词法分析程序的设计与实现

杨晨
学号 2021212171
北京邮电大学计算机学院

日期：2023 年 10 月 1 日

目录

1 概述

1.1 实验内容

1. 选定源语言，c 语言
2. 可以识别出用源语言编写的源程序中的每个单词符号，并以记号的形式输出每个单词符号。
3. 可以识别并跳过源程序中的注释。
4. 可以统计源程序中的语句行数、各类单词的个数、以及字符总数，并输出统计结果。
5. 检查源程序中存在的词法错误，并报告错误所在的位置。
6. 对源程序中出现的错误进行适当的恢复，使词法分析可以继续进行，对源程序进行一次扫描，即可检查并报告源程序中存在的所有词法错误。

1.2 开发环境

- Windows10
- Visual Studio Code

2 程序的功能模块划分

2.1 识别数值

在代码中，我参考了课上的 PPT 中关于数值识别的自动机设计，并根据您的需求进行了修改。这个自动机用于识别数值常量，并根据常规的数值表示规则确定常量的类型。

状态 2 是数值常量的起始状态，状态 3 表示遇到了小数点，状态 4 表示小数点后的数字，状态 5 表示遇到了指数符号，状态 6 表示指数符号后的正负号，状态 7 表示指数后的数字。

该自动机的工作过程如下：

- 初始状态为状态 2，开始处理数值常量。
- 通过循环遍历输入字符，并根据当前状态和输入字符执行相应的操作。
- 在状态 2 中，如果遇到数字，则保持状态 2 并继续前进，如果遇到小数点，则转移到状态 3，如果遇到指数符号，则转移到状态 5，如果遇到不是 e（E）的字母，则将 token 的类型设置为`"Error"`，并记录错误信息。
- 在状态 3 中，如果遇到数字，则转移到状态 4，表示遇到小数点后的数字部分。如果遇到指数符号，则转移到状态 5，遇到其他字符（非;），则将 token 的类型设置为`"Error"`，并记录错误信息。
- 在状态 4 中，如果遇到数字，则保持状态 4 并继续前进，如果遇到指数符号，则转移到状态 5，如果遇到其他字符，则将 token 的类型设置为`"Error"`，并记录错误信息。
- 在状态 5 中，如果遇到数字，则转移到状态 7，表示遇到指数后的数字部分。如果遇到正负号，则转移到状态 6，表示遇到指数后的正负号。如果遇到其他字符，则将 token 的类型设置为`"Error"`，并记录错误信息。

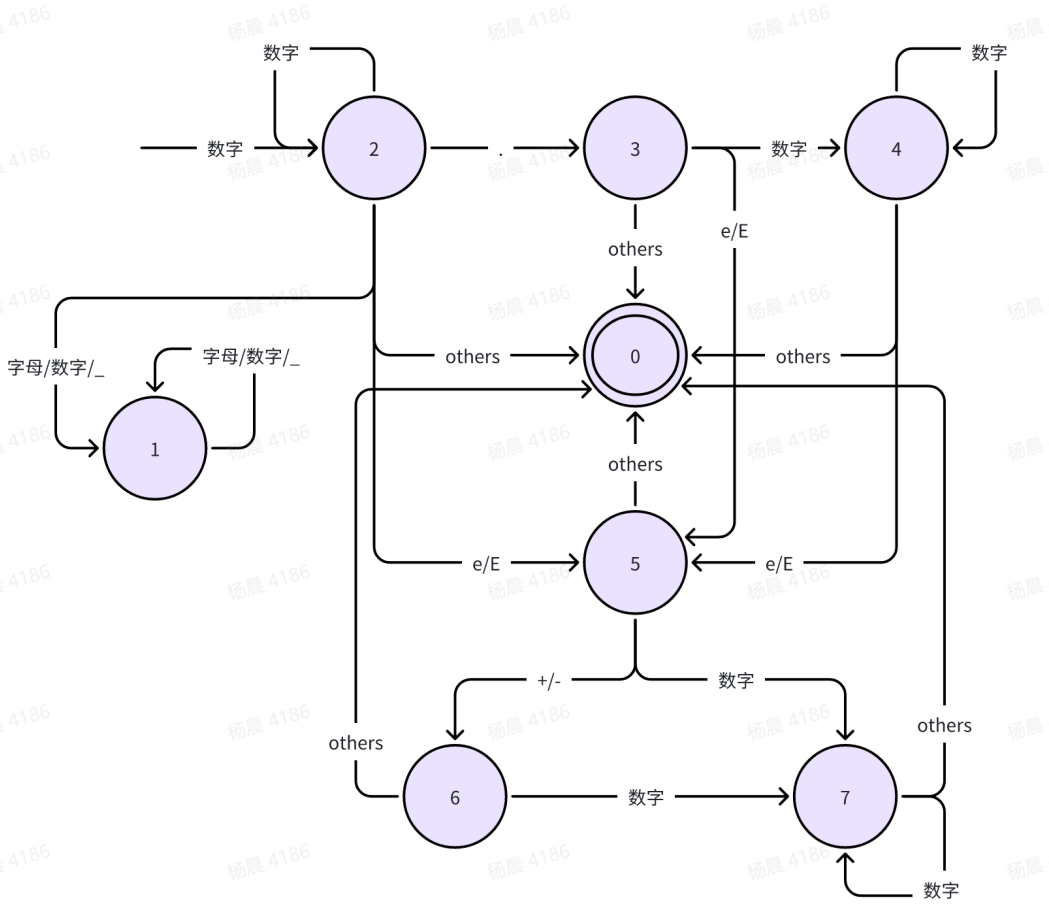

图 1: 识别 Numerical 的自动机

- 在状态 6 中，如果遇到数字，则转移到状态 7，表示遇到指数后的数字部分。如果遇到其他字符，则将 token 的类型设置为"Error"，并记录错误信息。
- 在状态 7 中，如果遇到数字，则保持状态 7 并继续前进，如果遇到其他字符，则将 token 的类型设置为"Error"，并记录错误信息。
- 当自动机状态变为 0 时，表示数值常量识别结束。

2.2 识别单字符常量

这个自动机实现了一个自动机来识别字符常量。自动机的初始状态为 1，表示开始处理字符常量。

该自动机的工作过程如下：

- 在状态 1 中，首先检查下一个字符。如果下一个字符是反斜杠（\），则表示遇到了转义字符，进入状态 3。如果下一个字符是单引号（'），则表示遇到了空字符，进入状态 0。否则，进入状态 2，表示遇到了普通字符。
- 在状态 2 中，继续读取下一个字符。如果下一个字符是单引号（'），则表示字符常量结束，进入状态 0。否则，表示遇到了多字符字符常量或者遇到了未闭合的单引号（'），将 token 的类型设置为"Error"，并记录错误信息。

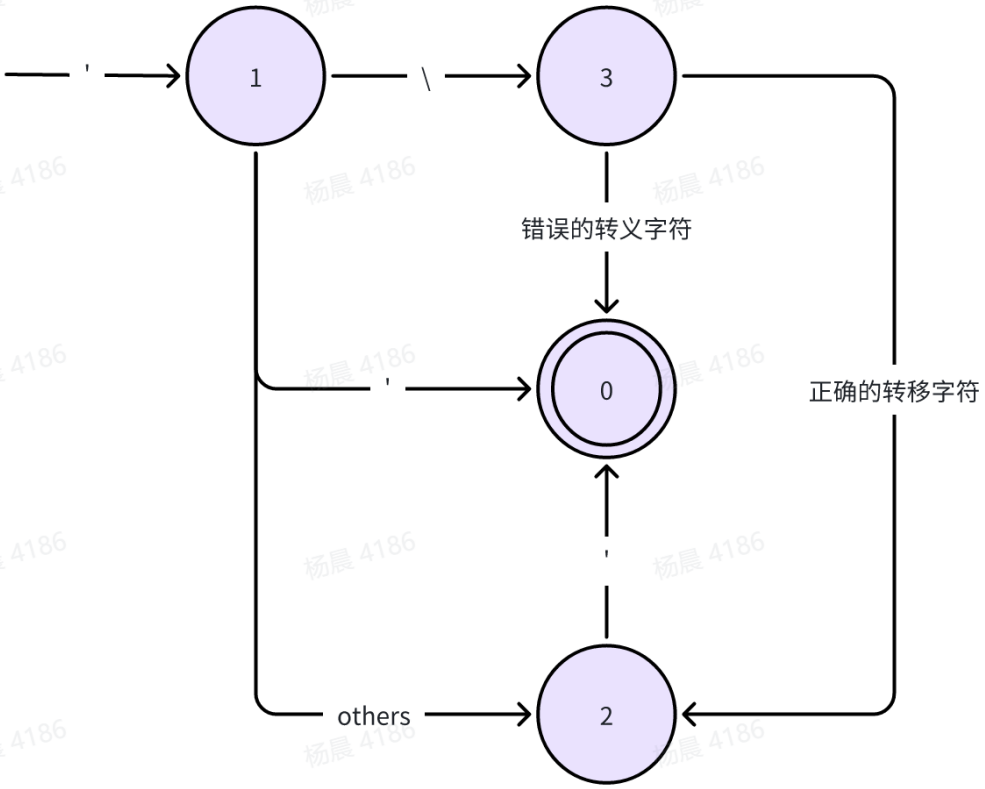

图 2: 识别 CharConstant 的自动机

- 在状态 3 中，继续读取下一个字符。如果下一个字符是合法的转义字符（例如：a、b、f、n、r、t、v、\、’、”、?），则进入状态 2，表示转义字符合法。否则，表示遇到了非法的转义字符，将 token 的类型设置为"Error"，并记录错误信息。
- 当自动机状态变为 0 时，表示字符常量识别结束。

2.3 识别字符串常量

这段代码实现了一个自动机来识别字符串字面串。自动机的初始状态为 1，表示开始处理字符常量。

自动机的工作原理如下：

- 在状态 1 中，首先检查下一个字符。如果下一个字符是反斜杠（\），则表示遇到了转义字符，进入状态 3。如果下一个字符是单引号（’），则表示遇到了空字符，进入状态 0。否则，进入状态 2，表示遇到了普通字符。
- 在状态 2 中，继续读取下一个字符。如果下一个字符是单引号（’），则表示字符常量结束，进入状态 0。否则，表示遇到了多字符字符常量，将 token 的类型设置为"Error"，并记录错误信息。
- 在状态 3 中，继续读取下一个字符。如果下一个字符是合法的转义字符（例如：a、b、f、n、r、t、v、\、’、”、?），则进入状态 2，表示转义字符合法。否则，表示遇到了非法的转义字符，将 token 的类型设置为"Error"，并记录错误信息。

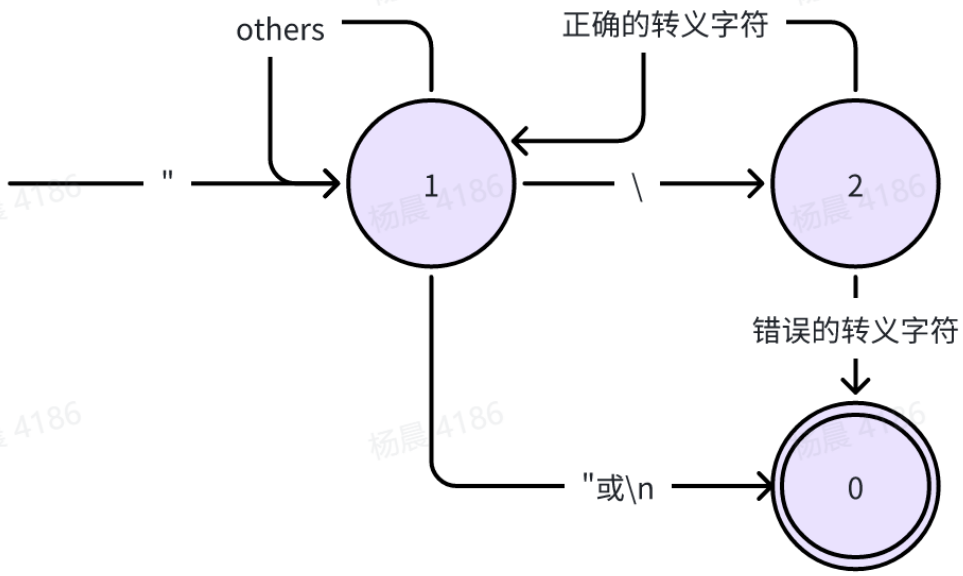

图 3: 识别 StringLiteral 的自动机

- 当自动机状态变为 0 时，表示字符常量识别结束，根据 buffer 中的内容设置 token 的值。

通过这个自动机，您可以识别并解析输入中的字符常量，并根据识别结果设置相应的 token 类型和值。当遇到转义字符时，自动机能够正确处理合法的转义字符，同时也能检测到非法的转义字符并报告错误。

2.4 各类标点符号

```
if (!isWhitespace(c))
{
    Token token("Punctuator", countLine, countColumn, "");
    eatChar();
    bool has_error = false;
    bool is_comment = false;
    switch (c) // 标点符号的特殊处理
    {
    case '.': // 识别.或...
        c = peekChar();
        if (c == '.')
        {
            eatChar();
            if ((c = peekChar()) == '.') // 识别...
            {
```

```
16                  eatChar();
17              }
18              else // 识别错误的省略号
19              {
20                  has_error = true;
21                  token.setType("Error");
22              }
23          }
24          break;
25      case '>': // >或>=或>>或>>=
26          c = peekChar();
27          if (c == '>') // >>或>>=
28          {
29              eatChar();
30              c = peekChar();
31              if (c == '=') // >>=
32              {
33                  eatChar();
34              }
35          }
36          else if (c == '=') // >=
37              eatChar();
38          break;
39      case '<': // <或<=或<<或<<=
40          c = peekChar();
41          if (c == '<') // <<或<<=
42          {
43              eatChar();
44              if ((c = peekChar()) == '=') // <<=
45              {
46                  eatChar();
47              }
48          }
49          else if (c == '=') // <=
50              eatChar();
51          break;
52      case '+': // +或++或+=
53          c = peekChar();
```

```
54          if (c == '=' || c == '+')
55              eatChar();
56          break;
57      case '-': // -或--或-=或->
58          c = peekChar();
59          if (c == '=' || c == '-' || c == '>')
60              eatChar();
61          break;
62      case '&': // &或&&或&=
63          c = peekChar();
64          if (c == '=' || c == '&')
65              eatChar();
66          break;
67      case '|': // |或||或|=
68          c = peekChar();
69          if (c == '=' || c == '|')
70              eatChar();
71          break;
72      case '*':
73      case '%':
74      case '^':
75      case '=':
76      case '!':
77          c = peekChar();
78          if (c == '=') // *=或%=或^=或==或!=
79              eatChar();
80          break;
81      case ';':
82      case '{':
83      case '}':
84      case ',':
85      case ':':
86      case '(':
87      case ')':
88      case '[':
89      case ']':
90      case '~':
91      case '?':
```

```
92          break; // 单字符标点符号
93      case '#':  // 预处理指令,如#include
94          is_comment = true; // 预处理指令不输出
95          do
96          {
97              c = eatChar();
98          } while (c != '\n'); // 跳过一行
99          break;
100     case '/': // /或/=或/*或//
101         c = peekChar();
102         if (c == '*' || c == '/') // 注释
103         {
104             ... // 在下一个subsection分析
105         }
106         else if (c == '=') // /=
107             eatChar();
108         break;
109     default: // 未知字符
110         has_error = true;
111         token.setType("Error");
112         token.setError("unexpected character");
113         count_Error++;
114         token.setValue(buffer);
115         break;
116     }
117     res &= !has_error; // 词法分析是否成功，有错误返回false
118     if (!is_comment)   // 注释不输出
119     {
120         if (token.getType() == "Punctuator")
121         {
122             count_Punctuator++;
123         }
124         token.setValue(buffer);
125         std::cout << token << std::endl;
126     }
127     else if (has_error) // 有错误输出错误信息
128     {
129         std::cout << token << std::endl;
```

```
    }
}
else
{
    eatChar();
}
```

该代码片段对字符流进行词法分析，并识别各种标点符号。其大体流程如下

- 代码片段首先检查当前字符 c 是否为空白字符。如果不是空白字符，分析过程继续。
- 创建一个 Token 对象，将其类型设为 `Punctuator`，并初始化行号和列号。
- 调用 `eatChar()` 函数来消耗当前字符，并移动到流中的下一个字符。
- 代码使用 switch 语句来处理不同的标点符号和特殊情况。
- 对于每个标点符号，代码检查下一个字符（`peekChar()`），并根据观察到的模式执行特定的操作。
- 如果在分析标点符号过程中遇到任何错误，将将变量 `has_error` 设为 `true`，并将 Token 对象的类型设为 `"Error"`。错误消息也相应地设置。
- 代码跟踪遇到的错误数量（`count_Error`）和识别的标点符号数量（`count_Punctuator`）。
- 如果分析的字符是预处理指令的一部分（例如以 `'#'` 开头的行），则视为注释并跳过直到行末。
- 如果字符不是注释，则将 Token 对象的值设置为处理后的缓冲区，并将其输出到控制台。
- 如果字符是注释，但同时存在错误，则将错误消息输出到控制台。
- 如果字符是空白字符，则使用 `eatChar()` 函数将其消耗掉。

上述代码片段展示了词法分析过程中对字符流中标点符号进行识别和处理的一部分。使用 switch 语句来处理不同的标点符号模式和特殊情况。代码跟踪分析过程中遇到的错误，并相应地输出识别的标记和错误消息。

2.5 识别注释

该自动机是用于识别和处理注释的。它基于状态转换的原理，根据输入字符的不同，切换自动机的状态，以便正确识别和处理不同类型的注释。

自动机的工作原理如下：

- 当遇到字符'/' 时，首先检查下一个字符。如果下一个字符是'*' 或'/'，则表示遇到了注释开始标记，进入注释处理状态。如果下一个字符是'*'，则进入状态 2，表示遇到了多行注释。如果下一个字符是'/'，则进入状态 4，表示遇到了单行注释。
- 状态 2 表示多行注释中的内容。如果下一个字符是'*'，则进入状态 3，表示可能是注释的结束。否则，继续保持在状态 2，表示注释内容继续。
- 状态 3 用于判断是否遇到了多行注释的结束。如果下一个字符是'*'，继续保持在状态 3。如果下一个字符是'/'，则表示多行注释结束，进入状态 0，注释处理结束。否则，说明还处于多行注释中，回到状态 2。

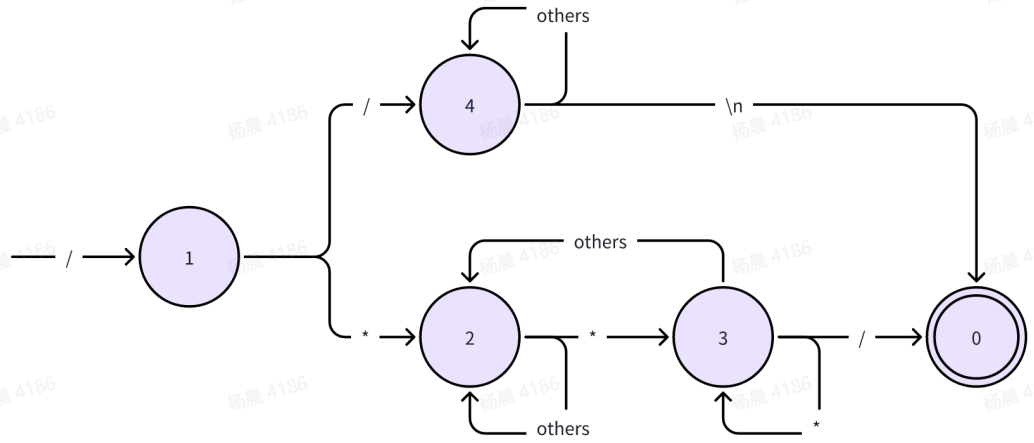

图 4: 识别 Comment 的自动机

- 状态 4 表示单行注释中的内容。如果下一个字符是'\n'，则表示单行注释结束，进入状态 0，注释处理结束。否则，继续保持在状态 4，表示注释内容继续。
- 状态 0 表示注释处理结束。

通过上述自动机，可以识别和处理不同类型的注释，包括多行注释和单行注释。自动机能够正确识别注释的开始和结束，并在注释处理结束后返回状态 0。

2.6 词法分析总框架

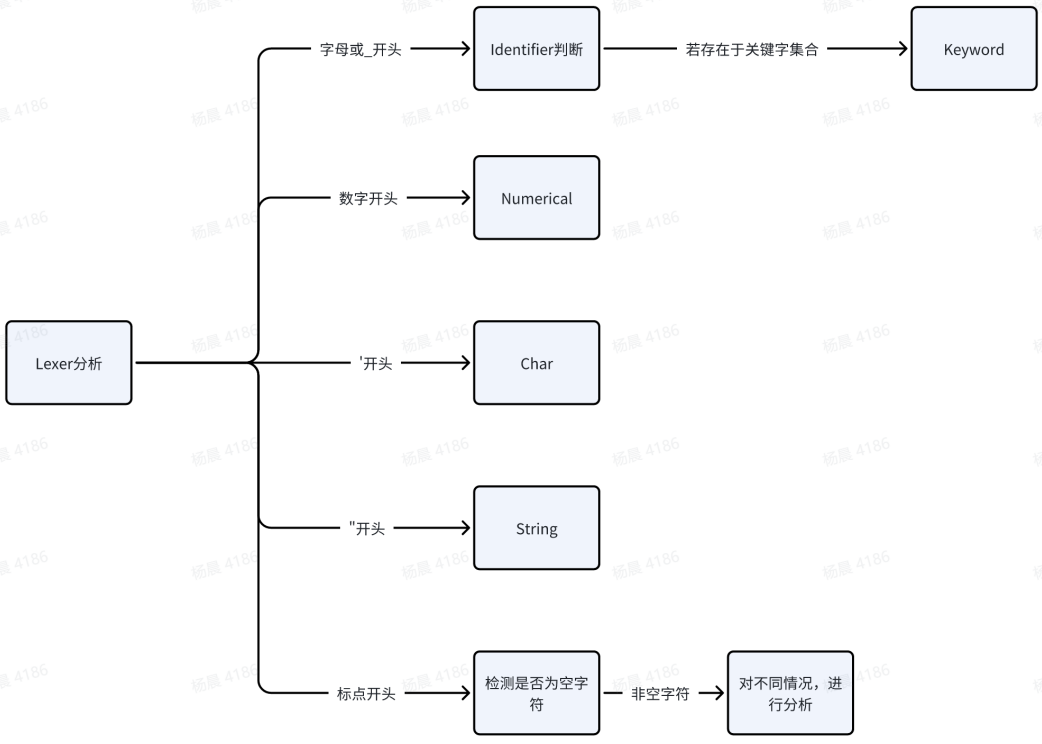

图 5: 词法分析流程

它从输入流中读取字符，直到文件结束或输入缓冲区中没有更多的字符为止。对于每个字符，它检查它是否是字母、数字、单引号、双引号或标点符号。

如果字符是字母，则假设它是标识符或关键字。它创建一个类型为 "Identifier" 的 Token 对象，并调用getNextIdentifier()函数从输入流中读取标识符的其余部分。如果标识符与预定义的关键字之一匹配，则词法分析器将标记的类型设置为"Keyword"，并增加一个计数器以记录找到的关键字数。

如果字符是数字，则假设它是数字常量。它创建一个类型为 "Numerical_Constant" 的 Token 对象，并调用 getNextNumerical() 函数从输入流中读取数字常量的其余部分。

如果字符是单引号，则会假设它是字符常量。它创建一个类型为 "Char_Constant" 的 Token 对象，并调用 getCharConstant() 函数从输入流中读取字符常量的其余部分。

如果字符是双引号，则会假设它是字符串字面量。它创建一个类型为 "String_Literal" 的 Token 对象，并调用 getStringLiteral() 函数从输入流中读取字符串字面量的其余部分。

如果字符是标点符号，则会创建一个类型为 "Punctuator" 的 Token 对象，并调用 eatChar() 函数从输入流中消耗字符。然后，它检查标点符号是否是特殊情况，例如点、大于号、小于号、加号、减号、和号、竖线、星号、百分号、脱字符、等号或感叹号。对于每个特殊情况，词法分析器执行适当的操作，例如从输入流中消耗额外的字符或如果遇到意外字符则将标记的类型设置为 "Error"。

词法分析器还处理注释和预处理指令。如果词法分析器遇到井号，它会假设它是预处理指令，例如 #include。它消耗该行的其余部分并忽略该指令。如果词法分析器遇到正斜杠，它会检查它是否是注释，单行注释或多行注释。如果它是注释，则词法分析器消耗注释的其余部分并忽略它。

3 使用说明

在 main.cpp 中，可以指定源代码文件的路径

```
Lexer lexer("../test/test1.c");
```

运行 main.cpp 后，将会输出词法分析的结果，如下

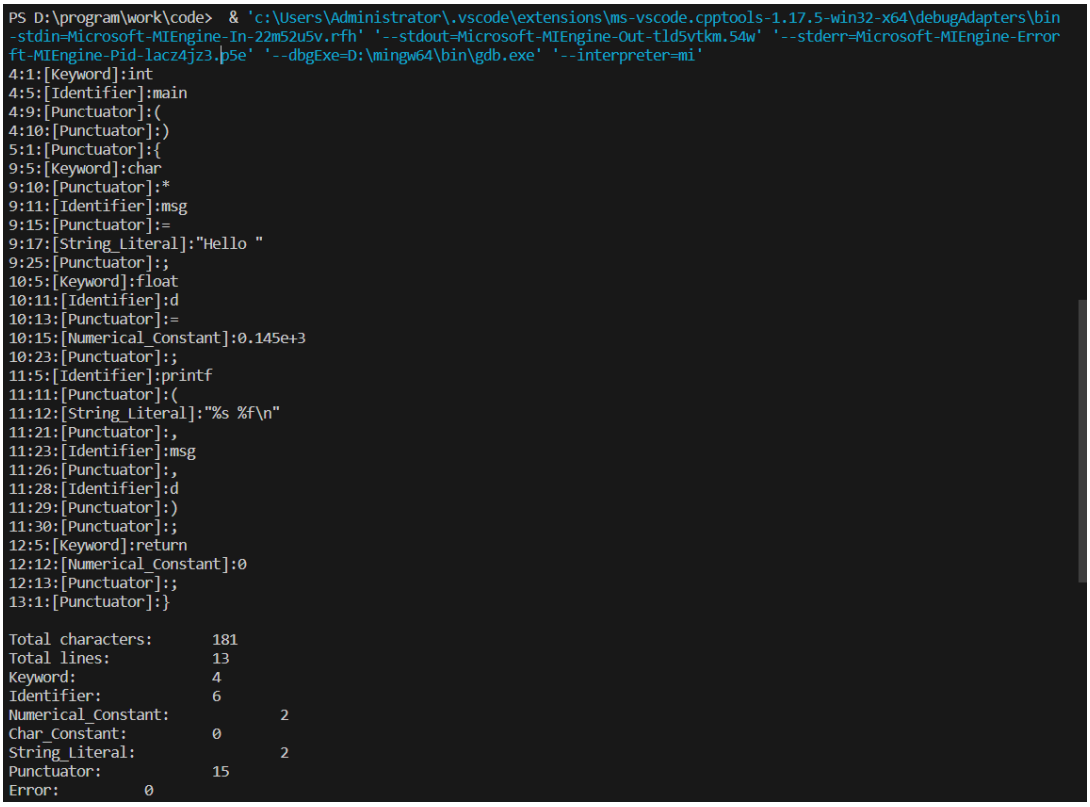
PS D:\program\work\code> & 'c:\Users\Administrator\.vscode\extensions\ms-vscode.cpptools-1.17.5-win32-x64\debugAdapters\bin
-stdin=Microsoft-MIEngine-In-22m52u5v.rfh' '--stdout=Microsoft-MIEngine-Out-tld5vtkm.54w' '--stderr=Microsoft-MIEngine-Error
ft-MIEngine-Pid-lacz4jz3.p5e' '--dbgExe=D:\mingw64\bin\gdb.exe' '--interpreter=mi'
4:1:[Keyword]:int
4:5:[Identifier]:main
4:9:[Punctuator]:(
4:10:[Punctuator]:)
5:1:[Punctuator]:{
9:5:[Keyword]:char
9:10:[Punctuator]:*
9:11:[Identifier]:msg
9:15:[Punctuator]:=
9:17:[String_Literal]:"Hello "
9:25:[Punctuator]:;
10:5:[Keyword]:float
10:11:[Identifier]:d
10:13:[Punctuator]:=
10:15:[Numerical_Constant]:0.145e+3
10:23:[Punctuator]:;
11:5:[Identifier]:printf
11:11:[Punctuator]:(
11:12:[String_Literal]:"%s %f\n"
11:21:[Punctuator]:,
11:23:[Identifier]:msg
11:26:[Punctuator]:,
11:28:[Identifier]:d
11:29:[Punctuator]:)
11:30:[Punctuator]:;
12:5:[Keyword]:return
12:12:[Numerical_Constant]:0
12:13:[Punctuator]:;
13:1:[Punctuator]:}
Total characters:	181
Total lines:	13
Keyword:	4
Identifier:	6
Numerical_Constant:	2
Char_Constant:	0
String_Literal:	2
Punctuator:	15
Error:	0

图 6: 程序运行效果

4 测试

4.1 测试集 1

该测试集用于测试程序在没有词法错误时，是否能正常运行。

4.1.1 输入的源代码

```
// Line comment *///*/
#include <stdio.h>

int main()
{
    /*
     * Block comment *
     */
    char *msg = "Hello ";
    char ch = 'w';
    float f = 0.145e+3;
    double d = 3.e3;
    float f2 = 0.145e-3;
    double d2 = 3.;
    int integer = 864;
    long long int longint = 1234567890123456789;
    printf("%s %f\n", msg, d);
    printf("%c\t%d\n", ch, integer);
    printf("%lld\v", longint);
    return 0;
}
```

4.1.2 输出结果

```
4:1:[Keyword]:int
4:5:[Identifier]:main
4:9:[Punctuator]:(
4:10:[Punctuator]:)
5:1:[Punctuator]:{
9:5:[Keyword]:char
9:10:[Punctuator]:*
```

```
9:11:[Identifier]:msg
9:15:[Punctuator]:=
9:17:[String_Literal]:"Hello "
9:25:[Punctuator]:;
10:5:[Keyword]:char
10:10:[Identifier]:ch
10:13:[Punctuator]:=
10:15:[Char_Constant]:'w'
10:18:[Punctuator]:;
11:5:[Keyword]:float
11:11:[Identifier]:f
11:13:[Punctuator]:=
11:15:[Numerical_Constant]:0.145e+3
11:23:[Punctuator]:;
12:5:[Keyword]:double
12:12:[Identifier]:d
12:14:[Punctuator]:=
12:16:[Numerical_Constant]:3.e3
12:20:[Punctuator]:;
13:5:[Keyword]:float
13:11:[Identifier]:f2
13:14:[Punctuator]:=
13:16:[Numerical_Constant]:0.145e-3
13:24:[Punctuator]:;
14:5:[Keyword]:double
14:12:[Identifier]:d2
14:15:[Punctuator]:=
14:17:[Numerical_Constant]:3.
14:19:[Punctuator]:;
15:5:[Keyword]:int
15:9:[Identifier]:integer
15:17:[Punctuator]:=
15:19:[Numerical_Constant]:864
15:22:[Punctuator]:;
16:5:[Keyword]:long
16:10:[Keyword]:long
16:15:[Keyword]:int
16:19:[Identifier]:longint
```

```
16:27:[Punctuator]:=
16:29:[Numerical_Constant]:12345678901234567890
16:48:[Punctuator]:;
17:5:[Identifier]:printf
17:11:[Punctuator]:(
17:12:[String_Literal]:"%s %f\n"
17:21:[Punctuator]:,
17:23:[Identifier]:msg
17:26:[Punctuator]:,
17:28:[Identifier]:d
17:29:[Punctuator]:)
17:30:[Punctuator]:;
18:5:[Identifier]:printf
18:11:[Punctuator]:(
18:12:[String_Literal]:"%c\t%d\n"
18:22:[Punctuator]:,
18:24:[Identifier]:ch
18:26:[Punctuator]:,
18:28:[Identifier]:integer
18:35:[Punctuator]:)
18:36:[Punctuator]:;
19:5:[Identifier]:printf
19:11:[Punctuator]:(
19:12:[String_Literal]:"%lld\v"
19:20:[Punctuator]:,
19:22:[Identifier]:longint
19:29:[Punctuator]:)
19:30:[Punctuator]:;
20:5:[Keyword]:return
20:12:[Numerical_Constant]:0
20:13:[Punctuator]:;
21:1:[Punctuator]:}

Total characters:       415
Total lines:            21
Keyword:                12
Identifier:             17
Numerical_Constant:     7
```

```
Char_Constant:          1
String_Literal:         4
Punctuator:             36
Error:                  0
No error found.
```

4.1.3 输出结果分析

代码总共包含了 415 个字符，分布在 21 行中。词法分析结果提供了每个词法单元的行号、列号和类型。

- 代码中包含了 12 个关键字。
- 代码中包含了 17 个标识符。
- 代码中包含了 7 个数值常量，包括浮点数和整数常量。
- 代码中包含了 1 个字符常量。
- 代码中包含了 4 个字符串字面值。
- 代码中包含了 36 个标点符号。

同时，词法分析结果未发现任何错误。

4.2 测试集 2

4.2.1 输入的源代码

```
1  int main()
2  {
3      double 2ch = 1.a;
4      double a = 1ee2;
5      char *num = "unclose\++";
6      char *str = "unclose\";
7      s = ''
8      w = \$
9      int a = '@;
10     . = 1.2.3;
11 }
12 /* unclose_Comment
```

4.2.2 输出结果

```
1:1:[Keyword]:int
1:5:[Identifier]:main
```

```
1:9:[Punctuator]:(
1:10:[Punctuator]:)
2:1:[Punctuator]:{
3:5:[Keyword]:double
3:12:[Error: illegal name]:2ch
3:16:[Punctuator]:=
3:18:[Error: illegal name]:1.a
3:21:[Punctuator]:;
4:5:[Keyword]:double
4:12:[Identifier]:a
4:14:[Punctuator]:=
4:16:[Error: Invalid exponent in numerical constant: exponent symbol
    without following digits]:1e
4:18:[Identifier]:e2
4:20:[Punctuator]:;
5:5:[Keyword]:char
5:10:[Punctuator]:*
5:11:[Identifier]:num
5:15:[Punctuator]:=
5:17:[Error: Invalid escape sequence in string literal]:"unclose\
5:26:[Punctuator]:++
5:28:[Error: unclosed string]:";
6:5:[Keyword]:char
6:10:[Punctuator]:*
6:11:[Identifier]:str
6:15:[Punctuator]:=
6:17:[Error: unclosed string]:"unclose\";
7:5:[Identifier]:s
7:7:[Punctuator]:=
7:9:[Char_Constant]:''
8:5:[Identifier]:w
8:7:[Punctuator]:=
8:9:[Error: unexpected character]:\
8:10:[Error: unexpected character]:$
9:5:[Keyword]:int
9:9:[Identifier]:a
9:11:[Punctuator]:=
9:13:[Error: multi-character character constant or unclosed
```

```
   character constant]:'@
9:15:[Punctuator]:;
10:5:[Punctuator]:.
10:7:[Punctuator]:=
10:9:[Numerical_Constant]:1.2
10:12:[Punctuator]:.
10:13:[Numerical_Constant]:3
10:14:[Punctuator]:;
11:1:[Punctuator]:}

Total characters:       188
Total lines:            13
Keyword:                6
Identifier:             8
Numerical_Constant:     2
Char_Constant:          1
String_Literal:         0
Punctuator:             21
Error:                  9
Error found.
```

4.2.3 输出结果分析

输出结果指示了在给定的代码中存在多个错误：

- 第 3 行第 12 列的[Error: illegal name]:2ch表示在该位置上发现了一个错误，指出标识符"2ch"不是一个合法的名称。
- 第 4 行第 16 列的[Error: Invalid exponent in numerical constant: exponent symbol without following digits]:1e表示在该位置上发现了一个错误，指出数字常量"1e"的指数部分缺少有效的数字。
- 第 5 行第 17 列的[Error: Invalid escape sequence in string literal]:"unclose\表示在该位置上发现了一个错误，指出字符串字面量"unclose"中有无效的转义序列。
- 第 5 行第 28 列的[Error: unclosed string]:"表示在该位置上发现了一个错误，指出字符串字面量"未闭合。
- 第 6 行第 17 列的[Error: unclosed string]:"unclose\"表示在该位置上发现了一个错误，指出字符串字面量"unclose"未闭合。
- 第 8 行第 9 列和第 10 列的[Error: unexpected character]:\$表示在该位置上发现了一个错误，指出意外的字符"\$"。
- 第 9 行第 13 列的[Error: multi-character character constant or unclosed character

`constant]:'@` 表示在该位置上发现了一个错误，指出字符常量`'@`是一个未闭合的字符常量。

5 实验总结

本次实验中我手工编写了一个词法分析程序，使我对词法分析的流程更加清楚，对相关知识点的掌握更加牢固。

为了实现 C 语言的词法分析，我首先参考了 C99 的 ISO 标准，仔细阅读了标准中对各种词法元素的定义，包括关键字、标识符、常量、字符串字面量、运算符等。然后根据标准中给出的语法和课上所学的自动机，我设计了词法分析所需的自动机。

在设计词法分析程序时，我参考教材和课上学习的内容，计了两个关键函数 eatChar() 和 peekChar()，通过合理利用预读取操作，可以简化词法分析的流程。eatChar() 函数从输入串中取得下一个字符，peekChar() 函数预读取下一个字符但不消耗该字符。通过预读取, 可以提前判断下一个输入的类型，帮助词法分析。

在具体实现时，我遇到了一些困难。在第一次编写的程序中，出现了许多疏漏的情况，无法正确解析输入的 C 代码。为了解决这些问题，我多次细致地阅读语法定义，设计不同的样例进行测试，反复调试程序。在这过程中，我找到并修复了程序中的许多 bug。

另外，我没有直接使用教材中的缓冲区方式，而是利用了 C++ 标准库中的 string 类来维护输入缓冲区。使用 string 类可以大大简化字符串处理，也使程序更易读和维护。

通过这个实验，我对词法分析的流程和实现有了更深入的理解，也提高了自己的编程能力，尤其是利用自动机识别字符串的能力。在阅读语法标准方面，我的英文文献阅读能力也得到了提高。总而言之，这个实验使我收获颇丰，对以后课程的学习非常有帮助。