

**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 1](#_Toc115888576)

[1引言 3](#_Toc115888577)

[1.1课题背景与意义 3](#_Toc115888578)

[1.2国内外研究现状 3](#_Toc115888579)

[1.2.1 抽象语法树（AST） 3](#_Toc115888580)

[1.2.2 形式语言自动机 4](#_Toc115888581)

[1.3课程设计的主要研究工作 4](#_Toc115888582)

[2系统需求分析与总体设计 5](#_Toc115888583)

[2.1系统需求分析 5](#_Toc115888584)

[2.2系统总体设计 5](#_Toc115888585)

[3系统详细设计 8](#_Toc115888586)

[3.1有关数据结构的定义 8](#_Toc115888587)

[3.2 主要算法设计 12](#_Toc115888588)

[4系统实现与测试 15](#_Toc115888589)

[4.1系统实现 15](#_Toc115888590)

[4.2系统测试 17](#_Toc115888591)

[5总结与展望 24](#_Toc115888592)

[5.1全文总结 24](#_Toc115888593)

[5.1工作展望 24](#_Toc115888594)

[6 体会 25](#_Toc115888595)

[参考文献 26](#_Toc115888596)

[附录一：test.txt测试文件 27](#_Toc115888597)

[附录二：词法分析完整输出 29](#_Toc115888598)

[附录三：语法分析完整输出 36](#_Toc115888599)

[附录四：项目源码 44](#_Toc115888600)

[Main.cpp 44](#_Toc115888601)

[get\_token.h 45](#_Toc115888602)

[get\_token.cpp 48](#_Toc115888603)

[syntax\_analyse.h 58](#_Toc115888604)

[syntax\_analyse.cpp 64](#_Toc115888605)

[format\_operation.h 90](#_Toc115888606)

[format\_operation.cpp 90](#_Toc115888607)

[CMakeList.txt 92](#_Toc115888608)

# 任务书

* **设计内容**

在计算机科学中，抽象语法树（abstract syntax tree或者缩写为AST），是将[源代码](https://baike.baidu.com/item/%E6%BA%90%E4%BB%A3%E7%A0%81" \t "_blank)的语法结构的用树的形式表示，树上的每个结点都表示源程序代码中的一种语法成分。之所以说是“抽象”，是因为在抽象语法树中，忽略了源程序中语法成分的一些细节，突出了其主要语法特征。

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

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

* **设计要求**

要求具有如下功能：

1. **语言定义**

选定C语言的一个子集，要求包含：

（1）基本数据类型的变量、常量，以及数组。不包含指针、结构，枚举等。

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

（3）函数定义、声明与调用。

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

（5）编译预处理（宏定义，文件包含）

（6）注释（块注释与行注释）

2. **单词识别**

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

3. **语法结构分析**

（1）外部变量的声明；

（2）函数声明与定义；

（3）局部变量的声明；

（4）语句及表达式；

（5）生成（1)-(4)（包含编译预处理和注释）的抽象语法树并显示。

4. **按缩进编排生成源程序文件**。

* **参考文献**

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

[2] 严蔚敏等.数据结构(C语言版).北京：清华大学出版社

# 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](https://baike.baidu.com/item/%E6%BA%90%E4%BB%A3%E7%A0%81" \t "_blank)的语法结构的用树的形式表示，树上的每个结点都表示源程序代码中的一种语法成分。之所以说是“抽象”，是因为在抽象语法树中，忽略了源程序中语法成分的一些细节，突出了其主要语法特征。

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

### 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=（∑，Q，δ，q0，F）。其中∑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∑的直积Q×∑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)：

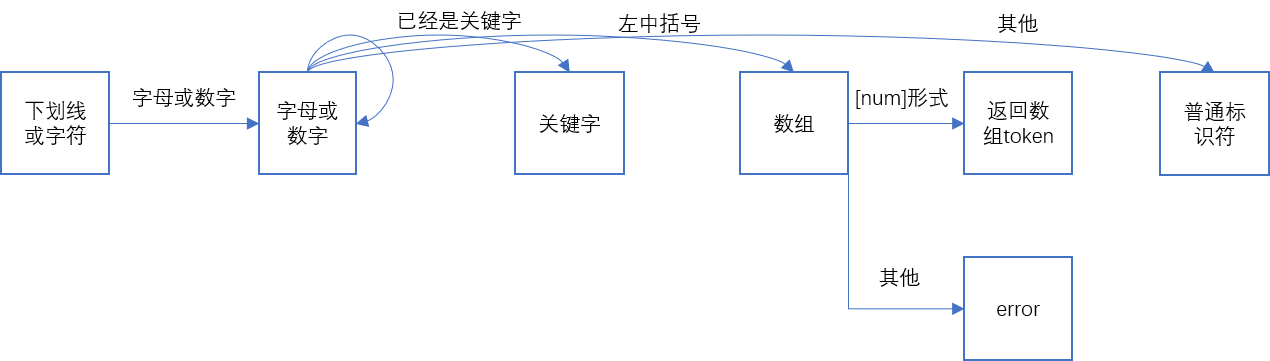


图2-1 有限状态机进行词法分析示例

#### 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）：

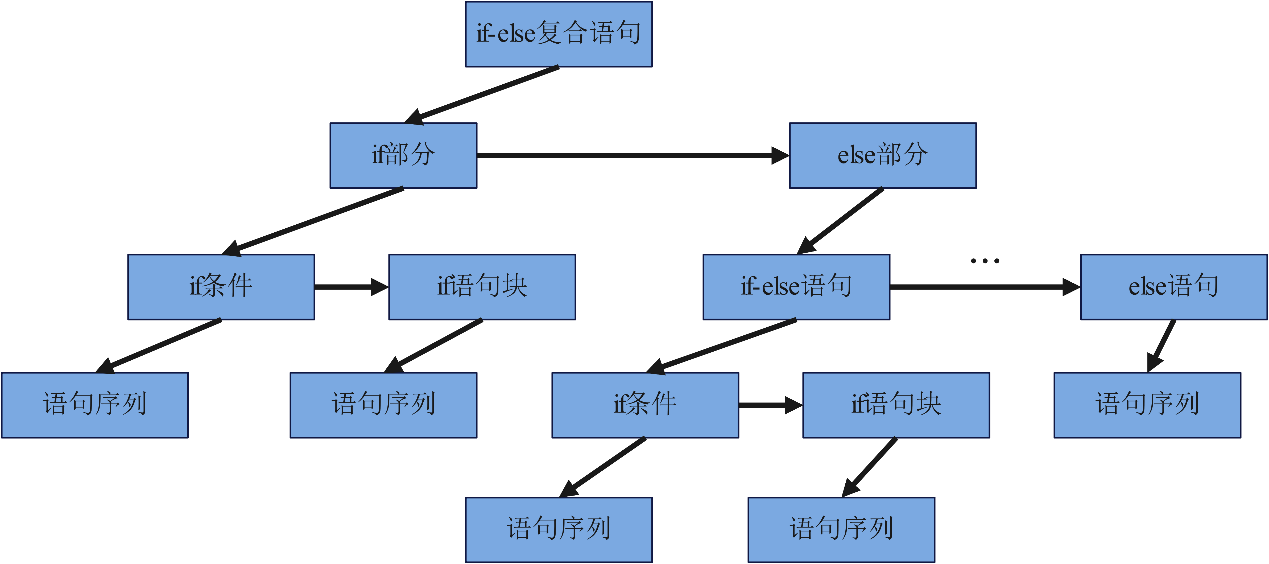


图2-2 数据结构示意图

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。

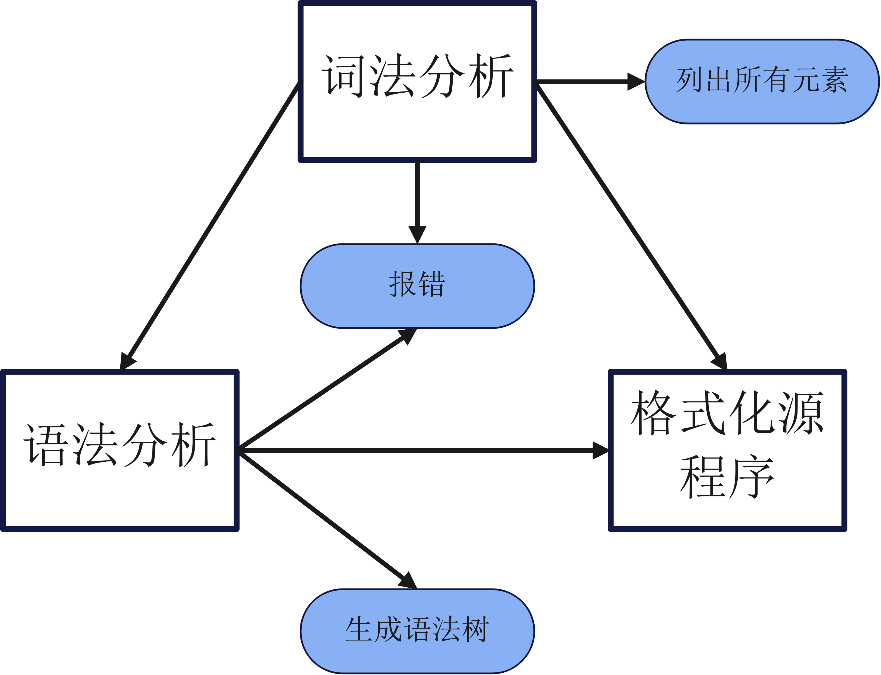


图2-3 系统总体设计

# 3系统详细设计

## 3.1有关数据结构的定义

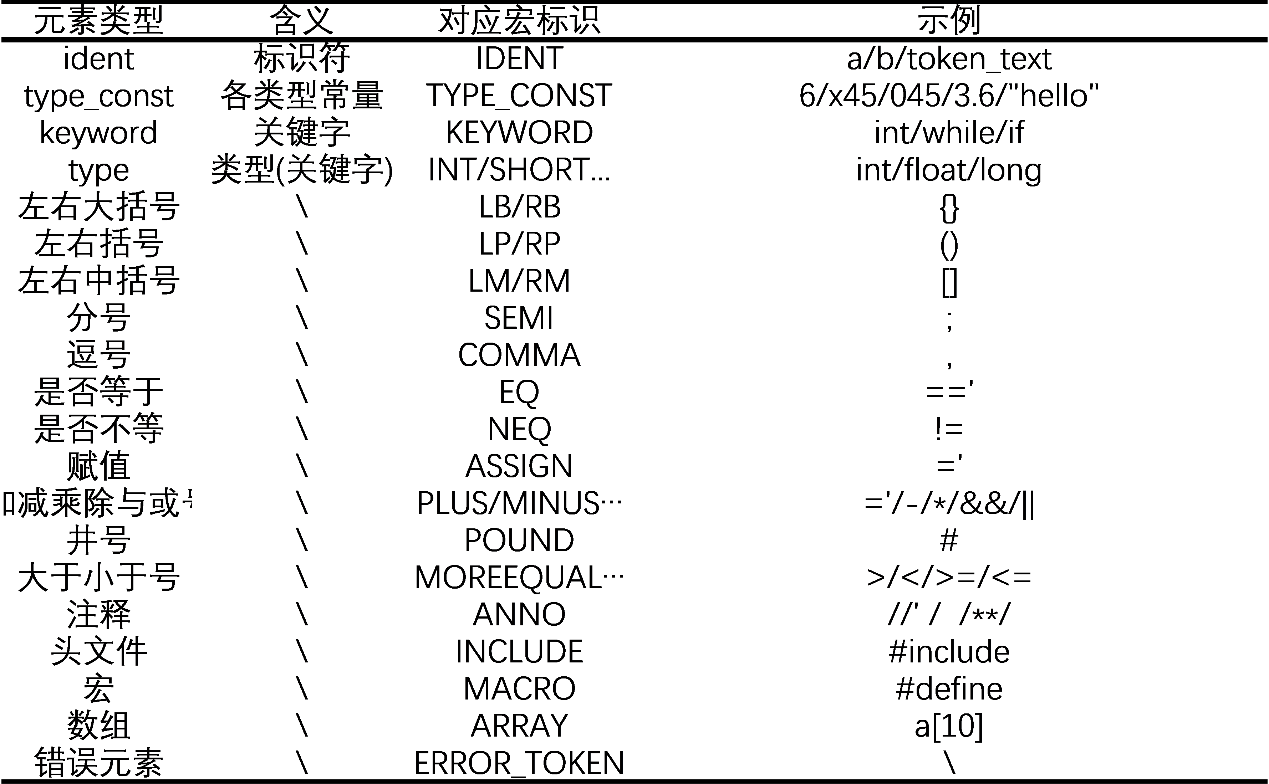
给定一段程序，在C语言的某个子集下，有标识符、常量、关键字(包括类型关键字)、括号、大括号、逗号、分号、赋值、运算、等与不等、注释、头文件、函数定义、函数声明、函数调用、数组、宏等元素。在词法分析中，要求按类别列出所有子集元素。各种元素的类别、含义与示例如图3-1。

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。

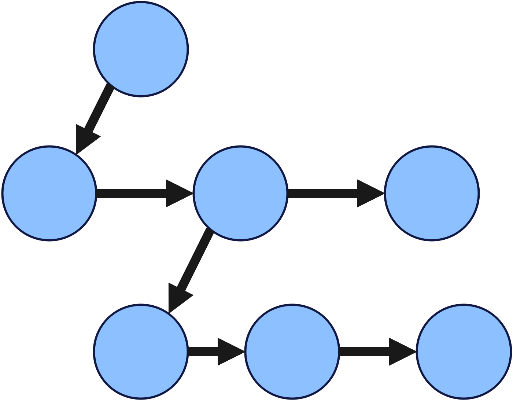
多叉树的数据结构选用了孩子兄弟表示法(左孩子右兄弟)，其示意图如下：

图3-2 多叉树示意图

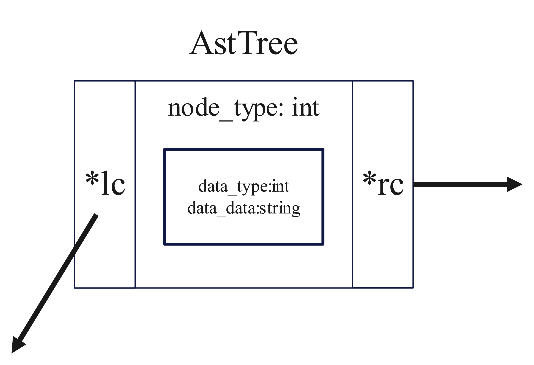
存储多叉树通过一个结构体进行链式存储。结构体的定义如下图所示：

图3-3 语法树节点结构

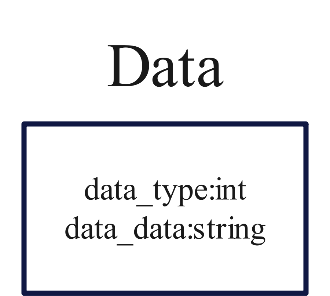
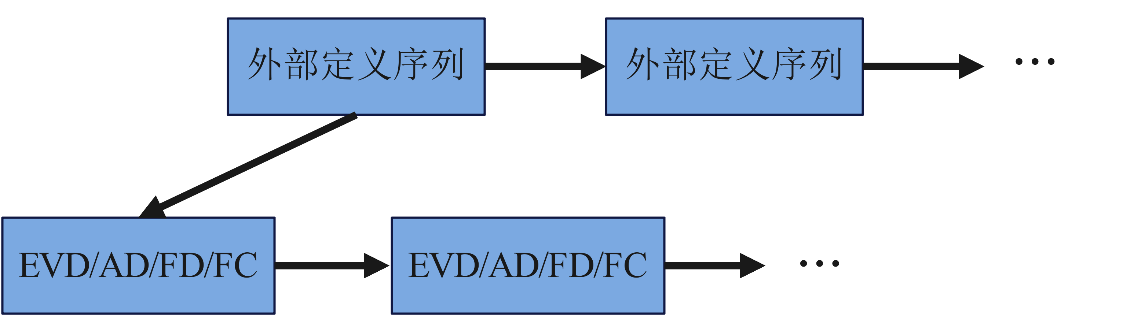
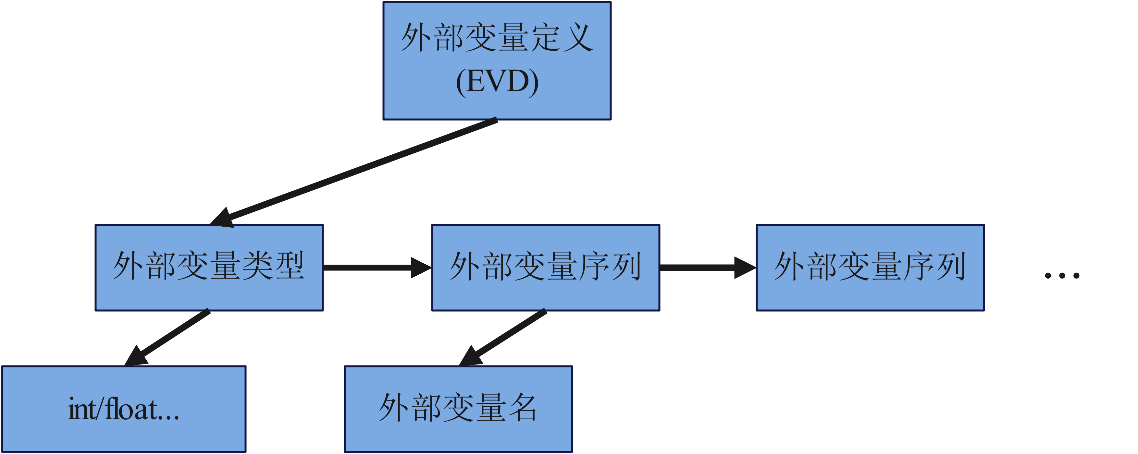
 其中，nodetype表示当前语句所属分支，lc和rc均为节点指针，data\_type和data\_data属于data结构体，包含语句的一些信息，用于后续输出多叉树和优先级比较。

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：

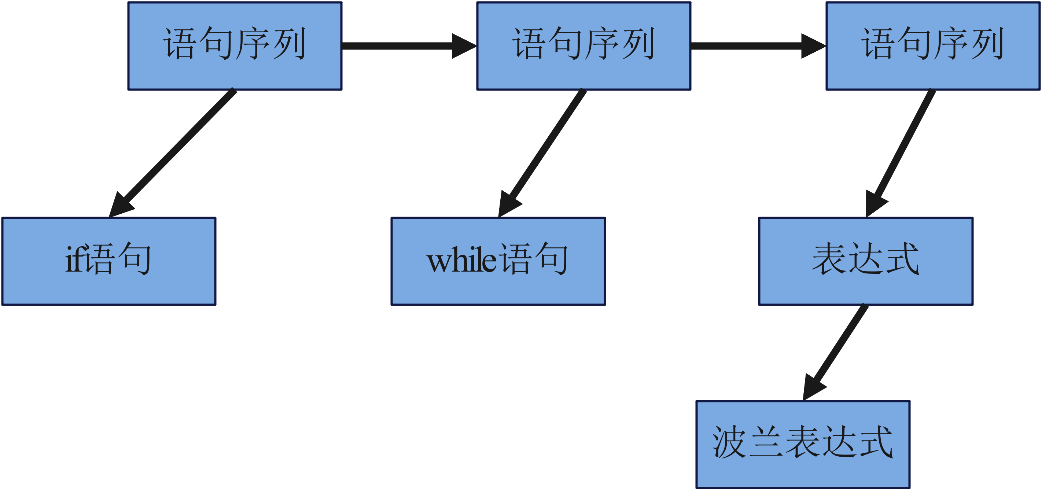


图3-7 语句序列

对于if、while等语句块，这里不一一列举，以if语句块为例：

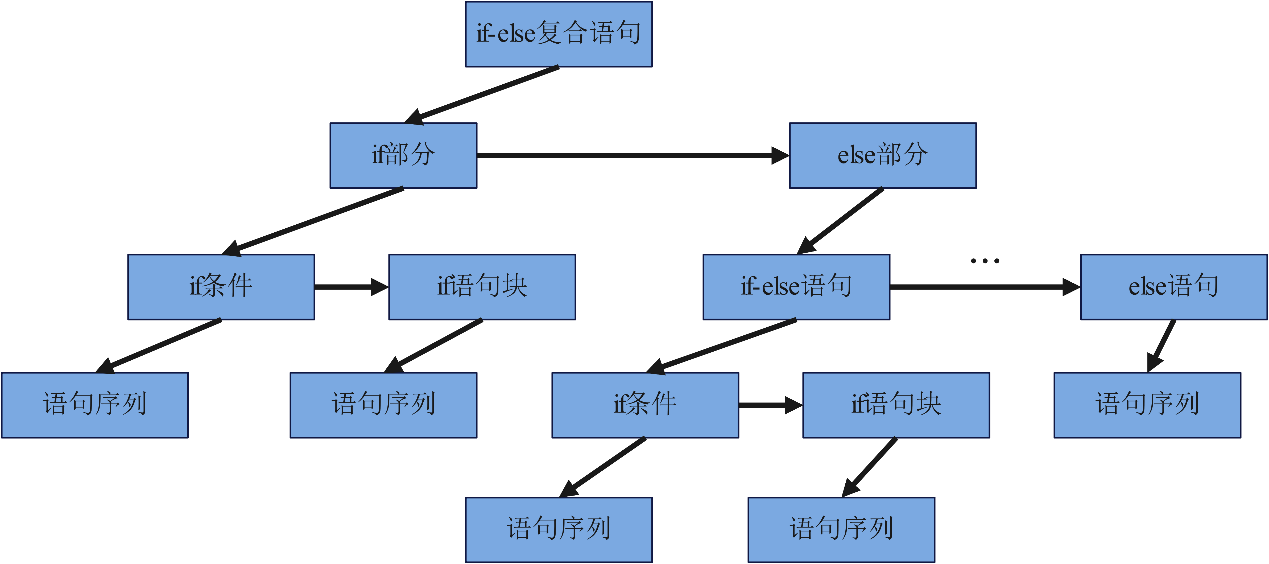


图3-8 if语句块

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：

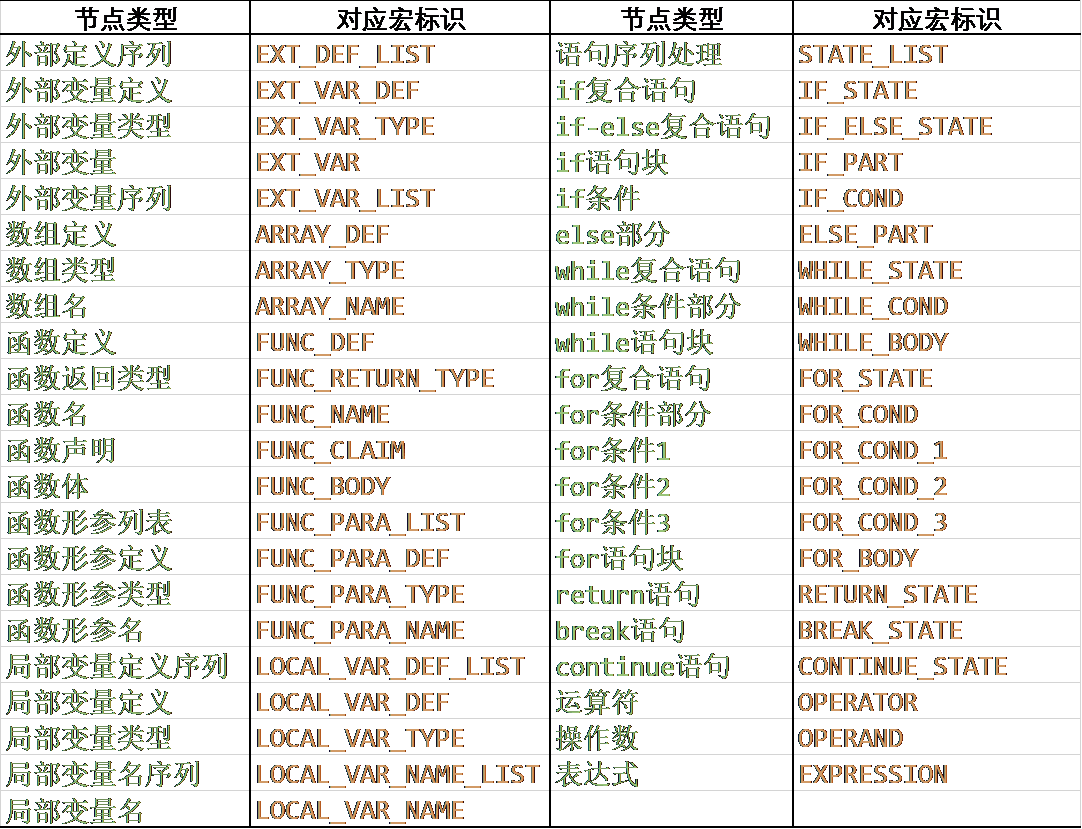


图3-9 节点类型

## 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。

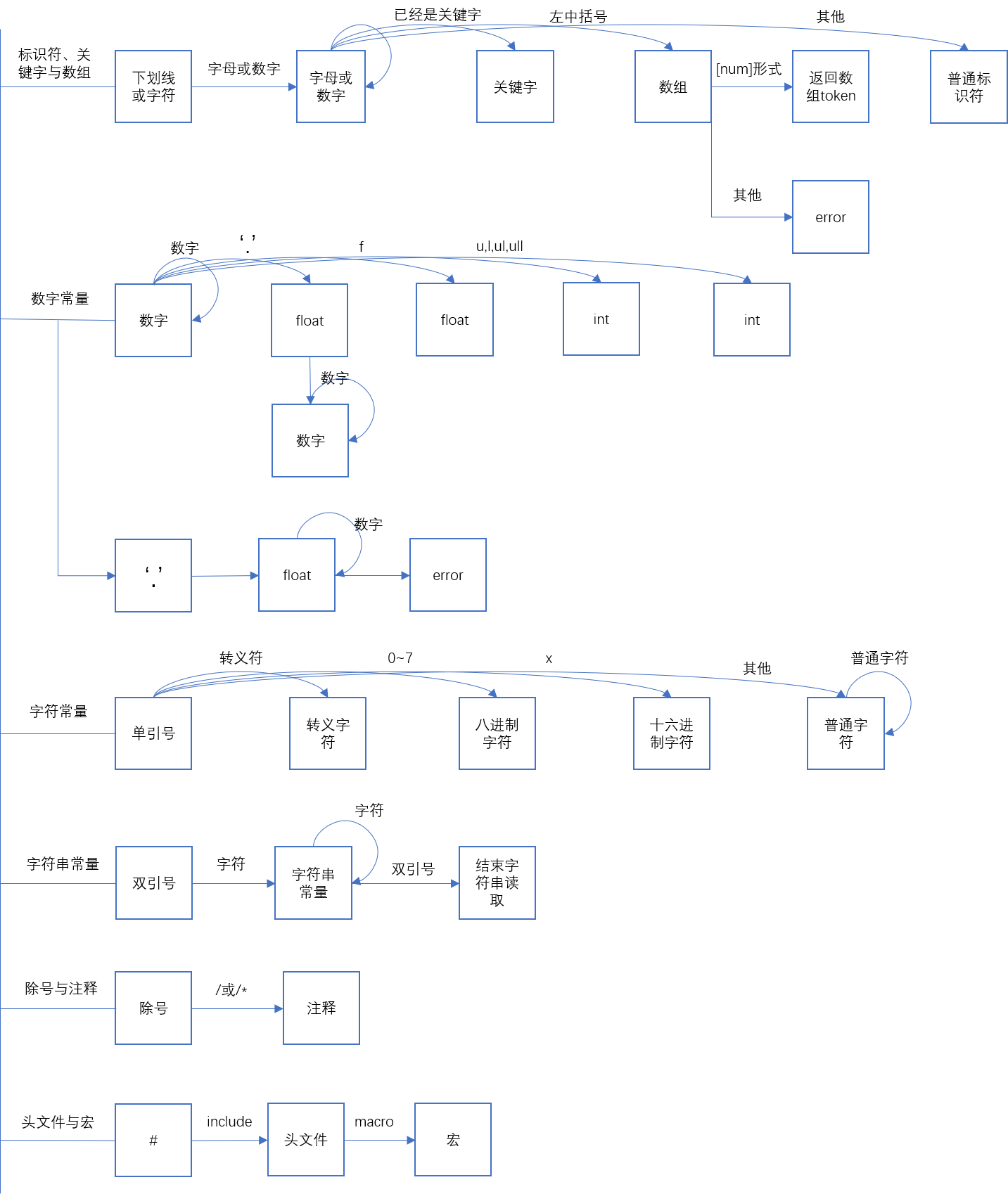


图3-10 完整的DNF状态图

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：

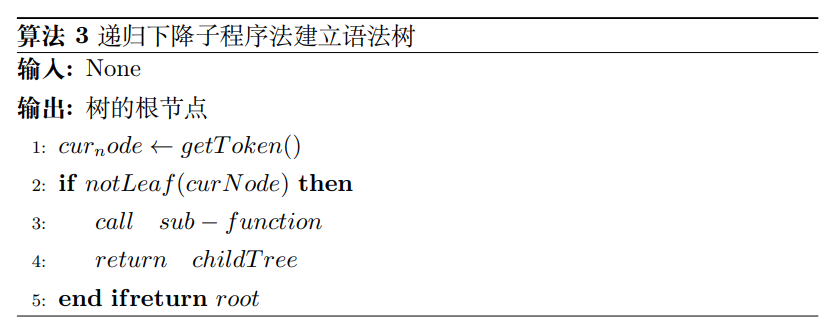


图3-11 递归下降子程序法伪代码

#### 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：

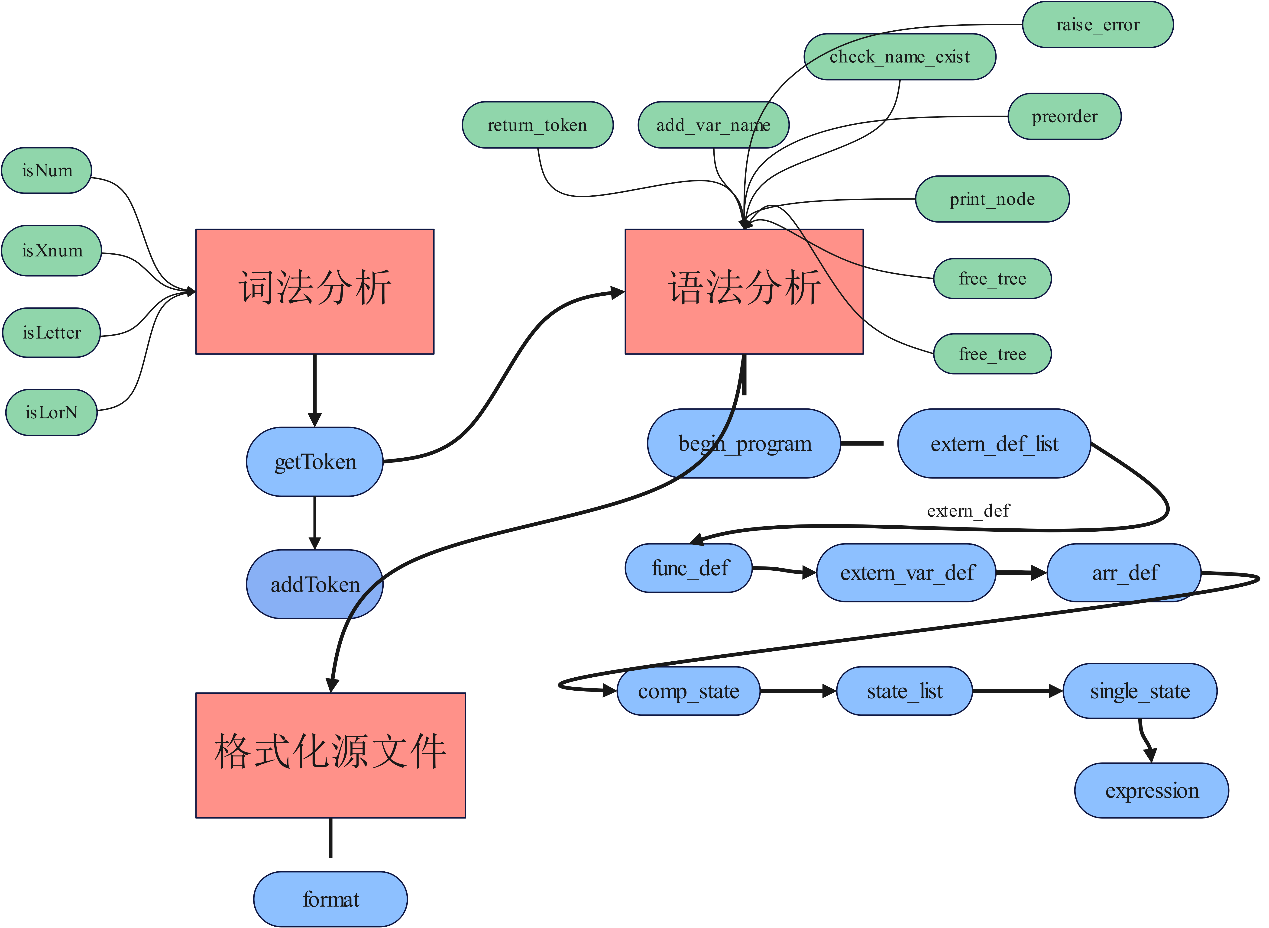


图3-9 程序整体架构

# 4系统实现与测试

## 4.1系统实现

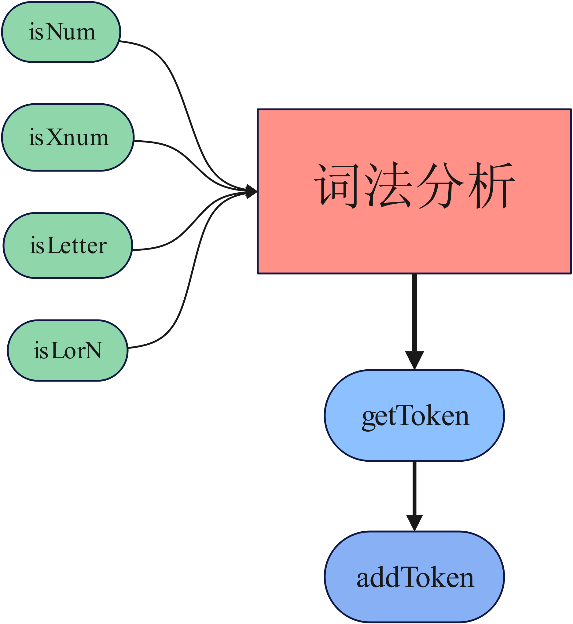
#### 运行环境

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。



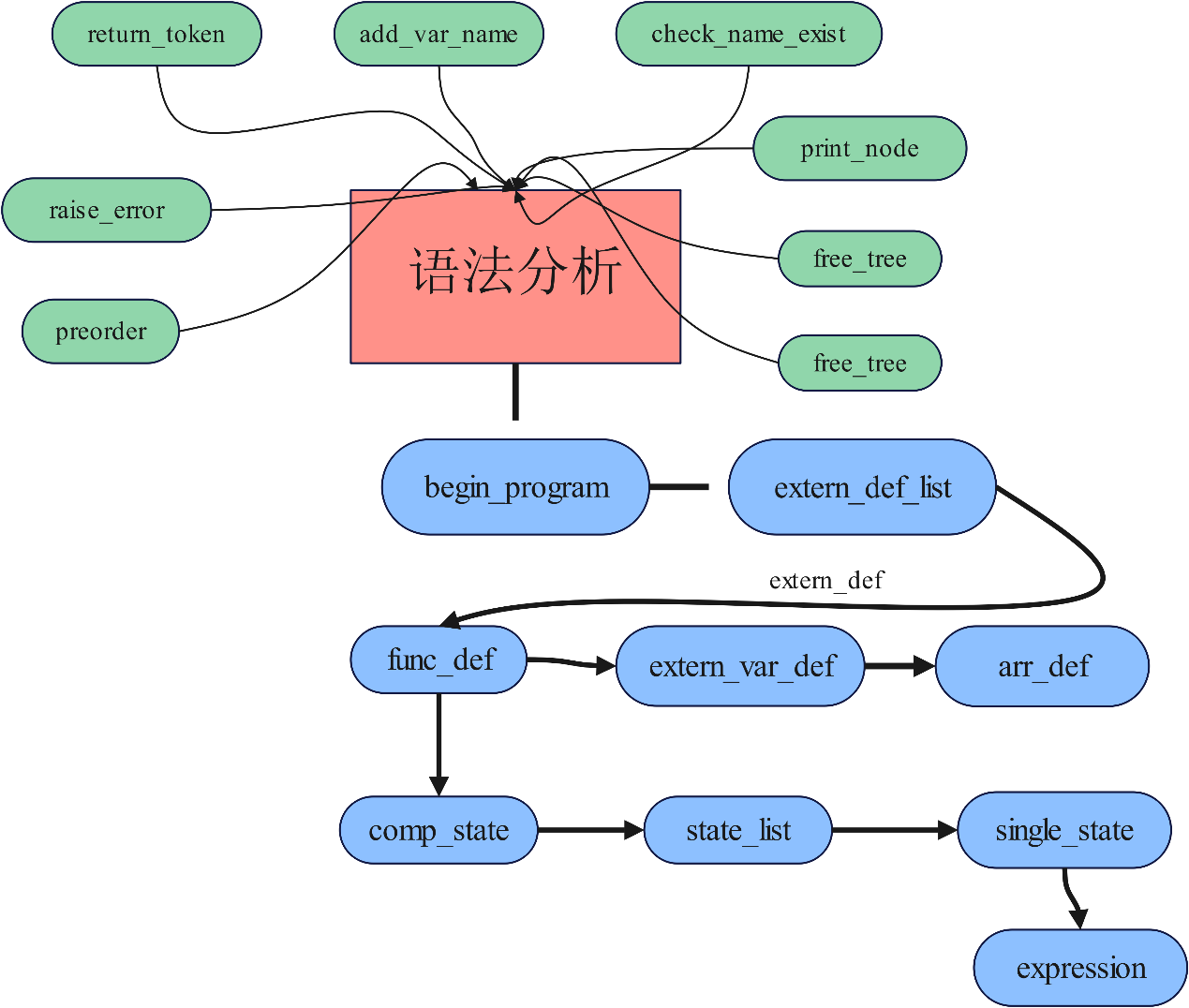
语法分析部分采用了递归下降子程序发，所用到的函数较多，下面先列出主要函数：

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)：

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，完整输出见附录）：

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

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:==
7. 操作数:a
8. 操作数:b
9. 表达式:
10. 运算符:=
11. 操作数:a
12. 运算符:+
13. 操作数:a
14. 操作数:b
15. else部分:
16. if-else复合语句:
17. if语句块:
18. if条件:
19. 表达式:
20. 运算符:<
21. 操作数:a
22. 操作数:b
23. 语句序列处理:
24. 表达式:
25. 运算符:=
26. 操作数:a
27. 运算符:-
28. 操作数:a
29. 操作数:b
30. 语句序列处理:
31. 表达式:
32. 运算符:=
33. 操作数:num[10]
34. 操作数:"string"
35. 语句序列处理:
36. if复合语句:
37. if语句块:
38. if条件:
39. 表达式:
40. 运算符:>
41. 操作数:a
42. 操作数:b
43. 语句序列处理:
44. 表达式:
45. 运算符:=
46. 操作数:a
47. 运算符:+
48. 操作数:a
49. 操作数:1
50. else部分:
51. 语句序列处理:
52. 表达式:
53. 运算符:=
54. 操作数:a
55. 运算符:/
56. 操作数:a
57. 操作数:b
58. 语句序列处理:
59. 表达式:
60. 运算符:=
61. 操作数:a
62. 运算符:+
63. 操作数:a
64. 操作数:1

运行结果符合设计目标。

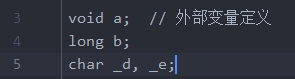
#### 报错功能测试

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）：

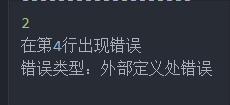
1. 出现void类型变量





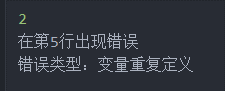
1. 变量命名不规范：出现美元符号



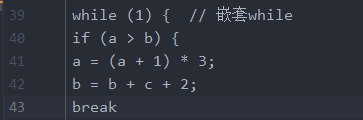


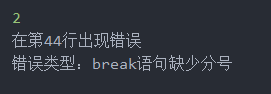
1. 变量重复定义





1. 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 ;
20. **if** ( a == b ) a = a + b ;
21. **else** **if** ( a < b ) {
22. a = a - b ;
23. num[10] = "string" ;
24. **if** ( a > b ) {
25. a = a + 1 ;
26. }
27. }
28. **else** {
29. a = a / b ;
30. a = a + 1 ;
31. }
32. **if** ( a > b ) a = a && b ;
33. **if** ( a > b ) {
34. a = ( a + 1 ) \* 3 ;
35. b = b + c + 2 ;
36. }
37. **while** ( 1 ) {
38. **if** ( a > b ) {
39. a = ( a + 1 ) \* 3 ;
40. b = b + c + 2 ;
41. **break** ;
42. }
43. **if** ( a < b ) {
44. **continue** ;
45. }
46. **while** ( a == 1 ) {
47. a = a || b ;
48. }
49. }
50. **for** ( i = 0 ; i < a ; i = i + 1 ) {
51. a = a + 1 ;
52. }
53. **for** ( ; ; i = i - 1 ) {
54. a = ( a + 1 ) \* 2 ;
55. }
56. **return** a + b ;
57. }
58. **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

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;
20. if (a == b)  // if
21. a = a + b;
22. else if (a < b) {  // 块if、嵌套if
23. a = a - b;
24. num[10] = "string";
25. if (a > b) {
26. a = a + 1;
27. }
28. }
29. else {
30. a = a / b;
31. a = a + 1;
32. }
33. if (a > b)
34. a = a && b;
35. if (a > b) {
36. a = (a + 1) \* 3;
37. b = b + c + 2;
38. }
39. while (1) {  // 嵌套while
40. if (a > b) {
41. a = (a + 1) \* 3;
42. b = b + c + 2;
43. break;
44. }
45. if (a < b) {
46. continue;
47. }
48. while (a == 1) {
49. a = a || b;
50. }
51. }
52. for (i = 0; i < a; i=i+1) {  // for循环
53. a = a + 1;
54. }
55. for (;; i=i-1) {  // 空条件for
56. a = (a + 1) \* 2;
57. }
58. return a + b;
59. }
60. /\*函数声明\*/
61. void def();
62. ~
63. //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、
64. //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
65. //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      =
43. int\_const       1
44. 分号      ;
45. ident       b
46. 赋值      =
47. int\_const       1
48. 分号      ;
49. ident       c
50. 赋值      =
51. float\_const     .14
52. 分号      ;
53. ident       h
54. 赋值      =
55. float\_const     3.13f
56. 分号      ;
57. ident       \_d
58. 赋值      =
59. char\_const      '\x22'
60. 分号      ;
61. ident       \_e
62. 赋值      =
63. char\_const      '\t'
64. 分号      ;
65. ident       a
66. 赋值      =
67. int\_const       1
68. 加法      +
69. ident       b
70. 分号      ;
71. ident       b
72. 赋值      =
73. int\_const       2
74. 加法      +
75. ident       b
76. 分号      ;
77. ident       b
78. 赋值      =
79. ident       c
80. 分号      ;
81. if      if
82. 左括号     (
83. ident       a
84. 是否等于        ==
85. ident       b
86. 右括号     )
87. 注释      // if
88. ident       a
89. 赋值      =
90. ident       a
91. 加法      +
92. ident       b
93. 分号      ;
94. else        else
95. if      if
96. 左括号     (
97. ident       a
98. 小于号     <
99. ident       b
100. 右括号     )
101. 左大括号        {
102. 注释      // 块if、嵌套if
103. ident       a
104. 赋值      =
105. ident       a
106. 减法      -
107. ident       b
108. 分号      ;
109. 数组      num[10]
110. 赋值      =
111. string\_const        "string"
112. 分号      ;
113. if      if
114. 左括号     (
115. ident       a
116. 大于号     >
117. ident       b
118. 右括号     )
119. 左大括号        {
120. ident       a
121. 赋值      =
122. ident       a
123. 加法      +
124. int\_const       1
125. 分号      ;
126. 右大括号        }
127. 右大括号        }
128. else        else
129. 左大括号        {
130. 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
263. 分号      ;
264. 右大括号        }
265. for     for
266. 左括号     (
267. 分号      ;
268. 分号      ;
269. ident       i
270. 赋值      =
271. ident       i
272. 减法      -
273. int\_const       1
274. 右括号     )
275. 左大括号        {
276. 注释      // 空条件for
277. ident       a
278. 赋值      =
279. 左括号     (
280. ident       a
281. 加法      +
282. int\_const       1
283. 右括号     )
284. 乘法      \*
285. int\_const       2
286. 分号      ;
287. 右大括号        }
288. return      return
289. ident       a
290. 加法      +
291. ident       b
292. 分号      ;
293. 右大括号        }
294. 注释      /\*函数声明\*/
295. void        void
296. ident       def
297. 左括号     (
298. 右括号     )
299. 分号      ;
300. 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
43. 函数形参定义:
44. 函数形参类型:int
45. 函数形参名:b
46. 函数体:
47. 局部变量定义序列:
48. 局部变量定义:
49. 局部变量类型:int
50. 局部变量名序列:
51. 局部变量名:test
52. 语句序列处理:
53. 表达式:
54. 运算符:=
55. 操作数:a
56. 操作数:1
57. 语句序列处理:
58. 表达式:
59. 运算符:=
60. 操作数:b
61. 操作数:1
62. 语句序列处理:
63. 表达式:
64. 运算符:=
65. 操作数:c
66. 操作数:.14
67. 语句序列处理:
68. 表达式:
69. 运算符:=
70. 操作数:h
71. 操作数:3.13f
72. 语句序列处理:
73. 表达式:
74. 运算符:=
75. 操作数:\_d
76. 操作数:'\x22'
77. 语句序列处理:
78. 表达式:
79. 运算符:=
80. 操作数:\_e
81. 操作数:'\t'
82. 语句序列处理:
83. 表达式:
84. 运算符:=
85. 操作数:a
86. 运算符:+
87. 操作数:1
88. 操作数:b
89. 语句序列处理:
90. 表达式:
91. 运算符:=
92. 操作数:b
93. 运算符:+
94. 操作数:2
95. 操作数:b
96. 语句序列处理:
97. 表达式:
98. 运算符:=
99. 操作数:b
100. 操作数:c
101. 语句序列处理:
102. if-else复合语句:
103. if语句块:
104. if条件:
105. 表达式:
106. 运算符:==
107. 操作数:a
108. 操作数:b
109. 表达式:
110. 运算符:=
111. 操作数:a
112. 运算符:+
113. 操作数:a
114. 操作数:b
115. else部分:
116. if-else复合语句:
117. if语句块:
118. if条件:
119. 表达式:
120. 运算符:<
121. 操作数:a
122. 操作数:b
123. 语句序列处理:
124. 表达式:
125. 运算符:=
126. 操作数:a
127. 运算符:-
128. 操作数:a
129. 操作数:b
130. 语句序列处理:
131. 表达式:
132. 运算符:=
133. 操作数:num[10]
134. 操作数:"string"
135. 语句序列处理:
136. if复合语句:
137. if语句块:
138. if条件:
139. 表达式:
140. 运算符:>
141. 操作数:a
142. 操作数:b
143. 语句序列处理:
144. 表达式:
145. 运算符:=
146. 操作数:a
147. 运算符:+
148. 操作数:a
149. 操作数:1
150. else部分:
151. 语句序列处理:
152. 表达式:
153. 运算符:=
154. 操作数:a
155. 运算符:/
156. 操作数:a
157. 操作数:b
158. 语句序列处理:
159. 表达式:
160. 运算符:=
161. 操作数:a
162. 运算符:+
163. 操作数:a
164. 操作数:1
165. 语句序列处理:
166. if复合语句:
167. if语句块:
168. if条件:
169. 表达式:
170. 运算符:>
171. 操作数:a
172. 操作数:b
173. 表达式:
174. 运算符:=
175. 操作数:a
176. 运算符:&&
177. 操作数:a
178. 操作数:b
179. 语句序列处理:
180. if复合语句:
181. if语句块:
182. if条件:
183. 表达式:
184. 运算符:>
185. 操作数:a
186. 操作数:b
187. 语句序列处理:
188. 表达式:
189. 运算符:=
190. 操作数:a
191. 运算符:\*
192. 运算符:+
193. 操作数:a
194. 操作数:1
195. 操作数:3
196. 语句序列处理:
197. 表达式:
198. 运算符:=
199. 操作数:b
200. 运算符:+
201. 运算符:+
202. 操作数:b
203. 操作数:c
204. 操作数:2
205. 语句序列处理:
206. while复合语句:
207. while条件部分:
208. 表达式:
209. 操作数:1
210. while语句块:
211. 语句序列处理:
212. if复合语句:
213. if语句块:
214. if条件:
215. 表达式:
216. 运算符:>
217. 操作数:a
218. 操作数:b
219. 语句序列处理:
220. 表达式:
221. 运算符:=
222. 操作数:a
223. 运算符:\*
224. 运算符:+
225. 操作数:a
226. 操作数:1
227. 操作数:3
228. 语句序列处理:
229. 表达式:
230. 运算符:=
231. 操作数:b
232. 运算符:+
233. 运算符:+
234. 操作数:b
235. 操作数:c
236. 操作数:2
237. 语句序列处理:
238. break语句:
239. 语句序列处理:
240. if复合语句:
241. if语句块:
242. if条件:
243. 表达式:
244. 运算符:<
245. 操作数:a
246. 操作数:b
247. 语句序列处理:
248. continue语句:
249. 语句序列处理:
250. while复合语句:
251. while条件部分:
252. 表达式:
253. 运算符:==
254. 操作数:a
255. 操作数:1
256. while语句块:
257. 语句序列处理:
258. 表达式:
259. 运算符:=
260. 操作数:a
261. 运算符:||
262. 操作数:a
263. 操作数:b
264. 语句序列处理:
265. for复合语句:
266. for条件部分:
267. for条件1:
268. 表达式:
269. 运算符:=
270. 操作数:i
271. 操作数:0
272. for条件2:
273. 表达式:
274. 运算符:<
275. 操作数:i
276. 操作数:a
277. for条件3:
278. 表达式:
279. 运算符:=
280. 操作数:i
281. 运算符:+
282. 操作数:i
283. 操作数:1
284. 语句序列处理:
285. for复合语句:
286. for条件部分:
287. for条件1:无
288. for条件2:无
289. for条件3:
290. 表达式:
291. 运算符:=
292. 操作数:i
293. 运算符:-
294. 操作数:i
295. 操作数:1
296. 语句序列处理:
297. return语句:
298. 表达式:
299. 运算符:+
300. 操作数:a
301. 操作数:b
302. 外部定义序列:
303. 函数声明:
304. 函数返回类型:void
305. 函数名:def
306. 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

1. #include <iostream>
2. #include <string>
3. #include "syntax\_analyse.h"
4. #include "get\_token.h"
5. #include "format\_operation.h"
6. **using** **namespace** std;
8. **extern** **int** line\_cnt;
9. **extern** string token\_text;
10. **FILE** \*fp;
12. **int** main() {
13. system("chcp 65001");
14. **char** filename[20];
15. cout << "enter the filename:" << endl;
16. cin >> filename;
17. **int** op;
18. **while** (1){
19. fp = fopen(filename,"r");
20. **if** (fp == nullptr){
21. cout << "文件打开错误" << endl;
22. **return** 0;
23. }
24. cout << "-------------------" << endl;
25. cout << "  输入序号选择功能" << endl;
26. cout << "    1. 词法分析" << endl;
27. cout << "    2. 语法分析" << endl;
28. cout << "    3. 格式化" << endl;
29. cout << "    0. 退出" << endl;
30. cout << "-------------------" << endl;
31. cin >> op;
32. **switch** (op) {
33. **case** 1:
34. **int** token;
35. cout << "类型\t\t" << "符号" << endl;
36. **while** ((token = getToken(fp)) != -1){
37. **if** (token == ERROR\_TOKEN){
38. cout << "在第" << line\_cnt << "行出现错误！" << endl;
39. cout << "错误信息为：" << token\_text << "over!" << endl;
40. **break**;
41. }
42. **else**{
43. cout << token\_kind\_arr[token] << "\t\t" << token\_text << endl;
44. }
45. }
46. cout << "词法分析结束，按任意键继续" << endl;
47. **break**;
48. **case** 2:
49. line\_cnt = 1;
50. syntax\_analyse();
51. cout << "语法分析结束，按任意键继续" << endl;
52. **break**;
53. **case** 3:
54. //fp = fopen("../test.txt","r");
55. format();
56. cout << "格式化结束，请按任意键继续" << endl;
57. //fclose(fp);
58. **break**;
59. **case** 0:
60. //system("clear");
61. cout << "感谢使用" << endl;
62. **return** 0;
63. **default**:
64. cout << "输入有误，按任意键重新输入" << endl;
65. }
66. fclose(fp);
67. }
68. **return** 0;
69. }

## get\_token.h

1. //
2. // Created by Mr.K on 2022-08-02.
3. //
5. #ifndef GET\_TOKEN\_H
6. #define GET\_TOKEN\_H
8. #include <string>
10. **using** **namespace** std;
12. /\*词法分析编码列表\*/
13. **typedef** **enum** token\_kind{
14. ERROR\_TOKEN = 1,
15. IDENT,  //标识符
16. INT\_CONST,  //整形常量
17. FLOAT\_CONST,
18. CHAR\_CONST,
19. STRING\_CONST,
20. KEYWORD,
21. **INT**,  // 8
22. **FLOAT**,
23. **CHAR**,
24. **LONG**,
25. **SHORT**,
26. DOUBLE,
27. **VOID**,  // 14
28. ELSE,
29. DO,
30. WHILE,
31. FOR,
32. IF,
33. BREAK,
34. SWITCH,
35. CASE,
36. TYPEDEF,
37. RETURN,
38. CONTINUE,
39. STRUCT,  //26
40. LB,  //左大括号
41. RB,  //右大括号
42. LM,  //左中括号
43. RM,  //右中括号
44. SEMI,  //分号31
45. COMMA,  //逗号
46. /\*EQ到MINUSMINUS为运算符，必须连在一起\*/
47. EQ,  //'=='
48. NEQ,  //‘!=’
49. ASSIGN,  //'='35
50. LP,  //左括号
51. 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;
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"};

79. // 判断是否是数字
80. **int** isNum(**char** c);
82. // 判断是否是十六进制数
83. **int** isXnum(**char** c);
85. // 判断是否是字母
86. **int** isLetter(**char** c);
88. // 判断是否是字母或数字
89. **int** isLorN(**char** c);
91. // 向字符串中添加字符
92. **int** add2token(string s, **char** c);
94. // 读取输入并返回当前读取token的类型
95. **int** getToken(**FILE**\* fp);
97. #endif

## get\_token.cpp

1. //
2. // Created by Mr.K on 2022-08-02.
3. //
5. #include "get\_token.h"
7. #include <cstdio>

10. string token\_text = "";  // 存放读取的一个词
11. **char** c;  // 存放从文件流中读取的字符
12. **int** line\_cnt = 1;  // 读取的代码行数



17. **int** isNum(**char** c)
18. {
19. **if** (c >= 48 && c <= 57) {
20. **return** 1;
21. }
22. **return** 0;
23. }
25. **int** isXnum(**char** c)
26. {
27. **if** (isNum(c) || c >= 'a' && c <= 'f' || c >= 'A' && c <= 'F') {
28. **return** 1;
29. }
30. **return** 0;
31. }
33. **int** isLetter(**char** c)
34. {
35. **if** (c >= 'a' && c <= 'z' || c >= 'A' && c <= 'Z') {
36. **return** 1;
37. }
38. **return** 0;
39. }
41. **int** isLorN(**char** c)
42. {
43. **if** (isNum(c) || isLetter(c)) {
44. **return** 1;
45. }
46. **return** 0;
47. }
49. **int** add2token(string s, **char** c){
50. token\_text += c;
51. **return** 1;
52. }
54. **int** getToken(**FILE**\* fp){
55. token\_text = "";
56. **do**{  // 过滤缩进和空行
57. c = fgetc(fp);
58. **if** (c == '\n') line\_cnt++;
59. } **while** (c == ' ' || c == '\n');
60. /\*标识符、关键字、数组的识别\*/
61. **if** (c == '\_' || isLetter(c)){
62. add2token(token\_text, c);
63. **while** (isLorN((c = getc(fp)))){
64. token\_text += c;
65. }
66. /\*判断是否是关键字\*/
67. **if** (token\_text == "int") {ungetc(c, fp); **return** **INT**;}
68. **if** (token\_text == "double") {ungetc(c, fp);**return** DOUBLE;}
69. **if** (token\_text == "char") {ungetc(c, fp);**return** **CHAR**;}
70. **if** (token\_text == "short") {ungetc(c, fp);**return** **SHORT**;}
71. **if** (token\_text == "long") {ungetc(c, fp);**return** **LONG**;}
72. **if** (token\_text == "float") {ungetc(c, fp);**return** **FLOAT**;}
73. **if** (token\_text == "if") {ungetc(c, fp);**return** IF;}
74. **if** (token\_text == "else") {ungetc(c, fp);**return** ELSE;}
75. **if** (token\_text == "do") {ungetc(c, fp);**return** DO;}
76. **if** (token\_text == "while") {ungetc(c, fp);**return** WHILE;}
77. **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);
205. **return** CHAR\_CONST;
206. }
207. }
208. **else**{
209. **if** (c != '\'') **return** ERROR\_TOKEN;
210. **else**{
211. add2token(token\_text, c);
212. **return** CHAR\_CONST;
213. }
214. }
215. }
216. **else**{
217. **return** ERROR\_TOKEN;
218. }
219. }
220. **else** **if** (c >= '0' && c <= '7'){
221. add2token(token\_text, c);
222. c = fgetc(fp);
223. **if** (c >= '0' && c <= '7'){
224. add2token(token\_text, c);
225. c = fgetc(fp);
226. **if** (c >= '0' && c <= '7'){
227. add2token(token\_text, c);
228. **if** ((c = fgetc(fp)) != '\'') **return** ERROR\_TOKEN;
229. **else**{
230. add2token(token\_text, c);
231. **return** CHAR\_CONST;
232. }
233. }
234. **else**{
235. **if** (c != '\'') **return** ERROR\_TOKEN;
236. **else**{
237. add2token(token\_text, c);
238. **return** CHAR\_CONST;
239. }
240. }
241. }
242. **else**{
243. **if** (c != '\'') **return** ERROR\_TOKEN;
244. **else**{
245. add2token(token\_text, c);
246. **return** CHAR\_CONST;
247. }
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. //
5. #ifndef SYNTAX\_ANALYSE\_H
6. #define SYNTAX\_ANALYSE\_H
8. #include <string>
10. **using** **namespace** std;
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. };
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. };
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. };

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,
127. }node\_kind;



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,
175. }error\_kind;



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);
243. #endif

## syntax\_analyse.cpp

1. /
2. // Created by Mr.K on 2022-08-07.
3. //
5. #include "syntax\_analyse.h"
6. #include "get\_token.h"
7. #include <cstdio>
8. #include <iostream>
9. #include <stack>
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;
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"};
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"};

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. }
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. }
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. }
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. }
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. }
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. }
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. }
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. }
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. }
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. }
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. }
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. }
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. }
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. }
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. }
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. }
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. }
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. }
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. }
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. }
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. }
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. }
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\_node);
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. }
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. }
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. }
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. }
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. }
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. }
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. }
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. }
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\_type == 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\_ERR);  // 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. }
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**;
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. //
5. #ifndef MY\_TINY\_COMPILER\_FORMAT\_OPERATION\_H
6. #define MY\_TINY\_COMPILER\_FORMAT\_OPERATION\_H
8. #include <vector>
9. #include <string>


13. // 读取每个部分
14. **int** read\_part(std::vector<std::string> &sep\_part);
16. **void** print\_part(std::vector<std::string> sep\_part);
18. // 格式化主函数
19. **void** format();


23. #endif //MY\_TINY\_COMPILER\_FORMAT\_OPERATION\_H

## format\_operation.cpp

1. //
2. // Created by Mr.K on 2022-08-28.
3. //
5. #include "format\_operation.h"
6. #include "get\_token.h"
7. #include <iostream>
8. #include <fstream>
10. **extern** **FILE**\* fp;
11. **extern** string token\_text;
12. **int** word\_type0;
13. **int** tabs;
14. ofstream fout;
16. // 存储每个部分的token
17. std::vector<std::string> sep\_part;
19. **int** read\_part(std::vector<std::string> &sep\_part){
20. word\_type0 = getToken(fp);
21. **while** (word\_type0 == ANNO){
22. word\_type0 = getToken(fp);
23. }
24. **if** (word\_type0 == -1){  // 读到结尾
25. **return** 0;
26. }
27. **if** (word\_type0 == INCLUDE || word\_type0 == MACRO){  // 根据头文件和宏结束表达
28. sep\_part.push\_back(token\_text);
29. **return** 1;
30. }
31. **if** (word\_type0 == FOR){
32. **while** (word\_type0 != RP){
33. sep\_part.push\_back(token\_text);
34. word\_type0 = getToken(fp);
35. }
36. sep\_part.push\_back(token\_text);
37. **return** read\_part(sep\_part);
38. }
39. **if** (word\_type0 == SEMI){  // 根据分号结束一句表达
40. sep\_part.push\_back(token\_text);
41. **return** 1;
42. }
43. **if** (word\_type0 == LB){  // 根据大括号结束一句表达
44. sep\_part.push\_back(token\_text);
45. //tabs++;
46. **return** 1;
47. }
48. **if** (word\_type0 == RB){  // 根据右括号结束一句表达
49. sep\_part.push\_back(token\_text);
50. //tabs--;
51. **return** 1;
52. }
53. // 普通token
54. sep\_part.push\_back(token\_text);
55. **return** read\_part(sep\_part);
56. }
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. }
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)
4. set(CMAKE\_CXX\_STANDARD 14)
6. set(CMAKE\_EXE\_LINKER\_FLAGS -**static**)
8. add\_executable(my\_tiny\_compiler main.cpp get\_token.cpp get\_token.h syntax\_analyse.cpp syntax\_analyse.h format\_operation.cpp format\_operation.h)