应的右单引号，考虑到字符型常量的长度一定是较少的并且处于单行之中，应将换行符作为终止条件，防止向后继续读取字符。

对于字符串常量，需要特判在串中使用`\\`进行换行的情况，`\\`之后不能为右单引号，否则应报错。字符型与字符串常量的过程状态转换图如图 3-7 所示。

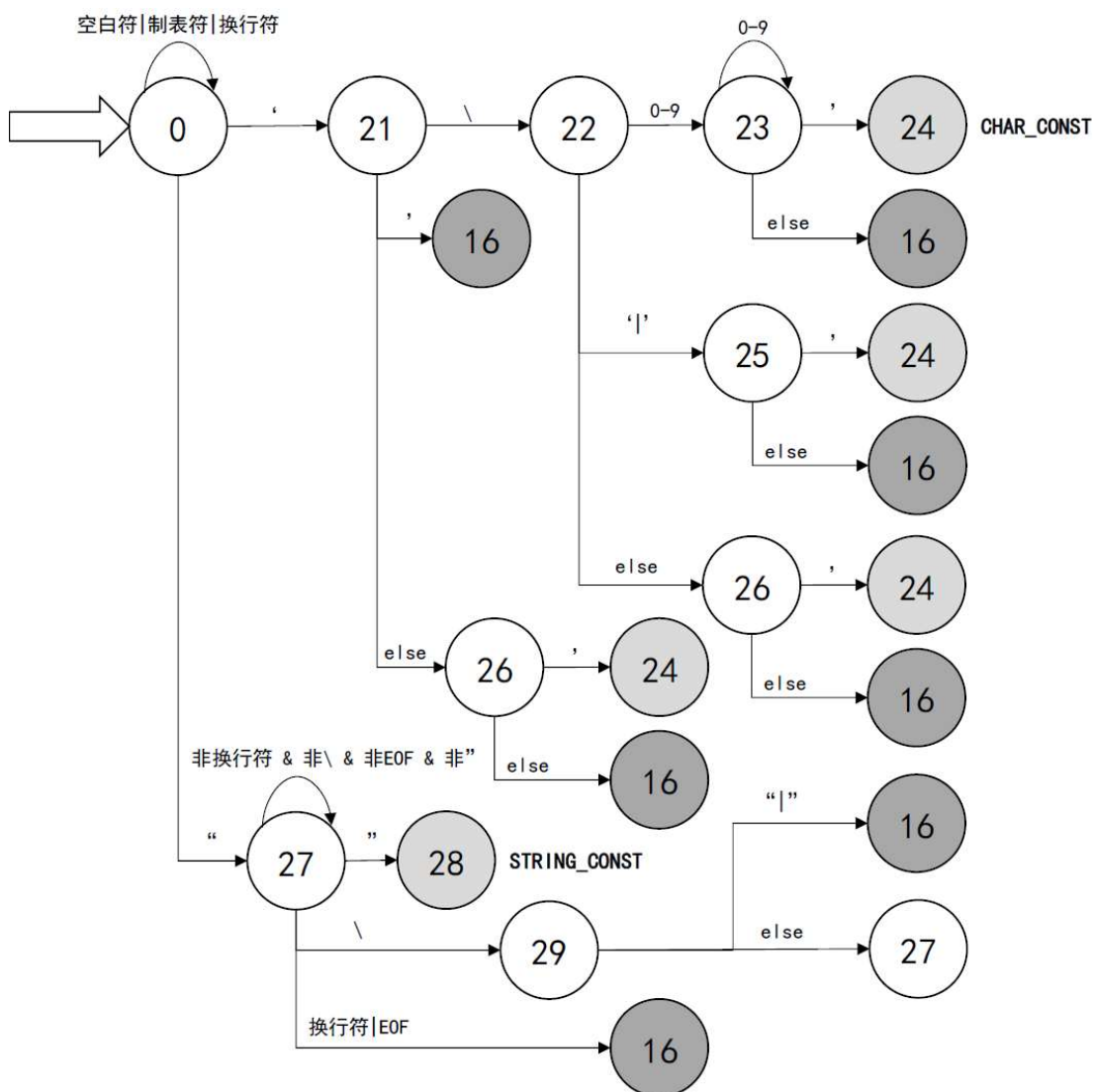


图3-7 字符型与字符串常量的过程状态转换图

处理运算符时，采用 switch-case 框架，对当前字符进行 switch 选择，若为=、+、-、&、|，则需要再读取一个字符，判断是否为连续的两个相同字符，以表示

不同含义；若为/，则需进行注释判断，分为行注释与块注释，其中行注释以“//”开始，以换行符为终止，块注释以“/*”为开始，以“*/”为终止。部分运算符与注释的过程状态转换图如图 3-8 所示。

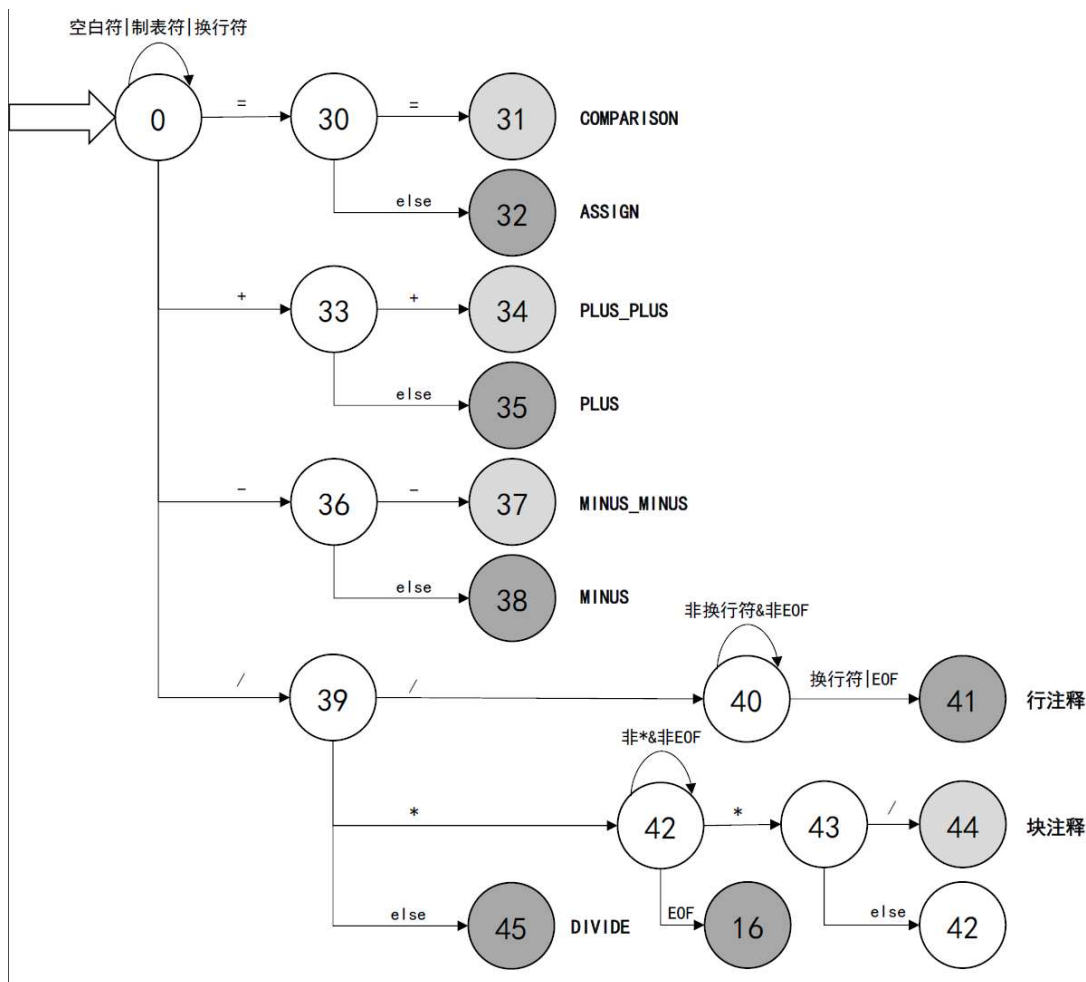


图3-8 部分运算符与注释的过程状态转换图

若当前字符为>、<、!，则需要再读取一个字符，判断是否为>=、<=、!=比较运算符。对于其他字符，可直接返回种类编码，无需进行任何判断。

若已到达文件末尾，读取 EOF，则可以直接返回 EOF，表示文件已分析完毕。若读取到定义之外的任何字符，则可以直接报错。剩下的所有情况的过程状态转换图如图 3-9 所示。

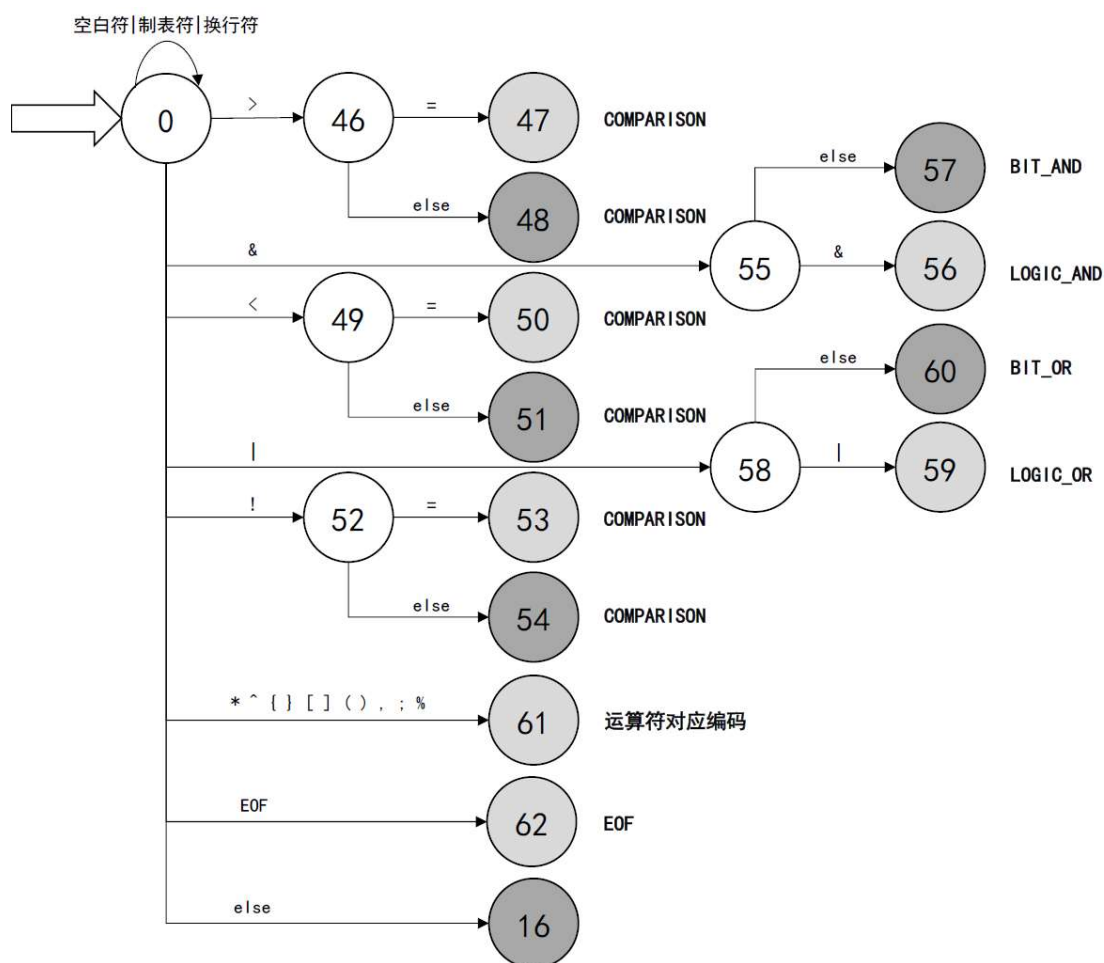


图3-9 部分运算符与异常情况的过程状态转换图

3.3.2 预编译函数算法设计

预编译函数声明为 `FILE* Pre_Process(FILE* fp)`，接收源文件指针作为参数，返回预编译完成后的文件指针。算法整体思路为：调用 `GetToken` 函数逐单词读取，若当前单词为`#`，则进入宏或头文件处理接口；若为注释，则进入注释处理接口；若为标识符，则在已存储的宏中进行查找，若查找成功则进行替换；若为其他类型，则直接放入中间文件。

进入宏或头文件处理接口后，首先判定下一单词是否为 `define` 或 `include`，若都不是，则报错。若为 `define`，则调用 `GetToken` 读取两个单词，若前者为标识符、后者为常量，且二者行数一致，则存储至 `Macro` 结构数组中，否则报错。若为 `include`，则调用 `GetToken` 读取下一单词，若为字符串常量且与 `include` 行数一致，或为左尖括号，再读取两个单词，分别为字符串与右尖括号，则格式正

确，否则报错。预编译算法流程图如图 3-10 所示。

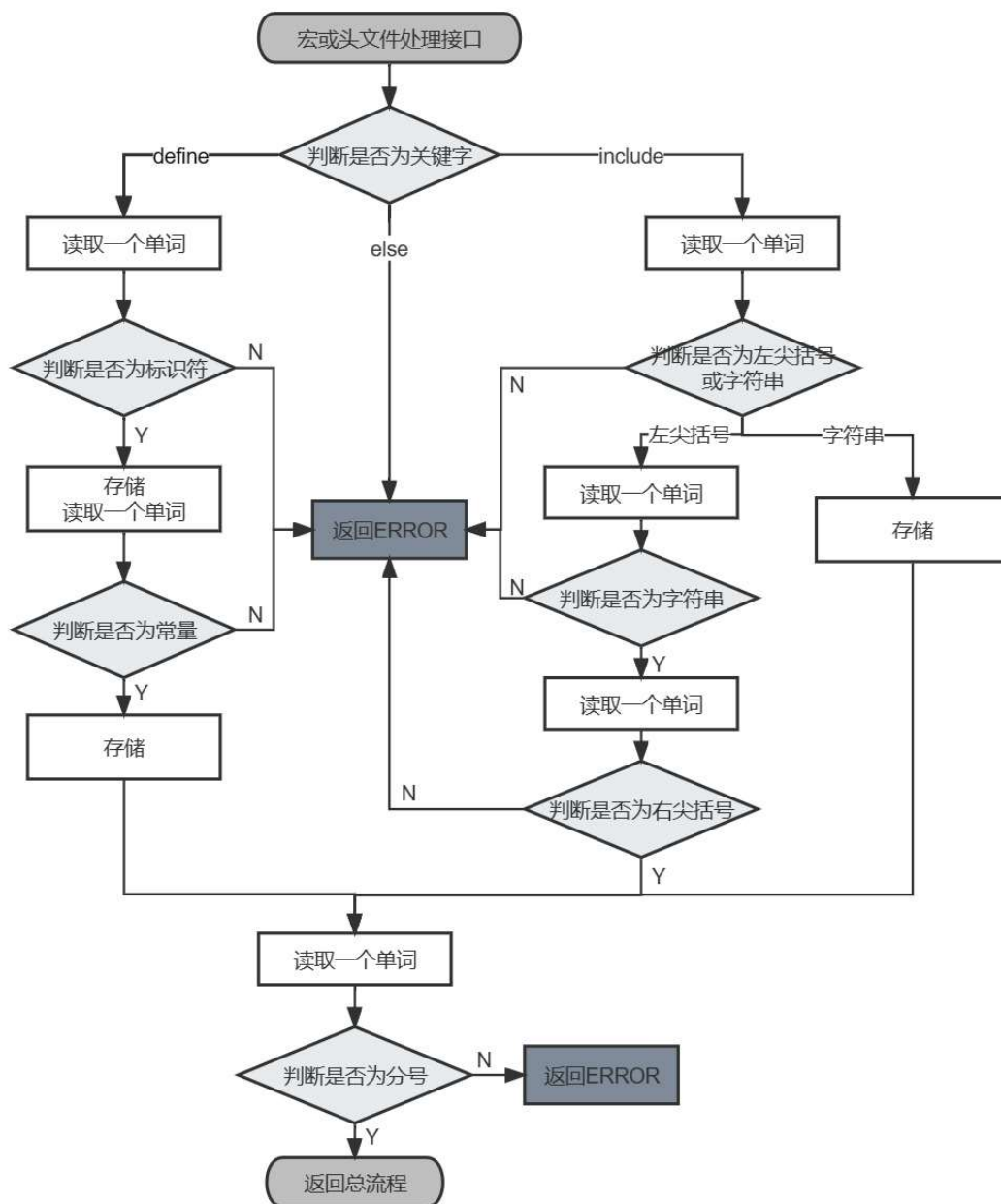


图3-10 预编译算法流程图

3.3.3 语法树生成算法设计

语法树生成算法主要思想为递归下降子程序法，每个语法单位对应一个子程序，每次根据识别出的前几个单词明确对应语法单位，调用相应子程序进行语法结构分析。由于语法树生成后需要进行文件格式化，需得到每个成分的缩进值，因此在语法树生成的同时还要生成一个可以同时存储行数与缩进值的结构数组。

首先需按照 BNF 范式定义 C 语言语法规则。对于<程序>函数 program，语法规则为 $\langle \text{程序} \rangle ::= \langle \text{外部定义序列} \rangle$ 。program 接收母树 T 作为参数，读取一个单词，调用 ExtDefList 函数分析外部定义序列，生成外部定义序列子树 C。若生成成功，对 T 进行初始化，数据域置为“程序”，缩进量置 0，首孩子指针域置空（后续函数对传递进去的母树的初始化操作均类似），最后将子树 C 作为 T 的第 1 棵子树插入。

对于<外部定义序列>函数 ExtDefList，语法规则为 $\langle \text{外部定义序列} \rangle ::= \langle \text{外部定义} \rangle \langle \text{外部定义序列} \rangle | \langle \text{外部定义} \rangle$ 。首先对母树 T 进行初始化，再对当前单词调用 ExtDef 函数分析外部定义，生成外部定义子树 C，将 C 插入 T。根据语法规则，对当前单词递归调用 ExtDefList 函数分析外部定义序列，生成外部定义序列子树 D，若生成成功，将 D 插入 T。需要注意的是，需要对外部定义序列分析结束作特判，分析结束既不属于成功，也不属于出错，若不进行特判，无法结束递归。若当前单词为 EOF，则文件已读取完毕，分析理应结束，返回 INFEASIBLE 即可。外部定义序列的语法树结构如图 3-11 所示。

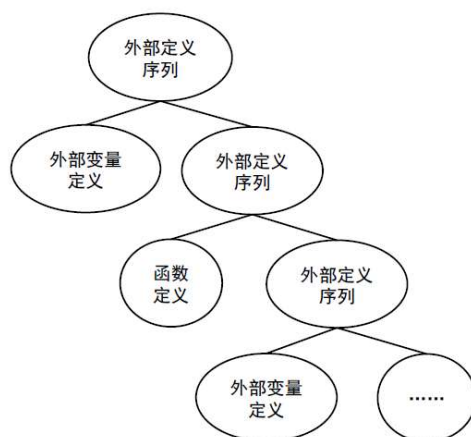


图3-11 外部定义序列的语法树结构

对于<外部定义>函数 ExtDef，语法规则为 $\langle \text{外部定义} \rangle ::= \langle \text{外部变量定义} \rangle | \langle \text{函数定义} \rangle$ 。当前单词若为类型关键字，则进行记录，并读取下一单词，否则返回 ERROR；下一单词若为标识符，则进行记录，并读取下一单词，否则返回 ERROR；下一单词若为左小括号，则调用 funcDef 函数分析函数定义，否则调用 ExtValDef 函数分析外部变量定义。

对于<外部变量定义>函数 ExtVarDef, 语法规则为 <外部变量定义>::=<类型说明符><变量序列>。首先初始化母树 T, 定义类型子树 C、变量名子树 D, 将已记录的类型与变量名复制进 C、D 中, 并插入 T。一行定义可能定义了多个变量, 因此对当前单词调用 ExtVarList 分析外部变量序列, 生成子树 E, 并插入变量名子树 D。外部变量定义的语法树结构如图 3-12 所示。

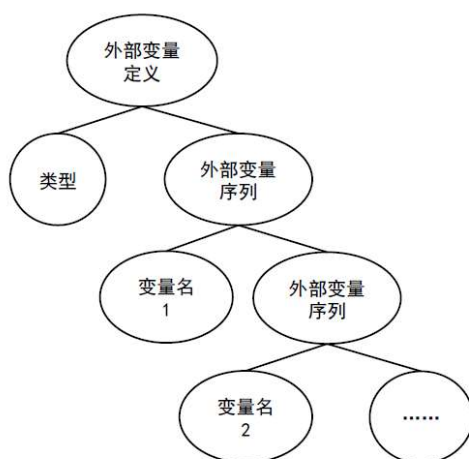


图3-12 外部变量定义的语法树结构

对于<外部变量序列>函数 ExtVarList, 语法规则为 <外部变量序列>::=<变量><变量序列> | <变量>。若当前单词为左中括号, 则进行数组判断, 先读取下一单词, 若为整型常量, 则与原变量名记录值进行拼接, 再读取下一单词, 需为右中括号, 否则返回 ERROR。若当前单词不为左中括号, 则先生成子树 C 并插入 T, 再读取下一单词, 若为 “=”, 则再生成初始化子树 E, 存储下一单词作为初始化值, 并插入 C。最后读取下一单词, 若为 “;”, 说明变量定义结束, 返回 OK; 若为 “,”, 说明后续仍有变量定义, 应递归调用 ExtVarList 函数继续分析外部变量序列。

需要注意, 形式参数定义与序列、局部变量定义与序列的算法思路和外部变量定义与序列大体一致, 在此不作赘述。

对于<函数定义>函数 funcDef, 语法规则为 <函数定义>::=<类型说明符><函数名>(<形式参数序列>)<复合语句>。首先初始化母树 T, 定义类型子树 C、函数名子树 D, 将已记录的类型与函数名复制进 C、D 中, 并插入 T。若当前单词为关键字 void, 则无需生成形参子树; 否则应调用 ParaDefList 函数生成形参

序列子树 E，插入形参子树 K，再将 K 插入 T。最后定义函数体子树 F，若当前单词为左大括号，调用 CompStat 函数分析复杂语句，生成复杂语句子树 G，G 插入 F，F 插入 T；若为分号，则视为函数声明，无需定义函数体。函数定义的语法树结构如图 3-13 所示。

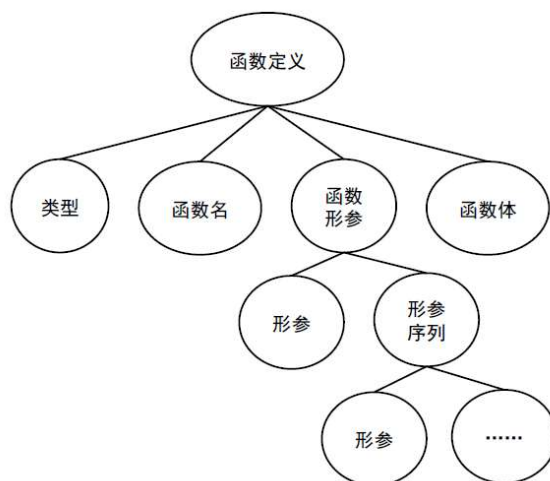


图 3-13 函数定义的语法树结构

对于<复合语句>函数 CompStat，语法规则为 $\langle \text{复合语句} \rangle ::= \{ \langle \text{局部变量定义序列} \rangle \langle \text{语句序列} \rangle \}$ 。首先初始化母树 T，需要注意的是，复合语句出现在程序块内，因此缩进值为 1，而非 0，同时记录缩进的队列需在此行进队一个缩进值自增 1 的元素。若当前单词为类型关键字，则调用 LocVarDef 函数分析局部变量定义，生成子树 C，再调用 StatList 函数分析语句序列，生成子树 D，将 C、D 插入 T；否则直接调用 StatList 函数分析语句序列，生成子树 D 并插入 T。分析结束后记录缩进的队列需在此行进队一个缩进值自减 1 的元素。最后读取单词，若不为右大括号，返回 ERROR。

对于<语句序列>函数 StatList，语法规则为 $\langle \text{语句序列} \rangle ::= \langle \text{语句} \rangle \langle \text{语句序列} \rangle \mid \langle \text{空} \rangle$ 。首先初始化母树 T，调用 Statement 函数分析语句，生成语句子树 C，若生成成功，插入 T，再递归调用自身生成第 2 棵子树；若生成失败，返回 ERROR；若未生成子树，说明分析结束，返回 INFEASIBLE。

对于<语句>函数 Statement，采用 switch-case 建构，根据当前单词的种类进行不同处理。

例如，若当前单词为关键字 if，则首先读取下两单词，若前者为左小括号且后者不为右小括号，视为条件格式正确，定义条件子树 C，对括号内条件调用 expression 函数分析表达式，生成表达式子树 D，将 D 插入 C。

定义 IF 子句子树 E，读取下一单词，若为左大括号，则调用 CompStat 函数分析复杂语句，生成程序块子树 F，将 F 插入 E；否则应为单句程序块，有同行与不同行两种情况，此时需要记录关键字 if 所在行数，以进行比较判断情况。若为同行，直接递归调用 Statement 函数分析语句，否则需在递归前后分别让记录缩进的队列在此行进队一个缩进值自增 1 与自减 1 的元素，最终生成单句程序子树 F，将 F 插入 E。

若当前单词为关键字 else，则作为 if-then-else 语句处理，对 ELSE 子句子树进行与 IF 子句子树相同的操作即可。最后将 C、E 与可选的 ELSE 子句子树插入母树 T。

if-then、if-else-then、for、while 语句的语法树结构如图 3-14 所示。

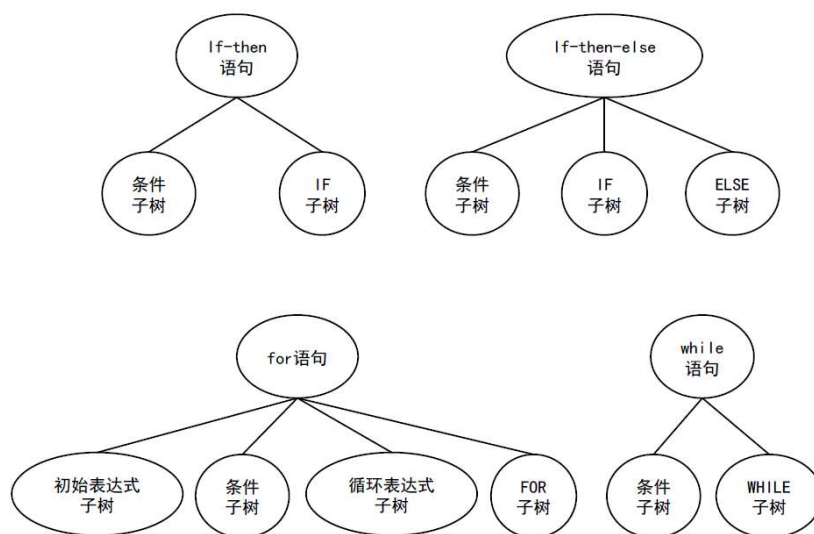


图 3-14 部分语句的语法树结构

对于<表达式>函数 expression，首先要解决运算符的优先级表示问题，考虑使用 Precede 函数，接收两个运算符作为参数，返回比较结果。在 Precede 函数中，建立一个二维数组，每个元素的两个下标分别对应一个运算符，元素值即为两个运算符的优先级关系，包括 “>” “<” “=” “?”，如表 3-3 所示，Precede 函数可直接查找下标，返回元素值。

表 3-3 运算符优先级关系

	+、-	*/、/	%	()	=	比较符	==、!=	&&		#
+、-	>	<	<	<	>	?	>	>	>	>	>
*/、/	>	>	>	<	>	?	>	>	>	>	>
%	>	<	<	<	>	?	>	>	>	>	>
(<	<	<	<	=	?	<	<	<	<	>
)	>	>	>	>	?	?	>	>	>	>	>
=	<	<	<	<	?	<	<	<	<	<	>
比较符	<	<	<	<	>	?	>	>	>	>	>
==、!=	<	<	<	<	>	?	<	>	>	>	>
&&	<	<	<	<	>	>	<	<	>	>	>
	<	<	<	<	>	>	<	<	>	>	>
#	<	<	<	<	?	<	<	<	<	<	=

对于一个表达式而言，何处终止是难以确定的，但观察可得，若表达式作为条件，位于小括号之间，则右小括号可作为终止符；若表达式作为单独语句，则分号可作为终止符。因此 `expression` 函数需获取一个终止符作为参数，以确定表达式何时终止。

`expression`函数中，定义两个栈，分别为存放运算符的栈`op`与存放操作数的栈`opn`，定义错误标志`error`。将`#`作为起止符，入栈`op`。若当前单词为常量，生成子树`node`，初始化为该常量，入栈`opn`；若为运算符，获取当前栈顶运算符，进行优先级比较。若结果为“<”，将当前运算符入栈`op`；若结果为“=”，`op`为空则`error`自增，否则出栈一个元素，读取下一个单词，以去括号；若结果为“>”，

opn出栈两个元素，op出栈一个元素，将前者作为后者的2棵子树构建树，再入栈opn；若结果为“？”，error自增。若当前单词为终止符，则将#赋给当前单词，结束循环。

循环结束后，若error非0，返回ERROR；否则将opn栈顶元素作为母树T的子树。

3.3.4 语法树遍历算法设计

采用树的先根遍历来遍历语法树，由于语法树的存储结构选择了邻接表，此处树的前根遍历与图的深度优先遍历几乎有完全一致的操作，降低了算法设计的难度。唯一的难点在于，语法树的遍历需根据所处成分进行缩进处理，思路如下。

每轮深搜开始前，将缩进量 indent 置 0。深搜过程中，indent 不断加上当前结点，即父结点的缩进值；回退时，indent 减去上一结点的缩进值：即 indent 作为根结点到当前输出的结点的最短路径上的结点的缩进值之和。在输出结点之前，打印与 indent 数量相同的制表符，便可成功解决缩进处理问题。

3.3.5 文件格式化算法设计

生成语法树的同时生成了存有与行数所对应的缩进值队列，队列中每推出一个元素，缩进格数更新为该元素的缩进值。直至达到某一行数，与当前队首元素的行数一致，则再次推出一个元素，更新缩进格数，以此类推。当队空时，退出循环，按最后更新的缩进格数进行缩进，直至读取文件尾。

4 系统实现与测试

4.1 系统实现

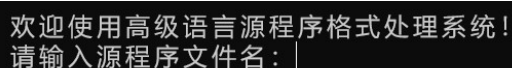
(1) 实现环境

硬件环境：Intel Core i7-12700H 2.30 GHz 内存 16G

软件环境：Visual Studio 2022

(2) 系统使用说明

系统选择菜单形式进行词法分析、语法分析、格式化程序等操作。系统启动后，需输入初始文件名，如图 4-1 所示。

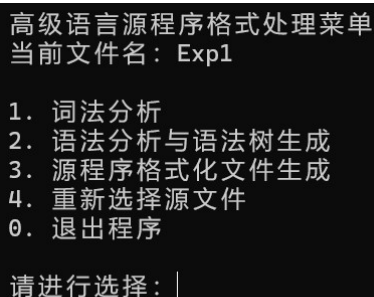


```

欢迎使用高级语言源程序格式处理系统！
请输入源程序文件名：|
    
```

图 4-1 文件输入界面

查找成功后进入选择界面，如图 4-2 所示。



```

高级语言源程序格式处理菜单
当前文件名：Exp1

1. 词法分析
2. 语法分析与语法树生成
3. 源程序格式化文件生成
4. 重新选择源文件
0. 退出程序

请进行选择：|
    
```

图 4-2 菜单界面

随后即可根据需要输入选项。词法分析时，先预编译生成中间文件，对中间文件循环调用 `GetToken` 识别各单词并进行输出，若有错误则进入错误队列，在文件输出完毕后统一输出错误行数；语法分析时，同样先预编译生成中间文件，对中间文件调用 `program`，开始构建语法树，并进行遍历；格式化时，同样先预编译、自动调用 `program`，根据生成的语法树调用 `FormatCFile` 生成格式化文件；重选文件时，通过 `goto` 返回菜单前的输入文件的步骤。

4.2 系统测试

测试模块有文件选择、词法分析、语法分析、格式化。测试用例有EXP1-11。

其中，EXP1与EXP2适用于词法分析，EXP1模拟一般程序，包含头文件、宏定义、注释。EXP2包含所有运算符，并且设置了多种常量，包括八进制、十六进制、转义字符、有前后缀，但不是合法程序，仅用于测试词法分析功能。

EXP1、EXP3、EXP4均可用于语法分析与格式化，EXP1与EXP3较短，分别包含for语句与if-then-else语句，EXP4较长，包含所有已定义的语句。

EXP5-6作为词法分析的反例，前者展示预编译错误，后者展示词法错误。

EXP7-11作为语法分析的反例，分别展示结尾符号缺失、变量定义有误、语句有误。

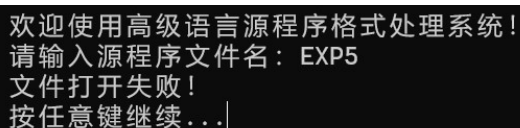
4.2.1 文件选择功能测试

(1) 正例

输入目录中存在的文件名（即EXP1-4，也可自行添加），会跳转至菜单界面，同时上方会显示当前文件名，以防误操作，如图4-2所示。

(2) 反例

输入目录中不存在的文件名，会显示错误，并重新进入该界面，如图4-3所示。



```

欢迎使用高级语言源程序格式处理系统！
请输入源程序文件名：EXP5
文件打开失败！
按任意键继续...|
    
```

图 4-3 文件输入错误测试

4.2.2 词法分析功能测试

(1) 正例

测试用例中包含头文件、宏定义、标识符、关键字、各类常量、定界符、运算符。

使用EXP1测试预编译、标识符、关键字等，EXP1代码如图4-4所示，其预编译结果如图4-5所示，词法分析的部分结果如图4-6所示。

```
1  #include<sss>
2
3  #include<>
4  int _x=0x123,y_=114514;
5  char fun(short y)
6  {
7  for(;i<1;i=i+1)
8  {
9  while(a!=b)
10 {
11 for(;;) x=5%6+7;
12 if(x==1)
13 {
14 continue; //123
15 }
16 else
17 {
18 fun(a, b);
19 int x[10];
20 if(a&&b) break;
21 else float z=0;
22 x=x+1;
23 }
24 }
25 }
26 return 0;
27 }
28 long y; //123
```

图 4-4 EXP1 代码

```
源程序预编译中...
按任意键继续...
预编译成功！词法分析进行中...
按任意键继续...
```

图 4-5 EXP1 预编译结果

行数	单词类别	单词值
4	关键字	int
4	标识符	_x
4	赋值运算符	=
4	整型常量	0x123
4	逗号	,
4	标识符	y_
4	赋值运算符	=
4	整型常量	114514
4	分号	;
5	关键字	char
5	标识符	fun
5	左小括号	(
5	关键字	short
5	标识符	y
5	右小括号)
6	左大括号	{
7	关键字	for

图 4-6 EXP1 词法分析部分结果

本例中，系统可以识别 int、char、short 等关键字，也可以将_x、y_、fun 归为标识符，还可以识别 0x123、114514 等常量与=、(、)等运算符。

使用 EXP2 进行词法分析，其代码如图 4-7 所示，分析的部分结果如 4-8 所示。观察发现，系统可以识别 u、l、ul、f、e 等后缀，也可以正确判断八进制、十六进制的前缀；可以识别普通字符与转义字符，也可以判断字符串常量。

```

1  auto, break, case, char, const, continue;
2  default, do, double, else, enum, extern, float, for;
3  goto, if, int, long, register, return, short, signed;
4  sizeof, static, struct, switch, typedef, union;
5  unsigned, void, volatile, while, _Alignas, _Alignof;
6  _Atomic, _Bool, _Complex, _Generic, _Imaginary;
7  _Noreturn, _Static_assert, _Thread_local;
8
9  a, INT, sll, _f;
10
11  4
12  4u
13  4l
14  4f
15  4.5
16  4e5
17  4.5f
18  4.5l
19  4.5e8l
20  0.36
21  .36e4
22  045
23  0x45A3
24  'a'
25  '\n'
26  "i love coding"
27  "i love course design ^w^"
28
29  +
30  ++
31  -
32  --
33  * / %
34  =
35  ==
36  > < >= <= !=

```

图4-7 EXP2部分代码

11	整型常量	4
12	无符号整型常量	4u
13	长整型常量	4l
14	单精度浮点型常量	4f
15	双精度浮点型常量	4.5
16	双精度浮点型常量	4e5
17	单精度浮点型常量	4.5f
18	长双精度浮点型常量	4.5l
19	长双精度浮点型常量	4.5e8l
20	双精度浮点型常量	0.36
21	双精度浮点型常量	.36e4
22	整型常量	045
23	整型常量	0x45A3
24	字符型常量	'a'
25	字符型常量	'\n'
26	字符串常量	"i love coding"
27	字符串常量	"i love course design ^w^"

图 4-8 EXP2 词法分析部分结果

(2) 反例

使用 EXP5 与 EXP6 分别进行词法分析，前者模拟头文件包含有误，后者模拟常量格式有误，包括违反标识符命名规则、超出十六进制数字范围、未正确

使用转义字符等。EXP5 的代码与运行结果如图 4-9 与 4-10 所示，EXP6 的代码与运行结果如图 4-11 与 4-12 所示。可以看出，系统可以判断头文件格式中的错误，也可以识别标识符命名与常量中的格式错误并进行输出。

```
1 #include<sss"
2
3 #include<>
4 int _x=0x123,y_=114514;
```

图 4-9 EXP5 代码

```
源程序预编译中...
按任意键继续...
预编译失败！请检查源程序的预处理指令部分！
按任意键继续...
```

图 4-10 EXP5 预编译结果

```
1 #include<sss>
2
3 int 1x;
4 int y = 0xG1;
5 char z = '\ '
6 z = ' '
7 float a = 123ef;
```

图 4-11 EXP6 代码

```
本文件目前错误单词位置如下：
序号      行号
1          3
2          4
3          5
4          6
5          6
6          7
```

图 4-12 EXP6 词法分析结果

4.2.3 语法分析功能测试

(1) 正例

测试用例中包含外部变量定义、函数定义、局部变量定义、复合语句、if-then 语句、if-then-else 语句、for 语句、while 语句、do-while 语句、return 语句、break 语句、continue 语句、函数调用语句、空语句、表达式。

使用EXP3测试外部变量定义、函数定义、局部变量定义、函数调用语句、if-then-else 语句、return 语句，EXP3代码如图4-13所示，语法分析结果如图4-14所示。

```

1  int i = 1, j = 2;
2  int fun(int a, float b)
3  {
4  int m = 3;
5  fun(c, d);
6  if (a > b)
7  m = a;
8  else
9  m = b;
10 return m;
11 }
12 float x, y;
    
```

图 4-13 EXP3 代码

```

外部变量定义:
  类型: int
  变量名:
    ID: i      初始化为: 1
    ID: j      初始化为: 2
函数定义:
  类型: int
  函数名: fun
  函数形参:
    类型: int, 参数名: a
    类型: float, 参数名: b
  函数体
    复合语句:
      局部变量定义:
        类型: int
        变量名:
          ID: m      初始化为: 3
      函数调用语句
        调用函数名: fun
        函数实参:
          ID: c
          ID: d
      条件语句(IF-THEN-ELSE):
        条件:
          表达式语句:
            >
            ID: b
            ID: a
          IF子句:
            表达式语句:
              =
              ID: a
              ID: m
          ELSE子句:
            表达式语句:
              =
              ID: b
              ID: m
        return语句
          表达式语句:
            ID: m
外部变量定义:
  类型: float
  变量名:
    ID: x
    ID: y
    
```

图 4-14 EXP3 语法树遍历结果

使用EXP4测试外部变量定义、函数定义、局部变量定义、函数调用语句、if-then-else语句、while语句、do-while语句、break语句、for语句、return语句，由于代码与语法树遍历结果过长，不便展示，建议自行使用系统对EXP4进行语法分析以观察结果。

(2) 反例

语法分析中的反例主要在结尾符号缺失、变量定义有误、语句有误三方面。

EXP7展示结尾符号缺失，具体为函数尾未添加右大括号，如图4-15所示。

<pre> 1 int func(int a, int b) 2 { 3 a = a + b; 4 return a; </pre>	<p>存在语法错误！错误行号：4 请进行检查！</p> <p>按任意键继续...</p>
--	--

图4-15 EXP7（符号缺失）

EXP8展示变量定义有误，具体为未遵守标识符命名规则，如图4-16所示。

<pre> 1 int 1a; 2 int func(int a, int b) 3 { 4 a = a + b; 5 return a; 6 }</pre>	<p>存在语法错误！错误行号：1 请进行检查！</p> <p>按任意键继续...</p>
---	--

图4-16 EXP8（变量定义有误）

EXP9展示函数定义有误，具体为形参定义缺失类型名，如图4-17所示。

<pre> 1 int a; 2 int func(a, int b) 3 { 4 a = a + b; 5 return a; 6 }</pre>	<p>存在语法错误！错误行号：2 请进行检查！</p> <p>按任意键继续...</p>
--	--

图4-17 EXP9（函数定义有误）

EXP10展示语句有误，具体为while语句缺少条件表达式，如图4-18所示。

<pre> 1 int func(int a) 2 { 3 while() 4 a = a - 1; 5 return a; 6 }</pre>	<p>存在语法错误！错误行号：3 请进行检查！</p> <p>按任意键继续...</p>
--	--

图4-18 EXP10（while语句有误）

EXP11展示语句有误，具体为for语句缺少右小括号，如图4-19所示。

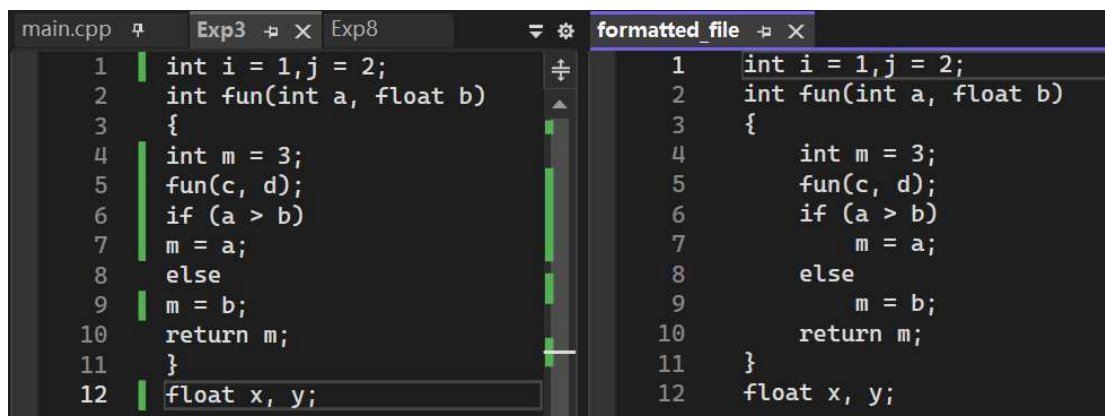
<pre> 1 int func(int a) 2 { 3 for(a = 1; a < 10; a = a + 1 4 a = a * ((2 + a) * a) 5 return a; 6 }</pre>	<p>存在语法错误！错误行号：4 请进行检查！</p> <p>按任意键继续...</p>
---	--

图4-19 EXP11（for语句有误）

可以看出，由于语法分析是基于词法分析之上构建的，词法分析所能检查出的错误，语法分析也可以判别。同时语法分析可以指出符号缺失、条件表达式缺失、定义有误等词法分析不一定能指出的问题，在已定义的语法规则内可以有效地对程序进行分析。

4.2.4 格式化功能测试

使用EXP3进行格式化，源文件与格式化文件如图4-20所示。



```
main.cpp  Exp3  Exp8  formatted_file
1  int i = 1,j = 2;
2  int fun(int a, float b)
3  {
4  int m = 3;
5  fun(c, d);
6  if (a > b)
7  m = a;
8  else
9  m = b;
10 return m;
11 }
12 float x, y;

1  int i = 1,j = 2;
2  int fun(int a, float b)
3  {
4      int m = 3;
5      fun(c, d);
6      if (a > b)
7          m = a;
8      else
9          m = b;
10     return m;
11 }
12 float x, y;
```

图4-20 EXP3源文件与格式化文件

5 总结与展望

5.1 全文总结

此次课程设计总结如下：

（1）数据结构方面，选择邻接表搭建抽象语法树，在课内知识的支持下，设计遍历、插入算法的难度有所降低，也更加熟悉树、图的操作。使用栈进行表达式操作，使用队列进行错误的记录与输出、处理格式化时的缩进值。

（2）算法方面，根据 DFA 状态转化图搭建词法分析流程框架，根据递归下降子程序的思想搭建语法分析的部件函数，按照巴克斯（BNF）范式定义的 C 语言语法规则对各部件函数进行有序调用。

（3）遍历语法树与格式化文件时，需要考虑缩进值的记录问题，因此在数据结构中设置缩进值，采用队列来有序处理格式化时的缩进值。

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）扩充知识体系与知识面，为将来的研究工作提供知识储备。

（2）坚持以实践为基础，拒绝纸上谈兵，多实操代码实现环节。

（3）除了通过书本学习，还需利用网上资源广泛学习，获取课堂知识体系之外的内容。尝试不同的语言、系统、算法，通过对比与类比开拓眼界，提升代码编写能力。

6 体会

本次课程设计我选择了问题二，虽然任务书中给出的一系列诸如“DFA 状态转换图”“递归下降子程序法”的专业词汇让人心生畏惧，但完成项目后也发现这些方法的代码实现并不难，几乎都是数据结构课程中的树、图操作，较易实现。而难点则是将词法分析与语法分析的过程抽象化，这一点在我绘制出状态转换图与语法树结构图之后也迎刃而解。这两类图形象易懂，很容易用代码实现。于是我得到启发，画图是一种很好的具体问题与代码实现间的桥梁，有时可以大大减轻思考的难度。

在实现抽象语法树时，我选用了邻接表，一是因为这是我们的课内知识，并且在数据结构实验中已经实现过，于我而言是较为熟练的；二是因为我们需要进行树的前根遍历，这在逻辑上与图的深度优先搜索是很类似的，直接选用类似图的物理存储结构也会使遍历操作更加方便。虽然我没有尝试过其他的数据结构进行对比，但就我的经验而言，在进行数据结构的设计时，一定要注意与自己的所知相结合，凭空创造数据结构是非常难的，实现的过程也会有很多困难。这也是我们需要不断学习新的知识来扩充已有的知识体系的理由。

语言的词法、语法规则，如果要深究，内容是非常多的，短短的三千行代码不可能完全展示，但只要迈出了从 0 到 1 的第一步，从 1 到 100 就会容易得多。而只有不断学习下去，才能扩充自己的眼界，为未来的学习与工作提供更好的基础。

参考文献

- [1] 王生原，董渊，张素琴，吕映芝等. 编译原理（第 3 版）. 北京：清华大学出版社. 前 4 章
- [2] 严蔚敏等.数据结构(C 语言版).北京：清华大学出版社

附录

由于篇幅限制，附录仅包含关键源代码 **main.cpp**、**Parser.cpp** 的部分内容，不包含头文件与预编译、词法分析、栈、队列、树的实现代码。

·**main.cpp**

```
#include "Lexer.h"
#include "Queue.h"
#include "Pre_Process.h"
#include "Parser.h"
#include "Formatter.h"

int main()
{
    char FileName[30]; //源文件名
    FILE *fp = NULL; //源文件指针
    int choice = 1;    //功能选择
    int kind = 0;      //当前单词种类编码
    CTree T;           //源文件抽象语法树

    CHOOSE_FILENAME:
        system("cls");
        printf("欢迎使用高级语言源程序格式处理系统！\n");
        printf("请输入源程序文件名：");
        scanf_s("%s", FileName, 30);

        while (choice)
        {
            /*若文件打开失败，显示错误原因并退至选择文件*/
            if (fopen_s(&fp, FileName, "r"))
            {
                printf("文件打开失败！\n");
                printf("按任意键继续...");
                getchar(); getchar();
                goto CHOOSE_FILENAME;
            }

            system("cls");
            printf("高级语言源程序格式处理菜单\n");
            printf("当前文件名： %s\n\n", FileName);
            printf("1. 词法分析\n");
            printf("2. 语法分析与语法树生成\n");
            printf("3. 源程序格式化文件生成\n");
            printf("4. 重新选择源文件\n");
```

```

printf("0. 退出程序\n\n");
printf("请选择: ");
scanf_s("%d", &choice);
system("cls");
switch (choice)
{
/*词法分析*/
case 1:

/*进行预编译并生成中间文件mid_file*/
printf("源程序预编译中...\n");
fp = freopen(FileName, "r", fp);
fp = Pre_Process(fp);
printf("按任意键继续...");
getchar(); getchar();
if (fp)
{
printf("预编译成功! 词法分析进行中...\n");
printf("按任意键继续...");
getchar(); getchar();
}
else
{
printf("预编译失败! 请检查源程序的预处理指令部分! \n");
printf("按任意键继续...");
getchar(); getchar();
return 0;
}
/*打开中间文件mid_file*/
fopen_s(&fp, "mid_file", "r");

/*逐词进行词法分析*/
LinkQueue Q; //链式队列存放错误行号
InitQueue(Q);
putchar('\n');
printf("  行数      单词类别      单词值\n");
while (!feof(fp))
{
kind = GetToken(fp);
/*识别为关键字*/

```

```

        if (kind >= AUTO && kind <= _THREAD_LOCAL)
            printf("  %d    关键字                %s\n", LineNum,
token_text);

        switch (kind)
        {
            /*识别为标识符*/
            case IDENT:
                printf("  %d    标识符                %s\n", LineNum,
token_text);

                break;
            /*识别为整型常量等常量*/
            case INT_CONST:
                printf("  %d    整型常量                %s\n", LineNum,
token_text);

                break;
            case UNSIGNED_CONST:
                printf("  %d    无符号整型常量                %s\n", LineNum,
token_text);

                break;
            case LONG_CONST:
                printf("  %d    长整型常量                %s\n", LineNum,
token_text);

                break;
            case UNSIGNED_LONG_CONST:
                printf("  %d    无符号长整型常量                %s\n", LineNum,
token_text);

                break;
            case FLOAT_CONST:
                printf("  %d    单精度浮点型常量                %s\n", LineNum,
token_text);

                break;
            case DOUBLE_CONST:
                printf("  %d    双精度浮点型常量                %s\n", LineNum,
token_text);

                break;
            case LONG_DOUBLE_CONST:
                printf("  %d    长双精度浮点型常量                %s\n", LineNum,
token_text);

                break;
            case CHAR_CONST:
                printf("  %d    字符型常量                %s\n", LineNum,
token_text);

                break;
            case STRING_CONST:

```

```

        printf(" %d    字符串常量           %s\n", LineNum,
token_text);

        break;
/*识别为赋值运算符等运算符*/
case ASSIGN:
    printf(" %d    赋值运算符           =\n", LineNum);
    break;
case PLUS:
    printf(" %d    加法运算符           +\n", LineNum);
    break;
case PLUS_PLUS:
    printf(" %d    自加运算符           ++\n", LineNum);
    break;
case MINUS:
    printf(" %d    减法运算符           -\n", LineNum);
    break;
case MINUS_MINUS:
    printf(" %d    自减运算符           --\n", LineNum);
    break;
case MULTIPLY:
    printf(" %d    乘法运算符           *\n", LineNum);
    break;
case DIVIDE:
    printf(" %d    除法运算符           /\n", LineNum);
    break;
case MOD:
    printf(" %d    取模运算符           %%\n", LineNum);
    break;
case COMPARISON:
    printf(" %d    关系运算符           %s\n", LineNum,
token_text);

    break;
case LOGIC_AND:
    printf(" %d    逻辑与           &&\n", LineNum);
    break;
case LOGIC_OR:
    printf(" %d    逻辑或           ||\n", LineNum);
    break;
case LOGIC_NOT:
    printf(" %d    逻辑非           !\n", LineNum);
    break;
case BIT_AND:
    printf(" %d    按位与           &\n", LineNum);
    break;

```

```

        case BIT_OR:
            printf("  %d    按位或          \n", LineNum);
            break;
        case BIT_XOR:
            printf("  %d    按位异或          ^\n", LineNum);
            break;
        /*识别为左小括号等定界符*/
        case L_PAREN:
            printf("  %d    左小括号          (\n", LineNum);
            break;
        case R_PAREN:
            printf("  %d    右小括号          )\n", LineNum);
            break;
        case L_BRACKET:
            printf("  %d    左中括号          [\n", LineNum);
            break;
        case R_BRACKET:
            printf("  %d    右中括号          ]\n", LineNum);
            break;
        case L_BRACE:
            printf("  %d    左大括号          {\n", LineNum);
            break;
        case R_BRACE:
            printf("  %d    右大括号          }\n", LineNum);
            break;
        case SEMI:
            printf("  %d    分号          ;\n", LineNum);
            break;
        case COMMA:
            printf("  %d    逗号          ,\n", LineNum);
            break;
        case ERROR_TOKEN:
            /*错误行号入队*/
            EnQueue(Q, LineNum);
            break;
    }
}
fclose(fp);
putchar('\n');

/*显示错误列表*/
if (QueueEmpty(Q))

```



```

        printf("本文件目前无错误单词！\n");
    else
    {
        int ErrorNum = 1; //错误序号
        int ErrorLine = 0; //错误行号
        printf("本文件目前错误单词位置如下：\n");
        printf("  序号          行号\n");
        while (!QueueEmpty(Q))
        {
            DeQueue(Q, ErrorLine);
            printf("  %d          %d\n", ErrorNum, ErrorLine);
            ErrorNum++;
        }
    }
    printf("\n按任意键继续...");
    LineNum = 1; //全局变量行号回退1
    getchar(); getchar();
    break;

    //.....后续功能代码省略

    }
}
return 0;
}

```

·Parser.cpp

#include "Parser.h"

```

queue<PrintArgu> PrintArguList;
int PIndentNum = 0;
int PKind = 0;
char PTokenKind[MAXLEN];
char PTokenText[MAXLEN];

```

/******

*函数名称：Program

*函数功能：分析当前语法成分是否为【程序】

*传入参数：源文件指针 fp, 树 T

*返回值：函数执行状态

*****/

Status Program(FILE* fp, CTree& T)

```

{
    /*程序=外部定义序列*/

```

```

CTree C; //外部定义序列子树
PKind = GetToken(fp);
PrintArguList.push({ PIndentNum, LineNum });
if (!ExtDefList(fp, C)) //外部定义序列分析
    return ERROR;

T.n = 1;
T.root = 0;
T.nodes[0].data = (char*)malloc((strlen("程序") + 1) * sizeof(char));
strcpy(T.nodes[0].data, "程序");
T.nodes[0].IndentNum = 0;
T.nodes[0].FirstChild = NULL;

InsertChild(T, T.root, 1, C);
return OK;
}

/*****
*函数名称: ExtDefList
*函数功能: 分析当前语法成分是否为【外部定义序列】
*传入参数: 源文件指针 fp, 树 T
*返回值: 函数执行状态
*****/
Status ExtDefList(FILE* fp, CTree& T)
{
    /*外部定义序列=外部定义+外部定义序列*/

    if (PKind == EOF)
        return INFEASIBLE;

    T.n = 1;
    T.root = 0;
    T.nodes[0].data = (char*)malloc((strlen("外部定义序列") + 1) * sizeof(char));
    strcpy(T.nodes[0].data, "外部定义序列");
    T.nodes[0].IndentNum = 0;
    T.nodes[0].FirstChild = NULL;

    CTree C; //外部定义子树
    if (!ExtDef(fp, C)) //外部定义分析
        return ERROR;
    InsertChild(T, T.root, 1, C);

    CTree D; //外部定义序列子树
    Status p;
    p = ExtDefList(fp, D); //外部定义序列分析
    if (p == OK)
    {
        InsertChild(T, T.root, 2, D);
        return OK;
    }
    else if (p == ERROR)

```

```

        return ERROR;
//序列分析结束返回 INFEASIBLE
else if (p == INFEASIBLE)
    return OK;
return OK;
}

/*****
*函数名称: ExtDef
*函数功能: 分析当前语法成分是否为【外部定义】
*传入参数: 源文件指针 fp, 树 T
*返回值: 函数执行状态
*****/
Status ExtDef(FILE* fp, CTree& T)
{
    /*外部定义=函数定义 or 外部变量定义*/

    //判断类型名是否合法, 并存储进 PTokenKind 备用
    if (PKind != VOID && PKind != INT && PKind != SHORT && PKind != LONG &&
        PKind != SIGNED && PKind != UNSIGNED && PKind != FLOAT &&
        PKind != DOUBLE && PKind != CHAR)
        return ERROR;
    strcpy(PTokenKind, token_text);

    //判断函数名 or 变量名是否合法, 并存储进 PTokenText 备用
    PKind = GetToken(fp);
    if (PKind != IDENT)
        return ERROR;
    strcpy(PTokenText, token_text);

    PKind = GetToken(fp);
    //标识符后是括号, 则进行函数分析
    if (PKind == L_PAREN)
    {
        if (!funcDef(fp, T))
            return ERROR;
    }
    //否则进行外部变量定义分析
    else
    {
        if (!ExtVarDef(fp, T))
            return ERROR;
    }
    return OK;
}

/*****

```

*函数名称: ExtVarDef

*函数功能: 分析当前语法成分是否为【外部变量定义】

*传入参数: 源文件指针 fp, 树 T

*返回值: 函数执行状态

*****/

Status ExtVarDef(FILE* fp, CTree& T)

```
{
    /*外部变量定义=类型名+变量名序列*/
    T.n = 1;
    T.root = 0;
    T.nodes[0].data = (char*)malloc((strlen("外部变量定义: ") + 1) * sizeof(char));
    strcpy(T.nodes[0].data, "外部变量定义: ");
    T.nodes[0].IndentNum = 1;
    T.nodes[0].FirstChild = NULL;

    CTree C;
    C.n = 1;
    C.root = 0;
    C.nodes[0].data = (char*)malloc((strlen("类型: ") + 1) * sizeof(char));
    strcpy(C.nodes[0].data, "类型: ");
    strcat(C.nodes[0].data, PTokenKind);
    C.nodes[0].IndentNum = 1;
    C.nodes[0].FirstChild = NULL;
    if (!InsertChild(T, T.root, 1, C))
        return ERROR;

    CTree D;
    D.n = 1;
    D.root = 0;
    D.nodes[0].data = (char*)malloc((strlen("变量名: ") + 1) * sizeof(char));
    strcpy(D.nodes[0].data, "变量名: ");
    D.nodes[0].IndentNum = 1;
    D.nodes[0].FirstChild = NULL;

    CTree E;
    if (!ExtVarList(fp, E))
        return ERROR;
    if (!InsertChild(D, D.root, 1, E))
        return ERROR;
    if (!InsertChild(T, T.root, 2, D))
        return ERROR;
    return OK;
}
```

*****/

*函数名称: ExtVarList

*函数功能: 分析当前语法成分是否为【外部变量序列】

*传入参数: 源文件指针 fp, 树 T

*返回值: 函数执行状态

```

*****/
Status ExtVarList(FILE* fp, CTree& T)
{// 初始时, PTokenText 保存了第一个变量名
    T.n = 1;
    T.root = 0;
    T.nodes[0].data = (char*)malloc((strlen("外部变量序列") + 1) * sizeof(char));
    strcpy(T.nodes[0].data, "外部变量序列");
    T.nodes[0].IndentNum = 0;
    T.nodes[0].FirstChild = NULL;

    //如果是数组, 检查后面的[]与其中的数并将其拼接进 PTokenText
    CTree C;
    if (PKind == L_BRACKET)
    {
        strcat_s(PTokenText, strlen(PTokenText) + strlen("[") + 1, "[");
        if ((PKind = GetToken(fp)) == INT_CONST)
        {
            strcat_s(PTokenText, strlen(PTokenText) + strlen(token_text) + 1, token_text);
            if ((PKind = GetToken(fp)) == R_BRACKET)
            {
                strcat_s(PTokenText, strlen(PTokenText) + strlen("]") + 1, "]");

                C.n = 1;
                C.root = 0;
                C.nodes[0].data = (char*)malloc((strlen("Array: ") + 1) * sizeof(char));
                strcpy(C.nodes[0].data, "Array: ");
                strcat(C.nodes[0].data, PTokenText);
                C.nodes[0].IndentNum = 1;
                C.nodes[0].FirstChild = NULL;

                PKind = GetToken(fp);
            }
            else
                return ERROR;
        }
        else
            return ERROR;
    }
    //如果是单独元素, 直接生成子树
    else
    {
        C.n = 1;
        C.root = 0;
        C.nodes[0].data = (char*)malloc((strlen("ID: ") + 1) * sizeof(char));
        strcpy(C.nodes[0].data, "ID: ");
        strcat(C.nodes[0].data, PTokenText);
        C.nodes[0].IndentNum = 1;
        C.nodes[0].FirstChild = NULL;

        if (PKind == ASSIGN)
        {
            CTree E;
            E.n = 1;
            E.root = 0;
            E.nodes[0].data = (char*)malloc((strlen("初始化为: ") + 1) * sizeof(char));
            strcpy(E.nodes[0].data, "初始化为: ");

```

```

        PKind = GetToken(fp);
        strcat(E.nodes[0].data, token_text);
        E.nodes[0].IndentNum = 1;
        E.nodes[0].FirstChild = NULL;

        if (!InsertChild(C, C.root, 1, E))
            return ERROR;

        PKind = GetToken(fp);
    }

    //PKind = GetToken(fp);
}

if (!InsertChild(T, T.root, 1, C))
    return ERROR;
if (PKind == SEMI)
{
    PKind = GetToken(fp);
    return OK;
}
else if (PKind == COMMA)
{
    if ((PKind = GetToken(fp)) != IDENT)
        return ERROR;
    strcpy(PTokenText, token_text);
    PKind = GetToken(fp);

    CTree D;
    if (!ExtVarList(fp, D))
        return ERROR;
    if (!InsertChild(T, T.root, 2, D))
        return ERROR;
    return OK;
}
else
    return ERROR;
return OK;
}

/*****
*函数名称: funcDef
*函数功能: 分析当前语法成分是否为【函数】
*传入参数: 源文件指针 fp, 树 T
*返回值: 函数执行状态
*****/
Status funcDef(FILE* fp, CTree& T)
{
    /*函数=类型+函数名+形参序列+函数体*/
    T.n = 1;
    T.root = 0;
    T.nodes[0].data = (char*)malloc((strlen("函数定义: ") + 1) * sizeof(char));

```

```
strcpy(T.nodes[0].data, "函数定义: ");
T.nodes[0].IndentNum = 1;
T.nodes[0].FirstChild = NULL;
```

```
Ctree C; //类型子树
C.n = 1;
C.root = 0;
C.nodes[0].data = (char*)malloc((strlen("类型: ") + 1) * sizeof(char));
strcpy(C.nodes[0].data, "类型: ");
strcat(C.nodes[0].data, PTokenKind);
C.nodes[0].IndentNum = 1;
C.nodes[0].FirstChild = NULL;
if (!InsertChild(T, T.root, 1, C))
    return ERROR;
```

```
Ctree D; //函数名子树
D.n = 1;
D.root = 0;
D.nodes[0].data = (char*)malloc((strlen("函数名: ") + 1) * sizeof(char));
strcpy(D.nodes[0].data, "函数名: ");
strcat(D.nodes[0].data, PTokenText);
D.nodes[0].IndentNum = 1;
D.nodes[0].FirstChild = NULL;
if (!InsertChild(T, T.root, 2, D))
    return ERROR;
```

```
Ctree K; //形参序列子树
K.n = 1;
K.root = 0;
K.nodes[0].data = (char*)malloc((strlen("函数形参: ") + 1) * sizeof(char));
strcpy(K.nodes[0].data, "函数形参: ");
K.nodes[0].IndentNum = 1;
K.nodes[0].FirstChild = NULL;
```

```
PKind = GetToken(fp);
if (PKind != R_PAREN && PKind != VOID && PKind != INT && PKind != LONG &&
    PKind != SIGNED && PKind != UNSIGNED && PKind != FLOAT && PKind !=
SHORT &&
    PKind != DOUBLE && PKind != CHAR)
    return ERROR;
//无参数时
if (PKind == VOID)
{
    if ((PKind = GetToken(fp)) == R_PAREN)
        return ERROR;
}
//有参数时
else if (PKind != R_PAREN)
{
    Ctree E;
    if (!ParaDefList(fp, E))
        return ERROR;
```

```

        if (!InsertChild(K, K.root, 1, E))
            return ERROR;
    }

    if (!InsertChild(T, T.root, 3, K))
        return ERROR;

CTree F; //函数体子树
F.n = 1;
F.root = 0;
F.nodes[0].data = (char*)malloc((strlen("函数体") + 1) * sizeof(char));
strcpy(F.nodes[0].data, "函数体");
F.nodes[0].IndentNum = 1;
F.nodes[0].FirstChild = NULL;

PKind = GetToken(fp);
if (PKind == L_BRACE)
{
    CTree G; //函数体内复杂语句子树
    if (!CompStat(fp, G))
        return ERROR;
    if (!InsertChild(F, F.root, 1, G))
        return ERROR;
}
else if (PKind != SEMI)
    return ERROR;
if (!InsertChild(T, T.root, 4, F))
    return ERROR;
return OK;
}

//省略部分函数代码

/*****
*函数名称: Statement
*函数功能: 分析当前语法成分是否为【语句】
*传入参数: 源文件指针 fp, 树 T
*返回值: 函数执行状态
*****/
Status Statement(FILE* fp, CTree& T)
{
    switch (PKind)
    {
        //IF -> if-then(-else)语句
        case IF:
        {
            if ((PKind = GetToken(fp)) != L_PAREN)
                return ERROR;
            if ((PKind = GetToken(fp)) == R_PAREN)
                return ERROR;

```



```
CTree C; //if-then(-else)语句_条件子树
C.n = 1;
C.root = 0;
C.nodes[0].data = (char*)malloc((strlen("条件: ") + 1) * sizeof(char));
strcpy(C.nodes[0].data, "条件: ");
C.nodes[0].IndentNum = 1;
C.nodes[0].FirstChild = NULL;
```

```
CTree D; //if-then(-else)语句_条件子树_条件表达式
if (!Expression(fp, D, R_PAREN))
    return ERROR;
if (!InsertChild(C, C.root, 1, D))
    return ERROR;
```

```
CTree E; //if-then(-else)语句_IF子句子树
E.n = 1;
E.root = 0;
E.nodes[0].data = (char*)malloc((strlen("IF子句: ") + 1) * sizeof(char));
strcpy(E.nodes[0].data, "IF子句: ");
E.nodes[0].IndentNum = 1;
E.nodes[0].FirstChild = NULL;
```

```
CTree F; //if-then(-else)语句_IF子句子树_IF子句程序块
int ifline = LineNum;
PKind = GetToken(fp);
if (PKind == L_BRACE)
{
    if (!CompStat(fp, F))
        return ERROR;
}
else
{
    if (LineNum != ifline)
    {
        PrintArguList.push({ ++PIndentNum, LineNum });
        if (!Statement(fp, F))
            return ERROR;
        PrintArguList.push({ --PIndentNum, LineNum });
    }
    else
    {
        if (!Statement(fp, F))
            return ERROR;
    }
}
if (!InsertChild(E, E.root, 1, F))
    return ERROR;
```

```

//PKind = GetToken(fp);
if (PKind == ELSE)
{
    CTree G; //if-then-else 语句_ELSE 子句子树
    G.n = 1;
    G.root = 0;
    G.nodes[0].data = (char*)malloc((strlen("ELSE 子句: ") + 1) * sizeof(char));
    strcpy(G.nodes[0].data, "ELSE 子句: ");
    G.nodes[0].IndentNum = 1;
    G.nodes[0].FirstChild = NULL;

    CTree H; //if-then-else 语句_ELSE 子句子树_ELSE 子句程序块
    int elseline = LineNum;
    PKind = GetToken(fp);
    if (PKind == L_BRACE)
    {
        if (!CompStat(fp, H))
            return ERROR;
    }
    else
    {
        if(elseline != LineNum)
        {
            PrintArguList.push({ ++PIndentNum, LineNum });
            if (!Statement(fp, H))
                return ERROR;
            PrintArguList.push({ --PIndentNum, LineNum });
        }
        else
        {
            if (!Statement(fp, H))
                return ERROR;
        }
    }
    if (!InsertChild(G, G.root, 1, H))
        return ERROR;

    //T 为 if-then-else 语句
    T.n = 1;
    T.root = 0;
    T.nodes[0].data = (char*)malloc((strlen("条件语句(IF-THEN-ELSE): ") + 1)
* sizeof(char));
    strcpy(T.nodes[0].data, "条件语句(IF-THEN-ELSE): ");
    T.nodes[0].IndentNum = 1;
    T.nodes[0].FirstChild = NULL;
    InsertChild(T, T.root, 1, C);
    InsertChild(T, T.root, 2, E);
    InsertChild(T, T.root, 3, G);
}
else
{
    //T 为 if-then 语句
    T.n = 1;

```

```

        T.root = 0;
        T.nodes[0].data = (char*)malloc((strlen("条件语句(IF-THEN): ") + 1) *
sizeof(char));
        strcpy(T.nodes[0].data, "条件语句(IF-THEN): ");
        T.nodes[0].IndentNum = 1;
        T.nodes[0].FirstChild = NULL;
        InsertChild(T, T.root, 1, C);
        InsertChild(T, T.root, 2, E);
    }
    //PKind = GetToken(fp);
    return OK;
}

```

//WHILE -> while 语句

case WHILE:

```

{
    if ((PKind = GetToken(fp)) != L_PAREN)
        return ERROR;
    if ((PKind = GetToken(fp)) == R_PAREN)
        return ERROR;

```

```

    T.n = 1;
    T.root = 0;
    T.nodes[0].data = (char*)malloc((strlen("while 语句: ") + 1) * sizeof(char));
    strcpy(T.nodes[0].data, "while 语句: ");
    T.nodes[0].IndentNum = 1;
    T.nodes[0].FirstChild = NULL;

```

```

    CTree C; //while 语句_条件子树
    if (!Expression(fp, C, R_PAREN))
        return ERROR;

```

```

    CTree D; //while 语句_执行子树
    PKind = GetToken(fp);
    if (PKind == L_BRACE)
    {
        if (!CompStat(fp, D))
            return ERROR;
    }
    else
    {
        PrintArguList.push({ ++PIndentNum, LineNum });
        if (!Statement(fp, D))
            return ERROR;
        PrintArguList.push({ --PIndentNum, LineNum });
    }

```

```

    InsertChild(T, T.root, 1, C);
    InsertChild(T, T.root, 2, D);
    return OK;
}

```

//DO -> do-while 语句

case DO:

```

{

```

```

T.n = 1;
T.root = 0;
T.nodes[0].data = (char*)malloc((strlen("do-while 语句: ") + 1) * sizeof(char));
strcpy(T.nodes[0].data, "do-while 语句: ");
T.nodes[0].IndentNum = 1;
T.nodes[0].FirstChild = NULL;

CTree C; //do-while 语句_执行子树
PKind = GetToken(fp);
if (PKind == L_BRACE)
{
    if (!CompStat(fp, C))
        return ERROR;
}
else
{
    PrintArguList.push({ ++PIndentNum, LineNum });
    if (!Statement(fp, C))
        return ERROR;
    PrintArguList.push({ --PIndentNum, LineNum });
}

CTree D; //do-while 语句_条件子树
if (PKind != WHILE)
    return ERROR;
if ((PKind = GetToken(fp)) != L_PAREN)
    return ERROR;
if ((PKind = GetToken(fp)) == R_PAREN)
    return ERROR;
if (!Expression(fp, D, R_PAREN))
    return ERROR;
if ((PKind = GetToken(fp)) != SEMI)
    return ERROR;
PKind = GetToken(fp);
InsertChild(T, T.root, 1, C);
InsertChild(T, T.root, 2, D);
return OK;
}

//FOR -> for 语句
case FOR:
{
    T.n = 1;
    T.root = 0;
    T.nodes[0].data = (char*)malloc((strlen("for 语句: ") + 1) * sizeof(char));
    strcpy(T.nodes[0].data, "for 语句: ");
    T.nodes[0].IndentNum = 1;
    T.nodes[0].FirstChild = NULL;

    CTree C; //for 语句_初始表达式子树
    C.n = 1;
    C.root = 0;
    C.nodes[0].data = (char*)malloc((strlen("初始表达式: ") + 1) * sizeof(char));
    strcpy(C.nodes[0].data, "初始表达式: ");
    C.nodes[0].IndentNum = 1;
    C.nodes[0].FirstChild = NULL;

```

```

CTree D; //for 语句_初始表达式子树_表达式子树
if ((PKind = GetToken(fp)) != L_PAREN)
    return ERROR;
PKind = GetToken(fp);
if (!Expression(fp, D, SEMI))
    return ERROR;
if (!InsertChild(C, C.root, 1, D))
    return ERROR;

CTree E; //for 语句_条件子树
E.n = 1;
E.root = 0;
E.nodes[0].data = (char*)malloc((strlen("条件: ") + 1) * sizeof(char));
strcpy(E.nodes[0].data, "条件: ");
E.nodes[0].IndentNum = 1;
E.nodes[0].FirstChild = NULL;

CTree F; //for 语句_条件子树_表达式子树
PKind = GetToken(fp);
if (!Expression(fp, F, SEMI))
    return ERROR;
if (!InsertChild(E, E.root, 1, F))
    return ERROR;

CTree G; //for 语句_更新表达式子树
G.n = 1;
G.root = 0;
G.nodes[0].data = (char*)malloc((strlen("更新表达式: ") + 1) * sizeof(char));
strcpy(G.nodes[0].data, "更新表达式: ");
G.nodes[0].IndentNum = 1;
G.nodes[0].FirstChild = NULL;

CTree H; //for 语句_更新表达式子树_表达式子树
PKind = GetToken(fp);
if (!Expression(fp, H, R_PAREN))
    return ERROR;
if (!InsertChild(G, G.root, 1, H))
    return ERROR;

CTree I; //for 语句_执行子树
PKind = GetToken(fp);
if (PKind == L_BRACE)
{
    if (!CompStat(fp, I))
        return ERROR;
}
else
{
    PrintArguList.push({ ++PIndentNum, LineNum });
    if (!Statement(fp, I))
        return ERROR;
    PrintArguList.push({ --PIndentNum, LineNum });
}
InsertChild(T, T.root, 1, C);
InsertChild(T, T.root, 2, E);

```

```

        InsertChild(T, T.root, 3, G);
        InsertChild(T, T.root, 4, I);
        return OK;
    }

//CONTINUE -> continue 语句
case CONTINUE:
{
    T.n = 1;
    T.root = 0;
    T.nodes[0].data = (char*)malloc((strlen("continue 语句") + 1) * sizeof(char));
    strcpy(T.nodes[0].data, "continue 语句");
    T.nodes[0].IndentNum = 1;
    T.nodes[0].FirstChild = NULL;

    if ((PKind = GetToken(fp)) != SEMI)
        return ERROR;
    PKind = GetToken(fp);
    return OK;
}

//BREAK -> break 语句
case BREAK:
{
    T.n = 1;
    T.root = 0;
    T.nodes[0].data = (char*)malloc((strlen("break 语句") + 1) * sizeof(char));
    strcpy(T.nodes[0].data, "break 语句");
    T.nodes[0].IndentNum = 1;
    T.nodes[0].FirstChild = NULL;

    if ((PKind = GetToken(fp)) != SEMI)
        return ERROR;
    PKind = GetToken(fp);
    return OK;
}

//RETURN -> return 语句
case RETURN:
{
    T.n = 1;
    T.root = 0;
    T.nodes[0].data = (char*)malloc((strlen("return 语句") + 1) * sizeof(char));
    strcpy(T.nodes[0].data, "return 语句");
    T.nodes[0].IndentNum = 1;
    T.nodes[0].FirstChild = NULL;

    CTree C; //return 语句_返回值子树
    if ((PKind = GetToken(fp)) != SEMI)
    {
        if (!Expression(fp, C, SEMI))
            return ERROR;
        InsertChild(T, T.root, 1, C);
    }
    PKind = GetToken(fp);
    return OK;
}

```

```

    }

    //左小括号 -> 表达式
    case L_PAREN:
    {
        if (!Expression(fp, T, R_PAREN))
            return ERROR;
        PKind = GetToken(fp);
        return OK;
    }

    //左大括号 -> 复杂语句
    case L_BRACE:
    {
        if (!CompStat(fp, T))
            return ERROR;
        return OK;
    }

    case IDENT:
    {
        strcpy(PTokenText, token_text);
        if ((PKind = GetToken(fp)) == L_PAREN)
        {
            T.n = 1;
            T.root = 0;
            T.nodes[0].data = (char*)malloc((strlen(" 函数调用语句 ") + 1) *
sizeof(char));

            strcpy(T.nodes[0].data, "函数调用语句");
            T.nodes[0].IndentNum = 1;
            T.nodes[0].FirstChild = NULL;

            CTree C; //for 语句_初始表达式子树
            C.n = 1;
            C.root = 0;
            C.nodes[0].data = (char*)malloc((strlen("调用函数名: ") + 1) * sizeof(char));
            strcpy(C.nodes[0].data, "调用函数名: ");
            strcat(C.nodes[0].data, PTokenText);
            C.nodes[0].IndentNum = 1;
            C.nodes[0].FirstChild = NULL;

            CTree D; //形参序列子树
            D.n = 1;
            D.root = 0;
            D.nodes[0].data = (char*)malloc((strlen("函数实参: ") + 1) * sizeof(char));
            strcpy(D.nodes[0].data, "函数实参: ");
            D.nodes[0].IndentNum = 1;
            D.nodes[0].FirstChild = NULL;

            int i = 1;
            while ((PKind = GetToken(fp)) != R_PAREN)
            {
                if (PKind == IDENT)
                {
                    CTree E; //形参序列子树
                    E.n = 1;

```

```

        E.root = 0;
        E.nodes[0].data = (char*)malloc((strlen("ID: ") + 1) * sizeof(char));
        strcpy(E.nodes[0].data, "ID: ");
        strcat(E.nodes[0].data, token_text);
        E.nodes[0].IndentNum = 1;
        E.nodes[0].FirstChild = NULL;

        if (!InsertChild(D, D.root, i++, E))
            return ERROR;
    }
}
if (!InsertChild(T, T.root, 1, C))
    return ERROR;
if (!InsertChild(T, T.root, 2, D))
    return ERROR;
if ((PKind = GetToken(fp)) == SEMI)
{
    PKind = GetToken(fp);
    return OK;
}
}
else if(PKind == ASSIGN)
{
    ungetc('=', fp);
    strcpy(token_text, PTokenText);
    PKind = IDENT;
}
else if (PKind == PLUS)
{
    ungetc('+', fp);
    strcpy(token_text, PTokenText);
    PKind = IDENT;
}
else if (PKind == MINUS)
{
    ungetc('-', fp);
    strcpy(token_text, PTokenText);
    PKind = IDENT;
}
}
case INT_CONST:
case UNSIGNED_CONST:
case LONG_CONST:
case UNSIGNED_LONG_CONST:
case FLOAT_CONST:
case DOUBLE_CONST:
case LONG_DOUBLE_CONST:
case CHAR_CONST:
case STRING_CONST:
{
    if (!Expression(fp, T, SEMI))
        return ERROR;
    PKind = GetToken(fp);
    return OK;
}
case INT:

```



```

case SIGNED:
case UNSIGNED:
case LONG:
case SHORT:
case FLOAT:
case DOUBLE:
case CHAR:
{
    strcpy(PTokenKind, token_text);
    if ((PKind = GetToken(fp)) != IDENT)
        return ERROR;
    strcpy(PTokenText, token_text);
    PKind = GetToken(fp);
    if (!LocVarDef(fp, T))
        return ERROR;
    //PKind = GetToken(fp);
    return OK;
}

case R_BRACE:
    //PrintArguList.push({ --PIndentNum, LineNum });
    return INFEASIBLE;

case SEMI:
{
    T.n = 1;
    T.root = 0;
    T.nodes[0].data = (char*)malloc((strlen("空语句") + 1) * sizeof(char));
    strcpy(T.nodes[0].data, "空语句");
    T.nodes[0].IndentNum = 1;
    T.nodes[0].FirstChild = NULL;

    PKind = GetToken(fp);
    return OK;
}
default:
    return ERROR;
}
}

```

/******

*函数名称: Expression

*函数功能: 分析当前语法成分是否为【表达式】

*传入参数: 源文件指针 fp, 树 T, 结束符 endsym

*返回值: 函数执行状态

*****/

Status Expression(FILE* fp, CTree& T, int endsym)

```

{
    T.n = 1;
    T.root = 0;
    T.nodes[0].data = (char*)malloc((strlen("表达式语句: ") + 1) * sizeof(char));
    strcpy(T.nodes[0].data, "表达式语句: ");

```

```

T.nodes[0].IndentNum = 1;
T.nodes[0].FirstChild = NULL;

CTree* Node = (CTree*)malloc(sizeof(CTree)); //储存当前栈顶元素
Node->n = 1;
Node->root = 0;
Node->nodes[0].data = (char*)malloc((strlen("#") + 1) * sizeof(char));
strcpy(Node->nodes[0].data, "#");
Node->nodes[0].FirstChild = NULL;

SqStack op;
SqStack opn;
InitStack(op);
InitStack(opn);
Push(op, Node);
int error = 0;

while ((PKind != BEGIN_END || strcmp(Node->nodes[0].data, "#")) && !error)
{
    if (PKind == IDENT || PKind == INT_CONST || PKind == UNSIGNED_CONST ||
        PKind == LONG_CONST || PKind == UNSIGNED_LONG_CONST || PKind ==
FLOAT_CONST ||
        PKind == DOUBLE_CONST || PKind == LONG_DOUBLE_CONST || PKind ==
CHAR_CONST)
    {
        Node = (CTree*)malloc(sizeof(CTree)); //储存当前栈顶元素
        Node->n = 1;
        Node->root = 0;
        Node->nodes[0].data = (char*)malloc((strlen("ID: ") + strlen(token_text) + 1) *
sizeof(char));
        strcpy(Node->nodes[0].data, "ID: ");
        strcat(Node->nodes[0].data, token_text);
        Node->nodes[0].IndentNum = 1;
        Node->nodes[0].FirstChild = NULL;

        Push(opn, Node);
        PKind = GetToken(fp);
    }
    else if (PKind == ASSIGN || PKind == PLUS || PKind == MINUS || PKind ==
MULTIPLY ||
        PKind == DIVIDE || PKind == MOD || PKind == COMPARISON || PKind ==
LOGIC_AND ||
        PKind == LOGIC_OR || PKind == L_PAREN || PKind == R_PAREN || PKind ==
BEGIN_END)
    {
        if (PKind == BEGIN_END)
            strcpy(token_text, "#");
        Node = (CTree*)malloc(sizeof(CTree)); //储存当前栈顶元素
        GetTop(op, Node);

        switch (precede(Node->nodes[0].data, token_text))
        {
            case '<':
                Node = (CTree*)malloc(sizeof(CTree)); //储存当前栈顶元素
                Node->n = 1;
                Node->root = 0;
                Node->nodes[0].data = (char*)malloc((strlen(token_text) + 1) * sizeof(char));

```

```

        strcpy(Node->nodelist[0].data, token_text);
        Node->nodelist[0].IndentNum = 1;
        Node->nodelist[0].FirstChild = NULL;

        Push(op, Node);
        PKind = GetToken(fp);
        break;
    case '=':
        if (!Pop(op, Node))
            error++;
        PKind = GetToken(fp);
        break;
    case '>':
    {
        CTree* Num1 = (CTree*)malloc(sizeof(CTree));
        CTree* Num2 = (CTree*)malloc(sizeof(CTree));
        Node = (CTree*)malloc(sizeof(CTree));
        if (!Pop(opn, Num1))
            error++;
        if (!Pop(opn, Num2))
            error++;
        if (!Pop(op, Node))
            error++;
        InsertChild(*Node, Node->root, 1, *Num1);
        InsertChild(*Node, Node->root, 2, *Num2);
        Push(opn, Node);
        break;
    }
    default:
        if (PKind == endsym)
            PKind = BEGIN_END;
        else
            error++;
        break;
    }
}
else if (PKind == endsym)
    PKind = BEGIN_END;
else
    error = 1;
GetTop(op, Node);
}
if (error)
    return ERROR;
GetTop(opn, Node);
InsertChild(T, T.root, 1, *Node);

return OK;
}

```

/******

*函数名称: Precede

*函数功能: 分析两个运算符的优先顺序

*传入参数：字符串 a, 字符串 b

*返回值：表示优先顺序的字符('>', '<', '=', '?')

*****/

char precede(char* a, char* b)

```
{
    int i, j;
    char OpPrecede[MAX_OPERATOR_NUM][MAX_OPERATOR_NUM] =
    {
        //      '+' or '-'      '*' or '/'      '%'      '('      ')'      '='
        '<' or '>' or '<=' or '>='      '== or !='      '&&'      '||'      '#'
        /* '+' or '-' */      '>',      '<',      '>',      '<',      '>',      '?',
        /* '*' or '/' */      '>',      '<',      '>',      '<',      '>',      '?',
        /* '%' */      '>',      '<',      '>',      '<',      '>',      '?',
        /* '(' */      '<',      '<',      '<',      '<',      '<',      '=',
        /* ')' */      '>',      '>',      '>',      '>',      '>',      '?',
        /* '=' */      '<',      '<',      '<',      '<',      '<',      '?',
        /* '<' or '>' */      '<',      '>',      '<',      '>',      '<',      '?',
        /* '<=' or '>=' */      '<',      '>',      '<',      '>',      '<',      '?',
        /* '==' or '!=' */      '<',      '>',      '<',      '>',      '<',      '?',
        /* '&&' */      '<',      '>',      '<',      '>',      '<',      '?',
        /* '||' */      '<',      '>',      '<',      '>',      '<',      '?',
        /* '#' */      '<',      '>',      '<',      '>',      '<',      '?',
    };
};
```

switch (a[0])

```
{
    case '+':
    case '-':
        i = 0; break;
    case '*':
    case '/':
        i = 1; break;
    case '%':
        i = 2; break;
    case '(':
        i = 3; break;
    case ')':
        i = 4; break;
    case '=':
        if (a[1] == '=')
            i = 7;
        else
            i = 5;
        break;
    case '<':
    case '>':
        i = 6; break;
}
```

```

    case '!':
        i = 7; break;
    case '&':
        if (a[1] == '&')
            i = 8;
        else
            return '?';
        break;
    case '|':
        if (a[1] == '|')
            i = 9;
        else
            return '?';
        break;
    case '#':
        i = 10; break;
    default:
        return '?';
}

```

```

switch (b[0])
{
    case '+':
    case '-':
        j = 0; break;
    case '*':
    case '/':
        j = 1; break;
    case '%':
        j = 2; break;
    case '(':
        j = 3; break;
    case ')':
        j = 4; break;
    case '=':
        if (b[1] == '=')
            j = 7;
        else
            j = 5;
        break;
    case '<':
    case '>':
        j = 6; break;
    case '!':
        j = 7; break;
    case '&':
        if (b[1] == '&')
            j = 8;
        else
            return '?';
        break;
    case '|':
        if (b[1] == '|')
            j = 9;
        else
            return '?';
        break;
}

```

```
case '#':  
    j = 10; break;  
default:  
    return '?';  
}  
return OpPrecede[i][j];  
}
```