

# Yacc 与 Lex 快速入门

Lex 与 Yacc 介绍

级别：初级

Ashish Bansal ([abansal@ieee.org](mailto:abansal@ieee.org)), 软件工程师, Sapient 公司

2000 年 11 月 01 日

Lex 和 Yacc 是 UNIX 两个非常重要的、功能强大的工具。事实上，如果你熟练掌握 Lex 和 Yacc 的话，它们的强大功能使创建 FORTRAN 和 C 的编译器如同儿戏。Ashish Bansal 为您详细的讨论了编写自己的语言和编译器所用到的这两种工具，包括常规表达式、声明、匹配模式、变量、Yacc 语法和解析器代码。最后，他解释了怎样把 Lex 和 Yacc 结合起来。

Lex 代表 Lexical Analyzar。Yacc 代表 Yet Another Compiler Compiler。让我们从 Lex 开始吧。

## Lex

Lex 是一种生成扫描器的工具。扫描器是一种识别文本中的词汇模式的程序。这些词汇模式（或者常规表达式）在一种特殊的句子结构中定义，这个我们一会儿就要讨论。

一种匹配的常规表达式可能会包含相关的动作。这一动作可能还包括返回一个标记。当 Lex 接收到文件或文本形式的输入时，它试图将文本与常规表达式进行匹配。它一次读入一个输入字符，直到找到一个匹配的模式。如果能够找到一个匹配的模式，Lex 就执行相关的动作（可能包括返回一个标记）。另一方面，如果没有可以匹配的常规表达式，将会停止进一步的处理，Lex 将显示一个错误消息。

Lex 和 C 是强耦合的。一个 .lex 文件（Lex 文件具有 .lex 的扩展名）通过 lex 公用程序来传递，并生成 C 的输出文件。这些文件被编译为词法分析器的可执行版本。

---

## Lex 的常规表达式

常规表达式是一种使用元语言的模式描述。表达式由符号组成。符号一般是字符和数字，但是 Lex 中还有一些具有特殊含义的其他标记。下面两个表格定义了 Lex 中使用的一些标记并给出了几个典型的例子。

### 用 Lex 定义常规表达式

字符	含义
A-Z, 0-9, a-z	构成了部分模式的字符和数字。
.	匹配任意字符，除了 \n。
-	用来指定范围。例如：A-Z 指从 A 到 Z 之间的所有字符。
[ ]	一个字符集合。匹配括号内的 任意 字符。如果第一个字符是 ^ 那么它表示否定模式。例如：[abC] 匹配 a, b, 和 C 中的任何一个。
*	匹配 0 个或者多个上述的模式。
+	匹配 1 个或者多个上述模式。
?	匹配 0 个或 1 个上述模式。
\$	作为模式的最后一个字符匹配一行的结尾。
{ }	指出一个模式可能出现的次数。 例如：A{1,3} 表示 A 可能出现 1 次或 3 次。

\	用来转义元字符。同样用来覆盖字符在此表中定义的特殊意义，只取字符的本意。
^	否定。
	表达式间的逻辑或。
"<一些符号>"	字符的字面含义。元字符具有。
/	向前匹配。如果在匹配的模版中的“/”后跟有后续表达式，只匹配模版中“/”前面的部分。如：如果输入 A01，那么在模版 A0/1 中的 A0 是匹配的。
()	将一系列常规表达式分组。

### 常规表达式举例

常规表达式	含义
joke[rs]	匹配 jokes 或 joker。
A{1,2}shis+	匹配 AAshis, Ashis, AAshi, Ashi。
(A[b-e])+	匹配在 A 出现位置后跟随的从 b 到 e 的所有字符中的 0 个或 1 个。

Lex 中的标记声明类似 C 中的变量名。每个标记都有一个相关的表达式。（下表中给出了标记和表达式的例子。）使用这个表中的例子，我们就可以编一个字数统计的程序了。我们的第一个任务就是说明如何声明标记。

### 标记声明举例

标记	相关表达式	含义
数字 (number)	([0-9])+	1个或多个数字
字符 (chars)	[A-Za-z]	任意字符
空格 (blank)	" "	一个空格
字(word)	(chars)+	1个或多个 <i>chars</i>
变量 (variable)	(字符)+(数字)*(字符)*(数字)*	

## Lex 编程

Lex 编程可以分为三步：

1. 以 Lex 可以理解的格式指定模式相关的动作。
2. 在这一文件上运行 Lex，生成扫描器的 C 代码。
3. 编译和链接 C 代码，生成可执行的扫描器。

注意: 如果扫描器是用 Yacc 开发的解析器的一部分，只需要进行第一步和第二步。关于这一特殊问题的帮助请阅读 [Yacc](#)和 [将 Lex 和 Yacc 结合起来](#)部分。

现在让我们看一看 Lex 可以理解的程序格式。一个 Lex 程序分为三个段：第一段是 C 和 Lex 的全局声明，第二段包括模式（C 代码），第三段是补充的 C 函数。例如，第三段中一般都有 main() 函数。这些段以%%来分界。那么，回到字数统计的 Lex 程序，让我们看一下程序不同段的构成。

---

## C 和 Lex 的全局声明

这一段中我们可以增加 C 变量声明。这里我们将为字数统计程序声明一个整型变量，来保存程序统计出来的字数。我们还将进行 Lex 的标记声明。

### 字数统计程序的声明

```
%{
    int wordCount = 0;
}%
chars [A-Za-z\_\'\".\\"]
numbers ([0-9])+
delim [" \"\n\t]
whitespace {delim}+
words {chars}+
%%
```

两个百分号标记指出了 Lex 程序中这一段的结束和三段中第二段的开始。

---

## Lex 的模式匹配规则

让我们看一下 Lex 描述我们所要匹配的标记的规则。（我们将使用 C 来定义标记匹配后的动作。）继续看我们的字数统计程序，下面是标记匹配的规则。

### 字数统计程序中的 Lex 规则

```
{words} { wordCount++; /*
    increase the word count by one*/ }
{whitespace} { /* do
    nothing*/ }
{numbers} { /* one may
    want to add some processing here*/ }
%%
```

---

## C 代码

Lex 编程的第三段，也就是最后一段覆盖了 C 的函数声明（有时是主函数）。注意这一段必须包括 `yywrap()` 函数。Lex 有一套可供使用的函数和变量。其中之一就是 `yywrap`。一般来说，`yywrap()` 的定义如下例。我们将在 [高级 Lex](#) 中探讨这一问题。

### 字数统计程序的 C 代码段

```
void main()
{
    yylex(); /* start the
    analysis*/
    printf(" No of words:
    %d\n", wordCount);
}
```

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

上一节我们讨论了 Lex 编程的基本元素，它将帮助你编写简单的词法分析程序。在 [高级 Lex](#) 这一节中我们将讨论 Lex 提供的函数，这样你就能编写更加复杂的程序了。

## 将它们全部结合起来

.lex文件是 Lex 的扫描器。它在 Lex 程序中如下表示：

```
$ lex <file name.lex>
```

这生成了 lex.yy.c 文件，它可以用 C 编译器来进行编译。它还可以用解析器来生成可执行程序，或者在链接步骤中通过选项 `-ll` 包含 Lex 库。

这里是一些 Lex 的标志：

- | `-c`表示 C 动作，它是缺省的。
- | `-t`写入 lex.yy.c 程序来代替标准输出。
- | `-v`提供一个两行的统计汇总。
- | `-n`不打印 `-v` 的汇总。

## 高级 Lex

Lex 有几个函数和变量提供了不同的信息，可以用来编译实现复杂函数的程序。下表中列出了一些变量和函数，以及它们的使用。详尽的列表请参考 Lex 或 Flex 手册（见后文的 [资源](#)）。

### Lex 变量

yyin	FILE* 类型。 它指向 lexer 正在解析的当前文件。
yyout	FILE* 类型。 它指向记录 lexer 输出的位置。 缺省情况下，yyin 和 yyout 都指向标准输入和输出。
yytext	匹配模式的文本存储在这一变量中（char*）。
yyldeng	给出匹配模式的长度。
yylineno	提供当前的行数信息。（lexer不一定支持。）

### Lex 函数

yylex()	这一函数开始分析。 它由 Lex 自动生成。
yywrap()	这一函数在文件（或输入）的末尾调用。如果函数的返回值是1，就停止解析。 因此它可以用来解析多个文件。代码可以写在第三段，这就能解析多个文件。 方法是使用 yyin 文件指针（见上表）指向不同的文件，直到所有的文件都被解析。最后，yywrap() 可以返回 1 来表示解析的结束。
yyless(int n)	这一函数可以用来送回除了前 n? 个字符外的所有读出标记。
yyomore()	这一函数告诉 Lexer 将下一个标记附加到当前标记后。

对 Lex 的讨论就到这里。下面我们来讨论 Yacc...

## Yacc

Yacc 代表 Yet Another Compiler Compiler。Yacc 的 GNU 版叫做 Bison。它是一种工具，将任何一种编程语言的所有语法翻译成针对此种语言的 Yacc 语 法解析器。它用巴科斯范式(BNF, Backus Naur Form)来书写。按照惯例，Yacc 文件有 .y 后缀。编译行如下调用 Yacc 编译器：

```
$ yacc <options>
    <filename ending with .y>
```

在进一步阐述以前，让我们复习一下什么是语法。在上一节中，我们看到 Lex 从输入序列中识别标记。如果你在查看标记序列，你可能想在这一序列出现时执行某一动作。这种情况下有效序列的规范称为语法。Yacc 语法文件包括这一语法规范。它还包含了序列匹配时你想要做的事。

为了更加说清这一概念，让我们以英语为例。这一套标记可能是：名词, 动词, 形容词等等。为了使用这些标记造一个语法正确的句子，你的结构必须符合一定的规则。一个简单的句子可能是名词+动词或者名词+动词+名词。(如 I care. See spot run.)

所以在我们这里，标记本身来自语言 (Lex)，并且标记序列允许用 Yacc 来指定这些标记(标记序列也叫语法)。

用 Yacc 来创建一个编译器包括四个步骤：

1. 通过在语法文件上运行 Yacc 生成一个解析器。
2. 说明语法：
  - 编写一个 .y 的语法文件 (同时说明 C 在这里要进行的动作)。
  - 编写一个词法分析器来处理输入并将标记传递给解析器。这可以使用 Lex 来完成。
  - 编写一个函数，通过调用 yyparse() 来开始解析。
  - 编写错误处理例程 (如 yyerror())。
3. 编译 Yacc 生成的代码以及其他相关的源文件。
4. 将目标文件链接到适当的可执行解析器库。

### 终端和非终端符号

**终端符号**：代表一类在语法结构上等效的标记。终端符号有三种类型：

**命名标记**：这些由 %token 标识符来定义。按照惯例，它们都是大写。

**字符标记**：字符常量的写法与 C 相同。例如，-- 就是一个字符标记。

**字符串标记**：写法与 C 的字符串常量相同。例如，"<<" 就是一个字符串标记。

lexer 返回命名标记。

**非终端符号**：是一组非终端符号和终端符号组成的符号。按照惯例，它们都是小写。在例子中，file 是一个非终端标记而 NAME 是一个终端标记。

## 用 Yacc 编写语法

如同 Lex 一样，一个 Yacc 程序也用双百分号分为三段。它们是：声明、语法规则和 C 代码。我们将解析一个格式为 姓名 = 年龄的文件作为例子，来说明语法规则。我们假设文件有多个姓名和年龄，它们以空格分隔。在看 Yacc 程序的每一段时，我们将为我们的例子编写一个语法文件。

## C 与 Yacc 的声明

C 声明可能会定义动作中使用的类型和变量，以及宏。还可以包含头文件。每个 Yacc 声明段声明了终端符号和非终端符号 (标记) 的名称，还可能描述操作符优先级和针对不同符号的数据类型。lexer (Lex) 一般返回这些标记。所有这些标记都必须在 Yacc 声明中进行说明。

在文件解析的例子中我们感兴趣的是这些标记：name, equal sign, 和 age。Name 是一个完全由字符组成的值。 Age 是数字。于是声明段就会像这样：

### 文件解析例子的声明

```
%
#define char* string; /*
to specify token types as char* */
#define YYSTYPE string /*
a Yacc variable which has the value of returned token */
%}
%token NAME EQ AGE
%%
```

你可能会觉得 YYSTYPE 有点奇怪。但是类似 Lex, Yacc 也有一套变量和函数可供用户来进行功能扩展。 YYSTYPE 定义了用来将值从 lexer 拷贝到解析器或者 Yacc 的 yylval (另一个 Yacc 变量) 的类型。默认的类型是 int。由于字符串可以从 lexer 拷贝, 类型被重定义为 char\*。关于 Yacc 变量的详细讨论, 请参考 Yacc 手册 (见 [资源](#))。

---

## Yacc 语法规则

Yacc 语法规则具有以下一般格式：

```
result: components { /*
    action to be taken in C */ }
;
```

在这个例子中, result 是规则描述的非终端符号。Components 是根据规则放在一起的不同的终端和非终端符号。如果匹配特定序列的话 Components 后面可以跟随要执行的动作。考虑如下的例子：

```
param : NAME EQ NAME {
    printf("\tName: %s\tValue(name): %s\n", $1, $3); }
    | NAME EQ VALUE {
    printf("\tName: %s\tValue(value): %s\n", $1, $3); }
;
```

如果上例中序列 NAME EQ NAME 被匹配, 将执行相应的 {} 括号中的动作。这里另一个有用的就是 \$1 和 \$3 的使用, 它们引用了标记 NAME 和 NAME (或者第二行的 VALUE) 的值。lexer 通过 Yacc 的变量 yylval 返回这些值。标记 NAME 的 Lex 代码是这样的：

```
char [A-Za-z]
name {char}+
%%
{name} { yylval = strdup(yytext);
return NAME; }
```

文件解析例子的规则段是这样的：

### 文件解析的语法

```
file : record file
    | record
    ;
record: NAME EQ AGE {
```

```
printf("%s is now %s years old!!!", $1, $3);}
;
%%
```

## 附加 C 代码

现在让我们看一下语法文件的最后一段，附加 C 代码。（这一段是可选的，如果有人想要略过它的话：）一个函数如 `main()` 调用 `yyparse()` 函数（Yacc 中 Lex 的 `yylex()` 等效函数）。一般来说，Yacc 最好提供 `yyerror(char msg)` 函数的代码。当解析器遇到错误时调用 `yyerror(char msg)`。错误消息作为参数来传递。一个简单的 `yyerror(char*)` 可能是这样的：

```
int yyerror(char* msg)
{
    printf("Error: %s
    encountered at line number: %d\n", msg, yylineno);
}
```

`yylineno` 提供了行数信息。

这一段还包括文件解析例子的主函数：

### 附加 C 代码

```
void main()
{
    yyparse();
}
int yyerror(char* msg)
{
    printf("Error: %s
    encountered \n", msg);
}
```

要生成代码，可能用到以下命令：

```
$ yacc -d <filename.y>
```

这生成了输出文件 `y.tab.h` 和 `y.tab.c`，它们可以用 UNIX 上的任何标准 C 编译器来编译（如 `gcc`）。

## 命令行的其他常用选项

- 1. `'-d', '--defines'`: 编写额外的输出文件，它们包含这些宏定义：语法中定义的标记类型名称，语义的取值类型 `YYSTYPE`，以及一些外部变量声明。如果解析器输出文件名叫 `'name.c'`，那么 `'-d'` 文件就叫做 `'name.h'`。如果你想将 `yylex` 定义放到独立的源文件中，你需要 `'name.h'`，因为 `yylex` 必须能够引用标记类型代码和 `yylval` 变量。
- 1. `'-b file-prefix', '--file-prefix=prefix'`: 指定一个所有 Yacc 输出文件名都可以使用的前缀。选择一个名字，就如输入文件名叫 `'prefix.c'`。
- 1. `'-o outfile', '--output-file=outfile'`: 指定解析器文件的输出文件名。其他输出文件根据 `'-d'` 选项描述的输出文件来命名。

Yacc 库通常在编译步骤中自动被包括。但是它也能被显式的包括，以便在编译步骤中指定 `-ly` 选项。这种情况下的编译命令行是：



```
$ cc <source file
names> -ly
```

## 将 Lex 与 Yacc 结合起来

到目前为止我们已经分别讨论了 Lex 和 Yacc。现在让我们来看一下他们是怎样结合使用的。

一个程序通常在每次返回一个标记时都要调用 `yylex()` 函数。只有在文件结束或者出现错误标记时才会终止。

一个由 Yacc 生成的解析器调用 `yylex()` 函数来获得标记。`yylex()` 可以由 Lex 来生成或完全由自己来编写。对于由 Lex 生成的 lexer 来说,要和 Yacc 结合使用,每当 Lex 中匹配一个模式时都必须返回一个标记。因此 Lex 中匹配模式时的动作一般格式为:

```
{pattern} { /* do smthg*/
return TOKEN_NAME; }
```

于是 Yacc 就会获得返回的标记。当 Yacc 编译一个带有 `_d` 标记的 `.y` 文件时,会生成一个头文件,它对每个标记都有 `#define` 的定义。如果 Lex 和 Yacc 一起使用的话,头文件必须在相应的 Lex 文件 `.lex` 中的 C 声明段中包括。

让我们回到名字和年龄的文件解析例子中,看一看 Lex 和 Yacc 文件的代码。

### Name.y - 语法文件

```
%
typedef char* string;
#define YYSTYPE string
%}
%token NAME EQ AGE
%%
file : record file
| record
;
record : NAME EQ AGE {
printf("%s is %s years old!!!\n", $1, $3); }
;
%%
int main()
{
yyparse();
return 0;
}
int yyerror(char *msg)
{
printf("Error
encountered: %s \n", msg);
}
```

### Name.lex - Lex 的解析器文件

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



```
#include <stdio.h>
#include <string.h>
extern char* yyval;
%}
char [A-Za-z]
num [0-9]
eq [=]
name {char}+
age {num}+
%%
{name} { yyval = strdup(yytext);
return NAME; }
{eq} { return EQ; }
{age} { yyval = strdup(yytext);
return AGE; }
%%
int yywrap()
{
return 1;
}
```

作为一个参考，我们列出了 *y.tab.h*, Yacc 生成的头文件。

### y.tab.h - Yacc 生成的头文件

```
# define NAME 257
# define EQ 258
# define AGE 259
```

我们对于 Lex 和 Yacc 的讨论到此为止。今天你想要编译什么语言呢？

## 参考资料

- 您可以参阅本文在 developerWorks 全球站点上的 [英文原文](#).
- [Lex and Yacc](#), Levine, Mason 和 Branson, O Reilly 及其合作公司, 2nd Ed.
- [Program Development in UNIX](#), J. T. Shen, Prentice-Hall India.
- [Compilers: Principles, Techniques and Tools](#), Ahoo, Sethi 和 Ullman, Addison-Wesley Pub. Co., 1985, 11.
- [Lex and Yacc and compiler writing](#)指导。
- Java 版的 Lex 指导, 叫做 [Jlex](#)。
- 使用 Lex 和 Yacc 的 [formalizing a grammar](#)实例。

## 关于作者

Ashish Bansal 具有印度瓦腊纳西 Banaras Hindu 大学技术学院的电子与通信工程学士学位。他目前是 Sapient 公司的软件工程师。他的Email是 [abansal@sapient.com](mailto:abansal@sapient.com)。

IBM 公司保留在 developerWorks 网站上发表的内容的著作权。未经IBM公司或原始作者的书面明确许可, 请勿转载。如果您希望转载, 请通过 [提交转载请求表单](#) 联系我们的编辑团队。