flex

fast lexical analyzer generator

名称 NAME

fex-快速的词法分析程序产生器

大纲 SYNOPSIS

flex [-bcdfhilnpstvwBFILTV78+? -C[aefFmr] -ooutput -Ppre-fix -Sskeleton] [--help --version] [filename ...]

概述 OVERVIEW

这个手册描述 flex，一个通过在文本上进行模式匹配并生产程序的工具。这个手册包括教程和参考两部分。

描述 DESCRIPTION

flex 是一个用来生成扫描器的工具，生成扫描器可以识别文本中的词汇模式。flex 从输入文件中或是从标准输入设备中（如果没给出输入文件名）读取信息来生成一个扫描器。信息以正则表达式和 C 代码对的形式组成，这种形式称为规则（rule）。flex 生成 C 源代码文件，lex.yy.c，其中定义了一个例程 yylex()。这个文件通过编译，并用-lfl 链接生成可执行文件。当可执行文件被执行时，它分析输入中可能存在的符合正则表达的内容。当找到任何一个与正则表达式相匹配内容时，相应的 C 代码将被执行。

一些简单的例子 SOME SIMPLE EXAMPLES

首先，看一些关于如何使用 flex 的简单例子。下面这个 flex 的输入说明当遇到一个字符串"username"时，会用用户的登录名来代替。

```
%option main  /* 生成 main 函数 */
%option outfile="t1.c"  /* 等同于命令行参数 -ot1.c */
%%
username printf("%s", getlogin()); /* username 是模式，后面跟着空白符和动作 */
```

```
*** 保存为 t1.lex
*** flex -B -L -l -ot1.c t1.lex
*** gcc t1.c -o t1.exe
```

缺省情况下，flex 扫描器把任何不匹配的文件拷贝到输出设备，所以这个扫描器的实际效果是拷贝它的输入文件随同字符串"username"到输出设备。这里的输入，只包含了一个规则。"username"是模式而"printf"是动作。 "%%"标识了规则的开始。

这里还有个简单的例子：

```
%option noyywrap
    int num_lines = 0, num_chars = 0; /* 注意行开头的缩进，否则得用%{%}括起来 */
```

```
%%
\n      ++num_lines; ++num_chars;
.       ++num_chars;

%%
main()
{
    yylex();
     printf( "# of lines = %d, # of chars = %d\n", num_lines, num_chars );
}
```

这个扫描器统计了输入的字符个数和行数（它在最后显示了这些数目）。第一行定义了两个全局变量"num_lines"和"num_chars"，这两个都被定义在第 2 个%%后的 yylex()和 main()使用。这里有两个规则，一个匹配新的一行（"\n"）并增加行的计数和字符的个数，另一个匹配任何不是新行的字符（在正则表达式中用"."表示）。

另一个稍微有点难度的例子。

```
/* scanner for a toy Pascal-like language */
%option noyywrap
%{
/* need this for the call to atof() below */
#include <math.h>
%}

DIGIT    [0-9]
ID       [a-z][a-z0-9]*

%%

{DIGIT}+    {
            printf( "An integer: %s (%d)\n", yytext, atoi( yytext ) );
            }

{DIGIT}+"."{DIGIT}*  {
                     printf( "A float: %s (%g)\n", yytext,
                     atof( yytext ) );
                     }

if|then|begin|end|procedure|function {
                     printf( "A keyword: %s\n", yytext );
                     }

{ID}                 printf( "An identifier: %s\n", yytext );
```

```
"+"|"-"|"*"|"/"            printf( "An operator: %s\n", yytext );

"{"[^}\n]*"}"           /* eat up one-line comments */

[ \t\n]+                /* eat up whitespace */

.                    printf( "Unrecognized character: %s\n", yytext );

%%

main( argc, argv )
    int argc;
    char **argv;
{
    ++argv, --argc;  /* skip over program name */
    if ( argc > 0 )
      yyin = fopen( argv[0], "r" );
    else
      yyin = stdin;
    yylex();
}
```

这是一个类似 Pascal 语言扫描器的简化版。它定义了 token 的不同类型和显示它遇到了什么。下一节讲解这个例子的细节。

输入文件的格式 FORMAT OF THE INPUT FILE

flex 的输入文件由三段组成，用一行中只有%%来分隔。

定义;definition
%%
规则;rules
%%
用户代码;code

定义段包含了简单名称的声明（这些声明可以简化扫描器的说明）和开始条件（这个在后面的章节中讲解）

名称的定义有如下的格式：
名称 定义
名称由字母或下划线开头，后跟任意字符或数字和破折号。定义是跟在名称后面，第一个不是空白符开始，直到一行结束。名称的定义可以在后面相关内容中被引用，使用"{名称}"，这个引用将会被扩展成"(定义)"。 如：

```
DIGIT    [0-9]
ID       [a-z][a-z0-9]*
```

定义了"DIGIT"为一个匹配一个数字的正则表达式，"ID"定义为匹配一个以字母开头后面跟零个或多个字母或数字的字符串的正则表达式。随后，可以引用它们，如

```
{DIGIT}+"."{DIGIT}*
```

上面这个和下面这一个是一样的

```
([0-9])+"."([0-9])*
```

匹配以一个数字开头后跟一个"."再跟零个或多个数字的字符串

flex 的输入规则段部分包含了一组以如下形式组成的规则
模式 动作

这里的模式不能缩进的，并且动作是在同一行上跟在模式后面。可以看下面更多关于模式和动作的描述。

最后的用户代码只是简单的拷贝到 lex.yy.c。这个和扫描器组成一起，调用扫描器或是被扫描器调用。如果被省略，那么输入文件中的第 2 个%%也可以被省略。

在定义和规则段，任何缩进的文本和以%{和%}包裹起来的字符将被逐字的拷贝到输出文件中（%{%}被移除）。%{和}%在一行的开头，不能缩进。

在规则段，出现在第一个规则之前的任何缩进或是%{%}文本一般用于定义变量，这些变量是扫描程序和在声明之后的代码（由扫描程序启动）使用的本地变量。在规则段的其他缩进的和%{%}文本还是拷贝到输出文件，但它们的含义不明确并可能会产生编译错误（这个特性在 POSIX 兼容部分讨论，查看后面关于其他这样的特性说明）。

在定义段（不是在规则段），一个没缩进的注释（如以"/*"开始）到下一个"*/"之间的文本也被逐字的拷贝到输出设备。

模式 PATTERNS

在输入文件中的模式是写扩展的的正则表达式写成的。如：

X	匹配字符'X'
.	任何字符除了换行符
[xyz]	一个字符类别，匹配任何一个，'x'或'y'或'z'
[abj-oZ]	一个包含一个范围的字符类别，匹配'a'、'b'、或是从'j'到'o'的，或是'z'任何字符
[^A-Z]	一个无效的字符类别，任何不在类别中的字符。
[^A-Z\n]	任何不是大写的和换行符
r*	零个或多个 r，r 可以是任何正则表达式
r+	一个或多个 r
r?	零个或一个 r，意思是可选的 r
r{2,5}	2 个到 5 个 r
r{2,}	2 个或更多
r{4}	只有 4 个 r
{name}	一个名称的扩展（上面已经说明了）
"[xyz]\"foo"	一个原义字符串：[xyz]"foo
\X	如果 X 是一个'a','b','f','n','r','t'或'v'那么 ANSI-C 会解释为\X，否则就

是表示字符 X（如'*'的转义操作）

\0　　一个 NUL 字符（ASCII 代码为 0）

\123　　八进制数 123

\x2a　　十六进制 2a

(r)　　匹配 r;圆括号用来取得优先权（看下面说明）

rs　　正则表达式 r 后面跟着正则表达式，称为连接表达式

r|s　　r 和 s 两者选一个

r/s　　r 后面中跟着 s，文本必须匹配 rs，但 action 看到的只是 r。（有些情况下这个匹配并不完全正确，这些情况在 BUG 部分有说明）

^r　　一行以 r 开头（如刚开始扫描或是紧跟在新的一行后面）

r$　　r 后面是行的结束。相当于"r/\n"。注意，flex 的新一行的概念是指 C 编译器用于解释 flex 的"\n"；在特殊情况下，在某些 DOS 系统，你自己必须过从输入中滤掉"\r"或是显示的使用 r/\r\n。

<s>r　　当满足开始条件 s，匹配 r（看下面有关于开始条件的讨论）

<s1,s2,s3>r　　和上一条一件，只是满足三个条件中的任意一个

<*>r　　任何条件，甚至是单独的一个

<<EOF>>　　文件的结束

<s1,s2><<EOF>>　满足任选条件的文件结束

注意在一个字符类别中，所有正则表达式操作失去它们特殊的意义，除非转义字符'\'和字符类别操作'-',']'和在类别的开始'^'。

上面列出的正则表达式按优先级别组合在一起，最上面的优先级最高，最下边的最低。 同一组的优先级相同。如

```
foo|bar*
```

等同于

```
(foo)|(ba(r*))
```

由于'*'的操作符优先级比连接高，连接又比选择('|')高。所以这个模式匹配字符串"foo"或者字符串"ba"后面跟着零个或多个 r。匹配"foo"或者零个或多个"bar"，可用

```
foo|(bar)*
```

如果匹配零个或多个"foo"或者"bar"，可用

```
(foo|bar)*
```

除了字符和字符范围，字符类别还可以包含表达式。这些表达式用[:和:]做为分界。有效的表达式有：

```
[:alnum:] [:alpha:] [:blank:]
[:cntrl:] [:digit:] [:graph:]
[:lower:] [:print:] [:punct:]
[:space:] [:upper:] [:xdigit:]
```

这些表达式全部被指派为一个字符串，对应于 C 的 isXXX 函数。例如，[:alnum:]指派那些能让 isalnum()返回 true 结果的字符，包括任意的字母的数字。一些系统不提供 isblank()，所以 flex 定义[:blank:]作为空白和制表符。

例如，下面的字符类别效果是一样的：

```
[[:alnum:]]
```

```
[[:alpha:][:digit:]
[[:alpha:]0-9]
[a-zA-Z0-9]
```

如果你的扫描器不是大小写敏感的，那么[:upper:]和[:lower:]等同于[:alpha:]。

模式需要注意的几点：
一就象[^A-Z]能够匹配"\n"（或是一个相同效果的转义序列）一样，非字符是一个明确出现在非字符类别中的字符，如（[^A-Z\n]）。这并不象其他的正则表达式对非字符的处理，这是由于历史原因造成的。匹配新一行的模式[^"]同样可以匹配所有输入的字符除非遇到另一个引号。
一一个规则最多可以拥有一个后续('/'操作或'$')。开始条件'^'和"<<EOF>>"只能出现在模式的开头，'/'和'$'也不能被括在圆括号里。 不出现在规则开关的'^'和不出现在规则结尾的'$'将被认为是普通字符。

下面的例子是非法的：

```
foo/bar$
<scl>foo<sc2>bar
```

注意第一行可以写成"foo/bar\n"。

下面的'$'和'^'将被认作是普通字符：

```
foo|(bar$)
foo|^bar
```

如果只是想得到"foo"或者是"bar"后面跟着一个新行符，那么可以用下面这一个（"|"的特别行为在下面有说明）：

```
foo       |
bar$      /* action goes here */
```

同样的技巧，这可以匹配 foo 和出现在行开头的 bar。

输入是如何被匹配的 HOW THE INPUT IS MATCHED

当生成的扫描器开始运行时，它通过搜索匹配它的任何一个模式的字符串来分析输入的文件。如果找到多个匹配，那么会选取一个匹配最多文本的模式（对于上下文模式，这包括后续部分的长度，即使这会返回到输入？）。如果多个模式匹配的长度相同，那么会选择最先出现在 flex 输入文件的模式。

一当确实了模式，那么相对应的匹配称为 token 可以从全局的字符指针 yytext 获得，它的长度是全局整形变量的 yyleng。接着会执行对应模式的动作（接下来在 actions 部分作详细说明），然后把再扫描剩余的文本寻找另一个匹配。

如果一个匹配也没有发现，那么缺省的规则将被执行：将接下来从输入中得到的字符作为匹配的字符并拷贝到标准输出设备。所以，一个最简单的合法的 flex 输入是：

```
%%
```

这个将生产一个扫描器，它只是简单的把输入的一个一个字符拷贝到输出上。

注意 yytext 可以用两种方法定义：作为字符指针或是字符数组。你可以通过在 flex 的输入的第一段（定义段）中使用特殊的指令 %pointer 或者 %array 来控制 flex 选择哪一种定义。缺省情况下是 %pointer，除非你使用了 -l lex 兼容选项，这种情况下， yytext 将是字符数组。使用 %pointer 的好处在于可让扫描更快并且在扫描大的 token 时不会造成缓冲溢出（除非你再也没有内存可以使用了）。不好的地方在于你控制那么能够修改 yytext 的动作（action）（看下一章），并且使用 unput() 函数来释放 yytext 的内容，当在不同 lex 版本间移植时，这会是很头痛的事情。

使用%array 的好处是你可以尽情地修改 yytext，调用 unput() 并不会释放 yytext（看下面）。还有，现在的 lex 程序可能会通过如下的形式来外部使用 yytext：

```
extern char yytext[];
```

如果用到%pointer，那么这个声明是错误的，%array 则正确。

%array 将 yytex 定义成一个拥有 YYLMAX 个字符的数组，缺省情况下，YYLMAX 是个相当大的数值。你可以在 flex 输入的第一段中使用#define 把 YYLMAX 定义成别的数据。如上面提到的一样，%pointer 使 yytext 动态增长以容纳下大型的 token，这意味着使用%pointer 的扫描器可以扫描大型的 token（如，匹配整个注释块），记得每一次扫描器都必须重置 yytex 大小，还得重头扫描一遍 token，所以匹配这样的 token 是比较慢的。如果 unput() 使得太多文本被入到 yytext 中，则 yytext 不再自动增长，取而代之的是，产生一个 run-time 错误。

另外需要注意的是，在 C++扫描器中你不能使用%array（查看下面 C++选项部分）。

动作 ACTIONS

在规则 rule 中的每一个模式有一个与之相对应的 action，这个 action 可以是任意 C 表达式。模式以第一个非转义的空白符结束；行里剩下的部分就是它的 action。如果 action 是空的，那么模式被匹配时，输入的 token 只是简单的被扔掉。例如，这里有一个程序，它删除了所有输入中的"zap me"：

```
%%
"zap me"
```

（使用缺省规则，这将拷贝输入中的所有其他字符到输出设备）

这里有一个程序，它把多个空白符和制表符压缩成一个空白符，并且扔掉所有在行尾的所有空白符：

```
%%
[ \t]+      putchar(' ');
[ \t]+$     /* ignore this token */
```

如果 action 中包含了'{'，那么 action 会一直延展，直到下一个配对的'}'被找到。flex 了解 C 字符串和注释中的花括号的作用，不会搞错。flex 还允许 action 以%{开始，以%}结束（不管 action 的原始花括号）。

一个只有单独一个'|'字符的 action 表示这个 action 如同下一个规则 rule 中的 action。下面有说明。

actions 可以包含任何 C 代码，包括 return 表达式直接从 yylex() 调用返回一个值。每次 yylex() 被调用，它继续处理剩下的 token 直到它文件的结尾或是一个 return 的返回。

在%pointer，action 可以随意修改 yytex 除了延长它（加些字符到它后面－这样会覆盖输入流中的字符）。在%array 中，没这样的限制，可以随意修改。

action 可以修改 yyleng，还可以作用 yymore()。（下面有说明）

这里有一些可以被包括在一个 action 中的特殊指令：

－ECHO 拷贝 yytext 到扫描器输出。

－BEGIN 后面跟着开始条件的名称，开始条件把扫描器放在相应的开始条件中。（下面有说明）？

一REJECT 命令扫描器继续查找第二个同样能匹配输入文件的规则（或是一个输入的前缀）。如上面在"How the Input is Matched"中说明的，那些被选择的规则和 yytext 和 yyleng 能被适当的设置。这可能会是在 flex 输入文件后面出现的贪婪匹配，或是一个匹配更少文本的模式。例如，下面例子会统计输入中的字数并在遇到任何一个"frob"时调用 special()。

```
    int word_count = 0;
%%
frob       special(); REJECT;
[^ \t\n]+     ++word_count;
```

如果不使用 REJECT，在输入中遇到的任何"frob"不会被统计成一个字，因为扫描器对一个 token 只执行一个 action。

允许使用多个 REJECT，每一个会搜索当前活动的规则的下一个更好的匹配。例如，当下面这个扫描器扫描 token"abcd"时，它会把"abcdabcdaba"写到输出设备上：

```
%%
a       |
ab      |
abc     |
abcd       ECHO; REJECT;
.|\n       /* eat up any unmatched charactor */
```

因为使用了特殊的 action'|'，前三个规则共享第四个 action。REJECT 是一个很耗性能的扫描器特性；如果它被在扫描器的每一个 action，那么会降低扫描匹配的速度。还有，REJECT 不能和-Cf 或是-CF 选项合用。（下面有说明）

注意与其他特殊 action 不一样的地方还有 REJECT 是一个分支；在 action 中紧跟其后代码并不会马上得到执行。

yymore() 告诉扫描器下一次它匹配一个规则，相应的 token 应当追加到当前的 yytext 而不是替换它。例如，给一个输入"mega-kludge"，下面的扫描器将输出"mega-mega-kludge"到输出设备上：

```
%%
mega-      ECHO; yymore();
kludge     ECHO;
```

第一个"mega-"被匹配并且回显到输出设备上。接着"kludge"被匹配，但前一个"mega-"仍然放在 yytext 的开头部分，所以"kludge"的实际回显将是"mega-kludge"。

使用 yymore() 需要注意两点。第一，yymore() 依赖反映当前 token 长度的 yyleng 值，所以在使用 yymore()，你不能修改 yyleng。第二，出现在扫描器 action 中的 yymore() 不可避免的会损失一定的扫描器匹配速度。

yyless(n)退回除了当前 token 前 n 个字符所有字符到输入流中，在流中，当扫描器查找下一个匹配时，这些字符将会被重新扫描。yytext 和 yyleng 会被适当的调整好（如，yyleng 会等于 n）。例如，在输入"foobar"后，下面将会显示"foobarbar"：

```
%%
foobar    ECHO; yyless(3);
[a-z]+    ECHO;
```

传递 0 给 yyless 将会导致所有的当前输入被会重新扫描一遍。除非你改变了扫描器如何处理随后的输入（例如使用 BEGIN），这将导致一个无限循环。

注意，yyless 是一个宏，只能在 flex 输入文件中使用，不能在其他源文件中使用。

－unput(c)输出字符 c 到输入流中。这会是下一次被扫描到的字符。下面的 action 获取得前的 token 并且使得它被圆括号括起来重新扫描。

```
    {
    int i;
    /* Copy yytext because unput() trashes yytext */
    char *yycopy = strdup(yytext);
    unput(')');
    for(i=yyleng-1; i>=0; --i)
        unput(yycopy[i]);
    unput('(');
    free(yycopy);
    }
```

注意到由于每一个 unput()把给出的字符回放在输入流的开始，所以回放字符串时要从后到前次序颠倒。

使用 unput()会存在一个问题，那就是当你作用%pointer 时，使用 unput()会破坏掉 yytext，每用一次，就吃掉 yytext 的一个字符，从最右到最左。如果你需要在使用 unput 后还仍然保存 yytext 的值，那么你可以先拷贝它，或是改用%array。（参考 How The Input Is Matched）

最后，注意你不能在输入流里放回一个 EOF 来表示文件的结束。

－input()从输入流中读取下一个字符。例如，下面的演示吃掉 C 的注释：

```
%%
"/*"    {
        register int c;
        for(;;)
        {
            while ((c=input())!='*' && c!= EOF)
            ; /* eat up text of comment */
            if (c=='*')
            {
                while((c=input()) == '*')
```

```
                ;
            if(c=='/')
                break;  /* found the end */
        }
        if(c==EOF)
        {
            error("EOF in comment");
            break;
        }
    }
}
```

（注意如果扫描器是用 C++编译的，那么 input() 被替换成 yyinput()，避免造成 C++流和输入流名字的冲突？）

－YY_FLUSH_BUFFER 扫清扫描器的内部缓冲，当下一次扫描器准备匹配一个 token 时，它先会使用 YY_INPUT 重填缓冲（看下面的关于扫描器的产生）。这个 action 是一个比 yy_flush_buffer() 函数更特殊，在下面 Multiple Input Buffers 节讨论。

－yyterminate() 在 action 里能被用做的返回语句。它结束扫描过程并返还一个 0 给扫描器的调用者，表示所有事情都搞定了。当文件结束时，yyterminate() 默认也会被调用。它是一个宏，可以被重定义。

扫描器的生产 THE GENERATED SCANNER

flex 的输出是一个文件 lex.yy.c，它包含了扫描函数 yylex()。yylex() 使用了一些表来匹配 token 和一些辅助例程和宏。缺省情况下，yylex() 定义如下：

```
int yylex()
    {
    ...various definistions and actions in here ...
    }
```

（如果你的环境支持函数原型，那么这会是"int yylex(void)"。）这个定义可以用定义宏"YY_DECL"来更改。例如，你可以用：

```
#define YY_DECL float lexscan(a,b) float a,b;
```

来赋与扫描函数一个 lexscan 的名字，使用两个 float 作为参数，返回一个 float。注意，如果你用 K&Rstyle/non-prototyped 函数定义传递参数给扫描函数，你必须用分号(;)来结束定义。？

不管 yylex() 什么时候被调用，它从全局输入文件 yyin（缺省是 stdin）中扫描 token。它继续直到遇到文件结束（碰到时返回值 0）或是其中一个 action 使用了 return 表达式。

如果遇到文件结束，并且 yyin 不是指向另一个新文件，那接下来的调用结果是未定义的（未知的）。如果 yyin 指向新文件或者 yyrestart() 被调用，那么扫描将在新文件继续。yyrestart() 有一个参数，一个 FILE* 指针（可以是 nil，如果你设置了 YY_INPUT 来获得扫描源而不是 yyin），初始化了 yyin，扫描器就能从其中进行扫描。设置 yyin 为一个新的输入文件或是使用 yyrestart() 来设置新文件从本质上讲并没有什么区别；后者只是为了兼容以前的 flex 版本，而且它能够在扫描的过程中切换扫描输入

文件。使用 yyin 作为参数调用它，它还能扔掉当前的输入的缓冲；但使用 YY_FLUSH_BUFFER 可获得更好的效果（上面有说明）。注意 yyrestart() 并不会重新设置开始条件 INITIAL（看下面 Start Conditions）。

如果是因为在执行一个 action 中使用了 return 语句而使 yylex() 停止扫描，扫描可以从返回的地方被重启。

缺省情况（或是为了更好的效率）扫描器使用了块读取 block-reads 而不是简单的使用 getc() 从 yyin 中获取字符。如何获取可以通过定义宏 YY_INPUT 来控制。YY_INPUTS 的调用序列是 "YY_INPUT(buf,result,max_size)"，max_size 提示 buf 的最大长度，返回读到的字符个数或是常量 YY_NULL（在 Unix 系统返回 0）表示已经结束。默认 YY_INPUT 是从全局文件指针"yyin" 中读取。

一个 YY_INPUT 的定义例子（在 flex 输入文件的定义段）

```
%{
#define YY_INPUT(buf,result,max_size) \
    {\
    int c=getchar(); \
    result=(c==EOF)?YY_NULL:(buf[0]=c,1); \
    }
%}
```

这个定义会改变每一次读取字符时的处理操作。

当扫描器从 YY_INPUT 处接收一个"文件结束符"时，会检查 yywrap() 函数。如果 yywrap() 返回 false（zero），那么扫描器假设函数已经到头了，设置 yyin 指向另一个文件，继续扫描。如果返回 true（non-zero），那么扫描器返回一个 0 给调用它的函数，并终止。注意，在这个的情况下，开始条件并不会更改，没有恢复成 INITIAL。

如果你没有提供你自己的 yywrap() 版本，那么你必须使用 %option noyywrap（这样效果如同 yywrap() 返回 1），同时你必须在链接时使用-lfl 选项去获得处理例程的缺省版本，这个版本始终返回 1。？

扫描器可以使用三种处理例程，从内存缓存到文件：yy_sacn_string()，yy_scan_bytes()，和 yy_scan_buffer()。在 Multiple Input Buffers 节讨论它们的使用。

扫描器把它的 ECHO 写到全局的 yyout（缺省是 stdout），可以通过给它赋一个新的 FILE 指针来重新定义它。

开始条件 START CONDITIONS

flex 提供了一种按条件激活规则 rule 的机制。所有模式以"<sc>"为前缀的 rule 只有在扫描器是在一个名为"sc"的启动条件时才会被激活。例如：

```
<STRING>[^"]* { /* eat up the string body ... */
    ...
    }
```

将只有在扫描器是在"STRING"开始条件中时才会被激活，如

```
<INITIAL,STRING,QUOTE>\.   {/* handle an escape ... */
    ...
    }
```

将只有当当前开始条件是"INITIAL"，"STRING"或"QUOTE"这三种情况之一时才会被激活。

开始条件是在 flex 输入文件的第一段中定义的，使用非缩进的形式，以%s 或%x 开始，后面跟着一串名字。前面的说明开始条件 start condition 开始，后面的说明 start condition 的结束。使用 BEGIN action 可以激活一个开始条件。直到下一个 BEGIN action 被执行，在给出开始条件的 rule 将被激活并且其他给出其他开始条件的 rule 并不会被激活。如果使用的是包含的开始条件，那么其他没有开始条件的 rule 也会被激活。如果使用的是排他的开始条件，那么只有以开始条件修饰的 rule 才会被激活。跟在同一个排他开始条件后的 rule 说明在扫描器中，这些 rule 是独立于 flex 输入中的其他 rule。所以，使用排他的开始条件使得定义一个迷你扫描器变得更简单，迷你扫描器可以扫描部件语法上的结构，这些结构和其他部分有所区别（如注释）。

包含和排他的开始条件之间的区别还有一些模糊，这里有个简单的例子来说明他们之间的这个区别。这些 rule 是：

```
%s example
%%
<example>foo            do_something();
bar                     something_else();
```

和这个是等价的：

```
%x example
%%

<example>foo         do_something();
<INITIAL,example>bar    something_else();
```

当在开始条件例子中，在第 2 个例子中在 bar 模式前面没有<INITIAL,example>修饰，这个模式将不会激活（匹配）。如果我们使用了<example>来修饰 bar，那么它只能在 example 而不是 INITIAL 时被激活，这和第一个例子不同，第一个例子中使用的是包含的开始条件（%s），所以在这两个开始条件中，bar 都会被激活。

还要注意的是，使用<*>指定的特殊的开始条件匹配每一个开始条件。因此，上面的例子也可以这样写：

```
%x example
%%

<example>foo     do_something();

<*>bar           something_else();
```

在开始条件中，缺省 rule（回显所有不能被匹配的字符）保持激活。它和这个是等价的：

```
<*>.|\n     ECHO;
```

只有在没有开始条件的 rule 被激活的地方， BEGIN(0) 返回到原始状态。这个是可以用开始条件 "INITIAL" 来引用的状态，所以 BEGIN(INITIAL) 等同于 BEGIN(0) 。（把开始条件括起来的圆括号不是必须的，但这是一种好的书写风格。）

BEGIN action 还可以被在 rule 段的开头指定一个缩进形式的代码。例如，下面这个会导致扫描器进行一个"SPECIAL" 的开始状态不管 yylex() 是被调用和全局 enter_special 是 true：

```
    int enter_special;
%x  SPECIAL
%%
if(enter_special)
    BEGIN(SPECIAL);
<SPECIAL>blahblahblah
    ...more rules follow...
```

演示如何使用开始条件，这里有一个扫描器，可以对字符串"123.456"提供两种不同的解释。缺省情况它会把字符串当成三个 token，整数"123"，一个点（'.'），和另一个整数"456"。但如果在行文本的前面使用了"expect-floats"，那么它就把字符串当成一个 token，一个浮点数 123.456：

```
%{
    #include <math.h>
%}

%%
expect-floats    BEGIN(expect);

<expect>+"."[0-9]+  {
    printf("found a float, =%f\n",  atof(yytext));
}
<expect>\n  {
        /* that's the end of the line, so
        * we need another "expect-number"
        * before we'll recognize any more
        * numbers
        */
        BEGIN(INITIAL);
        }

[0-9]+ {
        printf("found an integer, = %d\n",
            atoi(yytext));
        }
```

```
"." printf("found a dot\n");
```

这个扫描器可以在维护当前输入行数的同时，识别（或是抛掉）C 注释。

```
%x comment
%%
    int line_num = 1;
"/*"    BEGIN(comment);

<comment>[^*\n]*        /* eat anything that's not a '*' */
<comment>"*"+[^*/\n]*       /* eat up '*'s not followed by '/'s */
<comment>\n             ++line_num;
<comment>"*"+"/"        BEGIN(INITIAL);
```

这个扫描器费了很大劲来让规则 rule 匹配尽可能多的文本。In general, when attempting to write a high-speed scanner try to match as much possible in each rule, as it's a big win.（这一句不会翻@@一般情况下想生产一个让每一个 rule 都能匹配最多的文本又能快速扫描的扫描器是一个挑战）

注意，开始条件的名称是 integer 值，可以当成 integer 来存储。所以，上面的可以用下面这样的形式扩展：

```
%x comment foo
%%
    int line_num = 1;
    int comment_caller;
"/*"    {
    comment_caller = INITIAL;
    BEGIN(comment);
    }
    ...
<foo>"/*" {
    comment_caller = foo;
    BEGIN(comment);
    }
<comment>[^*\n]*        /* eat anything that's not a '*' */
<comment>"*"+[^*/\n]*       /* eat up '*'s not followed by '/'s */
<comment>\n             ++line_num;
<comment>"*"+"/"        BEGIN(comment_caller);
```

你还能用整数值宏 YY_START 访问当前开始条件 start condition。例如，上面对 comment_caller 的赋值可以写成

```
    comment_caller = YY_START;
```

flex 提供 YYSTATE 作为 YY_START 的别名（因为 YY_START 已经被 AT&T lex 使用了）。

注意开始条件并没有拥有它们自己的名字空间；%s 和%x 声明名字和#define 使用了相同的形式。

最后，这里有一个能匹配 C 语法的引号字符串的例子，使用了排他的开始条件，包含了扩展的转义功能（但不检查字符串是否太长）：

```
%x str

%%
    char string_buf[MAX_STR_CONST];
    char *string_buf_ptr;

\"  string_buf_ptr = string_buf; BEGIN(str);

<str>\"    {/* saw closing quote-all done */
    BEGIN(INITIAL);
    *string_buf_ptr = '\0';
    /* return string constant token type and
    * value to parse
    */
    }

<str>\n    {
    /* error - unterminated string constant */
    /* generate error message */
    }

<str>\\[0-7]{1-3} {
    /* octal escape sequence */
    int result;

    (void) sscanf(yytext + 1, "%o", &result);

    if (result > 0xff)
        /* error, constant is out-of-bounds */

    *string_buf_ptr++=result;
    }

<str>\\[0-9]+ {
```

```
    /* generate error - bad escape sequence; something
    * like '\48' or '\0777777'
    */
    }

<str>\\n  *string_buf_ptr++ = '\n';
<str>\\t  *string_buf_ptr++ = '\t';
<str>\\r  *string_buf_ptr++ = '\r';
<str>\\b  *string_buf_ptr++ = '\b';
<str>\\f  *string_buf_ptr++ = '\f';

<str>\\(.|\n)  *string_buf_ptr++ = yytext[1];

<str>[^\\\n\"]+    {
   char *yptr = yytext;
   while ( *yptr )
   *string_buf_ptr++ = *yptr++;
    }
```

通常，象上面的一些例子，你可以把所以有一样开始条件的 rule 写在一起。Flex 通常引入一个叫开始条件范围的概念使得这一切变得简单和清晰。一个开始条件的范围以这个开始：
<SCs>{
这里 SCs 是一个或多个开始条件的列表。在开始条件范围内，每个 rule 自动拥有一个前缀<Scs>，直到另一个匹配最开始的"{"的"}"被找到。 所以，来瞅瞅例子，

```
<ESC>{
    "\\n"   return '\n';
    "\\r"   return '\r';
    "\\f"   return '\f';
    "\\0"   return '\0';
    }
```

这和下面是等价的：

```
<ESC>"\\n"  return '\n';
<ESC>"\\r"  return '\r';
<ESC>"\\f"  return '\f';
<ESC>"\\0"  return '\0';
```

使用开始条件范围的格式看起来更加紧凑。

处理开始条件栈的时候，可以用到三个例程：

```
void yy_push_state(int new_state)
```

压入当前开始条件到开始条件栈的顶部并且转到 new_state，就象你使用了 BEGIN new_state 一样（调用的开始条件名同样也是 integer）。

```
void yy_pop_state()
```
从开始条件栈顶弹出一个开始条件，并通过 BEGIN 切换到这个开始条件。

```
int yy_top_state()
```
返回栈顶开始条件但不改变栈的内容。

开始条件栈自动增长所以没有大小的限制。如果内存用完，程序终止运行。

要使用开始条件栈，你的扫描器还得包含一个%option stack 的指令（下面选项部分有说明）。

多个输入缓存 MULTIPLE INPUT BUFFERS

一些扫描器（如那些支持"include"文件的）需要从多个输入流中读取。Flex 扫描器需要处理大量的缓存，YY_INPUT 对扫描环境相当敏感，不能只是简单的修改它来控制下一次从哪个输入读取东西。只有在扫描器遇到文件结束时，YY_INPUT 才会被调用，当读到象"include"这样要求转到别的输入源的文本时却还要等上好长一段时间才能转。

处理这类问题，flex 提供了一个机制，可以用来创建和切换多个输入缓冲。一个输入缓冲可以这样被创建：

```
YY_BUFFER_STATE yy_create_buffer(FILE *;file, int size)
```

这个使用了一个 FILE 指针和一个 size，创建一个绑定了给定文件的缓冲和存放 size 指定个数的字符足够大的空间。（如果有什么疑问，使用 YY_BUF_SIZE 来指定大小）。这函数返回一个 YY_BUFFER_STATE 句柄，该句柄可以传递给别的例程（看下面）。YY_BUFFER_STATE 是一个指针，指向一个复杂的 yy_buffer_state 结构，所以你可以安全的初始化 YY_BUFFER_STATE 变量为（(YY_BUFFER_STATE)0）如果你喜爱，并且能使用这个结构来正确的定义在源文件中而不是在你的扫描器中的输入缓存。(？很别扭不知道怎么翻，看不懂请参考原文)。注意，调用 yy_create_buffer 时使用的 FILE 指针只被当成 yyin（从 YY_INPUT 可看到）的值来使用；如果你重新定义 YY_INPUT 而不再使用 yyin，那么你可以安全的传递一个 nil 的 FILE 指针给 yy_create_buffer。你可以使用下面的函数来选择一个特殊的缓冲来扫描：

```
void yy_switch_to_buffer(YY_BUFFER_STATE new_buffer)
```

切换扫描器的输入缓存，接下来的 token 将来自新的缓存 new_buffer。注意，yywrap() 可以使用 yy_switch_buffer() 来让扫描继续，而不是打开一个新的文件并把 yyin 指向它。同时注意，通过 yy_switch_to_buffer() 或是 yywrap() 切换输入源并不会改变开始条件。

```
Void yy_delete_buffer(YY_BUFFER_STATE buffer)
```

这个用来取回绑在 buffer 上的存储空间（buffer 可以是 nil，这种情况例程什么也不做）你也可以使用以下函数来清空当前的 buffer 内容：

```
void yy_flush_buffer(YY_BUFFER_STATE buffer)
```

这个函数抛弃 buffer 的内容，所以下一次扫描器要从 buffer 中匹配 token，它必须得用 YY_INPUT

重新填写 buffer。

yy_new_buffer() 是 yy_create_buffer() 的一个别名，为了兼容 C++ 中的用创建动态对象的 new 和 delete 而提供的。

最后，YY_CURRENT_BUFFER 宏返回一个当前 buffer 的 YY_BUFFER_STATE 句柄。

这里有一个例子，使用了这些用于写扩展了 include 文件的扫描器（<<EOF>>特性下面有讨论）的特性:

```
/* the "incl" state is used for picking up the name
* of an include file
*/
%x incl

%{
#define MAX_INCLUDE_DEPTH 10
YY_BUFFER_STATE include_stack[MAX_INCLUDE_DEPTH];
int include_stack_pr = 0;
%}

%%
include           BEGIN(incl);

[a-z]+        ECHO;
[^a-z\n]*\n?      ECHO;

<incl>[ \t]*      /* eat the whitespace */
<incl>[^ \t\n]+      { /* got the include file name */
    if (include_stack_ptr >= MAX_INCLUDE_DEPTH)
    {
        fprintf(stderr, "Includes nested too deeply");
        exit(1);
    }
    include_stack[include_stack_ptr++] =  YY_CURRENT_BUFFER;
    yyin = ;fopen(yytext, "r");

    if (!yyin)  error(...);

    yy_switch_to_buffer(yy_create_buffer(yyin, YY_BUF_SIE));
    BEGIN(INITIAL);
    }
<<EOF>>{
    if (--include_stack_ptr < 0)
    {
        yyterminate();
    }
```

```
    else
    {
        yy_delete_buffer(YY_CURRENT_BUFFER);
        yy_switch_to_buffer(include_stack[include_stack_ptr]);
    }
    }
```

从内存字符串而不是从文件中扫描，可以用到三个例程来建立输入缓存。这几个例程为被扫描的字符串建立输入缓存，并且返回一个对应的 YY_BUFFER_STATE 句柄（你可以在完成处理后用 yy_delete_buffer() 删除掉它）。它们还可用 yy_switch_to_buffer() 切换到新的缓存，让接下来 yylex() 在字符串中开始扫描。

yy_scan_string(const char *str)
扫描一个用 NUL 结束的字符串

yy_scan_bytes(const char *bytes, int len)
从 bytes 指定的位置扫描 len 指定长度的字符（有可能包含 NUL 符号）。

注意这两个函数创建和扫描一个字符串和 byte 组的一个拷贝。（这可能是让人满意的，因为 yylex() 修改了它所扫描的缓存的内容。）你可以使用下边这个函数来避免拷贝操作：

yy_scan_buffer(char *base, yy_size_t size)
这个函数从 base 指定的开始的位置扫描由 size 指定个数的 byte，最后两个 byte 必须是 YY_END_OF_BUFFER_CHAR(ASCII NULL)。最后的这两个 byte 不被扫描；所以扫描的内容包括从 base[0] 到 base[size-2]。

如果你不能用这种方式成功的建立 base（例如你忘记了最后那两个 YY_END_OF_BUFFER_CHAR byte），那么 yy_scan_buffer() 返回一个 nil 指针而不是建立一个新的输入缓存。

yy_size_t 是一个 integer 类型，你可以传递一个表示缓存长度的 integer 表达给它。

文件结束符规则 END-OF-FILE RULES

特殊 rule "<<EOF>>" 表示当遇到一个文件结束符和 yywrap() 返回非零结果（表示再没有文件可处理）时应该采取什么 action。这个 action 必须完成以下四个事情之一：
一给 yyin 赋一个新的输入文件（上一个版本的 flex，在赋值后你必须调用特殊的 action YY_NEW_FILE； 这里再也不需要这么做了）；
一执行一个返回语句；
一执行特殊的 yyterminate() 动作；
一或者，就象上面例子中提到的那样，使用 yy_switch_to_buffer()，切换到一个新的输入缓存。

<<EOF>>规则不能在其他的模式中使用；其他的规则只能以一串开始条件来修饰。如果给出一个没有修饰的<<EOF>>规则，它可以应用到所有那些没有<<EOF>>action 的开始条件中(?)。只有 initial 开始条件指定<<EOF>>规则，可用

```
<INITIAL><<EOF>>
```

这些规则在捕获一些东西如找不到结束符的注释的过程中特别有用。例如：

```
%x  quote
%%

    ...other  rules  for  dealing  with  quotes...

<quote><<EOF>>       {
       error("unterminated  quote");
       yyterminate();
       }

<<EOF>>       {
       if  (*++filelist)
           yyin=fopen(*filelist,  "r");
       else
           yyterminate();
       }
```

一些宏 MISCELLANEOUS MACROS

可以定义宏 YY_USER_ACTION 来指供一个 action，这个 action 总是在被匹配的规则 rule 的 action 被执行前执行。例如，可以通过#define 定义把 yytext 转成小写的例程。当YY_USER_ACTION 被调用时，变量yy_act 给出被匹配的 rule 编号（rule 被编号，从1 开始）。假设你想知道每一个 rule 被匹配的次数。这也许是个办法：

```
#define  YY_USER_ACTION  ++ctr[yy_act]
```

这里的 ctr 是一个保存了不同 rule 计数的数组。注意宏 YY_NUM_RULES 给出了全部规则的数目（包括缺少的，即使你使用了-s），所以 ctr 的正常声明应当是：

```
int  ctr[YY_NUM_RULES];
```

宏 YY_USER_INIT 可以定义一个 action，这个 action 始终在第一个扫描之前被执行（在扫描器的内部初始化完成之前）。例如你可以用它来调用一个例程，从一个数据表中读取数据或是打开一个日志文件。

宏 yy_set_interactive(is_interactive) 能被用在控制当前缓存是否是可交互的。一个可交互缓存处理过程明显缓慢，但扫描器的输入源需要交互来避免由于等待填充缓存时带来的问题，必须要用到交互缓存（看下面关于-I 标志的讨论）。使用一个非零的值来调用这个宏表示缓存是交互的，零值表示不可交互的。注意到使用这个宏将覆盖 %option always-interactive 或者是 %option never-interactive（看下面选项 option 部分）。在扫描缓存之前调用 yy_set_interactive() 表示缓存

交互否则不是。

用一个非零值调用宏 yy_set_bool(at_bol) 可让当前缓存的扫描环境认为下一个 token 的匹配是在行的开头发生的。(? 原文看不懂，如果觉得翻得不对，请参考原文)

如果当前缓存中的下一个 token 的模式激活了'^'，那么宏 YY_AT_BOL() 返回 true，否则返回 false。

在生成扫描器时，所有 action 都被集中到一个大的 switch 表达式中并且用 YY_BREAK 分隔开，这个 YY_BREAK 可以被重定义。缺省情况下，YY_BREAK 只是一个简单的"break"把每一个 rule 的 action 从其他 rule 分隔开。例如，C++用户可用#define YY_BREAK 重定义成什么事也不做来避免由于 rule 的 action 以"return"结束而引发的表达式无法到达的警告。重定义要注意每一个 rule 是以"break"或是以"return"结束的。(? 看不明白说的，估计翻出来的东西也难看明白，请参考原文)

用户可使用的值 VALUES AVAILABLE TO THE USER

这一节总结各类用户可以在 rule 的 action 中使用的值。
－char *yytext 保存了当前 token 的文本。它可以被修改但不能更改增大文本的长度(就是你不能再追加上字符)。

如果特殊的指令 %array 出现在扫描器的描述第一段中，那么 yytext 就被定义成 yytext[YYLMAX]，这里 YYLMAX 是一个宏，如果你不喜欢它缺省值(通常是 8KB)，你可以在第一段中重定义它。使用%array 也许会让扫描器慢一些，但就可以在 yytext 上使用 input() 和 unput()，当 yytext 是一个字符指针时，这两个调用就有可能会释放它。相对%array，缺省的选项是%pointer。

当使用-+标志生成 C++ 扫描器时，你不能使用%array。

－int yyleng 保存了当前 token 的长度。

－FILE *yyin 是 flex 读取文本的缺省文件。它可以被定义，但这样做只有在扫描还未开始或是遇到了 EOF 了之后。在扫描的过程中间改变它会造成一些不可预知的后果，因为 flex 缓存了它的输入；可以选用 yyrestart() 来实现这个目的。一旦遇到文件结束符 end-of-file 扫描停止了，你可以给 yyin 赋一个新的输入文件并重新调用扫描器继续扫描。

－void yyrestart(FILE *new_file) 可以被调用，让 yyin 指向新的输入文件。这个切换过程是立即发生的，所以之前已经缓存起来的东西将丢失。注意，使用 yyin 作为参数来调用 yyrestart 会清空当前缓冲中的东西并从原来的文件继续扫描。

－FILE *yyout 动作 action 中的 ECHO 把它当作输入文件。用户可以给它重新赋值。

－YY_CURRENT_BUFFER 返回当前缓存的 YY_BUFFER_STATE 句柄。

－YY_START 返回一个整形数值，对应着当前的当前的开始条件 start condition。在接下来的过程中你可以和 BEGIN 一起使用这个值返回到这个开始条件。

与 YACC 的使用 INTERFACING WITH YACC

flex 的主要应用是与语法分析产生器 YACC 的使用。yacc 分析程序调用例程 yylex() 来获取想要的 token。yylex() 被设想能从全局的 yylval 中返回下一个 token 的类型。和 yacc 一起使用 flex，有一个-d 的选项让 yacc 生成一个 y.tab.h 文件，里面包含了所有的从 yacc 输入的 token 的定义。这个文件接下来被 include 在 flex 的扫描器中。例如，如果有一个 token 叫"TOK_NUMBER"，扫描器有一部分看起来就象这个：

```
%{
    #include "y.tab.h"
%}

%%

[0-9]+      yylval = atoi(yytext); return TOK_NUMBER;
```

选项 OPTIONS

flex 有下面这些选项：
-b 生成一个备份信息到 lex.backup。这是一个需要备份（backing-up）起来的扫描器状态和输入字符。通过添加一个 rule 可以移除备份状态。如果所有的备份状态被移除，并且使用了-Cf 或是-CF，那么扫描器会跑得更快一些（看-p 标志）。只有那些想要挤压 squeeze 每一个超出扫描器的终止周期 last cycle 的用户才需要考虑这个选项（看下边 Performance Considerations）。（？不知道在说什么）

-c 一个什么也不做，只为了兼容 POSIX。

-d 让生成的扫描器运行在 debug 模式。一当一个模式被识别和全局 yy_flex_debug 不为零（缺省情况下是零），那么扫描器会向 stderr 输出一行，形式如下

```
    --accepting rule at line 53("the matched text")
```

行数说明了 rule 在扫描器定义文件（flex 输入文件）中的位置。当扫描器做备份（？），接受缺省rule，到达输入缓存的结尾（或是遇到一个 NUL；在这一点上，这两者看起来是一样的），或是遇到文件结束符，类似的信息都会被生成。

-f 指示生成最快的扫描器。不做表格压缩，不考虑 stdio。结果是扫描器变得很大但是很快。这个选项和-Cfr 是等价的（看下面）。

-h 生成一个 flex 选项的"help"总结到 stdout 然后退出。-?和--help 跟-h 是同义的。

-i 指示 flex 生成一个大小写不敏感的扫描器。在 flex 输入文件模式中的大小写将被忽略，输入文件中的 token 被匹配时不管他们的大小写。从 yytext 中得到的匹配文本将保留原来的大小写格式（？it will not be folded 不会被修改）。

-l 开启与原始的 AT&T lex 最大的兼容。注意，这并意味的全部兼容。使用这个选项会损失一定的性能而且不能同时和-+，-f，-F，-Cf 或-CF 选项一起使用。查看下边的"Incompatibilities With Lex And POSIX"一节能知道兼容方面的更多细节。这个选项还导致在生成扫描器时宏

YY_FLEX_LEX_COMPAT 被#define。

-n 又是一个什么也不做的，不再被使用，出现只是为了兼容 POSIX。

-p 生成一个性能报告，输出到 stderr。报告里是那一些会对生成的扫描器造成严重的性能影响的 flex 输入文件特性的说明。如果你使用了两次这个标志，你还能得到一个造成较小性能损失的相关特性说明。

注意，REJECT，%option yylineno 和上下文变量的使用会造成性能上的损失；使用 yymore()，^操作符和-I 标志会造成较小的性能损失。

-s 让把不匹配的输入回显到输出设备上的默认 rule 不起作用。如果扫描器遇到不能匹配任何 rule 的输入，它会终止并返回一个错误。在查找扫描器的规则集漏洞时，这个选项非常有用。

-t 指示 flex 把它生成的扫描器输出到标准输出设备上，而不是生成 lex.yy.c。

-v 指定 flex 把它生成的扫描器的统计概要输出到 stderr 上。大部分的统计对于一般用户用处不大，但第一行标明了 flex 的版本（如同用-V 报告的一样），接下来一行是生成扫描器的时使用的标志，包含那么默认被使用的标志。

-w 不让生成警告消息。

-B 指示 flex 生成一个批处理扫描器，相反交互式的扫描器用标志-I（下面有说明）。一般情况下，如果你确定生成的扫描器不是交互式的，那么你就可以使用标志-B，而且你还能从中得到一点点性能的提升。如果你不是为了那点性能，你应当使用-Cf 或者是-CF（下面讨论），它们自动开启-B 标志。

-F 指定使用扫描器的快表（？什么东东），旁路 stdio。这样使用和-f 有一样有速度，并且一些模式集被认为是比较小的（另一些则是比较大的）（？完全不知道在讲什么）。一般情况下，如果模式组包含了"keyword"和一个选择 catch-all，"identifier"规则，如在模式组中：

```
"case"          return TOK_CASE;
"switch"         return TOK_SWITCH;
...
"default"         return TOK_DEFAULT;
[a-z]+          return TOK_ID;
```

你最好还是不要使用全表搜索。如果只有"identifier"规则出现并且你使用了 hash 表或是类似的表来检查关键字 keyword，你还是不要使用-F 为好。

这个选项和-CFr（下面有讨论）一样。它不能和选项-+一同使用。

-I 指示 flex 生成一个交互式扫描器。交互式扫描器就是向前查看下一个匹配的 token 是什么。结果就是总向前多看了一个字符，即使是在扫描器已经看够了文本已经排除了 token 的歧义。但向前查看给了扫描器强大的交互能力；例如，当一个用户输入新的一行，它并不会被识别为新的一行 token 直到他们（？）进入另一个 token，这通常意味着输入另一整行。（？如果你明白这里说的，请告诉我。QQ:158989725）

缺省时，flex 扫描器是交互式的，除非你使用了-Cf 或者是-CF 表格压缩 table-compression 选项（看下面说明）。那是因为你想要得到更好性能你应当使用这些选项中的一个，所以你如果不用，flex 就假设你想用一点运行时的性能来换取直观的交互行为。注意，你不能联合使用-I 和-Cf 或者-CF。所以，这个选项实际上是不需要的；在允许的情况下，该标志默认是打开的。

你可以使用-B（上面已经说明了）把扫描器生成不是交互式的。

-L　指示 flex 不要生成#line 指令。没有这个标志，flex 会让生成的扫描器拥有#line 指令，有了这个，在 action 中的出错消息就能正确的从 flex 的输入文件中定位（如果错误是由于输入文件中的 code 引起的），或者从 lex.yy.c（如果错误是 flex 的故障--你应当通过下面给出的 email 地址报告这一类错误）中定位。

-T　让 flex 以跟踪模式运行。这将会产生一堆消息到 stderr，这些消息是关于输入的形式和 the resultant non-deterministic and deterministic finite automata.（？啥意思）。这个选项主要用在 flex 的维护上。

-V　打印版本号到 stdout 并退出。--version 和-V 是同义的。

-7　指示 flex 生成 7-bit 扫描器，这样，它只能从它的输入中识别 7-bit 字符。使用-7 选项的好处是扫描器的表格可以只是使用-8 生成的扫描器的表格一半大小（关于-8，看下面）。不好的地方就是当输入是 8-bit 字符时，这样的扫描器通常会挂起或是崩溃（--!）。

注意，除非你使用了-Cf 或是-CF 这样的表格压缩选项，使用-7 只是节省了一小点表格的空间，但会让扫描器移植性较差。flex 缺省行为是生成一个 8-bit 的扫描器除非你使用了-Cf 或者是-CF，在这种情况下，除非你所在的区域是设置成生成 8-bit（通常是在 non-USA 地方）扫描器，flex 默认是生成一个 7-bit 扫描器。检查使用-v 选项得到 flex 的标志概述，你可以知道 flex 生成的是 7-bit 或是 8-bit 扫描器。

注意，如果你使用了-Cfe 或者-CFe（那些表格压缩选项，还使用了等价的的类，下面有描述），flex 还是能生成 8-bit 扫描器，因为通常使用这样的压缩，8-bit 的表格并不会比 7-bit 的大多少。

-8　批示 flex 生成一个 8-bit 扫描器，能够识别 8-bit 字符的扫描器。只有在使用了-Cf 或是-CF 选项时，才有必须使用这个选项，如果不是使用了-Cf 或-CF，flex 默认总是生成 8-bit 扫描器。

-+　指定你想让 flex 去生成一个 C++扫描器类 class。看下面 Generating C++ Scanners 部分有更多细节。

-C[aefFmr]　控制表格的压缩程序，更为一般的是控制生成小的扫描器还是快的扫描器。

-Ca ("align") 指示 flex 以更大的表格交换更快的执行效率，因为表中的元素为内存访问和计算对齐了。在一些 RISC 体系结构，取回和处理多个字比短的字有更好的效率（？看得一晕一晕的）。这个选项可能让你的扫描器表格大小增大一倍。

-Ce 指令 flex 创建等价的类 classes，也就是有相同词典属性的一组字符。（例如，如果 flex 的输入文件中的数字只有字符类别"[0-9]"一种形式，那么所有数字都归在同一个等价类中）。等价类 equivalence classes 通常能显示的减少最后表格/对象的文件大小（典型的能达到 2-5 的比例）并且能得到比较好的性能（一个数组可以查找每一个被扫描的字符）。

-Cf 指定生成全扫描器表 full scanner tables－flex 不压缩表格，这样有利于不同状态间传递函数。

-CF 指定应该使用更快的扫描器（如上面-F 选项中描述的）。这个选项不能和-+一起使用。

-Cm 指示 flex 构造 meta-equivalence classes，这是一个通常在一起被使用的一组 equivalence classes（或者是字符如果不是使用等价类的话）。在压缩表格时使用原等价类是很不错的，但它们也会造成性能上的波动（一个或二个"if"测试和一个数组查找每一个被扫描的字符）。

-Cr 导致生成的扫描器旁路 stdio 做为输入。相反，使用 fread()或是 getc()，扫描器将使用 read()系统调用，这导致在不同的系统之间有性能上的差异，但一般这种差异是微不足道的除非你同时使用了-Cf 或-CF。使用-Cr 有可能会导致古怪的行为，如你在使用扫描器之前使用 stdio 从 yyin 中读取数据（因为扫描器会失去你先前从 stdio 输入缓存读到的数据）。

-Cr 如果你定义了 YY_INPUT，那么这选项不起作用（看上面的 Generated Scanner）。

一个单独的-C 表示压缩扫描器表格但不使用等价类和原等价类。

选项-Cf 或-CF 和-Cm 在一起使用才有意义－如果表格没被压缩，那么就没有机会用到 meta-equivalence classes。这些选项可以自由混合使用，效果是累加的。

默认设置是-Cem，它指定 flex 生成等价类和原等价类。这个设置提供了最高的表格压缩等级。你可以以大表格的代价来换取更快的扫描速度，将下面的选上就可以办到：

最慢和最小
-Cem
-Cm
-Ce
-C
-C(f,F)e
-C(f,F)
-C(f,F)a
最快和最大

注意，最小表格扫描器通常能被最快的生产和编译，所以在开发过程中，你应该选用默认的最大压缩方式。

-Cfe 通常是在生成的扫描器的速度和大小之间的一个折中办法。

-ooutput
指定 flex 的输出文件，替代 lex.yy.c。如果你联合-o 和-t 选项，那么扫描器会被输出到 stdout 但它的#line 指令（看上面-L 选项说明）指示的是文件输出。

-Pprefix

把 flex 拿到到的所以全局可见的变量的缺省的 yy 前缀和函数名字为指定的前缀。例如，-Pfoo 改 yytext 成 footext。它还改变了缺省的输出文件名称 lex.yy.c 为 lex.foo.c。这里是所有受影响的名称：

```
yy_create_buffer
yy_delete_buffer
yy_flex_debug
yy_init_buffer
yy_flush_buffer
yy_load_buffer_state
yy_switch_to_buffer
yyin
yyleng
yylex
yylineno
yyout
yyrestart
yytext
Yywrap
```

（如果你使用了 C++扫描器，那么只有 yywrap 和 yyFlexLexer 受影响）。在你的扫描器自身中，你还可以用全局变量和函数两个版本名字之一来使用它们；但从外部来看，它们的名字已经被修改了。这个选项让你更简单地把多个 flex 程序链接成一个可执行程序。然而，注意到这个选项还修改了 yywrap() 的名字，所以你现在必须为你的扫描器提供你自己版本的例程并且使用%option noyywrap，就象用-lfl 链接一样不再提供一个默认的 yywrap()。

-Sskeleton_file

重载缺省的框架文件 skeleton file，使用从它构建 flex。你可能永远也用不到这个选项，除非你做 flex 的维护和开发。

flex 还在扫描器的说明中，而不是在 flex 命令中提供一个机制用来控制选项。在扫描器的说明文件（flex 的输入文件）的第一段中使用 %option 指令就可以实现。你可以用单个%option 指令指定多个选项，和使用多个指令。

大多数选项以名字的方式给出，可选择在名字的前面加个"no"表示取消它们的含义。flex 的标志或是它们的否定意义都相当于一个数字：

```
7bit       -7 option
8bit       -8 option
align      -Ca option
backup     -b option
batch      -B option
c++        -+ option
```

区分大小写，反之是缺省的-i（大小写不敏感）
caseful 或 case-sensitive

case-insensitive 或 caseless 表示 -i 选项

```
debug            -d option
default          opposite of -s option
ecs              -Ce option
fast             -F option
full             -f option
interactive      -I option
lex-compat       -l option
meta-ecs         -Cm option
perf-report      -p option
read             -Cr option
stdout           -t option
verbose          -v option
warn             opposite of -w option

       (use "%option nowarn" for -w)

array            equivalent to "%array"
pointer          equivalent to "%pointer" (default)
```

一些%option 提供一个命令行里没有的特性：

always-interactive
指示 flex 生成的扫描器总是把它的输入认为是"interactive"。正常情况下，扫描器会在每一个输入文件中调用 isatty() 来认为扫描器的输入源是 interactive 的（每次只读一个字符）（？啥意思，interactive 不是这样定义的）。当使用了这个选项，就不会再有 isatty() 这样的调用了。

main
指示 flex 为扫描器提供一个缺省的 main() 函数，它只是简单的调用了 yylex()。这个选项暗示 noyywrap（下面有说明）。

never-interactive
flex 生成的扫描器从不认为输入是交互的（不会调用 isatty()）。这和总是 interactive 正好相反。

stack
让开始条件栈可被使用（看上面的 Start Condition）。

stdinit
如果初始化设置（%option stdinit）了 yyin 和 yyout 为 stdin 和 stdout，而不是默认的那个 nil。一些 lex 程序依赖这个行为，甚至它们不是遵从 ANSI C 规范，ANSI C 不需要编译时常量 stdin

和 stdout。

yylineno
flex 生成的扫描器用全局变量 yylineno 维护着输入文件的当前行编号。option lex-compat 隐含有这个选项。

yywrap
如果没有设置(就如%option noyywrap)，当扫描器遇到 end-of-file 时，不会调用 yywrap()，但简单的假定没有更多的文件可以扫描（直到用户把 yyin 指向新的文件并再一次调用 yylex（））。

flex 通过扫描 rule 中的 action 来判断你是否使用了 REJECT 或是 yymore 属性。你可用 %option reject 表示要使用这个特性而用%option noyymore 表示不使用这个特性。

三个选项使用了字符串值，从'='开始：
%option outfile="ABC"等同于-oABC
%option prefix="XYZ"等同于-PXYZ

最后，%option yyclass="foo"只有当生成 C++扫描器（-+选项）时才有效。它告诉 flex 你已经派生 foo 作为 yyFlexLexer 的子类，所以 flex 将把你的 action 放在成员函数 foo::yylex()中而不是 yyFlexLexer::yylex()中。它还生成一个 yyFlexLexer::yylex()成员函数，如果被调用，将忽略运行时错误（调用 yyFlexLexer::LexerError()）。看下面的 Generating C++ Scanners 获取更多信息。

有一些选项可用来限制一个例程不出现在生成的扫描器中。下面这些如果不被设置（如 %option nounput）将导致相应的例程不出现在生成的扫描器中。

input
unput
yy_push_state
yy_pop_sate
yy_top_state
yy_scan_buffer
yy_scan_bytes
yy_scan_string

（虽然只有当你使用了%option stack 时 yy_push_state 和相关的例程才会出现）

性能考虑 PERFORMANCE CONSIDERATIONS

flex 的主要设计目标是生成一个高性能的扫描器。它已经对处理大量 rule 做了优化。除了上面概述的用-C 选项进行表格压缩之外，还有一些 option/action 会影响到扫描器的速度。从最大影响到最弱，有这一些：

REJECT

```
%option yylineno
arbitrary trailing context

pattern sets that require backing up
%array
%option interactive
%option always-interactive

'^'beginning-of-line operator
yymore()
```

头三个开销最大，最后两个最小。注意 unput() 有可能被用例程实现而造成更多操作，而 yyless() 是一个开销相当低的宏；所以如果只是回放一些你多扫了的文本，可以用 yyless()。

REJECT 当性能是非常重要的时候，应该尽可能避免使用它。它是一个开销特别昂贵的选项。

去除备份是件麻烦事并且对于复杂的扫描器来说，要做一大堆的事情。原则上，在开始的时候使用 -b 标志生成 lex.backup 文件。例如，在输入

```
%%
foo           return TOK_KEYWORD;
foobar        return TOK_KEYWORD;
```

备份文件看起来象：

```
State #6 is non-accepting -
associated rule line numbers:
2    3
out-transitions: [ o ]
jam-transitions: EOF [ \001-n  p-\177 ]

State #8 is non-accepting -
associated rule line numbers:
3
out-transitions: [ a ]
jam-transitions: EOF [ \001-`  b-\177 ]

State #9 is non-accepting -
associated rule line numbers:
3
out-transitions: [ r ]
jam-transitions: EOF [ \001-q  s-\177 ]

Compressed tables always back up.
```

第一行告诉我们，这里有一个扫描器状态，通过它可以实现传输'o'，但其他字符不行，并且当前扫描的

文本不匹配任何 rule。该状态出现在当试图匹配出现在输入文件在第 2 和第 3 行的 rule 的时候。如果扫描器是在这样的状态并且接着读到其他一些不是'o'，它将去后退去查找一个被匹配的 rule。有点伤脑筋的是当扫描器已经看到"fo"时，扫描器已经处在这样一个状态中。（？完全看不懂原文是什么意思）当这一切已经发生，如果任何其他不是'o'被看到，扫描器将不得不后退到简单匹配'f'（缺省 rule）。

状态#8 的注释说明这里有一个问题，当"foob"被扫描到的时候。确实，在其他不是'a'的任何字符上，扫描器将必须后退去接受"foo"。同样，状态#9 的注释考虑当"fooba"已经被扫描到并且当接下来的字符不是'r'。

最后的注释提醒我们，这里没有指向从 rule 移除备份的所有错误除非我们使用了-Cf 或是-CF，因为这里没有性能可从压缩过程获得。

移除后退的方法被加到"error"规则 rule：

```
%%
foo         return  TOK_KEYWORD;
foobar      return  TOK_KEYWORD;

fooba       |
foob        |
fo          {
            * false alarm, not really a keyword */
            return  TOK_ID;
            }
```

在"catch-all"中，同样也能做到在一组 keyword 中做到消除后退。

```
%%
foo         return  TOK_KEYWORD;
foobar      return  TOK_KEYWORD;

[a-z]+      return  TOK_ID;
```

这个在恰当的环境中是最好的解决办法。

后退的消息趋向于层叠。一组复杂的 rule 就不会得到数百个消息。如果有人能识别它们，然而，它通常是使用一把左右 rule 去消除后退。（虽然，这容易造成错误，并且有可能有一个错误的 rule 意外的匹配一个有效的 token。将来的 flex 可能有特性自动加一个 rule 来消除后退）。

谨记，你只有在消除了所以后退的情况，你才能从中获益。即使有只有一个漏了，那什么好处也得不到。

上下文环境变量（开始和结束部分没有固定长度）会造成一 REJECT 一样的性能损失（大体上的）。所以当有一个这样的 rule：

```
%%
mouse|rat/(cat|dog)     run();
```

最好写成：

```
%%
mouse/cat|dog           run();
rat/cat|dog             run();
```

或是

```
mouse|rat/cat           run();
mouse|rat/dog           run();
```

注意，这里使用特殊的'|'action 没有得到任何好处，而且有可能把事物弄糟（看下面 Deficiencies/Bugs 部分）。

其他方面用户可以提升扫描器的性能（容易做到的）原于这样的一个事实，越长的 token 被匹配，扫描器就能跑得越快。这是因为长的 token 匹配能在内部扫描循环中处理大部分的字符而不用再为 action 做一些额外的扫描器的环境设置（如 yytext）。回忆一下处理 C 注释的扫描器：

```
%x comment
%%
    int line_num = 1;

"/*"        BEGIN(comment);

<comment>[^*\n]*
<comment>"*"+[^*/\n]*
<comment>\n             ++line_num;
<comment>"*"+"/"        BEGIN(INITIAL);
```

写成下面这样扫描起来更快：

```
%x comment
%%
    int line_num = 1;

"/*"        BEGIN(comment);

<comment>[^*\n]*
<comment>[^*\n]*\n          ++line_num;
<comment>"*"+[^*/\n]*
<comment>"*"+[^*/\n]*\n     ++line_num;
<coment>"*"+"/"             BEGIN(INITIAL);
```

现在，每一个新行的处理不再要求别的 action 来做，新行的处理分散在 rule 中，使得匹配的文本尽可能的长。扫描器的速度不依赖于 rule 的数量或（就象这一节开头提到的那样）是 rule 能够使用多复杂的操作如'*'和'|'。

最后一个加快扫描速度的例子：假设你想要扫描一个包含标识符和关键字的文件，每一行和没有其他无相关的字符，并识别所有的关键字。一个自然的办法就是：

```
%%
asm           |
auto          |
break         |
...etc...
volatile      |
while             /* it's a keyword */

.|\n             /* it's not a keyword */
```

消除向后跟踪，引入一个 catch-all 规则：

```
%%
asm           |
auto          |
break         |
...etc...
volatile      |
while             /* it's a keyword */

[a-z]+        |
.|\n             /* it's not a keyword */
```

现在，如果它保证一行只有一个 word，那么我们可以把匹配的总个数减半，通过合并新行的识别和其他的 token：

```
%%
asm\n         |
auto\n        |
break\n       |
...etc...
volatile\n    |
while\n           /* it's a keyword */

[a-z]+\n|
```

```
.|\n            /* it's not a keyword */
```

这里有个需要小心的地方，我们现在又重新引入了后退到扫描器中。尤其是，当我们知道在输入流中除了字母或是新行再也没有其他任何字符，flex 没办法说明这种情况，所以它会想到可能需要后退当他扫描到一个象"auto"这样的 token 和接下来的字符是除新行或是字母的其他字符时。去消除后退的可能性，我们可用复制所有的 rule 除最后的新行除外，或是因为我们再也不会遇到一个我们无法分类的输入，我们可以再引入一个 catch-all rule，这个不再包含新行：

```
%%
asm\n         |
auto\n        |
break\n       |
...ect...
wolatile\n    |
while\n          /* it's a keyword */

[a-z]+\n      |
[a-z]+        |
.|\n            /* it's not a keyword */
```

用-Cf 编译，这是能跑得很快的 flex 扫描器，可用于处理特定问题。

最后注意：在匹配 NUL 时，flex 会慢一些，尤其是哪一个 token 包含多个 NUL 的时候。如果知道文本经常包含 NUL，写 rule 的时候最好少匹配一些文本。

关于性能最后还有一个要注意的：就如在上面 How the Input is Matched 一节中提到的，动态改变 yytext 去容纳大型的 token 是比较慢的处理方式，因为目前要求大型的 token 需要多头开始再扫描一次。所以如果性能是重点，你应该打算匹配数量比较多的文本而不是大型的，以 8k 为界。

生成 C++扫描器 GENERATING C++ SCANNERS

flex 提供了两种方式来生成可被 C++使用的扫描器。第一种方法是简单的编译一个扫描器，用 C++编译器而不是 C 编译器。你可能会碰到编译错误（把相关的报告发到 Author 节中的 email 地址）。你接下来可在 action 中用 C++代码。注意你的扫描器的默认输入源仍然是 yyin，并且默认的回显仍然有效。这两个仍旧是 FILE* 变量而不是 C++ stream。

你还能使用 flex 生成一个 C++扫描器类 class，使用-+选项（或是相同效果的 %option c++），这选项是自动指定，如果 flex 的执行程序的名字以一个'+'结束，比如 flex++ 。当使用这个选项，flex 默认生成扫描器到文件 lex.yy.cc 而不是 lex.yy.cc。生成的扫描器包含头文件 FlexLexer.h，这头文件定义接口到两个 C++类。

第一个 class，FlexLexer，提供一个抽象基类定义一般扫描器类接口 class interface。它提供下面这些成员函数：

```
const char* YYText()
```
返回最近匹配的 token 文本，等同于 yytext。

```
int YYLeng()
```
返回最近匹配的 token 的文本长度，等同于 yyleng。

```
int lineno() const
```
返回当前输入的行号，或是 1 如果%option yylineno 没被使用的话。

```
void set_debug(int flag)
```
为扫描器设置调试的标志，等同于指定 yy_flex_debug（看上面 Options 部分）。注意你必须用 %option debug 建立扫描器来在它里面包含调试程序。

```
int debug() const
```
返回当前的调试标志的设置。

一同被提供的还有成员函数等同于 yy_switch_to_buffer()，yy_create_buffer()（虽然第一个参数是一个 istream*对象指针并且不是 FILE*），yy_flush_buffer()，yy_delete_buffer()和 yyrestart()（再一次，第一个参数是 istream*对象指针）。

第二个在 FlexLexer.h 中定义的 class 是 yyFlexLexer，它是从 FlexLexer 派生而来。它定义了下的额外的成员函数：

```
yyFlexLexer(istream* arg_yyin=0, ostream* arg_yyout = 0)
```
使用给定的输入输出流构建一个 yyFlexLexer 对象。如果没有指定，默认的流分别是 cin 和 cout。

```
virtual int yylex()
```
在普通的 flex 扫描器中执行的相同的角色的是 yylex()：它扫描输入的流，消耗掉 token，直到有一个 rule 的 action 返回一个值。如果你从 yyFlexLexer 派生了一个子类 S 并且想要在 yylex()中访问 S 的成员函数和变量，那么你需要使用到选项%option yyclass="S"来告诉 flex 你将要使用子类而不是 yyFlexLexer。在这种情况下，flex 生成的是 S::yylex()，不是 yyFlexLexer::yylex()（同时也还会生成一个调用了 yyFlexLexer::LexerError()的木桩函数 yyFlexLexer::yylex()）。

```
virtual void switch_stream(istream* new_in=0, ostream* new_out=0)
```
重新设置 yyin 成 new_in（如果不是 nil 的话），同样 yyout 成 new_out，如果 yyin 已经被设置，那么删除原来的输入缓存。

```
int yylex(istream* new_in, ostream* new_out=0)
```
首先通过 switch_stream(new_in, new_out)切换到新的输入流，接着返回 yylex()的值。

另外，yyFlexLexer 定义下面这些保护虚函数，你可以从派生类来重新定义它们，这样能按你的要求剪裁扫描器：

```
virtual int LexerInput(char* buf, int max_size)
```
读取 max_size 个字符到 buf 中，并返回读取到的字符个数。返回 0 表示遇到输入的结束。注意交互式的扫描器（看-B 和-I 标志）定义宏 YY_INTERACTIVE。如果你重定义了 LexerInput()并且需要

执行不同的动作依赖于扫描器是不是从交互式的输入流中扫描，你可以通过#ifdef 来测试这个名字的值。

virtual void LexerOutput(const char* buf, int size)

从缓存 buf 输出 size 个字符，buf 以 NUL 结束，如果扫描器的 rule 能够匹配带有 NUL 的文本，buf 也要包含内容的 NUL。

virtual void LexerError(const char* msg)

报告一个严重错误消息。这个错误的默认版本把消息写到流 cerr 并退出。

注意，一个 yyFlexLexer 对象包含它所有的扫描状态。所以你可以使用这样的对象来创建"重进"扫描器。你可以初始化多个 yyFlexLexer 类的实例，你还可以在同一个程序中使用-P 选项（上面讨论过的）把多个 C++扫描器组在一起。

最后，注意%array 特性对于 C++扫描器是不可用的；你必须使用%pointer（默认值）。

这里有一个简单的 C++扫描器例子：

```
    // An example of using the flex C++ scanner class.
%{
int mylineno = 0;
%}

string \"[^\n"]+\"

ws    [ 、和]+

alpha  [A-Za-z]
dig [0-9]
name      ({alpha}|{dig}|\$)({alpha}|{dig}|[_.\-/$])*
num1      [-+]?{dig}+\.?([eE][-+]?{dig}+)?
num2      [-+]?{dig}*\.{dig}+([eE][-+]?{dig}+)?
number  {num1}|{num2}

%%

{ws}      /* skip blanks and tabs */

"/*"      {
    int c;

    while((c = yyinput()) != 0)
    {
        if(c == '\n')
            ++mylineno;
```

```
        else if(c == '*')
        {
           if((c = yyinput()) == '/')
              break;
            else
              unput(c);
           }
        }
    }

{number}  cout << "number " << YYText() << '\n';

\n      mylineno++;

{name}    cout << "name " << YYText() << '\n';

{string}  cout << "string " << YYText() << '\n';

%%

int main( int /* argc */, char** /* argv */ )
{
    FlexLexer* lexer = new yyFlexLexer;
    while(lexer->yylex() != 0)
          ;
    return 0;
}
```

如果你想创建多个（不同）lexer classes，你使用-P 标志（或是 prefix=option）来重命名每一个 yyFlexLexer 为其他的 xxFlexLexer。你接着可以包含<FlexLexer.h>到你的其他每一个 lexer 类的源代码中，首先象下面这样重命名 yyFlexLexer：

```
#undef yyFlexLexer
#define yyFlexLexer xxFlexLexer
#include <FlexLexer.h>

#undef yyFlexLexer
#define yyFlexLexer zzFlexLexer
#include <FlexLexer.h>
```

你为你的一个扫描器使用了%option prefix="xx"，为另外一个指定了%option prefix="zz"。

注意：当前的扫描器类的形式只是实验性的，在大的应用中可能需要做相应的改动。

与 lex 和 posix 不兼容的地方 INCOMPATIBILITIES WITH LEX AND POSIX

flex 是一个重写的 AT&T Unix lex 工具（虽然它们之间并没有共享任何源代码），有一些扩展和不兼容，如果想要让 flex 生成的扫描器能够能够接受两种类型的输入，那么就必须注意这两方面的内容。flex 是完全以 POSIX lex 的规格来编译的，除非使用了 %pointer（缺省值），unput() 的调用释放 yytext 的内容，这些都是属于 POSIX 规格说明。

在这一节我们讨论所有关于 flex，AT&T lex，和 POSIX 规格不兼容的地方。

flex 的-l 选项开启了与原始的 AT&T lex 的最大兼容，造成了扫描器的主要性能损失。下面的这些不兼容可以通过使用-l 选项来克服。

flex 和 lex 除了以下这些，其他方面都是完全兼容的：

－未列入文档的 lex 扫描器内部变量 yylineno 不被支持，除非使用了-l 或是%option yylineno。yylineno 应该是每一个 buffer 一个，而不是每一个扫描器配一个（单一全局变量）。

yylineno 不是 POSIX 规格说明的一部分。

－input() 是不可以被重定义的，虽然可以调用它来读取那些被 rule 匹配的字符。如果 input() 遇到一个 end-of-file，一般 yywrap() 将完成处理。一个"真正"end-of-file 是以 EOF 从 input() 返回。

input 被 YY_INPUT 宏定义所替代。

flex 约束 input() 不可被重定义是为了保持与 POSIX 规格一致，POSIX 规格没有指定任何控制扫描器输入，只是给 yyin 做了一个初始化赋值。

－unput() 不可被重定义。这个约束是为了保持与 POSIX 一致。

－flex 扫描器不象 lex 扫描器那样可以重入的。尤其是，如果你有一个交互式扫描器和一个异常处理通过 long-jump 跳出扫描器，接下来再调用扫描器，你会得到以下消息：

```
fatal flex scanner internal error--end of buffer missed
```

对于重入扫描器，首先使用

```
yyrestart(yyin);
```

注意这个调用会抛弃任何已经缓存的输入；通常对于一个交互式扫描器这并不是什么问题。

同时注意到，flex 的 C++扫描器类是可以重入的，所以能使用 C++选项，你就使用它。看上面的 Generating C++ Scanners 看有关细节。

－output() 不被支持。在 ECHO 宏中的输出使用的是 yyout 文件指针（缺省是 stdout）。

output() 不是 POSIX 规格的一部分。

－lex 不支持排他开始条件（%x），虽然 POSIX 规格说明中有这一个。

－当定义被扩展时，flex 用圆括号括起来。对于 lex，下面这些

```
NAME  [A-Z][A-Z0-9]*
%%
foo[NAME]:  printf("Found  it\n");
%%
```

不会匹配字符串"foo"因为当宏被扩展，rule 会等同于"foo[A-Z][A-Z0-9]*?"并且优先情况是'?'和"[A-Z0-9]*"结合在一起的。flex 中的 rule 会被扩展成为"foo([A-Z][A-Z0-9]*)?"并且"foo"会被匹配到。

注意到如果定义以^开始或是以$结束，那么它不会扩展成为圆括号包起来的，为了让这些操作不会失去它们的特殊含义。但<s>，/，和<<EOF>>操作不能做为 flex 的定义。

使用-l 使得 lex 在定义的周围没有圆括号。

POSIX 规格说明是让定义用圆括号括起来。

－一些 lex 的实现允许 rule 的 action 以单独一行开始，如果 rule 的模式后跟空白符：

```
%%
foo|bar<space  here>
  {foobar_action();}
```

flex 不支持这种特征。

－lex 的%r（生成一个 Ratfor 扫描器）选项不被支持。它不是 POSIX 规格的一部分。

－调用 unput()之后，yytext 是未定义的直到下一个 token 被匹配，除非扫描器是以%array 建立的。lex 或 POSIX 规格扫描器不会有这样的情况。-l 选项可以消除这种不兼容的情况。

－{}（数字范围）操作的优先级别是不一样的。lex 解释"abc{1,3}为"匹配一个，两个或是三个'abc'"，然而 flex 解释为"匹配'ab'跟着一个，两个或是三个'c'"。后者和 POSIX 规格一致。

－^操作的优先级别是不一样的。lex 解释"^foo|bar"为"匹配出现的行开头的'foo'或是其他地方的'bar'"，然而 flex 会解释为"行开头的'foo'或是行开头的'bar'"。后者和 POSIX 规格一致。

－lex 支持特殊的表大小声明如%a，但 flex 忽略它们。

一 #define 名字 FLEX_SCANNER 可以让扫描器同时使用 flex 或 lex。扫描器还包含了 YY_FLEX_MAJOR_VERSION 和 YY_FLEX_MINOR_VERSION 表示是哪个版本的 flex 生成了扫描器(如，2.5 版本，这两个宏将分别是 2 和 5)。

下面的 flex 特性不被 lex 或是 POSIX 规格包含：

```
C++ scanners
%option
start condition scopes
start condition stacks
interactive/non-interactive scanners
yy_scan_string() and friends
yyterminate()
yy_set_interactive()
yy_set_bol()
YY_AT_BOL()
<<EOF>>
<*>
YY_DECL
YY_START
YY_USER_ACTION
YY_USER_INIT
#line directives
%{}'s around actions
multiple actions on a line
```

增加了绝大部分的 flex 标志。最后一特性说明你可以在同一行里写上多个 action，以分号隔开，当用 lex 时，下面的

```
    foo handle_foo(); ++num_foos_seen;
```

会被剪裁成（吃惊么？）

```
    foo handle_foo();
```

flex 不会截断 action。没有用大括号括起来的 action 就以行结束做为终结。

诊断 DIAGNOSTICS

警告，rule 不能被匹配说明给出的 rule 跟在其他能够匹配同样的文本的 rule 后面。例如，后面的"foo"不能被匹配因为它跟在标识符"catch-all"rule 后面：

```
[a-z]+ got_identifier();
foo got_foo();
```

在扫描器中使用 REJECT 能够去掉这个警告。

警告，-s 选项给出了但默认的 rule 能够（可能只有在特殊的开始条件中）被匹配意味着它有可能默认 rule（匹配任何单个字符）是唯一一个能够匹配特殊的输入。因为给出了-s，推测这个不大可能发生。

reject_used_but_not_detected undefined or yymore_used_but_detected undefined
这些错误可能发现在编译的时候。它们表示扫描器使用了 REJECT 或 yymore() 但 flex 看不到实际情况，意思是 flex 扫描头两段搜索 action 并且找不到，但你却弄了一些进去（如通过#include 文件）。使用%option reject 或%option yymore 向 flex 表示你将要使用到这些特性。

flex scanner jammed
一个扫描器用-s 编译，遇到一个输入字符串不能被任何一个 rule 匹配。内部问题也有可能造成这个错误。

token too large, exceed YYMAX
你的扫描器使用了%array 并且其中一个 rule 匹配一个字符串长过 YYLMAX 常量（缺省是 8K）。你可以在 flex 的输入文件的定义段中通过#defing YYLMAX 增长这个数值。

scanner requires -8 flag to use the character 'x'
你的扫描器规格包含识别 8-bit 字符'x'并且你没有指定-8 标志，你的扫描器默认是 7-bit 的因为你使用了-Cf 或-CF 表格压缩选项。看标志-7 的讨论获取更多细节。

flex scanner push-back overflow
你使用了 unput() 放回太多文本以致扫描器的缓存不能存放下放回的文本和当前在 yytext 中的 token。理想是在这样的情况下扫描器能够动态改变缓存的大小，但目前还不能支持这样的功能。

input buffer overflow, can't enlarge buffer because scanner uses REJECT
扫描器工作在匹配一个超大的 token 并且需要扩展输入缓存。使用了 REJECT 的扫描器将不能工作。

fatal flex scanner internal error -- end of buffer missed
这个有可能发现在一个扫描器中，它在一个 long-jump 跳出了扫描器的活动框架后重入。在重入前扫描器使用：

```
yyrestart(yyin);
```

或切（如上面提到的）换到使用 C++扫描器 class。

too many start condition in <> construct
你在一个<>构造中列出了比现存的（所以你必须至少列出其中的一个两次）还多的开始条件。

文件 FILES

-lfl 扫描器链接时必须指定的库。

lex.yy.c 生成的扫描器（在一些系统中叫做 lexyy.c）。

lex.yy.cc 生成的 C++扫描器 class，当使用-+的时候。

<FlexLexer.h> 头文件定义 C++扫描器基础 class，FlexLexer 和它的派生 class，yyFlexLexer。

flex.skl 框架扫描器。只有当建立 flex 时才会用到这个文件，不是在 flex 执行时使用。

lex.backup 备份信息，使用-b 标志时产生的（在一些系统上叫做 lex.bck）。

不足和 BUG DEFICIENCIES / BUGS

在一些上下文环境中，模式不能被正确的匹配并产生警告消息（"dangerous trailing context"）。这是一些模式，模式中的第一部分的结束部分能够匹配第二部分的开头，如"zx/zy"，这里的'x*'匹配了在上下文环境中的'x'。（注意到 POSIX 草稿状态是未定义的，当文本被这样的模式匹配。）

一些上下文环境 rule，一些部分的长度是固定的不能被识别会造成上面提到过的性能损失。尤其是，使用'|'或是{n}（例如"foo{3}"）总是被认做变长的。

上下文联合特殊的'|'action 会导致更大的性能损耗。例如下面这个：

```
%%
abc |
xyz/def
```

unput()的使用让 yytex 和 yyleng 变得无效，除非使用了%array 或者-l 选项。

NUL 的模式匹配比匹配其他字符要慢。

输入缓存动态更改大小比较慢，因为它需要重新扫描所有当前 token 中已经匹配了的文本。

由于输入的缓存和超前读取，你不能在 flex 的 rule 中混合使用<stdio.h>中的例程，如 getchar()并期望它能正常工作。可换成调用 input()。

使用-v 标志列出的表的所有记录，不包括用来判断被匹配的 rule 的记录。表中的记录等同 DFA 状态如果扫描器没有使用 REJECT 的话，否则，记录数会比状态多。

REJECT 不能和-f 或-F 选项合用。

flex 的内部算法需要文档化。

参考 SEE ALSO

lex(1),yacc(1),sed(1),awk(1)

John Levine, Tony Mason, and Doug Brown, Lex & Yacc, O'Reilly and Associates. Be sure to get the 2nd edition. M. E. Lesk and E. Schmidt, LEX

- Lexical Analyzer Genera-tor

Alfred Aho, Ravi Sethi and Jeffrey Ullman, Compilers: Principles, Techniques and Tools, Addison-Wesley (1986). Describes the pattern-matching techniques used by flex(deterministic finite automata).

作者 AUTHOR

Vern Paxson, with the help of many ideas and much inspiration from Van Jacobson. Original version by Jef Poskanzer. The fast table representation is a partial implementation of a design done by Van Jacobson. The implementation was done by Kevin Gong and Vern Paxson.

Thanks to the many flex beta-testers, feedbackers, and contributors, especially Francois Pinard, Casey Leedom, Robert Abramovitz, Stan Adermann, Terry Allen, David Barker-Plummer, John Basrai, Neal Becker, Nelson H.F. Beebe, benson@odi.com, Karl Berry, Peter A. Bigot, Simon Blanchard, Keith Bostic, Frederic Brehm, Ian Brockbank, Kin Cho, Nick Christopher, Brian Clapper, J.T. Conklin, Jason Coughlin, Bill Cox, Nick Cropper, Dave Curtis, Scott David Daniels, Chris G. Demetriou, Theo Deraadt, Mike Donahue, Chuck Doucette, Tom Epperly, Leo Eskin, Chris Faylor, Chris Flatters, Jon Forrest, Jeffrey Friedl, Joe Gayda, Kaveh R. Ghazi, Wolfgang Glunz, Eric Goldman, Christopher M. Gould, Ulrich Grepel, Peer Griebel, Jan Hajic, Charles Hemphill, NORO Hideo, Jarkko Hietaniemi, Scott Hofmann, Jeff Honig, Dana Hudes, Eric Hughes, John Interrante, Ceriel Jacobs, Michal Jaegermann, Sakari Jalovaara, Jeffrey R. Jones, Henry engst, Klaus Kaempf, Jonathan I. Kamens, Terrence O Kane, Amir Katz, ken@ken.hilco.com, Kevin B. Kenny, Steve Kirsch, Winfried Koenig, Marq Kole, Ronald Lamprecht, Greg Lee, Rohan Lenard, Craig Leres, John Levine, Steve Liddle, David Loffredo, Mike Long, Mohamed el Lozy, Brian Madsen, Malte, Joe Marshall, Bengt Martensson, Chris Metcalf, Luke Mewburn, Jim Meyering, R. Alexander Milowski, Erik Naggum, G.T. Nicol, Landon Noll, James Nordby, Marc Nozell, Richard Ohnemus, Karsten Pahnke, Sven Panne, Roland Pesch, Walter Pelissero, Gaumond Pierre, Esmond Pitt, Jef Poskanzer, Joe Rahmeh, Jarmo Raiha, Frederic Raimbault, Pat Rankin, Rick Richardson, Kevin Rodgers, Kai Uwe Rommel, Jim Roskind, Alberto Santini, Andreas Scherer, Darrell Schiebel, Raf Schietekat, Doug Schmidt, Philippe Schnoebelen, Andreas Schwab, Larry Schwimmer, Alex Siegel, Eckehard Stolz, Jan-Erik Strvmquist, Mike Stump, Paul Stuart, Dave Tallman, Ian Lance Taylor, Chris Thewalt, Richard M. Timoney, Jodi Tsai, Paul Tuinenga, Gary Weik, Frank Whaley, Gerhard Wilhelms, Kent Williams, Ken Yap, Ron Zellar, Nathan Zelle, David Zuhn, and those whose names have slipped my marginal mail-archiving skills but whose contributions are appreciated all the same.

Thanks to Keith Bostic, Jon Forrest, Noah Friedman, John Gilmore,

Craig Leres, John Levine, Bob Mulcahy, G.T.Nicol, Francois Pinard, Rich Salz, and Richard Stallman for help with various distribution headaches.

Thanks to Esmond Pitt and Earle Horton for 8-bit character support; to Benson Margulies and Fred Burke for C++ sup port; to Kent Williams and Tom Epperly for C++ class sup port; to Ove Ewerlid for support of NUL's; and to Eric Hughes for support of multiple buffers.

This work was primarily done when I was with the Real Time Systems Group at the Lawrence Berkeley Laboratory in Berkeley, CA. Many thanks to all there for the support I received.

Send comments to **vern@ee.lbl.gov**.