第一章 lex 和 yacc [1]

本章内容

- 最简单的 lex 程序
- 用 lex 识别单词
- 语法
- 运行 lex 和 yacc
- lex 和手写的词法分析

lex 和 yacc 可以帮助你编写程序转换结构化输入。应用程序范围很广——既包括从输入文件中寻找模式的简单文本搜索程序，也包括将源程序变换为最佳的目标代码的 C 编译程序等。

在具有结构化输入的程序中，反复出现的两个任务是：将输入分隔成有意义的单元，然后找出这些单元之间的关系。对于文本搜索程序，这些单元可能是文本行，这些行包不包含目标匹配字符串存在着很大的区别。对于 C 程序，这些单元是变量名、常量、字符串、操作符、标点符号等。划分单元（通常称为标记）也称为词法分析（*lexical anslysis*，或者简写为 *lexing*）。lex 使用一系列对可能标记的描述，产生一个能识别那些标记的 C 例程（我们称为词法分析器，词法分析程序(lexer) ，或称为扫描程序）。赋予 lex 的那些描述称为 lex 规范。

lex 使用的标记描述称为正则表达式，它是 *grep* 和 *egrep* 命令使用的常见模式的扩展版本。lex 将这些正则表达式转变为词法分析程序能够用来极快地扫描输入文本的形式，而且速度不依赖于词法分析程序尝试匹配的表达式的数量。lex 词法分析程序儿乎总是比用 C 语言手工编写的词法分析程序快。

当将输入拆分成标记时，程序通常需要建立标记之间的关系。C 编译程序需要找到程序中的表达式、语句、声明、程序块和过程。这个任务称为分析，即定义程序能够理解的关系的规则列表，也就是语法。yacc 采用简明的语法描述并产生一个能分析语法的 C 例程，即分析程序。yacc 分析程序自动检测输入的标记序列是否匹配语法中的某条规则，并且一旦输入不匹配任一规则，它就会检测语法错误。yacc 分析程序一般并不像手写的分析程序那样快，但它使编写和修改分析程序更容易，因此速度上的损失是值得的。一个程序在分析程序中用去的时间通常可以忽略不计。

当任务涉及将输入拆分成单元并且建立那些单元之间的关系时，应该考虑使用 lex 和 yacc 。（搜索程序很简单，它不需要做任何分析，所以它使用 lex 但不需要 yacc。在第三章会再次提到，届时将只使用 lex 而不必用 yacc 来构建几个应用程序。）

[1] Jobn R.Levine, Tony Mason, Doug Brown 著，杨作梅，张旭东等译). 整理: qy8087@gmail.com

至此，我们希望已经激起了读者的学习兴趣。这一章并不是 lex 和 yacc 的完整指南，我们只对 lex 和 yacc 的用法做一个初步的介绍。

最简单的 lex 程序

下面的 lex 程序将它的标准输入拷贝到标准输出：

```
%%
.|\n  ECHO;
%%
```

它的行为非常类似于不带参数运行的 UNIX *cat* 命令。

lex 自动生成实际的 C 程序代码，这些代码负责处理读输入文件，有时（正如上述情况下）也负责写输出。

使用 lex 和 yacc 无论是构建程序的一部分，还是构建辅助编程的工具，一旦你掌握了它们，就会发现它们的价值：处理问题时它们能简化输入的困难，提供更易维护的编码库，并且能很容易地 “ 调整 ” 出程序的正确语义。

用 lex 识别单词

让我们构建一个识别不同类型英语单词的简单程序。先标识词性（名词、动词等），然后再扩展到处理符合简单的英语语法的多个单词的句子。

先列出要识别的一组动词：

is	am	are	were
was	be	being	been
do	does	did	will
would	should	can	could
has	have	had	go

例 1-1 展示了识别这些动词的简单 lex 规范。

例 1-1：单词识别程序 ch1-02.l

```
%{
/*
 * 这个例子演示了（非常）简单的识别
 * 动词/非动词
 */
%}
%%
```

```
[\t ]+      /* 忽略空白 */ ;
is |
am |
are |
were |
was |
be |
being |
been |
do |
does |
did |
will |
would |
should |
can |
could |
has |
have |
had |
go    { printf("%s: is a verb\n", yytext); }

[a-zA-Z]+   { printf("%s: is not a verb\n", yytext); }

.|\n     { ECHO; /* 通常的默认状态 */ }
%%

main()
{
  yylex();
}
```

下面是编译和运行这个程序时进行的操作，键入的文字采用黑体字。

```
% example1
did I have fun?
did: is a verb
I: is not a verb
have: is a verb
fun: is not a verb
?
^D
%
```

我们从第一部分开始解释正在运行的内容：

```
%{
/*
 * 这个例子演示了（非常）简单的识别
 * 动词/非动词
 */
%}
```

第一部分（定义段，即定义部分）介绍了将拷贝到最终程序中的原始 C 程序代码。例如，如果有后来文件中的代码必须包含的头文件，那么这部分尤其重要。用特殊的定界符“ %{ ”和“ %} ”括起 C 代码。lex 将“ %{ ”和“ % } ”之间的内容直接拷贝到生成的 C 文件，所以在这可以编写任何有效的 C 代码。

在这个示例中，定义段中惟一的内容是一些 C 注释。你也许想知道是否能包括没有定界符的注释。在“ % { ”和“ % } ”外部，lex 中的注释必须用空白缩进来正确标识它们。忘记缩进注释，lex 会将它们解释为别的东西，这就会出现出人意料的错误。

%%标记这一部分结束。

下一部分是规则段（即规则部分）。每个规则都由两部分组成：模式和动作，由空白分开。当 lex 生成的词法分析程序识别出某个模式时，将执行相应的动作。这些模式是 UNIX 样式的正则表达式，即由工具（例如 grep，sed 和 ed）使用的相同表达式的扩展版本。第六章描述了正规表达式的所有规则。示例中的第一条规则如下：

```
[\t ]+       /* 忽略空白 */ ;
```

方括号“ [] ”指示括号中的任何一个字符都与模式匹配。在我们的示例中，我们接受“ \t"（一个制表符）或“ ”（一个空格）。“ + ”意味着模式匹配加号前面的一个或多个连续的子模式的拷贝。因此，这个模式描述了空白（制表符和空格的任意组合）。规则的第二部分（动作）只是一个分号，即一个什么也不做的 C 语句，它的作用是忽略输入。

接下来的规则使用“ | ”（垂直竖线）动作。这是一个特殊的动作，意味着下一个模式应用相同的动作，因此所有的动词都使用为最后一个动词指定的动作[1]。

我们的第一套模式是：

```
is |
am |
are |
```

[1] 也可以在模式中使用垂直竖线，例如 **foo|bar** 是匹配字符串“ foo ”或“ bar ”的模式，在模式和垂直

```
were |
was |
be |
being |
been |
do |
does |
did |
will |
would |
should |
can |
could |
has |
have |
had |
go    { printf("%s: is a verb\n", yytext); }
```

这些模式匹配列表中的任意动词。一旦识别出一个动词，就执行这个动作，即 C printf 语句。yytext 数组包含匹配模式的文本。这个动作将打印识别出的动词，后跟字符串“ :is a verb\n ”。

最后两条规则是：

```
[a-zA-Z]+    { printf("%s: is not a verb\n", yytext); }

.|\n     { ECHO; /* 通常的默认状态 */ }
```

模式“ [a-zA-Z]+ ”是一种通用的模式：它表示至少包含一个字符的任意字母字符串。“ - ”字符在方括号之间使用时有一个特殊的含义：它指示从“ - ”的左边开始到“ - ”的右边结束的字符范围。当看到这些模式之一时，我们的动作是打印匹配的标记和字符串“ :is not a verb\n ”。

不用花很长时间就可了解到，匹配前面规则中列出的任意动词的任何单词也匹配这条规则。那么你也许想知道，当看到列表中的一个动同时为什么不执行两个动作。并且“ island ”以“ is ”开头，当它看到单词“ island ”时执行两个动作吗？答案是 lex 拥有一套简单的消除歧义的规则。使词法分析程序工作的两条规则是：

1. lex 模式只匹配输入字符或字符串一次。

2. lex 执行当前输入的最长可能匹配的动作。因为“ island ”是比“ is ”长的匹配，所以 lex 把“ island ”看做匹配上面那条“ 包括一切 ”的规则。

如果考虑 lex 词法分析程序如何匹配模式，就应该能够看出我们的示例只匹配列

出的动词。

最后一行是默认情况语句。特殊的字符“ . ”（英文的句号）匹配换行符以外任意的单个字符，“ \n ”匹配一个换行字符。特殊动作 ECHO 输出匹配的模式，复制标点符号或其他字符。尽管它是一种默认行为，我们还是特别列出了这种情况。我们看过一些由于这种特征而不能正确工作的复杂的词法分析程序，这种特征在默认模式匹配出乎意料的输入字符时产生奇怪的输出。（虽然对于无法匹配的输入字符都有一个默认的动作，但是好的词法分析程序都拥有清楚的规则来匹配所有可能的输入。）

规则段的结尾以另一个%%来界定。

最后的部分是用户子例程段，由任意合法的 C 代码组成。在 lex 生成代码结束之后，lex 将它复制到 C 文件。我们已经包含了一个 **main()**例程。

```
%%
main()
{
   yylex();
}
```

由 lex 产生的词法分析程序是一个称为 yylex()的 C 例程，我们可以调用它[1]。如果动作中不包含明确的此 **return** 语句，那么 **yylex()**直到处理了完整的输人之后才会返回。

将最初的示例保存在文件 *ch1-02.l* 中（因为它是第二个例子）。为厂在 UNIX 系统上创建一个可执行的程序，输入下面的命令：

```
% lex ch1-02.l
% cc lex.yy.c –o first –ll
```

lex 将 lex 规范译成 C 源文件，称为 lex.yy .c，我们对它进行编译并连接到 lex 库*-ll* 。然后，执行得到的程序来检查它是否像本节前面所看到的那样工作。试一试可以使你自己信服，这个简单的描述确实精确地识别出了我们想要识别的那些动词。

现在，我们已经理解了第一个示例。第二个示例（见例 l-2）将词法分析程序扩展为识别不同的词性。

例 1-2：具有多重词性的 lex 示例 ch1-03.l

```
%{
/*
 * 扩展第一个示例以识别其他的词性
```

[1]实际上，我们可以不包含主程序，因为 lex 库包含像这样的默认主程序。

```
*/

%}
%%

[\t ]+      /* 忽略空白 */ ;

is |
am |
are |
were |
was |
be |
being |
been |
do |
does |
did |
will |
would |
should |
can |
could |
has |
have |
had |
go    { printf("%s: is a verb\n", yytext); }

very |
simply |
gently |
quietly |
calmly |
angrily     { printf("%s: is an adverb\n", yytext); }

to |
from |
behind |
above |
below |
between { printf("%s: is a preposition\n", yytext); }

if |
then |
```

```
and |
but |
or    { printf("%s: is a conjunction\n", yytext); }

their |
my |
your |
his |
her |
its      { printf("%s: is an adjective\n", yytext); }

I |
you |
he |
she |
we |
they     { printf("%s: in a pronoun\n", yytext); }

[a-zA-Z]+ {
   printf("%s:  don't recognize, might be a noun\n", yytext);
  }

.|\n     { ECHO; /* 通常的默认状态 */ }

%%

main()
{
  yylex();
}
```

符号表

第二个示例实际上没有多少不同，只是列出了比前面更多的单词，原则上可以扩展这个示例为任意多的单词。虽然它是很方便的，但是如果在词法分析程序运行时能够构建一个单词表，那么就可以在添加新的单词时不用修改和重新编译 lex 程序。在下一个示例中我们就这样做，即在词法分析程序运行时从输人文件中读取声明的单词时允许动态地声明词性。声明行以词性的名字开始，后面跟着要声明的单词。例如，下面声明了 4 个名词和 3 个动词：

```
noun dog cat horse cow
verb chew eat lick
```

这个单词表是一个简单的符号表，这是 lex 和 yacc 应用程序中的公用结构。例如，C 编译程序存储符号表中的变量和结构名称、标签、枚举标记和所有在程序中使用的其他名字。每个名字都连同描述名字的信息一起存储。在一个 C 编译程序中，信息是指符号的类型、声明的作用域、变量类型等。在当前的示例中，信息是词性。

添加符号表可以完全地改变词法分析程序。不必在词法分析程序中为每个要匹配的单词放置独立的模式，只要有一个匹配任意单词的模式，再查阅符号表就能决定所找到的词性。因为词性的名字（名词、动词等）引入了一个声明行，所以它们现在是“ 保留字 ”。对于每个保留字仍然有一个独立的 lex 模式。还必须添加符号表维护例程，在这种情况下，**add_word ()**表示在符号表中放人一个新单词。**lookup_word()**表示查寻已经输入的单词。

在程序代码中，声明一个变量 **state**，用来记录是在查找单词（状态 LOOKUP）还是在声明它们（在这种情况，**state** 能记住我们正在声明的单词种类）。无论何时只要我们看到以词性名字开始的行，就可以将状态设置为声明单词的种类；每次看到\n 时都切换回正常的查找状态。

例 1-3 展示了定义段。

例 1-3：带符号表的词法分析程序（3 部分中的第 1 部分）ch1-04.l

```
%{
/*
 *带符号表的单词识别程序
 */

enum {
   LOOKUP = 0, /* 默认 – 查找而不是定义 */
   VERB,
   ADJ,
   ADV,
   NOUN,
   PREP,
   PRON,
   CONJ
};

int state;

int add_word(int type, char *word);
int lookup_word(char *word);
%}
```

为了在表中记录单独的单词类型并声明一个变量 **state**，定义一个 *enum*。在状态

变量（用来跟踪定义的内容）和符号表（用来记录每个定义的单词属于何种类型）中使用这种枚举类型。另外还声明了符号表例程。

例 1-4 展示了规则段。

例 1-4：带符号表的词法分析程序（3 部分中的第 2 部分）ch1-04.l

```
%%
\n { state = LOOKUP; } /* 行结束，返回到默认状态 */

   /* 无论何时，行都以保留的词性名字开始 */
   /* 开始定义该类型的单词 */
^verb { state = VERB; }
^adj  { state = ADJ; }
^adv  { state = ADV; }
^noun { state = NOUN; }
^prep { state = PREP; }
^pron { state = PRON; }
^conj { state = CONJ; }

[a-zA-Z]+  {
      /* 一个标准的单词，定义它或查找它 */
       if(state != LOOKUP) {
          /* 定义当前的单词 */
          add_word(state, yytext);
       } else {
          switch(lookup_word(yytext)) {
          case VERB: printf("%s: verb\n", yytext); break;
          case ADJ: printf("%s: adjective\n", yytext); break;
          case ADV: printf("%s: adverb\n", yytext); break;
          case NOUN: printf("%s: noun\n", yytext); break;
          case PREP: printf("%s: preposition\n", yytext); break;
          case PRON: printf("%s: pronoun\n", yytext); break;
          case CONJ: printf("%s: conjunction\n", yytext); break;
          default:
              printf("%s:  don't recognize\n", yytext);
              break;
          }
      }
}

.  /* 忽略其它的东西 */ ;

%%
```

为了声明单词，第一组规则将状态设置为对应于被声明的词性的类型。（模式开始处的“ ^ ”使模式只在输入行的开始处匹配。）在每行的开始处重新将状态设置为 **LOOKUP**，这样在我们添加新的单词后，可以交互地测试单词表，以确定它是否正确地工作。当模式“ [a-zA-Z] + ”匹配时，如果状态是 **LOOKUP**，使用 **lookup_word()**查找单词，找到就打印出它的类型。如果是其他任意状态时，用 **add_word()**定义单词。

例 1-5 中的用户子例程段包含同样的框架 **main()**例程和两个支持函数。

例 1-5：带符号表的 lexer（3 个部分中的第 3 部分）ch1-04.l

```
main()
{
   yylex();
}

/* 定义一个单词和类型的链表 */
struct word {
   char *word_name;
   int word_type;
   struct word *next;
};

struct word *word_list; /* 单词链表中的第一个元素 */

extern void *malloc();

int
add_word(int type, char *word)
{
   struct word *wp;

   if(lookup_word(word) != LOOKUP) {
      printf("!!! warning: word %s already defined \n", word);
      return 0;
   }

   /* 单词不在那里，分配一个新的条目并将它连接到列表上 */

   wp = (struct word *) malloc(sizeof(struct word));

   wp->next = word_list;

   /* 还必须复制单词本身 */
```

```
   wp->word_name = (char *) malloc(strlen(word)+1);
   strcpy(wp->word_name, word);
   wp->word_type = type;
   word_list = wp;
   return 1; /* 添加成功 */
}

int
lookup_word(char *word)
{
   struct word *wp = word_list;

   /* 向下搜索列表以寻找单词 */
   for(; wp; wp = wp->next) {
      if(strcmp(wp->word_name, word) == 0)
         return wp->word_type;
   }

   return LOOKUP;   /* 没有找到 */
}
```

最后两个函数创建并搜索单词的链表。如果有许多单词，则函数运行得很慢，因为寻找每个单词都必须搜索整个列表。在产品环境中，我们会采用一种较快的但很复杂的方案——大概采用一个散列表。虽然很慢，但我们的简单示例还是采用这种方法。

下面是最后一个例子的会话示例：

```
verb is am are was were be being been do
is
is: verb
noun dog cat horse cow
verb chew eat lick
verb run stand sleep
dog run
dog: noun
run: verb
chew eat sleep cow horse
chew: verb
eat: verb
sleep: verb
cow: noun
horse: noun
verb talk
```

```
talk
talk: verb
```

强烈建议读者研究这个示例，深入理解示例直到满意为止。

语法

对于某些应用，我们所完成的简单的词类识别也许足够用了；而另一些应用需要识别特殊的标记序列并执行适当的动作。传统上，对这样的一套动作的描述称为语法。它似乎特别适用于我们的例子。假设我们希望识别普通的句子，下面是简单句型的列表：

noun verb
noun verb noun

在这一点上，引进一些描述语法的符号似乎是很方便的。使用右向箭头“ → ”意味着可以用一个新的符号取代一套特殊的标记[1]。例如：

subject→noun|pronoun

指示一个新的符号 *subject* 是名词（noun）或代词（pronoun）。我们没有改变底层(undollyiog ）符号的含义；相反我们从已经定义的更加基础的符号中构建了新的符号。正如下面的例子那样，我们可以把一个对象定义如下：

object→noun

当不像英语语祛那样严格要求时，可以按以下方式定义句子：

sentence→subject verb object

实际上可以扩展句子的定义以适合更广的句型。然而，现在还是先构建一个 yacc 语法，这样能交互式地彻底检查我们的想法。为了将有用的值返回给新的分析程序，在介绍 yacc 语法之前，必须修改词法分析程序。

词法分析程序与语法分析程序的通信

当一起使用 lex 扫描程序和 yacc 语法分析程序时，语法分析程序（parser）是较高级别的例程。当它需要来自输入的标记时，就调用词法分析程序 **yylex()**。然后，词法分析程序从头到尾扫描输入识别标记。它一找到对语法分析程序有意义的标记就返回到语法分析程序，将返回标记的代码作为 **yylex()**的值。

[1]这里说的是符号而不是标记，因为我们用“ 标记 ”这个词特指从词法分析程序返回的符号，箭头左边的符号不是从词法分析程序中来的。所有的标记都是符号，但不是所有的符号都是标记。

不是所有的标记都对语法分析程序有意义。例如，在多数程序设计语言中，语法分析程序不能接收注释和空白。对于忽略的标记，词法分析程序不返回，以便它继续扫描下一个标记而不“打扰”语法分析程序。

词法分析程序和语法分析程序必须对标记代码的内容达成一致。通过让 yacc 定义标记代码来解决这个问题。在我们的语法中，标记是词性：NOUN、PRONOUN、VERB、ADVERB、ADJECTIVE、PREPOSITION 和 CONJUNCTION。yacc 使用预处理程序*#define* 将它们每一个都定义为小的整数。下面是这个示例中使用的定义：

```
# define NOUN 257
# define PRONOUN 258
# define VERB 259
# define ADVERB 260
# define ADJECTIVE 261
# define PREPOSITION 262
# define CONJUNCTION 263
```

输入的逻辑结束总是返回标记代码零。yacc 不为它定义符号，但是如果想定义的话你可以定义它。

yacc 可以生成包含所有标记定义的 C 头文件。可以把这个文件（在 UNIX 系统上称为 y.tab.h，在 MS -DOS 上称为 *ytab.h* 或 *yytab.h*）包含在词法分析程序中，并且在词法分析程序动作代码中采用这些预处理程序符号。

词法分析程序中的词性

例 1-6 展示了新的词法分析程序的声明和规则段。

例 1-6：从语法分析程序中调用词法分析程序 ch1-05.l

```
%{
/*
 * 我们现在构建一个由高级语法分析程序使用的词法分析程序
 */

#include "ch1-05y.h"    /*来自语法分析程序的标记代码 */

#define LOOKUP 0 /*默认情祝 – 不是一个定义的单词类型 */

int state;

%}

%%
```

```
\n  { state = LOOKUP; }

\.\n   {  state = LOOKUP;
       return 0; /* 句子结尾 */
   }

^verb  { state = VERB; }
^adj   { state = ADJECTIVE; }
^adv   { state = ADVERB; }
^noun  { state = NOUN; }
^prep  { state = PREPOSITION; }
^pron  { state = PRONOUN; }
^conj  { state = CONJUNCTION; }

[a-zA-Z]+ {
       if(state != LOOKUP) {
          add_word(state, yytext);
       } else {
       switch(lookup_word(yytext)) {
       case VERB:
         return(VERB);
       case ADJECTIVE:
         return(ADJECTIVE);
       case ADVERB:
         return(ADVERB);
       case NOUN:
         return(NOUN);
       case PREPOSITION:
         return(PREPOSITION);
       case PRONOUN:
         return(PRONOUN);
       case CONJUNCTION:
         return(CONJUNCTION);
       default:
         printf("%s:  don't recognize\n", yytext);
         /* 不返回，忽略*/
       }
       }
}
.   ;

%%
... same add_word() and lookup_word() as before ...
```

这里介绍几个重要的区别。将词法分析程序中使用的词性名字改变为与语法分析程序中的标记名字相一致。还要添加 **return** 语句将所识别的单词的标记代码传递给语法分析程序。词法分析程序中定义新单词的标记没有任何 **return** 语句，因为语法分析程序不“关心”它们。

这些返回语句表明 **yylex()**操作类似于协同程序。每次语法分析程序调用它时，都在它停止的那一点进行处理。这样就允许我们渐进地检查和操作输入流。我们的第一个程序不需要利用这一点，但是当将词法分析程序作为庞大的程序的一部分时会很有用。

增加一条规则来标记句子的结尾：

```
\.\n   {  state = LOOKUP;
          return 0; /* 句子结尾 */
       }
```

句号前面的反斜杠引用（quote）这个句号，所以这条规则与后跟一个换行的句号匹配。对词法分析程序所做的另一个改变是省略目前语法分析程序中提供的**main()**例程。

yacc 语法分析程序

最后，例 1-7 介绍了 yacc 语法中的第一步。

例 1 — 7 ：简单的 yacc 句子语法分析程序 ch1-05.y

```
%{
/*
 *用于识别英文句子基本语法的词法分析程序
 */
#include <stdio.h>
%}

%token NOUN PRONOUN VERB ADVERB ADJECTIVE PREPOSITION CONJUNCTION

%%
sentence: subject VERB object { printf("Sentence is valid.\n"); }
    ;

subject:  NOUN
    |   PRONOUN
    ;

object:       NOUN
    ;
```

```
%%

extern FILE *yyin;

main()
{
   while(!feof(yyin)) {
      yyparse();
   }
}

yyerror(s)
char *s;
{
   fprintf(stderr, "%s\n", s);
}
```

yacc 语法分析程序的结构类似于 lex 词法分析程序的结构（不是偶然的）。第一部分（定义段）有一个文字代码块，以“ %{ ”和“ %} ”括起。在这里，我们将它用于 C 注释（如同 lex 一样，C 注释属于 C 代码块，至少在定义段中）和单个包含文件。

然后是我们期望从词法分析程序中接收的所有标记的定义。在这个示例中，它们对应 8 个词性。尽管很好地选择标记名字可以告诉读者它们代表的内容，但是标记的名字对于 yacc 没有本质意义，虽然 yacc 允许为 yacc 符号使用任意有效的 C 标识符名，但是通常我们规定标记名都用大写字母，而语法分析程序中的其他名字大部分或完全是小写字母。

第一个%%指示规则段的开始，第几个%%指示规则的结束和用户子例程段的开始。最重要的子程序是 **main()**，这个子程序重复调用 **yyparse()**直到词法分析程序的输入文件结束。例程 **yyparse()**是由 yacc 生成的语法分析程序，所以我们的主程序雨复尝试分析句子直到输入结束。（当词法分析程序看到行的结尾处的句号时返回零标记 ;’ 已是语法分析程序的信号，指示当前分析的输入已完成。）

规则段

规则段将实际的语法描述为一套产生式规则或简称为规则（也有些人称它们为产生式）。每条规则都由“ : ”操作符左侧的一个名宇、右侧的符号列表和动作代码以及指示规则结尾的分号组成。默认情况下，第一条规则是最高级别的规则，也就是说，语法分析程序试图找到一个标记序列匹配这条初始规则，或者更普通地说，从初始规则中找到的规则。规则右侧的表达式是零个或多个名字的列表。像在 object 规则中被定义为 NOUN 一样，典型的简单规则的右侧有一个符号。然

后，规则左侧的符号在其他规则中能像标记一样使用。接下来我们构建复杂的语法。

我们在语法中使用特殊字符“ |”，它引入和前一条规则相同的左侧规则。它通常读作“ 或 ”，举例来说，在我们的语法中，Subject 可以是 NOUN 或 PRONOUN。规则的动作部分由 C 块组成，以“ { ”开始并以“ } ”结束。只要规则匹配，语法分析程序就在规则结尾处执行一个动作。在我们的 sentence 规则中，动作报告我们已经成功地分析了一个句子。因为 sentence。是最高层的符号，所以整个输入必需匹配 **sentence**。当词法分析程序报告输入结束时，分析程序返回到它的调用程序，在这种情况下是主程序。随后对 **yyparse()**的调用重置状态并再次开始处理。如果看到输入标记的“ subject VERB object ”列表，那么我们的示例打印一条消息。如果看到“ subject subject ”或一些其他的无效的标记列表会发生什么呢？语法分析程序调用 **yyerror()**（它在用户的子程序段提供），然后识别特殊的规则 error。可以提供错误恢复代码，尝试将分析程序返回到能继续分析的状态。如果错误恢复失败，就像这里的情况那样——没有错误恢复代码 yyparse()在发现错误后返回到调用程序。

第 3 段和最后一段（即用户子例程段）在第二个%%后开始。这一段包含任意 C 代码并且被逐字地复制到最终的语法分析程序。在示例中，我们为 yacc 生成的语法分析程序提供了一组函数：**main()**和 **yyerror()**，这组函数是使用 lex 生成的词法分析程序进行编译时所必需的。主例程继续调用语法分析程序，直到到达 yyin (lex 输入文件）上的文件尾。惟一其他必需的例程是 yylex()，它由词法分析程序提供。

在本章的最后一个示例（见例 1-8）中，扩展前面的语法来识别一组复杂的（尽管是不完全的）句子。我们请你进一步试用这个例子——你将会看到如何用明确的方式描述复杂的英语。

例 1-8：扩展的英语语法分析程序 ch1-06.y

```
%{
#include <stdio.h>
/* we found the following required for some yacc implementations. */
/* #define YYSTYPE int */
%}

%token NOUN PRONOUN VERB ADVERB ADJECTIVE PREPOSITION CONJUNCTION

%%

sentence: simple_sentence  { printf("Parsed a simple sentence.\n"); }
    | compound_sentence { printf("Parsed a compound sentence.\n"); }
    ;

simple_sentence: subject verb object
    |   subject verb object prep_phrase
```

```
    ;

compound_sentence: simple_sentence CONJUNCTION simple_sentence
    |  compound_sentence CONJUNCTION simple_sentence
    ;

subject:  NOUN
    |  PRONOUN
    |  ADJECTIVE subject
    ;

verb:     VERB
    |  ADVERB VERB
    |  verb VERB
    ;

object:      NOUN
    |  ADJECTIVE object
    ;

prep_phrase:  PREPOSITION NOUN
    ;

%%

extern FILE *yyin;

main()
{
   while(!feof(yyin)) {
      yyparse();
   }
}

yyerror(s)
char *s;
{
   fprintf(stderr, "%s\n", s);
}
```

通过引入小学英语课中传统的语法公式，我们扩展了 **sentence** 规则：一个句子可以是简单句，或者是包含两个或多个独立子句的由并列连接词连接的复合句。目前的词法分析程序不能区分并列连接词（例如“ and ”、“ but ”、“ or" ）和从属连接词（例如“ if " ）。

我们还将递归（recursion）概念引入了语法。递归方式（一条规则直接或间接引用它本身）是一个描述语法的强有力的工具，并且可以在我们编写的几乎每个 yacc 语法中采用该技术。在这里，**compound_sentence** 和 **verb** 规则引入了递归。前一个规则只是声明。compound_sentence 是两个或多个由连接词连接的简单句。第一个可能的匹配是：

```
simple_sentence CONJUNCTION simple_sentence
```

它定义了“ 两个子句 ”的情况。而

```
compound_sentence CONJUNCTION simple_sentence
```

则定义了“ 两个以上子句 ”的情况。在稍后的几章中将更加详细地讨论递归。

虽然这里的英语语法不是特别有用，但是用 lex 标识单词然后用 yacc 找到单词之间的关系的技术与后面几章中特定应用所采用的技术几乎相同。例如，在 C 语言语句中：

```
if(a == b) break; else func(&a);
```

编译程序采用 lex 标识标记“ **if** ”、“ **(** ”、“ **a** ”、“ == ”等，然后使用 yacc 建立 **if** 语句的表达式部分“ a ==b ”，**break** 语句是“ 真 ”分支，而函数调用它的“ 假 ”分支。

运行 lex 和 yacc

本章的最后部分描述如何在系统上构建这些工具。

我们调用不同的词法分析程序 *ch1-N.l*，其中的 N 对应特别的 lex 规范示例。同样，调用语法分析程序 *ch1-M.y*，M 是示例的编号。那么，为了构建输出，在 UNIX 上使用下面的命令：

```
% lex ch1-n.l
% yacc –d ch1-m.y
% cc –c lex.yy.c y.tab.c
% cc –o example-m.n lex.yy.o y.tab.o -ll
```

第一行基于 lex 规范运行 lex，并产生包含词法分析程序的 C 代码的文件 *lex.yy.c*。第二行用 yacc 生成 y.tab.h 和 y.tab.h（后者是由-d 开关创建的标记定义的文件）。下一行对这两个 C 文件进行编译。最后一行将它们连接在一起，并使用 lex 库中的程序 *libl.a*——在多数 UNIX 系统中通常位于*/usr/lib/libl.a* 目录下。如果没有使用 AT&T lex 和 yacc，而采用其他实现，你只需简单地替换命令名而几乎不必做其他改变（特别是，Berkele *yacc* 和 *flex* 仅需通过将 *lex* 和 *yacc* 命令改变为 *byacc*

和 *flex* 就可以工作，并且删除*-ll* 连接程序标志）。然而，我们知道读者之间有许多不同，而且这是事实。例如，如果我们使用 GNU bison 而不是 yacc，它将生成两个文件，称为 *chl-M.tab* 和 *chl -M.tab.h*。在命名有很多限制的系统上，例如 MS-DOS，这些名字将会改变（通常为 ytab.c 和 yab.h ）。参见附录一到附录八详细了解不同的 lex 和 yacc 实现。

lex 和手写的词法分析程序

有人经常告诉我们用 C 语言编写词法分析程序太容易了，没有必要学习 lex。也许是，也许不是。例 1-9 展示了适用于简单命令语言的（用来处理命令、数字、字符串和换行，忽略注释和空白）用 C 语言编写的词法分析程序。例 1-10 是用 lex 编写的等效的词法分析程序。长度上 lex 版本是 C 词法分析程序的三分之一。我们的经验是程序中的错误数一般与它的长 度成正比，我们预期词法分析程序的 C 版本要花三倍的时间来编写和排除错误。

例 1-9：用 C 语言编写的词法分析程序

```
#include <stdio.h>
#include <ctype.h>

#define NUMBER 400
#define COMMENT 401
#define TEXT 402
#define COMMAND 403

int main(int argc, char **argv)
{
   int val;
   while(val = lexer())
      printf("value is %d\n", val);
}

lexer()
{
   int c;

   while((c = getchar()) == ' ' || c == '\t')
      ;
   if(c == EOF)
      return 0;
   if(c == '.' || isdigit(c)) /* 数字 */
   {
      while((c = getchar()) != EOF && isdigit(c))
         ;
      ungetc(c, stdin);
```

```
        return NUMBER;
    }
    if(c == '#') /* 注释 */
    {
        int index = 1;
        while((c = getchar()) != EOF && c != '\n')
            ;
        ungetc(c, stdin);
        return COMMENT;
    }
    if(c == '"') /* 字符串 */
    {
        int index = 1;
        while((c = getchar()) != EOF && c != '"' && c != '\n')
            ;
        if(c == '\n')
            ungetc(c, stdin);
        return TEXT;
    }
    if(isalpha(c))   /* 命令 */
    {
        int index = 1;
        while((c = getchar()) != EOF && isalnum(c))
            ;
        ungetc(c, stdin);
        return COMMAND;
    }
    return c;
}
```

例 1-10：用 lex 编写的同样的词法分析程序

```
%{
#define NUMBER 400
#define COMMENT 401
#define TEXT 402
#define COMMAND 403
%}
%%
[ \t]+    ;
[0-9]+    |
[0-9]+\.[0-9]+   |
\.[0-9]+     { return NUMBER; }
#.*       { return COMMENT; }
\"[^\"\n]\"      { return TEXT; }
[a-zA-Z][a-zA-Z0-9]+   { return COMMAND; }
```

```
\n        { return '\n'; }
%%
#include <stdio.h>

int main(int argc, char **argv)
{
   int val;
   while(val = yylex())
      printf("value is %d\n", val);
}
```

lex 采用自然的方式来处理一些微妙的情况，对这些情况手工编写词法分析程序是很困难的。例如，假设正在跳过一个 C 语言注释，为了找到注释的结尾，需要寻找“ * ”，然后检查下字符是否为“ / ”。如果是，就完成了；如果不是，就要继续扫描。在 C 词法分析程序中非常常见的一个错误是不考虑下面这种情况，即下一个字符是它本身（一个星号），并且后跟一个斜杠。实际上，这意味着一些注释失败：

```
/** 注释 **/
```

（在示例中，我们已经看到这个错误，在一个和 yacc 的某个版本一起发布的手写的词法分析程序中!）

一旦你轻松地使用 lex，我们预言你一定会发现，正如我们所说的那样，用 lex 编写是如此容易，以至你永远不会去编写另一个手写的词法分析程序。

下一章我们将更深入地研究 lex 的使用。第三章我们研究 yacc。的使用。然后考虑几个描述 lex 和 yacc 的较复杂问题和特征的大型示例。

练习

1. 扩展英语语言语法分析程序来处理比较复杂的语法：主语中的介词短语、修饰形容词的副词等等。

2. 使语法分析程序更好地处理复合动词，例如“ has seen ”。你也许想为助动词添加新的单词和标记类型 AUXVERB。

3. 一些单词有多个词性，例如“ watch ”、“ fly ”、“ time ”或“ bear ”。如何处理它们呢？尝试添加新的单词和标记类型 NOUN_OR_VERB，并且将它作为 **subject**、**verb** 和 **object** 规则的可供选择的办法。它能很好地工作吗？

4. 当人们听到一个不熟悉的单词时，他们通常从上下文中猜测它的词性。词法分析程序能在运行中特征化新的单词吗？例如，以“ ing ”结尾的单词可能是

一个动词，一个在“ a ”或“ the ”后的单词可能是名词或形容词。

5. lex 和 yacc 是用于构建实际的英语语言语法分析程序的好工具吗？为什么不是？