Lex - 词法分析器生成器

M. E. Lesk 与 *E. Schmidt*
Bell Laboratories
Murray Hill, New Jersey 07974

译者声明：译者对译文不做任何担保，译者对译文不拥有任何权利并且不负担任何责任和义务。
原文：http://cm.bell-labs.com/7thEdMan/vol2/lex

摘要

Lex 帮助书写其控制流由在输入流中的正则表达式的实例来导向的程序。它适合于编辑器脚本类型的变换，和为解析例程做准备工作而分解输入。

Lex 源码是正则表达式和相应的程序片段的表格。Lex 把这个表格变换成读取输入流、复制它到输出流、并把输入划分到匹配给定表达式的字符串中的一个程序。随着每个这种字符串被识别出来，相应的程序片段就被执行。表达式通过用 Lex 生成的确定有限自动机来识别。用户书写的程序片段按照对应的正则表达式在输入流中出现的次序来执行。

用 Lex 写成的词法分析程序接受有歧义的规定，并在每个输入点上选择最长的匹配可能。如果需要，在输入上进行实质的超前查看(lookahead)，但输入流会被回退(backup)到当前划分的结束处，所以用户有操纵它的普遍自由。

Lex 可以生成用 C 语言或 Ratfor 语言写的分析器，Ratfor 可以自动的转换成可移植的 Fortran。它可以在 PDP-11 UNIX、 Honeywell GCOS 和 IBM OS 系统上得到。本手册只讨论在 UNIX 系统上的生成 C 语言的分析器，这是在 UNIX 第 7 版中唯一支持的 Lex 形式。设计 Lex 时简化了与编译器的编译系统 Yacc 的交接。

July 21, 1975

目录

---

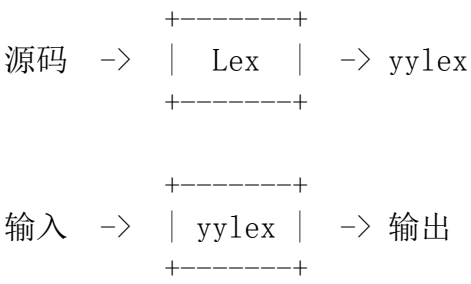
1. 介绍

Lex 是设计用于字符输入流的词法处理的一个程序生成器。它接受高级的、面向问题的对字符串匹配的规定，并生成识别正则表达式的通用语言写的一个程序。正则表达式在用户给 Lex 的源规定中指定。Lex 写出的代码识别在输入流中的这些表达式，并把输入流划分到匹配这些表达式的字符串中。在字符串间的分界上执行用户提供的程序片段。Lex 源文件对正则表达式关联上程序片段。随着每个表达式出现在给 Lex 写出的程序的输入中，相应的程序片段就被执行。

用户提供超出表达式所需要的额外代码来完成他的任务，可能包括用其他生成器写出的代码。生成的识别表达式的程序采用用户的程序片段所采用的通用编程语言。因此，提供了高级表达式语言来写要被匹配的字符串表达式，而用户写动作的自由不受侵犯。这避免了强制希望使用字符串操纵语言做输入分析的用户、去使用同样的并且经常不适合字符串处理的语言来书写处理程序。

Lex 不是完整的语言，而是体现了可增加到叫做“宿主语言”的不同编程语言中的新语言特征的一个生成器。正如同通用语言可以生成在不同计算机硬件上运行的代码，Lex 可以写出不同宿主语言的代码。宿主语言被用于 Lex 生成的输出代码和用户增加的程序片段。还为不同的宿主语言提供兼容的运行时间库。这使 Lex 适应不同的环境和不同的用户。每个应用都可以被定向到适合这个任务的硬件和宿主语言、用户背景和本地实现性质的各种组合上。目前唯一支持的宿主语言是 C，尽管过去曾经支持过 Fortran(Ratfor [2]形式)。Lex 自身存在于 UNIX、GCOS 和 OS/370；但 Lex 生成的代码可以采用于存在适当的编译器的任何地方。

Lex 把用户的表达式和动作(在本文中叫做源码)转换成宿主通用语言；生成的程序叫做 yylex。yylex 程序将识别在流(在本文中叫做输入)中的表达式，并在检测到的时候执行给每个表达式的动作。参见图 1。



Lex 概述
图 1

作为一个平凡的例子，考虑从输入中删除所有行结束处的空白或 tab 的程序。

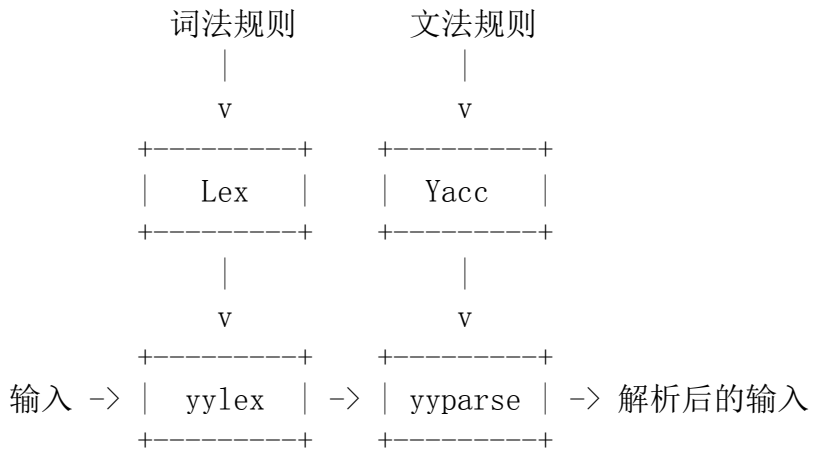
```
%%
[ \t]+$    ;
```

就是所需要的。这个程序包含标记规则开始的一个 %% 分界符和一个规则。这个规则包含匹配在行结束处之前的空白或 tab(按照 C 语言约定写为 \t)字符的一个或多个实例的一个正则表达式。方括号指示由空白和 tab 构成的字符类；同于 QED，+ 指示“一或多个”；而 $ 指示“行结束”。没有指定动作，所以 Lex 生成的程序(yylex)将忽略这些字符。所有其他东西都被复制。要把任何余下的空白或 tab 的字符串改变为一个单一的空白，可增加另一个规则：

```
%%
[ \t]+$    ;
[ \t]+     printf(" ");
```

为这个源码生成的有限自动机将立刻扫描这两个规则，在空白或 tab 的字符串的终止处观察是否有一个换行字符，并执行想要的规则动作。第一个规则匹配在行结束处的所有空白或 tab 的字符串，而第二个规则匹配所有余下的空白或 tab 的字符串。

Lex 可以单独的用做简单变换，或在词法层次上聚集(gather)出的分析和统计。Lex 还可以与解析器一起使用而进行词法分析阶段；交接 Lex 与 Yacc [3] 是特别容易的。Lex 程序只识别正则表达式；Yacc 写出接受一大类上下文无关文法的解析器，但是需要底层的分析器来识别输入记号(token)。所以，Lex 和 Yacc 的组合经常是适当的。在用做后面的解析器生成器的预处理器的时候，Lex 用来划分输入流，而解析器生成器指派文法结构到结果的词法块(piece)。这种情况下的(编译器的前半部分)控制流在图 2 中展示。其他生成器或手工写的额外的程序可以轻易的增加到 Lex 写出的程序。



Lex 与 Yacc 一起

图 2

Yacc 用户会发现 yylex 的名字就是 Yacc 期望词法分析器叫的名字，所以 Lex 采用这个名字简化了交接。

Lex 从源码中的正则表达式生成一个确定有限自动机[4]。为了节省空间，这个自动机是解释的而不是编译的。结果仍是一个快速的分析器。特别是，Lex 程序识别和划分输入流所花费的时间正比于输入的长度。Lex 规则的数目或规则的复杂性在决定速度上都不重要，除了包含前向(forward)上下文的规则需要大量的重新扫描之外。与规则的数目和复杂性一起增加的是自动机的大小，因此也是 Lex 生成的程序的大小。

在 Lex 写出的程序中，用户的片段(表示在找到每个正则表达式时要进行的动作)被收集一起成为 switch 语句中各个 case。自动机的解释器导向控制流。提供给用户在包含动作的例程中插入声明或增加语句，或者在这些动作例程之外增加例程的机会。

Lex 不受限制于可在提前查看一个字符基础上得到解释的源码。例如，如果有两个规则一个查找 ab 而另一个查找 abcdefg，则在输入流是 abcdefh 的时候，Lex 将识别 ab 并留下输入指针正好在 cd 之前。这种回退比起简单语言的处理要更加高代价。

2. Lex 源码

Lex 源码的一般格式为:

{定义}

%%

{规则}

%%

{用户子例程}

这里的定义和用户子例程部分经常被省略。第二个 %% 是可选的，但是需要第一个来标记规则的开始。绝对极小的 Lex 程序是

%%

(没有定义，没有规则)它被变换成无改变的复制输入到输出的一个程序。

在上述 Lex 程序的轮廓中，规则代表用户的控制决定；它们是个表格，在其中左列包含正则表达式(参见章节 3)而右列包含动作，在识别了表达式的时候要执行的程序片段。所以如下一个单独规则

```
integer    printf("found keyword INT");
```

在输入流中查找 integer 字符串并在它出现的时候打印消息“found keyword INT”。在这个例子中宿主过程语言是 C 并使用 C 库函数 printf 打印这个字符串。表达式的结束通过第一个空白或 tab 字符来指示。如果动作只是一个单一的 C 语言表达式，它可以给出在这行的右侧；如果它是复合的或多余一行，则应当包围在花括号中。作为稍微有用一些的例子，假设想要把一些美式拼写改为英式拼写。Lex 规则如下

```
colour      printf("color");
mechanise   printf("mechanize");
petrol      printf("gas");
```

将是一个好的开始。这些规则非常不够，因为单词 petroleum 将变成 gaseum；后面会讨论处理它的方式。

3. Lex 正则表达式

正则表达式的定义非常类似于 QED[5] 中的定义。正则表达式指定要被匹配的一组字符串。它包含文本字符(它匹配被比较的字符串中的相应的字符)和操作符字符(它指定重复、选择和其他特征)。字母表字母和数字总是文本字符；所以正则表达式

```
integer
```

匹配字符串 integer 只要它出现，而表达式

a57D
查找字符串 a57D。

操作符

操作符字符有

″ \ [ ] ^ - ? . * + | ( ) $ / { } % < >
如果它们被用做文本字符，应当使用转义(escape)。引号操作符(")指示在一对引号之间包含的任何东西都被当作文本字符。所以
xyz″++″
匹配字符串 xyz++，在它出现的时候。注意可以引用字符串的一部分。引用普通字符是无害和没有必要的；表达式
″xyz++″
同于上面的表达式。所以通过引用用做文本字符的所有非字母数字字符，用户可以避免记住上面的操作字符列表，并在 Lex 进一步扩展这个列表的时候是安全。

还可以通过前导一个 \ 来把一个操作符字符转义为一个文本字符，比如

xyz\+\+
是上述表达式的另一个更少可读性的等价者。引用机制的另一个用处是把空白介入到表达式中；如上所述正常情况下空白或 tab 结束一个规则。不包含在 [](见后)内的任何空白字符必须被引用。还识别使用 \ 的一些常见的 C 转义: \n 是换行、\t 是 tab 而 \b 是退格(backspace)。要录入 \ 自身需使用 \\。因为换行在一个表达式中是非法的，必须使用 \n；不需要转义 tab 和退格。除了空白、tab、换行和上述列表中字符之外的所有字符总是文本字符。

字符类

字符类可以使用运算符对 [] 来指定。构造 [abc] 匹配一个单一字符，可以是 a、b 或 c。在方括号内，忽略多数字符的意义。只有三个字符特殊：它们是 \ - 和 ^。- 字符指示范围。例如

[a-z0-9<>_]
指示包含所有小写字母、数字、尖括号和下划线的字符类。可以按任意次序给出范围。使用不都是大写字母、不都是小写字母、或不都是数字的在任何一对字符之间的“-”都是依赖实现的并会得到一个警告消息。(比如，[0-z] 在 ASCII 中比在 EBCDIC 中有更多字符)。如果需要在字符类中包含字符 - ，它应当在第一个或最后一个位置上；所以
[-+0-9]
匹配所有数字和两个算符。

在字符类中，^ 操作符必须出现为左方括号后的第一个字符；它只是结果的字符串关于计算机字符集的补集。所以

[^abc]
匹配除了 a、b 或 c 的所有字符，包括了特殊或控制字符；而
[^a-zA-Z]

是不是字母的任何字符。\ 字符提供在字符类方括号内的正常转义。

任意字符

为了匹配最任何的字符，操作符字符 . 是除了换行之外所有字符的类。转义八进制数是可能的，尽管是不可移植的:

[\40-\176]
匹配在 ASCII 字符集中所有可打印字符，从八进制的 40(空白)到八进制的 176(波浪线)。

可选的表达式

操作符 ? 指示表达式的一个可选元素。所以

ab?c
匹配 ac 或者 abc。

重复的表达式

操作符 * 和 + 指示类的重复。

a*
是任何数目的连续的 a 字符，包括零个；而
a+
是 a 的一个或多个实例。例如
[a-z]+
是所有小写字母的字符串。而
[A-Za-z][A-Za-z0-9]*
指示开始于字母字符的所有字母数字的字符串。这是识别计算机语言中标识符的典型表达式。

选择和组合

操作符 | 指示选择:

(ab|cd)
匹配要么 ab 要么 cd。注意使用圆括号作为组合，尽管它们在最外层不是必须的；
ab|cd
就足够了。可以使用圆括号形成更复杂的表达式:
(ab|cd+)?(ef)*
匹配字符串如 abefef、efefef、cdef 或 cddd；但不匹配 abc、abcd 或 abcdef。

上下文敏感

Lex 会识别少量的外围上下文。两个最简单的这种操作符是 ^ 和 $。如果一个表达式的第一个字符是 ^，这个表达式将只在一行的开始处被匹配(在一个换行字符之后，或在输入流的开始处)。这永不会冲突于 ^ 的其他意义，即表示字符类的补集，因为它只适用在 [] 操作符内。如果最后的字符是 $，则这个表达式

只在一行的结束处被匹配(在立即跟随着换行的时候)。$ 操作符是指示尾随上下文的 / 操作符字符的特殊情况。表达式

ab/cd

匹配字符串 ab，但只在它跟随着 cd 的时候。所以

ab$

同于

ab/\n

左上下文在 Lex 中通过在章节 10 中解说的开始条件来处理。如果一个规则只在 Lex 自动机解释器处在开始条件 x 下的时候执行，这个规则应前导着

<x>

使用了尖括号操作符字符。如果我们把“在一行的开始处”考虑为开始条件 ONE，则 ^ 操作符将等价于

<ONE>

后面会详细解说开始条件。

重复和定义

操作符 {} 指定要么重复(如果包围了数字)要么定义展开(如果包围了一个名字)，例如

{digit}

查找叫做 digit 的一个预定义的字符串并把它插入到这个表达式中这一点上。在 Lex 输入中这种定义在规则前面的第一部分中给出。相反的

a{1,5}

查找 a 的 1 到 5 次出现。

最后，初始的 % 是特殊的，它作为 Lex 源码分段的分隔符。

4. Lex 动作

当按如上规定所写出的表达式被匹配了的时候，Lex 就执行相应的动作。本章节描述 Lex 辅助书写动作的某些特征。注意有一个缺省动作把输入复制到输出。它进行在所有没有匹配的字符串上。所以希望抛弃整个输入而不产生任何输出的 Lex 用户必须提供匹配所有东西的规则。当与 Yacc 一起使用 Lex 的时候，这是正常情况。你可能认为动作就是替代复制输入到输出而所要做的；所以一般而言，只做复制的规则可以省略。从规则中被忽略了的、并出现在了输入中的字符组合好象应该被打印在输出中，以此引起对规则有缺口的注意。

可以做的最简单的事情是忽略输入。指定一个 C 空语句 ; 作为动作导致这种结果。常见的规则是

[\t\n] ;

它导致三个间隔字符(空白、tab 和换行)被忽略。

避免写动作的另一个容易的方式是动作字符 |，它指示给这个动作的规则是给下一个动作的规则。前面的例子还可以写为

```
" "    |
"\t"   |
"\n"   ;
```

有相同的结果却有着不同的风格。在 \n 和 \t 外围的引号是不必须的。

在更加复杂的动作中，用户经常希望知道匹配某个表达式如 [a-z]+ 的实际文本。Lex 把这个文本保留在叫做 yytext 的外部字符数组中。所以要打印找到的字符串，使用如下规则

```
[a-z]+    printf("%s", yytext);
```

将打印在 yytext 中的字符串。C 函数 printf 接受一个格式参数和要打印的数据；在这个情况下，格式是"打印字符串"(% 指示数据转换，而 s 指示字符串类型)，数据是在 yytext 中的字符。所以这正好把匹配的字符串放置到输出中。这个动作如此的常见以至于可以简写为 ECHO:

```
[a-z]+    ECHO;
```

它和上面的一样。因为缺省动作就是打印找到的字符串，你可能要问为什么要给出这么一个只是指定缺省动作的规则? 经常需要这种规则来避免匹配不是想要的某些规则。例如，如果有一个规则匹配 read，则它将正常的匹配包含在 bread 或 readjust 中的 read 的实例；为了避免如此，需要形如 [a-z]+ 的一个规则。这将在下面解说。

有时知道已经找到的东西的结束处是更加方便的；所以 Lex 还提供匹配的字符数目的计数 yyleng。要计数在输入中的字的数目和在字中字符的数目，用户可以写

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

它以字节为单位累计识别了的字中的字符数目。在匹配的字符串中的最后的字符可以如下这样访问

```
yytext[yyleng-1]
```

有时，Lex 动作可以确定仍未被识别的规则的正确字符跨度(span)。提供了两个例程在这种情况下做辅助。首先，可以调用 yymore() 来指示下一个识别的输入表达式要续加到这个输入的结束处。正常的，下一个输入字符串会覆写 yytext 中的当前内容。其次，可以调用 yyless (n) 来指示不是当前成功表达式匹配的所有字符都是现在就需要的。参数 n 指示在 yytext 中要保留的字符的数目。更多的以前匹配了的字符被返回给输入。这提供了与 / 操作符所提供的同类的超前查看，但有不同的形式。

例子: 考虑一种语言，它定义字符串为在引号(")之间的一组字符，并规定在字符串中包括 " 必须前导上 \。匹配它的正则表达式有些混乱，所以如下这么写会更好

```
\"[^"]*    {
        if (yytext[yyleng-1] == '\\')
                yymore();
        else
                ... 正常用户处理
 }
```

在面对字符串如 "abc\"def" 的时候，它首先匹配前五个字符 "abc\ ；接着调用 yymore() 导致字符串的下一部分 "def 被续加到结束处上。注意终止这个字符串的最后的引号将被选取到标记为“正常处理”的那部分代码中。

函数 yyless() 可以用来在各种条件下重处理文本。考虑老版本 C 语言的区分“=-a” 的歧义的问题。假设想要把它处理为“=- a”但要打印一个消息。可用一个规则

```
=-[a-zA-Z]   {
        printf("Op (=-) ambiguous\n");
        yyless(yyleng-1);
        ... 对 =- 的动作 ...
}
```

它打印一个消息，把在这个操作符之后的字母返回到输入，并把操作符当作“=-”。作为另一种选择，想要把它处理为“= -a”。要如此只需要把减号同字母一起返回到输入:

```
=-[a-zA-Z]   {
        printf("Op (=-) ambiguous\n");
        yyless(yyleng-2);
        ... 对 = 的动作 ...
}
```

将进行此种释义。注意对这两种情况的表达式可以轻易写为对第一种情况的

```
=-/[A-Za-z]
```

和对第二种情况的

```
=/-[A-Za-z]
```

；在规则动作中不需要回退。不需要识别整个标识符来察觉这个歧义。但是有“=-3”的可能性使

```
=-/[^ \t\n]
```

是更好的规则。

除了这些例程，Lex 还允许访问它所使用的 I/O 例程。它们是:

1. input() 返回下一个输入字符；
2. output(c) 把字符 c 写到输出；和
3. unput(c) 把字符 c 压回到输入流中以被后来的 input() 读取。

缺省的把这些例程作为宏定义提供，但用户可以屏弃它们并提供私有版本。这些例程定义了在外部文件和内部字符之间的关联，因而必须一致的都被保留或都被修改。它们可以被重定义，来导致同陌生的地方传送输入或输出，包括同其他程序或内部内存；但使用的字符集必须在所有例程中是一致的；从输入返回零值必须意味着文件结束；在 unput 和 input 之间的关联必须被保持，否则 Lex 超前查看将不能工作。Lex 根本不超前查看，如果不是必须的话，但结束于 + * ? 或 $，或包含 / 的所有规则隐含了超前查看。超前对于匹配是另一个表达式的前缀的表达式也是必须的。后面有对 Lex 使用的字符集的讨论。标准 Lex 库在回退上施加了 100 个字符的限制。

另一个用户可能想重定义的 Lex 库例程是 yywrap()，Lex 在到达文件结束的时候调用它。如果 yywrap 返回 1，Lex 继续在输入结束时的正常包装(wrapup)。但是，有时重新安排更多的来自新来源的输入是方便的。在这种情况下，用户应当提供一个重新安排新输入并返回 0 的 yywrap。这指示 Lex 继续处理。缺省的 yywrap 总是返回 1。

这个例程还是在程序结束的时候打印表格和总结的方便地方。注意写一个识别文件结束的正常规则是不可能的；对这个条件的唯一访问就是通过 yywrap。事实上，除非提供一个私有版本的 input()，包含空字符(\0)文件是不能被处理的，因为从输入返回的 0 值被当作文件结束。

5. 歧义源规则

Lex 可以处理有歧义的规定。当多于一个表达式可以匹配当前输入的时候，Lex 按如下来选择:

1. 首选最长匹配。
2. 在匹配相同数目字符的规则中，首选最先给出的规则。

所以假定规则

```
integer    关键字动作 ...;
[a-z]+     标识符动作 ...;
```

按这个次序给出。如果输入是 integers，它被接受为标识符，因为 [a-z]+ 匹配 8 个字符而 integer 只匹配 7 个。如果输入是 integer，两个规则都匹配 7 个字符，选择关键字规则因为它是第一个。任何更短的(比如 int)将不匹配表达式 integer 所以使用标识符释义。

首选最长匹配的原理使包含 .* 表达式的规则是危险的。例如 '.*' 好象是识别在单引号中的字符串的好方式。但是它诱使程序更加超前的读取，查找最远的单引号。假设输入是

```
'first' quoted string here, 'second' here
```

上述表达式将匹配

```
'first' quoted string here, 'second'
```

这不大可能是想要的。更好的规则有如下形式

```
'[^'\n]*'
```

它在上述输入上将停止在 'first' 之后。. 操作符不匹配换行的事实减轻了这种错误结果。所以表达式如 .* 停止于当前行之上。不要尝试通过表达式如 (.|\n)+ 或等价者来克服这个限制；Lex 生成的程序将尝试读取整个输入文件，导致内部缓冲区溢出。

注意 Lex 正常的划分输入流，不查找每个表达式的所有可能匹配。这意味着每个字符只被考虑一次且只一次。例如，假设想要计数在输入中 she 和 he 的出现次数。其 Lex 规则可能如下

```
she    s++;
he     h++;
\n     |
.      ;
```

这里的最后两个规则忽略除了 he 和 she 之外的所有东西。记住 . 不包括换行。因为 she 包括 he，Lex 将正常的不识别在 she 中包含的 he 实例，因为一旦它经过了 she 这些字符就不再考虑了。

有时用户可能想要屏弃这种选择。动作 REJECT 意味着“去做下一个选择”。这导致执行在当前规则之后第二选择的不论什么规则。输入指针的位置因而被调整。假设用户实际上想计数 he 的被包含实例:

```
she     {s++; REJECT;}
he      {h++; REJECT;}
\n      |
.       ;
```

这些规则是改变例子这么做的一种方式。它们在计数每个表达式之后被拒绝；只要是适当的，其他表达式将接着被计数。当然，在这个例子中用户可能会注意到 she 包含 he 而反之不然，可以省略在 he 上的 REJECT 动作；但是在其他情况下，不可能事先得知哪些输入字符同在各个类中。

考虑两个规则

```
a[bc]+    { ... ; REJECT;}
a[cd]+    { ... ; REJECT;}
```

如果输入是 ab，只有第一个规则匹配，而对于 ad 只有第二个规则匹配。输入字符串 accb 四个字符匹配第一个规则并接着三个字符匹配第二个规则。相反的，输入 accd 四个字符匹配第二个规则并接着三个字符匹配第一个规则。

一般的说，在 Lex 的目的不是划分输入流而是检测某个项目在输入中的所有实例，并且这些项目的实例可能交叠或相互包含的时候，REJECT 是有用的。假设想要一个输入的连字(digram）表；正常的连字交叠，比如字 the 被认为是包含 th 和 he 二者。假定要递增叫做 digram 的二维数组，正确的源码是

```
%%
[a-z][a-z]    {
        digram[yytext[0]][yytext[1]]++;
        REJECT;
}
.               ;
\n              ;
```

对于选出开始于所有字符而不是在前次发现之后的字母对，这里的 REJECT 是必须的。

6. Lex 源定义

记住 Lex 源码的格式:

```
{定义}
%%
{规则}
%%
{用户例程}
```

迄今为止只描述了规则。用户需要额外的选项来定义在他的程序中和 Lex 所使用的变量。这些定义可以在定义节或在规则节中给出。

记住 Lex 把规则转变为程序。不被 Lex 解释的任何源码都复制到生成的程序中。这种事情分三类。

1. 把开始于空白或 tab 的不是 Lex 规则或动作的一部分的任何行复制到 Lex 生成的程序中。在第一个 %% 分界符之前的这种源输入在代码中将在任何函数的外部；如果它紧接着出现在

第一个 %% 之后，它出现在 Lex 写的包含这些动作的函数内的声明的适当位置上。这种材料看起来象程序片段，并应当领先于第一个 Lex 规则。作为上述规定的副作用，开始于空白或 tab 的包含注释的行被传递到生成的程序中。这可以被用来包含在 Lex 源码或生成的代码中注释。注释应当遵从宿主语言的约定。

2. 把在只包含 %{ 和 %} 的两行之间的任何东西都复制过去。丢弃这些分界符。这个格式允许录入文本如必须开始在第 1 列的预处理语句，或复制不象程序的那些行。
3. 把在第二个 %% 分界符之后的任何东西，无论任何格式，都复制到 Lex 输出的后面。

意图用于 Lex 的定义在第一个 %% 分界符之前给出。在此节中的不包含在 %{ 和 %} 之间的并且开始于第 1 列的任何行都被假定为定义 Lex 替换字符串。这些行的格式是

名字　转换

并导致给名字关联上给为转换的字符串。名字和转换必须由至少一个空白或 tab 分隔，并且名字必须开始于一个字母。转换可以在规则中用 {名字} 语法来调出。例如，对数字使用 {D} 和对指数域(field)使用 {E} 可以简写识别数的规则:

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

注意前两个规则用于实数；二者都需要一个小数点并包含可选的指数域，但是第一个要求在小数点之前至少一个数字，而第二个要求在小数点之后至少一个数字。要正确的处理 Fortran 表达式如不包含实数的 35.EQ.I 所造成的问题，需要一个上下文敏感规则如

```
[0-9]+/"."EQ    printf("integer");
```

可以被用来补充整数的正常规则。

定义节还可以包含其他命令，包括宿主语言的选择，字符集表，开始条件列表，或为更大的源程序调整 Lex 自身内的数组的缺省大小。这些可能性在下面第 12 章节“源格式总结”中讨论。

7. 用法

在编译 Lex 源程序时有两个步骤。首先，Lex 源码必须被转换成生成的使用宿主通用语言的程序。接着这个程序必须被编译和装载，通常与一个 Lex 子例程库一起。生成的程序在叫做 lex.yy.c 的文件中。I/O 库依据 C 标准库[6]而定义。

Lex 生成的 C 程序在 OS/370 上稍有不同，因为 OS 编译器不如 UNIX 或 GCOS 编译器强力，并在编译时间做的更少。在 GCOS 和 UNIX 上生成的 C 程序是一样的。

UNIX

通过装载器标志 -ll 访问 Lex 库。所以一组适当的命令是

```
lex source cc lex.yy.c -ll
```

结果的程序被放置在正规文件 a.out 中用来以后执行。要与 Yacc 一起使用 Lex 请见后面章节。尽管缺省的 Lex I/O 例程使用 C 标准库，Lex 自动机自身不这样；如果给出私有版本的 input、output 和 unput，则这个库可以避免。

8. Lex 与 Yacc

如果你希望与 Yacc 一起使用 Lex，注意 Lex 所写出的是叫做 yylex() 的程序，这个名字是 Yacc 要求它的分析器叫的名字。正常的，在 Lex 库中的缺省 main 程序调用这个例程，但是如果装载了 Yacc 而使用了它的 main 程序，则 Yacc 会调用 yylex()。在这种情况下每个 Lex 例程都应结束于

```
return(token);
```

这里返回适当的记号值。访问 Yacc 给记号指派的名字的容易的方式是编译 Lex 的输出文件为 Yacc 输出文件的一部分，通过放置行

```
# include "lex.yy.c"
```

在 Yacc 输入的最后一节中。假设文法叫做“good”而词法规则叫做“better”，UNIX 命令序列可以是:

```
yacc good
lex better
cc y.tab.c -ly -ll
```

Yacc 库(-ly)应当在 Lex 库之前装载，来获得调用 Yacc 解析器的 main 程序。Lex 和 Yacc 程序的生成可以按任何次序来做。

9. 例子

作为一个平凡的问题，考虑复制输入文件并对可被 7 整除的所有正数加 3。下面是适合这件事的 Lex 源程序

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

规则 [0-9]+ 识别数字的字符串；atoi 把数字转换为二进制并存储结果在 k 中。使用算符 %(取余)检查 k 是否被 7 整除；如果是则对它加 3 并写出。这个程序会改变输入项如 49.63 或 X7。进一步的，它还增加所有能被 7 整除的负数的绝对值。要避免这些，在活跃规则之后增加一些规则，比如:

```
%%
int k;
-?[0-9]+    {
        k = atoi(yytext);
        printf("%d",
        k%7 == 0 ? k+3 : k);
}
-?[0-9.]+               ECHO;
[A-Za-z][A-Za-z0-9]+    ECHO;
```

包含“.”或前导一个字母的数值串将被最后两个规则选出来而不做改变。if-else 语句已经被替代为 C 条件表达式来节约空间；形式 a?b:c 意味着“if a then b else c”。

作为统计聚集的例子，下面是统计字长度的直方图的程序，这里的字被定义为字母串。

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

这个程序累计直方图，但不生成输出。在输入结束时打印这个表格。最后的语句 return(1); 指示 Lex 去进行包装(wrapup)。如果 yywrap 返回零(假)，它暗含了可获得进一步的输入而程序要继续读取和处理。提供永不返回真的 yywrap 导致无限循环。

作为更大的例子，下面是 N. L. Schryer 写的转换双精度 Fortran 到单精度 Fortran 的程序的一部分。因为 Fortran 不区分大写和小写字母，这个例程开始于为每个字母定义包含大小写二者的一组字符类:

```
a       [aA]
b       [bB]
c       [cC]
...
z       [zZ]
```

一个额外的类识别空白:

```
W    [ \t]*
```

第一个规则改变“double precision”为“real”，或“DOUBLE PRECISION”到“REAL”。

```
{d}{o}{u}{b}{l}{e}{W}{p}{r}{e}{c}{i}{s}{i}{o}{n}  {
        printf(yytext[0]=='d'? "real" : "REAL");
}
```

在整个程序中都小心的保持了最初程序的大小写。使用了条件算符来选择正确形式的关键字。下一个规则复制延续卡片指示(continuation card indication)来避免把它们混淆于常量:

```
^"      "[^ 0]    ECHO;
```

在正则表达式中，引号包围着空白。它被解释为“行开始，接着五个空白，接着不是空白或零的任何东西”。注意 ^ 有两个不同的意义。下列规则改变双精度常量为正常浮点常量。

```
[0-9]+{W}{d}{W}[+-]?{W}[0-9]+      |
[0-9]+{W}"."{W}{d}{W}[+-]?{W}[0-9]+      |
"."{W}[0-9]+{W}{d}{W}[+-]?{W}[0-9]+      {
        /* convert constants */
        for(p=yytext; *p != 0; p++)
        {
                if (*p == 'd' || *p == 'D')
                        *p=+ 'e'- 'd';
        ECHO;
        }
}
```

在识别了浮点常量之后，通过 for 循环扫描找到字母 d 或 D。程序接着加上 'e'-'d'，把它转换成字母表上的下一个元素。把修改之后的现在为单精度的常量也写出。随后是必须被重新拼写来除去初始的 d 的一系列名字。通过使用数组 yytext 同样的动作可以用于所有这种名字(下面只给出非常长的列表的一个样例)。

```
{d}{s}{i}{n}          |
{d}{c}{o}{s}          |
{d}{s}{q}{r}{t}       |
{d}{a}{t}{a}{n}       |
...
{d}{f}{l}{o}{a}{t}    printf("%s",yytext+1);
```

必须把初始的 d 改变为初始的 a 的另一个名字列表:

```
{d}{l}{o}{g}      |
{d}{l}{o}{g}10    |
{d}{m}{i}{n}1     |
{d}{m}{a}{x}1     {
        yytext[0] =+ 'a' - 'd';
        ECHO;
}
```

必须把初始的 d 改变为初始的 r 的一个例程:

```
{d}1{m}{a}{c}{h}    {yytext[0] =+ 'r'  - 'd';
```

为了避免避免把名字如 dsinx 检测为 dsin 的实例，某些最后的规则把更长的字选择为标识符并复制剩余的一些字符:

```
[A-Za-z][A-Za-z0-9]*    |
[0-9]+                  |
```

```
\n                    |
.                     ECHO;
```

注意这个程序是不完整的；它不处理 Fortran 的间隔(spacing)或使用关键字作为标识符的问题。

10. 左上下文敏感

有时会期望有一组词法规则应用于在输入中不同的时候。例如，一个编译器预处理器可以区分出预处理语句，并不同于正常语句的分析它们。这要求对预先(prior)上下文的敏感性，有多种方式处理这个问题。例如，^ 操作符是预先上下文操作符，识别直接前导的左上下文，就象 $ 识别直接随后的右上下文。可以扩展出毗连的左上下文，来生成类似于毗连右上下文的一种设施，但未必有它那么有用，因为有关的左上下文经常在更早时候出现，比如在一行开始处。

本章节描述处理不同环境的三种方式：简单的使用标志，在只有一些规则从一个环境改变到另一个环境的时候；使用规则的开始条件；和制作多个词法分析器在一起运行的可能性。在每种情况下，都有规则识别对改变其中分析随后文本的环境的要求，并设置某种参数来反映这种改变。这可以是由用户的动作代码显式的测试的一个标志；这种标志是处理这个问题的最简单的方式，因为根本就不涉及 Lex。但是让 Lex 把标志记住为规则上的开始条件可能更加方便。任何规则都可以关联上开始条件。它只在 Lex 处于那个开始条件的时候被识别。当前开始条件可以在任何时候改变。最后如果对不同环境的一组规则非常的类似，可以通过写多个不同的词法分析器来实现清晰性，并按需要从一个切换到另一个。

考虑下列问题：复制输入到输出，在开始于字母 a 的每行上把字 magic 改为 first，在开始于字母 b 的每行上把字 magic 改为 second ，并在开始于字母 c 的每行上把字 magic 改为 third。所有其他字和所有其他行都不变。

这些规则如此的简单以至于最容易的方式使用一个标志来做个工作：

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

就足够了。

要用开始条件处理同样的问题，必须在定义节中使用如下形式的行把每个开始条件介绍给 Lex

%Start　名字 1 名字 2 ...

这里的条件可以按任何次序命名。字 Start 可以简写为 s 或 S。条件可以在规则的头部使用尖括号 <> 引用:

<名字 1>表达式

只在 Lex 处在开始条件“名字 1”的时候被识别。要进入一个开始条件，执行动作语句

BEGIN 名字 1;

它把开始条件改为“名字 1”。要恢复正常状态，

BEGIN 0;

重置 Lex 自动机解释器的初始状态。一个规则可以活跃在多个开始条件下: <名字 1,名字 2,名字 3> 是一个合法前缀。任何不开始于 <> 前缀操作符的规则总是活跃的。

上述例子可以写为:

```
%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");
```

这里的逻辑完全同于处理这个问题的前种方法，但由 Lex 而不是用户代码做这种工作。

11. 字符集

Lex 生成的程序只通过例程 input、output 和 unput 处理字符 I/O。所以在这些例程中提供的字符表示被 Lex 接受并在 yytext 的返回值中采用。对于内部使用一个字符被表示为一个小整数，如果使用了标准库，它们的值等于表示主机计算机字符的位模式(pattern)的整数值。正常的，字母 a 用同字符常量 'a' 相同的形式来表示。如果通过提供转换这些字符的 I/O 例程改变了这种释义，则必须通过给出一个转换表格告知 Lex。这个表格必须在定义节中，并且必须由只包含“%T”的行包围着。这个表格包含如下形式的行

{整数} {字符串}

它指示关联于每个字符的值。所以下面的例子

```
%T
1     Aa
2     Bb
...
26     Zz
27     \n
28     +
29     -
30     0
31     1
```

...
39 9
%T

样例字符表。

把小写和大写字母一起映射到整数 1 到 26，换行映射到 27，+ 和 - 映射到 28 和 29，数字映射到 30 到 39。注意转义换行。如果提供了这个表格，在规则中或任何有效输入中出现的所有字符都要包含在表格中。没有字符可以被指派数值 0，没有字符可以被关联上比硬件字符集大小更大的数值。

12. 源格式总结

Lex 源文件的一般形式是:

{定义}
%%
{规则}
%%
{用户子例程}

定义节包含如下定义的组合

1. 定义，形如“名字 转换”。
2. 包含代码，形如“ 代码”。
3. 包含代码，形如

```
%{
代码
%}
```

4. 开始条件，形如

```
%S 名字 1 名字 2 ...
```

5. 字符集表格，形如

```
%T
数值 字符串
...
%T
```

6. 改变内部数组大小，形如

```
%x nnn
```

这里的 nnn 是表示数组大小的十进制整数而 x 选择如下参数:

字母	参数
p	位置
n	状态
e	树节点
a	转变
k	包装(pack)起来的字符类
o	输出数组大小

在规则节中的行有“表达式　动作”的形式，这里的动作通过使用花括号做界定可以接续在后续的行上。

Lex 中的正则表达式使用如下操作符:

x	字符 "x"。
"x"	"x"，即使 x 是运算符。
\x	"x"，即使 x 是运算符。
[xy]	字符 x 或 y。
[x-z]	字符 x、y 或 z。
[^x]	除了 x 的任何字符。
.	除了换行的任何字符。
^x	在一行开始处的 x。
<y>x	当 Lex 处在开始条件 y 时的 x。
x$	在一行结束处的 x。
x?	可选的 x。
x*	0,1,2, ... 个 x 的实例。
x+	1,2,3, ... 个 x 的实例。
x|y	x 或 y。
(x)	x。
x/y	x，但只在跟随着 y 时。
{xx}	来自定义节的 xx 的转换。
x{m,n}	x 的 m 到 n 次出现。

13. 告诫和缺陷

有些病态的表达式在转换成确定自动机的时候生成指数增长的表格；幸运的是这很少见。

REJECT 不重新扫描输入；而是记住前面扫描的结果。这意味着如果找到带有尾随上下文的规则，并且执行了 REJECT，用户一定不能使用 unput 来改变即将来自输入流的字符。这是对用户操纵仍未处理的输入的能力的唯一限制。

14. 致谢

从上文可明显的看出，Lex 外部是用 Yacc 模制的而内部采用了 Aho 的字符串匹配例程。所以 S. C. Johnson 和 A. V. Aho 二者很大程度上是 Lex 的实际创作者和调试者。感谢他们。

当前版本的 Lex 的代码是由 Eric Schmidt 设计书写和调试的。

15. 引用

1. B. W. Kernighan and D. M. Ritchie, The C Programming Language, Prentice-Hall, N. J. (1978).
2. B. W. Kernighan, Ratfor: A Preprocessor for a Rational Fortran, Software Practice and Experience, 5, pp. 395-496 (1975).
3. S. C. Johnson, Yacc: Yet Another Compiler Compiler, Computing Science Technical Report No. 32, 1975, Bell Laboratories, Murray Hill, NJ 07974.
4. A. V. Aho and M. J. Corasick, Efficient String Matching: An Aid to Bibliographic Search, Comm. ACM 18, 333-340 (1975).
5. B. W. Kernighan, D. M. Ritchie and K. L. Thompson, QED Text Editor, Computing Science Technical Report No. 5, 1972, Bell Laboratories, Murray Hill, NJ 07974.
6. D. M. Ritchie, private communication. See also M. E. Lesk, The Portable C Library, Computing Science Technical Report No. 31, Bell Laboratories, Murray Hill, NJ 07974.