華中科技大學

课程设计报告

题目：基于高级语言源程序格式处理工具

课程名称：程序设计综合课程设计
专业班级：CS2106
学　　号：U202115513
姓　　名：虞快
指导教师：纪俊文
报告日期：2022.9.30

计算机科学与技术学院

目录

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。之所以说是“抽象”，是因为在抽象语法树中，忽略了源程序中语法成分的一些细节，突出了其主要语法特征。

抽象语法树(Abstract Syntax Tree ,AST)作为程序的一种中间表示形式,在程序分析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。

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

□ 设计要求

要求具有如下功能：

1. **语言定义**

选定 C 语言的一个子集，要求包含：

（1）基本数据类型的变量、常量，以及数组。不包含指针、结构，枚举等。

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

（3）函数定义、声明与调用。

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

（5）编译预处理（宏定义，文件包含）

（6）注释（块注释与行注释）

2. **单词识别**

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

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**。

口 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

1 引言

1.1 课题背景与意义

在计算机科学中，编译原理是一个重要组成部分，旨在介绍编译程序构造的一般原理和基本方法。内容包括语言和文法、词法分析、语法分析、语法制导翻译、中间代码生成、存储管理、代码优化和目标代码生成。

编译原理的重要首先是由其内容决定的。根据名字，编译原理包括编译和原理两大部分内容，编译是其实践部分，原理是其理论部分。根据内容，编译原理可以分为前端和后端，前端是对程序语言到中间代码的转换，后端是中间代码到机器代码的转换，前端偏理论，后端偏实践。

实践部分的内容是编译器及其开发，它不仅是计算机科学中的各种知识和理论的练兵场，还具有一个其它系统开发所没有的特征，它的开发所依赖的和所要处理的都是计算机科学中的关键——程序语言。编译器作为一种软件系统，需要用软件语言来书写、构建和运行，语言又是编译器的处理对象，编译器的输入和输出又都是语言。也就是说语言既是编译器的用户，又是编译器的工具，编译器的开发需要开发人员截然不同的两种视角来考虑权衡，这是其它系统所不具备的特征(如果把解释器看作编译器之外的，解释器也具有这个特征，但解释器更偏重局部)。特别是编译器后端的优化部分，能让开发人员深刻理解语言设计的原因和结果。

本次课程设计将以 C 语言语法元素的子集为例，通过抽象语法树的语法结构实现编译原理中的词法分析和语法分析，并在两者的基础上实现源程序格式化处理的功能。

1.2 国内外研究现状

1.2.1 抽象语法树（AST）

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

成分的一些细节，突出了其主要语法特征。

抽象语法树(Abstract Syntax Tree ,AST)作为程序的一种中间表示形式,在程序分析等诸多领域有广泛的应用.利用抽象语法树可以方便地实现多种源程序处理工具,比如源程序浏览器、智能编辑器、语言翻译器等。

1.2.2 形式语言自动机

定义（不确定的有限自动机）NFA M 是一个五元组：M=（Σ，Q，δ，q0，F）。其中Σ是输入符号的有穷集合；Q 是状态的有限集合；q0∈Q 是初始状态；F 是终止状态集合，F⊆Q；δ是 Q 与Σ的直积 Q×Σ到 Q 的幂集 2Q 的映射。

NFA 与 DFA 的重要区别是：在 NFA 中δ(q,a)是一个状态集合，而在 DFA 中δ(q,a)是一个状态。根据定义,对于 NFA M 有映射：δ(q,a)={q1,q2,…,qk},k>=1。即 NFA M 在状态 q 时，接受输入符号 a 时，M 可以选择状态集 q1,q2,…,qk 中任何一个状态作为下一个状态，并将输入头向右边移动一个字符的位置。

定义（NFA 接受的语言）如果存在一个状态 p，有 p∈δ(q0,x)且 p∈F，则称句子 x 被 NFA M 所接受。被 NFA M 接受的所有句子的集合称为 NFA M 定义的语言，记作 T(M)={x ｜ p∈δ(q0,x) 且 p∈F}。

定理：设 L 是被 NFA 所接受的语言，则存在一个 DFA，它能够接受 L。

1.3 课程设计的主要研究工作

1. 完成词法分析功能，分析源程序中各个元素并列表展示。
2. 完成语法分析功能，生成语法树并展示。
3. 在语法分析的基础上完成报错的功能。
4. 在前面的基础上进行源程序格式的编排并以.c 文件保存在同一目录下。

2 系统需求分析与总体设计

2.1 系统需求分析

1. 词法分析：给定一段源程序，源程序中所有合法的 token 列出，并且在遇到错误 token 时抛出异常。

2. 语法分析：根据源程序内在的语法逻辑结构，生成抽象语法树以显示源程序的逻辑，并在语法错误时抛出异常。

3. 源程序格式化：按照 c 语言代码缩进规范编排源程序。

2.2 系统总体设计

根据目标和需求，程序主要分为三个功能模块：词法分析、语法分析和格式化源程序。词法分析只要根据形式语言自动机的知识，根据读取符号的情况跳转到不同的状态，从而获取每个语素即可。语法分析需要通过递归下降子程序法生成抽象语法树，最终通过先根遍历的方式显示程序结构。源程序格式处理在前两个功能的基础上按照一定的缩进范式编排源程序即可。

词法分析

词法分析，即要求把源程序中所有合法的 token 列出，并且在遇到错误 token 时抛出异常。

根据有限状态机(DNF)理论，可以依次读取输入流中的每个字符，当读取过程中满足一定的状态转移条件时，就可以跳转到对应的状态，继续读取输入流。以读取标识符、关键字和数组为例的状态机如下(完整的状态机模型见第三部分)：

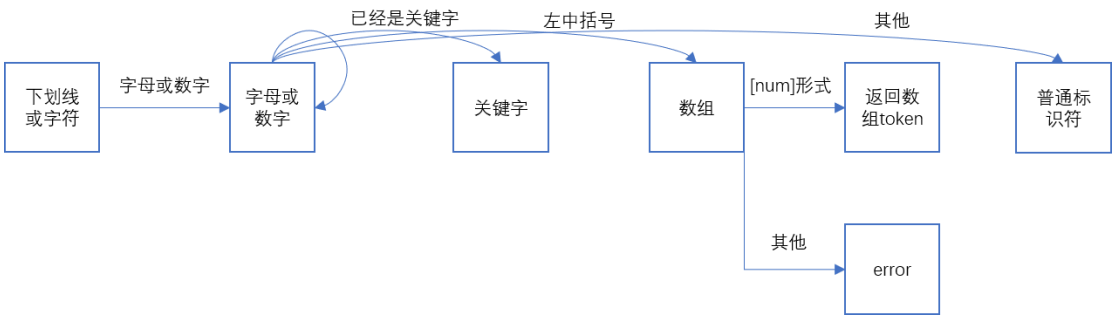


图2-1 有限状态机进行词法分析示例

语法分析

语法分析，就是要根据源程序内在的语法逻辑结构，生成抽象语法树以显示源程序的逻辑，并在语法错误时抛出异常。

对于抽象语法树的数据结构选取，考虑到生成语法树的美观性以及内在逻辑性，本程序选择了多叉树的孩子兄弟表示法(左孩子右兄弟)，基于链式结构存储多叉树。其结构如下图（以 if 语句块生成的语法树为例）：

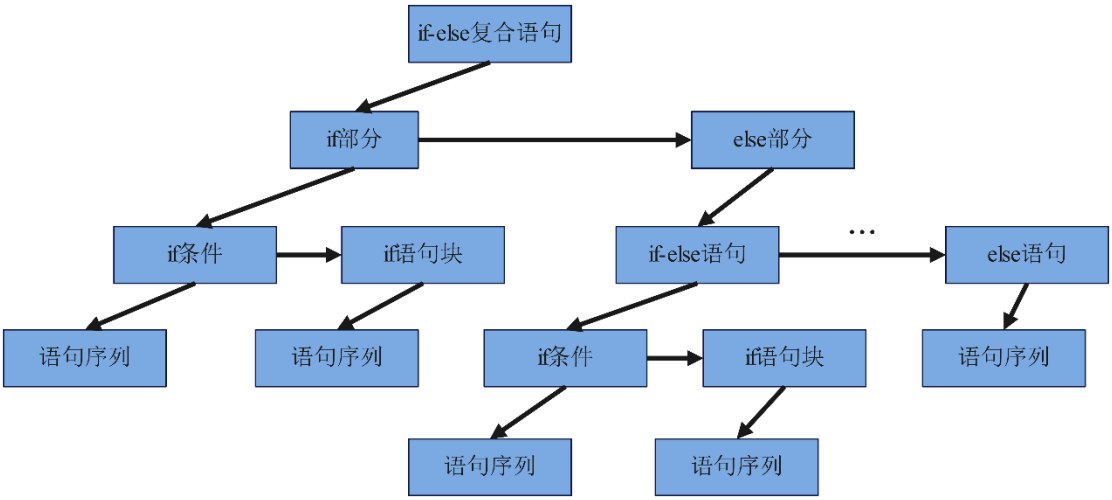


图 2-2 数据结构示意图

更详细的结构见第四部分。

源程序格式化

在前两个功能的基础上，程序的逻辑结构已经明确，可以通过先根遍历和块逻辑分析得到编排后的源程序。

整体系统架构

三大模块的相互关系见下图。更为详细的系统架构见系统详细设计部分。

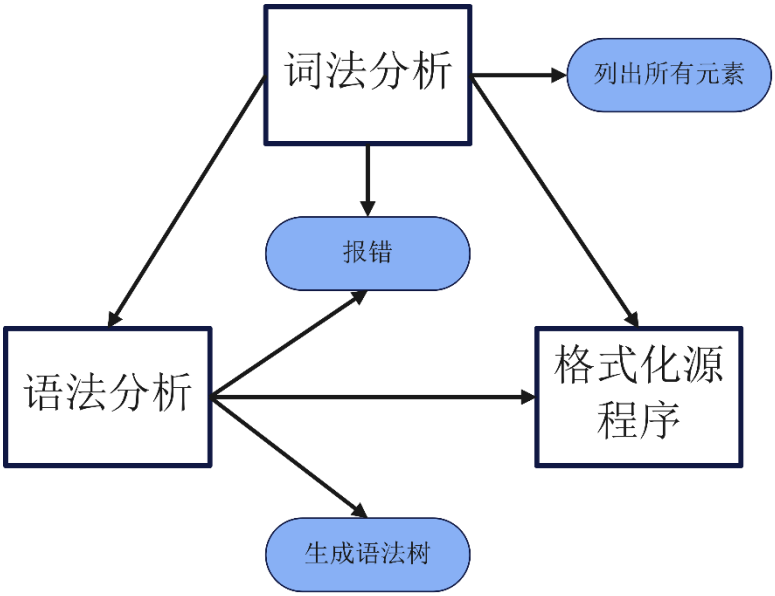


图 2-3 系统总体设计

3 系统详细设计

3.1 有关数据结构的定义

给定一段程序，在C语言的某个子集下，有标识符、常量、关键字(包括类型关键字)、括号、大括号、逗号、分号、赋值、运算、等与不等、注释、头文件、函数定义、函数声明、函数调用、数组、宏等元素。在词法分析中，要求按类别列出所有子集元素。各种元素的类别、含义与示例如图3-1。

元素类型	含义	对应宏标识	示例
ident	标识符	IDENT	a/b/token_text
type_const	各类型常量	TYPE_CONST	6/x45/045/3.6/"hello"
keyword	关键字	KEYWORD	int/while/if
type	类型(关键字)	INT/SHORT...	int/float/long
左右大括号	\	LB/RB	{}
左右括号	\	LP/RP	()
左右中括号	\	LM/RM	[]
分号	\	SEMI	;
逗号	\	COMMA	,
是否等于	\	EQ	=='
是否不等	\	NEQ	!=
赋值	\	ASSIGN	='
加减乘除与或	\	PLUS/MINUS…	='/-/*/&&/||
井号	\	POUND	#
大于小于号	\	MOREEQUAL…	>/</>=/<=
注释	\	ANNO	//' / /**/
头文件	\	INCLUDE	#include
宏	\	MACRO	#define
数组	\	ARRAY	a[10]
错误元素	\	ERROR_TOKEN	\

图3-1 元素类型及示例

为了逐个读取这些元素，设计全局变量token_text和word_type。前者用于存储元素内容，是string类型的变量，后者存储元素类型，是一个int型变量。Gettoken函数用于从输入流中读取每个元素，并返回word_type类型便于后续的语法分析，而token_text中则保存了对应的元素内容。

在语法分析中，先要确定语法分析树（AST）的数据结构。考虑到语法树要能够清晰地显示源程序的逻辑结构，因此每一种语句类型都要清楚地划分元素。这样，二叉树显然是不够的。考虑使用多叉树来描述逻辑结构。

多叉树的数据结构选用了孩子兄弟表示法(左孩子右兄弟)，其示意图如下：

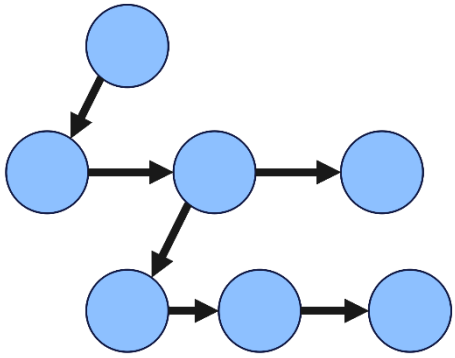


图3-2 多叉树示意图

存储多叉树通过一个结构体进行链式存储。结构体的定义如下图所示：

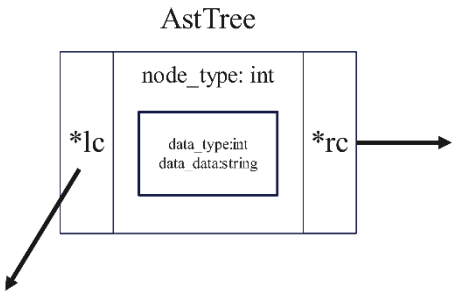


图3-3 语法树节点结构

其中，nodetype表示当前语句所属分支，lc和rc均为节点指针，data_type和data_data属于data结构体，包含语句的一些信息，用于后续输出多叉树和优先级比较。

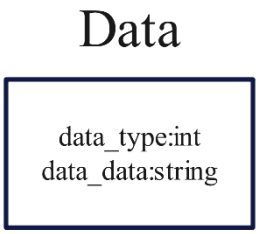


图3-4 data结构体

在参考实验指导书的前提下，详细的语法树设计在下面说明。由于空间限制，各个部分分开作图讲解。

首先设计的是外部定义序列，包括了外部变量定义、函数定义、函数声名和数组定义。下面的两张示意图展示的是外部定义序列的结构和外部变量定义的结构。

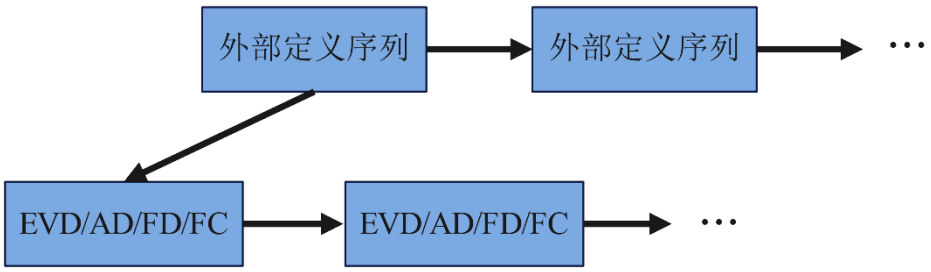


图3-5 外部定义序列结构

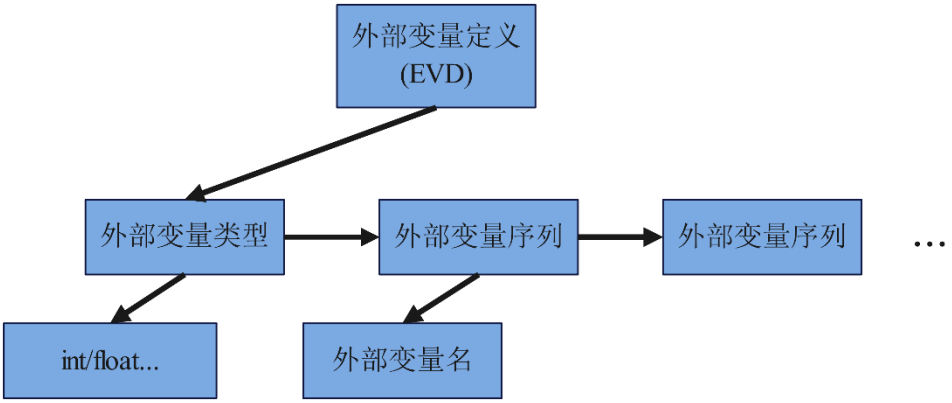


图 3-6 外部变量定义

语句序列的结构如下，每种类型的语句都是等价的，因此按兄弟节点链接，嵌套语句是孩子节点关系：

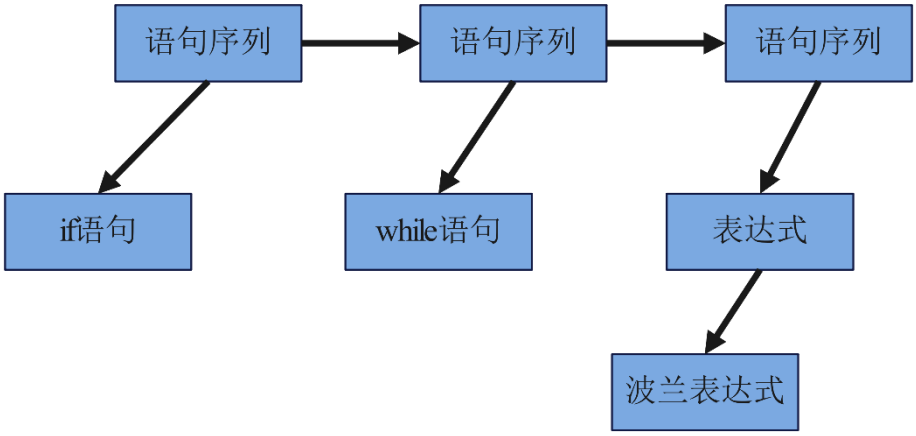


图 3-7 语句序列

对于 if、while 等语句块，这里不一一列举，以 if 语句块为例：

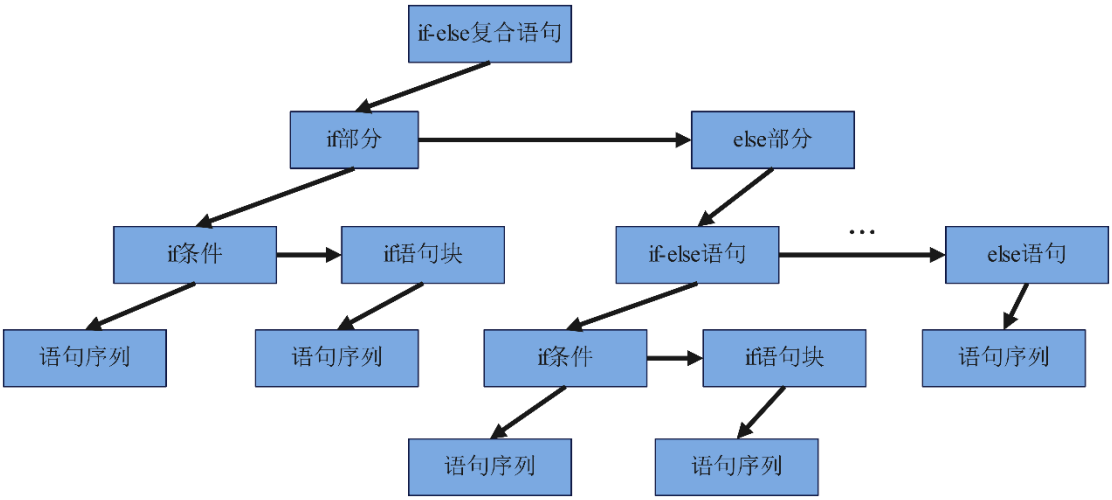


图 3-8 if 语句块

其中语句序列下挂的孩子节点为表达式节点，在这里省略。

下面的表展示的是完整的语法树节点类型：

节点类型	对应宏标识	节点类型	对应宏标识
外部定义序列	EXT_DEF_LIST	语句序列处理	STATE_LIST
外部变量定义	EXT_VAR_DEF	if复合语句	IF_STATE
外部变量类型	EXT_VAR_TYPE	if-else复合语句	IF_ELSE_STATE
外部变量	EXT_VAR	if语句块	IF_PART
外部变量序列	EXT_VAR_LIST	if条件	IF_COND
数组定义	ARRAY_DEF	else部分	ELSE_PART
数组类型	ARRAY_TYPE	while复合语句	WHILE_STATE
数组名	ARRAY_NAME	while条件部分	WHILE_COND
函数定义	FUNC_DEF	while语句块	WHILE_BODY
函数返回类型	FUNC_RETURN_TYPE	for复合语句	FOR_STATE
函数名	FUNC_NAME	for条件部分	FOR_COND
函数声明	FUNC_CLAIM	for条件1	FOR_COND_1
函数体	FUNC_BODY	for条件2	FOR_COND_2
函数形参列表	FUNC_PARA_LIST	for条件3	FOR_COND_3
函数形参定义	FUNC_PARA_DEF	for语句块	FOR_BODY
函数形参类型	FUNC_PARA_TYPE	return语句	RETURN_STATE
函数形参名	FUNC_PARA_NAME	break语句	BREAK_STATE
局部变量定义序列	LOCAL_VAR_DEF_LIST	continue语句	CONTINUE_STATE
局部变量定义	LOCAL_VAR_DEF	运算符	OPERATOR
局部变量类型	LOCAL_VAR_TYPE	操作数	OPERAND
局部变量名序列	LOCAL_VAR_NAME_LIST	表达式	EXPRESSION
局部变量名	LOCAL_VAR_NAME		

图 3-9 节点类型

3.2 主要算法设计

词法分析主要算法

此法分析部分主要用到的是 DNF 理论，列出完整的 DNF 状态转换图如图 3-9 所示。

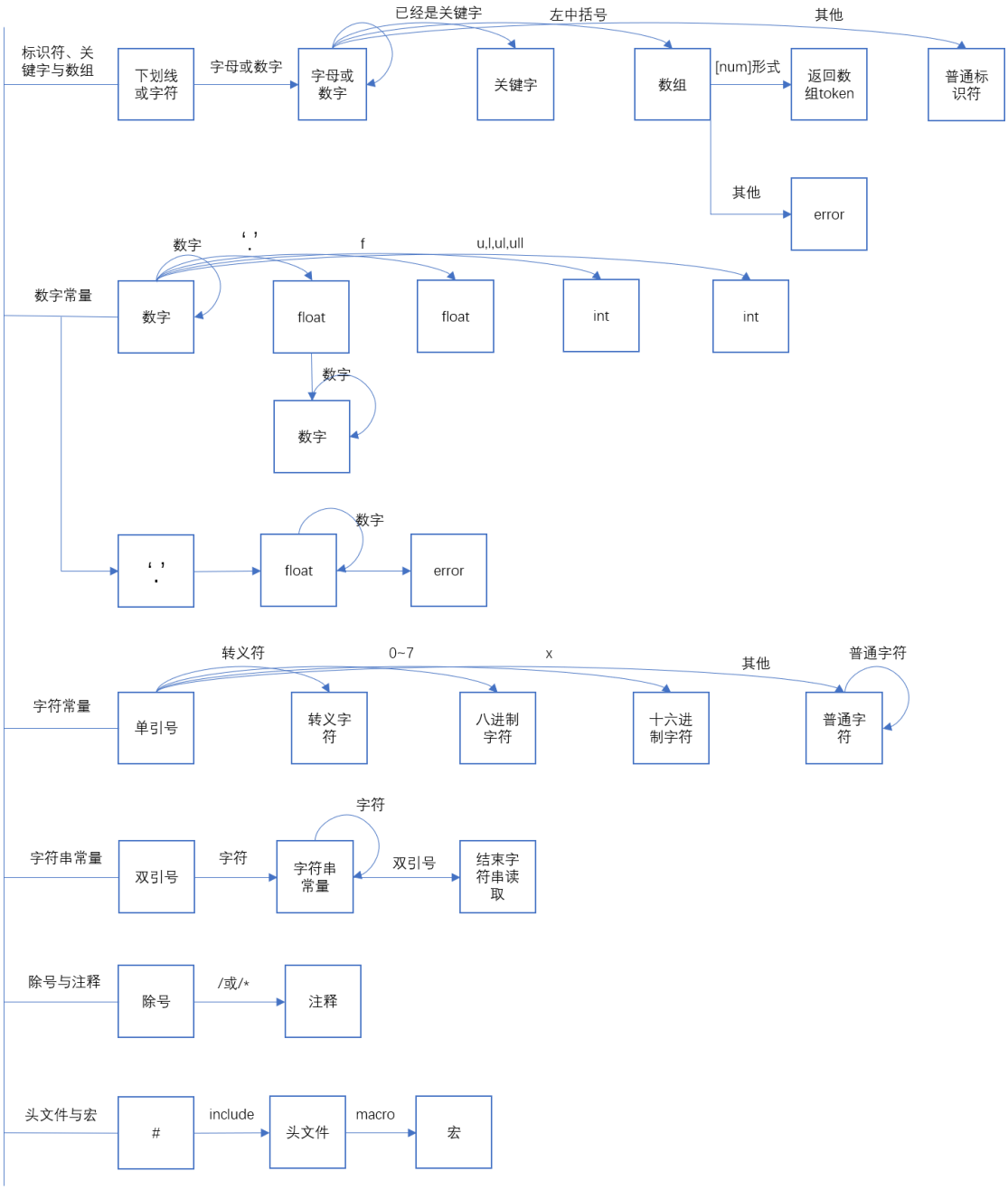


图 3-10 完整的 DNF 状态图

算法进行过程中，每次读取输入流中的一个字符，根据字符判定当前所处状

态。每当获取到一个 word_type 时，就根据对应的 type 输出元素类型，如果检测到错误就返回错误信息并结束程序。

语法分析主要算法

语法分析主要采用递归下降子程序法，每次读取一个元素就进行判断是哪个部分，逐步向细化的方向递归，在这个过程中逐步建立语法树。算法的伪代码如下：

算法 3 递归下降子程序法建立语法树

输入: None

输出: 树的根节点

```
1: cur_node ← getToken()
2: if notLeaf(curNode) then
3:     call   sub − function
4:     return   childTree
5: end ifreturn root
```

图 3-11 递归下降子程序法伪代码

程序整体架构

综合以上算法，给出程序整体架构与各个功能函数：

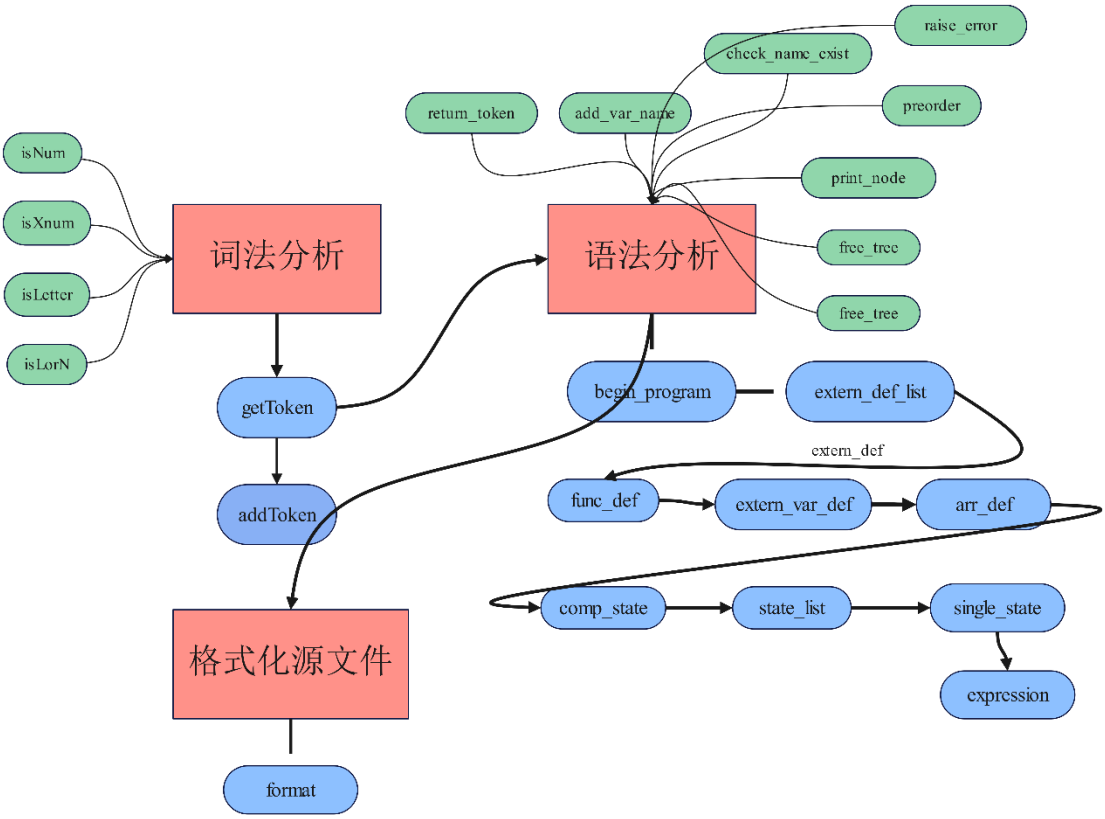

raise_error
check_name_exist
return_token
add_var_name
preorder
isNum
print_node
isXnum
词法分析
语法分析
free_tree
isLetter
free_tree
isLorN
getToken
begin_program
extern_def_list
extern_def
addToken
func_def
extern_var_def
arr_def
comp_state
state_list
single_state
格式化源文件
expression
format

图 3-9 程序整体架构

4 系统实现与测试

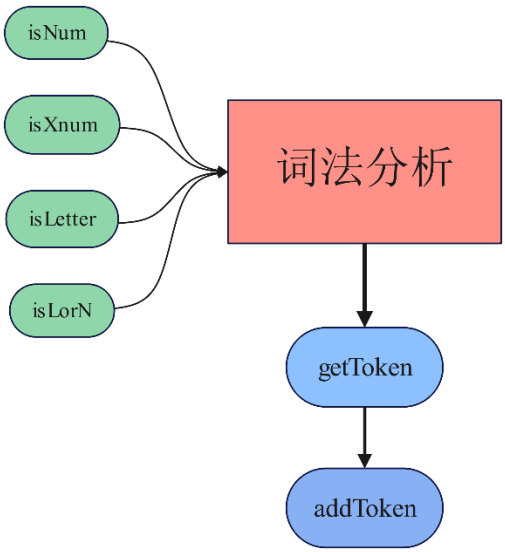
4.1 系统实现

运行环境

硬件环境：Windows/Linux 均可
软件环境：CLion 下 cmake 工程，运行时需用 cmake 构建编译运行

主要数据类型定义

词法分析主要使用函数 getToken(FILE* fp)，该函数从流中按照 DNF 理论读取一个最长可能元素，并返回该元素的类型。读取的元素将通过 addToken 函数存入 token_text 中。下图的蓝色区域就是主要函数，绿色区域是一些较为低级的辅助判断函数，这些具体的功能见附录中源程序。



语法分析部分采用了递归下降子程序发，所用到的函数较多，下面先列出主要函数：

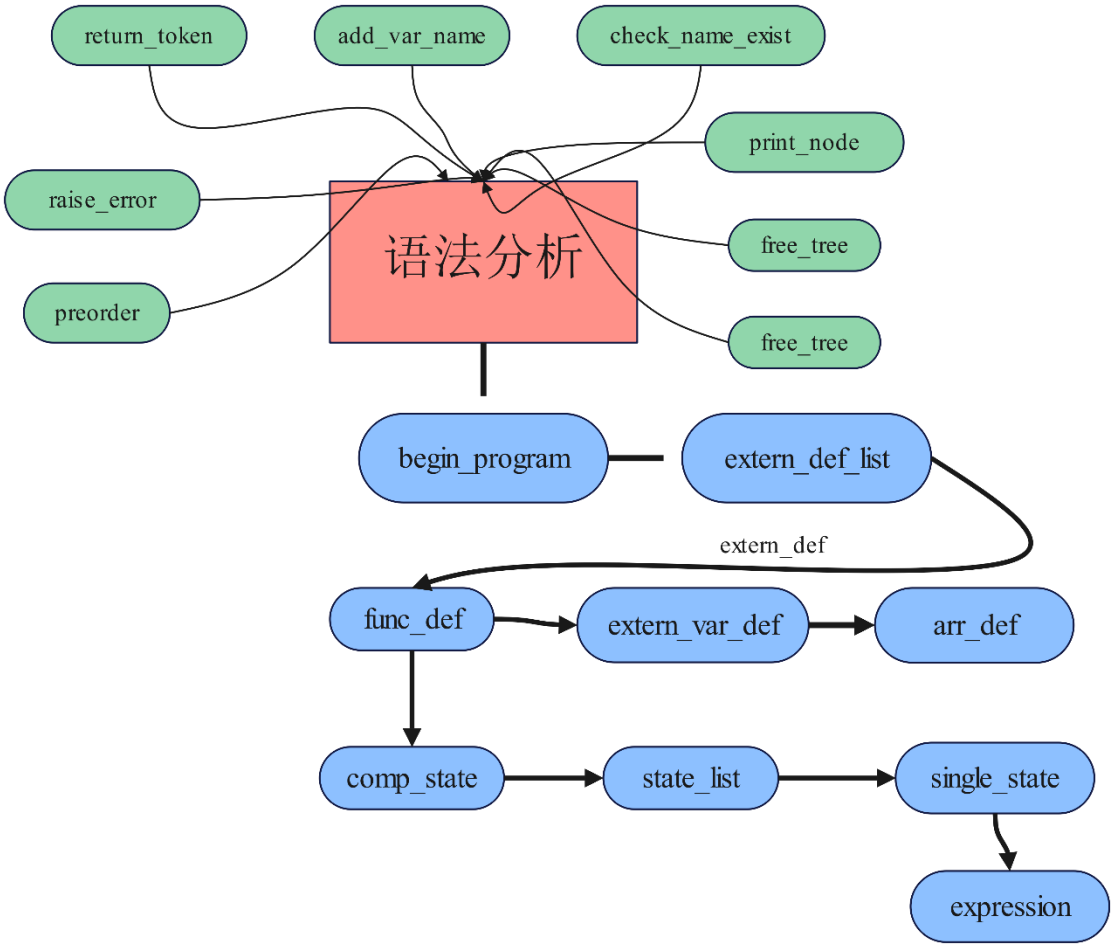
1. void syntax_analyse()：语法分析的主调用函数，通过这个函数开始递归下降。
2. Ast* begin_program()：语法分析的入口，进行一些预处理以及调用 extern_def_list
3. Ast* extern_def_list()：处理外部定义序列
4. Ast* extern_def()：辅助函数 extern_def 处理普通变量定义、数组定义还是函数定义

5. Ast* extern_var_def()：外部变量定义节点
6. Ast* extern_var_list()：extern_var_list 类型
7. Ast* arr_def()：数组定义节点
8. Ast* func_def()：函数定义节点
9. Ast* para_list()：函数形参列表
10. Ast* para_def()：函数形参节点
11. Ast* comp_state()：大括号扩着的复合语句
12. Ast* local_var_def_list()：局部变量定义序列节点
13. Ast* local_var_def()：局部变量定义节点，处理一行局部变量定义
14. Ast* state_list()：语句序列处理
15. Ast* single_state()：一条语句处理
16. void deal_with_if(Ast* cur_node)：处理 if 语句块
17. void deal_with_while(Ast* cur_node)：处理 while 语句块
18. void deal_with_for(Ast* cur_node)：处理 for 语句块
19. void deal_with_break(Ast* cur_node)：break 语句
20. void deal_with_continue(Ast* cur_node)：continue 语句
21. Ast* expression(int end_sym)：表达式处理
22. char precede(int c1,int c2)：优先级比较函数

还有一些较为底层的辅助判断函数：

1. void raise_error(int line_cnt, int error_type)：抛出错误
2. void preorder(Ast* cur_node, int depth)：先序遍历
3. void print_node(Ast* cur_node)：打印节点
4. void free_tree(Ast* root)：回收节点
5. void filter_anno_include()：获取第一个不是注释和头文件的 token
6. void return_token(FILE* fp)：将 token 返回到文件流
7. add_var_name(string token_text)：向某个域的定义添加变量
8. int check_name_exist(string token_text)：查看变量名是否被定义过

部分函数关系的示意图如下：



4. 2 系统测试

词法分析测试

目标：解析得到元素及类型列表

测试数据：test.txt(展示部分，完整测试内容见附录)：

```
#include <stdio.h>
#define TEST 1
int a;  // 外部变量定义
long b;
char _d, _e;
double c, h;
int num[10];
int i;
int func(int a, int b) {  // 函数定义
```

```
int test;
a = 1;
b = 1;
c = .14;
h = 3.13f;
_d = '\x22';
_e = '\t';
a = 1 + b;
b = 2 + b;
b = c;
```

测试结果（展示部分，完整输出见附录）：

```
类型       符号
头文件     #include <stdio.h>
宏         #define TEST 1
int      int
ident        a
分号       ;
注释       // 外部变量定义
long         long
ident        b
分号       ;
char         char
ident        _d
逗号       ,
ident        _e
分号       ;
double       double
ident        c
逗号       ,
ident        h
分号       ;
int      int
数组       num[10]
分号       ;
int      int
ident        i
分号       ;
int      int
ident        func
左括号     (
int      int
ident        a
逗号       ,
```

```
33. int     int
34. ident       b
35. 右括号      )
36. 左大括号        {
37. 注释       // 函数定义
38. int     int
```

经检验，该模块满足设计目标。

语法分析测试

目标：生成语法树
测试数据：测试数据：test.txt(展示部分，完整测试内容见附录)：

```
1.  #include <stdio.h>
2.  #define TEST 1
3.  int a;  // 外部变量定义
4.  long b;
5.  char _d, _e;
6.  double c, h;
7.  int num[10];
8.  int i;
9.  int func(int a, int b) {  // 函数定义
10. int test;
11. a = 1;
12. b = 1;
13. c = .14;
14. h = 3.13f;
15. _d = '\x22';
16. _e = '\t';
17. a = 1 + b;
18. b = 2 + b;
19. b = c;
```

测试结果（展示部分输出，完整输出见附录）：

```
1.  语句序列处理:
2.                  if-else 复合语句:
3.                      if 语句块:
4.                          if 条件:
5.                              表达式:
6.                                  运算符:==
```

```
                操作数:a
            操作数:b
        表达式:
            运算符:=
                操作数:a
            运算符:+
                操作数:a
            操作数:b
    else 部分:
        if-else 复合语句:
            if 语句块:
                if 条件:
                    表达式:
                        运算符:<
                            操作数:a
                        操作数:b
                    语句序列处理:
                        表达式:
                            运算符:=
                                操作数:a
                            运算符:-
                                操作数:a
                            操作数:b
                    语句序列处理:
                        表达式:
                            运算符:=
                                操作数:num[10]
                            操作数:"string"
                    语句序列处理:
                        if 复合语句:
                            if 语句块:
                                if 条件:
                                    表达式:
                                        运算符:>
                                            操作数:a
                                        操作数:b
                                    语句序列处理:
                                        表达式:
                                            运算符:=
                                                操作数:a
                                            运算符:+
                                                操作数:a
                                            操作数:1
            else 部分:
```

```
语句序列处理:
    表达式:
        运算符:=
            操作数:a
        运算符:/
            操作数:a
        操作数:b
语句序列处理:
    表达式:
        运算符:=
            操作数:a
        运算符:+
            操作数:a
        操作数:1
```

运行结果符合设计目标。

报错功能测试

目标：碰到简单错误时抛出错误
测试数据：在上面的 test.txt 上任意改动
测试结果（任选 4 种错误进行测试）：

1. 出现 void 类型变量

```
3    void a;  // 外部变量定义
4    long b;
5    char _d, _e;
```

```
在第3行出现错误
错误类型：变量类型错误
```

2. 变量命名不规范：出现美元符号

```
4    long $b;
5    //long b;
```

```
2
在第4行出现错误
错误类型：外部定义处错误
```

3. 变量重复定义

```
4    long b;
5    long b;
```

```
2
在第5行出现错误
错误类型：变量重复定义
```

4. Break 语句缺少分号

```
39    while (1) {  // 嵌套while
40    if (a > b) {
41    a = (a + 1) * 3;
42    b = b + c + 2;
43    break
```

```
2
在第44行出现错误
错误类型：break语句缺少分号
```

经检验，该功能完成设定目标

源程序格式处理测试

目标：格式化源程序

测试数据：test.txt

测试结果：

```
1.  #include <stdio.h>
2.  #define TEST 1
3.  int a ;
4.  long b ;
5.  char _d , _e ;
6.  double c , h ;
7.  int num[10] ;
8.  int i ;
9.  int func ( int a , int b ) {
10.     int test ;
11.     a = 1 ;
12.     b = 1 ;
13.     c = .14 ;
14.     h = 3.13f ;
15.     _d = '\x22' ;
16.     _e = '\t' ;
17.     a = 1 + b ;
18.     b = 2 + b ;
19.     b = c ;
```

```
    if ( a == b ) a = a + b ;
    else if ( a < b ) {
        a = a - b ;
        num[10] = "string" ;
        if ( a > b ) {
            a = a + 1 ;
        }
    }
    else {
        a = a / b ;
        a = a + 1 ;
    }
    if ( a > b ) a = a && b ;
    if ( a > b ) {
        a = ( a + 1 ) * 3 ;
        b = b + c + 2 ;
    }
    while ( 1 ) {
        if ( a > b ) {
            a = ( a + 1 ) * 3 ;
            b = b + c + 2 ;
            break ;
        }
        if ( a < b ) {
            continue ;
        }
        while ( a == 1 ) {
            a = a || b ;
        }
    }
    for ( i = 0 ; i < a ; i = i + 1 ) {
        a = a + 1 ;
    }
    for ( ; ; i = i - 1 ) {
        a = ( a + 1 ) * 2 ;
    }
    return a + b ;
}
void def ( ) ;
```

经检验，运行结果符合设计目标。

5 总结与展望

5.1 全文总结

已经完成的主要工作如下：

（1）实现基本的词法分析功能

（2）实现基本的语法分析功能，生成了语法分析树

（3）实现报错功能，可以对基本错误进行报错

（4）实现了格式化源程序的功能

（5）应用 cmake 实现了基本的跨平台功能

5.1 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）增加包含的元素种类，在原有的 C 语言子集上进行扩展

（2）增加平台可视化的功能

（3）考虑到有限状态机具有一定的局限性，可以考虑用正则表达式重写词法分析功能。

6 体会

1. 开始项目前先把基本的系统架构做好，有更好的想法可以后面加。
2. 每写一部分就要进行测试，否则后面代码量过大难以调试。
3. 写 C/C++尤其要注意内存管理

参考文献

将报告中引用的参考文献放在此处。

[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] 形式语言与自动机. CSDN.沉香屑.2017-07-11

[3] 严蔚敏等.数据结构(C 语言版).北京：清华大学出版社

[4]基本的 C 语言编译器的实现.CSDN.dongchangzhang.2017-05-24

附录一：test.txt 测试文件

```
#include <stdio.h>
#define TEST 1
int a;  // 外部变量定义
long b;
char _d, _e;
double c, h;
int num[10];
int i;
int func(int a, int b) {  // 函数定义
int test;
a = 1;
b = 1;
c = .14;
h = 3.13f;
_d = '\x22';
_e = '\t';
a = 1 + b;
b = 2 + b;
b = c;
if (a == b)  // if
a = a + b;
else if (a < b) {  // 块 if、嵌套 if
a = a - b;
num[10] = "string";
if (a > b) {
a = a + 1;
}
}
else {
a = a / b;
a = a + 1;
}
if (a > b)
a = a && b;
if (a > b) {
a = (a + 1) * 3;
b = b + c + 2;
}
while (1) {  // 嵌套 while
if (a > b) {
a = (a + 1) * 3;
b = b + c + 2;
```

```
break;
}
if (a < b) {
continue;
}
while (a == 1) {
a = a || b;
}
}
for (i = 0; i < a; i=i+1) {  // for 循环
a = a + 1;
}
for (;; i=i-1) {  // 空条件 for
a = (a + 1) * 2;
}
return a + b;
}
/*函数声明*/
void def();
~
//已经实现的报错：变量为 void 类型、变量为未知类型、变量命名不规范、函数没有返回值、函数形参错误、
//出现未知符号、变量重复定义、void 函数有返回值、break 语句缺少分号、continue 缺少分号、非循环
//出现 break/continue、使用未定义的变量
```

附录二：词法分析完整输出

```
1.  类型       符号
2.  头文件     #include <stdio.h>
3.  宏        #define TEST 1
4.  int      int
5.  ident        a
6.  分号       ;
7.  注释       // 外部变量定义
8.  long         long
9.  ident        b
10. 分号       ;
11. char         char
12. ident        _d
13. 逗号       ,
14. ident        _e
15. 分号       ;
16. double       double
17. ident        c
18. 逗号       ,
19. ident        h
20. 分号       ;
21. int      int
22. 数组       num[10]
23. 分号       ;
24. int      int
25. ident        i
26. 分号       ;
27. int      int
28. ident        func
29. 左括号     (
30. int      int
31. ident        a
32. 逗号       ,
33. int      int
34. ident        b
35. 右括号     )
36. 左大括号         {
37. 注释       // 函数定义
38. int      int
39. ident        test
40. 分号       ;
41. ident        a
42. 赋值       =
```

```
int_const       1
分号      ;
ident       b
赋值      =
int_const       1
分号      ;
ident       c
赋值      =
float_const     .14
分号      ;
ident       h
赋值      =
float_const     3.13f
分号      ;
ident       _d
赋值      =
char_const      '\x22'
分号      ;
ident       _e
赋值      =
char_const      '\t'
分号      ;
ident       a
赋值      =
int_const       1
加法      +
ident       b
分号      ;
ident       b
赋值      =
int_const       2
加法      +
ident       b
分号      ;
ident       b
赋值      =
ident       c
分号      ;
if      if
左括号      (
ident       a
是否等于        ==
ident       b
右括号      )
```

```
注释      // if
ident       a
赋值      =
ident       a
加法      +
ident       b
分号      ;
else        else
if      if
左括号     (
ident       a
小于号     <
ident       b
    右括号     )
    左大括号        {
    注释      // 块 if、嵌套 if
    ident       a
    赋值      =
    ident       a
    减法      -
    ident       b
    分号      ;
    数组      num[10]
    赋值      =
    string_const        "string"
    分号      ;
    if      if
    左括号     (
    ident       a
    大于号     >
    ident       b
    右括号     )
    左大括号        {
    ident       a
    赋值      =
    ident       a
    加法      +
    int_const       1
    分号      ;
    右大括号        }
    右大括号        }
    else        else
    左大括号        {
    ident       a
```

```
131.  赋值      =
132.  ident       a
133.  除法      /
134.  ident       b
135.  分号      ;
136.  ident       a
137.  赋值      =
138.  ident       a
139.  加法      +
140.  int_const       1
141.  分号      ;
142.  右大括号        }
143.  if      if
144.  左括号     (
145.  ident       a
146.  大于号     >
147.  ident       b
148.  右括号     )
149.  ident       a
150.  赋值      =
151.  ident       a
152.  与运算     &&
153.  ident       b
154.  分号      ;
155.  if      if
156.  左括号     (
157.  ident       a
158.  大于号     >
159.  ident       b
160.  右括号     )
161.  左大括号        {
162.  ident       a
163.  赋值      =
164.  左括号     (
165.  ident       a
166.  加法      +
167.  int_const       1
168.  右括号     )
169.  乘法      *
170.  int_const       3
171.  分号      ;
172.  ident       b
173.  赋值      =
174.  ident       b
```

```
175.  加法      +
176.  ident       c
177.  加法      +
178.  int_const       2
179.  分号      ;
180.  右大括号        }
181.  while       while
182.  左括号     (
183.  int_const       1
184.  右括号     )
185.  左大括号        {
186.  注释      // 嵌套 while
187.  if      if
188.  左括号     (
189.  ident       a
190.  大于号     >
191.  ident       b
192.  右括号     )
193.  左大括号        {
194.  ident       a
195.  赋值      =
196.  左括号     (
197.  ident       a
198.  加法      +
199.  int_const       1
200.  右括号     )
201.  乘法      *
202.  int_const       3
203.  分号      ;
204.  ident       b
205.  赋值      =
206.  ident       b
207.  加法      +
208.  ident       c
209.  加法      +
210.  int_const       2
211.  分号      ;
212.  break       break
213.  分号      ;
214.  右大括号        }
215.  if      if
216.  左括号     (
217.  ident       a
218.  小于号     <
```

```
219.  ident       b
220.  右括号     )
221.  左大括号        {
222.  continue        continue
223.  分号      ;
224.  右大括号        }
225.  while       while
226.  左括号     (
227.  ident       a
228.  是否等于        ==
229.  int_const       1
230.  右括号     )
231.  左大括号        {
232.  ident       a
233.  赋值      =
234.  ident       a
235.  或运算     ||
236.  ident       b
237.  分号      ;
238.  右大括号        }
239.  右大括号        }
240.  for     for
241.  左括号     (
242.  ident       i
243.  赋值      =
244.  int_const       0
245.  分号      ;
246.  ident       i
247.  小于号     <
248.  ident       a
249.  分号      ;
250.  ident       i
251.  赋值      =
252.  ident       i
253.  加法      +
254.  int_const       1
255.  右括号     )
256.  左大括号        {
257.  注释      // for 循环
258.  ident       a
259.  赋值      =
260.  ident       a
261.  加法      +
262.  int_const       1
```

```
分号      ;
右大括号        }
for     for
左括号     (
分号      ;
分号      ;
ident       i
赋值      =
ident       i
减法      -
int_const       1
右括号     )
左大括号        {
注释      // 空条件 for
ident       a
赋值      =
左括号     (
ident       a
加法      +
int_const       1
右括号     )
乘法      *
int_const       2
分号      ;
右大括号        }
return      return
ident       a
加法      +
ident       b
分号      ;
右大括号        }
注释      /*函数声明*/
void        void
ident       def
左括号     (
右括号     )
分号      ;
词法分析结束，按任意键继续
```

附录三：语法分析完整输出

```
1.  语法正确，语法树先序遍历如下：
2.  外部定义序列:
3.      外部变量定义:
4.          外部变量类型:int
5.          外部变量序列:
6.              外部变量:a
7.  外部定义序列:
8.      外部变量定义:
9.          外部变量类型:long
10.         外部变量序列:
11.             外部变量:b
12. 外部定义序列:
13.     外部变量定义:
14.         外部变量类型:char
15.         外部变量序列:
16.             外部变量:_d
17.         外部变量序列:
18.             外部变量:_e
19. 外部定义序列:
20.     外部变量定义:
21.         外部变量类型:double
22.         外部变量序列:
23.             外部变量:c
24.         外部变量序列:
25.             外部变量:h
26. 外部定义序列:
27.     数组定义:
28.         数组类型:int
29.         数组名:num[10]
30. 外部定义序列:
31.     外部变量定义:
32.         外部变量类型:int
33.         外部变量序列:
34.             外部变量:i
35. 外部定义序列:
36.     函数定义:
37.         函数返回类型:int
38.         函数名:func
39.         函数形参列表:
40.             函数形参定义:
41.                 函数形参类型:int
42.                 函数形参名:a
```

```
            函数形参定义:
                函数形参类型:int
                函数形参名:b
        函数体:
            局部变量定义序列:
                局部变量定义:
                    局部变量类型:int
                    局部变量名序列:
                        局部变量名:test
            语句序列处理:
                表达式:
                    运算符:=
                        操作数:a
                    操作数:1
            语句序列处理:
                表达式:
                    运算符:=
                        操作数:b
                    操作数:1
            语句序列处理:
                表达式:
                    运算符:=
                        操作数:c
                    操作数:.14
            语句序列处理:
                表达式:
                    运算符:=
                        操作数:h
                    操作数:3.13f
            语句序列处理:
                表达式:
                    运算符:=
                        操作数:_d
                    操作数:'\x22'
            语句序列处理:
                表达式:
                    运算符:=
                        操作数:_e
                    操作数:'\t'
            语句序列处理:
                表达式:
                    运算符:=
                        操作数:a
                    运算符:+
```

```
                操作数:1
            操作数:b
        语句序列处理:
            表达式:
                运算符:=
                    操作数:b
                运算符:+
                    操作数:2
                操作数:b
        语句序列处理:
            表达式:
                运算符:=
                    操作数:b
                操作数:c
            语句序列处理:
                if-else 复合语句:
                    if 语句块:
                        if 条件:
                            表达式:
                                运算符:==
                                    操作数:a
                                操作数:b
                        表达式:
                            运算符:=
                                操作数:a
                            运算符:+
                                操作数:a
                            操作数:b
                    else 部分:
                        if-else 复合语句:
                            if 语句块:
                                if 条件:
                                    表达式:
                                        运算符:<
                                            操作数:a
                                        操作数:b
                                语句序列处理:
                                    表达式:
                                        运算符:=
                                            操作数:a
                                        运算符:-
                                            操作数:a
                                        操作数:b
                                语句序列处理:
```

```
                                表达式:
                                    运算符:=
                                        操作数:num[10]
                                    操作数:"string"
                            语句序列处理:
                                if 复合语句:
                                    if 语句块:
                                        if 条件:
                                            表达式:
                                                运算符:>
                                                    操作数:a
                                                操作数:b
                                            语句序列处理:
                                                表达式:
                                                    运算符:=
                                                        操作数:a
                                                    运算符:+
                                                        操作数:a
                                                    操作数:1
                        else 部分:
                            语句序列处理:
                                表达式:
                                    运算符:=
                                        操作数:a
                                    运算符:/
                                        操作数:a
                                    操作数:b
                            语句序列处理:
                                表达式:
                                    运算符:=
                                        操作数:a
                                    运算符:+
                                        操作数:a
                                    操作数:1
            语句序列处理:
                if 复合语句:
                    if 语句块:
                        if 条件:
                            表达式:
                                运算符:>
                                    操作数:a
                                操作数:b
                            表达式:
                                运算符:=
```

```
                                操作数:a
                            运算符:&&
                                操作数:a
                            操作数:b
            语句序列处理:
                if 复合语句:
                    if 语句块:
                        if 条件:
                            表达式:
                                运算符:>
                                    操作数:a
                                操作数:b
                        语句序列处理:
                            表达式:
                                运算符:=
                                    操作数:a
                                运算符:*
                                    运算符:+
                                        操作数:a
                                    操作数:1
                                操作数:3
                        语句序列处理:
                            表达式:
                                运算符:=
                                    操作数:b
                                运算符:+
                                    运算符:+
                                        操作数:b
                                    操作数:c
                                操作数:2
            语句序列处理:
                while 复合语句:
                    while 条件部分:
                        表达式:
                            操作数:1
                    while 语句块:
                        语句序列处理:
                            if 复合语句:
                                if 语句块:
                                    if 条件:
                                        表达式:
                                            运算符:>
                                                操作数:a
                                            操作数:b
```

```
                                语句序列处理:
                                    表达式:
                                        运算符:=
                                            操作数:a
                                        运算符:*
                                            运算符:+
                                                操作数:a
                                            操作数:1
                                        操作数:3
                                语句序列处理:
                                    表达式:
                                        运算符:=
                                            操作数:b
                                        运算符:+
                                            运算符:+
                                                操作数:b
                                            操作数:c
                                        操作数:2
                                语句序列处理:
                                    break 语句:
                        语句序列处理:
                            if 复合语句:
                                if 语句块:
                                    if 条件:
                                        表达式:
                                            运算符:<
                                                操作数:a
                                            操作数:b
                                    语句序列处理:
                                        continue 语句:
                        语句序列处理:
                            while 复合语句:
                                while 条件部分:
                                    表达式:
                                        运算符:==
                                            操作数:a
                                        操作数:1
                                while 语句块:
                                    语句序列处理:
                                        表达式:
                                            运算符:=
                                                操作数:a
                                            运算符:||
                                                操作数:a
```

```
                                                    操作数:b
                语句序列处理:
                    for 复合语句:
                        for 条件部分:
                            for 条件 1:
                                表达式:
                                    运算符:=
                                        操作数:i
                                    操作数:0
                            for 条件 2:
                                表达式:
                                    运算符:<
                                        操作数:i
                                    操作数:a
                            for 条件 3:
                                表达式:
                                    运算符:=
                                        操作数:i
                                    运算符:+
                                        操作数:i
                                    操作数:1
                语句序列处理:
                    for 复合语句:
                        for 条件部分:
                            for 条件 1:无
                            for 条件 2:无
                            for 条件 3:
                                表达式:
                                    运算符:=
                                        操作数:i
                                    运算符:-
                                        操作数:i
                                    操作数:1
                语句序列处理:
                    return 语句:
                        表达式:
                            运算符:+
                                操作数:a
                            操作数:b
外部定义序列:
    函数声明:
        函数返回类型:void
        函数名:def
        函数形参列表:
```

```
307.    语法分析结束，按任意键继续
```

附录四：项目源码

Main.cpp

```
#include <iostream>
#include <string>
#include "syntax_analyse.h"
#include "get_token.h"
#include "format_operation.h"
using namespace std;

extern int line_cnt;
extern string token_text;
FILE *fp;

int main() {
    system("chcp 65001");
    char filename[20];
    cout << "enter the filename:" << endl;
    cin >> filename;
    int op;
    while (1){
        fp = fopen(filename,"r");
        if (fp == nullptr){
            cout << "文件打开错误" << endl;
            return 0;
        }
        cout << "--------------------" << endl;
        cout << "  输入序号选择功能" << endl;
        cout << "    1. 词法分析" << endl;
        cout << "    2. 语法分析" << endl;
        cout << "    3. 格式化" << endl;
        cout << "    0. 退出" << endl;
        cout << "--------------------" << endl;
        cin >> op;
        switch (op) {
            case 1:
                int token;
                cout << "类型\t\t" << "符号" << endl;
                while ((token = getToken(fp)) != -1){
                    if (token == ERROR_TOKEN){
                        cout << "在第" << line_cnt << "行出现错误!
" << endl;
```

```
                        cout << "错误信息为:
" << token_text << "over!" << endl;
                        break;
                    }
                    else{
                        cout << token_kind_arr[token] << "\t\t" << to
ken_text << endl;
                    }
                }
                cout << "词法分析结束，按任意键继续" << endl;
                break;
            case 2:
                line_cnt = 1;
                syntax_analyse();
                cout << "语法分析结束，按任意键继续" << endl;
                break;
            case 3:
                //fp = fopen("../test.txt","r");
                format();
                cout << "格式化结束，请按任意键继续" << endl;
                //fclose(fp);
                break;
            case 0:
                //system("clear");
                cout << "感谢使用" << endl;
                return 0;
            default:
                cout << "输入有误，按任意键重新输入" << endl;
        }
        fclose(fp);
    }
    return 0;
}
```

get_token.h

```
//
// Created by Mr.K on 2022-08-02.
//

#ifndef GET_TOKEN_H
#define GET_TOKEN_H

```

```
#include <string>

using namespace std;

/*词法分析编码列表*/
typedef enum token_kind{
    ERROR_TOKEN = 1,
    IDENT,  //标识符
    INT_CONST,  //整形常量
    FLOAT_CONST,
    CHAR_CONST,
    STRING_CONST,
    KEYWORD,
    INT,  // 8
    FLOAT,
    CHAR,
    LONG,
    SHORT,
    DOUBLE,
    VOID,  // 14
    ELSE,
    DO,
    WHILE,
    FOR,
    IF,
    BREAK,
    SWITCH,
    CASE,
    TYPEDEF,
    RETURN,
    CONTINUE,
    STRUCT,  //26
    LB,  //左大括号
    RB,  //右大括号
    LM,  //左中括号
    RM,  //右中括号
    SEMI,  //分号 31
    COMMA,  //逗号
    /*EQ 到 MINUSMINUS 为运算符，必须连在一起*/
    EQ,  //'=='
    NEQ,  //‘!=’
    ASSIGN,  //'='35
    LP,  //左括号
    RP,  //右括号
```

```
52.     TIMES,  //乘法
53.     DIVIDE,  //除法
54.     PLUS,  //加法 40
55.     OR, //或运算
56.     POUND,  //井号 42
57.     MORE,  //大于号
58.     LESS,  //小于号
59.     MOREEQUAL,  //大于等于
60.     LESSEQUAL,  //小于等于
61.     MINUS,  //减法
62.     AND,  //与运算
63.     ANNO,  //注释
64.     INCLUDE,  //头文件引用
65.     MACRO,  //宏定义
66.     ARRAY,  //数组
67. }token_kind;
68.
69. static string token_kind_arr[53] = {"a", "b", "ident", "int_const", "
    float_const", "char_const", \
70.     "string_const", "keyword", "int", "float", "char", "long", "short
    ", \
71.     "double", "void", "else", "do", "while", "for", "if", "break",\
72.     "switch", "case", "typedef", "return", "continue", "struct", "左大
    括号",\
73.     "右大括号", "左中括号", "右中括号", "分号", "逗号", "是否等于", "不等
    于",\
74.     "赋值", "左括号","右括号", "乘法", "除法", "加法", "或运算",\
75.     "井号", "大于号", "小于号", "大于等于", "小于等于", "减法", "与运算
    ", \
76.     "注释","头文件", "宏", "数组"};
77.
78.
79. // 判断是否是数字
80. int isNum(char c);
81.
82. // 判断是否是十六进制数
83. int isXnum(char c);
84.
85. // 判断是否是字母
86. int isLetter(char c);
87.
88. // 判断是否是字母或数字
89. int isLorN(char c);
90.
```

```
// 向字符串中添加字符
int add2token(string s, char c);

// 读取输入并返回当前读取 token 的类型
int getToken(FILE* fp);

#endif
```

get_token.cpp

```
//
// Created by Mr.K on 2022-08-02.
//

#include "get_token.h"

#include <cstdio>


string token_text = "";  // 存放读取的一个词
char c;  // 存放从文件流中读取的字符
int line_cnt = 1;  // 读取的代码行数



int isNum(char c)
{
    if (c >= 48 && c <= 57) {
        return 1;
    }
    return 0;
}

int isXnum(char c)
{
    if (isNum(c) || c >= 'a' && c <= 'f' || c >= 'A' && c <= 'F') {
        return 1;
    }
    return 0;
}

int isLetter(char c)
```

```
{
    if (c >= 'a' && c <= 'z' || c >= 'A' && c <= 'Z') {
        return 1;
    }
    return 0;
}

int isLorN(char c)
{
    if (isNum(c) || isLetter(c)) {
        return 1;
    }
    return 0;
}

int add2token(string s, char c){
    token_text += c;
    return 1;
}

int getToken(FILE* fp){
    token_text = "";
    do{  // 过滤缩进和空行
        c = fgetc(fp);
        if (c == '\n') line_cnt++;
    } while (c == ' ' || c == '\n');
    /*标识符、关键字、数组的识别*/
    if (c == '_' || isLetter(c)){
        add2token(token_text, c);
        while (isLorN((c = getc(fp)))){
            token_text += c;
        }
        /*判断是否是关键字*/
        if (token_text == "int") {ungetc(c, fp); return INT;}
        if (token_text == "double") {ungetc(c, fp);return DOUBLE;}
        if (token_text == "char") {ungetc(c, fp);return CHAR;}
        if (token_text == "short") {ungetc(c, fp);return SHORT;}
        if (token_text == "long") {ungetc(c, fp);return LONG;}
        if (token_text == "float") {ungetc(c, fp);return FLOAT;}
        if (token_text == "if") {ungetc(c, fp);return IF;}
        if (token_text == "else") {ungetc(c, fp);return ELSE;}
        if (token_text == "do") {ungetc(c, fp);return DO;}
        if (token_text == "while") {ungetc(c, fp);return WHILE;}
        if (token_text == "for") {ungetc(c, fp);return FOR;}
```

```
78.         if (token_text == "struct") {ungetc(c, fp);return STRUCT;}
79.         if (token_text == "break") {ungetc(c, fp);return BREAK;}
80.         if (token_text == "switch") {ungetc(c, fp);return SWITCH;}
81.         if (token_text == "case") {ungetc(c, fp);return CASE;}
82.         if (token_text == "typedef") {ungetc(c, fp);return TYPEDEF;}
83.         if (token_text == "return") {ungetc(c, fp);return RETURN;}
84.         if (token_text == "continue") {ungetc(c, fp);return CONTINUE;
    }
85.         if (token_text == "void") {ungetc(c, fp);return VOID;}
86.         /*判断是否是数组*/
87.         if (c == '['){
88.             do{
89.                 add2token(token_text, c);
90.                 c = fgetc(fp);
91.             } while (isNum(c));
92.             if (c != ']') return ERROR_TOKEN;
93.             else{
94.                 add2token(token_text, c);
95.                 return ARRAY;
96.             }
97.         }
98.         else{  // 普通的标识符
99.             ungetc(c, fp);
100.                return IDENT;
101.            }
102.        }
103.        /*数字的识别*/
104.        if (isNum(c)){
105.            do{
106.                add2token(token_text, c);
107.            } while (isNum(c = getc(fp)));
108.            if (c != '.' && c != 'u' && c != 'l') {  // int
109.                if(c != ' ' && c != ';' && c!=')' && c!='+' && c!='-'
    && c!='*' && c!='/'){
110.                    return ERROR_TOKEN;
111.                }
112.                ungetc(c,fp);
113.                return INT_CONST;
114.            }
115.            else if (c == '.'){  // float
116.                c = getc(fp);
117.                if (!isNum(c)) return ERROR_TOKEN;
118.                else{
```

```
119.                    ungetc(c,fp);
120.                    c='.';
121.                    add2token(token_text,c);
122.                    c=fgetc(fp);
123.                    do{
124.                        add2token(token_text, c);
125.                    } while (isNum(c=fgetc(fp)));
126.                    if(c == 'f'){
127.                        add2token(token_text, c);
128.                        return FLOAT_CONST;
129.                    }
130.                    else{
131.                        ungetc(c,fp);
132.                    }
133.                    return FLOAT_CONST;
134.                }
135.            }
136.            else if(c == 'u'){
137.                add2token(token_text, c);
138.                c=fgetc(fp);
139.                if(c == 'l'){
140.                    add2token(token_text, c);
141.                    c = fgetc(fp);
142.                    if(c == 'l'){
143.                        add2token(token_text, c);
144.                        return INT_CONST;
145.                    }
146.                    else{
147.                        ungetc(c,fp);
148.                        return INT_CONST;
149.                    }
150.                }
151.                else{
152.                    ungetc(c,fp);
153.                    return INT_CONST;
154.                }
155.            }
156.            else if(c=='l'){
157.                add2token(token_text, c);
158.                return INT_CONST;
159.            }
160.            else{
161.                return ERROR_TOKEN;
162.            }
```

```
163.        }
164.        /*数字的识别：小数点开头的浮点型*/
165.        if(c == '.'){
166.            do{
167.                add2token(token_text, c);
168.            } while (isNum(c = fgetc(fp)));
169.            ungetc(c,fp);
170.            return FLOAT_CONST;
171.        }
172.        /*字符型常量*/
173.        if (c == '\''){
174.            add2token(token_text, c);
175.            if ((c = getc(fp)) != '\\'){  // 普通字符
176.                add2token(token_text, c);
177.                if ((c = fgetc(fp)) != '\'') return ERROR_TOKEN;
178.                else{
179.                    add2token(token_text, c);
180.                    return CHAR_CONST;
181.                }
182.            }
183.            else{  // 转义字符
184.                add2token(token_text, c);
185.                c = fgetc(fp);
186.                if (c=='n' || c=='t' || c=='\\' || c=='\'' || c=='\"')
    {  // 普通转义字符
187.                    add2token(token_text, c);
188.                    if ((c = fgetc(fp)) != '\'') return ERROR_TOKEN;
189.                    else{
190.                        add2token(token_text, c);
191.                        return CHAR_CONST;
192.                    }
193.                }
194.                else if (c == 'x'){  // 十六进制字符
195.                    add2token(token_text, c);
196.                    c = fgetc(fp);
197.                    if (isXnum(c)){
198.                        add2token(token_text, c);
199.                        c = getc(fp);
200.                        if (isXnum(c)){
201.                            add2token(token_text, c);
202.                            if ((c = fgetc(fp)) != '\'') return ERROR_
    TOKEN;
203.                            else{
204.                                add2token(token_text, c);
```

```
                                return CHAR_CONST;
                            }
                        }
                        else{
                            if (c != '\'') return ERROR_TOKEN;
                            else{
                                add2token(token_text, c);
                                return CHAR_CONST;
                            }
                        }
                    }
                    else{
                        return ERROR_TOKEN;
                    }
                }
                else if (c >= '0' && c <= '7'){
                    add2token(token_text, c);
                    c = fgetc(fp);
                    if (c >= '0' && c <= '7'){
                        add2token(token_text, c);
                        c = fgetc(fp);
                        if (c >= '0' && c <= '7'){
                            add2token(token_text, c);
                            if ((c = fgetc(fp)) != '\'') return ERROR_
    TOKEN;
                            else{
                                add2token(token_text, c);
                                return CHAR_CONST;
                            }
                        }
                        else{
                            if (c != '\'') return ERROR_TOKEN;
                            else{
                                add2token(token_text, c);
                                return CHAR_CONST;
                            }
                        }
                    }
                    else{
                        if (c != '\'') return ERROR_TOKEN;
                        else{
                            add2token(token_text, c);
                            return CHAR_CONST;
                        }
                    }
```

```
248.                     }
249.                 }
250.                 else return ERROR_TOKEN;
251.             }
252.         }
253.         /*字符串常量*/
254.         if (c == '"'){
255.             do{
256.                 if(c!='\\') add2token(token_text,c);
257.                 if(c=='\\'){
258.                     c=fgetc(fp);
259.                     if (c == '\n'){
260.                         line_cnt++;
261.                     }
262.                     else if (c == '"') return ERROR_TOKEN;
263.                 }
264.             }while ((c=fgetc(fp)) != '"' && c!='\n');
265.             if (c == '"'){
266.                 add2token(token_text, c);
267.                 return STRING_CONST;
268.             }
269.             if (c == '\n') return ERROR_TOKEN;
270.         }
271.         /*除号与注释*/
272.         if (c == '/'){
273.             add2token(token_text, c);
274.             c = fgetc(fp);
275.             if (c == '/'){
276.                 do{
277.                     add2token(token_text, c);
278.                     c = fgetc(fp);
279.                 } while (c != '\n');
280.                 ungetc(c, fp);
281.                 return ANNO;
282.             }
283.             else if (c == '*'){
284.                 while (1){
285.                     add2token(token_text, c);
286.                     c = fgetc(fp);
287.                     if (c == '*'){
288.                         if ((c = fgetc(fp)) == '/'){
289.                             add2token(token_text, '*');
290.                             add2token(token_text, c);
291.                             return ANNO;
```

```
292.                     }
293.                 }
294.                 if (c == '\n'){
295.                     line_cnt++;
296.                 }
297.             }
298.         }
299.         else{
300.             ungetc(c, fp);
301.             return DIVIDE;
302.         }
303.     }
304.     /*头文件和宏*/
305.     if(c == '#'){
306.         add2token(token_text, c);
307.         if(isLetter(c=fgetc(fp))){
308.             do{
309.                 add2token(token_text, c);
310.             } while (isLetter(c = fgetc(fp)));
311.             if(token_text == "#include"){
312.                 do{
313.                     add2token(token_text,c);
314.                 } while ((c = fgetc(fp)) != '\n');
315.                 ungetc(c,fp);
316.                 return INCLUDE;
317.             }
318.             else if(token_text == "#define"){
319.                 do{
320.                     add2token(token_text,c);
321.                 } while ((c = fgetc(fp)) != '\n');
322.                 ungetc(c, fp);
323.                 return MACRO;
324.             }
325.             else{
326.                 return ERROR_TOKEN;
327.             }
328.         }
329.         else{
330.             return ERROR_TOKEN;
331.         }
332.     }
333.     /*非法字符*/
334.     if(c == '@' || c == '?'){
335.         return ERROR_TOKEN;
```

```
336.        }
337.        /*其他符号*/
338.        switch (c){
339.            case ',':
340.                add2token(token_text,c);
341.                return COMMA;
342.            case ';':
343.                add2token(token_text,c);
344.                return SEMI;
345.            case '=':
346.                c=fgetc(fp);
347.                if(c=='='){
348.                    add2token(token_text,c);
349.                    add2token(token_text,c);
350.                    return EQ;
351.                }
352.                ungetc(c,fp);
353.                add2token(token_text,'=');
354.                return ASSIGN;
355.            case '!':
356.                c=fgetc(fp);
357.                if(c=='='){
358.                    add2token(token_text,'!');
359.                    add2token(token_text,'=');
360.                    return NEQ;
361.                }
362.                else{
363.                    return ERROR_TOKEN;
364.                }
365.            case '+':
366.                add2token(token_text,'+');
367.                return PLUS;
368.            case '-':
369.                add2token(token_text,'-');
370.                return MINUS;
371.            case '&':
372.                c = fgetc(fp);
373.                if (c == '&'){
374.                    add2token(token_text, '&');
375.                    add2token(token_text, '&');
376.                    return AND;
377.                }
378.                else{
379.                    return ERROR_TOKEN;
```

```
380.                 }
381.             case '|':
382.                 c = fgetc(fp);
383.                 if (c == '|'){
384.                     add2token(token_text, '|');
385.                     add2token(token_text, '|');
386.                     return OR;
387.                 }
388.                 else{
389.                     return ERROR_TOKEN;
390.                 }
391.             case '(':
392.                 add2token(token_text,c);
393.                 return LP;
394.             case ')':
395.                 add2token(token_text,c);
396.                 return RP;
397.             case '{':
398.                 add2token(token_text,c);
399.                 return LB;
400.             case '}':
401.                 add2token(token_text,c);
402.                 return RB;
403.             case '[':
404.                 add2token(token_text,c);
405.                 return LM;
406.             case ']':
407.                 add2token(token_text,c);
408.                 return RM;
409.             case '*':
410.                 add2token(token_text,c);
411.                 return TIMES;
412.             case '>':
413.                 add2token(token_text,c);
414.                 if((c=fgetc(fp))=='='){
415.                     add2token(token_text,c);
416.                     return MOREEQUAL;
417.                 }
418.                 else{
419.                     ungetc(c,fp);
420.                     return MORE;
421.                 }
422.             case '<':
423.                 add2token(token_text,c);
```

```
424.                 if((c=fgetc(fp))=='='){
425.                     add2token(token_text,c);
426.                     return LESSEQUAL;
427.                 }
428.                 else{
429.                     ungetc(c,fp);
430.                     return LESS;
431.                 }
432.             case '~':  // 终止符
433.                 return -1;
434.             default:
435.                 return ERROR_TOKEN;
436.         }
437.     }
```

syntax_analyse.h

```
1.  //
2.  // Created by Mr.K on 2022-08-07.
3.  //
4.
5.  #ifndef SYNTAX_ANALYSE_H
6.  #define SYNTAX_ANALYSE_H
7.
8.  #include <string>
9.
10. using namespace std;
11.
12. typedef struct AstTree Ast;
13. struct AstTree{
14.     Ast* lc;
15.     Ast* rc;
16.     int node_type;  // 存储节点是语法树的什么部分
17.     struct Data{  // 存储节点对应的值以及类型
18.         int data_type;  // 在后面一些地方有用处
19.         string data_data;
20.         Data() : data_type(0) {};
21.     } data;
22.     AstTree() : lc(nullptr), rc(nullptr), node_type(0), data(Data())
    {};
23. };
24.
25. typedef struct VarList VL;
```

```
26. struct VarList{  // 第一个节点作为头节点
27.     string var_name;
28.     VL* next;
29.     VarList() : next(nullptr) {};
30. };
31. 
32. typedef struct VarListStack VLS;
33. struct VarListStack{
34.     VL* local_var_list;  // 放置当前大括号内的局部变量
35.     VLS* next;  // 下一层大括号内的变量
36.     VarListStack() : local_var_list(nullptr), next(nullptr) {};
37. };
38. 
39. 
40. /*语法树节点类型*/
41. typedef enum NodeKind{
42.     // 外部定义序列
43.     EXT_DEF_LIST = 1,
44.     // 外部变量定义
45.     EXT_VAR_DEF,
46.     // 外部变量类型
47.     EXT_VAR_TYPE,
48.     // 外部变量
49.     EXT_VAR,
50.     // 外部变量序列
51.     EXT_VAR_LIST,
52.     //数组定义
53.     ARRAY_DEF,
54.     //数组类型
55.     ARRAY_TYPE,
56.     // 数组名
57.     ARRAY_NAME,
58.     // 函数定义
59.     FUNC_DEF,
60.     // 函数返回类型
61.     FUNC_RETURN_TYPE,
62.     // 函数名
63.     FUNC_NAME,
64.     // 函数声明
65.     FUNC_CLAIM,
66.     // 函数体
67.     FUNC_BODY,
68.     // 函数形参列表
69.     FUNC_PARA_LIST,
```

```
70.     // 函数形参定义
71.     FUNC_PARA_DEF,
72.     // 函数形参类型
73.     FUNC_PARA_TYPE,
74.     // 函数形参名
75.     FUNC_PARA_NAME,
76.     // 局部变量定义序列
77.     LOCAL_VAR_DEF_LIST,
78.     // 局部变量定义
79.     LOCAL_VAR_DEF,
80.     // 局部变量类型
81.     LOCAL_VAR_TYPE,
82.     // 局部变量名序列
83.     LOCAL_VAR_NAME_LIST,
84.     // 局部变量名
85.     LOCAL_VAR_NAME,
86.     // 语句序列处理
87.     STATE_LIST,
88.     // if 复合语句
89.     IF_STATE,
90.     // if-else 复合语句
91.     IF_ELSE_STATE,
92.     // if 语句块
93.     IF_PART,
94.     // if 条件
95.     IF_COND,
96.     // else 部分
97.     ELSE_PART,
98.     // while 复合语句
99.     WHILE_STATE,
100.        // while 条件部分
101.        WHILE_COND,
102.        // while 语句块
103.        WHILE_BODY,
104.        // for 复合语句
105.        FOR_STATE,
106.        // for 条件部分
107.        FOR_COND,
108.        // for 条件 1\2\3
109.        FOR_COND_1,
110.        FOR_COND_2,
111.        FOR_COND_3,
112.        // for 语句块
113.        FOR_BODY,
```

```
114.        // return 语句
115.        RETURN_STATE,
116.        // break 语句
117.        BREAK_STATE,
118.        // continue 语句
119.        CONTINUE_STATE,
120.        // 运算符
121.        OPERATOR,
122.        // 操作数
123.        OPERAND,
124.        // 表达式
125.        EXPRESSION,
126.
127.    }node_kind;
128.
129.
130.
131.
132.    /*错误类型*/
133.    typedef enum ErrorKind{
134.        // 外部变量定义错误
135.        EXT_DEF_ERR,
136.        // 变量类型错误
137.        VAR_TYPE_ERR,
138.        // 外部定义重复
139.        VAR_DEF_DUPLICATED,
140.        //语法错误
141.        GRAMMA_ERR,
142.        // 函数定义处错误
143.        FUNC_DEF_ERR,
144.        // 函数形参定义出错
145.        FUNC_PARA_DEF_ERR,
146.        // 复合语句内错误
147.        COMP_LIST_ERR,
148.        // 局部变量定义错误
149.        LOCAL_VAR_DEF_ERR,
150.        // 局部变量名重复
151.        LOCAL_VAR_DUPLICATED,
152.        // if 语句错误
153.        IF_ERR,
154.        // while 语句错误
155.        WHILE_ERR,
156.        // for 语句错误
157.        FOR_ERR,
```

```
158.        // void 函数有返回值
159.        VOID_FUNC_WITH_RETURN,
160.        // break 语句缺少分号
161.        BREAK_LACK_SEMI,
162.        // 在非循环中出现 break
163.        BREAK_IN_UNCYCLE,
164.        // continue 缺少分号
165.        CONTINUE_LACK_SEMI,
166.        // 在非循环中出现 continue
167.        CONTINUE_IN_UNCYCLE,
168.        // 表达式出现未知符号
169.        UNKNOWN_TOKEN,
170.        // 表达式处错误
171.        EXPRESSION_ERR,
172.        // 未定义变量
173.        UNDEF_VAR,
174.
175.    }error_kind;
176.
177.
178.
179.
180.    // 抛出错误
181.    void raise_error(int line_cnt, int error_type);
182.    // 常用一系列错误处理
183.    void deal_with_error_1(int line_cnt, int error_type, Ast* cur_node
    );
184.    // 先序遍历
185.    void preorder(Ast* cur_node, int depth);
186.    // 打印节点
187.    void print_node(Ast* cur_node);
188.    // 回收节点
189.    void free_tree(Ast* root);
190.    //获取第一个不是注释和头文件的 token
191.    void filter_anno_include();
192.    // 将 token 返回到文件流
193.    void return_token(FILE* fp);
194.    //向某个域的定义添加变量
195.    int add_var_name(string token_text);
196.    // 查看变量名是否被定义过
197.    int check_name_exist(string token_text);
198.    //最终调用函数
199.    void syntax_analyse();
200.    // 一开始的语法分析，进行一些预处理以及调用 extern_def_list
```

```
201.    Ast* begin_program();
202.    // 外部定义序列
203.    Ast* extern_def_list();
204.    // 辅助函数 extern_def 处理普通变量定义、数组定义还是函数定义
205.    Ast* extern_def();
206.    // 外部变量定义节点
207.    Ast* extern_var_def();
208.    // extern_var_list 类型
209.    Ast* extern_var_list();
210.    // 数组定义节点
211.    Ast* arr_def();
212.    // 函数定义节点
213.    Ast* func_def();
214.    // 函数形参列表
215.    Ast* para_list();
216.    // 函数形参节点
217.    Ast* para_def();
218.    // 大括号扩着的复合语句
219.    Ast* comp_state();
220.    // 局部变量定义序列节点
221.    Ast* local_var_def_list();
222.    // 局部变量定义节点，处理一行局部变量定义
223.    Ast* local_var_def();
224.    // 语句序列处理
225.    Ast* state_list();
226.    // 一条语句处理
227.    Ast* single_state();
228.    // 处理 if 语句块
229.    void deal_with_if(Ast* cur_node);
230.    // 处理 while 语句块
231.    void deal_with_while(Ast* cur_node);
232.    // 处理 for 语句块
233.    void deal_with_for(Ast* cur_node);
234.    // break 语句
235.    void deal_with_break(Ast* cur_node);
236.    // continue 语句
237.    void deal_with_continue(Ast* cur_node);
238.    // 表达式处理
239.    Ast* expression(int end_sym);
240.    // 优先级比较函数
241.    char precede(int c1,int c2);
242.
243.    #endif
```

syntax_analyse.cpp

```
1.  /
2.  // Created by Mr.K on 2022-08-07.
3.  //
4.
5.  #include "syntax_analyse.h"
6.  #include "get_token.h"
7.  #include <cstdio>
8.  #include <iostream>
9.  #include <stack>
10.
11. extern string token_text;
12. extern int line_cnt;
13. extern FILE *fp;
14. int mistake = 0;
15. int isVoid = 0, isInCyc = 0, hasReturn = 0;
16. int word_type, word_type_cp;
17. string token_text_cp;
18. VLS* var_list_stack;
19.
20. string node_kind_arr[] = {"a", "外部定义序列", "外部变量定义", "外部变量
    类型", "外部变量",\
21.     "外部变量序列", "数组定义", "数组类型", "数组名", "函数定义", "函数返
    回类型",\
22.     "函数名", "函数声明", "函数体", "函数形参列表", "函数形参定义", "函数
    形参类型",\
23.     "函数形参名", "局部变量定义序列", "局部变量定义", "局部变量类型", "局
    部变量名序列",\
24.     "局部变量名", "语句序列处理", "if 复合语句", "if-else 复合语句", "if 语
    句块", \
25.     "if 条件", "else 部分", "while 复合语句", "while 条件部分", "while 语句
    块", \
26.     "for 复合语句", "for 条件部分", "for 条件 1", "for 条件 2", "for 条件
    3", "for 语句块",\
27.     "return 语句","break 语句", "continue 语句", "运算符", "操作数", "表达
    式"};
28.
29. string error_kind_arr[] = {"外部定义处错误", "变量类型错误","变量重复定义
    ","语法错误",\
30.     "函数定义处错误", "函数形参定义出错", "复合语句内错误", "局部变量定义错
    误", "局部变量名重复",\
31.     "if 语句错误", "while 语句错误", "for 语句错误", "void 函数有返回值
    ","break 语句缺少分号", "在非循环中出现 break",\
```

```
32.     "continue 缺少分号", "在非循环中出现 continue", "表达式出现未知符号
    ", "表达式处错误",\
33.     "未定义变量"};
34.
35.
36. // 抛出错误
37. void raise_error(int line_cnt, int error_type){
38.     cout << "在第" << line_cnt << "行出现错误" << endl;
39.     cout << "错误类型: " << error_kind_arr[error_type] << endl;
40.     exit(0);
41. }
42.
43. // 常用一系列错误处理
44. void deal_with_error_1(int line_cnt, int error_type, Ast* cur_node){

45.     free_tree(cur_node);
46.     mistake = 1;
47.     raise_error(line_cnt, error_type);
48. }
49.
50. // 先序遍历
51. void preorder(Ast* cur_node, int depth){
52.     if (cur_node == nullptr) return;
53.     else{
54.         for (int i = 0; i < depth; i++){
55.             cout << "\t";
56.         }
57.         print_node(cur_node);
58.     }
59.     preorder(cur_node->lc, depth+1);
60.     preorder(cur_node->rc, depth);
61. }
62.
63. // 打印节点
64. void print_node(Ast* cur_node){
65.     cout << node_kind_arr[cur_node->node_type] << ":";
66.     if (cur_node->data.data_data.empty()){
67.         cout << endl;
68.     }
69.     else{
70.         cout << cur_node->data.data_data << endl;
71.     }
72. }
73.
```

```
74. // 回收节点
75. void free_tree(Ast* root){
76.     if (root){
77.         free_tree(root->lc);
78.         free_tree(root->rc);
79.         delete root;
80.     }
81. }
82. 
83. //获取第一个不是注释和头文件的 token
84. void filter_anno_include(){
85.     word_type = getToken(fp);
86.     while (word_type == ANNO || word_type == INCLUDE || word_type ==
    MACRO){
87.         word_type = getToken(fp);
88.     }
89. }
90. 
91. // 将 token 返回到文件流
92. void return_token(FILE* fp){
93.     ungetc(' ', fp);  // 去掉会出现 bug!
94.     for (int i = token_text.size()-1; i >= 0; i--){
95.         ungetc(token_text[i], fp);
96.     }
97. }
98. 
99. //向某个域的定义添加变量
100.   int add_var_name(string token_text){
101.       int flag = 0;
102.       VLS* p = var_list_stack;
103.       while (p->next != nullptr){
104.           p = p->next;  // 一直找到当前语句块所在范围
105.       }
106.       VL* cur_var_list = p->local_var_list;
107.       VL *q = cur_var_list;
108.       while (q->next != nullptr){
109.           q = q->next;
110.           if (q->var_name == token_text){
111.               flag = 1;
112.               break;
113.           }
114.       }
115.       if (flag == 1){
116.           mistake = 1;
```

```
117.            raise_error(line_cnt, VAR_DEF_DUPLICATED);
118.            return flag;
119.        }
120.        else{
121.            q->next = new VarList;
122.            q->next->var_name = token_text;
123.            return flag;
124.        }
125.    }
126.
127.    // 查看变量名是否被定义过
128.    int check_name_exist(string token_text){  // 返回 1 表示定义过
129.        if (mistake == 1) return 0;
130.        int flag = 0;
131.        VLS* p = var_list_stack;
132.        while (p->next != nullptr){
133.            p = p->next;
134.        }
135.        VL* q = p->local_var_list;
136.        while (q->next != nullptr){
137.            q = q->next;
138.            if (q->var_name == token_text){
139.                flag = 1;
140.                return 1;
141.            }
142.        }
143.        q = var_list_stack->local_var_list;  // 全局域
144.        while (q->next != nullptr){
145.            q = q->next;
146.            if (q->var_name == token_text){
147.                flag = 1;
148.                return 1;
149.            }
150.        }
151.        if (flag == 0){
152.            raise_error(line_cnt, UNDEF_VAR);
153.            mistake = 1;
154.        }
155.        return flag;
156.    }
157.
158.    //最终调用函数
159.    void syntax_analyse(){
160.        Ast* root = begin_program();
```

```
161.      if (root == nullptr || mistake == 1){
162.          cout << "出现语法错误，语法分析结束" << endl;
163.          return;
164.      }
165.      else{
166.          cout << "语法正确，语法树先序遍历如下：" << endl;
167.          preorder(root, 0);
168.      }
169.  }
170.
171.  // 一开始的语法分析，进行一些预处理以及调用 extern_def_list
172.  Ast* begin_program(){
173.      filter_anno_include();
174.      // 建立最外层变量表，即全局变量
175.      var_list_stack = new VarListStack;
176.      var_list_stack->local_var_list = new VarList;
177.      Ast* cur_node = extern_def_list();
178.      if (cur_node == nullptr){  // 某个地方语法有错
179.          mistake = 1;
180.          return nullptr;
181.      }
182.      else{
183.          cur_node->node_type = EXT_DEF_LIST;
184.          return cur_node;
185.      }
186.  }
187.
188.  // 外部定义序列
189.  Ast* extern_def_list(){
190.      if (mistake == 1 || word_type == -1) return nullptr;
191.      Ast* cur_node = new AstTree;
192.      cur_node->node_type = EXT_DEF_LIST;
193.      cur_node->lc = extern_def();
194.      filter_anno_include();
195.      cur_node->rc = extern_def_list();
196.      return cur_node;
197.  }
198.
199.  // 辅助函数 extern_def 处理普通变量定义、数组定义还是函数定义
200.  Ast* extern_def(){  // 进入时存储着类型判断 token
201.      if (!(word_type >= 8 && word_type <= 14)){
202.          raise_error(line_cnt, EXT_DEF_ERR);
203.      }
204.      word_type_cp = word_type;  // 保存类型
```

```
205.        word_type = getToken(fp);
206.        if (word_type != IDENT && word_type != ARRAY){
207.            raise_error(line_cnt, EXT_DEF_ERR);
208.        }
209.        token_text_cp = token_text;  // 保存变量名
210.        int next_word_type = getToken(fp);  // 看看下一个是啥来判断是否是函数定义
211.        Ast* p = new AstTree;
212.        if (next_word_type == LP){
213.            word_type = next_word_type;  // 现在 w_t 存储的是最新读入的 token 的类型,就是‘（’
214.            p = func_def();
215.        }
216.        else if (word_type == ARRAY){  // 简化处理：数组不会连续定义
217.            if (next_word_type != SEMI && next_word_type != COMMA){
218.                deal_with_error_1(line_cnt-1, EXT_DEF_ERR, p);
219.            }
220.            word_type = next_word_type;
221.            p = arr_def();
222.        }
223.        else{
224.            if (next_word_type != SEMI && next_word_type != COMMA){
225.                deal_with_error_1(line_cnt-1, EXT_DEF_ERR, p);
226.            }
227.            word_type = next_word_type;  // 现在 w_t 存储的是最新读入的 token 的类型
228.            p = extern_var_def();
229.        }
230.        return p;
231.    }
232.
233.    // 外部变量定义节点
234.    Ast* extern_var_def(){  // 进入时 word_type_cp 存储着类型，token_text_cp 存储变量名，w_t、token_text 存储后一个 token
235.        if (mistake == 1) return nullptr;
236.        if (word_type_cp == VOID){
237.            raise_error(line_cnt, VAR_TYPE_ERR);
238.        }
239.        Ast* cur_node = new AstTree;
240.        cur_node->node_type = EXT_VAR_DEF;
241.        Ast* p = new AstTree;
242.        p->node_type = EXT_VAR_TYPE;
243.        p->data.data_type = word_type_cp;
244.        p->data.data_data = token_kind_arr[word_type_cp];
```

```
245.        cur_node->lc = p;
246.        // p 的右子树为 extern_var_list 类型
247.        p->rc = extern_var_list();
248.        return cur_node;
249.    }
250.
251.    // extern_var_list 类型
252.    Ast* extern_var_list(){  // w_t_cp 存储类型，t_t_cp 存储变量名，w_t 存储分隔符类型
253.        if (mistake == 1) return nullptr;
254.        // 当前读入的变量没问题，可以生成
255.        int state = add_var_name(token_text_cp);
256.        if (state == 1){
257.            raise_error(line_cnt, VAR_DEF_DUPLICATED);
258.            return nullptr;
259.        }
260.        Ast* cur_node = new AstTree;
261.        cur_node->node_type = EXT_VAR_LIST;
262.        cur_node->lc = new AstTree;
263.        Ast* p = cur_node->lc;
264.        p->node_type = EXT_VAR;
265.        p->data.data_data = token_text_cp;
266.        // 开始判断
267.        if (word_type != COMMA && word_type != SEMI){
268.            raise_error(line_cnt, GRAMMA_ERR);
269.        }
270.
271.        if (word_type == SEMI){  // 这一行的变量定义结束
272.            return cur_node;
273.        }
274.        // 连续的同种变量定义，注意保持进入函数时的一致性
275.        else if (word_type == COMMA){
276.            word_type = getToken(fp);
277.            if (word_type != IDENT){
278.                raise_error(line_cnt, EXT_DEF_ERR);
279.            }
280.            word_type_cp = word_type;  // 保持一致性
281.            token_text_cp = token_text;
282.            word_type = getToken(fp);
283.            cur_node->rc = extern_var_list();
284.            return cur_node;
285.        }
286.        else{
287.            deal_with_error_1(line_cnt, UNKNOWN_TOKEN, cur_node);
```

```
288.             return nullptr;
289.         }
290.         // 退出时 w_t 保持在最后一个分隔符
291.     }
292.
293.     // 数组定义节点
294.     Ast* arr_def(){  // 进入时 word_type_cp 存储着类型，token_text_cp 存储数组名，w_t、token_text 存储后一个 token,也就是分号
295.         if (mistake == 1) return nullptr;
296.         if (word_type_cp == VOID){
297.             raise_error(line_cnt, VAR_TYPE_ERR);
298.         }
299.         Ast* cur_node = new AstTree;
300.         cur_node->node_type = ARRAY_DEF;
301.         Ast* p = new AstTree;
302.         p->node_type = ARRAY_TYPE;
303.         p->data.data_data = token_kind_arr[word_type_cp];
304.         cur_node->lc = p;
305.         p->rc = new AstTree;  // 数组名节点
306.         p = p->rc;
307.         p->node_type = ARRAY_NAME;
308.         p->data.data_data = token_text_cp;
309.         // 可以再加一个数组大小的节点. to do.
310.         return cur_node;
311.         // 退出时 w_t,t_t 保持在最后一个分隔符
312.     }
313.
314.     // 函数定义节点
315.     Ast* func_def(){  // 进入时 w_t_cp 存储着类型，t_t_cp 存储数组名，w_t、t_t 存储后一个 token,也就是左括号
316.         if (mistake == 1) return nullptr;
317.         Ast* cur_node = new AstTree;
318.         cur_node->node_type = FUNC_DEF;  // 先设为函数定义
319.         Ast* p = new AstTree;
320.         p->node_type = FUNC_RETURN_TYPE;
321.         p->data.data_data = token_kind_arr[word_type_cp];
322.         if (word_type_cp == VOID) isVoid = 1;  // 后面有用
323.         else isVoid = 0;
324.         cur_node->lc = p;
325.         p->rc = new AstTree;  // 函数名节点
326.         p = p->rc;
327.         p->node_type = FUNC_NAME;
328.         p->data.data_data = token_text_cp;
329.         // 进入函数域，变量栈加一层
```

```
330.        VLS * last = var_list_stack;
331.        while (last->next != nullptr){
332.            last = last->next;
333.        }
334.        last->next = new VarListStack;
335.        last->next->local_var_list = new VarList;
336.        p->rc = para_list();  // 结束时读在右括号
337.        // 判断是定义还是声明
338.        filter_anno_include();
339.        if (word_type == SEMI){
340.            cur_node->node_type = FUNC_CLAIM;
341.            // 没有函数体节点
342.        }
343.        else if (word_type == LB){
344.            p->rc->rc = comp_state();
345.            p->rc->rc->node_type = FUNC_BODY;
346.        }
347.        else{
348.            raise_error(line_cnt, FUNC_DEF_ERR);
349.        }
350.        return cur_node;
351.    }
352.
353.    // 函数形参列表
354.    Ast* para_list(){
355.        if (mistake == 1) return nullptr;
356.        filter_anno_include();
357.        Ast* cur_node = new AstTree;
358.        cur_node->node_type = FUNC_PARA_LIST;
359.        if (word_type == RP){
360.            return cur_node;
361.        }
362.        cur_node->lc = para_def();  // 结束时读在右括号
363.        return cur_node;
364.    }
365.
366.    // 函数形参节点
367.    Ast* para_def(){  // 初次进入时 w_t 存的是形参类型
368.        if (mistake == 1) return nullptr;
369.        if (word_type == RP){  // 遇到右括号结束
370.            return nullptr;
371.        }
372.        else if (word_type == COMMA){  // 碰到逗号继续读，这里有个 bug 就
    是连续逗号不会报错
```

```
373.             filter_anno_include();
374.             return para_def();
375.         }
376.         else if (word_type >= 8 && word_type <= 13){  // wt 是形参类型
377.             word_type_cp = word_type;  // wtc 保存形参类型，这里也有 bug 就是后面又是类型也不会报错
378.             filter_anno_include();
379.             if (word_type != IDENT){
380.                 raise_error(line_cnt, FUNC_PARA_DEF_ERR);
381.             }
382.             Ast* cur_node = new AstTree;  // 函数形参定义节点
383.             cur_node->node_type = FUNC_PARA_DEF;
384.             Ast* p = new AstTree;  // 函数形参类型
385.             p->node_type = FUNC_PARA_TYPE;
386.             p->data.data_data = token_kind_arr[word_type_cp];
387.             cur_node->lc = p;
388.             p->rc = new AstTree;
389.             p = p->rc;  // 函数形参名
390.             p->node_type = FUNC_PARA_NAME;
391.             int state = add_var_name(token_text);
392.             if (state == 1){
393.                 mistake = 1;
394.                 raise_error(line_cnt, VAR_DEF_DUPLICATED);
395.                 return nullptr;
396.             }
397.             else{
398.                 p->data.data_data = token_text;
399.             }
400.             filter_anno_include();  // 进入递归前还要读一个
401.             cur_node->rc = para_def();
402.             return cur_node;
403.         }
404.         else{
405.             mistake = 1;
406.             raise_error(line_cnt, FUNC_PARA_DEF_ERR);
407.             return nullptr;
408.         }
409.         // 结束时读在右括号
410.     }
411.
412.     // 大括号扩着的复合语句
413.     Ast* comp_state(){  // 进入时读了个'{'
414.         if (mistake == 1) return nullptr;
415.         Ast* cur_node = new AstTree;  // 语句块节点
```

```
416.        //node_type 先不用写，不知道是哪种块
417.        filter_anno_include();
418.        if (word_type >= 8 && word_type <= 13){
419.            cur_node->lc = local_var_def_list();
420.        }
421.    //    if (cur_node->lc == nullptr){
422.    //        cur_node->lc = new AstTree;
423.    //        cur_node->lc->node_type = LOCAL_VAR_DEF_LIST;
424.    //    }
425.        Ast* p = cur_node->lc;
426.        p->rc = state_list();  // 语句序列处理,出来时会多读一个
427.        if (word_type == RB){
428.            return cur_node;
429.        }
430.        else{
431.            mistake = 1;
432.            raise_error(line_cnt, COMP_LIST_ERR);
433.            return nullptr;
434.        }
435.    }
436.
437.    // 局部变量定义序列节点
438.    Ast* local_var_def_list(){
439.        if (mistake == 1) return nullptr;
440.        Ast* cur_node = new AstTree;
441.        cur_node->node_type = LOCAL_VAR_DEF_LIST;
442.        cur_node->lc = local_var_def();  // 按行为节点单位处理局部变量定
    义
443.        return cur_node;
444.    }
445.
446.    // 局部变量定义节点，处理一行局部变量定义
447.    Ast* local_var_def(){
448.        if (mistake == 1) return nullptr;
449.        Ast* cur_node = new AstTree;
450.        cur_node->node_type = LOCAL_VAR_DEF;
451.        cur_node->lc = new AstTree;  // 局部变量类型节点
452.        Ast* p = cur_node->lc;
453.        p->node_type = LOCAL_VAR_TYPE;
454.        p->data.data_data = token_kind_arr[word_type];
455.        p->rc = new AstTree;  // 局部变量名列表
456.        p = p->rc;
457.        p->node_type = LOCAL_VAR_NAME_LIST;
458.        filter_anno_include();  // 读取一个变量
```

```
459.        if (word_type != IDENT){
460.            deal_with_error_1(line_cnt, LOCAL_VAR_DEF_ERR, cur_node);
461.            return nullptr;
462.        }
463.        p->lc = new AstTree;  // 局部变量名
464.        p = p->lc;
465.        p->node_type = LOCAL_VAR_NAME;
466.        p->data.data_data = token_text;
467.        // 把变量加入到变量表
468.        int state = add_var_name(token_text);
469.        if (state == 1){
470.            deal_with_error_1(line_cnt, LOCAL_VAR_DUPLICATED, cur_node
    );
471.            return nullptr;
472.        }
473.        // 开始循环读取变量
474.        while (1){
475.            filter_anno_include();
476.            if (word_type == SEMI){  // 定义结束
477.                // 多读一个保持一致性
478.                filter_anno_include();
479.                break;
480.            }
481.            else if (word_type == COMMA){
482.                filter_anno_include();
483.                if (word_type != IDENT){
484.                    deal_with_error_1(line_cnt, LOCAL_VAR_DEF_ERR, cur
    _node);
485.                    return nullptr;
486.                }
487.                p->rc = new AstTree;  // 变量名
488.                p = p->rc;
489.                p->node_type = LOCAL_VAR_NAME;
490.                p->data.data_data = token_text;
491.                state = add_var_name(token_text);
492.                if (state == 1){
493.                    deal_with_error_1(line_cnt, LOCAL_VAR_DUPLICATED,
    cur_node);
494.                    return nullptr;
495.                }
496.            }
497.            else{
```

```
498.                 deal_with_error_1(line_cnt, LOCAL_VAR_DEF_ERR, cur_nod
    e);
499.                 return nullptr;
500.             }
501.         }
502.         if (word_type >= 8 && word_type <= 13){
503.             cur_node->rc = local_var_def();  // 正常情况下从这里退出，多
    读一个
504.         }
505.         return cur_node;
506.     }
507.
508.     // 语句序列处理
509.     Ast* state_list(){  // 进入时应该多读一个
510.         if (mistake == 1) return nullptr;
511.         Ast* cur_node = new AstTree;
512.         cur_node->node_type = STATE_LIST;  // 语句序列节点
513.         cur_node->lc = single_state();  // 处理一条语句
514.         if (cur_node->lc == nullptr){
515.             return cur_node;  // 语句序列已经结束
516.         }
517.         else{
518.             filter_anno_include();  // 多读一个准备递归
519.             if (word_type != RB){
520.                 cur_node->rc = state_list();
521.             }
522.             return cur_node;
523.         }
524.     }
525.
526.     // 一条语句处理
527.     Ast* single_state(){  // 进入时多读一个
528.         if (mistake == 1) return nullptr;
529.         Ast* cur_node = new AstTree;
530.         switch (word_type) {
531.             case IF: {
532.                 deal_with_if(cur_node);
533.                 return cur_node;
534.             }
535.             case WHILE: {
536.                 deal_with_while(cur_node);
537.                 return cur_node;
538.             }
539.             case FOR: {
```

```
540.                 deal_with_for(cur_node);
541.                 return cur_node;
542.             }
543.             case RETURN:{
544.                 hasReturn = 1;
545.                 if(isVoid == 1){
546.                     deal_with_error_1(line_cnt, VOID_FUNC_WITH_RETURN,
    cur_node);
547.                 }
548.                 cur_node->node_type = RETURN_STATE;
549.                 filter_anno_include();
550.                 cur_node->lc = expression(SEMI);
551.                 return cur_node;
552.             }
553.             case BREAK: {
554.                 deal_with_break(cur_node);
555.                 return cur_node;
556.             }
557.             case CONTINUE: {
558.                 deal_with_continue(cur_node);
559.                 return cur_node;
560.             }
561.             case INT_CONST:
562.             case FLOAT_CONST:
563.             case CHAR_CONST:
564.             case IDENT:
565.             case ARRAY:
566.                 return expression(SEMI);
567.         }
568.         return cur_node;
569.     }
570.
571.     // 处理 if 语句块
572.     void deal_with_if(Ast* cur_node){
573.         filter_anno_include();
574.         if (word_type != LP){
575.             deal_with_error_1(line_cnt, IF_ERR, cur_node);
576.         }
577.         filter_anno_include();
578.         cur_node->lc = new AstTree;  // if-part 节点
579.         Ast* p = cur_node->lc;
580.         p->node_type = IF_PART;
581.         p->lc = new AstTree;  // condition 节点
582.         Ast* q = p->lc;
```

```
583.        q->node_type = IF_COND;
584.        q->lc = expression(RP);  // condition 下挂的是表达式
585.        if (q->lc == nullptr) {
586.            deal_with_error_1(line_cnt, IF_ERR, cur_node);
587.        }
588.        filter_anno_include();
589.        if (word_type == LB){  // 多语句 if
590.            filter_anno_include();
591.            q->rc = state_list();
592.        }
593.        else if (word_type >= 2 && word_type <= 19){  // 可能的开头
    token
594.            q->rc = single_state();
595.        }
596.        else{
597.            deal_with_error_1(line_cnt, IF_ERR, cur_node);
598.        }
599.        filter_anno_include();
600.        if (word_type == ELSE){
601.            cur_node->node_type = IF_ELSE_STATE;
602.            p->rc = new AstTree;
603.            q = p->rc;
604.            q->node_type = ELSE_PART;
605.            filter_anno_include();
606.            if (word_type == LB){  // 多语句 else
607.                filter_anno_include();
608.                q->lc = state_list();
609.            }
610.            else if (word_type >= 2 && word_type <= 19){  // 可能的开头
    token
611.                q->lc = single_state();
612.            }
613.            else{
614.                deal_with_error_1(line_cnt, IF_ERR, cur_node);
615.            }
616.        }
617.        else{
618.            cur_node->node_type = IF_STATE;
619.            return_token(fp);
620.        }
621.    }
622.
623.    // 处理 while 语句块
624.    void deal_with_while(Ast* cur_node){
```

```
625.        isInCyc = 1;
626.        filter_anno_include();
627.        if (word_type != LP){
628.            deal_with_error_1(line_cnt, WHILE_ERR, cur_node);
629.            return;
630.        }
631.        filter_anno_include();
632.        cur_node->lc = new AstTree();
633.        Ast* p = cur_node->lc;
634.        p->node_type = WHILE_COND;
635.        p->lc = expression(RP);
636.        if (p->lc == nullptr){
637.            deal_with_error_1(line_cnt, WHILE_ERR, cur_node);
638.            return;
639.        }
640.        Ast* q = new AstTree;  // while_body
641.        q->node_type = WHILE_BODY;
642.        filter_anno_include();
643.        if (word_type == LB){  // 语句块
644.            filter_anno_include();  // 读入一个，一致性
645.            q->lc = state_list();
646.        }
647.        else if (word_type >= 2 && word_type <= 19){  // 此时读入的是单
    句的第一个 token
648.            q->lc = single_state();
649.        }
650.        else{
651.            deal_with_error_1(line_cnt, WHILE_ERR, cur_node);
652.            return;
653.        }
654.        p->rc = q;
655.        cur_node->node_type = WHILE_STATE;
656.        isInCyc = 0;
657.    }
658.
659.    // 处理 for 语句块
660.    void deal_with_for(Ast* cur_node){
661.        isInCyc = 1;
662.        filter_anno_include();
663.        if (word_type != LP){
664.            deal_with_error_1(line_cnt, FOR_ERR, cur_node);
665.            return;
666.        }
667.        Ast* p = new AstTree;
```

```
668.        cur_node->lc = p;
669.        p->node_type = FOR_COND;
670.        // 开始处理条件部分
671.        Ast* q = new AstTree;
672.        q->node_type = FOR_COND_1;
673.        p->lc = q;
674.        filter_anno_include();  // 一致性
675.        q->lc = expression(SEMI);
676.        if (q->lc == nullptr){
677.            q->data.data_data = "无";
678.        }
679.        q->rc = new AstTree;
680.        q = q->rc;
681.        q->node_type = FOR_COND_2;
682.        filter_anno_include();
683.        q->lc = expression(SEMI);
684.        if (q->lc == nullptr){
685.            q->data.data_data = "无";
686.        }
687.        q->rc = new AstTree;
688.        q = q->rc;
689.        q->node_type = FOR_COND_3;
690.        filter_anno_include();
691.        q->lc = expression(RP);
692.        if (q->lc == nullptr){
693.            q->data.data_data = "无";
694.        }
695.        // 处理 for_body
696.        q = new AstTree;
697.        q->node_type = FOR_BODY;
698.        filter_anno_include();
699.        if (word_type == LB) {
700.            filter_anno_include();  // 一致性
701.            q->lc = state_list();
702.        }
703.        else if (word_type >= 2 && word_type <= 19){
704.            q->lc = single_state();
705.        }
706.        else{
707.            deal_with_error_1(line_cnt, FOR_ERR, cur_node);
708.            return;
709.        }
710.        cur_node->node_type = FOR_STATE;
711.        isInCyc = 0;
```

```
712.  }
713.
714.  // break 语句
715.  void deal_with_break(Ast* cur_node){
716.      filter_anno_include();
717.      if (word_type != SEMI){
718.          deal_with_error_1(line_cnt,BREAK_LACK_SEMI, cur_node);
719.          return;
720.      }
721.      if (isInCyc == 0){
722.          deal_with_error_1(line_cnt, BREAK_IN_UNCYCLE, cur_node);
723.          return;
724.      }
725.      cur_node->node_type = BREAK_STATE;
726.  }
727.
728.  // continue 语句
729.  void deal_with_continue(Ast* cur_node){
730.      filter_anno_include();
731.      if (word_type != SEMI){
732.          deal_with_error_1(line_cnt,CONTINUE_LACK_SEMI, cur_node);
733.          return;
734.      }
735.      if (isInCyc == 0){
736.          deal_with_error_1(line_cnt, CONTINUE_IN_UNCYCLE,cur_node);
737.          return;
738.      }
739.      cur_node->node_type = CONTINUE_STATE;
740.  }
741.
742.  // 表达式处理
743.  Ast* expression(int end_sym){  // 进入时已经读入一个单词
744.      if (mistake == 1) return nullptr;
745.      if (word_type == end_sym) return nullptr;  // 空语句
746.      stack<Ast*> op;  // 存放运算符
747.      stack<Ast*> num;  // 存放操作数
748.      Ast* op_node;
749.      Ast* num_node;
750.      while (word_type != end_sym){
751.          if (word_type == IDENT && check_name_exist(token_text) !=
    1){
752.              raise_error(line_cnt, UNDEF_VAR);
```

```
753.             }
754.             if (word_type == IDENT || word_type == INT_CONST || word_t
    ype == FLOAT_CONST || word_type == CHAR_CONST || word_type == ARRAY |
    | word_type == STRING_CONST){
755.                 num_node = new AstTree();
756.                 num_node->node_type = OPERAND;
757.                 num_node->data.data_data = token_text;
758.                 num.push(num_node);
759.                 filter_anno_include();  // 为下一轮做准备
760.             }
761.             else{
762.                 if (word_type == RP){
763.                     if (num.size() < 2 || op.size() < 2){
764.                         raise_error(line_cnt, EXPRESSION_ERR);  // to
    do. 清理栈内存
765.                         return nullptr;
766.                     }
767.                     while (op.top()->data.data_type != LP){
768.                         op_node = op.top();  // 获取一个运算符
769.                         op.pop();
770.                         op_node->node_type = OPERATOR;
771.                         if (num.size() < 2){
772.                             raise_error(line_cnt, EXPRESSION_ERR);
773.                             return nullptr;
774.                         }
775.                         else{
776.                             op_node->rc = num.top();  // 获取两个操作
    数
777.                             num.pop();
778.                             op_node->lc = num.top();
779.                             num.pop();
780.                             num.push(op_node);
781.                         }
782.                     }
783.                     op.pop();  // 弹出左括号
784.                 }
785.                 else if (word_type == LP){
786.                     op_node = new AstTree();
787.                     op_node->node_type = OPERATOR;
788.                     op_node->data.data_data = token_text;
789.                     op_node->data.data_type = LP;  // data_type 的作用只
    是优先级比较
790.                     op.push(op_node);
791.                 }
```

```
792.                else if (word_type >= EQ && word_type <= AND){
793.                    if (op.empty()){  // 栈里面还没有运算符
794.                        op_node = new AstTree();
795.                        op_node->node_type = OPERATOR;
796.                        op_node->data.data_data = token_text;
797.                        op_node->data.data_type = word_type;
798.                        op.push(op_node);
799.                    }
800.                    else{
801.                        switch (precede(op.top()->data.data_type, word
    _type)) {
802.                            case '>':  // 要先把上一个表达式计算出来
803.                                if (num.size() < 2 || op.size() < 1){

804.                                    raise_error(line_cnt, EXPRESSION_E
    RR);  // to do. 清理栈内存
805.                                    return nullptr;
806.                                }
807.                                op_node = op.top();
808.                                op.pop();
809.                                op_node->node_type = OPERATOR;
810.                                op_node->rc = num.top();
811.                                num.pop();
812.                                op_node->lc = num.top();
813.                                num.pop();
814.                                num.push(op_node);
815.                                op_node = new AstTree();  // 然后把这一
    轮的运算符压入栈中
816.                                op_node->node_type = OPERATOR;
817.                                op_node->data.data_data = token_text;

818.                                op_node->data.data_type = word_type;
819.                                op.push(op_node);
820.                                break;
821.                            case '<':  // 直接压入栈中
822.                                op_node = new AstTree();
823.                                op_node->node_type = OPERATOR;
824.                                op_node->data.data_data = token_text;

825.                                op_node->data.data_type = word_type;
826.                                op.push(op_node);
827.                                break;
828.                            case '\0':
```

```
829.                                 raise_error(line_cnt, EXPRESSION_ERR);
    // to do. 清理栈内存
830.                                 return nullptr;
831.                             default:
832.                                 break;
833.                         }
834.                     }
835.                 }
836.                 else{
837.                     raise_error(line_cnt, EXPRESSION_ERR);  // to do.
    清理栈内存
838.                     return nullptr;
839.                 }
840.                 filter_anno_include();  // 为下一轮做准备
841.             }
842.         }
843.         if (op.empty() && num.empty()){  // 空语句
844.             return nullptr;
845.         }
846.         else if (op.empty() && num.size() == 1){
847.             Ast* cur_node = new AstTree();
848.             cur_node->node_type = EXPRESSION;
849.             cur_node->lc = num.top();
850.             num.pop();
851.             return cur_node;
852.         }
853.         else if (op.size() >= 1 && num.size() >= 2){
854.             while (!op.empty() && num.size() >= 2){
855.                 op_node = op.top();
856.                 op.pop();
857.                 op_node->node_type = OPERATOR;
858.                 op_node->rc = num.top();
859.                 num.pop();
860.                 op_node->lc = num.top();
861.                 num.pop();
862.                 num.push(op_node);
863.             }
864.             if (op.empty() && num.size() == 1){
865.                 Ast* cur_node = new AstTree();
866.                 cur_node->node_type = EXPRESSION;
867.                 cur_node->lc = op_node;
868.                 return cur_node;
869.             }
870.             else{
```

```
871.                 raise_error(line_cnt, EXPRESSION_ERR);  // to do. 清理
    栈内存
872.                 return nullptr;
873.             }
874.         }
875.         else{
876.             raise_error(line_cnt, EXPRESSION_ERR);  // to do. 清理栈内
    存
877.             return nullptr;
878.         }
879.     }
880.
881.     // 优先级比较函数
882.     char precede(int c1,int c2){
883.         if(c1==PLUS||c1==MINUS){
884.             switch (c2){
885.                 case PLUS:
886.                 case MINUS:
887.                 //case RP:
888.                 //case POUND:
889.                 case MORE:
890.                 case LESS:
891.                 case MOREEQUAL:
892.                 case LESSEQUAL:
893.                 case EQ:
894.                 case NEQ:
895.                 case ASSIGN:
896.                     return '>';
897.                 case TIMES:
898.                 case DIVIDE:
899.                 case LP:
900.                 case AND:
901.                 case OR:
902.                     return '<';
903.                 default:
904.                     return '\0';
905.                     break;
906.
907.             }
908.         }
909.         else if(c1==TIMES||c1==DIVIDE){
910.             switch (c2){
911.                 case PLUS:
912.                 case MINUS:
```

```
913.                 //case LP:
914.                 //case POUND:
915.                 case TIMES:
916.                 case DIVIDE:
917.                 case MORE:
918.                 case LESS:
919.                 case MOREEQUAL:
920.                 case LESSEQUAL:
921.                 case EQ:
922.                 case NEQ:
923.                 case ASSIGN:
924.                     return '>';
925.                 case LP:
926.                 case AND:
927.                 case OR:
928.                     return '<';
929.                 default:
930.                     return '\0';
931.             }
932.         }
933.         else if (c1==LP){
934.             switch (c2){
935.                 case PLUS:
936.                 case MINUS:
937.                 case TIMES:
938.                 case DIVIDE:
939.                 case LP:
940.                 case AND:
941.                 case OR:
942.                 case MORE:
943.                 case LESS:
944.                 case MOREEQUAL:
945.                 case LESSEQUAL:
946.                 case EQ:
947.                 case NEQ:
948.                     return '<';
949.                 default:
950.                     return '\0';
951.             }
952.         }
953.     //    else if(c1==RP){
954.     //        switch (c2){
955.     //            case PLUS:
956.     //            case MINUS:
```

```
957.  //            case TIMES:
958.  //            case DIVIDE:
959.  //            case LP:
960.  //            case MORE:
961.  //            case LESS:
962.  //            case MOREEQUAL:
963.  //            case LESSEQUAL:
964.  //            case EQ:
965.  //            case NEQ:
966.  //            case PLUSPLUS:
967.  //            case MINUSMINUS:
968.  //            case POUND:
969.  //                return '>';
970.  //            default:
971.  //                return '\0';
972.  //        }
973.  //    }
974.      else if(c1==ASSIGN){
975.          switch (c2){
976.              case PLUS:
977.              case MINUS:
978.              case TIMES:
979.              case DIVIDE:
980.              case LP:
981.              case MORE:
982.              case LESS:
983.              case MOREEQUAL:
984.              case LESSEQUAL:
985.              case EQ:
986.              case NEQ:
987.              case AND:
988.              case OR:
989.                  return '<';
990.  //            case POUND:
991.  //                return '>';
992.              default:
993.                  return '\0';
994.          }
995.      }
996.      else if(c1==MORE||c1==LESS||c1==MOREEQUAL||c1==LESSEQUAL){
997.          switch (c2){
998.              case PLUS:
999.              case MINUS:
1000.             case TIMES:
```

```
1001.                 case DIVIDE:
1002.                 case LP:
1003.                 case AND:
1004.                 case OR:
1005.                     return '<';
1006.                 //case RP:
1007.                 case MORE:
1008.                 case LESS:
1009.                 case MOREEQUAL:
1010.                 case LESSEQUAL:
1011.                 case EQ:
1012.                 case NEQ:
1013.                 //case POUND:
1014.                     return '>';
1015.                 default:
1016.                     return '\0';
1017.             }
1018.         }
1019.         else if(c1==EQ||c1==NEQ){
1020.             switch (c2){
1021.                 case PLUS:
1022.                 case MINUS:
1023.                 case TIMES:
1024.                 case DIVIDE:
1025.                 case LP:
1026.                 case MORE:
1027.                 case LESS:
1028.                 case MOREEQUAL:
1029.                 case LESSEQUAL:
1030.                 case AND:
1031.                 case OR:
1032.                     return '<';
1033.                 //case RP:
1034.                 case EQ:
1035.                 case NEQ:
1036.                 //case POUND:
1037.                     return '>';
1038.                 default:
1039.                     return '\0';
1040.             }
1041.         }
1042. //    else if(c1==POUND){
1043. //        switch (c2){
1044. //            case PLUS:
```

```
1045.  //            case MINUS:
1046.  //            case TIMES:
1047.  //            case DIVIDE:
1048.  //            case LP:
1049.  //            case MORE:
1050.  //            case LESS:
1051.  //            case MOREEQUAL:
1052.  //            case LESSEQUAL:
1053.  //            case RP:
1054.  //            case EQ:
1055.  //            case NEQ:
1056.  //            case ASSIGN:
1057.  //            case PLUSPLUS:
1058.  //            case MINUSMINUS:
1059.  //                return '<';
1060.  //            case POUND:
1061.  //                return '=';
1062.  //            default:
1063.  //                return '\0';
1064.  //        }
1065.  //    }
1066.      else if(c1 == AND||c1 == OR){
1067.          switch (c2){
1068.              case PLUS:
1069.              case MINUS:
1070.              case TIMES:
1071.              case DIVIDE:
1072.              //case LP:
1073.              case ASSIGN:
1074.              //case POUND:
1075.                  return '>';
1076.              case LP:
1077.              case MORE:
1078.              case LESS:
1079.              case MOREEQUAL:
1080.              case LESSEQUAL:
1081.              case EQ:
1082.              case NEQ:
1083.                  return '<';
1084.              default:
1085.                  return '\0';
1086.          }
1087.      }
1088.      return '\0';  // 不知道会不会出现这种情况
```

```
1089.  }
```

format_operation.h

```
1.  //
2.  // Created by Mr.K on 2022-08-28.
3.  //
4.
5.  #ifndef MY_TINY_COMPILER_FORMAT_OPERATION_H
6.  #define MY_TINY_COMPILER_FORMAT_OPERATION_H
7.
8.  #include <vector>
9.  #include <string>
10.
11.
12.
13. // 读取每个部分
14. int read_part(std::vector<std::string> &sep_part);
15.
16. void print_part(std::vector<std::string> sep_part);
17.
18. // 格式化主函数
19. void format();
20.
21.
22.
23. #endif //MY_TINY_COMPILER_FORMAT_OPERATION_H
```

format_operation.cpp

```
1.  //
2.  // Created by Mr.K on 2022-08-28.
3.  //
4.
5.  #include "format_operation.h"
6.  #include "get_token.h"
7.  #include <iostream>
8.  #include <fstream>
9.
10. extern FILE* fp;
11. extern string token_text;
```

```
int word_type0;
int tabs;
ofstream fout;

// 存储每个部分的 token
std::vector<std::string> sep_part;

int read_part(std::vector<std::string> &sep_part){
    word_type0 = getToken(fp);
    while (word_type0 == ANNO){
        word_type0 = getToken(fp);
    }
    if (word_type0 == -1){  // 读到结尾
        return 0;
    }
    if (word_type0 == INCLUDE || word_type0 == MACRO){  // 根据头文件和宏结束表达
        sep_part.push_back(token_text);
        return 1;
    }
    if (word_type0 == FOR){
        while (word_type0 != RP){
            sep_part.push_back(token_text);
            word_type0 = getToken(fp);
        }
        sep_part.push_back(token_text);
        return read_part(sep_part);
    }
    if (word_type0 == SEMI){  // 根据分号结束一句表达
        sep_part.push_back(token_text);
        return 1;
    }
    if (word_type0 == LB){  // 根据大括号结束一句表达
        sep_part.push_back(token_text);
        //tabs++;
        return 1;
    }
    if (word_type0 == RB){  // 根据右括号结束一句表达
        sep_part.push_back(token_text);
        //tabs--;
        return 1;
    }
    // 普通 token
    sep_part.push_back(token_text);
```

```
55.     return read_part(sep_part);
56. }
57. 
58. void print_part(std::vector<std::string> sep_part){
59.     if (sep_part[sep_part.size() - 1] == "}") tabs--;
60.     for (int i = 0; i < tabs; i++){
61.         std::cout << '\t';
62.         fout << '\t';
63.     }
64.     for (string part : sep_part){
65.         if (part == "{") tabs++;
66.         std::cout << part << " ";
67.         fout << part << " ";
68.     }
69.     std::cout << endl;
70.     fout << endl;
71. }
72. 
73. void format(){
74.     fout.open("../out.c");
75.     while (read_part(sep_part)){
76.         print_part(sep_part);
77.         sep_part.clear();
78.     }
79.     fout.close();
80. }
```

CMakeList.txt

```
1.  cmake_minimum_required(VERSION 3.22)
2.  project(my_tiny_compiler)
3.  
4.  set(CMAKE_CXX_STANDARD 14)
5.  
6.  set(CMAKE_EXE_LINKER_FLAGS -static)
7.  
8.  add_executable(my_tiny_compiler main.cpp get_token.cpp get_token.h sy
    ntax_analyse.cpp syntax_analyse.h format_operation.cpp format_operati
    on.h)
```