## 2.6 利用Lex自动生成扫描程序

本节将重复前一节完成的用于TINY语言的扫描程序的开发，但此次是利用Lex扫描程序生成器从作为正则表达式的TINY记号的描述中生成一个扫描程序。由于Lex存在着多个不同的版本，所以我们的讨论仅限于对于所有的或大多数版本均通用的特征。 Lex最常见的版本是**flex**（Fast Lex），它是由Free Software Foundation创建的**Gnu compiler package**的一部分，可以在许多Internet 站点上免费得到。

Lex 是一个将包含了正则表达式的文本文件作为其输入的程序，此外还包括每一个表达式被匹配时所采取的动作。 Lex生成一个包含了定义过程 **yylex**的C源代码的输出文件，其中yylex是与输入文件相对应的DFA表驱动的实现，它的运算与**getToken**过程类似。接着编译通常称作**lex.yy.c**或**lexyy.c**的Lex输出文件，并将它们链接到一个主程序上以得到一个可运行的程序，这与在前一节中将**scan.c**文件与**tiny.c**文件链接相似。

下面将首先讨论用于编写正则表达式的Lex惯例和Lex输入文件的格式，之后还会谈到附录

B中给出的TINY扫描程序的Lex输入文件。

### 2.6.1 正则表达式的Lex约定

Lex 约定与2.2.3节中所谈到的十分相似，但它并不列出所有的 Lex元字符且不逐个地描述它们，我们将给出一个概述，之后再在一个表格中写出 Lex 约定。

Lex允许匹配单个字符或字符串，只需像前面各节中所做地一样按顺序写出字符即可。 Lex还允许通过将字符放在引号中而将元字符作为真正的字符来匹配。引号可用于并不是元字符的字符前后，但此时的引号却毫无意义。因此，在要被直接匹配的所有字符前后使用引号很有意义，而不论该字符是否为元字符。例如，可以用**if**或**"if"**来匹配一个**if**语句开始的保留字if。另一方面，如要匹配一个左括号，就必须写作**"("**，这是因为左括号是一个元字符。另一个方法是利用反斜杠元字符****，但它只有在单个元字符时才起作用：如要匹配字符序列**(***，就必须重复使用反斜杠，写作**\(***。很明显，**"(*"**更好一些。另外将反斜杠与正规字符一起使用就有了特殊意义。例如：**\n**匹配一新行，**\t**匹配一个制表位（这些都是典型的C约定，大多数这样的约定在Lex中也可行）。

Lex按通常的方法解释元字符*****、**+**、**(**、**)**和**|**。Lex 还利用问号作为元字符指示可选部分。为了说明前面所讲到的Lex表示法，可为*a* 串和*b* 串的集合写出正则表达式，其中这些串是以*aa*或*bb*开头，末尾则是一个可选的*c*：

```
(aa|bb)(a|b)*c?
```

或写作：

```
("aa"|"bb")("a"|"b")*"c"?
```

字符类的Lex 约定是将字符类写在方括号之中。例如**[abxz]**就表示*a*、*b*、*x* 或*z* 中的任意一个字符，此外还可在Lex 中将前面的正则表达式写作：

```
(aa|bb)[ab]*c?
```

在这个格式的使用中还可利用连字符表示出字符的范围。因此表达式**[0-9]**表示在Lex 中，任何一个从0~9的数字。句点也是一个表示字符集的元字符：它表示除了新行之外的任意字符。互补集合——也就是不包含某个字符的集合——也可使用这种表示法：将插入符 ^作为括号中的第1个字符，因此**[^0-9abc]**就表示不是任何数字且不是字母*a*、*b* 或*c* 中任何一个的其他任意字符。

例：为一个标有符号的数集写出正则表达式，这个集合可能包含了一个小数部分或一个以字母E开头的指数部分（在2.2.4节中，这个表达式的写法略有不同）：

```
("+"|"-")?[0-9]+("."[0-9]+)?(E("+"|"-")?[0-9]+)?
```

Lex有一个古怪的特征：在方括号（表示字符类）中，大多数的元字符都丧失了其特殊状况，且无需用引号引出。甚至如果可以首先将连字符列出来的话，则也可将其写作正则字符。因此，可将前一个数字的正则表达式**("+"|"-")**写作**[-+]**，（但不可写作**[+-]**，这是因为元字符“ - ”用于表示字符的一个范围）。又例如：**[."?]**表示了句号、引号和问号3个字符中的任一个字符（这3个字符在括号中都失去了它们的元字符含义）。但是一些字符即使是在方括号中也仍旧是元字符，因此为了得到真正的字符就必须在字符前加一个反斜杠（由于引号已失去了它们的元字符含义，所以不能用它），因此**[\^\\]**就表示了真正的字符^或\。

Lex中一个更为重要的元字符约定是用花括号指出正则表达式的名字。在前面已经提到过可以为正则表达式起名，而且只要没有递归引用，这些名字也可使用在其他的正则表达式中。

例如：将2.2.4节中的**signedNat**定义如下：

```
nat = [0-9] +
signedNat= ("+"|"-") ? nat
```

在本例和其他示例中，我们使用斜体字将名字和普通的字符序列区分开来。但是 Lex 文件是普通的文本文件，因此无需使用斜体字。相反地， Lex 却遵循将前面定义的名字放在花括号中的约定。因此，上一例在Lex 中就表示为（Lex 还在定义名字时与等号一起分配）：

```
nat [0-9] +
signedNat (+|-)? {nat}
```

请注意，在定义名字时并未出现花括号，它只在使用时出现。

表2-2是讨论过的Lex元字符约定的小结列表。Lex 中还有许多我们用不到的元字符，这里也就不讲了（参见本章末尾的“ 注意与参考 ”）。

**表2-2 Lex中的元字符约定**

| 格 式 | 含 义 |
|---|---|
| **a** | 字符*a* |
| **"a"** | 即使*a* 是一个元字符，它仍是字符*a* |
| **\a** | 当*a* 是一个元字符时，为字符*a* |
| **a*** | *a* 的零次或多次重复 |
| **a+** | *a* 的一次或多次重复 |
| **a?** | 一个可选的*a* |
| **a\|b** | *a* 或*b* |
| **(a)** | *a* 本身 |
| **[abc]** | 字符*a*、*b* 或*c* 中的任一个 |
| **[a-d]** | 字符*a*、*b*、*c* 或*d* 中的任一个 |
| **[^ab]** | 除了*a* 或*b* 外的任一个字符 |
| **.** | 除了新行之外的任一个字符 |
| **{xxx}** | 名字*xxx* 表示的正则表达式 |

### 2.6.2 Lex 输入文件的格式

Lex输入文件由 3个部分组成：定义（ definition）集、规则（ rule）集以及辅助程序（auxiliary routine）集或用户程序（user routine）集。这3个部分由位于新一行第 1列的双百分号分开，因此，Lex输入文件的格式如下所示：

```
{definitions}
%%
{rules}
%%
{auxiliary routines}
```

为了正确理解Lex如何解释这样的输入文件，就必须记住该文件的一些部分是正则表达式信息，Lex利用这个信息指导构成它的C输出代码，而文件的另一部分则是提供给 Lex的真正的C代码，Lex会在适当的位置逐字地将它插入到输出代码中。在我们逐个解释完这 3个部分以及给出一些示例之后，将会告诉大家Lex 在这里所用的规则。

定义部分出现在第1个双百分号之前。它包括两样东西：第 1件是必须插入到应在这一部分

中分隔符“ **%{** ”和“ **%}** ”之间的任何函数外部的任意C代码（请注意这些字符的顺序）。第2件是正则表达式的名字也得在该部分定义。这个名字的定义写在另一行的第 1列，且其后（后面有一个或多个空格）是它所表示的正则表达式。

第2个部分包含着一些规则。它们由一连串带有C代码的正则表达式组成；当匹配相对应的正则表达式时，这些C代码就会被执行。

第3个部分包括着一些C代码，它们用于在第2个部分被调用且不在任何地方被定义的辅助程序。如果要将Lex输出作为独立程序来编译，则这一部分还会有一个主程序。当第 2个双百分号无需写出时，就不会出现这一部分（但总是需要写出第 1个百分号）。

下面给出一些示例来说明Lex输入文件的格式。

**例**2.20　以下的Lex输入指出一个给文本添加行号的扫描程序，它将其输出发送到屏幕上（如果被重定向，则是一个文件）：

```
%{
/* a Lex program that adds line numbers
   to lines of text, printing the new text
   to the standard output
*/
#include <stdio.h>
int lineno = 1;
%}
line *.\n
%%
{line} { printf ( "%5d %s", lineno++, yytext ); }
%%
main()
{ yylex(); return 0; }
```

例如，运行从这个输入文件本身的Lex中获取的程序会得到以下输出：

```
1 %{
2 /* a Lex program that adds line numbers
3    to lines of text, printing the new text
4    to the standard output
5 */
6 #include <stdio.h>
7 int lineno = 1;
8 %}
9 line .*\n
10 %%
11 { line } { printf ( "%5d %s", lineno++,yytext); }
12 %%
13 main ( )
14 { yylex( ) ; return 0; }
```

下面为这个使用了这些行号的Lex 输入文件作出解释。首先，第1行到第8行都是位于分隔符**%{**和**%}**之间，这样就使这些行可以直接插入到由Lex 产生的C代码中，而它是位于任何过程的外部。特别是从第 2行到第 5行的注释可以插入到程序开头的附近，还将从外部插入**#include**指示与第6行和第7行上的整型变量**lineno**的定义，因此**lineno**就变成了一个全程变量且在最初被赋值为1。出现在第1个**%%**之前的其他定义是名字**line**的定义，**line**被定义

为正则表达式**".*\n"**，它与零个或多个其后接有一新行的字符匹配（但不包括新行）。换而言之：由**line**定义的正则表达式与输入的每一行都匹配。在第10行的**%%**之后，第11行包括了Lex 输入文件的行为部分。此时每当匹配了一个 **line**时，都写下了一个要完成的行为（根据Lex 约定，**line**前后都用花括号以示与其作为一个名字相区别）。正则表达式之后是**action**，即每当匹配正则表达式都要被执行的 C代码。在这个示例中，该行为由包含在一个 C块的花括号中的C语句组成（请记住在名字line前后的花括号与构成下面行为中的C代码块的花括号有完全不同的作用）。这个C语句将打印行号（在一个有5个空格的范围内且右对齐）以及在它后面要增加**lineno**的串**yytext**。**yytext**的名字是Lex 赋予并由正则表达式匹配的串的内部名字，此时的正则表达式是由输入的每一行组成（包括新行）⊖ 。最后，当Lex 生成C代码结束时在第2个双百分号（第13行和第14行）之后插入C代码。在本例中，代码包括了一个调用函数 yylex 的**main**过程（**yylex**是由Lex构造的过程的名字，这个Lex 实现了与正则表达式和在输入文件的行为部分中与给出的行为相关的DFA）。

**例**2.21 考虑下面的Lex输入文件：

```
%{
/* a Lex program that changes all numbers
   from decimal to hexadecimal notation,
   printing a summary statistic to stderr
*/
#include <stdlib.h>
#include <stdio.h>
int count = 0;
%}
digit [0-9]
number {digit}+
%%
{number} { int n = atoi (yytext);
           printf ("%x", n);
           if (n > 9) count++;}
%%
main( )
{ yylex ( );
  fprintf ( stderr, "number of replacements = %d",
                    count);
  return 0 ;
}
```

它在结构上与前例类似，但**main**过程打印了在调用了**yylex**之后替换到**stderr**的次数。这个例子与前例的不同还在于它并没有匹配所有的文本。而实际上只在行为部分匹配了数字；在行为部分中，行为的C代码第1次将匹配串（**yytext**）转变成一个整型*n*，接着又将其打印为十六进制的格式**(printf("%x",...))**，最后如果这个数大于9则增加**count**（如果小于或等于9，则与在十六进制中没有分别）。因此，为串指定的唯一行为就是数字序列。 Lex还生成了一个可匹配所有非数字字符的程序，并将它传送到输出中。这是 Lex 的一个缺省行为（default action）的示例。如果字符或字符串与行为部分中的任何一个正则表达式都不匹配，则缺省地，

⊖ 本节最后将用一个表格列出这一节中所讨论过的Lex 内置名字。

Lex 将会匹配它并将它返回到输出中（Lex 还可被迫生成一个程序错误，但这里不讨论它了）。Lex 的内部定义宏**ECHO**也可特别指定缺省行为（下一个示例将会学到它）。

**例**2.22 考虑下面的Lex 输入文件：

```
%{
/* Selects only lines that end or
   begin with the letter 'a'.
   Deletes everything else.
*/
#include <stdio.h>
%}
ends_with_a .*a\n
begins_with_a a.*\n
%%
{ends_with_a} ECHO;
{begins_with_a} ECHO;
.*\n ;
%%
main( )
{  yylex( );  return 0;  }
```

这个Lex 输入将以字符*a* 开头或结尾的所有输入行均写到输出上，并消除其他行。行的消除是由**ECHO**规则下的规则引起的，在这个规则中，为 C行为代码编写一个分号就可为正则表达式**.*\n**指定“空”行为。

这个Lex 输入还有一个值得注意的特征：所列的规则具有二义性（ ambiguous），这是因为串可匹配多个规则。实际上，无论它是否是以*a* 开头或结尾的行的一部分，任何输入行都可与表达式**.*\n**匹配。Lex 有一个解决这种二义性的优先权系统。首先，Lex 总是匹配可能的最长子串（因此Lex 总是生成符合最长子串原则的扫描程序）。接着，如果最长子串仍与两个或更多的规则匹配，Lex 就选取在行为部分所列的第 1个规则。正是由于这个原因，上面的 Lex 输入文件就将**ECHO**行为放在第1个。如果已按下面的顺序列出行为：

```
.*\n;
{ends_with_a} ECHO;
{begins_with_a} ECHO;
```

则由Lex 生成的程序就不会再生成任何文件的输出，这是因为第 1个规则已匹配了输入的每一行了。

**例**2.23 在本例中，Lex 生成了将所有的大写字母转变成小写字母的程序，但这不包括 C- 风格注释中的字母（即：任何位于分隔符**/*...*/**之间的字母）：

```
%{
/*  Lex program to convert uppercase to
    lowercase except inside comments
*/
#include <stdio.h>
#ifndef FALSE
#define FALSE 0
#endif
#ifndef TRUE
```

```
#define TRUE 1
#endif
%}
%%
[A-Z]  {putchar(tolower(yytext[0]));
       /* yytext[0] is the single
          uppercase char found */
       }
"/*"   { char c ;
         int done = FALSE;
         ECHO;
         do
         { while ((c=input())!='*')
             putchar(c);
           putchar(c);
           while((c=input())=='*')
             putchar(c);
           putchar(c);
           if (c == '/') done = TRUE;
         } while (!done);
       }
%%
void main(void )
{ yylex();}
```

这个示例显示如何编写代码以回避较难的正则表达式，并且像执行一个 Lex 行为一样直接执行一个小的DFA。读者可以回忆一下2.2.4节中用C注释的一个正则表达式极难编写，相反地，我们只为开始C注释的串编写了正则表达式即："/*"，之后还提供了搜索结束串"*/"的行为代码，同时为注释中的其他字符提供适当的行为（此时仅是返回它们而不是继续进行）。这是通过模拟例2.9中的DFA来完成的（参见图2-4）。一旦识别出串"/*"，则就是在状态3，因此代码就在这里找到了 DFA。首先做的事情是在字符中循环直到看到一个星号（与状态 3中的***other***循环相对应）为止，如下所示：

```
while ((c=input())!='*') putchar(c);
```

这里又使用了Lex 另一个内部过程**input**，该过程的使用——并不是利用**getchar**的一个直接输入——保证使用了Lex 输入缓冲区，而且还保留了这个输入串的内部结构（但请注意，我们确实使用了一个直接输出过程**putchar**，2.6.4节将谈到它）。

DFA代码的下一步与状态4相对应。再次循环直到看不到星号为止；之后如在字符前有一个前斜杠就退出；否则就返回到状态3中。

本节的最后是小结在各例中介绍到的Lex约定。

**(1) 二义性的解决**

Lex输出总是首先将可能的最长子串与规则相匹配。如果某个子串可与两个或更多的规则匹配，则Lex的输出将找出列在行为部分中的第1个规则。如果没有规则可与任何非空子串相匹配，则缺省行为将下一个字符复制到输出中并继续下去。

**(2) C代码的插入**

1) 任何写在定义部分**%{**和**%}**之间的文本将被直接复制到外置于任意过程的输出程序之中。2) 辅助过程中的任何文本都将被直接复制到Lex 代码末尾的输出程序中。3) 将任何跟在行为部

分（在第1个**%%**之后）的正则表达式之后（中间至少有一个空格）的代码插入到识别过程**yylex**的恰当位置，并在与对应的正则表达式匹配时执行它。代表一个行为的 C代码可以既是一个C语句，也可以是一个由任何说明及由位于花括号中的语句组成的复杂的C语句。

**(3) 内部名字**

表2-3列出了在本章中所提到过的Lex内部名字，大多数都已在前面的示例中讲过了。

**表2-3 一些Lex内部名字**

| Lex 内部名字 | 含义/使用 |
|---|---|
| **lex.yy.c**或**lexyy.c** | Lex 输出文件名 |
| **yylex** | Lex 扫描例程 |
| **yytext** | 当前行为匹配的串 |
| **yyin** | Lex 输入文件（缺省：**stdin**） |
| **yyout** | Lex 输出文件（缺省：**stdout**） |
| **input** | Lex 缓冲的输入例程 |
| **ECHO** | Lex 缺省行为（将**yytext**打印到**yyout**） |

上表中有一个前面未曾提到过的特征： Lex 为一些文件备有其自身的内部名字： **yyin**和**yyout**，Lex 从这些文件中获得输入并向它们发送输出。通过标准的 Lex 输入例程**input**就可自动地从文件**yyin**中得到输入。但是在前述的示例中，却回避了内部输出文件 **yyout**，而只通过**printf**和**putchar**写到标准输出中。一个允许将输出赋到任一文件中的更好的实现方法是用**fprintf(yyout,...)**和**putc(...,yyout)**取代它们。

### 2.6.3 使用Lex 的TINY扫描程序

附录B中有一个Lex 输入文件**tiny.l**的列表，**tiny.l**将生成TINY语言的扫描程序（2.5节已描述了TINY语言的记号，参见表2-1）。下面对这个输入文件（第3000行到第3072行）做一些说明。

首先，在定义部分中，直接插入到Lex 输出中的C代码是由3个**#include**指示(**globals.h**、**util.h**和**scan.h**）及**tokenString**特性组成的。在扫描程序和其他的 TINY编译器之间有必要提供一个界面。

定义部分更深的内容还包括了定义 TINY记号的正则表达式的名字的定义。请注意，**number**的定义利用了前面定义的名字 **digit**，而**identifier**的定义利用了前面定义的**letter**。由于新行会导致增加**lineno**，所以定义还区分了新行和其他的空白格（空格和制表位，以及第3019行和第3020行）。

Lex输入的行为部分由各种记号的列表和 return语句组成，其中 **return**语句返回在**globals.h**中定义的恰当记号。在这个Lex定义中，在标识符规则之前列出了保留字规则。假若首先列出标识符规则，Lex 的二义性解决规则就会总将保留字识别为标识符。我们还可以写出与前一节中的扫描程序中相同的代码，在这里只能识别出标识符，然后再在表中查找保留字。由于单独识别的保留字使得由Lex 生成的扫描程序代码中的表格变得很大（而且扫描程序使用的存储器也会因此变得很大），因此在真正的编译中倾向于使用它。

Lex 输入还有一个“怪僻”：即使TINY注释的正则表达式很容易书写，也必须编写识别注释的代码以确保正确地更新了**lineno**。正则表达式实际是：

```
"{"[^\}]*"}"
```

（请注意，方括号中的花括号的作用是删除右花括号的元字符含义——引号在这里不起作用）[⊖]。

我们还注意到：并未为在遇到输入文件末尾时返回**EOF**编写代码。Lex 过程**yylex**在遇到**EOF**时有一个缺省行为——它返回 0值。正是由于这个原因，在 globals.h中的TokenType定义（第179行）中首先写出记号**ENDFILE**，所以它有0值。

最后，**tiny.1**文件包括了辅助过程部分中的**getToken**过程的定义（第3056行到第3072行）。虽然这个代码包含了Lex内部代码（如**yyin**和**yyout**）的一些在主程序中能更好地直接完成的特殊初始化，它还是允许直接使用Lex生成的扫描程序，而无需改变TINY编译器中的任何其他文件。实际上在生成C扫描程序**lex.yy.c**（或**lexyy.c**）后，就可编译这个文件，并可将它与其他的TINY源文件链接以生成一个基于Lex版本的编译器了。但是这个版本的编译器却缺少了以前版本的一项服务，这是因为没有源代码回应提供的行号（参见练习 **2.35**）。