

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：

学 号：

姓 名：

指导教师：

报告日期：

计算机科学与技术学院

任 务 书

1. 设计内容

在计算机科学中，抽象语法树（Abstract Syntax Tree 或者缩写为 AST），是将源代码的语法结构的用树的形式表示，树上的每个结点都表示源程序代码中的一种语法成分。抽象语法树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。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。

2. 设计要求

（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) 至 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。

3. 参考文献

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

目 录

任务书	I
1 引言	2
1.1 课题背景与意义	2
1.2 国内外研究现状	2
1.3 课程设计的主要研究工作	2
2 系统需求分析与总体设计	3
2.1 系统需求分析	3
2.2 系统总体设计	3
3 系统详细设计	5
3.1 有关常量、全局变量、数据类型以及数据结构的定义	5
3.2 主要算法设计	6
4 系统实现与测试	16
4.1 系统实现	16
4.2 系统测试	17
5 总结与展望	24
5.1 全文总结	24
5.2 工作展望	24
6 体会	25
参考文献	26
附录	27

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,在程序分析等诸多领域有广泛的应用。利用抽象语法树可以方便地实现多种源程序处理工具,比如源程序浏览器、智能编辑器、语言翻译器等。

通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,对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。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,但随着程序语言的设计的不断变化,程序规模的不断增大,处理工具的效率问题一直是核心研究项目之一。

近十年来,国外关于高级语言格式处理工具的设计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。对于国内而言,现阶段对于高级语言格式处理主要着眼于特定处理器的特定部分。

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

系统包含四个主功能：文件选择、词法分析、语法分析以及缩进编排。

（1）文件选择功能：系统的输入功能，通过外部输入设备输入待处理 C 语言源程序文件名，从而实现对指定文件的处理。并可以更换处理文件，方便多文件处理。

（2）词法分析功能：包含两个过程：预编译过程和词法分析过程。考虑到源程序内头文件、宏定义以及注释对词法分析带来困难，故将词法分析模块分解为两个子问题，分别通过函数调用实现。在预编译过程中，调用 `pre_process` 函数，主要准确识别头文件、宏定义以及注释，如头文件、宏定义以及注释出现编写错误，则输出预编译失败，并退出程序；如正确则生成中间文件 `C_mid_file`，存放除去头文件、宏定义以及注释外的源程序。而词法分析过程中，调用 `gettoken` 函数识别中间文件中源程序单词，并输出单词类型、对应单词值和错误对应行数。

（3）语法分析功能：包含三个过程：预编译过程、生成抽象语法树和遍历语法树过程。同样调用 `pre_process` 函数处理头文件、宏定义及注释后生成 `C_mid_file` 文件。若预处理成功，则通过调用 `program` 函数生成抽象语法树，如果出现语法错误，则退出函数，输出错误所在行；生成抽象语法树成功，则调用 `TraverseTree` 遍历树函数，显示抽象语法树。

(4) 缩进编排功能：包含三个过程：预编译过程、生成抽象语法树和缩进编排过程。在经过预编译过程和生成抽象语法树后，调用 PrintFile 函数，生成 C_print_file 文件。

由于预编译操作分离与词法分析，故四个主功能可独立完成所选操作。除了彼此独立性外，阶段查错功能也为 C 语言源程序词法语法错误查找以及程序调试带来便利。为使界面更简洁明了，每个操作完成后使用 system("cls") 清楚界面内容。

系统模块流程图如图 2-1 所示。

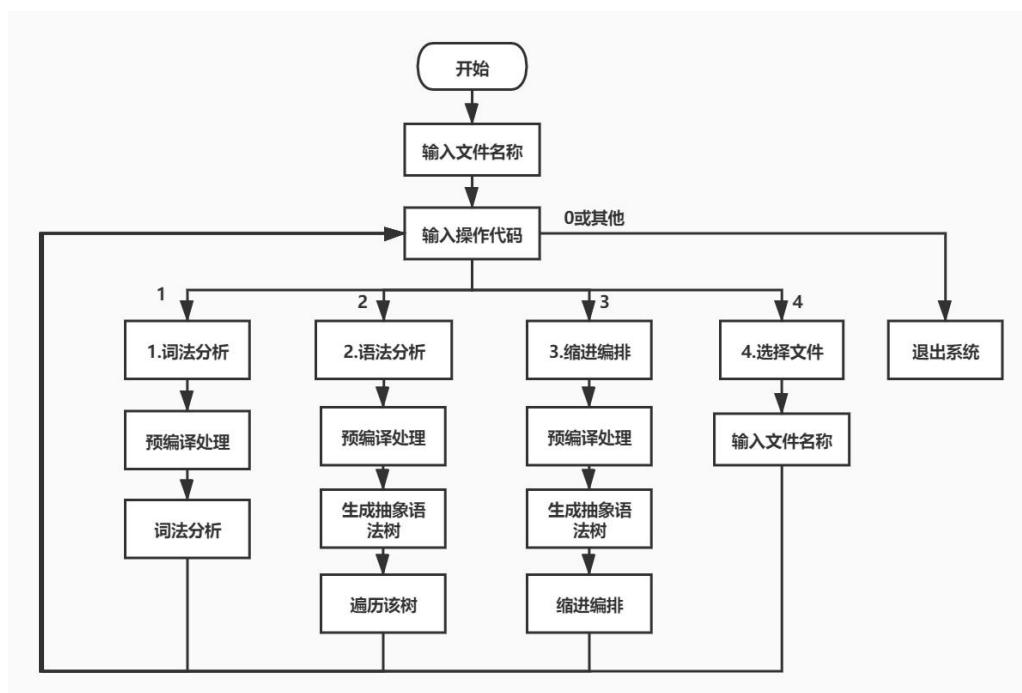


图 2-1 系统模块流程图

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：一部分常量用来指定函数的返回值，用于判断函数执行的情况。定义的常量有：OK(1)，ERROR(0)，INFEASIBLE(-1)。

(2) 全局变量定义：token_text 用于保存单词的自身值；line_num 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；kind 和 tokentext 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 SElemType 为树指针类型 CTree*。

(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。对于预编译处理，分别构建用于存储宏定义与头文件的数据结构 define_data 与 include_data。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，CTree 中包含树的结点个数、根结点位置以及头结点数组；CTNode 中包含存储头结点内容的数据、缩进量的 indent 以及指向单链表的指针；CNode 中包含存储孩子结点数据的数据域和指向下一孩子结点的指针域。

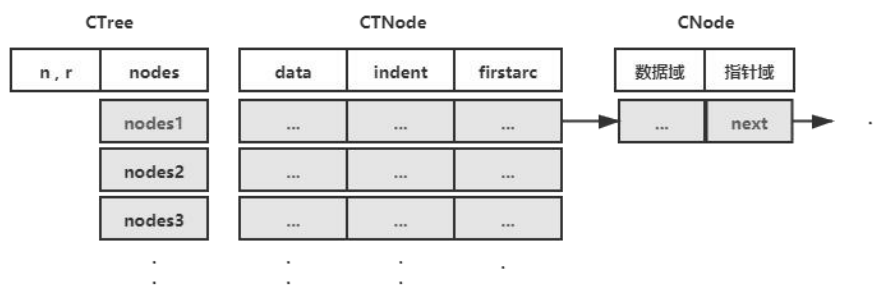


图 3-1 抽象语法树的存储结构图

3.2 主要算法设计

3.2.1 词法分析函数算法设计

词法分析函数声明为 `int gettoken(FILE* fp);`。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 token_kind{...};`。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。

通过构建过程状态转换图（如图 3-2 所示），来处理各类单词。每次从状态 0 开始，从源程序文件中读取一个字符，可以到达下一个状态，当到达环形的状态时，表示成功的读取到了一个单词，返回单词的编码，单词自身值保存在全局变量 `token_text` 中。如果结束状态上标有星号，则表示多读取字符，应回退读取字符至文件的输入缓存区。每调用一次得到一个单词的种类码和自身值。

另外，还需跳过空白符，当读取回车符时，行数 `line_num` 增一。

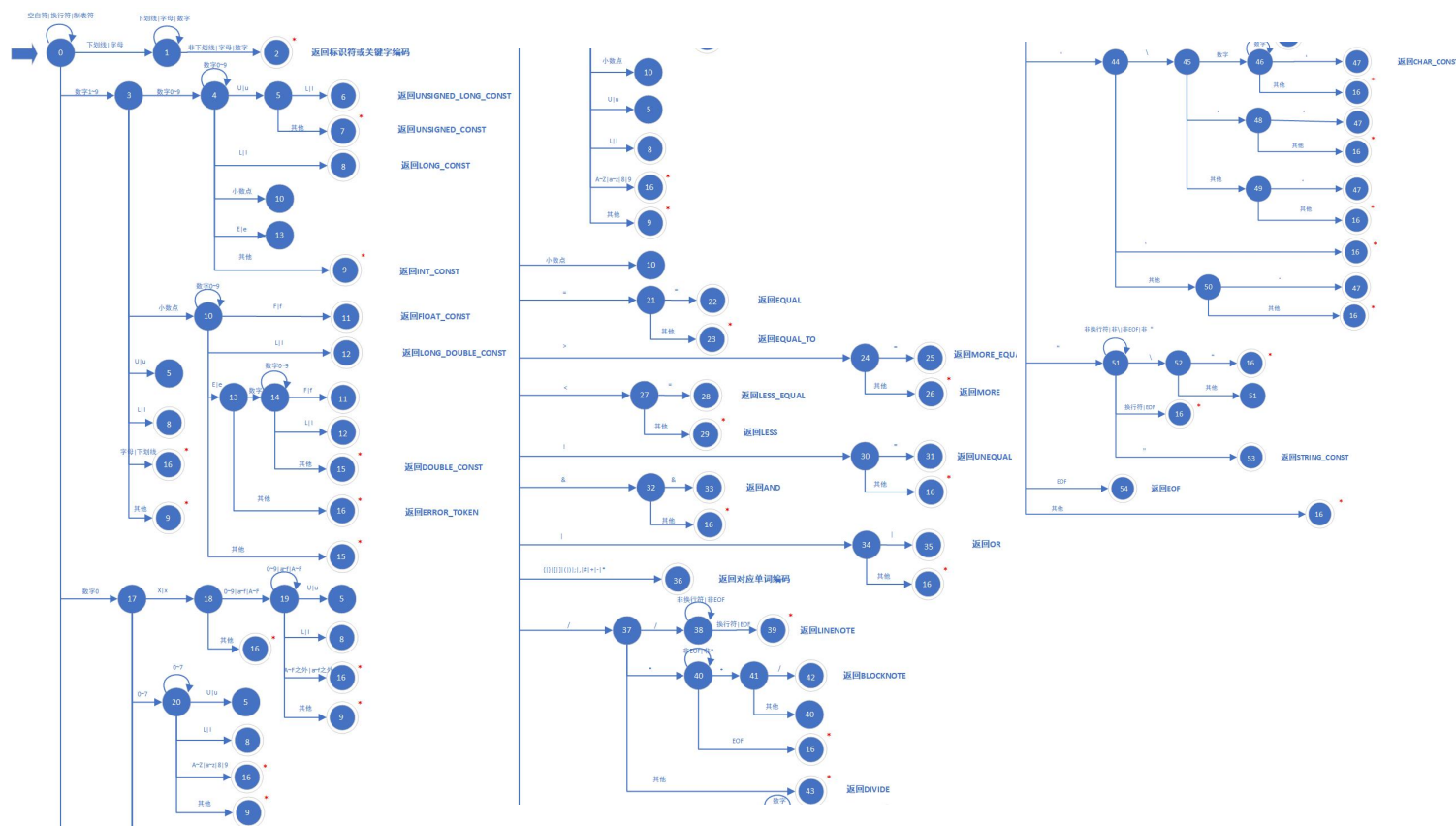


图 3-2 过程状态转换图

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。（如图 3-2-1）

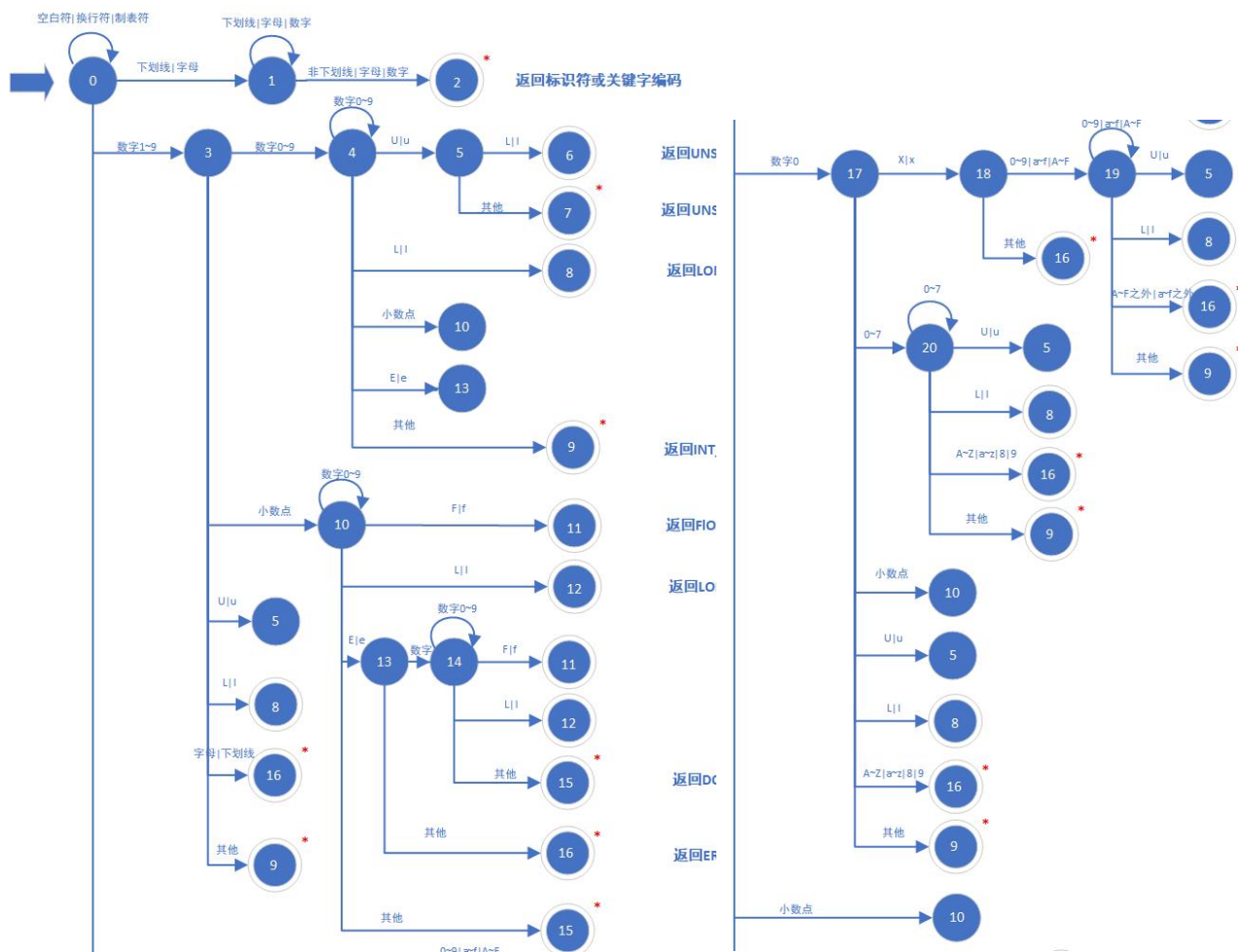


图 3-2-1 对于分析整型及浮点型常量的过程状态转换图

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’。（如图 3-2-2）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。

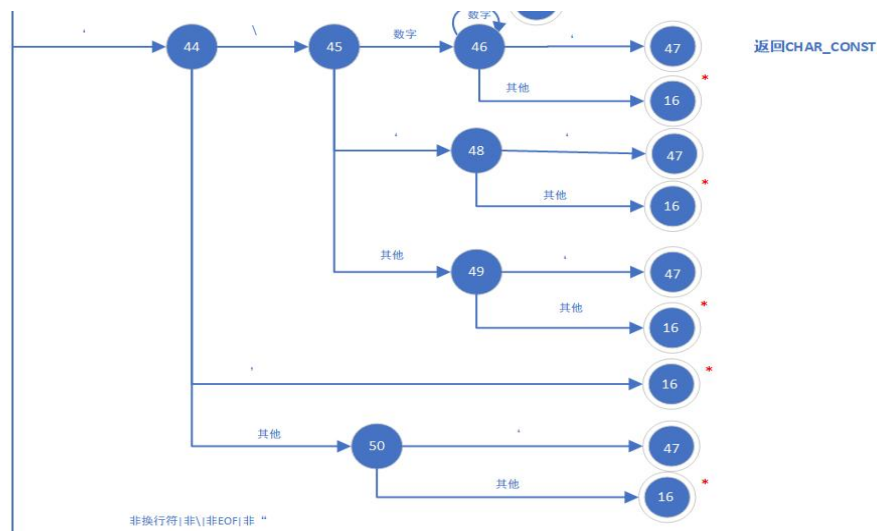


图 3-2-2 对于分析字符型常量的过程状态转换图

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\”。（如图 3-2-3）

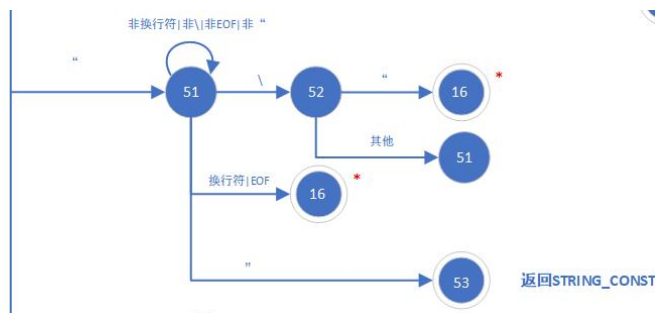


图 3-2-3 对于分析字符串型常量的过程状态转换图

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。（如图 3-2-4）

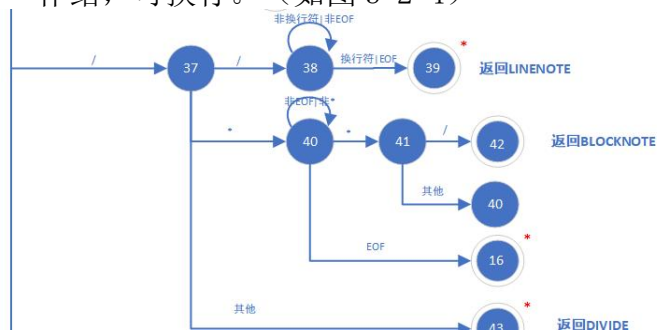


图 3-2-4 对于分析注释的过程状态转换图

3.2.2 预编译函数算法设计

预编译函数声明为 `status pre_process(FILE* fp);`。算法整体思路如下：首先调用 `gettoken` 函数读取一个单词，若该单词为井号，则进入宏定义或头文件判定与处理流程中；若为注释，则进入注释处理流程中；若为标识符，则将其与已存储的宏定义经行比较，并将定义值替换该标识符存放至中间文件中；若为其他单词，则将其直接存放至中间文件中。在处理过程中，还需关注换行操作。故设置变量 `pre_line_num` 用于存储处理前行数与处理后行数 `line_num` 进行比较，进行对应差值数量换行符的添加。

在宏定义与头文件判定流程中，首先判定第一关键字为 `define` 还是 `include`，否则函数返回，从而进行对应处理。若为 `define`，则调用 `gettoken` 函数再次读取一个单词，若为标识符且与 `define` 所在行一致时则存储至 `define_data` 结构数组中，否则返回 `ERROR`。再次调用词法分析函数读取一个单词，若其为常量且行数与 `define` 所在行数一致时，将其存入数组中，否则返回 `ERROR`。在头文件判定中，首先调用 `gettoken` 函数若为字符串常量且所在行数与 `include` 所在行一致则将其存入 `include_data` 结构数组中；若为左尖括号，则判断是否为文件定义值，再判断是否存在右尖括号，若判断正确则存入结构数组中，反之返回 `ERROR`。同时为避免出现结尾分号的情况，也会加以判定。相关处理简要流程图如图 3-3 所示。

在注释处理中，将 `pre_line_num` 与处理后的 `line_num` 的差值数量的换行符存放至中间文件中，以使中间文件内各程序语句所在行数正确。

3.2.3 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。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”

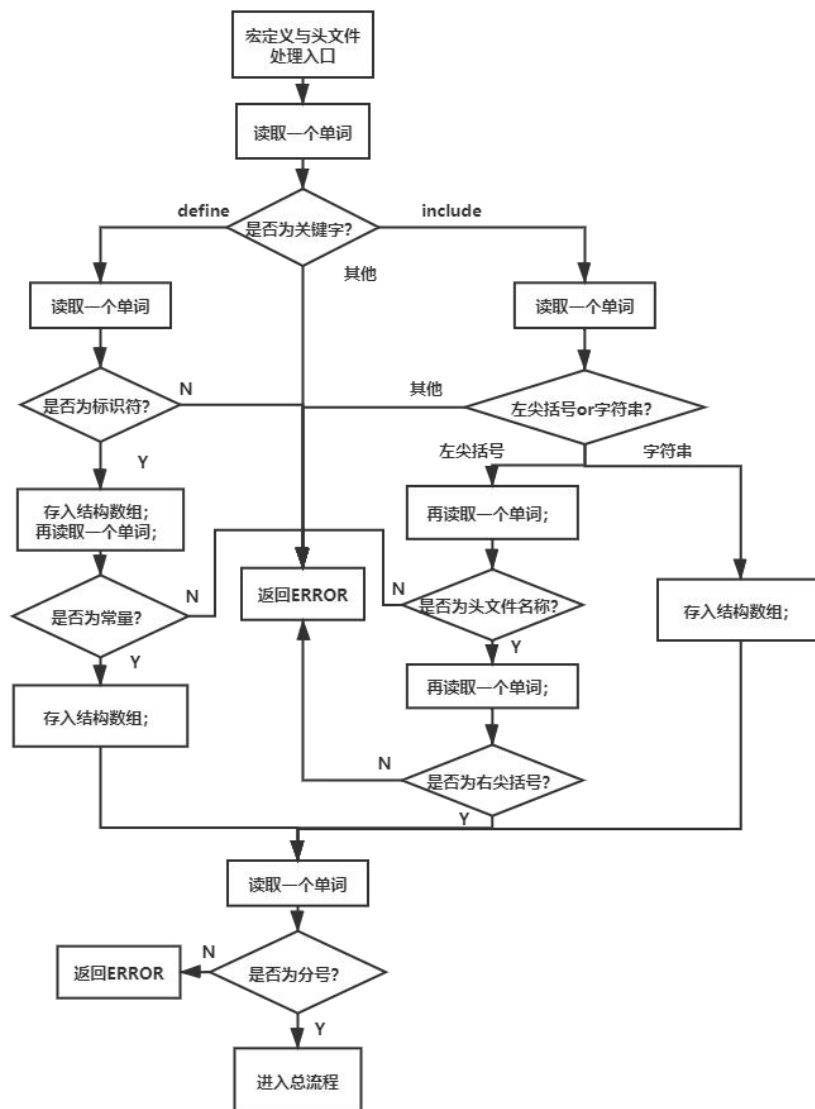


图 3-3 宏定义与头文件处理简要流程图

和 0 分别存入根节点的数据域 data 及 indent 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 ERROR。

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 为文件结尾 EOF 时,返回 INFEASIBLE。将“外部定义序列”和 0 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，否则返回 ERROR。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，返回 OK；否则返回 ERROR。

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 kind 中，

否则返回 ERROR。第二个单词必为标识符，若是则保存至 tokentext 中，否则返回 ERROR。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，若构建成功，则返回 OK；否则返回 ERROR。此时的局部语法树如图 3-4 所示。

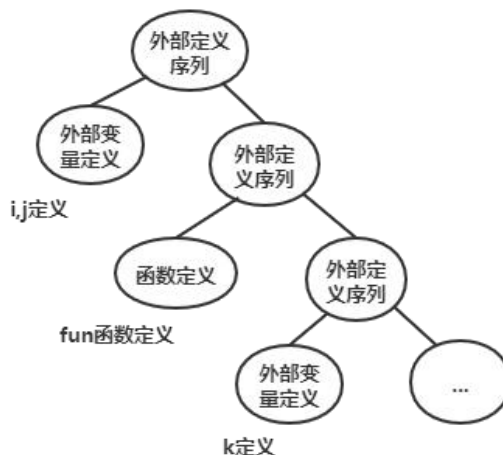


图 3-4 外部定义局部语法树

构建<外部变量定义>函数时，首先将“外部变量定义”和 1 存入根节点数据域中。生成类型子树 c，将预保存的类型关键字 kind 和 1 存入 c 的数据域中，作为 T 的第一棵子树，调用变量序列函数，生成第 2 棵子树，当生成子树出现错误时，返回 ERROR，否则返回 OK。在构建<变量序列>函数时，先将“变量序列”和 0 存入根节点数据域中，当单词为左方括号时，进入判断数组流程中，即读取数字以及右方括号，读取成功则将其和 1 存入子树 c 的数据域中作为 T 的第一棵子树；若单词不为左方括号，则下一个单词必为逗号或分号，否则错误，判定正确后将其存入子树 c 中作为 T 的第一颗子树。接着调用自身外部变量定义序列函数，生成子树 t，作为第 2 棵子树。若生成子树失败，则返回 ERROR，否则返回 OK。外部变量局部生成树如图 3-5 所示。

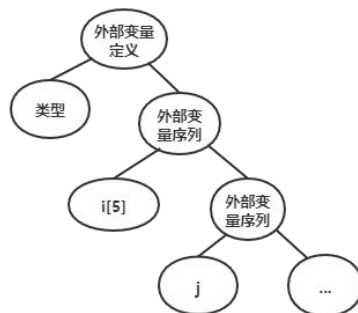


图 3-5 外部变量定义局部语法树

构建<函数定义>函数时，将“函数定义”和 1 存入根结点数据域中。将预存类型 kind 和 1 存入子树结点 c 的数据域中，将其作为 T 的第一棵子树。将预存函数名 gettoken 和 1 存入子树结点 p 的数据域中。然后判断形参，分为空与其他情况。若为 void 或者右小括号，则表明此形参为空，否则调用形参序列函数。若调用形参序列函数则生成子树 q，作为 p 的子树。将 p 作为 T 的第 2 棵子树。将“函数体”和 1 存入子树 f 的数据域中，此时调用复合语句函数生成 f 的子树，然后将 f 作为 T 的第 3 棵子树。函数定义局部生成树如 3-6 所示。其中形参序列子树构建与上述外部变量序列构建类似，再此不加赘述。



图 3-6 函数定义局部语法树

构建<复合语句>函数时，将“复合语句”和 1 存入根节点数据域中。此时进入复合语句处理流程，需要注意的是此时缩进值需自增 1，将当前行数及缩进值推入缩进处理队列中。读取单词，若为类型关键字，则调用局部变量定义序列函数构建子树 c 作为 T 的第一棵子树，然后调用语句序列函数构建 T 的第 2 棵子树；若不为类型关键字，则直接调用语句序列函数构建子树 p 作为 T 的第一棵子树。处理完成后，将缩进量自减 1，并与行数一同推入缩进处理队列中。并判断最后是否为右大括号作结，若是则返回 OK，否则返回 ERROR。其中局部变量定义序列子树构建与上述外部变量序列构建类似，再此不加赘述。

构建<语句序列>函数时，调用语句函数生成树 c，若生成成功则将“语句序列”和 0 存入根节点数据域中，生成错误则返回 ERROR，若无子树生成则返回 INFEASIBLE。将子树 c 作为根节点的第一棵子树，然后调用自身语句序列函数，生成 T 的第 2 棵子树，生成成功则返回 OK，否则返回 ERROR。

构建<语句>函数时，根据单词值进入不同的处理流程中。当单词值为 if 时，

判断括号及括号内内容是否正确,对于括号内的内容需调用表达式函数构建子树 p。将“条件”和 1 存入 c 的数据域中,并将建立好的表达式子树 p 作为 c 的第一棵子树。再读取单词,若为左大括号,则调用复合语句函数生成子树 q;否则,缩进值自增 1 与行数一同存入缩进处理队列中,再调用自身生成子树 q,再将缩进值自减 1。修改 p 子树,将“IF 子句”和 1 存入 p 的根节点数据域中,将 q 作为 p 的子树。再次判断下一个单词,若为 else,则继续按照 if 判断模式进行处理,生成子树 t。将 c 作为 T 的第一棵子树, p 作为第 2 棵子树, t 作为第 3 棵子树。构建 if-else 或 if 语句树。对于 for 语句,需要判定初始表达式,终止条件,循环表达式与 for 子句,生成的内容子树与 if-else 语句生成模式类似。对于 while 语句,则判定循环条件与 while 子句内容即可,与 if-else 语句生成模式类似。上述四种语句语法树简图如 3-7 所示。对于 continue 和 break 语句,需判断最后是否以分号作结。而对于 return 的情况,需判断是否有表达式以及是否以分号作结。而如果出现标识符或者常量,则需判断此句是否为表达式语句。另外空语句的情况也需要注意。若识别出左大括号,则调用自身语句函数;若识别出右大括号,则返回 INFEASIBLE,表示结束。

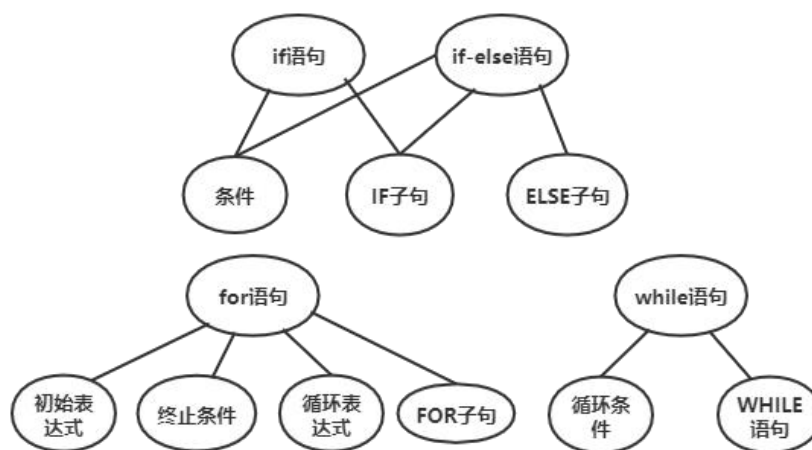


图 3-7 四种语句语法树简图

构建<表达式>函数前,需构建优先级比较函数用于处理表达式优先级问题。优先级比较函数采用二位数组比较的方法,形参为两个待比较优先级的符号。先将各符号编号,再根据行列构建,如表 3-1 所示的符号优先级二维数组,根据形参的两类符号,查找对应优先级大小关系,并返回相应优先级大小关系 >、< 或 =。构建<表达式>函数的形参必须包含结束标志符分号或者右小括号。定义两个

表3-1 符号优先级二维数组表

	+	-	*	/	%	()	=	>或<	==或!=	#	&&	
+	>	>	<	<	<	<	>		>	>	>	>	>
-	>	>	<	<	<	<	>		>	>	>	>	>
*	>	>	>	>	>	<	>		>	>	>	>	>
/	>	>	>	>	>	<	>		>	>	>	>	>
%	>	>	<	<	<	<	>		>	>	>	>	>
(<	<	<	<	<	<	=		<	<	>	<	<
)	>	>	>	>	>	>			>	>	>	>	>
=	<	<	<	<	<	<		<	<	<	>	<	<
>或<	<	<	<	<	<	<	>		>	>	>	>	>
==或!=	<	<	<	<	<	<	>		<	>	>	>	>
#	<	<	<	<	<	<		<	<	<	=	<	<
&&	<	<	<	<	<	<	>	>	<	<	>	>	>
	<	<	<	<	<	<	>	>	<	<	>	>	>

栈，分别为运算符栈 **op** 和操作数栈 **opn**；定义错误标记 **error**。将“表达式语句”和 1 存入根节点的数据域中。定义#为起止符，先将#入栈 **op**。进入循环操作至运算符栈栈顶为#号或者 **error** 不为 0，循环操作如下：当 **w** 为常量时，创建结点 **node**，将“ID”及常量值和 1 存入 **node** 数据域中，进栈 **opn**；当 **w** 为操作符时，获取 **op** 栈顶结点赋给 **node** 数据域，调用优先级比较函数。若返回值为‘<’，则将 **w** 推入 **op** 栈中；若返回值为‘=’，如果 **op** 栈顶为空则表示错误，否则出栈一个元素，然后直接读取下一个单词，达到去括号的目的；如果返回值为‘>’，则 **opn** 出栈一个元素赋给 **child1** 数据域，再出栈一个元素赋给 **child2** 数据域；**op** 出栈赋给 **node** 数据域。若有元素出栈未成功，则 **error** 自增 1。将 **child1** 作为 **node** 的第 1 棵子树，**child2** 作为第 2 棵子树。构建完成后将 **node** 推入 **opn** 栈中。当遇到 **w** 为结束标志符时，将#赋值给 **w**。若 **w** 所遇情况不在上述讨论范围内，则 **error** 自增 1。当退出函数时，若 **error** 不为 0，则返回 **ERROR**；否则，获取 **opn** 的栈顶元素作为 **T** 的子树，完成表达式语句树的构建。

3.2.4 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：

在此函数中，将 `indent` 作为处理缩进量的局部遍历，初始化为 0。其值取决于遍历结点的数据域中的 `indent` 值，当深度探索时，`indent` 会不断加上双亲结点数据域内的 `indent` 值；当遍历回退时，则会减去双亲结点数据域内的 `indent` 值。即局部变量 `indent` 值为该节点至根节点最短路径经过的结点数据域内 `indent` 的值。于是在输出时，会根据此时的局部变量 `indent` 打印相同数量的制表符。由于序列结点数据域设置 `indent` 值为 0，故巧妙避免了多余的语法输出。

3.2.5 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。

4 系统实现与测试

4.1 系统实现

(1) 系统实现环境

硬件环境：Intel Core i5-8250U 1.60GHz 内存 8G；

软件环境：Visual Studio 2019

(2) 系统架构组成

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、选择文件和退出程序操作。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 `line_num` 初始化为 1，表明操作行从第 1 行开始。于是调用 `pre_process` 预编译处理函数生成 `C_mid_file` 文件，若生成失败，则退出系统。生成成功后，再将处理行数 `line_num` 置为 1，根据中间文件进行词法分析处理，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。语法分析则先调用预编译处理函数生成中间文件，然后初始化行数为 1，使用中间文件，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 `TraverseTree` 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。而缩进编排功能则先进行预编译处理，再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 `PrintFile` 函数生成格式化文件 `C_print_file`。而文件选择操作，则使用 `goto` 函数，直接跳转至文件选择输入位置。其他输入则为退出系统操作。系统界面如 4-1 所示，具体系统架构如图 4-2 所示。

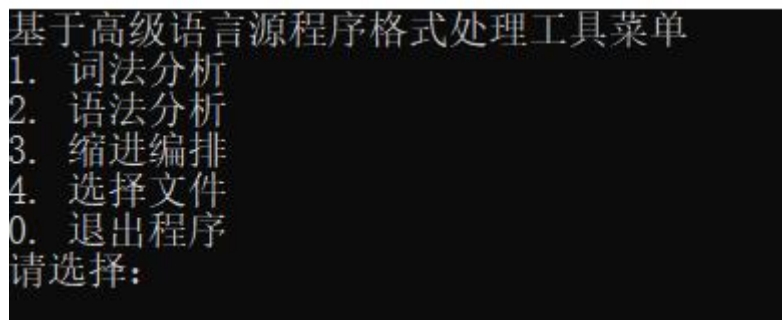


图 4-1 系统界面

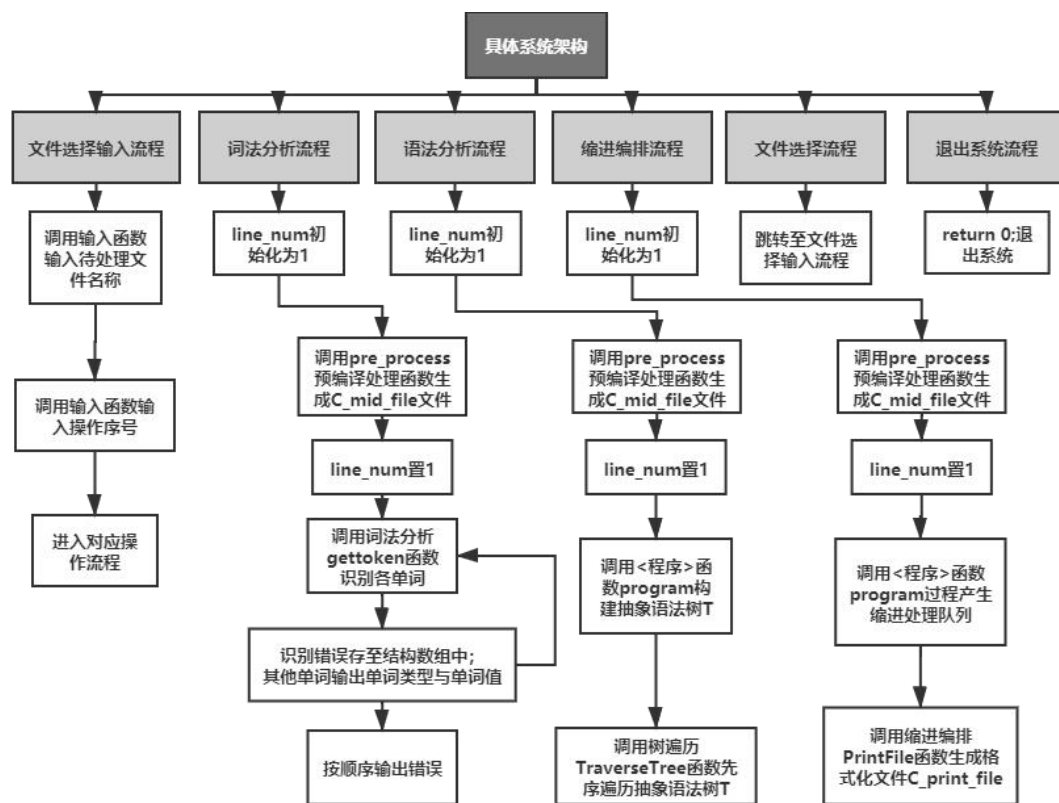


图 4-2 具体系统架构

4.2 系统测试

对于系统测试环节，具体包含文件选择功能测试、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 文件选择功能测试

1. 正确文件输入测试：

使用已保存待处理 C 语言源程序的文件，如 Cfile1.txt。输入后进入系统菜单。

2. 错误文件输入测试：

使用错误的文件名称进行输入，如 C.txt。输入后，系统输出“选择文件错误，请重新选择”。（如图 4-3）



图 4-3 错误文件输入测试

4.2.2 词法分析功能测试

1. 正确词法分析测试:

预编译处理与常量、标识符、关键字、定界符、运算符处理如下。

预编译处理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 4-4 所示。相关测试结果如 4-5 和 4-6 所示。

```
#include <stdio.h>
#include "stdlib.h"
#define OK 1
int b[30];
double c;
char string_S1;    //注释
/*注释*/
double func(int a, int b)
{
    char c;
    int group[30];
    a = OK; a=0xab4;
    a = 5ul; a = 067u;
    b = 1.f; b = .2e2L;
    c = 'a';
    c = '\t';
    c = '\n';
    c = '\\';
    return b;
}
float i;
```

图 4-4 部分正确词法测试用例

```
.. 正在预编译文件...
预编译成功! 按任意键继续...
```

图 4-5 预编译处理成功结果

单词类别	单词值
关键字	int
标识符	b
左中括号	[
右中括号]
常量	30
分号	;
关键字	double
标识符	c
分号	;
关键字	char
标识符	string_S1
分号	;
关键字	double
标识符	c
赋值运算符	=
无符号长整型常量	5ul
分号	;
标识符	a
赋值运算符	=
无符号整型常量	067u
分号	;
标识符	b
赋值运算符	=
浮点型常量	1.f
分号	;
标识符	b
赋值运算符	=
长双精度浮点型常量	.2e2L
分号	;
标识符	c
赋值运算符	=
字符型常量	'a'
分号	;
标识符	c
赋值运算符	=
字符型常量	'\t'
分号	;

错误列表 错误总数0

按任意键继续...

图 4-6 词法分析处理结果(截取关键部分)

对于字符串型常量，则单独进行测试，包含换行与不换行的字符串。测试用例如图 4-7，结果如图 4-8。

```
""
"abc"
"abc_a\bc"
"abc\
def"
```

图 4-7 字符串常量正确词法测试用例

```
单词类别      单词值
字符串常量    ""
字符串常量    "abc"
字符串常量    "abc_a\bc"
字符串常量    "abc\
def"
错误列表 错误总数0
按任意键继续...
```

图 4-8 词法分析结果

对于运算符，包含双目运算符“+、-、*、/、%”，关系运算符“>、<、>=、<=、==、!=”，逻辑运算符“&&、||”以及赋值运算符“=”。

2. 错误词法分析测试：

(1) 错误预编译处理

设置头文件、宏定义及注释错误，如图 4-9 所示。则程序显示预编译错误，如图 4-10 所示。

```
#include <stdio.h>
#include stdlib.h
#define OK
//注释
/*注释
char a;
int fuc(int a){
i=a+1;
return 0;
}
```

图 4-9 设置错误预编译测试用例

```
..正在预编译文件...
预编译失败！退出系统！请检查错误！
```

图 4-10 设置错误预编译测试结果

(2) 错误单词分析处理

设置标识符错误，如以数字开头或者含其他符号的标识符；设置字符串错误，如字符串最后以\作结“xxx\”以及直接换行，无右引号；设置字符型常量错误，如‘\’，‘ ‘’；设置其他常量错误，如“123ef”“0921”“0xZa12”。测试用例如图 4-11，测试结果如图 4-12。

```
int 1a;
"abc\
char a="\;
a="";
float b=123ef;
int c=0921;
c=0xZa12;
```

图 4-11 设置单词错误测试用例

```
错误列表 错误总数7
序号      行数
1          1
2          2
3          3
4          4
5          5
6          6
7          7
按任意键继续...
```

图 4-12 设置单词错误测试结果（局部）

4.2.3 语法分析功能测试

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。

测试用例 1 中包含外部变量定义、函数定义、局部变量定义及 return 语句，如 4-13 所示。测试结果如图 4-14 所示。

```
int b[30];
double c;
char string_S1;
double func(int a, int b)
{
    char c;
    int group[30];
    return b;
}
float i;
```

图 4-13 正确语法分析测试用例 1

```
外部变量定义:
  类型: int
  Array: b[30]
外部变量定义:
  类型: double
  ID: c
外部变量定义:
  类型: char
  ID: string_S1
函数定义:
  类型: double
  函数名: func
  形参:
    类型: int
    ID: a
    类型: int
    ID: b
  函数体:
    复合语句:
      局部变量定义:
        类型: char
        ID: c
      局部变量定义:
        类型: int
        Array: group[30]
      return语句:
        表达式语句:
          ID: b
外部变量定义:
  类型: float
  ID: i
按任意键继续...
```

图 4-14 正确语法分析测试结果 1

测试用例 2 中包含函数定义、局部变量定义、表达式语句、if-else 语句、if 语句、for 语句、while 语句、continue 语句以及 break 语句，如图 4-15。

测试结果如图 4-16。

```
void func(){
    int a;int b;int c;
    a=a+(b-c)*(b%f)/c;
    if(a>b)
        a=1;
    else {
        b=1;
    }
    for(a=0;a<1;a=a+1){
        continue;
    }
    while(b){
        if(a==b) {
            break;
        }
    }
}
```

图 4-15 正确语法分析测试用例 2

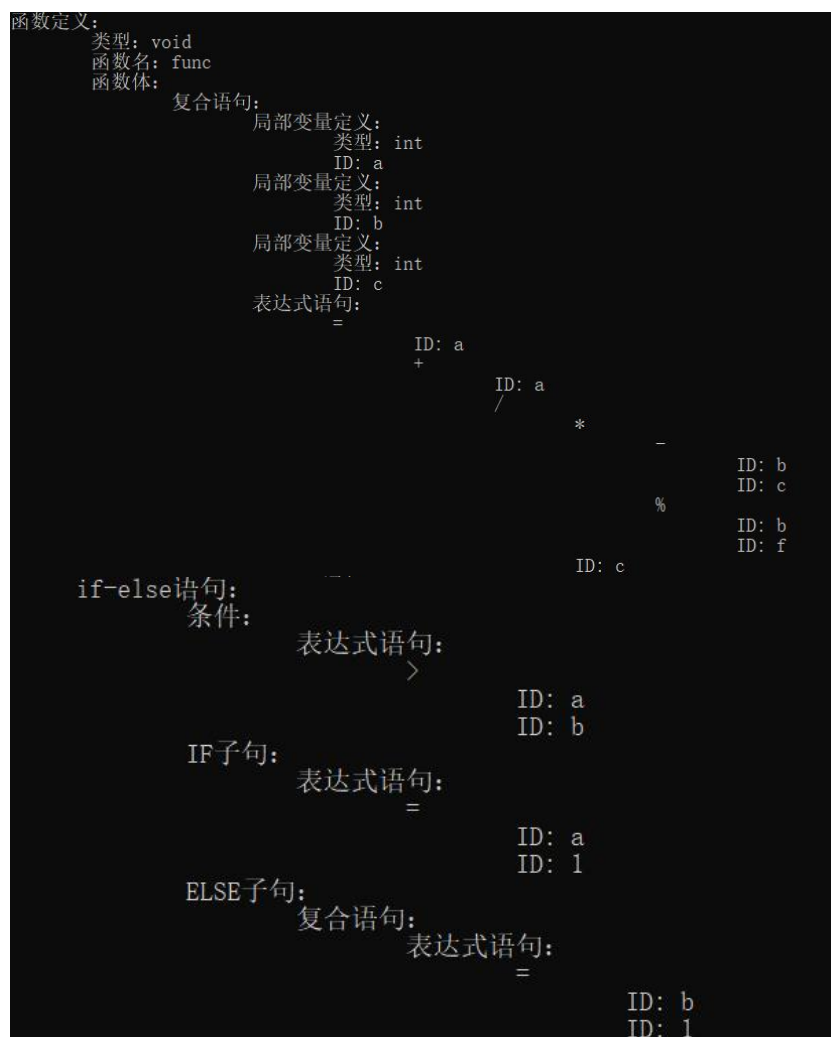


图 4-16 正确语法分析测试结果 2（部分）

2. 错误语法分析测试:

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，则是未加分号或者右大括号。如图 4-17 所示。

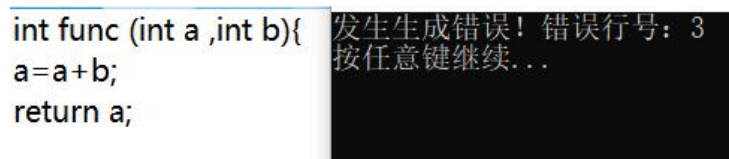


图 4-17 错误语法分析测试用例及结果-结尾符号

(2) 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 4-18。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 4-19。

```
int 0a;
int func (int a,int b){
a=a+b;
return a;
}
```

发生生成错误！错误行号：1
按任意键继续...

图 4-18 错误语法分析测试用例及结果-变量定义

```
int a;
int func ( a ,int b){
a=a+b;
return a;
}
```

发生生成错误！错误行号：2
按任意键继续...

图 4-19 错误语法分析测试用例及结果-函数定义

(3) 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。如未添加条件表达式语句（如图 4-20），表达式括号缺失（如图 4-21）等。

```
int func (int a){
for(a=1;;a=a+1){
a=a-1;
}
return a;
}
```

发生生成错误！错误行号：2
按任意键继续...

图 4-20 错误语法分析测试用例及结果-条件缺失

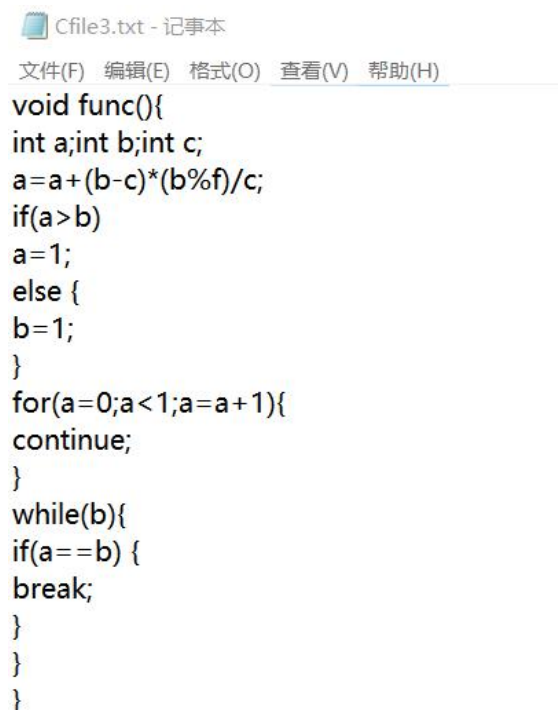
```
int func (int a){
for(a=1;a<10;a=a+1
a=a*((2+a)*a;
}
return a;
}
```

发生生成错误！错误行号：3
按任意键继续...

图 4-21 错误语法分析测试用例及结果-括号缺失

4.2.4 缩进编排功能测试

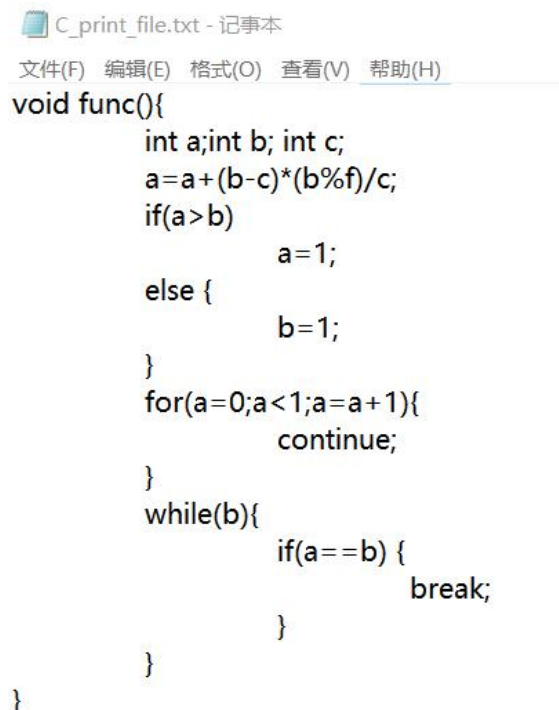
对待格式化 C 语言源程序进行缩进编排。编排前源文件如图 4-22 所示，编排后格式化文件如图 4-23 所示。



```

void func(){
int a;int b;int c;
a=a+(b-c)*(b%f)/c;
if(a>b)
a=1;
else {
b=1;
}
for(a=0;a<1;a=a+1){
continue;
}
while(b){
if(a==b) {
break;
}
}
}
    
```

图 4-22 缩进编排前原文件



```

void func(){
    int a;int b; int c;
    a=a+(b-c)*(b%f)/c;
    if(a>b)
        a=1;
    else {
        b=1;
    }
    for(a=0;a<1;a=a+1){
        continue;
    }
    while(b){
        if(a==b) {
            break;
        }
    }
}
    
```

图 4-23 缩进编排后格式化文件

4.2.5 退出系统功能测试

输入除编号数外的数字，即可退出系统。

5 总结与展望

5.1 全文总结

此次课程设计总结如下：

（1）选择邻接表模型构建抽象语法树数据结构，在课程知识支持的基础上，与无向图邻接表模型相类比，熟悉子树插入以及树的遍历操作；

（2）词法分析根据 DFA 状态转化图的构建相关词法分析流程框架；语法分析主要通过递归下降子程序法构建递进下降的语法部件函数，按巴克斯（BNF）范式定义的语法规则将各部件函数进行递归调用；

（3）对于抽象语法树的缩进输出，在数据结构中设置缩进值；对于缩进编排格式生成格式化文件，通过抽象语法树的构建过程中记录有关深度产生缩进处理队列；

（4）整体系统采用输入操作编码值进入对应流程的模式，通过函数调用模式进行流程化处理；

5.2 工作展望

在今后的研究中，围绕着如下方面开展工作：

（1）扩展知识体系，不断扩展知识面，为以后工作提供知识储备；

（2）坚持以实践为基础，不能纸上谈兵。而是要多编代码，从而掌握优化代码规范性、泛化性和可读性。

（3）扩展知识获取和实践平台渠道。除通过书本学习外，还需利用网上广泛学习资源，获取课堂知识体系之外的知识内容。尝试不同环境的代码编译，通过对比和类比，弄清各环境之间编译操作的异同，提升自身代码编写能力。

6 体会

此次课程设计选择了第二个项目，为编译操作的前端处理过程。在词法分析中通过分析不同的情况，从而判断对应单词类型以及相关错误。这一过程特别考验逻辑思维能力以及思维扩展能力。换句话说，通过词法分析函数的构建，让我对 C 语言的词法体系有了更加全面的了解，特别是对 C 语言的词法错误，例如字符串换行处理，字符型常量转义错误等。同时通过画状态转换图，让我在思路混乱时豁然开朗，真正感受到流程图对梳理算法逻辑的巨大帮助，在今后的学习和编码中，要学会画图，通过画图可能会达到大大减轻思维复杂的效果。除此之外，对于各类情况的考虑使思维发散，体会到从 0 到 1 的乐趣，在思维的不断扩散中逐渐扩大词法体系。让我明白了，当遇见庞大的待处理事物时不要担心，而是需要从小到大，由整体到局部进行处理。通过思维扩散和细节处理，把握处理事物的核心。

而在抽象语法树的生成和遍历处理前，则务必要选择好合适的数据结构来存储此树，切不可操之过急，胡乱选择。通过对比发现，数据结构课程中学到过邻接表模型，虽然是在无向图的构建中掌握的，但对于树而言亦可看作无向图，且逻辑上构建十分清晰。从而在树的生成、子树插入以及先序遍历时可与无向图的生成、顶点插入以及深度优先搜索遍历相比对，发现几乎一致。这在一定程度上减轻了辅助函数构造的难度。让我明白了，在进行相关数据结构设计时，要学会与学过的知识相联系，而不是凭空想象，想象的基础也必须是在已存在的知识体系上。所以，要通过不断学习新的知识来扩充自己的知识体系，这能在今后的学习和工作中提供不少便利。

而在语法分析时，采用递归下降子程序的方法。采用这种方法，瞬间让悬着的心落下来了。在设计前，一想到 C 语言无比巨大的语法体系就瑟瑟发抖。而通过将语法分析拆分成各个模块并进行各模块之间的联系的方法，让我豁然开朗。原来无比复杂的处理居然可以表示成逻辑如此清晰的小问题框架。果然大问题化小问题是设计思路混乱时的定心丸，不仅可以理顺逻辑，还可以减轻流程调试及过程分析的困难，并且代码的可读性大幅度提升。

最后，此次课程设计让我感受最深的则是学无止境的道理，只有不断学习才能扩展自己的思维，才能够便利往后的学习与工作。

参考文献

- [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

• 主函数 main.cpp

```
#include "lexer.h"
#include "parser.h"
#include "production.h"
#include "preprocess.h"
#include "printfile.h"

int main() {
    char filename[30];
    int ch = 1;
    int w;
    CTree T;
    FILE* fp, *mid_fp;
    extern define_data data_Def[10]; //用于储存 define 宏定义的内容
    extern int data_Def_num;
    extern int line_num;
    extern char token_text[100];
    int error_line[100]; //记录错误行数
    int error_line_num=0; //记录错误总个数
    int i; //用于循环
Choosefile:
    system("cls");
    printf("输入文件名: ");
    scanf("%s", filename);
    if (!(fp = fopen(filename, "r")))
    {
        printf("... 选择文件错误, 请重新选择...\n");
        getchar(); getchar();
        goto Choosefile;
    }
    else fclose(fp);

    while (ch)
    {
        system("cls");
        printf("基于高级语言源程序格式处理工具菜单\n1. 词法分析\n2. 语法分析\n3. 缩进\n4. 选择文件\n0. 退出程序\n请选择: ");
        scanf("%d", &ch);
        system("cls");
        switch (ch)
        {
            case 1:
```

```

error_line_num = 1;
line_num = 1; //行数初始化为1
if (!(fp = fopen(filename, "r")))
{ printf("...选择文件错误, 请重新选择...\n");
  getchar(); getchar();
  break;
}
printf("...正在预编译文件...\n");
if (pre_process(fp))
{
    printf("预编译成功! 按任意键继续...\n");
    getchar(); getchar();
}
else
{
    printf("预编译失败! 退出系统! 请检查错误!\n");
    getchar(); getchar();
    return 0;
}
mid_fp = fopen("C_mid_file.txt", "r");
line_num = 1; //行数初始化为1
system("cls");
printf("\n");
printf("  单词类别                单词值\n");
while (!feof(mid_fp))
{
    w = gettoken(mid_fp);
    if (w >= AUTO && w <= DEFINE)
    {
        printf("  关键字                %s\n", token_text);
    }
    switch (w)
    {
        case POUND:
            printf("  井号                %s\n", token_text);
            break;
        case IDENT:
            printf("  标识符                %s\n", token_text);
            break;
        case INT_CONST:
            printf("  整型常量                %s\n", token_text);
            break;
        case UNSIGNED_CONST:
            printf("  无符号整型常量            %s\n", token_text);
            break;
    }
}

```

```

case LONG_CONST:
    printf(" 长整型常量                %s\n", token_text);
    break;
case UNSIGNED_LONG_CONST:
    printf(" 无符号长整型常量          %s\n", token_text);
    break;
case DOUBLE_CONST:
    printf(" 双精度浮点型常量          %s\n", token_text);
    break;
case FLOAT_CONST:
    printf(" 浮点型常量                %s\n", token_text);
    break;
case LONG_DOUBLE_CONST:
    printf(" 长双精度浮点型常量            %s\n", token_text);
    break;
case CHAR_CONST:
    printf(" 字符型常量                  %s\n", token_text);
    break;
case STRING_CONST:
    printf(" 字符串常量                  %s\n", token_text);
    break;
case EQUAL_TO:
    printf(" 赋值运算符                  =\n");
    break;
case PLUS:
    printf(" 加法运算符                  +\n");
    break;
case MINUS:
    printf(" 减法运算符                  -\n");
    break;
case MULTIPLY:
    printf(" 乘法运算符                  *\n");
    break;
case DIVIDE:
    printf(" 除法运算符                  /\n");
    break;
case MOD:
    printf(" 取模运算符                  %%\n");
    break;
case MORE:
    printf(" 关系运算符                  >\n");
    break;
case LESS:
    printf(" 关系运算符                  <\n");
    break;

```



```

case EQUAL:
    printf(" 关系运算符          ==\n");
    break;
case UNEQUAL:
    printf(" 关系运算符          !=\n");
    break;
case MORE_EQUAL:
    printf(" 关系运算符          >=\n");
    break;
case LESS_EQUAL:
    printf(" 关系运算符          <=\n");
    break;
case AND:
    printf(" 逻辑与          &&\n");
    break;
case OR:
    printf(" 逻辑或          ||\n");
    break;
case LS:
    printf(" 左小括号          (\n");
    break;
case RS:
    printf(" 右小括号          )\n");
    break;
case LM:
    printf(" 左中括号          [\n");
    break;
case RM:
    printf(" 右中括号          ]\n");
    break;
case LL:
    printf(" 左大括号          {\n");
    break;
case RL:
    printf(" 右大括号          }\n");
    break;
case SEMI:
    printf(" 分号          ;\n");
    break;
case COMMA:
    printf(" 逗号          ,\n");
    break;
case ERROR_TOKEN:
    error_line[error_line_num] = line_num;
    error_line_num++;

```

```

        break;
    }
}
printf("\n");
printf("错误列表 错误总数%d\n", error_line_num);
if (error_line_num)
{
    printf("\n");
    printf("序号 行数\n");
    for (i = 0; i < error_line_num; i++) {
        printf("    %d    %d\n", i+1, error_line[i]);
    }
}
fclose(fp);
fclose(mid_fp);
printf("\n 按任意键继续...");
getchar();
break;
case 2:
    printf("... 正在预编译文件...\n");
    line_num = 1; //行数初始化为1
    fp = fopen(filename, "r");
    if (pre_process(fp))
    {
        printf("预编译成功! 按任意键继续...\n");
        getchar(); getchar();
    }
    else
    {
        printf("预编译失败! 退出系统! 请检查错误!\n");
        getchar(); getchar();
        return 0;
    }
    mid_fp = fopen("C_mid_file.txt", "r");
    line_num = 1; //行数初始化为1
    system("cls");
    if (!program(mid_fp, T))
    {
        printf("发生生成错误! 错误行号: %d\n", line_num);
        printf("按任意键继续...\n");
        getchar();
        break;
    }
    TraverseTree(T, PrintTree);
    printf("\n 按任意键继续...\n");

```

```

        fclose(fp);
        fclose(mid_fp);
        getchar();
        break;
    case 3:
        line_num = 1; //行数初始化为1
        fp = fopen(filename, "r");
        if (!pre_process(fp))
        {
            printf("预编译失败！退出系统！请检查错误！\n");
            getchar(); getchar();
            return 0;
        }
        line_num = 1;
        mid_fp = fopen("C_mid_file.txt", "r");
        if (!program(mid_fp, T))
        {
            printf("程序存在语法错误！无法缩进打印！\n");
            getchar(); getchar();
            break;
        }
        else { printf("程序语法正确！\n"); }
        fclose(fp);
        fp = fopen(filename, "r");
        PrintFile(fp);
        printf("缩进编排文件生成成功！\n");
        getchar(); getchar();
        fclose(fp);
        fclose(mid_fp);
        break;
    case 4:
        goto Choosefile;

    default:
        return 0;
    }
}
return 1;
}

```

• lexer.h

#pragma once

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

enum token_kind{AUTO, BREAK, CASE, CHAR, CONST, CONTINUE, DEFAULT, DO,
DOUBLE, ELSE, ENUM, EXTERN, FLOAT, FOR, GOTO, IF,
INT, LONG, REGISTER, RETURN, SHORT, SIGNED, SIZEOF, STATIC,
STRUCT, SWITCH, TYPEDEF, UNION, UNSIGNED, VOID, VOLATILE, WHILE, INCLUDE, DEFINE, //以上为
关键字
IDENT, //标识符
INT_CONST, UNSIGNED_CONST, LONG_CONST, UNSIGNED_LONG_CONST, DOUBLE_CONST, FLOAT_CONST,
LONG_DOUBLE_CONST, CHAR_CONST, STRING_CONST, //常量
PLUS, MINUS, MULTIPLY, DIVIDE, MOD, //双目运算符
MORE, LESS, EQUAL, UNEQUAL, MORE_EQUAL, LESS_EQUAL, //关系运算符
AND, OR,
EQUAL_TO, //赋值运算符
LL, RL, LM, RM, LS, RS, //括号 (L 大, M 中, S 小)
LINENOTE, BLOCKNOTE, SEMI, COMMA, ERROR_TOKEN, POUND //其他单词, 其中 SEMI 为分号, COMMA 为逗号, POUND 为井号, ERROR_TOKEN 标识错误

};

typedef struct keyword { //处理关键字所构建的结构
    char key[10];
    int enum_key;
}keyword;

extern keyword n[IDENT]; //储存各类关键字
extern char token_text[100]; //暂存常量
extern int line_num; //检测运行行数
int gettoken(FILE* fp);
```

• parser.h

```
#pragma once

#include<queue>
#include<stack>
#include"lexer.h"
#include"profuction.h"
using namespace std;
#define OK 1
#define ERROR 0
#define INFEASIBLE -1
struct print
```

```
{
    int indent;                //记录缩进值
    int linenum;               //记录缩进的行号
};

typedef int status;
extern int indent0;           //记录源代码的缩进值
extern queue<print> printList; //存储各个打印格式单元
extern int w;                 //获得 gettoken 函数的返回值即读入的单词种类编码
extern char kind[100];        //存取类型关键字
extern char tokenText0[100];  //存取变量名或函数名

status program(FILE* fp, CTree& T); //语法单位<程序>的子程序
status ExtDefList(FILE* fp, CTree& T); //语法单位<外部定义序列>的子程序
status ExtDef(FILE* fp, CTree& T); //语法单位<外部定义>的子程序
status ExtVarDef(FILE* fp, CTree& T); //语法单位<外部变量定义>子程序
status VarList(FILE* fp, CTree& T); //语法单位<变量序列>子程序
status funcDef(FILE* fp, CTree& T); //语法单位<函数定义>子程序
status ParamList(FILE* fp, CTree& T); //语法单位<形参序列>子程序
status FormParDef(FILE* fp, CTree& T); //语法单位<形参定义>子程序
status CompStat(FILE* fp, CTree& T); //语法单位<复合语句>子程序
status LocVarList(FILE* fp, CTree& T); //语法单位<局部变量定义序列>子程序
status LocVarDef(FILE* fp, CTree& T); //语法单位<局部变量定义>子程序
status StatList(FILE* fp, CTree& T); //语法单位<语句序列>子程序
status Statement(FILE* fp, CTree& T); //语法单位<语句>子程序
status exp(FILE* fp, CTree& T, int endsym); //语法单位<表达式>子程序
char precede(char* a, char* b); //比较 a 与 b 的优先级
status PrintTree(char* data, int indent); //打印缩进
```

• preprocess.h

```
#pragma once
//预处理
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

typedef int status;
#define OK 1
#define ERROR 0
typedef struct define_data { //用于储存 define 宏定义数据
    char ident[50];
    char string[50];
}define_data;

typedef struct include_data { //用于储存 include 头文件定义数据
```

```

    char string[50];
}include_data;

extern define_data data_Def[10]; //用于储存 define 宏定义的内容
extern include_data data_Includ[10]; //用于储存 include 文件包含的内容
extern int data_Def_num;

status pre_process(FILE* fp);

```

• printfile.h

```

#pragma once

#include<stdio.h>
#include<string.h>
#include"parser.h"
#define ERROR 0
#define OK 1
typedef int status;

status PrintFile(FILE* fp);

```

• profuction.h

```

#pragma once

#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<stack>
#include<queue>
using namespace std;

#define OK 1
#define ERROR 0
#define INFEASIBLE -1
typedef int status; //函数返回值类型

/*树的相关说明：*/
typedef struct Child //孩子结点
{
    int child;
    Child* next;
}*CNode;
typedef struct CTNode

```

```
{
    char* data;
    int indent;          //该结点的孩子要增加的缩进量
    CNode firstchild; //孩子链表头结点
}CTNode;
typedef struct CTree
{
    CNode nodes[1000];
    int n, r;           //结点数和根的位置
}CTree;

status InitTree(CTree& T);
status InsertChild(CTree& T, int p, int i, CTree c);
status GetParent(CTree T, int child, int& parent);
status TraverseTree(CTree T, status(*visit)(char*, int));

/*栈的相关说明*/
typedef CTree* SElemType; //定义元素类型
typedef struct
{
    SElemType* base;          //在栈构造之前和销毁之后, base 的值为 NULL
    SElemType* top;           //栈顶指针
    int stacksize;           //当前已分配的存储空间, 以元素为单位
}SqStack;
status InitStack(SqStack& S);
status GetTop(SqStack S, SElemType& e);
status Push(SqStack& S, SElemType e);
status Pop(SqStack& S, SElemType& e);
```

• lexer.cpp

```
#include"lexer.h"

int line_num = 1;

char token_text[100];

keyword n[IDENT] = {

    {"auto", AUTO}, {"break", BREAK}, {"case", CASE}, {"char", CHAR}, {"const", CONST}, {"continu
e", CONTINUE}, {"default", DEFAULT}, {"do", DO},

    {"double", DOUBLE}, {"else", ELSE}, {"enum", ENUM}, {"extern", EXTERN}, {"float", FLOAT}, {"f
```

```
or", FOR}, {"goto", GOTO}, {"if", IF},

{"int", INT}, {"long", LONG}, {"register", REGISTER}, {"return", RETURN}, {"short", SHORT}, {"
"signed", SIGNED}, {"sizeof", SIZEOF}, {"static", STATIC},

{"struct", STRUCT}, {"switch", SWITCH}, {"typedef", TYPEDEF}, {"union", UNION}, {"unsigned",
UNSIGNED}, {"void", VOID}, {"volatile", VOLATILE}, {"while", WHILE},

{"include", INCLUDE}, {"define", DEFINE}

};
```

```
int gettoken(FILE* fp) {
    char c; //用于暂存读取的单个字符
    int i=0; //用于做 token_text 的存储
    int j=0; //用于与关键字做比对

    while ((c = fgetc(fp)) == ' ' || c == '\n' || c == '\t' || c==EOF) //跳过空白符和制表
符
    {
        if (c == '\n') line_num++; //读取换行符时计数
        if (c == EOF) return EOF;
    }
    if (c == EOF) return EOF;
    //判断标识符或者关键字
    if ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z')) {
        do {
            token_text[i++] = c;
            c = fgetc(fp);
        } while ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') || c == '_' || (c >= '0'
&& c <= '9') || c=='.');
        token_text[i] = '\0';
        ungetc(c, fp);

        for (; j < IDENT; j++) {
```



```

        if (!strcmp(token_text, n[j].key)) return n[j].enum_key;
    }

    return IDENT;
}

//判断标识符
if (c == '_' ) {
    do {
        token_text[i++] = c;
        c = fgetc(fp);
    } while ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') || c == '_' || (c >= '0'
&& c <= '9') || c == '.' );
    token_text[i] = '\0';
    ungetc(c, fp);
    return IDENT;
}

//判断数字（整型或浮点型）
if (c > '0' && c <= '9') { //首非0
    token_text[i++] = c;
    c = fgetc(fp);
    if (c >= '0' && c <= '9') {
        do {
            token_text[i++] = c;
            c = fgetc(fp);
        } while (c >= '0' && c <= '9');

        if (c == 'u' || c == 'U') {
stationA: //u/U 后缀
            token_text[i++] = c;
            c = fgetc(fp);

```

```

        if (c == 'l' || c == 'L') {
            token_text[i++] = c;
            token_text[i] = '\0';
            return UNSIGNED_LONG_CONST;
        }
        else {
            ungetc(c, fp);
            token_text[i] = '\0';
            return UNSIGNED_CONST;
        }
    }
    else if (c == 'l' || c == 'L') {
stationB: //l/L 后缀(int)
        token_text[i++] = c;
        token_text[i] = '\0';
        return LONG_CONST;
    }
    else if (c == '.') {
        goto station1;
    }
    else if (c == 'e' || c == 'E') {
        goto stationE;
    }
    else {
        ungetc(c, fp);
        token_text[i] = '\0';
        return INT_CONST;
    }
}

else if (c == '.') {

```

station1://情况1（小数点）*可能没数字情况

```
do {  
    token_text[i++] = c;  
    c = fgetc(fp);  
} while (c >= '0' && c <= '9');  
if(c=='f' || c=='F') {
```

stationC: //f/F 后缀

```
    token_text[i++] = c;  
    token_text[i] = '\0';  
    return FLOAT_CONST;  
}  
else if (c == 'l' || c == 'L') {
```

stationD: //l/L 后缀（double）

```
    token_text[i++] = c;  
    token_text[i] = '\0';  
    return LONG_DOUBLE_CONST;  
}  
else if (c == 'e' || c == 'E') {
```

stationE: //e/E 的情况

```
    token_text[i++] = c;  
    c = fgetc(fp);  
    if (c >= '0' && c <= '9') {  
        do {  
            token_text[i++] = c;  
            c = fgetc(fp);  
        } while (c >= '0' && c <= '9');  
        if (c == 'f' || c == 'F') {  
            goto stationC;  
        }  
        else if (c == 'l' || c == 'L') {  
            goto stationD;  
        }
```

```

        }

        else {

            ungetc(c, fp);

            token_text[i] = '\0';

            return DOUBLE_CONST;

        }

    }

    else goto stationERROR;

}

else {

    ungetc(c, fp);

    token_text[i] = '\0';

    return DOUBLE_CONST;

}

}

else if (c == 'u' || c == 'U') {

    goto stationA;

}

else if (c == 'l' || c == 'L') {

    goto stationB;

}

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

stationERROR:

    ungetc(c, fp);

    token_text[i] = '\0';

    return ERROR_TOKEN;

}

else {

    ungetc(c, fp);

    token_text[i] = '\0';

```

```

        return INT_CONST;
    }
}

if (c == '0') { //首为0
    token_text[i++] = c;
    c = fgetc(fp);
    if (c == 'x' || c == 'X') { //判断十六进制数
        token_text[i++] = c;
        c = fgetc(fp);
        if ((c >= '0' && c <= '9') || (c >= 'A' && c <= 'F') || (c >= 'a' && c <= 'f'))
        {
            do {
                token_text[i++] = c;
                c = fgetc(fp);
            } while ((c >= '0' && c <= '9') || (c >= 'a' && c <= 'f') || (c >= 'A'
&& c <= 'F'));
            if (c == 'u' || c == 'U') goto stationA;
            else if (c == 'l' || c == 'L') goto stationB;
            else if ((c >= 'g' && c <= 'z' && c != 'u' && c != 'l') || (c >= 'G' &&
c <= 'Z' && c != 'U' && c != 'L')) goto stationERROR;
            else {
                ungetc(c, fp);
                token_text[i] = '\0';
                return INT_CONST;
            }
        }
    }
    else goto stationERROR;
}

else if (c >= '0' && c <= '7') { //判断八进制数

```

```

do {
    token_text[i++] = c;

    c = fgetc(fp);

} while (c >= '0' && c <= '7');

if (c == 'u' || c == 'U') goto stationA;

else if (c == 'l' || c == 'L') goto stationB;

else if ((c >= 'a' && c <= 'z' && c != 'u' && c != 'l') || (c >= 'A' && c <=
'Z' && c != 'U' && c != 'L')) || c == '_' || (c >= '8' && c <= '9')) goto stationERROR;

else {
    ungetc(c, fp);

    token_text[i] = '\0';

    return INT_CONST;

}

}

else if (c == '.') goto station1;

else if (c == 'u' || c == 'U') goto stationA;

else if (c == 'l' || c == 'L') goto stationB;

else if ((c >= 'a' && c <= 'z' ) || (c >= 'A' && c <= 'Z' ) || c == '_' || (c >=
'8' && c <= '9')) goto stationERROR;

else {
    ungetc(c, fp);

    token_text[i] = '\0';

    return INT_CONST;

}

}

if (c == '.') { //出现小数点的情况*后面必须出现数字

    token_text[i++] = c;

    c = fgetc(fp);

    if (c >= '0' && c <= '9') {

```

```
do {  
    token_text[i++] = c;  
    c = fgetc(fp);  
} while (c >= '0' && c <= '9');  
  
if (c == 'f' || c == 'F')  
{  
    goto stationC;  
}  
else if (c == 'l' || c == 'L')  
{  
    goto stationD;  
}  
else if (c == 'e' || c == 'E') {  
    goto stationE;  
}  
else {  
    ungetc(c, fp);  
    token_text[i] = '\\0';  
    return DOUBLE_CONST;  
}  
}  
  
else {  
    goto stationERROR;  
}  
}  
  
switch (c) {  
case '=':  
    token_text[i++] = c;  
    c = fgetc(fp);
```

```
    if (c == '=') {  
        token_text[i++] = c;  
        token_text[i] = '\0';  
        return EQUAL;  
    }  
    else {  
        ungetc(c, fp);  
        token_text[i] = '\0';  
        return EQUAL_T0;  
    }  
case '{':  
    token_text[i++] = c;  
    token_text[i] = '\0';  
    return LL;  
  
case '}':  
    token_text[i++] = c;  
    token_text[i] = '\0';  
    return RL;  
  
case '[':  
    token_text[i++] = c;  
    token_text[i] = '\0';  
    return LM;  
  
case ']':  
    token_text[i++] = c;  
    token_text[i] = '\0';  
    return RM;  
  
case '(':
```



```
    token_text[i++] = c;
    token_text[i] = '\0';
    return LS;

case')':
    token_text[i++] = c;
    token_text[i] = '\0';
    return RS;

case';':
    token_text[i++] = c;
    token_text[i] = '\0';
    return SEMI;

case',':
    token_text[i++] = c;
    token_text[i] = '\0';
    return COMMA;

case'#':
    token_text[i++] = c;
    token_text[i] = '\0';
    return POUND;

case'>':
    token_text[i++] = c;
    c = fgetc(fp);
    if (c == '=') {
        token_text[i++] = c;
        token_text[i] = '\0';
        return MORE_EQUAL;
```

```
    }

    else {

        ungetc(c, fp);

        token_text[i] = '\0';

        return MORE;

    }

case '<':

    token_text[i++] = c;

    c = fgetc(fp);

    if (c == '=') {

        token_text[i++] = c;

        token_text[i] = '\0';

        return LESS_EQUAL;

    }

    else {

        ungetc(c, fp);

        token_text[i] = '\0';

        return LESS;

    }

case '!':

    token_text[i++] = c;

    c = fgetc(fp);

    if (c == '=') {

        token_text[i++] = c;

        token_text[i] = '\0';

        return UNEQUAL;

    }

    else {

        goto stationERROR;
```

```
    }

case '&':
    token_text[i++] = c;
    c = fgetc(fp);
    if (c == '&') {
        token_text[i++] = c;
        token_text[i] = '\0';
        return AND;
    }
    else {
        goto stationERROR;
    }

case '|':
    token_text[i++] = c;
    c = fgetc(fp);
    if (c == '|') {
        token_text[i++] = c;
        token_text[i] = '\0';
        return OR;
    }
    else {
        goto stationERROR;
    }

case '+':
    token_text[i++] = c;
    token_text[i] = '\0';
    return PLUS;
```

```
case '-' :
```

```
    token_text[i++] = c;
    token_text[i] = '\0';
    return MINUS;
```

```
case '*' :
```

```
    token_text[i++] = c;
    token_text[i] = '\0';
    return MULTIPLY;
```

```
case '%' :
```

```
    token_text[i++] = c;
    token_text[i] = '\0';
    return MOD;
```

```
case '/' :
```

```
    token_text[i++] = c;
    c = fgetc(fp);
    if (c == '/') {    //判断行注释
        do {
            token_text[i++] = c;
            c = fgetc(fp);
        } while (c != '\n' && c != EOF);
        if (c == '\n' || c == EOF) {
            ungetc(c, fp);
            token_text[i] = '\0';
            return LINENOTE;
        }
    }
}
```

```
else if (c == '*') {    //判断块注释
```

```
station2://临时判断是否是*/的情况
```

```

do {
    token_text[i++] = c;
    c = fgetc(fp);
} while (c != '\n' && c != EOF&&c!='*');
if (c == '*') {
    token_text[i++] = c;
    c = fgetc(fp);
    if (c == '/') {
        token_text[i++] = c;
        token_text[i] = '\0';
        return BLOCKNOTE;
    }
    else goto station2;
}
else if (c == '\n') {
    line_num++;
    goto station2;
}
else if (c == EOF) {
    goto stationERROR;
}
}
else {
    ungetc(c, fp);
    token_text[i] = '\0';
    return DIVIDE;
}

case '\':
    token_text[i++] = c;
    c = fgetc(fp);

```

```

if (c == '\\') { //判断 '\x' 的情况
    token_text[i++] = c;
    c = fgetc(fp);
    if (c >= '0' && c <= '9') { //判断'\数字'的情况
        do {
            token_text[i++] = c;
            c = fgetc(fp);
        } while (c >= '0' && c <= '9');
        if (c == '\\') {
            token_text[i++] = c;
            token_text[i] = '\\0';
            return CHAR_CONST;
        }
        else goto stationERROR;
    }
else if (c == '\\') { //判断'\''的情况
    token_text[i++] = c;
    c = fgetc(fp);
    if (c == '\\') {
        token_text[i++] = c;
        token_text[i] = '\\0';
        return CHAR_CONST;
    }
    else goto stationERROR;
}
else { //判断'\其他'的情况
    token_text[i++] = c;
    c = fgetc(fp);
    if (c == '\\') {
        token_text[i++] = c;
        token_text[i] = '\\0';
    }
}

```

```

        return CHAR_CONST;
    }

    else {
        do {
            token_text[i++] = c;
            c = fgetc(fp);
        } while (c != '\\' && c != '\n' && c != EOF); //添加智能识别功能，防止无限报错【亮点】

        if (c == '\\') {
            return ERROR_TOKEN;
        }

        else goto stationERROR;
    }
}

else if (c == '\\') { //判断''的情况
    c = fgetc(fp);
    if (c == '\\') return ERROR_TOKEN;
    else goto stationERROR;
}

else { //判断 'x' 的情况，x 不能为' 和\在上面讨论过
    token_text[i++] = c;
    c = fgetc(fp);
    if (c == '\\') {
        token_text[i++] = c;
        token_text[i] = '\0';
        return CHAR_CONST;
    }

    else {
        do {
            token_text[i++] = c;

```

```

        c = fgetc(fp);
    } while (c != '\\' && c != '\n' && c != EOF); //添加智能识别功能, 防止
无限报错【亮点】

```

```

        if (c == '\\') {
            return ERROR_TOKEN;
        }
        else goto stationERROR;
    }
}

```

case'":

```

    token_text[i++] = c;
    while ((c = fgetc(fp)) != '"') {
        token_text[i++] = c;
        if (c == '\\') {
            c = fgetc(fp);
            if (c == '"') { //识别字符串最后一个字符为'\\' 的错误情况【亮点】
                do {
                    token_text[i++] = c;
                    c = fgetc(fp);
                } while ((c != '"') && (c != '\n') && (c != EOF));
                if (c == '"') { token_text[i++] = c; token_text[i] == '\0'; return
STRING_CONST; }

                else goto stationERROR;
            }
            else if (c == '\n') { line_num++; token_text[i++] = c; }
            else token_text[i++] = c;
        }
    }
    else if (c == '\n') { //判断直接换行情况【亮点】
        line_num++;
        do {

```



```

        token_text[i++] = c;

        c = fgetc(fp);

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

    if (c == '"') { return ERROR_TOKEN; }

    else goto stationERROR;

}

else if (c == EOF) goto stationERROR;

}

token_text[i++] = c;

token_text[i] = '\0';

return STRING_CONST;

default:

    if (c == EOF) return EOF;

    else return ERROR_TOKEN;

} //switch 结束
}

```

• parser.cpp

```

#include "parser.h"

int w; //获得 gettoken 函数的返回值即读入的单词种类编码

char kind[100]; //储存类型关键字

char tokenText0[100]; //储存第一个函数名或者变量名

int indent0 = 0; //初始化缩进值

queue<print> printList; //用于方便打印缩进

status program(FILE* fp, CTree& T) //语法单位<程序>的子程序
{
    CTree c;

```

```

    struct print elem = { indent0, line_num };

    printList.push(elem); //存入程序的缩进值

    w = gettoken(fp);

    if (!ExtDefList(fp, c)) return ERROR; //调用外部定义序列函数

    T.n = 1; T.r = 0;

    T.nodes[0].data = (char*)malloc((strlen("程序") + 1) * sizeof(char)); //定义语法树的
    根结点 nodes[0]

    strcpy(T.nodes[0].data, "程序");

    T.nodes[0].indent = 0;

    T.nodes[0].firstchild = NULL;

    InsertChild(T, T.r, 1, c); //将 c 子树插入到语法树中

    return OK;
}

status ExtDefList(FILE* fp, CTree& T) //语法单位<外部定义序列>的子程序
{
    CTree c;

    status flag; //查看是否建立第二棵子树

    if (w == EOF) return INFEASIBLE;

    T.n = 1; T.r = 0;

    T.nodes[0].data = (char*)malloc((strlen("外部定义序列") + 1) * sizeof(char)); //创建
    外部定义序列树

    strcpy(T.nodes[0].data, "外部定义序列");

    T.nodes[0].indent = 0;

    T.nodes[0].firstchild = NULL;

    if (!ExtDef(fp, c)) return ERROR;

    InsertChild(T, T.r, 1, c); //处理一个外部定义, 得到一棵子树, 作为根的第一棵子树

    flag = ExtDefList(fp, c);

    if (flag == OK) InsertChild(T, T.r, 2, c); //得到的子树, 作为根的第二棵子树

    if (flag == ERROR) return ERROR;

    return OK;
}

```

```
}
```

```
status ExtDef(FILE* fp, CTree& T) //语法单位<外部定义>的子程序
```

```
{
    status flag;

    if (w != INT && w != LONG && w != SHORT && w != SIGNED && w != UNSIGNED &&
        w != FLOAT && w != DOUBLE && w != CHAR && w != VOID) return ERROR;

    strcpy(kind, token_text); //保存类型关键字

    w = gettoken(fp);

    if (w != IDENT) return ERROR;

    strcpy(tokenText0, token_text); //保存第一个变量名或函数名到 tokenText0

    w = gettoken(fp);

    if (w != LS) flag = ExtVarDef(fp, T); //调用外部变量定义子程序
    else flag = funcDef(fp, T); //调用函数定义子程序

    if (!flag) return ERROR;

    return OK;
}
```

```
status ExtVarDef(FILE* fp, CTree& T) //语法单位<外部变量定义>子程序
```

```
{
    CTree c; CTree p;

    T.n = 1; T.r = 0;

    T.nodes[0].data = (char*)malloc((strlen("外部变量定义: ") + 1) * sizeof(char)); //
生成外部变量定义结点

    strcpy(T.nodes[0].data, "外部变量定义: ");

    T.nodes[0].indent = 1;

    T.nodes[0].firstchild = NULL;

    c.n = 1; c.r = 0; //生成外部变量类型结点

    c.nodes[0].data = (char*)malloc((strlen(kind) + strlen("类型: ") + 1) * sizeof(char));

    strcpy(c.nodes[0].data, "类型: ");

    strcat(c.nodes[0].data, kind);
}
```

```

    c.nodes[0].indent = 1;

    c.nodes[0].firstchild = NULL;

    if (!InsertChild(T, T.r, 1, c)) return ERROR;           //c 作为 T 的第一个孩子

    if (!VarList(fp, p)) return ERROR;

    if (!InsertChild(T, T.r, 2, p)) return ERROR;           //p 作为 T 的第二个孩子

    return OK;
}

status VarList(FILE* fp, CTree& T) //语法单位<变量序列>子程序
{
    CTree c; CTree t; //c 树用来构建此变量结点, t 树用来构建可能存在的下一变量结点

    T.n = 1; T.r = 0; //生成变量序列结点

    T.nodes[0].data = (char*)malloc((strlen("变量序列") + 1) * sizeof(char));
    strcpy(T.nodes[0].data, "变量序列");

    T.nodes[0].indent = 0;

    T.nodes[0].firstchild = NULL;

    if (w == LM) //区分是数组变量还是变量
    {
        strcat(tokenText0, "["); //识别数组, 只支持一维数组

        if ((w = gettoken(fp)) == INT_CONST)
        {
            strcat(tokenText0, token_text);

            if ((w = gettoken(fp)) == RM)
            {
                strcat(tokenText0, "]"); //识别数组成功

                c.n = 1; c.r = 0; //创建数组结点

                c.nodes[0].data = (char*)malloc((strlen(tokenText0) + strlen("Array: ")
+ 1) * sizeof(char));

                strcpy(c.nodes[0].data, "Array: ");

                strcat(c.nodes[0].data, tokenText0); // 初始时, tokenText0 保存了第一个
变量名

```

```

        c.nodes[0].indent = 1;

        c.nodes[0].firstchild = NULL;

        w = gettoken(fp);

    }

    else return ERROR;

}

else return ERROR;

}

else

{

    c.n = 1; c.r = 0;  //创建变量结点

    c.nodes[0].data = (char*)malloc((strlen(tokenText0) + strlen("ID: ") + 1) *
sizeof(char));

    strcpy(c.nodes[0].data, "ID: ");

    strcat(c.nodes[0].data, tokenText0);

    c.nodes[0].indent = 1;

    c.nodes[0].firstchild = NULL;

}

if (!InsertChild(T, T.r, 1, c))    return ERROR;//识别的变量结点作为 T 的第一个孩子

if (w != COMMA && w != SEMI) return ERROR;

if (w == SEMI)                                //如果标识符后是分号，直接结束

{

    w = gettoken(fp);

    return OK;

}

w = gettoken(fp);

if (w != IDENT) return ERROR;  //如果 w 不是标识符则报错，反之后面还有第二个变量

strcpy(tokenText0, token_text);

w = gettoken(fp);

if (!VarList(fp, t)) return ERROR;

if (!InsertChild(T, T.r, 2, t)) return ERROR;

```

```

    return OK;
}

status funcDef(FILE* fp, CTree& T) //语法单位<函数定义>子程序
{
    CTree c; CTree p; //c 结点为函数返回值类型, p 结点为函数名结点

    CTree q; CTree f; CTree s; //q 结点为参数序列结点, f 为函数体结点, s 为可能的复合语句
    结点

    T.n = 1; T.r = 0; //生成函数定义结点 T

    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].indent = 1;
    T.nodes[0].firstchild = NULL;

    c.n = 1; c.r = 0; //生成函数返回值类型结点 c

    c.nodes[0].data = (char*)malloc((strlen(kind) + strlen("类型: ") + 1) * sizeof(char));
    strcpy(c.nodes[0].data, "类型: ");
    strcat(c.nodes[0].data, kind);
    c.nodes[0].indent = 1;
    c.nodes[0].firstchild = NULL;

    if (!InsertChild(T, T.r, 1, c)) return ERROR;

    w = gettoken(fp);

    //函数括号内可能无参数, 可能是 void, 可能是参数序列, 其他情况报错

    if (w != RS && w != VOID && w != INT && w != LONG && w != SHORT && w != SIGNED && w !=
    UNSIGNED && w != FLOAT && w != DOUBLE && w != CHAR)

        return ERROR;

    p.n = 1; p.r = 0; //生成函数名结点

    p.nodes[0].data = (char*)malloc((strlen(tokenText0) + strlen("函数名: ") + 1) *
    sizeof(char));

    strcpy(p.nodes[0].data, "函数名: ");
    strcat(p.nodes[0].data, tokenText0);

    p.nodes[0].indent = 1;

```

```

p.nodes[0].firstchild = NULL;

if (w == RS || w == VOID) //判断 void 情况或参数序列情况
{
    if (w == VOID)
    {
        w = gettoken(fp);

        if (w != RS) return ERROR;
    }
}

else
{
    if (!ParameList(fp, q)) return ERROR;

    if (!InsertChild(p, p.r, 1, q)) return ERROR;
}

if (!InsertChild(T, T.r, 2, p)) return ERROR;

w = gettoken(fp);

if (w != SEMI && w != LL) return ERROR;

f.n = 1; f.r = 0; //生成函数体结点

f.nodes[0].data = (char*)malloc((strlen("函数体: ") + 1) * sizeof(char));

strcpy(f.nodes[0].data, "函数体: ");

f.nodes[0].indent = 1;

f.nodes[0].firstchild = NULL;

if (w == LL) //存在函数体则判断复合语句
{
    if (!CompStat(fp, s)) return ERROR;

    if (!InsertChild(f, f.r, 1, s)) return ERROR;

    if(!InsertChild(T, T.r, 3, f))return ERROR;
}

//若是函数声明则直接返回 OK

return OK;
}

```

```

status ParamList(FILE* fp, CTree& T) //语法单位<形参序列>子程序
{
    CTree c; //c 用于生成形参子树
    CTree p; //p 用于生成可能出现的下一个形参序列
    T.n = 1; T.r = 0; //生成形参序列结点
    T.nodes[0].data = (char*)malloc((strlen("形参序列") + 1) * sizeof(char));
    strcpy(T.nodes[0].data, "形参序列");
    T.nodes[0].indent = 0;
    T.nodes[0].firstchild = NULL;
    if (!FormParDef(fp, c)) return ERROR;
    if (!InsertChild(T, T.r, 1, c)) return ERROR;
    w = gettoken(fp);
    if (w != RS && w != COMMA) return ERROR;
    if (w == COMMA)
    {
        w = gettoken(fp);
        if (!ParamList(fp, p)) return ERROR;
        InsertChild(T, T.r, 2, p);
    }
    return OK;
}

status FormParDef(FILE* fp, CTree& T) //语法单位<形参>子程序
{
    //w 此前已读取第一个形参类型
    CTree c; //用于生成形参类型结点
    CTree p; //用于生成形参变量结点
    T.n = 1; T.r = 0; //生成形参结点
    T.nodes[0].data = (char*)malloc((strlen("形参: ") + 1) * sizeof(char));
    strcpy(T.nodes[0].data, "形参: ");

```



```

T.nodes[0].indent = 1;

T.nodes[0].firstchild = NULL;

if (w != INT && w != LONG && w != SHORT && w != SIGNED && w != UNSIGNED &&
    w != FLOAT && w != DOUBLE && w != CHAR) return ERROR;

c.n = 1; c.r = 0;    //生成形参类型结点

c.nodes[0].data = (char*)malloc((strlen(token_text) + strlen("类型: ") + 1) *
sizeof(char));

strcpy(c.nodes[0].data, "类型: ");

strcat(c.nodes[0].data, token_text);

c.nodes[0].indent = 1;

c.nodes[0].firstchild = NULL;

InsertChild(T, T.r, 1, c);

w = gettoken(fp);

if (w != IDENT) return ERROR;

p.n = 1; p.r = 0;    //生成形参变量结点

p.nodes[0].data = (char*)malloc((strlen(token_text) + strlen("ID: ") + 1) *
sizeof(char));

strcpy(p.nodes[0].data, "ID: ");

strcat(p.nodes[0].data, token_text);

p.nodes[0].indent = 1;

p.nodes[0].firstchild = NULL;

InsertChild(T, T.r, 2, p);

return OK;
}

status CompStat(FILE* fp, CTree& T) //语法单位<复合语句>子程序
{
    CTree c; CTree p; //c 用于生成局部变量定义子树, p 用于生成语句序列子树

    status flag;

    print elem;

    T.n = 1; T.r = 0;    //生成复合语句结点

```

```

T.nodes[0].data = (char*)malloc((strlen("复合语句: ") + 1) * sizeof(char));
strcpy(T.nodes[0].data, "复合语句: ");
T.nodes[0].indent = 1;
T.nodes[0].firstchild = NULL;
//注意其中局部变量说明和语句序列都可以为空
w = gettoken(fp);
elem = { ++indent0, line_num };
printList.push(elem); //添加缩进值
if (w == INT || w == LONG || w == SHORT || w == SIGNED || w == UNSIGNED || w == FLOAT
|| w == DOUBLE || w == CHAR)
{
    if (!LocVarList(fp, c)) return ERROR;
    if (!InsertChild(T, T.r, 1, c)) return ERROR;
    flag = StatList(fp, p);
    if (!flag) return ERROR;
    if (flag == OK)
        if (!InsertChild(T, T.r, 2, p)) return ERROR;
}
else
{
    flag = StatList(fp, p);
    if (!flag) return ERROR;
    if (flag == OK)
        if (!InsertChild(T, T.r, 1, p)) return ERROR;
}
elem = { --indent0, line_num };
printList.push(elem);
if (w != RL) return ERROR;
w = gettoken(fp);
return OK;
}

```

```

status LocVarList(FILE* fp, CTree& T) //语法单位<局部变量定义序列>子程序
{
    CTree c; CTree p; //c 生成局部变量定义子树, p 生成可能存在的下一个局部变量定义序列子树

    status flag;

    if (w != INT && w != LONG && w != SHORT && w != SIGNED && w != UNSIGNED && w != FLOAT
&& w != DOUBLE && w != CHAR)

        return INFEASIBLE;

    //读到的后继单词不为类型说明符时, 变量定义序列结束

    T.n = 1; T.r = 0; //生成局部变量定义序列结点

    T.nodes[0].data = (char*)malloc((strlen("局部变量定义序列") + 1) * sizeof(char));

    strcpy(T.nodes[0].data, "局部变量定义序列");

    T.nodes[0].indent = 0;

    T.nodes[0].firstchild = NULL;

    if (!LocVarDef(fp, c)) return ERROR;

    if (!InsertChild(T, T.r, 1, c)) return ERROR;

    flag = LocVarList(fp, p);

    if (flag == OK) InsertChild(T, T.r, 2, p);

    if (!flag) return ERROR;

    return OK;
}

```

```

status LocVarDef(FILE* fp, CTree& T) //语法单位<局部变量定义>子程序
{
    CTree c; CTree p; //c 生成局部变量类型结点, p 生成变量序列子树

    if (w != INT && w != LONG && w != SHORT && w != SIGNED && w != UNSIGNED && w != FLOAT
&& w != DOUBLE && w != CHAR) return ERROR;

    T.n = 1; T.r = 0; //生成局部变量定义结点

    T.nodes[0].data = (char*)malloc((strlen("局部变量定义: ") + 1) * sizeof(char));

    strcpy(T.nodes[0].data, "局部变量定义: ");

    T.nodes[0].indent = 1;

```

```

T.nodes[0].firstchild = NULL;

c.n = 1; c.r = 0;      //生成局部变量类型结点

c.nodes[0].data = (char*)malloc((strlen(token_text) + strlen("类型: ") + 1) *
sizeof(char));

strcpy(c.nodes[0].data, "类型: ");
strcat(c.nodes[0].data, token_text);

c.nodes[0].indent = 1;
c.nodes[0].firstchild = NULL;
if (!InsertChild(T, T.r, 1, c)) return ERROR;

w = gettoken(fp);
if (w != IDENT) return ERROR;
strcpy(tokenText0, token_text);
w = gettoken(fp);
if (!VarList(fp, p)) return ERROR;
if (!InsertChild(T, T.r, 2, p)) return ERROR;

return OK;
}

status StatList(FILE* fp, CTree& T) //语法单位<语句序列>子程序
{
    CTree c; CTree p; //c 生成语句树, p 生成可能出现的语句序列树

    status flag;

    flag = Statement(fp, c);
    if (flag == INFEASIBLE) return INFEASIBLE;
    if (!flag) return ERROR;
    else
    {
        T.n = 1; T.r = 0;      //生成语句序列结点

        T.nodes[0].data = (char*)malloc((strlen("语句序列") + 1) * sizeof(char));
        strcpy(T.nodes[0].data, "语句序列");
    }
}

```

```

    T.nodes[0].indent = 0;

    T.nodes[0].firstchild = NULL;

    InsertChild(T, T.r, 1, c);

    flag = StatList(fp, p);

    if (!flag) return ERROR;

    if (flag == OK)

        if (!InsertChild(T, T.r, 2, p))

            return ERROR;

    }

    return OK;
}

status Statement(FILE* fp, CTree& T) //语法单位<语句>子程序
{

    CTree c; CTree p; CTree q; CTree k;

    print elem;

    if (w == IF) { //分析条件语句, p 用于生成表达式树, q 用于生成 if 模块子句数, k 用于生成 else
        模块子句数

        w = gettoken(fp);

        if (w != LS) return ERROR;

        w = gettoken(fp);

        if (w == RS) return ERROR;

        if (!exp(fp, p, RS)) return ERROR;

        c.n = 1; c.r = 0; //生成 if 语句子树

        c.nodes[0].data = (char*)malloc((strlen("条件: ") + 1) * sizeof(char));

        strcpy(c.nodes[0].data, "条件: ");

        c.nodes[0].indent = 1;

        c.nodes[0].firstchild = NULL;

        InsertChild(c, c.r, 1, p);

        w = gettoken(fp);
    }
}

```

```

    if (w == LL)
    {
        if (!CompStat(fp, q)) return ERROR;
    }
else
{
    elem = { ++indent0, line_num };
    printList.push(elem);
    if (!Statement(fp, q)) return ERROR;
    elem = { --indent0, line_num };
    printList.push(elem);
}

p.n = 1; p.r = 0;
p.nodes[0].data = (char*)malloc((strlen("IF 子句: ") + 1) * sizeof(char));
strcpy(p.nodes[0].data, "IF 子句: ");
p.nodes[0].indent = 1;
p.nodes[0].firstchild = NULL;
InsertChild(p, p.r, 1, q);
if (w == ELSE)
{
    w = gettoken(fp);
    if (w == LL)
    {
        if (!CompStat(fp, k)) return ERROR;
    }
else
{
    elem = { ++indent0, line_num };
    printList.push(elem);
    if (!Statement(fp, k)) return ERROR;
    elem = { --indent0, line_num };

```

```

        printList.push(elem);
    }

    q.n = 1; q.r = 0; //生成 else 语句子树

    q.nodes[0].data = (char*)malloc((strlen("ELSE 子句: ") + 1) * sizeof(char));
    strcpy(q.nodes[0].data, "ELSE 子句: ");
    q.nodes[0].indent = 1;
    q.nodes[0].firstchild = NULL;

    InsertChild(q, q.r, 1, k);

    T.n = 1; T.r = 0;

    T.nodes[0].data = (char*)malloc((strlen("if-else 语句: ") + 1) *
sizeof(char));

    strcpy(T.nodes[0].data, "if-else 语句: ");
    T.nodes[0].indent = 1;
    T.nodes[0].firstchild = NULL;

    InsertChild(T, T.r, 1, c);
    InsertChild(T, T.r, 2, p);
    InsertChild(T, T.r, 3, q);
}

else
{
    T.n = 1; T.r = 0;

    T.nodes[0].data = (char*)malloc((strlen("if 语句: ") + 1) * sizeof(char));
    strcpy(T.nodes[0].data, "if 语句: ");
    T.nodes[0].indent = 1;
    T.nodes[0].firstchild = NULL;

    InsertChild(T, T.r, 1, c);
    InsertChild(T, T.r, 2, p);
}

return OK;
}

else if (w == LL) {

```

```

    if (!CompStat(fp, T)) return ERROR;

    return OK;
}

else if (w == FOR) {
    //分析 for 语句

    T.n = 1; T.r = 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].indent = 1;

    T.nodes[0].firstchild = NULL;

    c.n = 1; c.r = 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].indent = 1;

    c.nodes[0].firstchild = NULL;

    InsertChild(T, T.r, 1, c);

    c.n = 1; c.r = 0;

    c.nodes[0].data = (char*)malloc((strlen("终止条件: ") + 1) * sizeof(char));
    strcpy(c.nodes[0].data, "终止条件: ");

    c.nodes[0].indent = 1;

    c.nodes[0].firstchild = NULL;

    InsertChild(T, T.r, 2, c);

    c.n = 1; c.r = 0;

    c.nodes[0].data = (char*)malloc((strlen("循环表达式: ") + 1) * sizeof(char));
    strcpy(c.nodes[0].data, "循环表达式: ");

    c.nodes[0].indent = 1;

    c.nodes[0].firstchild = NULL;

    InsertChild(T, T.r, 3, c);

    c.n = 1; c.r = 0;

    c.nodes[0].data = (char*)malloc((strlen("for 子句: ") + 1) * sizeof(char));
    strcpy(c.nodes[0].data, "for 子句: ");

```



```

    c.nodes[0].indent = 1;

    c.nodes[0].firstchild = NULL;

    InsertChild(T, T.r, 4, c);

    w = gettoken(fp);

    if (w != LS) return ERROR;

    w = gettoken(fp);

    if (!exp(fp, c, SEMI)) return ERROR;

    InsertChild(T, T.nodes[0].firstchild->child, 1, c);

    w = gettoken(fp);

    if (w == SEMI) return ERROR;

    if (!exp(fp, c, SEMI)) return ERROR;

    InsertChild(T, T.nodes[0].firstchild->next->child, 1, c);

    w = gettoken(fp);

    if (!exp(fp, c, RS)) return ERROR;

    InsertChild(T, T.nodes[0].firstchild->next->next->child, 1, c);

    w = gettoken(fp);

    if (w == LL)
    {
        if (!CompStat(fp, c)) return ERROR;
    }
    else
    {
        elem = { ++indent0, line_num };

        printList.push(elem);

        if (!Statement(fp, c)) return ERROR;

        elem = { --indent0, line_num };

        printList.push(elem);
    }

    InsertChild(T, T.nodes[0].firstchild->next->next->next->child, 1, c);

    return OK;
}

```

```

else if (w == WHILE) {
    w = gettoken(fp);
    if (w != LS) return ERROR;
    w = gettoken(fp);
    if (w == RS) return ERROR;
    if (!exp(fp, c, RS)) return ERROR;
    w = gettoken(fp);
    if (w == LL)
    {
        if (!CompStat(fp, p)) return ERROR;
    }
    else
    {
        elem = { ++indent0, line_num };
        printList.push(elem);
        if (!Statement(fp, p)) return ERROR;
        elem = { --indent0, line_num };
        printList.push(elem);
    }
    T.n = 1; T.r = 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].indent = 1;
    T.nodes[0].firstchild = NULL;
    InsertChild(T, T.r, 1, c);
    InsertChild(T, T.r, 2, p);
    return OK;
}

else if (w == CONTINUE) {
    T.n = 1; T.r = 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].indent = 1;

    T.nodes[0].firstchild = NULL;

    w = gettoken(fp);

    if (w != SEMI) return ERROR;

    w = gettoken(fp);

    return OK;
}

else if (w == BREAK) {
    T.n = 1; T.r = 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].indent = 1;

    T.nodes[0].firstchild = NULL;

    w = gettoken(fp);

    if (w != SEMI) return ERROR;

    w = gettoken(fp);

    return OK;
}

else if (w == RETURN) {
    T.n = 1; T.r = 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].indent = 1;

    T.nodes[0].firstchild = NULL;

    w = gettoken(fp);

    if (w == SEMI) return ERROR;

    if (!exp(fp, c, SEMI)) return ERROR;

    w = gettoken(fp);

    InsertChild(T, T.r, 1, c);

    return OK;
}

```

```

    }

    else if (w == LS) {

        if (!exp(fp, T, RS)) return ERROR;

        w = gettoken(fp);

        return OK;

    }

    else if (w == IDENT || w == INT_CONST || w == UNSIGNED_CONST || w == LONG_CONST || w
== UNSIGNED_LONG_CONST || w == DOUBLE_CONST || w == FLOAT_CONST || w == LONG_DOUBLE_CONST
|| w == CHAR_CONST) {

        if (!exp(fp, T, SEMI)) return ERROR;

        w = gettoken(fp);

        return OK;

    }

    else if (w == RL) {

        return INFEASIBLE;

    }

    else if (w == SEMI) {

        T.n = 1; T.r = 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].indent = 1;

        T.nodes[0].firstchild = NULL;

        w = gettoken(fp);

        return OK;

    }

    else return ERROR;

}

status exp(FILE* fp, CTree& T, int endsym)//语法单位<表达式>子程序
{

```

```

//已经读入了第一个单词在 w 中

SqStack op;          //运算符栈

SqStack opn; //操作数栈

CTree* node = (CTree*)malloc(sizeof(CTree));

CTree* child1 = (CTree*)malloc(sizeof(CTree));

CTree* child2 = (CTree*)malloc(sizeof(CTree));

T.n = 1; T.r = 0; //生成表达式结点

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].indent = 1;

T.nodes[0].firstchild = NULL;

int error = 0;

node->n = 1; node->r = 0; //设立起止符号

node->nodes[0].data = (char*)malloc((strlen("#") + 1) * sizeof(char));

strcpy(node->nodes[0].data, "#");

node->nodes[0].firstchild = NULL;

InitStack(opn); InitStack(op); Push(op, node); //初始化, 将起止符#入栈

while ((w != POUND || strcmp(node->nodes[0].data, "#")) && !error) //当运算符栈栈顶
不是起止符号, 并没有错误时
{
    if (w == IDENT || w == INT_CONST || w == UNSIGNED_CONST || w == LONG_CONST
        || w == UNSIGNED_LONG_CONST || w == DOUBLE_CONST || w == FLOAT_CONST
        || w == LONG_DOUBLE_CONST || w == CHAR_CONST)
    {
        node = (CTree*)malloc(sizeof(CTree));

        node->n = 1; node->r = 0;

        node->nodes[0].data = (char*)malloc((strlen(token_text) + strlen("ID: ") + 1)
* sizeof(char));

        strcpy(node->nodes[0].data, "ID: ");

        strcat(node->nodes[0].data, token_text);

        node->nodes[0].indent = 1;

```

```

        node->nodes[0].firstchild = NULL;

        Push(opn, node);          //根据 w 生成一个结点, 结点指针进栈 opn

        w = gettoken(fp);

    }

    else if (w == PLUS || w == MINUS || w == MULTIPLY || w == DIVIDE ||

        w == MOD || w == LS || w == RS || ((w>=MORE)&&(w<= LESS_EQUAL)) || w == EQUAL_TO

||

        w == AND || w == OR || w == POUND)

    {

        node = (CTree*)malloc(sizeof(CTree));

        GetTop(op, node);

        if (w == POUND) strcpy(token_text, "#");

        switch (precede(node->nodes[0].data, token_text))

        {

        case '<':

            node = (CTree*)malloc(sizeof(CTree));

            node->nodes[0].data = (char*)malloc((strlen(token_text) + 1) *

sizeof(char));

            strcpy(node->nodes[0].data, token_text);

            node->nodes[0].indent = 1;

            node->nodes[0].firstchild = NULL;

            node->n = 1; node->r = 0;

            Push(op, node);          //根据 w 生成一个结点, 结点指针进栈 opn

            w = gettoken(fp);

            break;

        case '=':

            if (!Pop(op, node)) error++;

            w = gettoken(fp);

            break;    //去括号

        case '>':

            if (!Pop(opn, child2)) error++;

```

```

        if (!Pop(opn, child1)) error++;

        if (!Pop(op, node)) error++;

        //根据运算符栈退栈得到的运算符 node 和操作数的结点指针 child1 和 child2,
        //完成建立生成一个运算符的结点, 结点指针进栈 opn

        InsertChild(*node, node->r, 1, *child1);

        InsertChild(*node, node->r, 2, *child2);

        Push(opn, node);

        break;

    default:

        if (w == endsym) w = POUND; //遇到结束标记), w 被替换成#

        else error++;

    }

}

else if (w == endsym) w = POUND; //遇到结束标记分号, w 被替换成#

else error = 1;

GetTop(op, node);

}

if (error) return ERROR;

GetTop(opn, node);

InsertChild(T, T.r, 1, *node);

return OK;

}

char precede(char* a, char* b)

{

    int i, j;          //指示运算符对应的编号

    //定义一个二维数组, 用于存放优先级

    char precede[13][13] =

    { //          +      -      *      /      %      (      )

    =      >和<  ==和!=  #      &&      ||

    /* + */      '>', '>', '<', '<', '<', '<', '>', '?', '>',      '>',      '>', '>',


```

```

'>',
/* - */      '>', '>', '<', '<', '<', '<', '>', '?', '>',      '>',      '>', '>',
'>',
/* * */      '>', '>', '>', '>', '>', '<', '>', '?', '>',      '>',      '>', '>',
'>',
/* / */      '>', '>', '>', '>', '>', '<', '>', '?', '>',      '>',      '>', '>',
'>',
/* % */      '>', '>', '<', '<', '<', '<', '>', '?', '>',      '>',      '>', '>',
'>',
/* ( */      '<', '<', '<', '<', '<', '<', '=', '?', '<',      '<',      '>', '<',
'<',
/* ) */      '>', '>', '>', '>', '>', '>', '?', '?', '>',      '>',      '>', '>',
'>',
/* = */      '<', '<', '<', '<', '<', '<', '?', '<', '<',      '<',      '>', '<',
'<',
/* >和< */   '<', '<', '<', '<', '<', '<', '>', '?', '>',      '>',      '>', '>',
'>',
/* ==和!= */ '<', '<', '<', '<', '<', '<', '>', '?', '<',      '>',      '>', '>',
'>',
/* # */      '<', '<', '<', '<', '<', '<', '?', '<', '<',      '<',      '=', '<',
'<',
/* && */      '<', '<', '<', '<', '<', '<', '>', '>', '<',      '<',      '>', '>',
'>',
/* || */      '<', '<', '<', '<', '<', '<', '>', '>', '<',      '<',      '>', '>', '>'
};

switch (a[0])
{
case '+':
    i = 0; break;
case '-':
    i = 1; break;

```



```
case '*':  
    i = 2; break;  
case '/':  
    i = 3; break;  
case '%':  
    i = 4; break;  
case '(':  
    i = 5; break;  
case ')':  
    i = 6; break;  
case '=':  
    if (a[1] == '=') i = 9;  
    else i = 7;  
    break;  
case '>':  
case '<':  
    i = 8; break;  
case '!':  
    if (a[1] == '=') i = 9;  
    else return '?';  
    break;  
case '#':  
    i = 10;  
    break;  
case '&':  
    if (a[1] == '&') i = 11;  
    else return '?';  
    break;  
case '|':  
    if (a[1] == '|') i = 12;  
    else return '?';
```

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

```

    case '#':
        j = 10;
        break;
    case '&':
        if (b[1] == '&') j = 11;
        else return '?';
        break;
    case '|':
        if (b[1] == '|') j = 12;
        else return '?';
        break;
    default:
        return '?';
}

return precede[i][j];
}

```

```

status PrintTree(char* data, int indent) //打印函数

```

```

{
    int i;
    for (i = 0; i < indent; i++)
        printf("\t");
    printf("%s\n", data);
    return OK;
}

```

• preprocess.cpp

```

#include "preprocess.h"
#include "lexer.h"
define_data data_Def[10]; //用于储存 define 宏定义的内容，全局
include_data data_Includ[10]; //用于储存 include 文件包含的内容，全局
int data_Def_num; //宏定义个数

status pre_process(FILE* fp) {

```

```

int w; //接受 gettoken 读取的返回值
int i=0, j=0, m; //i 是宏定义个数, j 是 include 个数
int pre_line_num=1; //用于记录换行情况
char container; //暂时存储字符判断结尾处的分号
int a, b; //比较行数
int flag=0; //判断语句中是否出现 define 的定义
FILE* mid_fp;
char filename[50];
strcpy(filename, "C_mid_file.txt"); //中间文件
mid_fp = fopen(filename, "w");
w = gettoken(fp);
do {
    if (w == POUND) {
        w = gettoken(fp);
        if (w == DEFINE) {
            w = gettoken(fp);
            a = line_num;
            if (w == ERROR_TOKEN) return ERROR;
            else if (w == SEMI) return ERROR;
            else if (w == POUND) return ERROR;
            else {
                strcpy(data_Def[i].ident, token_text);
            }
            w = gettoken(fp);
            b = line_num;
            if (w == ERROR_TOKEN) return ERROR;
            else if (w == SEMI) return ERROR;
            else if (w == POUND) return ERROR;
            else {
                strcpy(data_Def[i++].string, token_text);
            }
            if (a != b) return ERROR;
            data_Def_num = i;
            fprintf(mid_fp, "\n");
            w = gettoken(fp);
            pre_line_num = line_num;
            continue;
        }
        else if (w == INCLUDE) {
            w = gettoken(fp);
            if (w == ERROR_TOKEN) return ERROR;
            else if (w == STRING_CONST) {
                strcpy(data_Includ[j++].string, token_text);
                if ((container = fgetc(fp)) != ';' ) { ungetc(container, fp);
                fprintf(mid_fp, "\n"); w = gettoken(fp); pre_line_num = line_num; continue; }
            }
        }
    }
} while (w != POUND);

```

```

        else return ERROR;
    }
    else if (w == LESS) {
        w = gettoken(fp);
        a = line_num;
        if (w != IDENT) return ERROR;
        else {
            w = gettoken(fp);
            b = line_num;
            if (w != MORE) return ERROR;
            if (a != b) return ERROR;
            if ((container = fgetc(fp)) != ';'') { ungetc(container, fp);
fprintf(mid_fp, "\n"); w = gettoken(fp); pre_line_num = line_num; continue; }
            else return ERROR;
        }
    }
}
else return ERROR;
}
data_Def_num = i;

if (w != POUND) {

    if (w == IDENT) { //是标识符时，判断是不是 define 的类型
        for (m = 0; m < data_Def_num; m++) {
            if (!strcmp(token_text, data_Def[m].ident)) {
                if (pre_line_num != line_num) {
                    fprintf(mid_fp, "\n%s ", data_Def[m].string);
                    flag = 1;
                }
                else { fprintf(mid_fp, "%s ", data_Def[m].string); flag = 1; }
            }
        }
        if (flag != 0) {
            flag = 0;
        }
        else {
            if (pre_line_num != line_num) {
                fprintf(mid_fp, "\n%s ", token_text);
            }
            else { fprintf(mid_fp, "%s ", token_text); }
        }
    }
    else if (w == LINENOTE) {
        pre_line_num = line_num;
    }
}

```

```

        w = gettoken(fp);
        continue;
    }
    else if (w == BLOCKNOTE) {

        for(m=0;m<line_num-pre_line_num;m++) fprintf(mid_fp, "\n");
        pre_line_num = line_num;
        w = gettoken(fp);
        continue;
    }
    else if (w == ERROR_TOKEN) {
        if (pre_line_num != line_num) {
            fprintf(mid_fp, "\n%s", token_text);
        }
        else { fprintf(mid_fp, "%s", token_text); }
    }
    else {
        if (pre_line_num != line_num) {
            fprintf(mid_fp, "\n%s ", token_text);
        }
        else { fprintf(mid_fp, "%s ", token_text); }
    }
}
pre_line_num = line_num;

w = gettoken(fp);
} while (w != EOF);
fclose(mid_fp);
return OK;
}

```

• printfile.cpp

```

#include "printfile.h"

status PrintFile(FILE* fp) {
    int indentnum=0, line = 1;
    int i; char c;
    FILE* print_fp;
    char filename[30] = "C_print_file.txt";
    print_fp = fopen(filename, "w");
    while (!printList.empty())
    {
        indentnum = printList.front().indent;
        printList.pop();
    }
}

```

```

while (!printList.empty() && line == printList.front().linenum) printList.pop();
while (!printList.empty() && line < printList.front().linenum)
{
    for (i = 0; i < indentnum; i++)
        fputc('\t', print_fp);
    while ((c = fgetc(fp)) != '\n')
        fputc(c, print_fp);
    fputc(c, print_fp);
    line++;
}
}
while ((c = fgetc(fp)) != EOF)
    fputc(c, print_fp);
fclose(print_fp);
return OK;
}

```

• profuction.cpp

```
#include "profuction.h"
```

```
/*树的相关函数*/
```

```
status InitTree(CTree& T) //初始化树
```

```

{
    T.n = 0; T.r = -1;    //将根的位置设为-1, 说明当前树没有根
    return OK;
}

```

```
status InsertChild(CTree& T, int p, int i, CTree c) { //插入孩子, 在 p 结点处插入孩子子
树 c, i 表示第 i 棵子树
```

```

    if (p < 0 || p >= T.n) return ERROR;
    if (!c.n) return ERROR;
    int k = 0, j = T.n;
    CNode prior = T.nodes[p].firstchild;
    CNode cur = prior;
    CNode t;
    while (k < c.n) //添加子树
    {
        T.nodes[j] = c.nodes[k++];
        for (t = T.nodes[j].firstchild; t; t = t->next)
            t->child += T.n;
        j++;
    }
    t = (CNode)malloc(sizeof(CTNode));

```

```

t->child = T.n + c.r; //子树根的位置
t->next = NULL;
T.n += c.n;
if (i == 1)
{
    if (prior)
    {
        t->next = prior;
        prior = t;
    }
    else prior = t;
    T.nodes[p].firstchild = prior;
}
else if (i < 1) return ERROR; //如果 i 输入非法，返回 ERROR
else
{
    k = 1; //k 表示当前是第几棵子树
    while (k < i && cur)
    {
        k++;
        prior = cur;
        cur = cur->next;
    }
    if (cur)
    {
        prior->next = t;
        t->next = cur;
    }
    else if (k == i)
    {
        prior->next = t;
        t->next = NULL;
    }
    else return ERROR;
}
return OK;
}

status GetParent(CTree T, int child, int& parent)
{
    CNode t; int i = 0;
    if (child >= T.n) return ERROR;
    for (; i < T.n; i++)
    {
        t = T.nodes[i].firstchild;
    }
}

```



```

        while (t && t->child != child)
            t = t->next;
        if (t && t->child == child)
        {
            parent = i;
            return OK;
        }
    }
    return INFEASIBLE;
}

status TraverseTree(CTree T, status(*visit)(char*, int))
{
    stack<int> stack;
    int status = 0; //用来标志是否所有孩子都被访问过
    int i; int j;
    int indent = 0; //缩进量
    int parent;
    CNode t;
    int visited[100] = { 0 }; //访问标志数组初始化
    if (!T.n) return OK; //树为空直接结束
    for (i = 0; i < T.n; i++)
    {
        indent = 0;
        if (!visited[i]) //对尚未访问的结点进行访问
        {
            stack.push(i);
            visited[i] = 1;
            if (GetParent(T, i, parent) == OK)
                indent += T.nodes[parent].indent;
            if (T.nodes[i].indent)
                visit(T.nodes[i].data, indent);
            t = T.nodes[i].firstchild;
            while (t && !status) //如果该结点有孩
子
            {
                while (visited[t->child] && t->next) //寻找未被访问过的孩子
                    t = t->next;
                if (!visited[t->child])
                {
                    visited[t->child] = 1;
                    if (T.nodes[t->child].firstchild)
                        stack.push(t->child);
                    GetParent(T, t->child, parent);
                    indent += T.nodes[parent].indent;

```

```

        if (T.nodes[t->child].indent)
            visit(T.nodes[t->child].data, indent);
        t = T.nodes[t->child].firstchild;
        status = 0;
    }
    else status = 1;
}
}
while (!stack.empty())                //循环到栈为空
{
    j = stack.top();
    indent -= T.nodes[j].indent;
    t = T.nodes[j].firstchild;
    status = 0;
    while (t && !status)                //如果该结点有孩子
    {
        while (visited[t->child] && t->next)    //寻找未被访问过的孩子
            t = t->next;
        if (!visited[t->child])
        {
            visited[t->child] = 1;
            if (T.nodes[t->child].firstchild)
                stack.push(t->child);
            GetParent(T, t->child, parent);
            indent += T.nodes[parent].indent;
            if (T.nodes[t->child].indent)
                visit(T.nodes[t->child].data, indent);
            t = T.nodes[t->child].firstchild;
            status = 0;
        }
        else status = 1;
    }
    if (t) stack.pop();
}
return OK;
}

/*栈的相关函数*/
status InitStack(SqStack& S)
{
    //构造一个空栈 S
    S.base = (SElemType*)malloc(100 * sizeof(SElemType));
    if (!S.base) exit(OVERFLOW);    //存储分配失败
    S.top = S.base;
}

```

```

    S.stacksize = 100;
    return OK;
}

status GetTop(SqStack S, SElemType& e)
{
    //若栈不空, 则用 e 返回 S 的栈顶元素, 并返回 OK; 否则返回 ERROR
    if (S.top == S.base) return ERROR;
    e = *(S.top - 1);
    return OK;
}

status Push(SqStack& S, SElemType e)
{
    //插入元素 e 为新的栈顶元素
    if (S.top - S.base >= S.stacksize)
    {
        //栈满, 追加存储空间
        S.base = (SElemType*)realloc(S.base, (S.stacksize + 100) * sizeof(SElemType));
        if (!S.base) exit(OVERFLOW); //存储分配失败
        S.top = S.base + S.stacksize;
        S.stacksize += 100;
    }
    *S.top++ = e;
    return OK;
}

status Pop(SqStack& S, SElemType& e)
{
    //若栈不空, 则删除 S 的栈顶元素, 用 e 返回其值, 并返回 OK
    if (S.top == S.base) return ERROR;
    e = *(--S.top);
    return OK;
}

```