Lex - A Lexical Analyzer Generator

M. E. Lesk and E. Schmidt

概要

Lex 是编写接受正则表达式、匹配输入为主要控制流程的程序的辅助工具。它非常适合用来转换编辑脚本，以及为文法解析器分词解析输入流。

Lex 定义源码由正则表达式和相应的程序片段组成的表构成。这个表被转换为读入输入流，将之按照与正则表达式的匹配分解，然后输出的程序。当字串被识别后，还会调用相应的程序片段。这种识别过程由 Lex 产生机器人来完成，十分准确。用户书写的程序片断按照相对应的规则被识别的顺序来调用。

使用 Lex 生成的词法分析器能处理存在歧义的一组规则，为输入选择最长的匹配。如果有必要，会对输入流向前预读；在预读时，在此之前输入流会被备份起来，因此用户一般都能巧妙的处理之。

Lex 能产生 C 语言或者是 Ratfor 语言形式的词法解析器（Ratfor 是一种可以自动转换为可移植 Fortran 的语言）。Lex 支持 PDP-11，UNIX，Honeywell GCOS, 及 IBM OS 系统。但是本文只讨论在 UNIX 系统下面产生 C 语言形式解析器的情况，这也是在 UNIX 第七版下 Lex 唯一支持的形式。Lex 同时设计了简洁的 Yacc 接口，供这种编译器的编译系统（compiler-compiler system）使用。

1.绪论

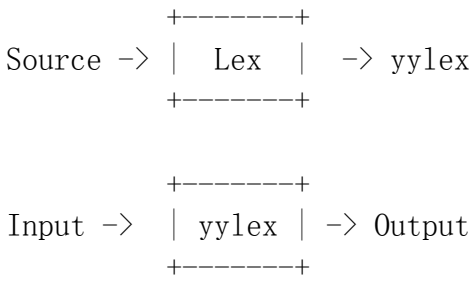
Lex 是用来产生对输入流字串做词法分析的程序的工具。它接受一个高级的，针对字串匹配方法的规则，以之生成一个识别正则表达式的通用程序。这些正则表达式由用户在提供给 Lex 的规则文件中给出。Lex 生成的代码在输入流中匹配这些正则表达式，并按照匹配的情况将输入流分割开来。分割开后，就执行用户提供的程序段。正则表达式和用户提供的程序段在 Lex 定义源码中联系起来。每当一个正则表达被匹配成功，相应的程序段就会被执行。

为了完成任务而超出匹配工作部分的代码，是用户提供的，也可能由其他的生成工具产生。用户的代码调用一般用途的识别正则表达式的程序（Lex 产生）。因此，我们提供了一种高级的描述表达式的语言，以编写需要匹配的表达式类型，也很好的保证了用户编写相应操作的自由。也避免了用户使用不擅长字串处理的语言，而是转而使用专门处理字串的语言，编写输入分析处理程序。

Lex 并不是完整的一门语言，而是一个生成器，可以作为一种新特征加入到不同程序设计语言（通常称之为“宿主语言”）中。Lex 可以为不同的宿主语言产生

代码，而成为能产生运行于不同计算机环境的代码的通用语言。Lex 输出代码以宿主语言书写，用户提供的程序片断也同样如此。同时提供兼容不同语言所使用的运行时库。这样 Lex 就能适应不同的环境与不同的用户。计算机硬件、始于任务的宿主语言、用户的知识背景及具体的实现方法因此结合起来，方便生成应用程序。但是现在情况下，唯一支持的宿主语言是 C 语言，尽管过去 Fortran 以 Ratfor[2]形式支持过。Lex 可运行于 UNIX，GCOS 和 OS/370 上；其产生的代码可以在任何宿主语言编译器存在的地方使用。

Lex 将用户输入的表达式和动作代码（本文中统称源码）转换为一般的宿主语言形式；其产生的程序名为 yylex。yylex 程序识别输入流（本文统称输入）中的表达式，为每一个检测到的表达式调用合适的动作。如图 1 所示



An overview of Lex
图 1

看一个小例子，去除掉输入中行尾所有空白符（blank key）与制表符（tab key）的程序

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

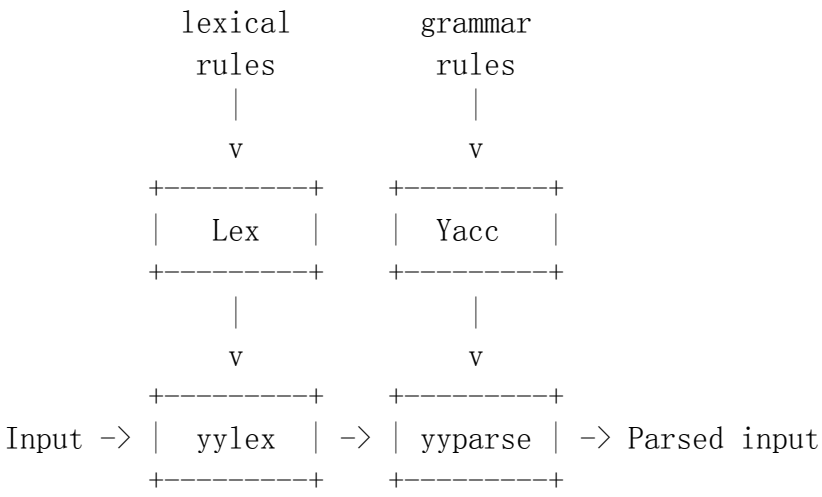
以上即为所需要的全部规则。%%标志了规则的开始，然后是一条规则。这条规则包含了一个正则表达式，它匹配行结束之前有一个以上空白符或者制表符（写为 \t，与 C 语言的习惯相同）的字串式。括号表示由空白符和制表符组成字符集；+表示“一个或一个以上……”；$表示“一行的结束”，如同 QED 中一样。没有指定动作，因此 Lex 产生的程序（yylex）将忽略这些字符。其它的原样拷贝。如果要将剩下的多余的空白符和制表符用一个空白符代替，如下加入一条规则：

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

从这个源码产生的机器人程序将精确的按照这两条规则检查，查找到字串结尾的空白符和制表符（无论有没有换行符号），并执行相应的动作。第一条规则匹配所有行结束前有空白符和制表符的字串，第二条规则则匹配所有剩下来的含有空白符和制表符的字串。

Lex 可以单独用作产生简单转换程序，或者词法意义上的统计分析程序。还可以将 Lex 和语法解析器构造程序一起用作文本分析与解析工作；特别是，将 Lex 和 Yacc 接合起来使用相当容易[3]。Lex 产生的程序只处理正则表达式；Yacc 能产生遵循大量与上下文无关的规则的解析器，但是却需要低级的词法分析器来识别输入的 token。因此，Lex 和 Yacc 的接合通常十分合情合理。当用作语法解

析器的预处理器时，Lex（产生的程序）的作用是分割输入流，然后语法解析器来为 Lex（产生的程序）的结果判别结构。这个过程（这可能是某个编译器的前半部分）的流程如图 2 所示。外部程序，由其他工具产生的或者是手工编写的，可以轻松的加入到 Lex 生成的程序里去。



Lex with Yacc
Figure 2

Yacc 的用户会记得 yylex 正是 Yacc 希望用户提供的词法分析器的名字,因此 Lex 干脆使用它，以简化与 Yacc 的接合。

Lex 能按照源码中的正则表达式生成十分准确的机器人代码（解析器）[4]。这个机器人程序是解释型的，而不是编译型的，原因是为了节省空间。但是它生成的程序运行效率高。特别的说，Lex 产生的程序识别和分割一条输入的时间和输入的长度是成比例的。Lex 规则的条数和规则的复杂性不会十分影响解析速度，除非规则明显要求了大量重复扫描操作。规则的数量和复杂性真正增加的是机器人代码的大小，因此也决定 Lex 产生的应用程序的大小。

在 Lex 产生的代码中，用户的程序段（表示的是匹配到正则表达式时该执行的动作）被集合在一个 switch 语句中 case 条目下。机器人解释器控制着程序流程。此外，用户还能增添声明、另外的包含操作的过程到过程中，或者是在操作过程的外部添加子程序。

Lex 不会因为需要预读一个字符而局限于输入源，不往前解析。例如，如果这里有两条规则，一个寻找 ab，另一个寻找 abcdefg，输入流是 abcdefh，Lex 将会识别为 ab，并将游标至于 cd. . .之前。因为如果向前预测，备份输入的代价太大，以至于超过处理简单语言的代价。

2. Lex Source.

Lex 源码的一般形式是：

```
{定义（definitions）}
%%
{规则（rules）}
%%
{用户子程序（user subroutines）}
```

当然定义和用户子程序常常可以省略。第二个%%是可选的，但是第一个%%必须放在规则的开始。因此最小的 Lex 程序是

```
%%
```

（没有定义，也没有规则），这个源码生成的程序仅仅将输入毫无改变的拷贝到输出。

从上面展示的 Lex 源码看出，规则代表了用户的控制意图；它们以表的形式出现，左边的一列包含正则表达式（见第 3 部分），右边的列包含操作，即正则表达式被匹配成功时要执行的程序片断。因此，一条规则将以如下形式出现

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

这表示了搜寻输入流中的字串“integer”，并在发现时打印消息“found keyword INT”。这个例子中宿主语言是 C，并且用到了 C 库函数 printf 来打印消息。正则表达式结束用单个空白符或者制表符来标志。如果操作就是一条 C 语句，那么就可以直接写在正则右边；如果是组合起来的一些语句，或者超过了一行，就必需要用大括号括起来。看一个更加实用的例子，假设要求将一些单词从英国英语拼写法变成美国拼写法形式。Lex 规则可以如下开始

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

但是这仅仅是个开始，这些规则还不够，例如单词 petroleum 还需要改变成为 gaseum；解决这个问题的方法我们等一下再讨论。

3. Lex 支持的正则表达式

Lex 对于正则表达式的定义与 QED[5]中的定义类似。正则表达式表示了一组相似的字串。它由文本符号（用来匹配待比较字串中的相应字符）和操作符号（表示了重复、选择及其它意思）组成。文本符号通常是字母和数字；因此如下的 Lex 正则表达式

```
integer
```

将匹配输入中出现的任何“integer”文本符号，而

```
a57D
```

就是找寻的“a57D”。

操作符（Operators）

全部操作符号如下

″ \ [ ] ^ - ? . * + | ( ) $ / { } % < >

如果需要将这些操作符也当做字符使用，必须使用一个转义符（escape）。引号操作符(")表示包含在其中的内容将看作文本符号。因此

xyz"++"

将匹配 xyz++。注意只需要将字串用引号括起来。将一般文本符号括起来也没有害处，同样也不必要；如下的表达式

"xyz++"

和上面意思完全一致。因此，使用引号将非字母和数字的字符括起来，当做文本符号，能够使得我们避免记忆上面的操作符号列表，也能使得 Lex 以后安全的扩展这个操作列表。

操作符也能通过在前面加一个“\”符号转义为普通文本字符

xyz\+\+

但这种取代方法比较上面的方式来说比较难读。引号的另一个作用是在表达式中包含入一个空白；因为，一般情况下，空白符或者制表符表示一个规则的结束。任何不在[]（接下来会看到）中的空白符都必须用引号括起来。还有几个 C 语言的转义字符也是接受的：\n 表示换行，\t 表示制表符，\b 表示回退。使用\\表示\本身。因为换行在表达式中是无法表示的，因此必需要使用\n；制表符和回退符则不要求必须使用。除开上面的空白符、制表符、换行符，其它的都是文本字符。

字符集（Character classes）

字符集使用一对操作符[]来声明。结构[abc]只匹配一个字符，可以是 a，b 或者 c。在方括号之间，大部分操作符的意思都会忽略。只有三个字符是特殊的：\，-和^。-表示范围。例如

[a-z0-9<>_]

表示这个字符集包括了所有的小写字母，数字，尖括号和下划线。范围的左右顺序两种都可以。如果-两边不都是大写字母、小写字母、数字等平台独立的符号，将获得一个警告。（比如，[0-z]在 ASCII 中比在 EBCDIC 中含有的字符多）。如果需要将字符-也包含在字符集中，它必须放在最前面或者最后面；因此

[-+0-9]

就匹配所有的数字和正负号。

^操作符在字符集中必须紧跟在左边方括号后；它表示取反操作，即结果需要匹配在当前计算机字符集中除去若干字符后的集合。因此

[^abc]

能与除 a，b 或者 c 的所有字符匹配，也包含所有的特殊字符和控制字符；又如

[^a-zA-Z]

匹配所有非字母的字符。\字符在字符集中同样用作转义的作用。

任意字符（Arbitrary character）

为了匹配几乎所有的字符，使用操作符.，它表示了所有的字符的字符集（除开换行）。通过八进制来转义也能够达到效果，但是这样可能不具有可移植性

```
[\40-\176]
```

上面的字符集就匹配所有的 ASCII 可打印的字符，从八进制的 40（空白符）到八进制 176（一种加在西班牙语 n 上的发音符号）。

可选式（Optional expressions）

操作符?表示一个可选字符。因此

```
ab?c
```

可以匹配 ac 或者 abc。

重复式（Repeated expressions）

操作符*和+表示重复的意义。

```
a*
```

匹配任何 a 后面有字符的字串，没有也行；而

```
a+
```

匹配一个 a 或者多个 a 的字串。例如

```
[a-z]+
```

匹配所有小写字母的字串。还有

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

匹配所有以字母开头由字母和数字组成的字串。这是一个典型的用来识别计算机语言的表达式。

选择与分组（Alternation and Grouping）

操作符|表示选择，如下：

```
(ab|cd)
```

匹配 ab 或者 cd。注意圆括号是用来分组的，尽管这里并不十分需要这样；如此

```
ab|cd
```

也行。圆括号的作用是构造更加复杂的表示式，如下：

```
(ab|cd+)?(ef)*
```

它能匹配 abefef，efefef 或者 cddd 等；但是不能匹配 abc，abcd 或者 abcdef。

上下文敏感性（Context sensitivity）

Lex 也能识别一些上下文的特征。为此而有操作符^和$。如果表示式的第一个字符是^，将只匹配一行开始的字串（在换行符之后，或者是输入流的开始）。这个^不会与刚才那个取反操作的^发生冲突，因为那个只在[]中才有意义。如果行

尾是$，表示式将只匹配行尾的字串（紧接着换行符）。最后一个介绍的操作符是一个特殊的操作符\，表示拖尾的意思。例如，如下 Lex 表示式

```
ab/cd
```

匹配字串 ab，但是必须是跟着 cd 的 ab。因此

```
ab$
```

就与下式等价

```
ab/\n
```

根据开始条件（start conditions）来处理左部的上下文，这个我们在第 10 部分讨论。如果希望某条规则只在 Lex 自动解释器发现某个开始条件时才匹配，那么必须为该规则加上 Lex 夹在尖括号中的前缀

```
<x>
```

考虑“行开始”条件（操作符^），设这个开始条件是 ONE，那么操作符^就等价于

```
<ONE>
```

我们后面再详细的讨论开始条件。

重复和定义（Repetitions and Definitions）

操作符{}既可表示重复（如果括起来的是数字），也可表示定义（如果括起来的是一个名字）。例如

```
{digit}
```

寻找一个预先定义的 digit 类型，因此在{}之间插入这样的名字。预先定义的类型在 Lex 规则输入文件的第一部分规则之前给出。此外，

```
a{1,5}
```

匹配 a 的一次到五次出现。

最后，%是特殊符号，作为 Lex 源码的分隔符。

4.Lex 操作（Lex Actions）

当正则表达式被匹配时，Lex 将执行对应的操作（action）。这一部分描述 Lex 用来辅助编写操作的机制。注意存一个默认的操作，它会对所有没有任何匹配的字串执行，即将输入拷贝到输出。因此，Lex 用户如果想接受所有的输入而不输出任何东西的话，那么就必须提供匹配任何字串的规则。Lex 在和 Yacc 一起合作时，这就是正常的情况。用户只需要考虑除开拷贝输入到输出意外的操作；因此，一般的，可以省略仅仅拷贝的规则。另一方面，输入中被规则忽略的字符组合，会被直接输出，因此需要注意规则存不存在遗漏。

处理字串最简单的事情莫过于将输入完全忽略了。这只需指定一个 C 语言中的空语句，即以; 作为操作代码。最经常使用的即为

```
[ \t\n]    ;
```

这一句使得空白符，制表符和换行符被忽略掉。

另一个定义操作的简洁方式是使用字符|，它代表当前的操作与下一条规则的操作相同。前面的例子等价可以写成这样

```
" "
"\t"
"\n"
```

（译注：原文的这部分如上所示，但是据文中的意思理解，不合逻辑，窃以为应该是

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

）
其中\n 和\t 的双引号是不必要。

在更加复杂的操作代码里，用户经常需要能知道匹配正则表达式的实际文本串，比如[a-z]+。Lex 将这个文本串存在一个外部字符串数组 yytext 里面。因此，打印匹配到得字串，可以设立规则如下

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

来打印出 yytext 中得字串。C 语言中，函数 printf 接受一个格式串参数和要打印得数据项；在这里，格式串就是“print string”（%表示数据转换，s 表示字串类型），数据项即为 yytext 中得字符。因此这里只是将匹配的字串打印到输入而已。由于这个操作十分普遍，因此也可以用 ECHO 来简写：

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

上面这样也是等价的。由于默认的操作也是打印发现的字符，那为什么还需要这样的操作呢？比如上面这条，不仅仅是定义了默认的操作吗？实际这种规则是必须的，用来避免匹配那些不想匹配的规则。例如，一条匹配文字输入的规则，也同业可能匹配其他的输入（比如面包、调味品^_^）；为了避免这一点，就需要[a-z]+这样一条规则。我们后面将详细解释这一点。

有时候知道最后找到的字串的末尾字符反而更加方便；因此，Lex 提供了匹配到的字串的字符数量变量 yyleng。如果需要统计找到的单词量及找到的字符数，用户可以来写为[a-zA-Z]+ {words++; chars+=yyleng;}，这条规则通过累加找到的单词字符数来达到这个目的。匹配到的字串中的最后一个字符也可以这样得到

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

有的时候，Lex 操作需要决定当前的字串范围是否合适。Lex 提供了两个过程来辅助解决这种情况。首先，yymore()用来表示接下来读入的字符串将附着在当前的字符串的末尾。一般的，接下的输入将导致复写 yytext。其次，yyless(n)用来表示不是匹配得到的当前所有的字符串都是需要的。参数 n 表明了 yytext 中

需要保留的字符数。其他的多余的字符将被退回到输入流中。这种机制提供类型操作符/提供的预测的功能，但却是以另外一种形式。

例子：考虑一种语言，它定义了字串类型为夹在双引号（"）中的一些字符，且允许在其中使用"字符，但是必须以\引导。这些看似糊涂的规则可以使用如下正则表达式来刻画，我们可以这样写

```
\"[^"]*    {
           if (yytext[yyleng-1] == '\\')
                yymore();
           else
                ... normal user processing
           }
```

面对一个如同"abc\"def"的字串时，这个正则表达式将首先匹配前五个字符"abc\；然后就调用 yymore()，它将使得接下的部分，"def，将之附着在前面的字串最后。注意结束的引号必须在标有``normal processing''的部分被读入处理。

在很多情况下函数 yyless()可以用来重复处理文本。考虑 C 语言的一个问题，区别``=-a''的多义。假设需要处理为``=- a''，同时打印一条信息。可以用如下的规则

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

这条规则即是打印出一条信息，操作符视之为``=-''，将操作符后的字符退回输入流。另一种对这种问题的处理办法是处理为``= -a''。要如此处理的话，将负号和字符一起退回输入流即可：

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

以上的规则就会执行这种解释。注意的是，这两种情况下也可以简单的写为

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

以上针对第一种情况

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

以上针对第二种情况；这种写法的规则就不需要回退了。它不需要识别整个字串来判别其多义性。当然还有可能出现``=-3'',因此如下

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

的规则将更有适用性。

除开上面的函数，Lex 还提供其它用来访问 I/O 的函数过程。它们是：

1) input()返回下一个输入字符；

2) output(c)向输出写入一个字符；

3) unput(c)回退字符 c 到输入流中，以被下一次 input()读取。

默认的，这些函数过程以宏定义的形式提供，用户也可以覆盖它们以提供自定义的版本。这些函数过程定义了外部文件和内部字符之间的关系，因此或者全部修改或者全部保留。它们可以被重定义，以使得输入或者输出能传递到一些不寻常的地方，包括其他的程序中或者内存中；其使用的字符集必须是一致的；输入过程返回值 0 即表示文件结束；还有 unput 和 input 之间的依存关系也必须保留，否则 Lex 的预测机制就会失效。Lex 只会在必要的时候才采取预测行为，以+＊？或者 $结尾或者包含了/的规则即暗示着执行预测。预测机制在匹配前缀时也同样是必要的。Lex 使用的字符集的讨论后面会讨论到。默认的，标准的 Lex 库遵守 100 个字符的备份限制。

另一个用户有时需要重定义的 Lex 库函数过程是 yywrap()，它在 Lex 到达文件结束（end-of-file）时调用。如果 yywrap 返回 1，Lex 将采取正常输入结束时的 wrapup 操作。但有些时候，可能从新的源文件再输入一些东西比较好。这种情况下面，用户就必须提供一个 yywrap 函数，返回 0，安排好新的输入过程，指导 Lex 继续处理。

这个过程在需要在程序结束时打印表格，总结等等时也是很方便的。我们注意到，编写一个识别文件结束的规则又是不可能的；唯一的解决这种问题的方法就是通过 yywrap。还有，事实上，除非提供一个自定义的 input()版本，否则含有 null 值的文件是无法处理的，因为 input 如果返回一个 0 值，就意味着文件结束（end-of-file）。

5. 构成歧义的规则（Ambiguous Source Rules）

Lex 也能处理好构成歧义的规则。当出现几个可以匹配到当前输入的正则表达式时，Lex 将按如下规则选择：

1)首选最长的匹配。

2)在匹配同样多字符的情况下，选择第一个给出的规则。

因此，假设有如下排列的规则

```
integer    keyword action ...;
[a-z]+     identifier action ...;
```

如果输入是“integers”，因为[a-z]＋匹配 8 字符而 integer 只匹配 7 个，所以被当做 identifier 对待。如果输入是“integer”，两条规则都匹配 7 个字符，

但是会选择 keyword 这一条，因为它排在前面。其它更短的输入串（例如 int）都不会匹配到正则表达式 integer，因此将之作为 idetifier 对待。

首选最长匹配的原则，也存在一些危险，比如含有.*的正则表达式。例如'.*'，看起来这是一个很好的用来匹配单引号括起来的字串的方法。但是它却使得解析程序尽量的向前读，找到相隔最远的单引号。假设现在是如下的输入

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

上面的正则表达式将匹配

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

这个结果可能正好不希望看到的。比较好的定义规则的方式是

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

这个规则在匹配时对上面的输入的会在'first'后停住。如此这样避免错误的因素主要是由于事实上操作符.不匹配换行符。因此象.*的正则表达式只会在当前行停住。不要更错误的使用(.|\n)+或者与之等价的正则表达式；Lex 以此产生程序会尝试将输入文件全部读入，导致内部缓冲区溢出。

需要注意的是，Lex 正常划分输入流时，不会找出所有正则表达式的可能匹配。这即意味着每一个字符都只会计算一次也只计算一次。例如，假设需要分别统计输入文本中的 she 和 he 出现的次数。可能有人会这样写 Lex 规则

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

其中最后两条规则将导致忽略除开 he 和 she 的所有字串。请注意.不包括换行符。由于 she 包含乐 he，正常的 Lex 将会忽略掉 she 中包含的 he，因为它已经在 she 中计算过一次了。

有时候，用户也希望能覆盖这种现象。动作 REJECT 就意味着“执行下一个可选项（go do the next alternative）”。它使得当前规则后面的备选规则能够被执行。因此也会相应调整输入指针的位置。假如用户一定要统计出包含在 she 里面的 he 的个数，那么：

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

以上给出改变了上面的那组规则以适合这个任务的方法。在每条规则都被执行后，输入字串才被丢弃；即在任何第一条规则匹配的情况，另一条规则都会处理记数。当然在这个例子里，我们会注意到只有 she 包含 he，相反则不可能，因此用户也可以忽略掉 he 动作中的 REJECT；但是在其它情况下，在两种匹配下给出那个输入字串是优先的往往是困难的。

考虑如下的两条规则

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

如果输入是 ab，只有第一条规则被匹配到，对 ad 则是第二条被匹配。输入 accb 有 4 个字符被第一条规则匹配到，第二条只匹配 3 个字符。相对的，输入 accd 被第二条规则匹配 4 个字符，第一条则只匹配 3 个。

一般的，REJECT 适合在使用 Lex 检测输入流中某些项的所有出现，而不是用来分割输入流，这些项常常出现相互包含的情况。假设需要统计一个输入里面的连字表；正常的，连字是常常重叠的，比如单词 the 中就包含了 th 和 he。现有一个二维数组 digram 用来记数，合适的规则源码如下

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

在上面的规则中，REJECT 就是必要的，用来找出来从任意字符开始的连字，而不是仅仅出现在第一个的字符。

6.Lex 源码中的定义（Lex Source Definitions）

大家应该还记得 Lex 源码的格式如下：

```
{definitions}
%%
{rules}
%%
{user routines}
```

到现在为止我们只描述了其中的规则部分。然而用户还需要另外的选择，即定义变量以供自己的程序使用，也供 Lex 使用。这些既可以在定义部分声明也可以在规则部分声明。

Lex 将规则转换为程序。不被 Lex 截取处理的代码将直接拷贝到生成的程序中。这种不被截取处理的代码有三种

1)以空白符或者制表符开头的，不属于任何规则或者操作代码的代码行将直接拷贝到 Lex 生成的程序中。其中，在第一个%%分隔符前的源码将成为全局代码，对于程序中的所有的代码均可见；如果是紧接再第一个%%符号后面的代码，声明将出现在 Lex 生成的包含操作的函数里的合适位置。与以上所说不同，以空白符或者制表符开头，包含了注释的行，将直接传递到 Lex 生成的程序代码中。这种机制可以达成在 Lex 源码和生成代码中同时包含注释的目的。但是注释需要遵守宿主语言的规定。

2)包含在%{和%}中得代码行也将如上一样直接拷贝到生成代码中。分隔符会被省略。这种格式使得预处理输入文本的代码成为可能，比如必须从第一列开始读，或者达到拷贝不像是程序的行的目的。

3)在第三个%%分隔符之后的内容（与格式无关），将直接拷贝到 Lex 生成的代码之后。

定义部分在第一个%%分隔符之前给出。在这部分没有包含在%{和%}之间的行，均从第一列开始，都是定义的用来 Lex 的缩略语。这些定义行的格式就如同名字翻译，使得一种翻译串与名字联系起来。名字和翻译串之间至少需要用一个空白符或者制表符来分隔，且名字必须用字母开头。然后就可以用{名字}的形式来在规则中调用翻译串。例如，使用{D}代替数字，{E}代替指数部分，就可以如下简化来编写规则：

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

注意前面表示实数的两条规则;它们都包含一个十进制的小数及一个可选的指数部分，第一条要求在小数点之前至少有一个数字，而第二条要求在小数点之后至少有一个数字。为了正确的处理 Fortran 语言中形如 35.EQ.I 的表达式带来的困难（Fortran 没有实数类型），需要一个上下文敏感的规则，如下

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

这可以用来作为整数规则的附加规则。

定义部分还可以含有其它的命令行，包括宿主语言的选取，字符集表，一系列开始情况，或者是调整 Lex 内建的默认的数组以生成适合大量输入的程序。这些问题将在下面``源码格式总结（Summary of Source Format）''（第 12 部分）部分讨论。

7.使用方法（Usage）

编译 Lex 源码需要两个步骤。第一，Lex 源码先被转换为一个以宿主语言写成的生成程序。然后编译这个程序并装载，经常我们在这个阶段需要使用 Lex 库程序。生成程序存在文件 lex.yy.c 中。其中的 I/O 库按照标准 C 库来定义[6]。

在 OS/370 系统环境下 Lex 产生的 C 程序稍稍不同，因为这个 OS 下的编译器没有 UNIX 或者 GCOS 下的编译器功能强大，虽然编译时更快。在 GCOS 和 UNIX 系统环境下产生的 C 程序是一样的。

UNIX：在 UNIX 下 Lex 库使用装载参数参数-ll 来指示。因此合适的编译命令是 cc lex.yy.c -ll。编译结果通常是 a.out，以备后面使用。如果要将 Lex 和 Yacc 结合起来使用请看下面部分。尽管默认 Lex 的 I/O 库使用标准 C 的库程序，Lex 自动控制器（Lex automata）本身并不是这样；如果自定义版本的 input，output 和 unput 存在的话，就不会使用标准库。

8.Lex 与 Yacc（Lex and Yacc）

如果你需要结合 Lex 和 Yacc 使用，您会注意到 Lex 生成的程序名称为 yylex()，而这刚好是 Yacc 需要的词法分析器的名字。一般的，Lex 库中的默认主程序调用这个过程（yylex()），但是如果装载了 Yacc 并使用了其主程序，Yacc 就会调用 yylex()。这种情况下，每条 Lex 规则都需要以

```
return(token);
```

结束，以返回合适的 token 值。使得 Lex 能访问 Yacc 的 token 名字的一个简单方法是，在 Yacc 输出文件的输入的最后一部分放置一条语句#include "lex.yy.c"，将 Lex 程序包含进来，并和 Yacc 一起编译。假设语法规则文件名为``good''，词法规则文件名为``better''，则编译的 UNIX 命令序列即为：

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

Yacc 库（-ly）必须被在 Lex 库之前被装载，以让 Yacc 主程序来调用 Yacc 语法解析器。而 Lex 程序和 Yacc 程序的生成的顺序可以随意。

9.例子（Examples）

先看一个微不足道的例子，将输入中中能被 7 整除的数字加 3 输出，其余的原样输出。以下是合适的 Lex 源码

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

规则[0-9]+识别出数字字串；atoi 将字串转换为数并存储在 k 中。操作符%（求余数）用来检查 k 是不是能被 7 整除；如果能的话，将之加上 3 后，然后输出。但是这个程序也会改变如 49.63 或者 X7 这样的输入项，因而使您反感。更进一步说，它也会改变一切能被 7 整除的负数的绝对值部分（这其实是错误的）。为了避免这一点，只需在有操作的这条规则后多加入几条规则，如下：

```
%%
                    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;
```

最后两条规则将识别含有小数点的或者以字母开头的数字字串，不对之做改变。If-else 语句也被 C 中的条件表达式简化；a?b:c 形式即意味着“如果 a 成立则为 b 否则为 c”。

再举一个统计信息收集的例子，这个程序给出单词的长度的柱状报告图，这里的单词指的是由字母组成的字串。

```
        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);
}
```

这个程序只给出柱状图，而没有任何的输出。在输入的结尾，它打印出柱状图表格。最后的一句是 return(1)；表明 Lex 将调用 wrapup()函数。如果 yywrap 返回 0（false），意味着还需要更多的输入，程序也会继续读入输入和处理。因此，yywrap 函数永远不应该返回一个 true 值，这样会导致无限循环。

再来举一个稍微大一些的例子，它是 N. L. Schryer 所写的用来将 Fortran 中双精度数转换为单精度数的程序的一部分。因为 Fortran 不区分大小写，这个程序以定义一些包含了两种写法的字符集开始：

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

还有一条规则用来辨别空白：

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

第一条规则将``double precision''字串转换为``real''字串，``DOUBLE PRECISION''字串转换为``REAL''字串。

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

为了不破坏原文的大小写形式，设计这个程序时必须时时小心。条件操作符就是用来选择合适的关键字形式的。接下来的这条规则用来直接输出接着的 card indication（译注：不明白，没学过 Fortran，没法翻，所以原文保留），以避免和常数混淆：

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

上面这个正则表达式表示：“一行以五个空白符开始，然后紧接一个任意字符，除开空白符和 0”。请注意两个^符号的不同意义。接下来是一些将双精度常数转换为单精度常数的规则。

```
[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;
          }
```

在浮点常数被识别后，一个 for 循环用来寻找其中的字母 d 或者 D。然后程序将之加上'e'-'d',使之转换为接下来的字母表中的那个字母。修改过后的单精度常数被原样输出。接下来还有一些名字必须被重新拼写，以去掉它们开头的字母 d。通过使用字符数组 yytext，对所有这样的名字执行同样的操作（以下仅仅是规则的一部分，因为这个列表太长了）：

```
{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);
```

另一些名字必须将开头的 d 改为 a：

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

还有一个规则用来将开头的 d 改为 r：

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

为了避免将如 dsinx 的字串识别为 dsin，最后还需要一些用来能匹配到最长的字串的规则，以将这些“幸存”的字符原样输出：

```
[A-Za-z][A-Za-z0-9]*    |
[0-9]+                  |
\n                      |
.                       ECHO;
```

注意，这个程序到此并不完整；它需要处理 Fortran 中空白的问题，以及使用和关键字一样的标识符的问题。

10. 上下文环境的左敏感性（Left Context Sensitivity）

有时也需要创建根据不同输入情况匹配的一组词法规则。例如，一个编译预处理器需要区分预处理语句和分析它们与普通语句的区别。这就需要考虑对于前导上下文的敏感性，也有几种处理这种问题的方法。例如^操作符，就是一个前导上下文的操作符，它识别紧接其后（即左部）的文字，就如同$操作符识别紧靠着的右部文字一样。也可以扩展邻近的左上下文的概念，以产生一个类似与右上下文的匹配的机制，但因为相应的左上下文常常更早出现，比如就是在一行的开始，所以这样扩展并不是十分有用。

本部分将讨论三种处理不同环境的方法：使用标志的简单方法，这适用于只有少量的规则需要靠环境来决定匹配；在规则中使用开始条件的方法；还有可能的，使得多个词法分析器同时运行的方法。在每一种方法中，都存在这样的规则，它们决定紧接着将分析输入文字的环境是否需要改变，并设置一些参数来表示这种改变。这种参数能被用户定义的操作代码直接显式访问；这种设立标志的方法是处理这种问题的最简单的方法，且与 Lex 系统无关。当然，使得 Lex 存储这些标志，做为规则开始的条件，会更加方便。任意一条规则都可以和一个开始条件相联系。仅仅当 Lex 处于这种条件下时规则才会被正确识别。当前的开始条件也可能随时被改变。最后，如果为不同环境设置的规则之间没有太明显的相似性，编写几个不同的词法分析器，根据需要调整使用，能够最清楚地达成区分的目的。

考虑如下的问题：将输入输出，将其中以字母 a 开头的每行中的单词 magic 改写为 first，将以字母 b 开头的每行中的单词 magic 改写为 second，将以字母 c 开头的每行中的单词 magic 改写为 third。其它的单词和行保留不变。

解决这个问题的最简单的方法是使用标志，如下简单的规则组即足够了：

```
        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;
        }
        }
```

使用开始条件方法处理同样的问题，先需要在 Lex 的定义部分如下引入开始条件

```
%Start   name1 name2 ...
```

其中开始条件可以以任意次序出现。单词 Start 可以简写为 s 或者 S。可以在规则的开始用<>括起来开始条件来引用之：

```
<name1> expression
```

上面规则就表示 Lex 仅在开始条件 name1 时才会匹配这条规则。进入一个开始条件，可以通过执行如下的操作语句

```
BEGIN name1;
```

就将开始条件改为了 name1。回复正常状态可以用

```
BEGIN 0;
```

复位到 Lex 自动解释器的初始状态。一条规则可以被很多个开始状态激活：书写合法的前缀式即可。没有以<>前缀开始的规则总是处于激活状态的。

于是上面的那个例子，可以这样改写：

```
%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");
```

以上规则的逻辑和刚才解决问题的逻辑一摸一样，只不过 Lex 做了本来用户代码做的保存修改标志的工作。

11.字符集（Character Set）

产生的程序仅仅通过 input，output 和 unput 三个过程来处理字符的输入/输出。因此，Lex 和其调用的返回 yytext 值得程序遵循这三个过程中所提供的字符表现方法。内部实现中，字符使用小整数来表示，如果使用标准库的话，这个小整数等于宿主计算机中表示字符的字节模式的整数值。一般的，字母 a 就是用字符

常量'a'来表示的。如果通过提供转译字符的输入/输出过程而使得这种表示变化了，必须以转译表的形式通知 Lex。转译表必须定义在定义部分，使用内容为``%T''的行括起来。表中则含有如下形式的行

{integer} {character string}

表示与相应字符相联系的值。因此，如下的例子

```
%T
 1     Aa
 2     Bb
...
26     Zz
27     \n
28     +
29     -
30     0
31     1
...
39     9
%T
```

Sample character table.

将大小写的字母一起映射为整数 1 到 26，换航为 27，＝为 28－为 29，数字则从 30 到 39。注意换行符使用的转义方式。如果提供了这样的转译表，那么无论在规则中或者在其它的输入中出现的字符都必须是在该表中出现的。不能有赋为 0 值的字符，也能有值超过硬件字符集大小的字符。

12.源码格式总结(Summary of Source Format)

Lex 源码的一般格式是：

```
{定义（definitions）}
%%
{规则（rules）}
%%
{用户子程序（user subroutines）}
```

定义部分包含如下内容

1)定义，格式是``名字（name）空白符 翻译（translation）''

2)包含进来的代码，格式是``空白符 代码（code）''

3)包含进来的代码，格式是

```
%{
代码（code）
%}
```

4)开始条件，格式是

%S name1 name2 ...

5)字符集表，格式是

%T
数字（number）空白符 单个字符组成的串（character-string）
...
%T

6)自定义内部数组大小，格式是

%x nnn

nnn 是一个十进制整数，表示数组的大小；x 可以是如下的参数之一：

Letter	Parameter
p	positions
n	states
e	tree nodes
a	transitions
k	packed character classes
o	output array size

规则部分的每行都是``表达式（expression） 操作代码（action）''的格式，操作代码可以在大括号的包围下延续多行。

Lex 中正则表达式使用如下的操作符：

x	字符"x"
"x"	"x",即使 x 也是操作符。
\x	"x",即使 x 也是操作符。
[xy]	字符 x 或者 y。
[x-z]	字符 x, y 或者 z。
[^x]	除 x 以外的任意字符。
.	除换行符以外任意字符。
^x	行首出现字符 x。
<y>x	Lex 处于开始条件 y 下，字符 x。
x$	行尾出现字符 x。
x?	可选字符 x。
x*	0,1,2, ... 个字符 x。
x+	1,2,3, ... 个字符 x。
x|y	一个 x 或者一个 y。
(x)	一个 x。
x/y	一个 x，但是必须有一个 y 紧跟这个 x。
{xx}	定义部分定义的 xx 的翻译。
x{m,n}	m 到 n 个字符 x。

13. 警告和错误（Caveats and Bugs）

存在一些十分病态的表达式，当被转换到某些机器上时，可能使得规则表变得很长；幸运的是，它们非常稀少。

REJECT 操作不会重新扫描输入；它记录下前面扫描的结果。这就意味着如果一条含有拖尾要求的规则被匹配，且 REJECT 被执行后，用户将不能使用 unput 函数来改变接下来输入流的字符。这仅仅会给用户定义处理仍未处理的输入（the not-yet-processed input）带来限制。

14.致谢（Acknowledgments）

最应该先特别提出的是，Yacc 规划了 Lex 的外部接口，及内部使用的是 Aho 的字符串匹配算法。因此，S. C. Johnson 和 A. V. Aho 都是 Lex 大部分内容及调试 debugger）的早起发明者。对他们我们无限感谢。

现在版本的 Lex 代码是 Eric Schmidt 设计，编写和调试的。

15.参考文献（References）

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.