第二章 使用 lex[1]

本章内容：

- 正则表达式
- 单词计数程序
- 分析命令行
- C 源代码分析程序
- 小结
- 练习

上一章我们示范了如何使用 lex 和 yacc。现在我们介绍如何单独使用 lex，包括一些说明 lex 是一个好工具的应用示例，本章不想解释 lex 的每一个细节，我们将在第六章“ lex 规范参考 ”中概述。

lex 是构建词法分析程序的工具。词法分析程序把随机输入流标记化（tokenize），即，将它拆分成词法标记。然后，可以进一步处理这种被标记化的输出，通常是由 yacc 来处理的，或者它就成为“ 最终产品 ”。在第一章中，我们介绍了如何将它作为英语语法中的中间步骤。现在，要进一步查看 lex 规范的细节以及如何使用 lex 规范；我们的示例中将 lex 用做最后的处理步骤，而小是作为将信息传递给基于 yacc 的语法分析程序的中间步骤。

当编写 lex 规范时，可以创建 lex 匹配输入所用的一套模式。每次匹配一个模式时，lex 程序就调用你提供的 C 代码来处理被匹配的文本。采用这种方式，lex 程序将输入拆分成称为标记的字符串。lex 本身不产生可执行程序。相反，它把 lex 规范转化成包含 C 例程 **yylex()**的文件。程序调用 **yylex()**来运行词法分析程序。

使用普通的 C 编译程序，编译 lex 产生的文件以及想要的作任何其他文件和库（注意 lex 和 C 编译程序甚至不必在同一台计算机上运行。作者们经常将 UNIX lex 的 C 代码放在其他计算机上，在这些计算机上 lex 不可用，但 C 代码可用。）

正则表达式

在描述 lex 规范的结构以前，需要先描述 lex 使用的正则表达式。正则表达式被广泛应用于 UNIX 环境，并且 lex 可以使用丰富的正则表达式语言。

正则表达式是一种使用“ 元（meta）”语言的模式描述。元语言用于描述特定模式。这种元语言中使用的字符是 UNIX 和 MS-DOS 中使用的标准的 ASCII 字符集（有时会导致混淆）。形成正则表达式的字符是：

[1] Jobn R.Levine, Tony Mason, Doug Brown 著，杨作梅，张旭东等译). 整理: qy8087@gmail.com

. 匹配除换行符（“ \n ”）以外的任何单个字符。

* 匹配前面表达式的零个或多个拷贝。

[] 匹配括号中的任意字符的字符类。如果第一个字符是音调符号（"^"），它的含义将改变为匹配除括号中的字符以外的任意字符。方括号中的短划线指示一个字符范围，例如“ [0-9] ”和“ [0123456789] ”的含义相同。“ - ”或“] ”作为“ [”后的第一个字符时照字面意义解释。即在字符类中包括短划线或方括号。当处理非英文字母表时。POSIX 引进了特殊的方括号结构。参见附录八“ POSIX lex 和 yacc ”可以得到更详细的资料。除了识别以“ \ ”开始的 C 转义序列以外，其他的元字符在方括号中没有特殊的含义。

^ 作为正则表达式的第一个字符匹配行的开头。也用于方括号中的否定。

$ 作为正则表达式的最后一个字符匹配行的结尾。

{} 当括号中包含一个或 2 个数字时，指示前面的模式被允许匹配多少次，例如：A{1,3}表示匹配字母 A 一次到三次。如果包含名称，则认为是以该名称替换。

\ 用于转义元字符，并且作为通常 C 转义序列的一部分，例如，“ \n ”是换行符，而“ * ”是星号。

+ 匹配前面的正则表达式的一次或多次出现。例如：[0-9]+ 匹配“ l ”、“ 111 ”或“ 123456 ”、但不匹配空字符串（如果加号换成星号，它还可以匹配空字符串）。

? 匹配前面的正则表达式的零次或一次出现。例如：-?[0-9]+ 匹配包括一个可选的前导减号的有符号的数字。

| 匹配前面的正则表达式或随后的正则表达式。例如：cow|pig|heep 匹配三个单词中的任意一个。

“...” 引号中的每个字符解释为字面意义——除 C 转义序列外元字符会失去它们特殊的含义。

/ 只有在后面跟有指定的正则表达式时才匹配前面的正则表达式。例如：0/1 匹配字符串“ 01 ”中的“ 0 ”，但是不匹配字符串“ 0 ”或“ 02 ”中的任何字符。由跟在斜线后的模式所匹配的内容不被“ 使用 ”，并且会被转变成随后的标记。每个模式只允许一个斜线。

() 将一系列正则表达式组成一个新的正则表达式。例如：(01) 表示字符序列 01。当用*、+和 | 构建复杂的模式时，圆括号很有用。

要注意这些操作符中有些只操作单个字符（例如[]），而另一些则操作正则表达

式。通常，复杂的正则表达式都是由简单的正则表达式构建而成的。

正则表达式的示例

下面准备了一些示例。首先是适用于“ 数字 ”的正则表达式：

```
[0-9]
```

可以用这种形式构建一个整数的正则表达式：

```
[0-9]+
```

上述情况至少需要一个数字。下面的形式可以允许没有数字：

```
[0-9]*
```

添加一个可选的一元减号：

```
-?[0-9]+
```

然后，将它扩展为允许小数。首先，指定一个小数（目前，我们坚持最后一个字符总为数字）：

```
[0-9]*\.[0-9]+
```

注意句点前面的“ \ ”使句点转义为文字含义上的句点，而不是一个通配符。这种模式匹配“ 0.0 ”、“ 4.5 ”或“ .31415 ”，但是它不匹配“ 0 ”或“ 2 ”。下面想合并匹配它们的定义，不考虑一元减号，可以使用如下形式：

```
([0-9]+)|([0-9]*\.[0-9]+)
```

使用分组符号“ () ”确定由“ | ”分开的正则表达式。现在添加一元减号：

```
-?(([0-9]+)|([0-9]*\.[0-9]+))
```

通过指定浮点风格的指数，可以进一步扩展这种形式。首先，为指数编写一个正则表达式：

```
[eE][-+]?[0-9]+
```

这种形式匹配大写或小写字母 E，然后是一个可选的加号或减号，接着是一个数字串。例如，这种形式匹配“ e12 ”或“ E-3 ”。然后，使用这种表达式构建最终的表达式，指定一个实数的最终表达式为：

```
-?(([0-9]+)|([0-9]*\.[0-9]+)([eE][-+]?[0-9]+)?)
```

表达式使指数部分可选。编写一个使用这种表达式的真正的词法分析程序。毫无疑问，程序要检查输入，并且每当它匹配符合正则表达式要求的数字时它都会告诉我们。

程序如例 2-1 所示。

例 2-1：小数的 lex 规范

```
%%
[\n\t ]   ;
-?(([0-9]+)|([0-9]*\.[0-9]+)([eE][-+]?[0-9]+)?)
   { printf("number\n"); }
.  ECHO;
%%
main()
{
   yylex();
}
```

词法分析程序忽略空白并向输出回送它认为不是数字的一部分的任意字符。例如，下面是具有一些接近于有效数字的结果：

```
.65ea12
number
eanumber
```

我们鼓励你演示这个示例和所有的示例，直到你能理解它们是如何工作的为止。例如，试着改变这个识别一元加号和一元减号的表达式。

还有一个用于脚本和简单配置文件的常见正则表达式，它匹配以升音符号“ # ”开始的表达式[1]。我们可以如下所示构建这样的正则表达式：

```
#.*
```

这里“ . ”匹配除换行符以外的任意字符，而“ * ”匹配前面表达式的零个或多个字符。这个表达式匹配注释行上换行前的所有字符，换行表示行的结束。

最后，下面的正则表达式匹配被引用的字符串：

```
\"[^"\n]*["\n]
```

似乎使用下面的比较简单的表达式就足够了：

[1]也称为散列符、磅字符等等。

```
\".*\"
```

可惜的是，如果同一个输入行上有两个被引用的字符串，就会导致 lex 匹配错误。例如：

```
“how” to “do”
```

只匹配一个模式，因为“ * ”匹配尽可能多的字符串。了解这个问题后，我们可以尝试下面的表达式：

```
\"[^"]*\"
```

因为表达式“ [^”]* ”匹配除一个引号之外的任意字符（包括“ \n”），所以如果尾部的引号没有出现，这个正则表达式就会导致 lex 内部输入缓冲区溢出。因此如果用户错误地遗漏一个引号，这个模式就可能扫描整个文件寻找另一个引号。因为标记存储在固定大小的缓冲区中[1]，迟早总的阅读量将超过缓冲区，而词法分析程序将崩溃。例如：

```
“How”,she said,”is it that I cannot find it
```

继续匹配第二个被引用的字符串直到它发现另一个引号。这可能是后面的成百上千个字符。所以我们需要添加一条新规则，即引用的字符串不能超过一行而且以前面说明的复杂的（但是比较安全）正则表达式结束。lex 以一种不同的方式来处理较长的字符串。参见第六章“ lex 规范参考 ”的“ yymore ”节。

单词计数程序

下面来看一下 lex 规范的实际结构。使用基本的单词计数程序(类似于 UNIX 程序 *wc*)。

lex 规范由二部分组成：定义段、规则段和用户子例程段。第一部分(定义段)处理 lex 用在词法分析程序中的选项，并且一般建立词法分析程序运行的执行环境。

单词计数示例的定义段如下：

```
%{
unsigned charCount = 0, wordCount = 0, lineCount = 0;
%}

word   [^ \t\n]+
eol    \n
```

[1]各版本的缓冲区的大小是不同的，有时只有 100 个字节，而有时达到 8K 个字节。要得到更多的详细资料，参见第六章的“ yytext ”节。

由“ %{ ”和“ % ”括住的部分是 C 代码，它们将被逐字地拷贝到词法分析程序中。这些 C 代码一开始即被放入输出代码中，所以这里包含的定义部分可以由规则段中的代码引用。在我们的示例中，这个代码块声明了程序中使用的三个变量，它们用来跟踪遇到的字符．单词和行的数目。

最后的两行是定义。lex 提供了一种简单的替换机制，从而使定义长的或复杂的模式变得很容易。我们在这里添加两个定义。第一个定义提供了单词描述：除了空格、制表符和换行符以外的字符的非空组合。第二个定义描述行结束字符，即换行。在文件的第二部分(规则段)使用这些定义。

规则段包含指定词法分析程序的模式和动作。下面是示例中的单词计数的规则段：

```
%%
{word}    { wordCount++; charCount += yyleng; }
{eol}     { charCount++; lineCount++; }
.         charCount++
```

规则段以“ %% ”开始。在模式中，lex 用 *substitution*(即定义段中的实际的正则表达式)代替大括号{}中的名字。在词法分析程序识别了完整的单词之后，我们的示例增加单词和字符的数目。

大括号中封闭的多个语句组成的动作生成一个 C 语言复合语句。lex 的多数版本将模式后的所有语句当做一个动作，而另一些版本只读取行的第一条语句并且默默地忽略其他的语句。如果动作包含多条语句或多个行，为了安全起见也为了使代码更加清晰，通常使用大括号。

值得重复的是，lex 总是尝试匹配可能最长的字符串。因此，词法分析程序将把字符串“ well-being ”作为一个单词。

示例中也使用 lex 的内部变最 **yyleng**，它包含词法分析程序识别的字符串长度。如果匹配了 well-being，**yyleng** 就为 10。

当词法分析程序识别一个换行时，它就增加字符数和行数。同样，如果它识别任意其他字符，它就增加字符数。对于这个词法分析程序，它识别的唯一的“ 其他字符 ”是空格或制表位 ;其他的字符匹配第一个正则表达式并且被当做一个单词。

词法分析程序总是尝试匹配可能最长的字符串，但是当存在两个同样长度的字符串时，词法分析程序使用 lex 规范中的早期规则。因此，单词“ I ”由{word} 规则匹配，而不是由“ . ”规则匹配。理解并使用这一原则会使词法分析程序更加清晰和安全。

lex 规范的第三部分和最后部分是用户子例程段。再说一次，它通过“ %% ”和前面的段分开。用户子例程段包含任何有效的 C 代码。它被逐字拷贝到生成的

词法分析程序中。一般说来，这 1 部分包含支持例程。对于这个示例，我们的“ 支持 ” 代码是主程序：

```
main()
{
   yylex();
   printf(“%d %d %d\n”, lineCount, wordCount, charCount);
}
```

首先，它调用词法分析程序的入口点 **yylex()**，然后调用 **printf()**打印这次运行的结果。要注意，我们的示例没有做任何花哨的操作：它既不接受命令行参数，也不打开任何文件，只是使用 lex 默认地读取标准输入。我们的大部分示例程序都假设你知道如何编写做这些事情的 C 代码程序。然而，lex 重新连接输入流的方法也值得一看。如例 2-2 所示。

例 2-2：单词计数程序的用户子例程 ch2-02.l

```
main(argc,argv)
int argc;
char **argv;
{
   if (argc > 1) {
      FILE *file;

      file = fopen(argv[1], "r");
      if (!file) {
         fprintf(stderr,"could not open %s\n",argv[1]);
         exit(1);
      }
      yyin = file;
   }
   yylex();
   printf("%d %d %d\n",charCount, wordCount, lineCount);
   return 0;
}
```

这个示例假设调用程序的第二个参数是要处理的文件[1]。**lex** 词法分析程序从标准 I/O 文件 **yyin** 中读取输入，所以当需要时，只需要改变 **yyin**。**yyin** 的默认值是 **stdin**，因为默认输入源是标准输入。

我们以 *ch2-02.l* 作为文件名存储了这个示例，因为它是第二章的第二个例子，而且传统上 lex 源文件以 *.l* 结尾。运行它，获得如下结果：

[1]传统上，第一个名字是程序名，但这与你使用的系统环境有关，因此你必须根据你的系统环境调整示例以得到正确的结果。

```
% ch2-02 ch2-02.l
467 72 30
```

我们的单词计数示例和标准 UNJX 单词计数程序之间的显著区别是 :我们的单词计数示例只处理单个文件。可以使用 lex 的文件尾处理程序来调整它。

当 **yylex()**到达输入文件的尾端时，它调用 **yywrap()**，该函数返回数值 0 或 1。如果值为 1，那么程序完成而且没有输入。换句话说，如果值为 0，那么词法分析程序假设 **yywrap()**已经打开了它要读取的另一个文件，而且继续读取 **yyin**。默认的 **yywrap()**总是返回 1。通过提供自己的 **yywrap()**版本，可以使程序读取命令行上命名的所有文件，一次读取一个。

处理多个文件要求对代码做很多更改。例 2-3 展示了最终的完整的单词计数程序。

例 2-3：多文件的单词计数程序

```
%{
/*
 * ch2-03.l
 *
 * 多文件的单词计数程序示例
 *
 */

unsigned long charCount = 0, wordCount = 0, lineCount = 0;

#undef yywrap    /* 默认情况下有时是一个宏 */

%}

word [^ \t\n]+
eol  \n
%%
{word} { wordCount++; charCount += yyleng; }
{eol}  { charCount++; lineCount++; }
.   charCount++;
%%

char **fileList;
unsigned currentFile = 0;
unsigned nFiles;
unsigned long totalCC = 0;
unsigned long totalWC = 0;
unsigned long totalLC = 0;
```

```
main(argc,argv)
int argc;
char **argv;
{
       FILE *file;

       fileList = argv+1;
       nFiles = argc-1;

       if (argc == 2) {
       /*
        * 因为不需要打印摘要行，所以处理单个文件的情况与
        * 处理多个文件的情况不同
        */
       currentFile = 1;
       file = fopen(argv[1], "r");
       if (!file) {
          fprintf(stderr,"could not open %s\n",argv[1]);
          exit(1);
       }
       yyin = file;
       }
       if (argc > 2)
       yywrap(); /* 打开第一个文件 */

   yylex();
       /*
        * 处理零个或一个文件与处理多个文件的又一不同之处
        */
       if (argc > 2) {
       printf("%8lu %8lu %8lu %s\n", lineCount, wordCount,
          charCount, fileList[currentFile-1]);
       totalCC += charCount;
       totalWC += wordCount;
       totalLC += lineCount;
       printf("%8lu %8lu %8lu total\n",totalLC, totalWC, totalCC);
   } else
       printf("%8lu %8lu %8lu\n",lineCount, wordCount, charCount);

   return 0;
}

/*
 * 词法分析程序调用 yywrap 处理 EOF。(比如，在本例中我们连接到一个新文件。)
```

```
 */

yywrap()
{
   FILE *file;

   if ((currentFile != 0) && (nFiles > 1) && (currentFile < nFiles))
   {
      /*
       * 打印出前一个文件的统计信息
       */
      printf("%8lu %8lu %8lu %s\n", lineCount, wordCount,
         charCount, fileList[currentFile-1]);
      totalCC += charCount;
      totalWC += wordCount;
      totalLC += lineCount;
      charCount = wordCount = lineCount = 0;
      fclose(yyin);/* 处理完这个文件 */
   }

   while (fileList[currentFile] != (char *)0) {
      file = fopen(fileList[currentFile++], "r");
      if (file != NULL) {
         yyin = file;
         break;
      }
      fprintf(stderr,
         "could not open %s\n",
         fileList[currentFile-1]);
   }
   return (file ? 0 : 1); /* 0 表示还有更多的输入 */
}
```

示例使用 **yywrap()**执行连续的处理。还有一些其他的方式，但是这是最简单和最方便的。每次词法分析程序调用 **yywrap()**时，都尝试从命令行中打开下一个文件名并将打开的文件赋给 **yyin**，如果存在另一个文件就返回 0，如果没有就返回 1。

示例报告独立文件的大小和一整套文件最后的累积量。如果只有一个文件，那么指定文件的数目只报告一次。

运行 lex (*ch2-03.l*)文件上的最终的单词计数程序，然后运行生成的 C 文件(*ch2-03.c*)的单词计数程序。

```
% ch2-03.pgm ch2-03.l
   107    337    2200
% ch2-03.pgm ch2-03.l ch2-03.c
   107    337    2220   ch2-03.l
   405    1382   9356   ch2-03.c
   512    1719   11576  total
```

不同系统的运行结果也不同，因为 lex 的不同版本会产生不同的 C 代码。我们不能投入太多的时间来美化输出，它只是读者的一个练习。

分析命令行

现在，我们看一下采用 lex 分析命令输入的另一个示例。通常，lex 程序会读取文件，使用预定义宏 **input()**从输人中得到下个字符，使用 **unput()**将一个字符放回逻辑输入流中。词法分析程序有时需要使用 **unput()**在输入流中提前取字符。例如，词法分析程序不能确定它已经找到了单词尾，除非它看见了单词尾端的标点符号，但是因为标点符号不是单词的一部分，所以它必须将标点符号放回输入流作为下一个标记。

为了扫描命令行而不是一个文件，我们必须重新编写 **input()**和 **unput()**。这里使用的实现只在 AT&T lex 中工作，因为其他版本不让你重新定义这两个程序。（例如，flex 直接读取输入缓冲区而且从不使用 **input ()**。如果使用另一个版本的 lex，参见第六章的“ 从字符串中输入 ”一节中的介绍来了解如何完成同样的事情。采用调用程序的命令行参数，并识别 3 种截然不同的参数类：**help**、**verbose** 和 **filename**。例 2-4 创建读取标准输入的词法分析程序，几乎和前面的单词计数程序示例一样。

例 2 -4：分析命令行输入的 lex 规范 ch2-04.l

```
%{
unsigned verbose;
char *progName;
%}

%%

-h     |
"-?"   |
-help  { printf("usage is: %s [-help | -h | -? ] [-verbose | -v] "
           "[(-file| -f) filename]\n", progName);
       }
-v     |
-verbose { printf("verbose mode is on\n"); verbose = 1; }
```

```
%%

main(argc, argv)
int argc;
char **argv;
{
   progName = *argv;
   yylex();
}
```

定义段包括文字代码块。变量 **verbose** 和 **progName** 是用在规则段的变量。

在规则段，第一条规则识别关键字*-help* 及其简写形式*-h* 和*-?*。注意规则后面的动作只是打印一个用法字符串[1]。第二条规则识别关键字*-verbose* 及其简写形式*-v*。在这种情况下，设置全局变量 **verbose**（前面定义的）的值为 1。

在用户子例程段，**main()**例程保存程序名，这个程序名用在帮助命令的用法字符串中，然后调用 **yylex()**。

当词法分析程序仍然读取标准输入而不是命令行时，这个示例不分析命令行的参数。给例 2-5 添加代码取代标准的 **input()**和 **unput()**程序。（这个例子是特定于 AT&T lex 的。参见附录五中有关 flex 的这方面的相关内容。）

例 2-5：分析命令行的 lex 规范 ch2-05.l

```
%{
#undef input
#undef unput
int input(void);
void unput(int ch);
unsigned verbose;
char *progName;
%}

%%

-h     |
"-?"   |
-help  { printf("usage is: %s [-help | -h | -? ] [-verbose | -v]"
          " [(-file| -f) filename]\n", progName);
       }
-v     |
-verbose { printf("verbose mode is on\n"); verbose = 1; }
```

[1]因为字符串不适合放入一行，所以在编译时使用将这个字符串拆分成两个连续的字符串的 ANSI C 技术。如果有早于 ANSI 的 C 编译程序，则必须将两个字符串粘贴在一起。

```
%%
char **targv;/* remembers arguments */
char **arglim;   /* end of arguments */

main(int argc, char **argv)
{
    progName = *argv;
    targv = argv+1;
    arglim = argv+argc;
    yylex();
}

static unsigned offset = 0;

int
input(void)
{
    char c;

    if (targv >= arglim)
        return(0);   /* EOF */
    /* end of argument, move to the next */
    if ((c = targv[0][offset++]) != '\0')
        return(c);
    targv++;
    offset = 0;
    return(' ');
}

/* simple unput only backs up, doesn't allow you to */
/* put back different text */
void
unput(int ch)
{
    /* AT&T lex sometimes puts back the EOF ! */
    if(ch == 0)
        return;   /* ignore, can't put back EOF */
    if (offset) {/* back up in current arg */
        offset--;
        return;
    }
    targv--;  /* back to previous arg */
    offset = strlen(*targv);
```

```
}
```

在定义部分，因为在 AT&T 中 lex 默认将 **input** 和 **unput** 定义为宏，所以我们将它们解除定义（**#under**），然后，重新定义为 C 函数。

在这个示例中不改变规则段。相反，大部分改动发生在用户子例程段。在新的部分，添加了三个变量：**targv**，跟踪当前的参数；**arglim**，标记参数的结束；**offset**，跟踪当前参数中的位置。这些变量设置在 **main()**中。用于指向来自命令行的参数向量，

input()程序处理来自词法分析程序的获取字符的调用。当前的参数用尽时，它就移到下一个参数（如果有一个的话）并继续扫描。如果没有参数，就把它作为词法分析程序的文件尾条件并返回一个零字节。

unput()程序处理来自词法分析程序的将字符“ 推回 ”输入流的调用。通过颠倒指针的方向来完成这项工作，在字符串中反向移动。在这种情况下，首先假设推回的字符与位于那儿的字符相同，除非动作代码明确地推回其他的东西，否则这种情况总是真的。一般情况下，动作程序可以推回它想推回的任何东西，而目．**unput()**的专用版本必须能处理这种情况。

结果示例仍然回送它不识别的输入，并为它所理解的输入打印出两条消息。例如下面显示了运行的示例：

```
% ch2-05 –verbose foo
verbose mode is on
 foo %
```

现在输入来自命令行，而且不识别的输入被回送。词法分析程序不识别的任何文本在默认的规则处都会失败，这些文本将被回送到输出。

起始状态

最后，添加一个*-file* 开关并识别文件名。使用起始状态完成这项工作，在词法分析程序中它是一种捕获上下文敏感信息的方法。用起始状态标记规则只是告诉词法分析程序在起始状态有效时识别规则。在这种情况下，为了识别*-file* 参数后的文件名，使用起始状态标明寻找文件名的时间，如例 2-6 所示。

例 2-6：具有文件名的 lex 命令扫描程序 ch2-06.l

```
%{
#undef input
#undef unput
unsigned verbose;
unsigned fname;
char *progName;
```

```
%}

%s FNAME

%%
[ ]+        /* 忽略空白 */ ;
<FNAME>[ ]+  /* 忽略空白 */ ;

-h     |
"-?"   |
-help     { printf("usage is: %s [-help | -h | -? ] [-verbose | -v]"
          " [(-file| -f) filename]\n", progName);
        }
-v     |
-verbose  { printf("verbose mode is on\n"); verbose = 1; }


-f     |
-file     { BEGIN FNAME; fname = 1; }

<FNAME>[^ ]+ { printf("use file %s\n", yytext); BEGIN 0; fname = 2;}

[^ ]+  ECHO;
%%
char **targv;/* 记录参数 */
char **arglim;   /* 参数结束 */

main(int argc, char **argv)
{
   progName = *argv;
   targv = argv+1;
   arglim = argv+argc;
   yylex();
   if(fname < 2)
      printf("No filename given\n");
}
```

…input()和 unput()函数与 2-5 相同

在定义部分，添加行 “ %s FNAME"，它在词法分析程序中创建新的起始状态。在规则部分，添加以 “ <FNAME> ” 开始的规则。这些规则只在词法分析程序处于 FNAME 状态时被识别。没有明确状态的任何规则无论当前是什么状态都会进行匹配。*-file* 参数切换为 FNAME 状态，激活匹配文件名的模式。一旦它匹配了文件名，就切换回正则状态。

规则段的动作中的代码改变当前状态。用 BEGIN 语句输入一条新状态。例如，为了改变为 FNAME 状态，可以使用语句“ BEGIN FNAME; ”；要改回到默认状态，使用“ BEGIN 0 "（默认地，状态零也称为 INITIAL）。

除了改变 lex 状态以外，还增加了独立变量 fname，所以如果参数丢失，示例程序可以识别出来。要注意，如果 fname 的值没有变成 2，主程序就打印一条错误消息。

在这个示例中，其他一些改变只处理文件名参数。这里的 **input()**在每个命令行的参数之后都返回一个空白。规则忽略空白，然而如果没有空白，词法分析程序中*-f ile* 参数和*-file* 参数的表现是一样的。

我们提到过，没有明确起始状态的规则无论活跃的起始状态是什么都会进行匹配（如例 2-7 所示）。

例 2-7：初始状态的例子 ch2-07.l

```
%s MAGIC

%%
<MAGIC>.+ { BEGIN 0; printf("Magic:"); ECHO; }
magic     BEGIN MAGIC;
.+     ECHO;
%%

main()
{
   yylex();
}
```

当看到关键字“ magic ”时切换到 MAGIC 状态；否则，只回送输入。如果处于 MAGIC 状态，就在下一个被回送的标记前插入字符串“ Magic: ”。创建一个具有三个单词的输入文件：magic、two 和 three，并在整个词法分析程序中运行它。

```
% ch2 07 < magic.input
Magic：two
three
```

现在，我们稍微改动一下示例，使具有起始状态的规则跟在一个没有起始状态的规则之后，如示例 2-8 所示。

例 2-8：打破起始状态的示例 ch2-08.l

```
%{
   /* 这个示例故意不工作！ */
%}
%s MAGIC
```

```
%%
magic     BEGIN MAGIC;
.+        ECHO;
<MAGIC>.+ { BEGIN 0; printf("Magic:"); ECHO; }
%%
main()
{
   yylex();
}
```

虽然有同样的输人，但是却得到非常不同的结果：

```
% ch2-08 < magic.input
two
three
```

可以将没有起始状态的规则隐式地认为具有一个“ 通配符 ”起始状态，它们匹配所有的起始状态。这常常是错误的根源。flex 和 lex 的其他新版本都有“ 惟一的起始状态 ”，可以解决通配符问题。参见第七章的“ 起始状态 ”一节可以得到更多的详细资料。

C 源代码分析程序

最后的示例检查 C 源文件并计算所看到的不同类型的行的数目，这些行有些包含代码，有些只包含注释或者是空白。因为单个行可以既包含注释又包含代码，所以这就是问题所在，这样我们就必须决定如何计数这样的行。

首先，描述空白行。将除换行符以外什么也没有的行当做空白行。同样，具有空白区或制表符但没有其他东西的行也是空白行。正则表达式这样描述空白行：

```
^[ \t]*\n
```

“ ^ ”操作符指示这种模式必须位于行的开始。同样，要求整个行只有一个换行符“ \n ”在行尾。

对代码行或注释行的描述是：任何不完全是空白的行。

```
^[ \t]*\n
\n  /* 空白行被前一条规则匹配 */
.   /* 其他东西 */
```

使用新的规则“ \n ”计算所看到的不是都为空白的行的数目。使用第二个新规则

抛弃不感兴趣的字符。下面增加的是用于描述注释的规则：

```
^[ \t]*"/*".*"*/"[ \t]*\n
```

它描述了单个行上的单个的、自包含的注释，即在“ /* ”和“ */ ”之间有可选的文本．因为“ * ”和“ / ”是两个特殊的模式字符，它们以字面意义出现时要引起来。实际上，这种模式不很正确，因为有时为如下形式：

```
/* 注释 */ /*注释
```

上述形式就不符合这个规律。注释可以跨越多行而且“ . ”操作符不包含“ \n ”字符。当然，如果允许用“ \n ”字符，那么一个长的注释可能会造成 lex 内部缓冲区溢出。相反，当看到注释的开头时，添加一个起始状态 COMMENT 并进入那个状态可以防止这种问题。当看到注释的结尾时，返回到默认的起始状态。不需要对单行的注释使用起始状态。下面是识别注释的开头的规则：

```
^[ \t]*"/*"
```

动作中有一个 BEGIN 语句可以切换到 COMMENT 状态。一个 COMMENT 单独出现在一行的开头对于句子来说是很重要的，不这样做会导致计数错误，例如：

```
int counter; /* 这是
   一个奇怪的注释 */
```

因为第一行不是单独的一行。我们需要把第一行作为代码行计算，而把第二行作为注释行计算。下面是实现这种想法的规则：

```
.+"/*".*"*/".*\n
.*"/*".*"*/".+\n
```

上述两个表达式描述的字符串集合有重叠，但是它们是不同的。下面的表达式符合第一个规则但不符合第二个：

```
int counter; /* 注释 */
```

因为第二个规则要求在注释后面接有文字。同样，下面的表达式符合第二条规则。但不符合第一条规则：

```
/* 注释 */ int counter;
```

它们都符合下面的表达式：

```
/* 注释 #1 */ int counter; /* 注释 #2 */
```

最后，我们需要完成检测注释的正则表达式。我们决定采用起始状态，所以当处于 COMMENT 状态时，仅仅需要寻找换行字符：

```
<COMMENT>\n
```

并对它们计数。当检测到“ 注释尾 ”字符时，如果注释尾部没有其他东西，就把它当做注释行计算，否则就继续处理：

```
<COMMENT>”*/”[ \t]*\n
<COMMENT>”*/”
```

第一行算作注释行，第二行继续进行处理。当我们把这些规则放在一起时，需要进行一些粘接工作，因为需要覆盖默认起始状态和 COMMENT 起始状态中的一些情况。例 2-9 展示了最终的正则表达式的代码清单以及与它们有关的动作。

例 2-9:C 源代码分析程序 ch2-09.l

```
%{
int comments, code, whiteSpace;
%}

%s COMMENT

%%
^[ \t]*"/*"   { BEGIN COMMENT; /* 进入注释处理状态 */ }
^[ \t]*"/*".*"*/"[ \t]*\n {
        comments++; /* 自包含注释 */
        }

<COMMENT>"*/"[ \t]*\n   { BEGIN 0;  comments++;}
<COMMENT>"*/"           { BEGIN 0; }
<COMMENT>\n             { comments++; }
<COMMENT>.\n            { comments++; }

^[ \t]*\n               { whiteSpace++; }

.+"/*".*"*/".*\n        { code++; }
.*"/*".*"*/".+\n        { code++; }
.+"/*".*\n              { code++; BEGIN COMMENT; }
.\n                     { code++; }

.          ; /* 忽略其它 */
%%
main()
{
   yylex();
```

```
    printf("code: %d, comments %d, white space %d\n",
        code, comments, whiteSpace);
}
```

添加规则“ <COMMENT>\n ”和“ <COMMENT>.\n ”来处理注释中空行情况，以及注释中的文本。强制它们匹配行结束字符意味着如下形式不会被当成两行注释：

```
/* 这是注释的开始
和结束*/ int counter;
```

而是把它们当成一行注释和一行代码。

小结

本章我们概述了使用 lex 的基本原理。通常对于编写简单的应用程序（例如本章开发的单词计数程序和代码行计数实用程序），lex 就足够用了。lex 用一定数量的特殊字符来描述正则表达式。当正则表达式匹配输入字符串时，执行相应的动作，就是你指定的一段 C 代码。首先，它匹配最长的表达式，然后，如果两个表达式具有相同的长度，则首先匹配在 lex 应用程序中出现最早的那一个。灵活地使用起始状态，当激活特定规则时可以进一步改进，正如我们为代码行计数程序所做的那样。

我们还讨论了一些具有特殊目的的例程，它们被 lex 产生的状态机所使用。例如 yywrap()，用于处理文件结束条件并按顺序处理多个文件。使用这个例程可以使我们的单词计数示例检查多个文件。

本章主要将 lex 单独用做一种处理语言。后面的文章主要集中在如何将 lex 和 yacc 集成以构建其他类型的工具。但是 lex 自己就能处理许多其他方面的冗长任务，而不需要全面的 yacc 语法分析程序。

练习

1. 使单词计数程序智能化地区分单词的含义，区别字母串（也许还有连字号和省略符号）这样的真正的单词和标点符号块。这个程序不要超过 10 行。

2. 改进 C 代码分析程序：计数括号、关键字等。尝试标识函数定义和描述，它们在所有大括号外面，其后跟着一个“ (”。

3. lex 真的和我们所说的一样快吗？将它与 *egrep*、*awk*、*sed* 或你所拥有的其他模式匹配程序进行比较。编写一个 lex 应用程序，这个程序可以寻找包含一些字符串的行并可以打印出这些行。（为了公平比较，确认要打印整行。）比

较它与其他程序花费在扫描一组文件上的时间，如果你有更多版本的 lex,它们有很明显的运行速度差别吗？