# 第08章 用 flex 做词法分析

## 8.1 flex 简介

**flex** 是一个快速词法分析生成器，它可以将用户用正则表达式写的分词匹配模式构造成一个有限状态自动机（一个C函数），目前很多编译器都采用它来生成词法分析器。flex 的主页：http://flex.sourceforge.net/ 。下面以两个简单的例子来说明 flex 的使用方法。

首先安装 flex ， 可以在终端输入以下命令来安装：

```
$ sudo apt-get install flex
```

或者在其主页上下载相应版本的安装文件安装。

然后新建一个文本文件，输入以下内容：

```
%%
[0-9]+  printf("?");
#       return 0;
.       ECHO;
%%

int main(int argc, char* argv[]) {
    yylex();
    return 0;
}

int yywrap() {
    return 1;
}
```

将此文件另存为 hide-digits.l 。注意此文件中的 %% 必须在本行的最前面（即 %% 前面不能有任何空格）。

之后，在终端输入：

```
$ flex hide-digits.l
```

此时目录下多了一个 “lex.yy.c” 文件，把这个 C 文件编译并运行一遍：

```
$ gcc -o hide-digits lex.yy.c
$ ./hide-digits
```

然后在终端不停的敲入任意键并回车，可以发现，敲入的内容中，除数字外的字符都被原样的输出了，而每串数字字符都被替换成 **?** 了。最后敲入 **#** 后程序退出了。如下：

```
eruiewdkfj
eruiewdkfj
1245
?
fdsaf4578
fdsaf?
...
#
```

也就是说，**hide-digits** 这个程序的作用就是不停从标准输入（键盘）中读入字符，将其中的数字串替换成 **?** 后再输出到标准输出（终端），当遇到 **#** 后程序退出。想象一下，如果要手工编程实现一个这样的程序，需要的代码量肯定比 **hide-digits.l** 文件多很多吧。

当在命令行中运行 flex 时，第二个命令行参数（此处是 hide-digits.l）是提供给 flex 的分词模式文件， 此模式文件中主要是用户用正则表达式写的分词匹配模式，用flex 会将这些正则表达式翻译成 C 代码格式的函数 **yylex** ，并输出到 **lex.yy.c** 文件中，该函数可以看成一个有限状态自动机。

下面再来详细解释一下 **hide-digits.l** 文件中的代码，首先第一段是：

```
%%
[0-9]+  printf("?");
#       return 0;
.       ECHO;
%%
```

flex 模式文件中，%% 和 %% 之间的内容被称为 **规则（rules）**，本文件中每一行都是一条规则，每条规则由 **匹配模式（pattern）** 和 **事件（action）** 组成， 模式在前面，用正则表达式表示，事件在后面，即 C 代码。每当一个模式被匹配到时，后面的 C 代码被执行。

简单来说，flex 会将本段内容翻译成一个名为 **yylex** 的函数，该函数的作用就是扫描输入文件（默认情况下为标准输入），当扫描到一个完整的、最长的、可以和某条规则的正则表达式所匹配的字符串时，该函数会执行此规则后面的 C 代码。如果这些 C 代码中没有 **return** 语句，则执行完这些 C 代码后， **yylex** 函数会继续运行，开始下一轮的扫描和匹配。

当有多条规则的模式被匹配到时， **yylex** 会选择匹配长度最长的那条规则，如果有匹配长度相等的规则，则选择排在最前面的规则。

第二段中的 main 函数是程序的入口， flex 会将这些代码原样的复制到 **lex.yy.c** 文件的最后面。最后一行的 yywrap 函数的作用后面再讲，总之就是 flex 要求有这么一个函数。

```
int main(int argc, char *argv[]) {
    yylex();
    return 0;
}

int yywrap() { return 1; }
```

因此，程序开始运行后，就开始执行 **yylex** 函数，然后开始扫描标准输入。当扫描出一串数字时，**[0-9]+** 被匹配到，因此执行了 printf("?") ，当扫描到其他字符时，若不是 # ，则 . 被匹配，后面的 ECHO 被执行， ECHO 是 flex 提供的一个宏，作用是将匹配到的字符串原样输出，当扫描到 # 后， # 被匹配， return 0 被执行， yylex 函数返回到 main 函数，之后程序结束。

下面再来看一个稍微复杂一点的例子：

```
%{
#define T_WORD 1
int numChars = 0, numWords = 0, numLines = 0;
%}

WORD            ([^ \t\n\r\a]+)

%%
\n                      { numLines++; numChars++; }
{WORD}          { numWords++; numChars += yyleng; return T_WORD; }
<<EOF>>         { return 0; }
.                       { numChars++; }
%%
```

```
int main() {
        int token_type;
        while (token_type = yylex()) {
                printf("WORD:\t%s\n", yytext);
        }
        printf("\nChars\tWords\tLines\n");
        printf("%d\t%d\t%d\n", numChars, numWords, numLines);
        return 0;
}

int yywrap() {
        return 1;
}
```

将此文件另存为 `word-spliter.l` 。注意此文件中的 **%{** 和 **%}** 必须在本行的最前面（前面不能有空格），同时，注意 **%}** 不要写成 **}%** 了。在终端输入：

```
$ flex word-spliter.l
$ gcc -o word-spliter lex.yy.c
$ ./word-spliter < word-spliter.l
```

将输出：

```
WORD:       %{
WORD:       #define
...
WORD:       }

Chars       Words   Lines
470 70      27
```

可见此程序其实就是一个原始的分词器，它将输入文件分割成一个个的 **WORD** 再输出到终端，同时统计输入文件中的字符数、单词数和行数。此处的 **WORD** 指一串连续的非空格字符。

下面，详细介绍 `flex` 输入文件的完整格式，同时解释一下本文件的代码。一个完整的 `flex` 输入文件的格式为：

```
%{
Declarations
%}
Definitions
```

```
%%
Rules
%%
User subroutines
```

输入文件的第 1 段 **%{** 和 **%}** 之间的为 **声明 (Declarations)** ，都是 C 代码，这些代码会被原样的复制到 **lex.yy.c** 文件中，一般在这里声明一些全局变量和函数，这样在后面可以使用这些变量和函数。

第 2 段 **%} 和 %%** 之间的为 **定义 (Definitions)**，在这里可以定义正则表达式中的一些名字，可以在 **规则 (Rules)** 段被使用，如本文件中定义了 **WORD** 为 **([^ \t\n\r\a]+)** ， 这样在后面可以用 **{WORD}** 代替这个正则表达式。

第 3 段为 **规则 (Rules)** 段，上一个例子中已经详细说明过了。

第 4 段为 **用户定义过程 (User subroutines)** 段，也都是 C 代码，本段内容会被原样复制到 **yylex.c** 文件的最末尾，一般在此定义第 1 段中声明的函数。

以上 4 段中，除了 **Rules** 段是必须要有的外，其他三个段都是可选的。

输入文件中最后一行的 **yywrap** 函数的作用是将多个输入文件打包成一个输入，当 **yylex** 函数读入到一个文件结束（EOF）时，它会向 yywrap 函数询问， yywrap 函数返回 1 的意思是告诉 **yylex** 函数后面没有其他输入文件了，此时 yylex 函数结束，yywrap 函数也可以打开下一个输入文件，再向 yylex 函数返回 0 ，告诉它后面还有别的输入文件，此时 yylex 函数会继续解析下一个输入文件。总之，由于我们不考虑连续解析多个文件，因此此处返回 1 。

和上一个例子不同的是，本例中的 action 中有 return 语句，而 **main** 函数内是一个 while 循环，只要 yylex 函数的返回值不为 0 ，则 yylex 函数将被继续调用，此时将从下一个字符开始新一轮的扫描。

另外，本例中使用到了 flex 提供的两个全局变量 **yytext** 和 **yyleng**，分别用来表示刚刚匹配到的字符串以及它的长度。

为方便编译，使用 makefile 进行编译及运行：

```
run: word-spliter
	./word-spliter < word-spliter.l

word-spliter: lex.yy.c
	gcc -o $@ $<

lex.yy.c: word-spliter.l
	flex $<
```

将以上内容保存为 `makefile` ，和 `word-spliter.l` 文件放在当前目录，再在终端输入：

```
make
```

将输出和前面一样的内容。 makefile 的语法本站就不介绍了，后文中的大部分程序都将使用 makefile 编译。

好了， flex 的使用就简单介绍到这，以上介绍的功能用来解析 TinyC 文件已经差不多够了。有兴趣的读者可以到其主页上去阅读一下它的手册，学习更强大的功能。下面介绍如何使用 flex 对 TinyC 源文件进行词法分析。

## 8.2 使用 flex 对 TinyC 源文件进行词法分析

上一节的第二个例子 **word-spliter** 就是一个原始的分词器，在此例的框架上加以扩展就可以做为 TinyC 的词法分析器了。

word-spliter 中只有 WORD 这一种类型的 token ，所有连续的非空格字符串都是一个 WORD ，它的正则表达式非常简单： **[^ \t\n\r\a]+** 。 该程序中为 WORD 类型的 token 定义了一个值为 1 的编号： T_WORD ，每当扫描出一个完整的 WORD 时，就向 main 函数返回 T_WORD ，遇到文件结束则返回 0 。 main 函数则根据 yylex 函数的返回值进行不同的处理。

从 word-spliter 程序的框架和流程中可以看出，词法分析器的扩展方法非常简单：

(1) 列出 TinyC 中所有类型的 token;

(2) 为每种类型的 token 分配一个唯一的编号，同时写出此 token 的正则表达式;

(3) 写出每种 token 的 rule （相应的 pattern 和 action ）。

TinyC 中的 token 的种类非常少，按其词法特性，分为以下三大类。

第 1 类为单字符运算符，一共 15 种：

```
+ * - / % = , ; ! < > ( ) { }
```

第 2 类为双字符运算符和关键字，一共 16 种：

```
⩽, ⩾, ==, ≠, &&, ||
void, int, while, if, else, return, break, continue, print, readint
```

第 3 类为整数常量、字符串常量和标识符（变量名和函数名），一共 3 种。

除第 3 类 token 的正则表达式稍微麻烦一点外，第 1 、 2 类 token 的正则表达式就是这些运算符或关键字的字面值。

token 的编号原则为：单字符运算符的 token 编号就是其字符的数值，其他类型的 token 则从 256 开始编号。

各类 token 的正则表达式及相应的 action 见下面的 scaner.l 文件，该文件的框架和上一节中的 word-spliter.l 是完全一样的，只不过 token 的类别多了。

```
%{
#include "token.h"
int cur_line_num = 1;
void init_scanner();
void lex_error(char* msg, int line);
%}

/* Definitions, note: \042 is '"' */
INTEGER             ([0-9]+)
UNTERM_STRING       (\042[^\042\n]*)
STRING              (\042[^\042\n]*\042)
IDENTIFIER          ([_a-zA-Z][_a-zA-Z0-9]*)
OPERATOR            ([+*-/%=,;!<>(){}])
SINGLE_COMMENT1     ("//"[^\n]*)
SINGLE_COMMENT2     ("#"[^\n]*)

%%
```

```
[\n]                { cur_line_num++;                        }
[ \t\r\a]+          { /* ignore all spaces */                }
{SINGLE_COMMENT1}   { /* skip for single line comment */    }
{SINGLE_COMMENT2}   { /* skip for single line commnet */    }

{OPERATOR}          { return yytext[0];         }

"⩽"                 { return T_Le;              }
"⩾"                 { return T_Ge;              }
"⩵"                 { return T_Eq;              }
"≠"                 { return T_Ne;              }
"&&"                { return T_And;             }
"||"                { return T_Or;              }
"void"              { return T_Void;            }
"int"               { return T_Int;             }
"while"             { return T_While;           }
"if"                { return T_If;              }
"else"              { return T_Else;            }
"return"            { return T_Return;          }
"break"             { return T_Break;           }
"continue"          { return T_Continue;        }
"print"             { return T_Print;           }
"readint"           { return T_ReadInt;         }

{INTEGER}           { return T_IntConstant;     }
{STRING}            { return T_StringConstant;  }
{IDENTIFIER}        { return T_Identifier;      }

<<EOF>>             { return 0; }

{UNTERM_STRING}     { lex_error("Unterminated string constant", cur_
.                   { lex_error("Unrecognized character", cur_line_n

%%

int main(int argc, char* argv[]) {
    int token;
    init_scanner();
    while (token = yylex()) {
        print_token(token);
        puts(yytext);
    }
    return 0;
}

void init_scanner() {
    printf("%-20s%s\n", "TOKEN-TYPE", "TOKEN-VALUE");
```

```
    printf("-----------------------------------------------\n");
}

void lex_error(char* msg, int line) {
    printf("\nError at line %-3d: %s\n\n", line, msg);
}

int yywrap(void) {
    return 1;
}
```

---

上面这个文件中，需要注意的是，正则表达式中，用双引号括起来的字符串就是原始字符串，里面的特殊字符是不需要转义的，而双引号本身必须转义（必须用 **\"** 或 **\042** ），这是 flex 中不同于常规的正则表达式的一个特性。

除单字符运算符外的 token 的编号则在下面这个 **token.h** 文件，该文件中同时提供了一个 print_token 函数，可以根据 token 的编号打印其名称。

---

```
#ifndef TOKEN_H
#define TOKEN_H

typedef enum {
    T_Le = 256, T_Ge, T_Eq, T_Ne, T_And, T_Or, T_IntConstant,
    T_StringConstant, T_Identifier, T_Void, T_Int, T_While,
    T_If, T_Else, T_Return, T_Break, T_Continue, T_Print,
    T_ReadInt
} TokenType;

static void print_token(int token) {
    static char* token_strs[] = {
        "T_Le", "T_Ge", "T_Eq", "T_Ne", "T_And", "T_Or", "T_IntConst
        "T_StringConstant", "T_Identifier", "T_Void", "T_Int", "T_Wh
        "T_If", "T_Else", "T_Return", "T_Break", "T_Continue", "T_Pr
        "T_ReadInt"
    };

    if (token < 256) {
        printf("%-20c", token);
    } else {
        printf("%-20s", token_strs[token-256]);
    }
}

#endif
```

下面来编译一下这两个文件， makefile 文件为：

```
out: scanner

scanner: lex.yy.c token.h
	gcc -o $@ $<

lex.yy.c: scanner.l
	flex $<
```

将以上 3 个文件保存在终端的当前目录，再输入 make ，编译后生成了 **scanner** 文件。

下面来测试一下这个词法分析器，将 samples.zip 文件下载并解压到 samples 目录，此文件包中有很多测试文件，我们先测试一下其中的一个文件，输入：

```
$ ./scanner < samples/sample_6_function.c > out.txt
```

再打开 out.txt 文件看看，可以看出 sample_6_function.c 文件的所有 token 都被解析出来了：

```
TOKEN-TYPE          TOKEN-VALUE
-------------------------------------------------
T_Int               int
T_Identifier        main
(                   (
)                   )
...
```

下面全部测试一下这些文件，在终端输入以下内容：

```
for src in $(ls samples/*.c); do ./scanner < $src > $src.lex; done
```

再在终端输入： bash test.sh 。之后，查看一下 samples 目录下新生成的 ".lex" 文件。可以看出所有源文件都被解析完成了。

TinyC 语言中只有两类词法错误，一种是未结束的字符串，即只有前面一个双引号的字符串，另外一种就是非法字符，如 ~ @ 等（双引号内部的除外），scanner.l 文件中可以识别出这两种词法错误，同时定位出错误所在的行，详见该文件的 Rules 段的最后两条 Rule 。

## 第 8 章完