编号：

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 实习 | 一 | 二 | 三 | 四 | 五 | 六 | 七 | 八 | 九 | 十 | 总评 | 教师签名 |
| 成绩 |  |  |  |  |  |  |  |  |  |  |  |  |

武汉大学计算机学院

**《编译原理》课程**

词法分析

实习报告

编 号： B401

实习题目： 构造词法分析器

专业（班）： 信息安全

学生学号： 2021302121464

学生姓名： 易乐

任课教师： 杜　卓　敏

２０２４ 年 5 月 18 日

**目 录**

[**第一部分 语言形式化描述（正规式/文法） ……………………**](#_第一部分__语言形式化描述)

[**第二部分 单词编码表 ………………………**](#_第二部分__单词编码表)

[**第三部分 状态转换图 ………………………**](#_第三部分__状态转换图)

[**第四部分 词法分析算法 ………………………**](#_第四部分__词法分析算法)

[**第五部分 测试计划（报告） ………………………**](#_第五部分__测试计划（报告）)

# 第一部分 语言形式化描述

1. 基本单词符号的定义
   1. 标识符

标识符包括用户自定义标识符与关键字。

* + 1. 所有关键字如下表所示：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| void | int | char | if | else |
| while | for | main | break | continue |
| return | NULL |  |  |  |

* + 1. 用户自定义标识符，以字母/下划线开头，可以由数字/字母/下划线组成。

标识符的文法定义如下：

E ::= nA

A ::= mA | ε

n ::= letter | \_

m ::= letter | digit | \_

其中，E为文法的开始符号，letter、digit、\_是终结符，分别表示所有的字母、所有的数字、下划线。

* 1. 常量

常量包括无符号整型常数、字符常量与字符串字面量。

常量的文法定义如下：

E ::= nA | 'B | "C

n ::= digit

A ::= nA | ε

B ::= any'

C ::= anyC | "

其中，E为文法的开始符号，digit、any、'、"是终结符，分别表示所有的数字、任一字符、单引号、双引号。

* 1. 符号包括界限符与运算符
     1. 所有界限符如下表所示

|  |  |  |  |
| --- | --- | --- | --- |
| ; | , | ( | ) |
| [ | ] | { | } |

* + 1. 所有运算符如下表所示

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| + | - | \* | / | % | = |
| > | < | ! | >= | <= | == |
| != | += | -= | /= | \*= | %= |

1. 语言的基本文法

<语句> ::= <赋值语句> | <条件语句> | <循环语句>

<赋值语句> ::= <标识符> = <表达式>

<表达式> ::= <项> | <项> <双目运算符> <表达式> | <单目运算符><表达式>

<项> ::= <标识符> | <常量>

<条件语句> ::= if ( <表达式> ) <语句> [else <语句>]

<循环语句> ::= while( <表达式> ) <语句> | for ( <表达式>;<表达式>;<表达式> ) <语句>

# 第二部分 单词编码表

1. 单词表如下表所示：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 类别号 | 名称 | 类别号 | 名称 | 类别号 | 名称 |
| 0 | void | **14** | 字符串 | **28** | % |
| 1 | int | **15** | 字符 | **29** | = |
| 2 | char | **16** | ; | **30** | < |
| 3 | if | **17** | , | **31** | > |
| 4 | else | **18** | ( | **32** | ! |
| 5 | while | **19** | ) | **33** | += |
| 6 | for | **20** | [ | **34** | -= |
| 7 | main | **21** | ] | **35** | \*= |
| 8 | break | **22** | { | **36** | /= |
| 9 | continue | **23** | } | **37** | %= |
| 10 | return | **24** | + | **38** | == |
| 11 | NULL | **25** | - | **39** | <= |
| 12 | 标识符 | **26** | \* | **40** | >= |
| 13 | 常数 | **27** | / | **41** | != |

1. 标识符表（不同的源程序的表中数据不同）

|  |  |
| --- | --- |
| 序号 | 自身值 |
|  |  |

1. 常数表（不同的源程序的表中数据不同）

|  |  |
| --- | --- |
| 序号 | 自身值 |
|  |  |

1. 字符常量表（不同的源程序的表中数据不同）

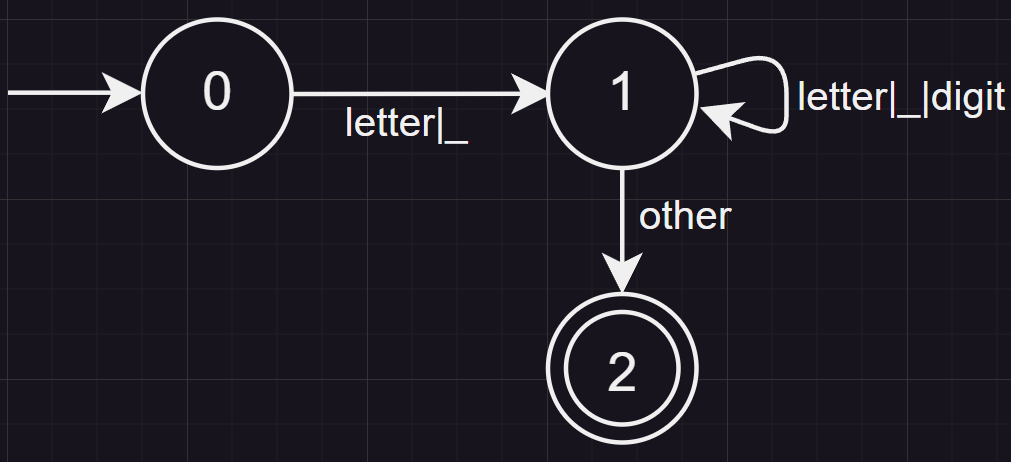
|  |  |
| --- | --- |
| 序号 | 自身值 |
|  |  |

1. 字符串字面量表（不同的源程序的表中数据不同）

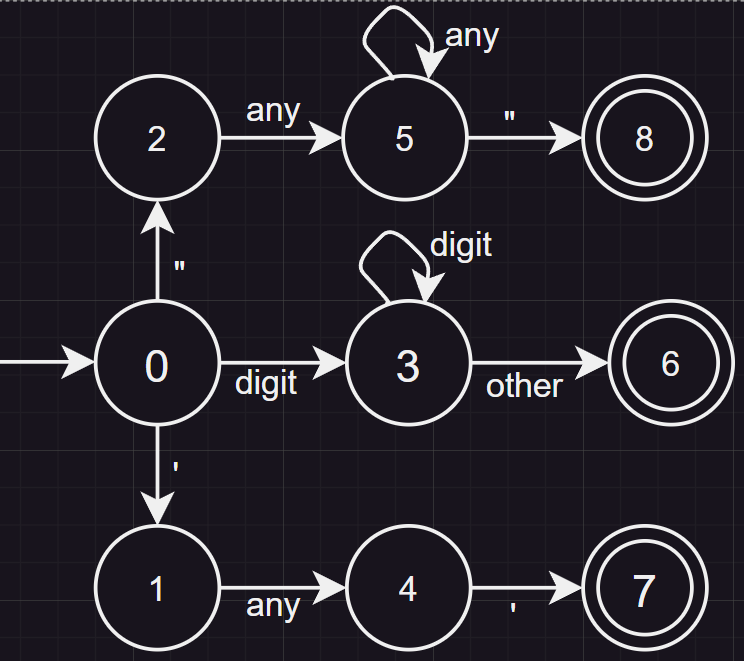
|  |  |
| --- | --- |
| 序号 | 自身值 |
|  |  |

# 第三部分 状态转换图

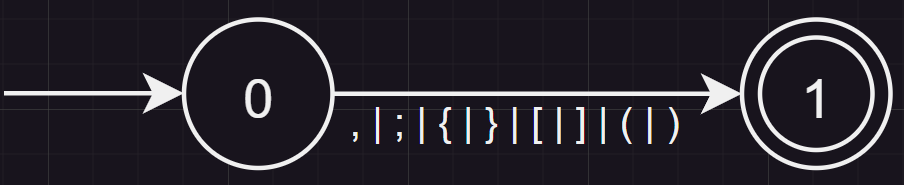
1. 各类单词的状态转换图
   1. 标识符DFA



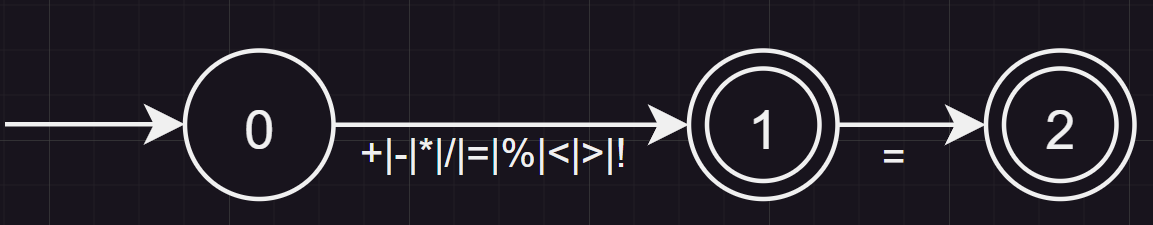
* 1. 常量DFA



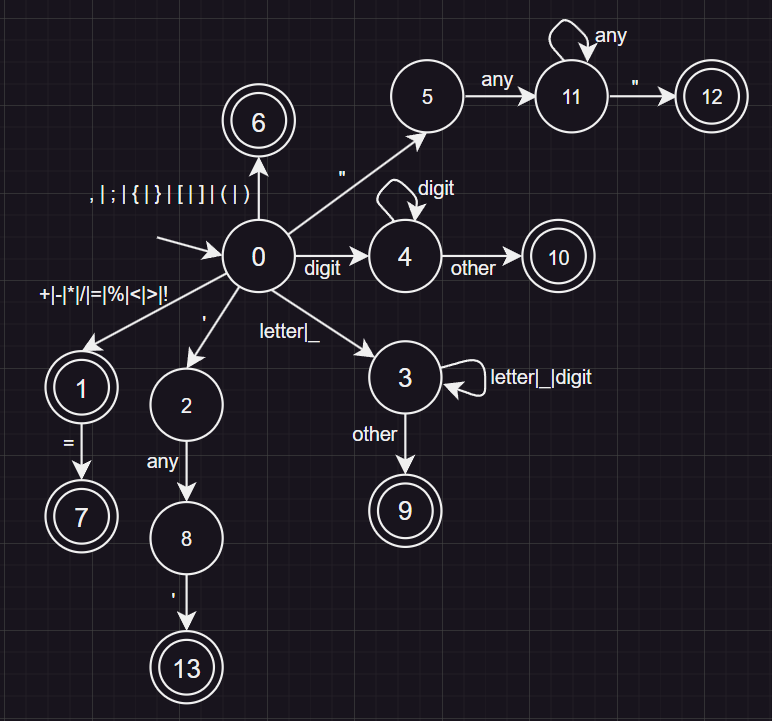
* 1. 符号DFA
     1. 界限符DFA



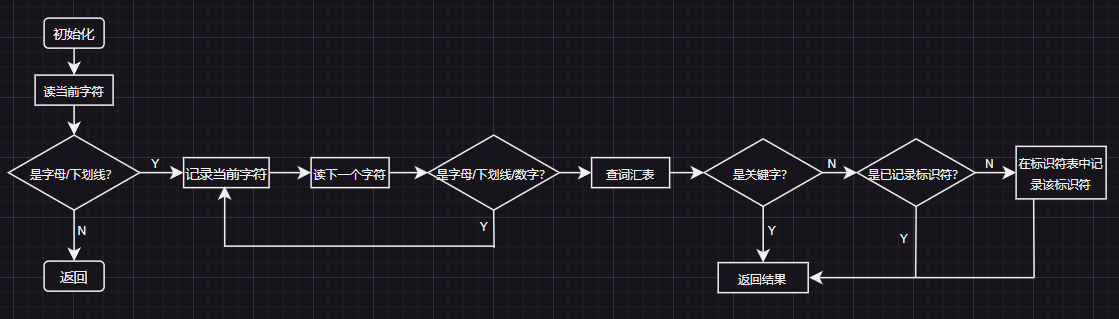
* + 1. 运算符DFA



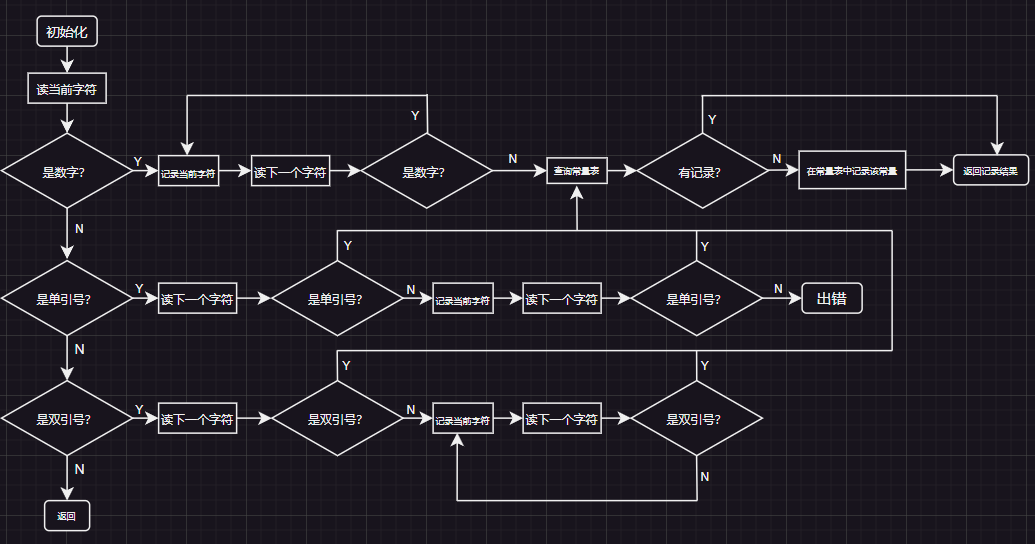
1. 合并后的总DFA



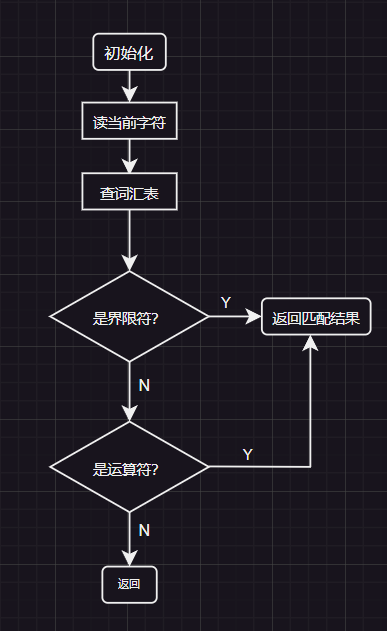
1. 各类单词的扫描程序框图
   1. 标识符



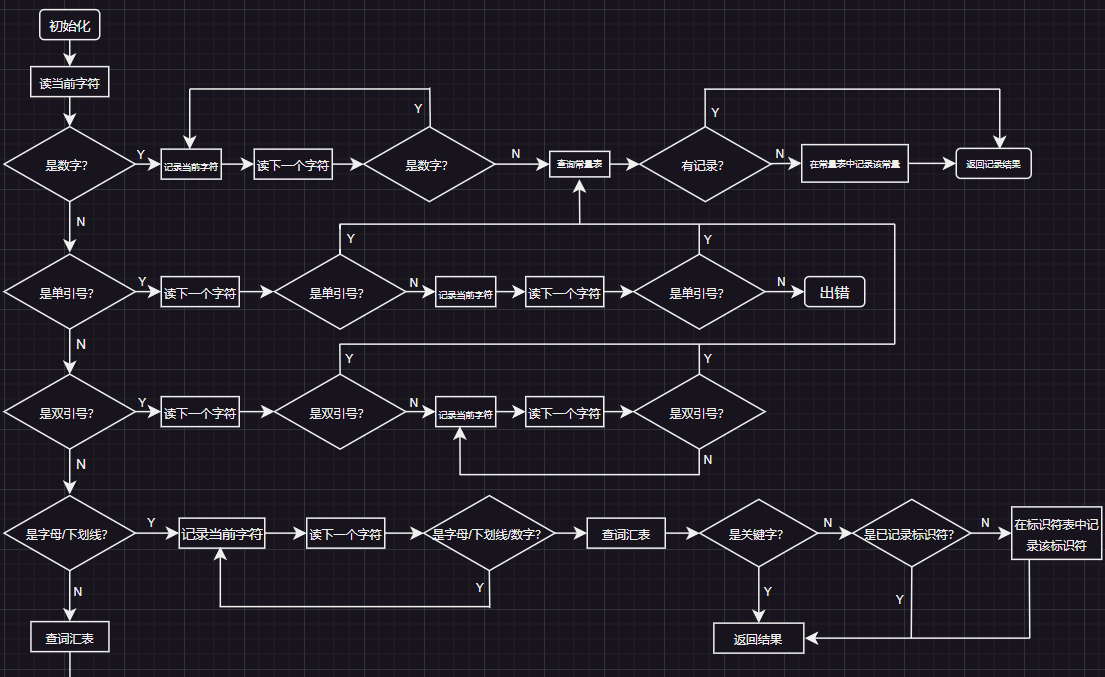
* 1. 常量

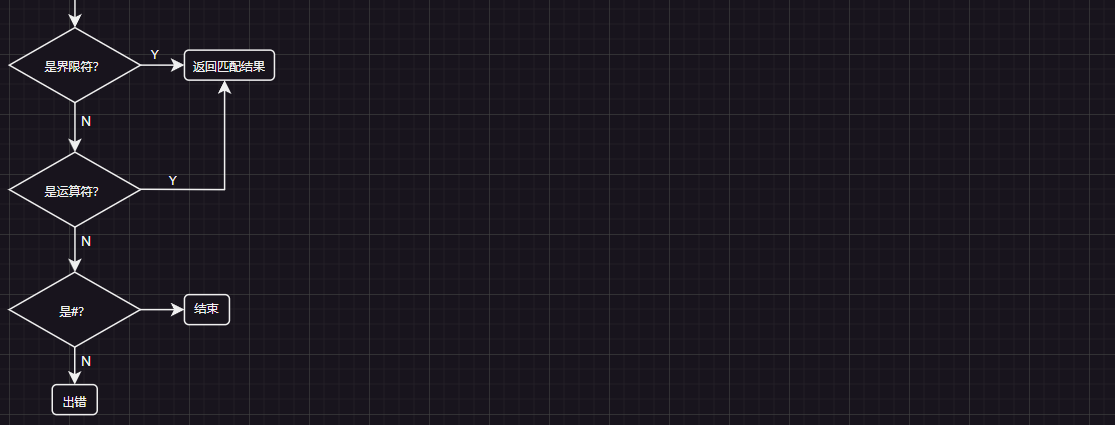


* 1. 符号



1. 合并后的总扫描程序框图





# 第四部分 词法分析算法

1. 设计预处理程序
   1. 要求：预处理程序应该删除源程序中的注释、多余的空格、回车符、换行符、制表符等非必要信息。
   2. 设计思路：
      1. 明确目标：要移除注释和多余的空格类字符，同时应该保证处理后不改变程序的行数，也不会修改其他字符。
      2. 分析目标：
         1. 需要识别注释(多行注释和单行注释)、多余空格类字符进行删除；
         2. 需要逐行处理源程序，便于识别换行符正确换行。
      3. 细节注意：
         1. 注意不能修改字符串字面量中的内容；
         2. 注意多行注释跨行的处理以及前后空格类字符的处理；
         3. 注意行首的空格类字符的删除。
   3. 具体代码实现如下：
2. *// 预处理函数*
3. void preprocessing(FILE \*fp, FILE \*out\_fp) {
4. char buffer[BUFFER\_SIZE+1];
5. bool in\_comment = false; *// 在多行注释中*
6. bool change = false; *// 多行注释有换行*
7. while (fgets(buffer, BUFFER\_SIZE, fp) != NULL) {
8. int i = 0;
9. char lastc = '\0';
10. bool in\_string = false; *// 在字符串中*
11. bool is\_head = true; *// 一行开头*
12. bool is\_space = false; *// 有空格*
13. while (buffer[i]) {
14. *// 不在字符串中，不在多行注释中*
15. if (!in\_string && !in\_comment) {
16. *// 接下来是字符串*
17. if (buffer[i] == '\"') {
18. if (is\_space) {
19. fprintf(out\_fp, " ");
20. is\_space = false;
21. }
22. in\_string = true;
23. lastc = buffer[i++];
24. fprintf(out\_fp, "%c", lastc);
25. }
26. *// 接下来是多行注释*
27. else if (buffer[i] == '/' && buffer[i+1] == '\*') {
28. in\_comment = true;
29. i = i + 2;
30. }
31. *// 正常时*
32. else {
33. *// 判断是否为一行的开头*
34. if (is\_head) {
35. *// 找到第一个不为空格的字符*
36. while (buffer[i] && isspace(buffer[i]))
37. i++;
38. *// 未找到，说明一行全为空格，直接换行*
39. if (!buffer[i]) {
40. fprintf(out\_fp, "\n");
41. break;
42. }
43. *// 有第一个不为空格的符号，则会重新循环*
44. else
45. is\_head = false;
46. continue;
47. }
48. *// 如果当前符号是空格（非换行）*
49. if (isspace(buffer[i]) && buffer[i] != '\n') {
50. is\_space = true;
51. *// 找到后面第一个非空格或到结尾*
52. while (buffer[i] && isspace(buffer[i]))
53. i++;
54. *// 后面未出现非空格则直接换行*
55. if (!buffer[i]) {
56. fprintf(out\_fp, "\n");
57. break;
58. }
59. else *// 后面出现非空格则重新判断*
60. continue;
61. }
62. *// 是换行符，能到这说明前面不可能是空格*
63. if (buffer[i] == '\n') {
64. fprintf(out\_fp, "\n");
65. *// 结束该行的处理*
66. break;
67. }
68. *//是单行注释，无论前面是否是空格，都要直接换行*
69. if (buffer[i] == '/' && buffer[i+1] == '/') {
70. fprintf(out\_fp, "\n");
71. *// 结束该行的处理*
72. break;
73. }
74. if (is\_space) {
75. fprintf(out\_fp, " ");
76. is\_space = false;
77. }
78. lastc = buffer[i++];
79. fprintf(out\_fp, "%c", lastc);
80. }
81. }
82. *// 在字符串中*
83. else if (in\_string) {
84. lastc = buffer[i++];
85. fprintf(out\_fp, "%c", lastc);
86. if (lastc == '\"')
87. in\_string = false;
88. }
89. *// 在多行注释中*
90. else if (in\_comment) {
91. if (buffer[i+1]) {
92. *// 多行注释结束*
93. if (buffer[i] == '\*' && buffer[i+1] == '/') {
94. *// 多行注释中有换行*
95. if (change) {
96. in\_comment = false;
97. change = false;
98. *// 找到下一个非空格字符*
99. i = i + 2;
100. while (buffer[i] && isspace(buffer[i]))
101. i++;
102. *// 说明后面没有非空格字符*
103. if (!buffer[i]) {
104. fprintf(out\_fp, "\n");
105. break;
106. }
107. else
108. continue;
109. }
110. *// 多行注释中无换行*
111. else {
112. i = i + 2;
113. *// 多行注释结束时到行末*
114. if (!buffer[i] || buffer[i] == '\n') {
115. fprintf(out\_fp, "\n");
116. in\_comment = false;
117. change = false;
118. break;
119. }
120. *// 多行注释结束时未到行末*
121. else {
122. *// 前有空格，后有空格，直接忽视该多行注释即可*
123. if (is\_space && isspace(buffer[i])) {
124. in\_comment = false;
125. change = false;
126. continue;
127. }
128. }
129. }
130. }
131. *// 多行注释未结束*
132. else
133. i++;
134. }
135. else {
136. fprintf(out\_fp, "\n");
137. change = true;
138. break;
139. }
140. }
141. }
142. }
143. }

代码解释：

* + 1. 主循环处理：
       1. 使用fgets逐行读取源代码，对每一行进行处理；
       2. 对于每个字符，根据当前的状态（是否在字符串中、是否在多行注释中）采取相应的处理策略。
    2. 字符串和多行注释的处理：
       1. 当遇到双引号(")，如果当前不在字符串或多行注释中，切换到字符串状态；
       2. 当遇到多行注释起始符(/ \*)，如果当前不在字符串中，则切换到多行注释状态。
    3. 空白字符的处理：
       1. 在非字符串、非多行注释文本中，连续的空白字符（除换行符外）被压缩为一个空格；
       2. 行首的空白字符在非多行注释、非字符串文本中被完全忽略。
    4. 输出：
       1. 对于每个需要输出的字符，根据上述条件判断，决定是否输出到结果文件中。

1. 设计词法分析程序
   1. 要求：。输入以“#”为结束符的源程序，输出为各类单词表和单词串文件。源程序和输出的单词串均以文件的形式存放。单词的自身值均为其对应的表的指针，如标识符表的指针、常数表的指针等。
   2. 设计思路：
      1. 核心功能：
         1. 读取和识别单词：从文件读取文本，并识别关键字、标识符、常数、字符常量、字符串及运算符等；
         2. 单词分类和存储：将识别的单词根据其类型存储到对应的表中，例如标识符表、常数表、字符表和字符串表；
         3. 输出文件：创建一个新的输出文件，其中原始代码中的单词被其在相应表中的指针替换，实现代码的标记化；
         4. 错误处理：能够识别未定义字符错误、字符常量错误和字符串未闭合错误，并将错误类型和位置记录到文件中；
         5. 结束处理：当遇到“#”结束符时，停止读取和处理文件，确保程序能够正确结束。
      2. 数据结构：
         1. 使用链表动态记录用户自定义标识符、常数、字符串字面量以及字符常量；
         2. 使用一个定长的字符串二维数组记录关键字、界限符、运算符，数组下标代表类别号；
      3. 词法分析处理：
         1. 读取操作：逐行读取源文件，使用缓冲区存储读取的内容；
         2. 单词识别与输出：遍历缓冲区中的字符，根据当前字符和上下文状态识别单词的类型并进行相应处理；
         3. 错误记录：识别单词失败则说明出现未定义字符错误，字符常量错误与字符串未闭合错误在各自状态中进行处理；
         4. 输出处理：将识别的单词的类别号等输入结果文件中保存。
   3. 具体代码实现如下：
2. #define TOKEN\_NUM 42
3. #define BUFFER\_SIZE 1024
4. *// 常数表*
5. typedef struct Constant {
6. int id;
7. int value;
8. struct Constant \*next;
9. } Constant;
10. *// 标识符表*
11. typedef struct Identifier {
12. int id;
13. char \*value;
14. struct Identifier \*next;
15. } Identifier;
16. *// 字符常量表*
17. typedef struct Character {
18. int id;
19. char value;
20. struct Character \*next;
21. } Character;
22. *// 字符串字面量表*
23. typedef struct String {
24. int id;
25. char \*value;
26. struct String \*next;
27. } String;
28. *// 单词表*
29. char TokenList[][TOKEN\_NUM] = {
30. "void", "int", "char",
31. "if", "else", "while", "for", "main",
32. "break", "continue", "return", "NULL", "", "", "", "",
33. ";", ",", "(", ")", "[", "]", "{", "}", "+",
34. "-", "\*", "/", "%", "=", "<", ">", "!", "+=", "-=",
35. "\*=", "/=", "%=", "==", "<=", ">=", "!="
36. };
37. *// 词法分析函数*
38. void lexer(FILE \*fp, FILE \*out\_fp, FILE \*out\_fp\_table) {
39. *// 初始化各链表*
40. Constant \*head\_constant = (Constant \*)malloc(sizeof(Constant));
41. head\_constant->id = -1;
42. head\_constant->value = 0;
43. head\_constant->next = NULL;
44. Character \*head\_character = (Character \*)malloc(sizeof(Character));
45. head\_character->id = -1;
46. head\_character->value = '\0';
47. head\_character->next = NULL;
48. String \*head\_string = (String \*)malloc(sizeof(String));
49. head\_string->id = -1;
50. head\_string->value = NULL;
51. head\_string->next = NULL;
52. Identifier \*head\_identifier = (Identifier \*)malloc(sizeof(Identifier));
53. head\_identifier->id = -1;
54. head\_identifier->value = NULL;
55. head\_identifier->next = NULL;
56. char buffer[BUFFER\_SIZE+1];
57. int line = 1; *// 当前行数*
58. bool end\_flag = false; *// 结束标志*
59. while (fgets(buffer, BUFFER\_SIZE, fp) != NULL) {
60. int i = 0;
61. while (buffer[i]) {
62. *// 跳过空格，处理换行*
63. if (isspace(buffer[i])) {
64. if (buffer[i] == '\n')
65. line++;
66. i++;
67. continue;
68. }
69. *// 常数处理*
70. if (buffer[i] <= '9' && buffer[i] >= '0') {
71. int tmp = buffer[i++] - '0';
72. *// 记录并计算该常数值，直到第一个非数字字符*
73. while (buffer[i] <= '9' && buffer[i] >= '0')
74. tmp = tmp \* 10 + (buffer[i++] - '0');
75. *// 查询常数表*
76. Constant \*p = head\_constant;
77. while (p->next) {
78. if (p->next->value == tmp) {
79. fprintf(out\_fp, "<13, Constant, %d>\n", p->next->id);
80. break;
81. }
82. p = p->next;
83. }
84. *// 未在常数表中查询到结果，在表末新建该常数记录*
85. if (p->next == NULL) {
86. Constant \*q = (Constant \*)malloc(sizeof(Constant));
87. q->id = p->id + 1;
88. q->value = tmp;
89. q->next = NULL;
90. p->next = q;
91. fprintf(out\_fp, "<13, Constant, %d>\n", p->next->id);
92. }
93. }
94. *// 字符常量处理*
95. else if (buffer[i] == '\'') {
96. i++;
97. *// 判断是否为正常的字符常量*
98. if (buffer[i] && buffer[i+1] == '\'') {
99. char ch = buffer[i];
100. *// 查询字符常量表*
101. Character \*p = head\_character;
102. while (p->next) {
103. if (p->next->value == ch) {
104. fprintf(out\_fp, "<15, Constant, %d>\n", p->next->id);
105. break;
106. }
107. p = p->next;
108. }
109. *// 未在字符常量表中查询到结果，在表末新建该字符常量记录*
110. if (p->next == NULL) {
111. Character \*q = (Character \*)malloc(sizeof(Character));
112. q->id = p->id + 1;
113. q->value = ch;
114. q->next = NULL;
115. p->next = q;
116. fprintf(out\_fp, "<15, Constant, %d>\n", p->next->id);
117. }
118. i = i + 2;
119. }
120. *// 字符常量的错误记录*
121. else {
122. fprintf(out\_fp, "--------------------Character error in line %d.\n", line);
123. *// 直到下一个空格或结尾*
124. while (buffer[i] && !isspace(buffer[i]))
125. i++;
126. }
127. }
128. *// 字符串字面量处理*
129. else if (buffer[i] == '\"') {
130. i++;
131. int len = 0;
132. while (buffer[i + len] && buffer[i + len] != '\"')
133. len++;
134. *// 判断字符串是否正确闭合*
135. if (buffer[i + len]) {
136. *// 记录字符串字面量*
137. char \*pc = (char \*)malloc(sizeof(char)\*(len+1));
138. for (int j = 0; j < len; j++)
139. pc[j] = buffer[i + j];
140. pc[len] = '\0';
141. *// 查询字符串字面量表*
142. String \*p = head\_string;
143. while (p->next) {
144. if (!strcmp(p->next->value, pc)) {
145. fprintf(out\_fp, "<14, Constant, %d>\n", p->next->id);
146. free(pc);
147. break;
148. }
149. p = p->next;
150. }
151. *// 未在字符串字面量表中查询到结果，在表末新建记录*
152. if (p->next == NULL) {
153. String \*q = (String \*)malloc(sizeof(String));
154. q->id = p->id + 1;
155. q->next = NULL;
156. q->value = pc;
157. p->next = q;
158. fprintf(out\_fp, "<14, Constant, %d>\n", p->next->id);
159. }
160. i = i + len + 1;
161. }
162. *// 字符串未正确闭合的错误记录*
163. else {
164. fprintf(out\_fp, "--------------------String error in line %d.\n", line);
165. i = i + len;
166. }
167. }
168. *// 标识符处理*
169. else if ((buffer[i] <= 'Z' && buffer[i] >= 'A') || (buffer[i] <= 'z' && buffer[i] >= 'a') || (buffer[i] == '\_')) {
170. int len = 1;
171. while ((buffer[i+len] <= 'Z' && buffer[i+len] >= 'A') || (buffer[i+len] <= 'z' && buffer[i+len] >= 'a') || (buffer[i+len] == '\_')
172. || (buffer[i+len] <= '9' && buffer[i+len] >= '0'))
173. len++;
174. *// 记录标识符*
175. char \*pc = (char \*)malloc(sizeof(char)\*(len+1));
176. for (int j = 0; j < len; j++)
177. pc[j] = buffer[i+j];
178. pc[len] = '\0';
179. *// 查询是否为关键字*
180. int idx\_kevword;
181. for (idx\_kevword = 0; idx\_kevword <= 11; idx\_kevword++) {
182. if (!strcmp(TokenList[idx\_kevword], pc)) {
183. fprintf(out\_fp, "<%d, Keyword, %s>\n", idx\_kevword, TokenList[idx\_kevword]);
184. free(pc);
185. break;
186. }
187. }
188. if (idx\_kevword <= 11) {
189. i = i + len;
190. continue;
191. }
192. *// 查询标识符表*
193. Identifier \*p = head\_identifier;
194. while (p->next) {
195. if (!strcmp(p->next->value, pc)) {
196. fprintf(out\_fp, "<12, Identifier, %d>\n", p->next->id);
197. free(pc);
198. break;
199. }
200. p = p->next;
201. }
202. *// 未在标识符表中查询到结果，在表末新建记录*
203. if (p->next == NULL) {
204. Identifier \*q = (Identifier \*)malloc(sizeof(Identifier));
205. q->next = NULL;
206. q->id = p->id + 1;
207. q->value = pc;
208. p->next = q;
209. fprintf(out\_fp, "<12, Identifier, %d>\n", p->next->id);
210. }
211. i = i + len;
212. }
213. *// 符号处理*
214. else {
215. *// 查询是否为界限符*
216. char ch = buffer[i];
217. int idx\_delimiter;
218. for (idx\_delimiter = 16; idx\_delimiter <= 23; idx\_delimiter++) {
219. if (TokenList[idx\_delimiter][0] == ch) {
220. fprintf(out\_fp, "<%d, Delimiter, %s>\n", idx\_delimiter, TokenList[idx\_delimiter]);
221. break;
222. }
223. }
224. if (idx\_delimiter <= 23) {
225. i++;
226. continue;
227. }
228. *// 查询是否为运算符*
229. int idx\_operator;
230. int both\_flag = 0;
231. for (idx\_operator = 24; idx\_operator <= 32; idx\_operator++) {
232. if (TokenList[idx\_operator][0] == ch) {
233. if (buffer[i+1] == '=') {
234. fprintf(out\_fp, "<%d, Operator, %s>\n", idx\_operator+9, TokenList[idx\_operator+9]);
235. both\_flag = 1;
236. } else
237. fprintf(out\_fp, "<%d, Operator, %s>\n", idx\_operator, TokenList[idx\_operator]);
238. break;
239. }
240. }
241. if (idx\_operator <= 32) {
242. i = i + 1 + both\_flag;
243. continue;
244. }
245. *// 查询是否为结束符号*
246. if (ch == '#') {
247. end\_flag = true;
248. break;
249. }
250. *// 未定义符号的错误记录*
251. fprintf(out\_fp, "--------------------Undefined error in line %d.\n", line);
252. i++;
253. while (buffer[i] && !isspace(buffer[i]))
254. i++;
255. }
256. }
257. if (end\_flag)
258. break;
259. }
260. *// 打印常数表，字符常量表，字符串表，标识符表，同时销毁*
261. *// 打印标识符表*
262. fprintf(out\_fp\_table, "<Identifier>\n-id, value-\n");
263. Identifier \*p\_ide = head\_identifier->next, \*q\_ide = NULL;
264. while (p\_ide) {
265. q\_ide = p\_ide;
266. p\_ide = p\_ide->next;
267. q\_ide->next = NULL;
268. fprintf(out\_fp\_table, "%d, %s\n", q\_ide->id, q\_ide->value);
269. free(q\_ide->value);
270. free(q\_ide);
271. q\_ide = NULL;
272. }
273. free(head\_identifier);
274. fprintf(out\_fp\_table, "\n");
275. *// 打印字符常量表*
276. fprintf(out\_fp\_table, "<Constant-Character>\n-id, value-\n");
277. Character \*p\_cha = head\_character->next, \*q\_cha = NULL;
278. while (p\_cha) {
279. q\_cha = p\_cha;
280. p\_cha = p\_cha->next;
281. q\_cha->next = NULL;
282. fprintf(out\_fp\_table, "%d, %c\n", q\_cha->id, q\_cha->value);
283. free(q\_cha);
284. q\_cha = NULL;
285. }
286. free(head\_character);
287. fprintf(out\_fp\_table, "\n");
288. *// 打印字符串表*
289. fprintf(out\_fp\_table, "<Constant-String>\n-id, value-\n");
290. String \*p\_str = head\_string->next, \*q\_str = NULL;
291. while (p\_str) {
292. q\_str = p\_str;
293. p\_str = p\_str->next;
294. q\_str->next = NULL;
295. fprintf(out\_fp\_table, "%d, %s\n", q\_str->id, q\_str->value);
296. free(q\_str->value);
297. free(q\_str);
298. q\_str = NULL;
299. }
300. free(head\_string);
301. fprintf(out\_fp\_table, "\n");
302. *// 打印常数表*
303. fprintf(out\_fp\_table, "<Constant-Constant>\n-id, value-\n");
304. Constant \*p\_con = head\_constant->next, \*q\_con = NULL;
305. while (p\_con) {
306. q\_con = p\_con;
307. p\_con = p\_con->next;
308. q\_con->next = NULL;
309. fprintf(out\_fp\_table, "%d, %d\n", q\_con->id, q\_con->value);
310. free(q\_con);
311. q\_con = NULL;
312. }
313. free(head\_constant);
314. fprintf(out\_fp\_table, "\n");
315. }

代码解释：

* + 1. 初始化数据结构：使用malloc为各种单词类型（常数、标识符、字符常量和字符串字面量）分别初始化链表；
    2. 读取和处理循环：通过fgets从文件中逐行读取内容到缓冲区buffer。使用嵌套的while循环遍历每行中的每个字符，根据字符的类型和上下文进行处理；
    3. 字符分类和处理：
       1. 空白字符：跳过，但换行符增加行数计数；
       2. 数字：识别整数常量，并在常数表中查找或插入；
       3. 字符常量：处理形如'a'的字符常量，并在字符常量表中查找或插入，包括错误处理；
       4. 字符串字面量：处理形如"hello"的字符串，并在字符串字面量表中查找或插入，包括处理未闭合的字符串错误；
       5. 标识符和关键字：识别以字母或下划线开头的序列，并判断其是否为预定义的关键字或需要插入标识符表；
       6. 符号：识别各种操作符和界定符，并分类处理；
       7. 结束符号：识别#作为文件的结束符，触发结束标志；
       8. 其他：未定义错误。

# 第五部分 测试计划（报告）

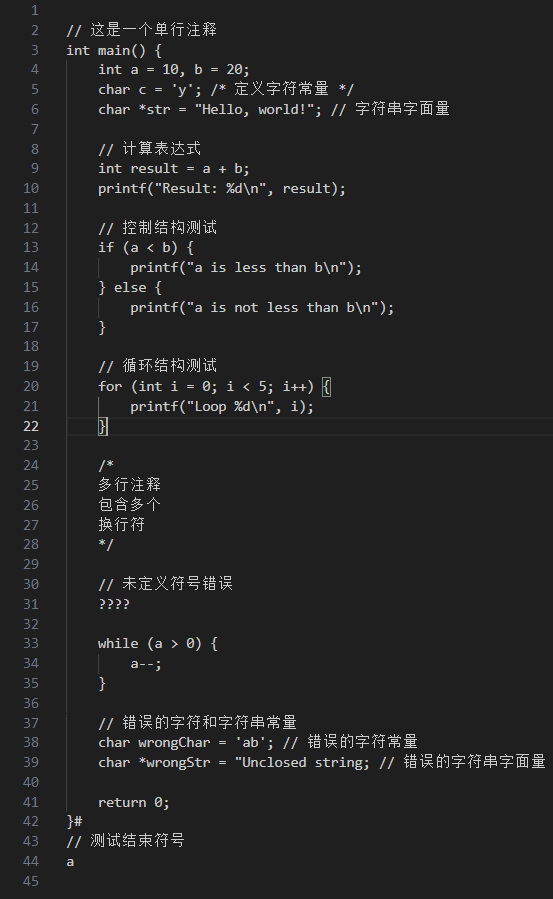
1. 设计测试用例思路

要对预处理和词法分析程序进行充分测试，需要构建一个C语言的源文件作为输入，这个测试用例应当涵盖程序可能遇到的各种情况，包括各种数据类型、结构以及错误情况。其具体包含内容如下所示：

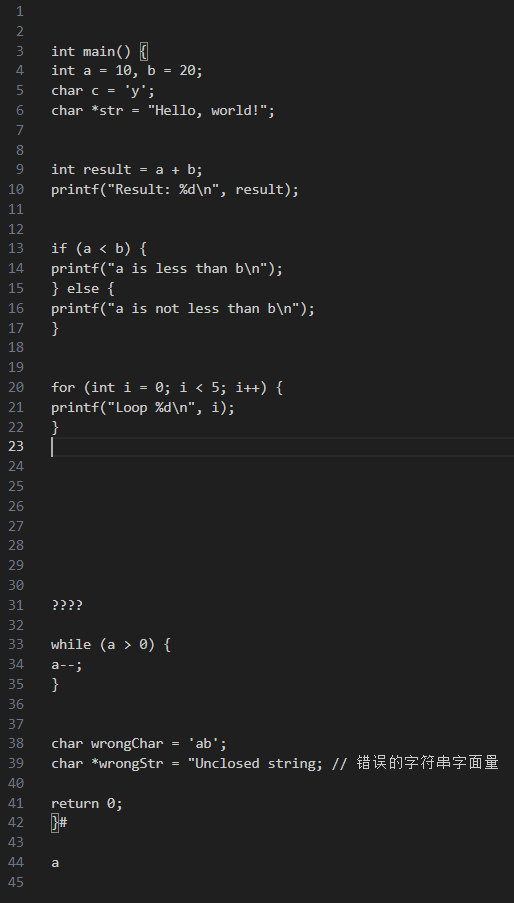
* 1. 算术表达式。
  2. 各种结构(顺序、分支、循环)。
  3. 函数定义和调用。
  4. 单行和多行注释。
  5. 常数、字符串和字符常量。
  6. 错误情况，例如出现未定义的符号、未闭合的字符串或字符常量出错。
  7. 已定义的运算符和界限符。
  8. 已定义的关键字。
  9. 自定义的标识符。

1. 具体的测试用例

源代码、测试用例与结果的具体记录：<https://github.com/nagumiyug/Lexer>

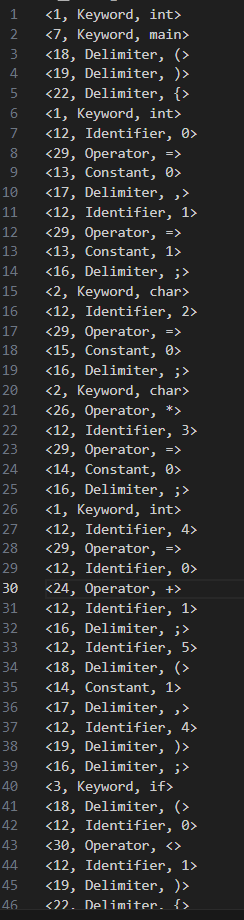
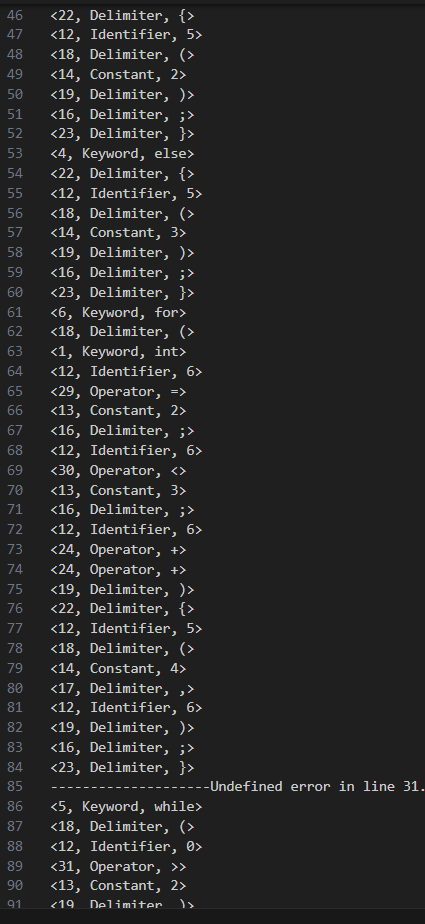


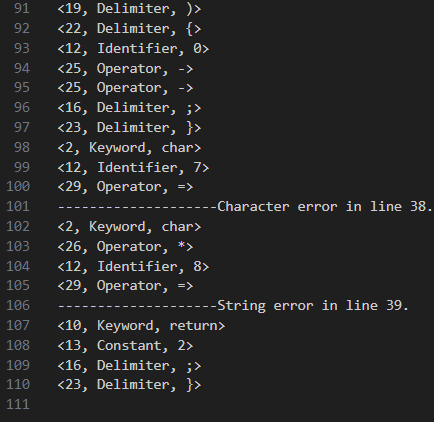
1. 测试结果
   1. 预处理结果



处理后的结果成功删除了源程序中的注释和多余的空格类符号，同时还确保处理后的程序的行数与源程序保持一致。

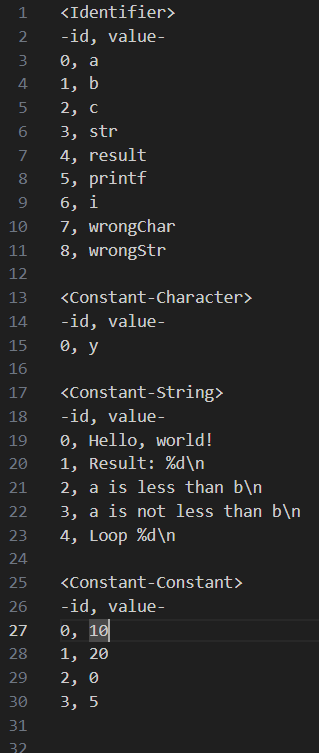
* 1. 词法分析结果



词法分析后的结果成功将程序中的各单词转化为类别号+类别名+自身值的组合序列，同时还准确在“#”出现的地方停止，还成功识别并指出了三处错误与其出现的行数。

* 1. 词法分析中产生的4个表



按照源程序中各单词的出现顺序进行记录的标识符表、字符常量表、字符串字面量表以及常数表都正确建立并完成记录。