FLEX 中文手册

- 一些简单的例子
- 输入文件的格式
- 模式
- 如何匹配输入
- 动作
- 生成的扫描器
- 开始条件
- 文件结尾规则
- 与 yacc 一起使用

一、一些简单的例子

首先给出一些简单的例子，来了解一下如何使用 flex。下面的 flex 输入所定义的扫描器，用来将所有的“

username”字符串替换为用户的登陆名字：

%% username printf("%s", getlogin());

默认情况下，flex 扫描器无法匹配的所有文本将被复制到输出，所以该扫描器的实际效果是将输入文件

复制到输出，并对每一个“username”进行展开。在这个例子中，只有一个规则。“username”是模式

（pattern），“printf”是动作（action）。“%%”标志着规则的开始。

这里是另一个简单的例子：

```
int num_lines = 0, num_chars = 0;
```

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

%% int main(void)

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

该扫描器计算输入的字符个数和行数（除了最后的计数报告，并未产生其它输出）。第一行声明了两

个全局变量，“num_lines”和“num_chars”，可以在 yylex()函数中和第二个“%%”后面声明的 main()函数中

使用。有两个规则，一个是匹配换行符（“\n”）并增加行数和字符数，另一个是匹配所有不是换行符的

其它字符（由正规表达式“.”表示）。

一个稍微复杂点的例子：

/* scanner for a toy Pascal-like language */

%{

```
/* 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);

%%

int main(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 语言的简单扫描器的初始部分，用来识别不同类型的标志（tokens）并给出报告。

这个例子的详细介绍将在后面的章节中给出。

二、输入文件的格式

flex 输入文件包括三个部分，通过“%%”行来分开：

definitions（定义） %% rules（规则） %% user code（用户代码）

定义部分，包含一些简单的名字定义（name definitions），用来简化扫描器的规范，还有一些开始状态

（start conditions）的声明，将会在后面的章节中说明。名字定义的形式如下：

name definition

“name”由字母或者下划线（“_”）起始，后面跟字母，数字，“_”或者“-”（破折号）组成。定义由名字

后面的一个非空白（non-white-space）字符开始，直到一行的结束。可以在后面通过“{name}”来引用定

义，并展开为“(definition)”。例如，

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

定义了“DIGIT”为一个正规表达式用来匹配单个数字，“ID”为一个正规表达式用来匹配一个字母，后面

跟零个或多个字母和数字。后面的引用如下，

{DIGIT}+"."{DIGIT}*

等同于

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

用来匹配一个或多个数字，后面跟一个“.”，然后是零个或者多个数字。

flex 输入的规则部分包括一系列的规则，形式如下：

pattern action

模式（pattern）不能有缩进，动作（action）必须在同一行。

参见后面对模式和动作的进一步描述。

最后，用户代码部分将被简单的逐字复制到“lex.yy.c”中，作为随同程序用来调用扫描器或者被扫描器

调用。该部分是可选的，如果没有，输入文件中第二个“%%”也可以省略掉。

在定义部分和规则部分，任何缩进的文本或者包含在“%{”和“%}”中的文本，都会被逐字的复制到输出中（并去掉“%{}”）。“%{}”本身不能有缩进。

在规则部分，在第一个规则之前的任何缩进的或者%{}中的文本，可以用来声明扫描程序的局部变量。其它在规则部分的缩进或者%{}中的文本也会被复制到输出，但是它的含义却不好定义，而且可能会产生编译时错误（这一特点是为了与 POSIX 相同；参见后面的其它特点）。

在定义部分（但不是在规则部分），一条未缩进的注释（即，由“/*”起始的行）也会被逐字的拷贝到输出，直到下一个“*/”。

三、模式

输入中的模式，使用的是扩展的正规表达式集。它们是：

'x'

匹配字符’x’

'.'

除了换行符以外的任意字符（字节）

'[xyz]'

一个字符类别（character class）；在这个例子中，该模式匹配一个’x’，或者一个’y’，或者一

个'z' '[abj-oZ]'

一个带有范围的字符类别；匹配一个’a’，或者一个’b’，或者从’j’到’o’的任意字母，或者一个’Z’

'[^A-Z]'

一个反选的字符类别（negated character class），即任意不属于这些的类别。在这个例子中，

表示任意一个非大写字母的字符。 '[^A-Z\n]'

任意一个非大写字母的字符，或者一个换行符

'r*'

零个或者多个 r，其中 r 是任意的正规表达式

'r+'

一个或者多个 r

'r?'

零个或者一个 r（也就是说，一个可选的 r）

'r{2,5}'

两个到五个 r

'r{2,}'

两个或者更多个 r

'r{4}'

确切的 4 个 r

'{name}'

“name”定义的展开（参见前面）

' "[xyz]\"foo" '（这里单引号和双引号之间没有空格）

文字串：'[xyz]"foo'

'\x'

如果 x 是一个'a'，'b'，'f'，'n'，'r'，'t'或者'v'，则为 ANSI-C 所解释的\x。否则，为一个文字'x'（

用来转义操作符，例如'*'）。 '\0'

一个 NUL 字符（ASCII 代码 0）

'\123'

八进制值为 123 的字符

'\x2a'

十六进制值为 2a 的字符

'(r)'

匹配一个 r；括号用来改变优先级（参见后面）

'rs'

正规表达式 r，后面跟随正规表达式 s；称作“concatenation”

'r|s'

或者 r，或者 s

'r/s'

一个 r，但是后面要跟随一个 s。在文本匹配时，s 会被包含进来，以判断该规则是否是最长的匹配，但是在动作执行前会被返回给输入。因此，动作只会看到匹配 r 的文本。这种模式称作 trailing context。（有些'r/s'组合，flex 会匹配错误；参见后面的不足和缺陷章节中，关于“危险的尾部相关”的注解）

'^r'

一个 r，但是只在一行的开始（即，刚开始扫描，或者一个换行符刚被扫描之后）

'r$'

一个 r，但是只在一行的结尾（即，正好在换行符之前）。等同于“r/\n”。注意，flex 中对换行符的概念跟用来编译 flex 的 C 编译器中对'\n'的解释是一模一样的。特别的是，在一些 DOS 系统上，必须在输入中自己过滤出'\r'，或者显示的使用 r/\r\n 来表示 r$。

'<s>r'

一个 r，但是只在起始条件（start condition）s（参见下面关于起始条件的讨论）下匹配。<s1, s2, s3>相同，但是在任意起始条件 s1，s2，s3 下都可以。

'<*>r'

一个 r，在任意的起始条件下，甚至是互斥的（exclusive）起始条件。

'<<EOF>>'

文件结尾

'<s1, s2><<EOF>>'

文件结尾，当在起始条件 s1 或 s2 下匹配。

注意，在字符类别里面，除了转义符（'\'），字符类别操作符'-'，']'和类别开始处的'^'，所有其它的

正规表达式操作符不再具有特殊的含义。

上面列出的正规表达式，是按照优先级由高到低排列的。同一级别的具有相同的优先级。例如，

foo|bar*

等同于

(foo)|(ba(r*))

因为，'*'操作符的优先级比串联高，串联的优先级比间隔符高（'|'），所以，该模式匹配字符串"foo"

或者字符串"ba"后面跟随零个或多个 r。如果要匹配"foo"或者零个或多个"bar"，可以使用：

foo|(bar)*

如果要匹配零个或者多个"foo"，或者零个或多个"bar"：

(foo|bar)*

除了字符和序列字符，字符类别也可以包含字符类别表达式。这些表达式由'[:'和':]'分隔符封装（并且

必须在字符类别的分隔符'['和']'之中）。有效的表达式包括：

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

这些表达式都指定了与标准 C 中‘isXXX’函数相对应的字符类别。例如，

‘[:alnum:]’指定了‘isalnum()’返

回值为真的字符集，即，任意的字母或者数字。一些系统没有提供‘isblank()’，

则 flex 定义‘[:bland:]’为

一个空格符（blank）或者一个制表符（tab）。

例如，下面的字符类别是等同的：

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

如果你的扫描器是大小写无关的（使用‘-i’命令行选项），则‘[:upper:]’和

‘[:lower:]’等同于‘[:alpha:]’。

关于模式的一些注意事项：

> 一个反选的字符类别例如上面的“[^A-Z]”将会匹配一个换行符，除非“\n”（或者等同的转义序

列）在反选字符类别中显示的指出（如“[^A-Z\n]”）。这一点不像许多其它正

规表达式工具，但不幸的

是这种不一致是由历史造成的。匹配换行符意味着像[^"]*这样的模式能够匹

配整个的输出直到遇到另

一个引号。

> 一条规则中只能最多有一个尾部相关的情况（‘/’操作符或者‘$’操作符）。起始条件，‘^’和

“<<EOF>>”只能出现在模式的开始处，并且和‘/’，‘$’一样，都不能包含在圆括号中。‘^’如果不出现在

规则的开始处，或者‘$’不出现在规则的结尾，将会失去它的特殊属性，并且被作为普通字符。下面的

例子是非法的：

```
foo/bar$ <sc1>foo<sc2>bar
```

注意，第一个可以写作“foo/bar\n”。下面的例子中，‘$’和‘^’将会被作为普通字符：

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

如果想匹配一个“foo”，或者一个“bar”并且后面跟随一个换行符，可以使用下面的方式（特殊动作‘|’，

将在下面介绍）：

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

类似的技巧可以用来匹配一个 foo，或者一个 bar 并且在一行的起始处。

四、如何匹配输入

当生成的扫描器运行时，它会分析它的输入，来查找匹配任意模式的字符串。如果找到多个匹配，则采取最长文本的匹配方式（对于尾部相关规则，也包括尾部的长度，虽然尾部还要返回给输入）。如果找到两个或者更多的相同长度的匹配，则选择列在 flex 输入文件中的最前面的规则。

一旦匹配确定，则所匹配的文本（称作标识，token）可以通过全局字符指针 yytext 来访问，文本的长度存放在全局整形 yyleng 中。与匹配规则相应的动作（action）将被执行（后面会有关于动作的详细描述），然后剩余的输入再被继续扫描匹配。

如果没有找到匹配，则执行默认的规则：紧接着的输入字符被认为是匹配的，并且复制到标准输出。因此，最简单合法的 flex 输入是：

%%

生成的扫描器只是简单的将它的输入（一次一个字符的）复制到它的输出。

注意，yytext 可以通过两种方式来定义：作为一个字符指针，或者作为一个字符数组。可以在 flex 输入的第一部分（定义部分）通过专门的指令‘%pointer’或者‘%array’来控制 flex 使用哪种定义。缺省的为‘%pointer’，除非使用‘-l’lex 兼容选项，使得 yytext 为一个数组。使用‘%pointer’的优点是能够进行足够快的扫描，并且当匹配非常大的标识时不会有缓冲溢出（除非是动态内存耗尽）。缺点是在动作中对 yytext 的修改方式将会有所限制（参见下一章节），并且调用‘unput()’函数将会破坏 yytext 的现有内容，在不同 lex 版本中移植时，这将是一个非常头痛的事情。

使用‘%array’的优点是，可以按照自己的意愿来修改 yytext，并且调用‘unput()’也不会破坏 yytext（参见下面）。而且，已有的 lex 程序有时可以通过如下的声明方式，在外部访问 yytext：

extern char yytext[];

这种定义在使用‘%pointer’时是错误的，但是对于‘%array’却可以。

‘%array’定义了 yytext 为一个 YYLMAX 个字符的数组，缺省情况下，YYLMAX 是一个相当大的值。可以在 flex 输入的第一部分简单的通过#define YYLMAX 来改变大小。正如上面提到的，‘%pointer’指定的 yytext 是通过动态增长来适应大的标识的。这也意味着‘%pointer’的扫描器能够接受非常大的标识（例如匹配整个的注释块），不过要记住，每次扫描器都要重新设定 yytext 的长度，而且必须从头扫描整个标识，所以匹配这样的标识时速度会很慢。目前，如果调用‘unput()’并且返回太多的文本，yytext 将不会动态增长，而只是产生一个运行时错误。

而且要注意，不能在 C++扫描器类（c++选项，参见下面）中使用‘%array’。

五、动作

规则中的每一个模式都有一个相应的动作，它可以是任意的 C 语句。模式结束于第一个非转义的空白字符；该行的剩余部分便是它的动作。如果动作是空的，则当模式匹配时，输入的标识将被简单的丢弃。例如，下面所描述的程序，是用来删除输入中的所有“zap me”：

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

（输入中的所有其它字符，会被缺省规则匹配，并被复制到输出。）

下面的程序用来将多个空格和制表符压缩为单个空格，并且丢弃在一行尾部的所有空格：

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

如果动作包括‘{’，则动作的范围直到对称的‘}’，并且可以跨过多行。flex 可以识别 C 字符串和注释，因此字符串和注释中的大括号不会起作用。但是，也

允许动作由‘%{’开始，并且将直到下一个‘%}’之间的动作看作是文本（包括在动作里面出现的普通大括号）。

只包含一个垂直分割线（‘|’）的动作意味着“与下一个规则相同”。参见下面的例子。

动作能够包含任意的 C 代码，包括 return 语句来返回一个值给调用‘yylex()’的程序。每次调用‘yylex()’，它将持续不断的处理标识，直到文件的结尾或者执行了 return。

动作可以自由的修改 yytext，除了增加它的长度（在尾端增加字符的，将会覆盖输入流中后面的字符）。不过这种情形不会发生在使用‘%array’时（参见前面）；在那种情况下，yytext 可以任意修改。

动作可以自由修改 yyleng，除非他们不应该这么做，比如动作中也使用了‘yymore()’（参见下面）。

在动作中可以包含许多特殊指令:

* ‘ECHO’将 yytext 复制到扫描器的输出。
* BEGIN 后面跟随起始状态的名字，使扫描器处于相应的起始状态下（参见下面）。
* REJECT 指示扫描器继续采用“次优”的规则匹配输入（或者输入的前面一部分）。规则根据前面在“如何匹配输入”一节中描述的方式来选择，并且 yytext 和 yyleng 也被设为适当的值。它可能是和最初选择的规则匹配相同多的文本，但是在 flex 输入文件中排在后面的规则，也可能是匹配较少的文本的规则。例如，下面的将会计算输入中的单词，并且还在“frob”出现时调用程序 special()：

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

如果没有 REJECT，输入中任何“frob”都不会被计入单词个数，因为扫描器在通常情况下只是对每一个标识执行一个动作。可以使用多个 REJECT，对于每一个，都将查找当前活动规则的下一个最优选择。例如，当下面的扫描器扫描标识“abcd”时，它将往输出写入“abcdabcaba”：

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

（前三个规则都执行第四个的动作，因为他们使用了特殊的动作‘|’。）对于扫描器执行效率来说，REJECT 是一种相当昂贵的特征；如果在扫描器的任意动作中使用了它，它将使得扫描器的所有匹配速度降低。而且，REJECT 不能和‘-Cf’或‘-CF’选项一起使用（参见下面）。还要注意的是，不像其它特有动作，REJECT 是一个分支跳转；在动作中紧接其后面的代码将不会被执行。

* ‘yymore()’告诉扫描器在下一次匹配规则时，相应的标识应该被追加到现在的 yytext 的值中，而不是替代它。例如，假设有输入“mega-kludge”，下面的将会向输出写入“mega-mega-kludge”：

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

首先是“mega-”被匹配，并且回显到输出。然后是“kludge”被匹配，但是先前的“mega-”还保留在 yytext 的起始处，因此“kludge”规则中的‘ECHO’实际将会写出“mega-kludge”

使用‘yymore()’时，有两点需要注意的。首先，‘yymore’依赖于 yyleng 的值能够正确地反映当前标识的长度，所以在使用‘yymore()’时，一定不要修改 yyleng。其次，在扫描器的动作中使用‘yymore()’，会给扫描器的匹配速度稍微有些影响。

* ‘yyless(n)’将当前标识的除了前 n 个字符之外的都回送给输入流，扫描器在查找接下来的匹配时，会重新扫描它们。yytext 和 yyleng 会相应做适当的调整（例如，yyleng 将会等于 n）。例如，在输入“foobar”时，下面的规则将会输出“foobarbar”：

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

如果 yyless 的参数为 0，则会使当前整个输入字符串被重新扫描。除非已经改变扫描器接下来如何处理它的输入（例如，使用 BEGIN），否则将会产生一个

无限循环。注意，yyless 是一个宏，并且只能用在 flex 输入文件中，其它源文件则不可以。

* ‘unput(c)’将字符 c 回放到输入流，并将其作为下一次扫描的字符。下面的动作将会接受当前的标识，并使它封装在括号中而被从新扫描。

```
{
    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，由于‘unput()’每次都将字符放回到输入流的起始处，因此如果要回放字符串，则必须从后往前操作。在使用‘unput()’时，一个重要的潜在问题是如果使用‘%pointer’（缺省情况），调用‘unput()’会破坏 yytext 的内容，将会从最右面的字符开始，每次向左吞并一个。如果需要保留 yytext 的值直到调用‘unput()’之后（如上面的例子），必须将其复制到别处，或者使用‘%array’来构建扫描器（参见如何匹配输入）。最后，注意不能将 EOF 放回来标识输入流出去文件结尾。

* ‘input()’从输入流中读取下一个字符。例如，下面的方法可以去掉 C 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’，则‘input()’由‘yyinput()’代替，为了避免与‘C++’流 input 的名字冲突。）

＊YY_FLUSH_BUFFER 刷新扫描器内部的缓存，以至于扫描器下次匹配标识时，将会首先使用 YY_INPUT 重新填充缓存（参见下面的生成的扫描器）。这个动作是较为常用的函数 `yy_flush_buffer()’的特殊情况，将在下面的多输入缓存章节介绍。

＊‘yyterminate()’可以在动作中的用来代替 return 语句。它将终止扫描，并返回 0 给扫描器的调用者，用来表示“全部完成”。默认情况下，‘yyterminate()’也会在遇到文件结尾时被调用。它是一个宏，可以被重定义。

六、生成的扫描器

‘lex.yy.c’是 flex 的输出文件，其中包含了扫描程序‘yylex()’，许多的数据表，用来匹配标识，还有许多的辅助程序和宏。默认情况下，‘yylex()’按下面的方式被声明：

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

（如果环境支持函数原形，则为“int yylex(void)”。）可以通过定义“YY_DECL”宏来改变该定义。例如，可以使用：

1. define YY_DECL float lexscan(a, b) float a, b;

来给出扫描程序的名字 lexscan，返回一个浮点值，并且接受两个浮点参数。注意，如果是使用 K&R 风格/无原形的函数声明，在给出扫描程序参数时，必须用分号（‘;’）来结束定义。

只要调用‘yylex()’，它便从全局输入文件 yyin（缺省值为 stdin）中扫描标识，并且直到文件的结尾（这种情况下将返回 0）或者其中的一个动作执行了 return 语句。

如果扫描器到达文件的结尾，接下来的调用则是未定义的，除非 yyin 被指向一个新的输入文件（这种情况下，将继续扫描那个文件），或者调用了‘yyrestart()’。‘yyrestart()’接受一个参数，一个‘FILE *’指针（可以为空，如果已经设置了 YY_INPUT 来指定输入源，而不是 yyin），并且初始化 yyin 来扫描那个文件。基本上，直接将新的输入文件赋值给 yyin 和使用‘yyrestart()’没有区别；后者可以用来与之前的 flex 版本相兼容，因为它可以用来在扫描过程的中间来改变输入文件。它也可以用来丢弃当前的输入缓存，通过将 yyin 作为参数进行调用；不过更好的方式是用 YY_FLUSH_BUFFER(参见上面)。注意，‘yyrestart()’不会重设开始状态为 INITIAL（参见下面的开始状态）。

如果‘yylex()’是由于动作中执行了一个 return 语句而停止扫描的，则扫描器可以被再次调用，并且它将会从上次离开的地方继续扫描。默认情况下（出于效率目的），扫描器使用块读取方式从 yyin 中读入字符，而不是简单的调用‘getc()’。

可以通过定义 YY_INPUT 宏来控制扫描器获得输入的方式。YY_INPUT 的调用方式为“YY_INPUT(buf,result,max_size)”。它执行的操作为在字符数组中放入 max_size 个字符，并且通过整数变量 result 返回值，或者是读入字符的个数或者是常量 YY_NULL（在 Unix 系统下为 0）来表示 EOF。缺省的 YY_INPUT 从全局文件指针“yyin”中读取字符。

一个定义 YY_INPUT 的例子（在输入文件的定义部分部分）：

%{

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

%}

该定义将会改变输入的处理方式为一次处理一个字符。

当扫描器从 YY_INPUT 获得文件结束标识时，它将检查‘yywrap()’函数。如果‘yywrap()’返回假（零），则认为调用函数已经运行并将 yyin 设为指向另一个输入文件，并开始继续扫描。如果返回真（非零），则扫描器终止，并返回 0 给调用者。注意，对于每一种情况，开始状态都会保持不变，并不转变为 INITIAL。

如果不提供自己的‘yywrap()’版本，则必须使用‘%option noyywrap’（这种情况，跟从‘yywrap()’返回 1 一样），或者必须与‘-lfl’连接来获得缺省的程序版本，其也是返回 1。

有三个函数可以用来扫描内存中缓存，而不是文件：‘yy_scan_string()’，‘yy_scan_bytes()’，和 ‘yy_scan_buffer()’。关于它们的讨论，可以参见下面的多输入缓存章节。

扫描器将‘ECHO’的输出写到全局变量 yyout 中（缺省值为 stdout），用户可以通过简单的将其它文件指针赋值给 yyout 来重定义它。

七、开始条件

flex 提供了一种机制，可以条件执行规则。任何模式以“<sc>”为前缀的规则，都将只在扫描器处于名为“sc”的开始条件下才被执行。例如，

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

将只在开始条件为"STRING"时才被执行，而

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

将只在开始条件为"INITIAL"，"STRING"，或者"QUOTE"时才被执行。

开始条件在输入的定义部分（第一部分）中被声明，使用‘%s’和‘%x’，后面跟名字列表，并且声明不能有缩进。前一种形式声明相容的开始条件（inclusive），后一种形式声明互斥的开始条件（exclusive）。通过使用 BEGIN 动作来激活一个开始条件。一直到下一次执行 BEGIN 动作之前，具有该开始条件的规则将是可执行的，而具有其它开始条件的规则将是不可执行的。如果开始条件是相容的，则没有任何开始条件的规则也将是可执行的。如果开始条件是互斥的，则只有符合开始条件的规则是可执行的。对于一个扫描器，具有同一互斥开始条件的一组规则相对独立于其它任何规则。因此，互斥的开始条件可以用来指定一个小型扫描器，以扫描在语法上不同于其它部分的输入（例如，注释）。

如果对于相容开始条件和互斥开始条件之间的区别还是不很清楚，这里有一个简单的例子可以说明二者之间的关系。一组规则：

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

相当于

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

如果没有‘<INITIAL,example>’来限定，当处于‘example’开始条件时，第二个例子中的模式‘bar’将不被执行（也就是说，不会被匹配）。如果我们只是使用‘<example>’来限定‘bar’，虽然会被执行，但只是在处于‘example’开始条件时被执行，在 INITIAL 时则不会。但是第一个例子中，它将都会被执行，因为第一个例子中的‘example’开始条件是一个相容的（‘%s’）开始条件。

还要注意的是，特殊的开始条件‘<*>’用来匹配每一个开始条件。因此，上面的例子还可以写成，

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

在开始条件下，缺省规则（‘ECHO’任何无法匹配的字符）仍然有效。它相当于：

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

‘BEGIN(0)’回到到最初的状态，即只有不具有开始条件的规则才被执行。这个状态还可以通过开始条件“INITIAL”来指出，所以‘BEGIN(INITIAL)’相当于‘BEGIN(0)’。（开始条件的名字使用括号括住并不是必须的，但却是一种好的风格。）

动作 BEGIN 还可以在规则部分的开始处，通过缩紧的代码给出。例如，下面的将会使扫描器每当调用‘yylex()’并且全局变量 enter_special 为真时，进入“SPECIAL”开始条件：

```
int enter_special;
```

%x SPECIAL %%

```
if ( enter_special )
    BEGIN(SPECIAL);
```

<SPECIAL>blahblahblah ...more rules follow...

为了说明开始条件的用法，这里有一个扫描器用来对“123.456”这样的字符串提供两种不同的解析方式。缺省情况下，它将把它作为三个标识，整数“123”，点（‘.’），和整数“456”。但是如果在字符串的同一行中，先有了字符串“expect-floats”，它将被作为单个标识，浮点数 123.456：

%{

1. include <math.h>

%} %s expect

%% expect-floats BEGIN(expect);

<expect>[0-9]+"."[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]+ {

Version 2.5 December 1994 18

```
            printf( "found an integer, = %d\n",
                    atoi( yytext ) );
            }
```

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

这里是一个扫描器，用来识别（并且丢弃）C 注释，同时维护当前输入的行数。

%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);

注意，开始条件的名字实际上是整数值，也同样可以被保存。因此，上面的例子可以扩展成下面的样式：

%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);

而且，可以使用具有整数值的宏 YY_START 来访问当前的开始条件。例如，

上面对 comment_caller 的赋值可以被替换为

comment_caller = YY_START;

Flex 提供了 YYSTATE 作为 YY_START 的别名（因为 AT&T 的 lex 中是这样用的）。

注意，开始条件没有自己的名字域，对于%s 和%x 声明的名字和通过#define 定义的样式是一样的。

最后，这里有一个例子，通过互斥的开始条件来匹配 C 风格的带有引号的字符串，包括通过转义符连接的跨行字符串。（不过没有对字符串是否太长进行检查）：

%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 parser
         */
        }
```

<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，像上面的一些例子，可能要连着写一串都是起始于相同的开始条件的规则。Flex 引入了一种开始条件域的概念，可以使这变得简洁，容易一些。

一个开始条件域起始于：

<SCs>{

这里 SCs 是一个开始条件列表。在开始条件域中，每一个规则都回使用前缀‘<SCs>’。作用域直到遇到和最初的‘{’匹配的‘}’结束。所以，

<ESC>{

```
    ″\\n″   return ’\n’;
    ″\\r″   return ’\r’;
    ″\\f″   return ’\f’;
    ″\\0″   return ’\0’;
```

}

相当于：

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

开始条件域可以嵌套使用。

有三个函数可以用来操作开始条件的栈：

`void yy_push_state(int new_state)'

将当前的开始条件压入开始条件栈，并且转换为 new_state，就像执行‘BEGIN new_state’一样（记着，开始条件名字也是整数）。

`void yy_pop_state()'

弹出栈顶的内容，并且通过 BEGIN 转换到该状态。

`int yy_top_state()'

返回栈顶的内容，并且不改变栈的内容。

开始条件栈是动态增长的，因此没有内置的大小限制。如果内存被耗尽，程序便异常中断。

若要使用开始条件栈，扫描器必须包含一个‘%option stack’指令（参见下面的选项一节）。

八、文件结尾规则

特殊规则"<<EOF>>"指的是在遇到文件结尾处，并且 yywrap()返回非零时（即，表示不再有其它文件需要处理）所要执行的动作。其动作必须完成下面的其中一件事情：

* 赋值给 yyin 一个新的输入文件（在之前的 flex 版本中，赋值之后还需要调用专门的动作 YY_NEW_FILE；现在不需要了）；
* 执行一个 return 语句；
* 执行特殊动作‘yyterminate()’；
* 或者，如上面的例子中，使用‘yy_switch_to_buffer()’转换到一个新的缓存中。

规则<<EOF>>可以不和其它模式一起使用；可以只通过开始条件加以限制。

如果给出一个无条件的<<EOF>>规则，它将会应用到所有没有<<EOF>>动作的开始条件。如果只指定初始开始条件，则要使用

<INITIAL><<EOF>>

这样的规则可以用来帮助捕获未结束的注释等类似事情。例子：

%x quote %%

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

<quote><<EOF>> {

```
        error( "unterminated quote" );
        yyterminate();
        }
```

<<EOF>> {

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

九、与 yacc 一起使用

Flex 的主要用途之一就是与 yacc 分析生成器一起使用。yacc 分析器将会调用名字为‘yylex()’的函数来获得下一个输入标识。该函数应该返回下一个输入标识的类型，并且将关联的值放在全局变量 yylval 中。若要使用与 yacc 一起使用 flex，需要给 yacc 使用‘-d’选项，用来指示生成包含出现在 yacc 输入中的所有‘%tokans’的定义的文件‘y.tab.h’。然后将该文件包含在 flex 扫描器中。例如，如果其中一个标识为"TOK_NUMBER"，则扫描器的部分内容可能为：

%{

1. include "y.tab.h"

%}

%%

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