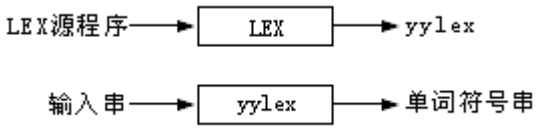
【附录 B】词法分析程序生成器 LEX 的使用方法

B.1 LEX 概述

程序设计语言从机器语言发展到今天的像 PASCAL、C 等这样的高级语言，使人们可以摆脱与机器有关的细节进行程序设计。但是用高级语言写程序时程序员必须在程序中详尽地告诉计算机系统怎样去解决某个问题，这在某种程度上说也是一件很复杂的工作。人们希望有新的语言--非常高级的语言，用这种语言，程序员仅仅需要告诉计算机系统要解决什么问题，计算机系统能自动地从这个问题的描述去寻求解决问题的途径，或者说将这个问题的描述自动地转换成用某种高级语言如 C?FORTRAN 表示的程序?这个程序就可以解决给定的问题。这种希望虽然还没有能够完全变成现实，但是在某些具体的问题领域里已经部分地实现了。这里要介绍的 LEX 和下一章要介绍的 YACC 就是在编译程序设计这个领域里的两种非常高级的语言，用它们可以很方便的描述词法分析器和语法分析器，并自动地产生出相应的高级语言(C)的程序。

图 B.1 LEX 示意图



LEX 是一个词法分析器(扫描器)的自动产生系统，它的示意图如图 B.1。

LEX 源程序是用一种面向问题的语言写成的?这个语言的核心是正规表达式(正规式)，用它描述输入串的词法结构。在这个语言中用户还可以描述当某一个词形被识别出来时要完成的动作，例如在高级语言的词法分析器中，当识 别出一个关键字时，它应该向语法分析器返回该关键字的内部编码。LEX 并不是一个完整的语言，它只是某种高级语言(称为 LEX 的宿主语言)的扩充，因此 LEX 没有为描述动作设计新的语言，而是借助其宿主语言来描述动作。我们只介绍 C 作为 LEX 的宿主语言时的使用方法，在 UNIX 系统中，FORTRAN 语言的一种改进形式 RATFOR 也可以做 LEX 的宿主语言。

LEX 自动地把表示输入串词法结构的正规式及相应的动作转换成一个宿主语言的程序，即词法分析程序，它有一个固定的名字 yylex，在这里 yylex 是一个 C 语言的程序。

yylex 将识别出输入串中的词形，并且在识别出某词形时完成指定的动作。

看一个简单的例子:写一个 LEX 源程序，将输入串中的小写字母转换成相应的大写字母。

程序如下:

%%

[a-z] Printf("%c"，yytext[0]+′A′-′a′);

上述程序中的第一行%%是一个分界符，表示识别规则的开始?第二行就是识别规则。左边是识别小写字母的正规式，右边就是识别出小写字母时采取的动作，将小写字母转换成相应的大写字母。

LEX 的工作原理是将源程序中的正规式转换成相应的确定有限自动机，而相应的动作则插入到 yylex 中适当的地方，控制流由该确定有限自动机的解释器掌握，对不同的源程序这个解释器是相同的?关于 LEX 工作原理的详细介绍请参阅本书第 4 章。

B.2 LEX 源程序的格式

LEX 源程序的一般格式是:

{辅助定义部分}

%%

{识别规则部分}

%%

{用户子程序部分}

其中用花括号括起来的各部分都不是必须有的。当没有"用户子程序部分"时，第二个%%也可以省去。第一个%%是必须的，因为它标志着识别规则部分的开始，最短的合法的 LEX 源程序是：

%%

它的作用是将输入串照原样抄到输出文件中。

识别规则部分是 LEX 源程序的核心。它是一张表，左边一列是正规式，右边一列是相应的动作。下面是一条典型的识别规则：

integer printf ("found keyword INT");

这条规则的意思是在输入串中寻找词形"integer"，每当与之匹配成功时，就打印出"found keyword INT"这句话。

注意在识别规则中，正规式与动作之间必须用空格分隔开。动作部分如果只是一个简单的 C 表达式，则可以写在正规式右边同一行中，如果动作需要占两行以上，则需用花括号括起来，否则会出错。上例也可以写成：

integer {printf("found keyword INT");}

下面先介绍识别规则部分的写法，再介绍其余部分。

B.3 LEX 用的正规式

一个正规式表示一个字符串的集合。正规式由正文字符与正规式运算符组成。正文字符组成基本的正规式，表示某一个符号串;正规式运算符则将基本的正规式组合成为复杂的正规式，表示字符串的集合。

例如

ab

仅表示字符串 ab，而

(ab)+

表示字符串的集合:

{ab，abab，ababab，…}。

LEX 中的正规式运算符有下列十六种:

"\[]八-。· +|()/邢{}%<>

上述运算符需要作为正文字符出现在正规式中时，必须借助于双引号"或反斜线\，具体用法是：

xyz"++"或 xyz\+\+

表示字符串 xyz++

为避免死记上述十多个运算符，建议在使用非数字或字母字符时都用双引号或反斜线。

要表示双引号本身可用\"，要表示反斜线用"\"或\\

前面说过，在识别规则中空格表示正规式的结束，因此要在正规式中引进空格必须借助双引号或反斜线，但出现在方括号[]之内的空格是例外。

几个特殊符号:

\n 是回车换行(newline)

\t 是 tab

\b 是退格(back space)

下面按照运算符的功能分别介绍上述正规式运算符。

1) 字符的集合

用方括号对可以表示字符的集合。正规式

[a b c]

与单个字符 a 或 b 或 c 匹配

在方括号中大多数的运算符都不起作用，只有\-和∧例外。

运算符-表示字符的范围，例如

[a-z0-9<>　-]

表示由所有小写字母，所有数字。尖括号及下划线组成的字符集合。

如果某字符集合中包括-在内，则必须把它写在第一个或最后一个位置上，如

[-+0-9]

与所有数字和正负号匹配。

在字符集合中，运算符∧必须写在第一个位置即紧接在左方括号之后，它的作用是求方括号中除∧之外的字符组成的字符集合相对于计算机的字符集的补集，例如

[∧abc]

与除去 a。b 和 c 以外的任何符号匹配。

运算符\在方括号中同样发挥解除运算符作用的功能。

2) 与任意字符匹配的正规式

运算符。形成的正规式与除回车换行符以外的任意字符匹配。

在 LEX 的正规式中，也可以用八进制数字与\一起表示字符，如

[\40-\176]

与 ASCII 字符集中所有在八进制 40(空格)到八进制 176(~)之间的可打印字符匹配。

3) 可有可无的表达式

运算符。指出正规式中可有可无的子式，例如：

ab。 c

与 ac 或 abc 匹配，即 b 是可有可无的。

4) 闭包运算

运算符　和+是 LEX 正规式中的闭包运算符，它们表示正规式中某子式的重复，例如：

a

表示由 0 个或多个 a 组成的字符串的集合，而

a+

表示由 1 个或多个 a 组成的字符串的集合，下面两个正规式是常用的：

[a-z]+

[A-Za-z][A-Za-z0-9]

第一个是所有由小写字母组成的字符串的集合，第二个是由字母开头的字母数字串组成的集合。

5) 选择和字符组

运算符|表示选择：

(ab|cd)

与 ab 或 cd 匹配。

运算符()表示一组字符，注意()与[]的区别。(ab)表示字符串 ab，而[ab]则表示单个字符 a 或 b

圆括号()用于表示复杂的正规式，例如：

(ab|cd+)。(ef)

与 abefef，efef，cdef，cdcd 匹配，但不与 abc，abcd 或 abcdef 匹配。

6) 上下文相关性

LEX 可以识别一定范围的上下文，因此可在一定程度上表示上下文相关性。

若某正规式的第一个字符是∧，则仅当该正规式出现在一行的开始处时才被匹配，一行的开始处是指整个输入串的开始或者紧接在一个回车换行之后，注意∧还有另一个作用即求补，∧的这两种用法不可能发生矛盾。

若某正规式的最后一个字符是$，则仅当该表达式出现在一行的结尾处时才被匹配，一行的结尾处是指该表达式之后紧接一个回车换行。

运算符/指出某正规式是否被匹配取决于它的后文，例如：

ab/cd

仅在 ab 之后紧接 cd 的情况下才与 ab 匹配。

$其实是/的一个特殊情形，例如下面两个正规式等价：

ab邓，ab/\n

某正规式是否被匹配，或者匹配后执行什么样的动作也可能取决于该表达式的前文，前文相关性的处理方法在后面专门讨论，将用到运算符<>。

7) 重复和辅助定义

当被{}括起来的是数字对时，{}表示重复;当它括起来的是一个名字时，则表示辅助定义的展开。例如

a{1，5}

表示集合{a，aa，aaa，aaaa，aaaaa}。

{digit}

则与预先定义的名叫 digit 的串匹配，并将有定义插入到它在正规式中出现的位置上，辅助定义在后面专门讨论。

最后，符号%的作用是作为 LEX 源程序的段间分隔符。

B.4 LEX 源程序中的动作

前面说过当 LEX 识别出一个词形时，要完成相应的动作。这一节叙述 LEX 为描述动作提供的帮助。

首先应指出，输入串中那些不与任何识别规则中的正规式匹配的字符串将被原样照抄到输出文件中去。因此如果用户不仅仅是希望照抄输出，就必须为每一个可能的词形提供识别规则，并在其中提供相应的动作。用 LEX 为工具写程序语言的词法分析器时尤其要注意。

最简单的一种动作是滤掉输入中的某些字符串，这种动作用 C 的空语句";"来实现。

例　滤掉输入串中所有空格。tab 和回车换行符，相应的识别规则如下：

[\t\n];

如果相邻的几条规则的动作都相同，则可以用|表示动作部分，它指出该规则的动作与下一条规则的动作相同。例如上例也可以写成：

""|

"\t"|

"\n";

注意\t 和\n 中的双引号可以去掉。

外部字符数组 yytext 的内容是当前被某规则匹配的字符串，例如正规式[a-z]+与所有由小写字母组成的字符串匹配，要想知道与它匹配的具体字符串是什么，可用下述规则：

[a-z]+printf("%s"，yytext);

动作 printf("%s"，yytext)就是将字符数组 yytext 的内容打印出来，这个动作用得很频繁，LEX 提供了一个宏 ECHO 来表示它，因此上述识别规则可以写成：

[a-z]+ECHO;

请注意，上面说过缺省的动作就是将输入串原样抄到输出文件中，那么上述规则起什么作用呢。这一点将在"规则的二义性"一节中解释。

有时有必要知道被匹配的字符串中的字符个数，外部变量 yyleng 就表示当前 yytext 中字符的个数。例如，要对输入串中单词的个数和字符的个数进行计数(单词假定是由大写或小写字母组成的字符串)，可用下述规则：

```
[a-zA-Z]+{words++;
        chars+=yyleng;
        }
```

注意被匹配的字符串的第一个字符和最后一个字符分别是

yytext[0]和 yytext[yyleng-1]

1) yymore()

当需下一次被匹配的字符串被添加在当前识别出的字符串后面，即不使下一次的输入替换 yytext 中已有的内容而是接在它的内容之后，必须在当前的动作中调用 yymore()

例　假设一个语言规定它的字符串括在两个双引号之间，如果某字符串中含有双引号，则在它前面加上反斜线\。用一个正规式来表达该字符串的定义很不容易，不如用下面较简明的正规式与 yymore()配合来识别：

```
\"[∧"]  {
        if (yytext[yyleng-1]=='\\'yy
        more();
        else
          …normal user processing
        }
```

当输入串为"abc\"def"时，上述规则首先与前 5 个字符"abc\匹配，然后调用 yymore()使余下部分"def 被添加在前一部分之后，注意作为字符串结尾标志的那个双引号由"normal user processing"部分负责处理

2) yyless(n)

如果当前匹配的字符串的末尾部分需要重新处理，那么可以调用 yyless(n)将这部分子串"退回"给输入串，下次再匹配处理。yyless(n)中的 n 是不退回的字符个数，即退回的字符个数是 yyleng-n。

例　在 C 语言中。=-a"具有二义性，假定要把它解释为"=-a"，同时给出信息，可用下面的识别规则：

```
=-[a-zA  Z]{
          printf("Operator (=-)
          ambiguous\n");
          yyless(yyleng-1);
          …action for =-…
          }
```

上面的规则先打印出一条说明出现二义性的信息，将运算符后面的字母退回给输入串，最后将运算符按"=-"处理。另外，如果希望把"=-a"解释为″=-a″，这只需要把负号与字母一起退回给输入串等候下次处理，用下面的规则即可：

```
=-[a-zA  Z]{
          printf("Operator (=-)
          ambiguous\n");
          yyless (yyleng-1);
          …action for =…
          }
```

3) yywrap()

当 LEX 处理到输入串的文件尾时，自动地调用 yywrap()，如果 yywrap()返回值是 1，那么 LEX 就认为对输入的处理完全结束，如果 yywrap()返回的值是 0，LEX 就认为有新的输入串等待处理。

LEX 自动提供一个 yywrap()，它总是返回 1，如果用户希望有一个返回 0 的 yywrap()，那么就可以在"用户子程序部分"自己写一个 yywrap()，它将取代 LEX 自动提供的那个 yywrap()，在用户自己写的 yywrap()中，用户还可以作其它的一些希望在输入文件结束处要作的动作，如打印表格。输出统计结果等，使用 yywrap()的例子在后面举出。

B.5 识别规则的二义性

有时 LEX 的程序中可能有多于一条规则与同一个字符串匹配，这就是规则的二义性，在这种情况下，LEX 有两个处理原则：

1) 能匹配最多字符的规则优先

2) 在能匹配相同数目的字符的规则中，先给出的规则优先

例　设有两规则按下面次序给出：

integerkeyword action…

[a-z]+identifier action…

如果输入是 integers，则它将被当成标识符处理，因为规则 integer 只能匹配 7 个字符，而[a-z]+能匹配 8 个字符；如果输入串是 integer，那么它将被当作关键字处理，因为两条规则都能与之匹配，但规则 integer 先给出。

B.6 LEX 源程序中的辅助定义部分

LEX 源程序的第一部分是辅助定义，到目前为止我们只涉及到怎样写第二部分，即识别规则部分的写法，现在来看第一部分的写法。

在 LEX 源程序中，用户为方便起见，需要一些辅助定义，如用一个名字代表一个复杂的正规式。辅助定义必须在第一个%%之前给出，并且必须从第一列开始写，辅助定义的语法是：

name　translation

例如用名字 IDENT 来代表标识符的正规式的辅助定义为

IDENT　[a-zA-Z][a-zA-Z0-9]

辅助定义在识别规则中的使用方式是用运算符{}将 name 括起来，LEX 自动地用 translation 去替换它，例如上述标识符的辅助定义的使用为：

{IDENT}　action for identifier

…

下面我们用辅助定义的手段来写一段识别 FORTRAN 语言中整数和实数的 LEX 源程序：

```
D[0-9]
E[DEde][-+]。{D}+
%%
{D}+printf("integer");
{D}+" ."{D}　({E})。|
{D}　" ."{D}+({E})。|
{D}+{E}printf("real");
```

请注意在辅助定义部分中可以使用前面的辅助定义。例如，定义 E 时使用了 D，但所用的辅助定义必须是事先已定义过的，不能出现循环定义。上面的规则只是说明辅助定义的用法，并不是识别 FORTRAN 中数的全部规则，因为它不能处理类似 35.EQ.I 这样的问题，即会把 35.EQ.I 中的 35
.E 当作实数，怎么解决这种问题请读者思考。

除了上面介绍的辅助定义之外，用户还需要在 LEX 源程序中使用变量，还需要有一些自己写的子程序。

前面已经见过两个常用的变量即 yytext 和 yylong，也介绍过几个 LEX 提供的子程序 yymore，yyless 和 yywrap，现在介绍用户如何自己定义变量和写子程序。

LEX 是把用户写的 LEX 源程序转换成一个 C 语言的程序 yylex，在转换过程中，LEX 是把用户自己的变量定义和子程序照抄到 yylex 中去，LEX 规定属于下面三种情况之一的内容就照抄过去：

1) 以一个空格或 tab 起头的行，若不是 LEX 的识别规则的一部分，则被照抄到 LEX 产生的程序中去。如果这样的行出现在第一个%%之前，它所含的定义就是全局的，即 LEX 产生的程序中的所有函数都可使用它。如果这样的行紧接在第一个%%之后但在所有识别规则之前，它们就是局部的，将被抄到涉及它的动作相应的代码中去。注意，这些行必须符合 C 语言的语法，并且必须出现在所有识别规则之前。

这一规定的一个附带的作用是使用户可以为 LEX 源程序或 LEX 产生的词法分析器提供注解，当然注解必须符合 C 语言文法。

2) 所有被界于两行%{和%}之间的行，无论出现在哪里也无论是什么内容都被照抄过去，要注意%{和%}必须分别单独占据一行。例如

```
%{
#defineENDOFFILE 0
#include "head.h"
int flag
%}
```

提供上面的措施主要因为在 C 语言中有一些行如上例中的宏定义或文件蕴含行必须从第一列开始写。

3) 出现在第二个%%之后的任何内容，不论其格式如何，均被照抄过去。

B.7 怎样在 UNIX 系统中使用 LEX

假定已经写好了一个 LEX 源程序。怎样在 UNIX 系统中从它得到一个词法分析器呢。LEX 自动地把 LEX 源程序转换成一个 C 语言的可运行的程序，

这个可运行的程序放在叫 lex.yy.c 的文件中，这个 C 语言程序再经过 C 编译，即可运行。

例　有一名叫 source 的 LEX 源程序，第一步用下面的命令将它转换成 lex.yy.c:

邝　$ lex source

($是 UNIX 的提示符)。lex.yy.c 再用下面的命令编译即得到可运行的目标代码 a.out:

邝　$ cc lex.yy.c -ll

上面的命令行中的-ll 是调用 LEX 的库，是必须使用的。

LEX 可以很方便地与 YACC 配合使用，这将在下面附录 C 中介绍。

B.8 LEX 源程序例子

这一节举两个例子看看 LEX 源程序的写法

1) 将输入串中所有能被 7 整除的整数加 3，其余部分照原样输出，先看下面的 LEX 源程序：

```
%%
  int k;
[0-9]+{
  scanf(-1，yytext，"%d"，&k);
  if(k%7==0)
    printf("%d"，k+3);
  else
    printf("%d"，k);
  }
```

上面的程序还有不足的地方，如对负整数，只是将其绝对值加上 3，而且像 X7，49　63 这样的项也做了修

改，现在只要对上面的源程序稍作修改就避免了这些问题。

```
%%
   int k;
-。[0-9]+{
            scanf(-1，yytext，"%d"，&k);
            printf("%d"，k%7==0。k+3;k);
            }
-。[0-9]+ECHO;
[A-Za-z][A-Za-z0-9]+ECHO;
```

2) 下一个例子统计输入串中各种不同长度的单词的个数，统计结果在数组 lengs 中，单词定义为由字母组成的串，源程序如下：

```
   int lengs[100];
%%
[a-z]+lengs[yyleng]++;
   ·    |
   \      n;
%%
yywrap()
{
int i;
printf("Length No.words  \。);
for (i=0;i<100;i++)
   if (lengs[i]>0)
      printf ("%5d%10d\n"，i，
         lengs[i]);
return(1);
}
```

在上面的源程序中，当 LEX 读入输入串时，它只统计而不输出，到输入串读入完毕后，才在调用 yywrap()时输出统计结果，为此用户自己提供了 yywrap()，注意 yywrap()的最后一个语句是返回值 1。

B.9 再谈上下文相关性的处理

在第 B.3 节中介绍 LEX 用的正规式时提到了上下文相关性的表示，这里再详细介绍 LEX 提供的处理上下文相关的措施。

要处理的问题是某些规则在不同的上下文中要采取不同的动作，或者说同样的字符串在不同的上下文中有不同的解释。例如，在程序设计语言中，同一个等号"="，在说明部分表示为变量赋初值，这时的动作应是修改符号表内容；而在语句部分等号就是赋值语句的赋值号，这时又应该产生相应于赋值语句的代码。因此要依据等号所处的上下文来判断它的含义。LEX 提供了两种主要的方法，

1) 使用标志来区分不同的上下文

标志是用户定义的变量，用户在不同的上下文中为它置不同的值，以区分它在哪个上下文中，这样识别规则就可以根据标志当前值决定在哪个上下文中并采取相应的动作。

例　将输入串照原样输出，但对 magic 这个词，当它出现在以字母 a 开头的行中，将其改为 first，出现在以 b 开头的行中将其改为 second，出现在以 c 开头的行中则改为 third。使用标志 flag 的 LEX 源程序如下：

```
  int flag;
%%
^a      {flag=娄a娄;ECHO;}
^b      {flag=娄b娄;ECHO;}
^c      {flag=娄c娄;ECHO;}
\n      {flag=0;ECHO;}
magic   {
         switch (flag)
         {
         case 娄a娄:printf(′first′);break;
         case 娄b娄:printf(′second′);break;
         case 娄c娄:printf(′third′);break;
         default:ECHO; break;
         }
        }
```

2) 使用开始条件来区分不同上下文

在 LEX 源程序中用户可以用名字定义不同的开始条件。当把某个开始条件置于某条识别规则之前时，只有在 LEX 处于这个开始条件下这条规则才起作用，否则等于没有这条规则。LEX 当前所处的开始条件可以随时由用户程序(即 LEX 动作)改变。

开始条件由用户在 LEX 源程序的"辅助定义部分"定义，语法是

%Start name1 name2 name3…

其中 Start 可以缩写成 S 或 s。开始条件名字的顺序可以任意给出，有很多开始条件时也可以由多个%Start 行来定义它们。

开始条件在识别规则中的使用方法是把它用尖括号括起来放在识别规则的正规式左边：

<name1> expression

要进入开始条件如 name1，在动作中用语句

BEGIN name1

它将 LEX 所处的当前开始条件改成 name1

要恢复正常状态，用语句

BEGIN 0

它将 LEX 恢复到 LEX 解释器的初始条件。

一条规则也可以在几个开始条件下都起作用，如

<name1，name2，name3> rule

使 rule 在三个不同的开始条件下都起作用。要使一条规则在所有开始条件下都起作用，就不在它前面附加任何开始条件。

例　解决 1)中的问题，这次用开始条件，LEX 源程序如下：

```
%start      AA BB CC
%%
^a          {ECHO;BEGIN AA;}
^b          {ECHO;BEGIN BB;}
^c          {ECHO;BEGIN CC;}
\n         {ECHO;BEGIN 0;}
```

<AA>magic printf(′first′);

<BB>magic printf(′second′);

<CC>magic printf(′third′);

B.10 LEX 源程序格式总结

为使用方便起见，将 LEX 源程序的格式，LEX 的正规式的格式等总录于此。

LEX 源程序的一般格式为：

{definitions}

%%

{rules}%%

{user subroutines}

辅助定义部分包括以下项目：

1) 辅助定义，格式为：

name translation

2) 直接照抄的代码，格式为：

空格代码

3) 直接照抄的代码，格式为：

%{

代码

%}

4) 开始条件，格式为：

%S name1 name2 …

还有几个其他项目，不常使用故略去。

识别规则部分的格式是

expression action

其中 expression 必须与 action 用空格分开，动作如果多于一行，要用花括号括起来。

LEX 的正规式用的运算符有以下一些：

x 字符 x

"x" 字符 x，若为运算符，则不起运算符作用

\x 同上

[xy] 字符 x 或 y

[x-z] 字符 x，或 y，或 z

[∧x] 除 x 以外的所有字符

· 除回车换行外的所有字符

∧x 出现在一行开始处的 x

<y>x 当 LEX 处于开始条件 y 时，x

x$ 出现在一行末尾处的 x

x。 可有可无的 xx 0 个或多个 x

x+ 1 个或多个 x

x|y x 或 y

(x) 字符 x

x/y 字符 x 但仅当其后紧随 y

{xx} 辅助定义 xx 的展开

x{m，n}　　m 到 n 个 x