第三章 使用 yacc[1]

本章内容：

- 语法
- 移进/归约分析
- yacc 语法分析程序
- 词法分析程序
- 算术表达式和歧义性
- 变量和有类型的标记
- 符号表
- 函数和保留字
- 用 make 构建语法分析程序
- 小结
- 练习

前一章集中考虑了单独使用 lex 的情况。尽管使用 lex 可以产生词法分析程序，但本章还是要学习使用 yacc。凡是在 lex 识别正则表达式的地方，yacc 都可以识别完整的语法。lex 将输入流分成块（标记），然后 yacc 取得这些块并将它们逻辑性地归组到一起。

本章我们创建个台式计算器。先从简单的算术开始．然后添加内置函数、用户变量．最后是用户定义的函数。

语法

yacc 采用你指定的语法并编写识别语法中有效“ 句子 ”的分析程序，这里，我们采用很一般的术语“ 句子 ”，在 C 语言语法中，句子是语法上有效的 C 程序[2]。

正如我们在第一章看到的那样，语法是语法分析程序用来识别有效输入符号的一系列规则。例如，下面是本章后面用于构建计算器的语法形式。

statement→NAME = expression
expression→NUMBER+NUMBER|NUMBER-NUMBER

竖线“ | ”意味着同一个符号有两种可能性，例如，表达式可以是加法也可以是减法。箭头左侧的符号被认定为规则的左边，通常被缩写成 LHS(left-hand side)，右侧的符号是规则的右边，通常缩写成 RHS(right-hand side)。有些规则可以有相

[1] Jobn R.Levine, Tony Mason, Doug Brown 著，杨作梅，张旭东等译). 整理: qy8087@gmail.com
[2]程序可以在语法上有效但在语义上无效。例如，将字符串赋值给 int 变量的 C 程序 yacc 只处理语法，其他的确认取决于用户。

同的左边，竖线是对应于这种情况的一个速记符号（short hand)。实际上，出现在输入中的和被词法分析程序返回的符号是终结符或标记.而规则的左侧出现的是非终结的符号（或非终结符)。终结符和非终结符必须是不同的，标记出现在规则左侧是错误的。

通常使用树状结构来表示所分析的句子。例如，如果用语法分析输入“ fred=12+13 ”, 树状结构如图 3-l 所示。“ 12+13 ”是一个表达式，“ fred=*expressio* ”是一条语句。实际上 yacc 语法分析程序不把这种树状结构作为数据结构，尽管这件事对你来说并不难做。

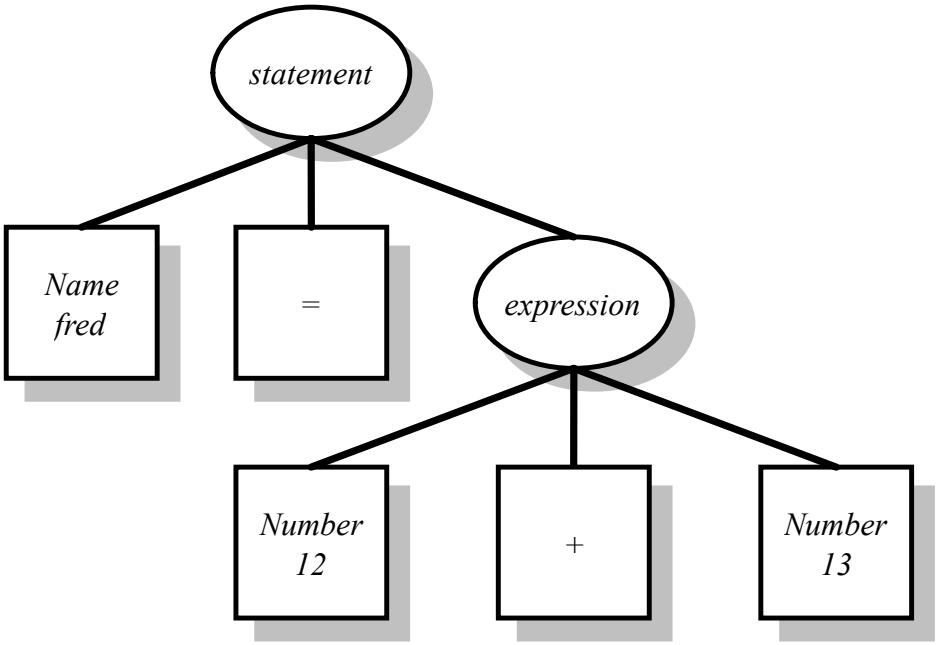

图 3-1 语法分析树

每个语法都包括起始符号，这些起始符号必须位于语法分析树的根部。在这个语法中，slatement 是起始符号。

递归规则

规则可以直接或间接地引用本身，这一特性使分析任意长的输入序列成为可能。让我们来扩展语法以处理更长的算术表达式：

expression→NUMBER
| *expression + NUMBER*
| *expression + NUMBER*

现在，通过重复应用表达式规则来分析一个类似“ fred＝14+23-11+7 ”的表达式，如图 3-2 所示。yacc 对分析递归规则非常有效，所以几乎可以在使用的每个语法

中看到递归规则。

移进/归约分析

yacc 语法分析程序通过寻找可以匹配目前为止所看到的标记的规则来工作。yacc 处理语法分析程序时创建了一组状态，每个状态都反映一个或多个部分被分析的规则中的一个可能的位置。当语法分析程序读取标记时，每次它读取一个没完成规则的标记，就把它压入内部堆栈中，并且切换到一种反映它刚刚读取的标记的新状态。这个动作称为移进（shift）。当它发现组成某条规则右侧的全部符号时，它就把右侧符号弹出堆栈，而将左侧符号压入堆栈中，并胯下切换到反映堆栈上新符号的新状态。这个动作称为归约（reduction），因为它通常减少堆栈上项目的数目。（但并不总是这样、因为它可能拥有右侧为空的规则。）无论 yacc 什么时候归约规则，它都执行与这条规则有关的用户代码。实际上，这就是语法分析程序如何进行分析的过程。

在图 3-1 中可以看出如何使用简单的规则分析输入“ fred=12+13 ”。语法分析程序从将标记一次一个地移进到内部堆栈开始：

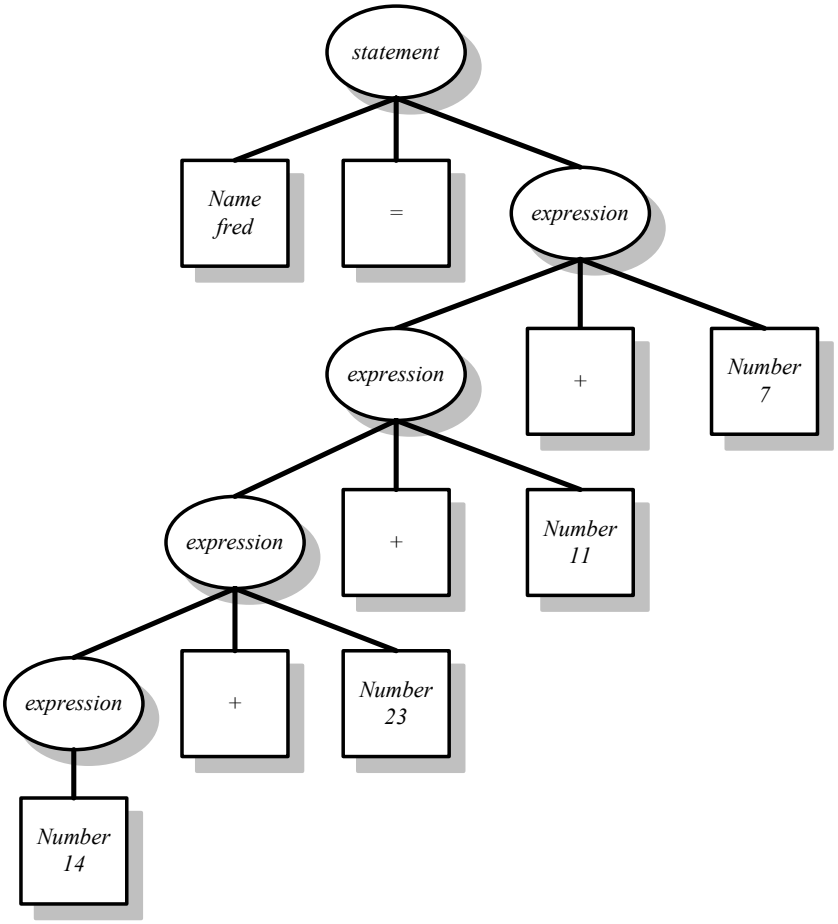

图 3-2 使用递归规则的语法分析

```
fred
fred =
fred = 12
fred = 12
fred = 12+
fred = 12 + 13
```

这里，它可以归约规则“ expression→NUMBER+NUMBER ”，所以它从堆栈中弹出 12、加号和 13，并用下面的表达式代替它们：

```
fred = expression
```

现在，它归约了规则“ statement→NAME=expression ”，所以它弹出 fred、=和 *expression*，并用语句（statement）代替它们。当到达输入尾部时，堆栈就归约到了开始符号，所以根据这一语法这个输入是有效的。

yacc 不能分析什么

虽然 yacc 的语法分析技术是很全面的，但也有 yacc 不能处理的语法。它不能处理歧义语法，在歧义语法中，同样的输入符合多个分析树[1]。它也不能处理需要向前看多于一个的标记才能确定它是否已经匹配一条规则的语法。考虑下面这个极端人为的示例：

phrase→cart_animal AND CART
　　| *work_animal AND PLOW*
cart_animal→HORSE | *GOAT*
work_anima→HORSE | *OX*

这个语法不是歧义的，因为任何有效的输入只有一个可能的语法分析树，但是 yacc 不能处理它，因为它要求向前查看两个符号。特别是，在输入“ HORSE AND CART ”中，它不能决定 HORSE 是 **cart_animal** 还是 **work_animal**，直到它看见 CART，而 yacc 不能看得那么远。

如果将第一条规则改为如下形式：

phrase→cart_animal CART
　　| *work_animal PLOW*

yacc 就不会遇到麻烦、因为它会提前看下一个标记来决定 HORSE 的输入是后跟 CART 还是后跟 PLOW。如果后跟 CART，HORSE 为 **cart_animal**，如果后跟 PLOW 则为 **work_animal**。

[1]实际上，可以在后面看到，yacc 可以处理有限的但是却是有用的歧义语法。

实际上，这些规则不像这里看到的那样复杂和容易混淆。一个原因是 yacc 明确地知道能分析什么语法和不能分析什么语法。如果给出一个它不能处理的语法，它就会告诉你，因此不会出现过分复杂的分析程序悄悄失败的问题。另一个原因是 yacc 能处理的语法可以很好地响应人们实际上编写的程序。往往使 yacc 混淆的语法结构也会使人混淆，所以如果有一些语言设计方面的灵活度，就应该考虑将语言改变为使 yacc 和它的用户更加容易理解。

关于移进和归约分析的更多信息请见第八章。想要了解 yacc 如何才能把规范转换成可工作的 C 程序，请参见由 Aho、Sethi 和 Ullman 编著的经典著作《Compilers:Principles,Technique,and Tools》，Addison-Wesley 出版社 1986 年出版，由于它的封面插图所以又称为 “ 龙书 ”。

yacc 语法分析程序

yacc 语法具有和 lex 规范一样的三部分结构（lex 结构模仿了 yacc 的结构）。第一部分（定义段）处理 yacc 生成的语法分析程序（从现在起称它为语法分析程序）的控制信息，而且通常建立语法分析程序运行的执行环境。第二部分包含语法分析程序的规则，第三部分是被逐字拷贝到生成的 C 程序中的 C 代码。首先用最简单的语法编写语法分析程序，如图 3-l 所示，然后将它扩展为更加有用和现实的程序。

定义段

定义段包含语法中使用的标记的描述、分析程序堆栈中使用的值的类型和其他的东西。它还包括文字块、由%{…%}封闭的 C 代码。下面通过声明两个符号标记来开始第一个语法分析程序。

```
%token NAME NUMBER
```

可以将单个被引起来的字符作为标记而不用声明它们，所以不需要声明“ = ”、“ + ”或 “ - ”。

规则段

语法部分只是由一列语法规则组成，大部分语法规则和我们前面使用的规则具有相同的格式。因为 ASCII 键盘没有 “ → ” 键，所以在规则的左侧和右侧之间使用冒号，而且在每个规则的尾端都有一个分号：

```
%token NAME NUMBER
%%
statement:    NAME ‘=’ expression
```

```
       |      expression
       ;
expression:  NUMBER '+' NUMBER
       |      NUMBER '-' NUMBER
       ;
```

和 lex 不同，yacc 不关心规则部分的行边界，你会发现大量的空白使语法更容易阅读。分析程序中增加了一条新规则：语句可以是纯表达式，也可以是一个赋值。如果用户输入一个纯表达式，就可以打印出它的结果。

第一条规则左侧的符号通常是起始符号，但可以在定义部分使用**%start** 声明覆盖它。

符号值和动作

yacc 语法分析程序中的每个符号都有一个值，该值给出了符号的特定实例的额外信息。如果符号代表一个数字，那么值就是某个数字。如果它表示个文字性的文本串，这个值可能就是一个指向该文本串拷贝的指针。如果它表示程序中的一个变量，这个值就是指向描述这个变量的符号表项的指针。有些标记没有有用的值，例如，表示闭括号的标记，因为一个闭括号和另一个闭括号是相同的。

非终结符号可以有你想要的任何值，这些值由分析程序中的代码创建。通常动作代码构建与输入相对应的语法分析树，以便后面的代码能一次处理整个语句，甚至整个程序。

在当前的语法分析程序中，NUMBER 的值或表达式是这个数字或表达式的数值，而 NAME 的值是符号表的指针。

在真正的语法分析程序中，不同符号的值使用不同的数据类型，例如，数字符号 *int* 和 *double*、字符串符号 *char* *和较高级符号的结构指针。如果有多个值类型，就必须有语法分析程序中使用的所有值类型的列表，这样 yacc 可以创建包含它们的称为 *YYSTYPE* 的 C 联合类型定义。（幸运的是，yacc 提供了许多方法以帮助你确保每个符号都使用了正确的值类型。）

在计算器的第一个版本中，惟一感兴趣的值是输入数字和计算出来的表达式的数值。默认情况下,yacc 使所有值类型为 *int*，这对计算器的第一种版本已经足够用了。

无论语法分析程序何时归约一个规则，它都执行与规则有关的用户 C 代码，通常称为规则的动作。这个动作出现在规则之后且在分号或竖线之前的大括号中。动作代码可以将右侧符号的值引用为**$1**、**$2**…，而且通过设置**$$**可以设置左侧的值。在语法分析程序中，*expression* 符号的值是它所代表的表达式的值。我们添加一些代码来计算和打印表达式，引出图 3-2 中所使用的语法。

```
%token NAME NUMBER
%%
statement:    NAME '=' expression
          |   expression       { printf("= %d\n", $1); }
          ;

expression:   expression '+' NUMBER   { $$ = $1 + $3; }
          |   expression '-' NUMBER   { $$ = $1 - $3; }
          |   NUMBER                  { $$ = $1; }
          ;
```

构建表达式的规则计算出适当的值，而且将表达式识别为一条语句的规则打印出结果。在构建表达式的规则中，第一个和第二个数字的值分别是**$1** 和**$3**。操作符的值为**$2**，尽管在这个语法中，操作符没有感兴趣的值。因为在每个归约后 yacc 都执行默认动作，在运行任何明确的动作代码之前，将值**$1** 赋介**$$**，所以最后一条规则上的动作不是绝对必要的.

词法分析程序

要试验语法分析程序，需要一个词法分析程序供给标记。正如第一章所提及的那样，语法分析程序是较高级的例程，而且无论何时只要它需要输入中的标记就调用词法分析程序 **yylex()**。词法分析程序一发现分析程序感兴趣的标记，它就返回给分析程序，将标记代码作为一个值返回。yacc 将语法分析程序中的标记名定义为 C 预处理程序名字，放在 y.tab.h 中（或 MS-DOS 系统上的一些类似名字），所以词法分析程序可以使用它们。

下面是为语法分析程序提供标记的简单的词法分析程序：

```
%{
#include "y.tab.h"
extern int yylval;
%}

%%
[0-9]+    { yylval = atoi(yytext); return NUMBER; }
[ \t]+    ;             /* 忽略空白 */
\n        return 0;     /* 逻辑上的 EOF */
.         return yytext[0];
%%
```

数字串是数字，忽略空白，而且换行返回输入结束的标记（数字零），告诉语法分析程序没有更多的东西需要读取。词法分析程序中的最后一条规则是非常普通的截流器（catch-all），也就是将所有其他未处理的字符作为单个字符标记返回给

语法分析程序。字符标记通常是标点符号，例如圆括号、分号和单字符操作符。如果语法分析程序接收到不认识的标记，就产生语法错误，所以这条规则可以使你很容易地处理所有的单字符标记，而让 yacc 的错误检测机制捕获和“抗议”无效的输入。

无论词法分析程序何时将标记返回给语法分析程序，如果标记有相关的值，词法分析程序在返回之前都必须在 **yylval** 中存储值。在这个示例中，明确地声明了 yylval。在更复杂的语法分析程序中，yacc 将 **yylval** 定义为一个联合（union),并将定义放置在 *y.tab.h* 中。

编译并运行简单的语法分析程序

在 UNIX 系统上，yacc 利用语法创建了 *y.tab.c*。（C 语言分析程序）和包含标记号定义的头文件 *y.tab.h*。lex 创建 *lex.yy.c*（C 语言词法分析程序)。你只需要用 yacc 和 lex 库将它们编译在一起。这些库包含所有支持例程的可用的默认版本，包括用来调用语法分析程序 **yyparse()**并退出的 **main()**。

```
% yacc -d ch3-01.y  # 生成 y.tab.c 和 y.tab.h
% lex ch3-01.l      # 生成 lex.yy.c
% cc -o ch3-01 y.tab.c lex.yy.c -ly -ll     # 编译和链接 C 文件
% ch3-01
99+12
= 111
% ch3-01
2 + 3 -14+33
= 24
% ch3-01
100 + -50
syntax error
```

第一个版本似乎在工作。当输入某些不符合语法的东西时，第三段测试正确地报告语法错误。

算术表达式和歧义性

可以使算术表达式变得更加一般和实际，扩展处理乘法和除法、一元否定和带括号的表达式的。*expression* 规则：

```
expression:  expression '+' expression { $$ = $1 + $3; }
          |  expression '-' expression { $$ = $1 - $3; }
          |  expression '*' expression { $$ = $1 * $3; }
          |  expression '/' expression
```

```
            {  if($3 == 0)
                  yyerror("divide by zero");
               else
                  $$ = $1 / $3;
            }
        |   '-' expression      { $$ = -$2; }
        |   '(' expression ')'  { $$ = $2; }
        |   NUMBER              { $$ = $1; }
        ;
```

除法动作检测被零除的情况，因为在多数 C 程序的实现中，除以零会使程序崩溃。调用 **yyerror()**（标准的 yacc。错误处理例程）报告该错误。

但是这个语法存在一个问题，它极端含糊。例如，输入 2+3*4 意味着(2+3)*4 或 2+(3*4)，输入 3-4-5-6 意味着 3-(4-(5-6))或(3-4)-(5-6)或者其他的可能。图 3-3 展示了 2+3*4 的两种可能的分析。

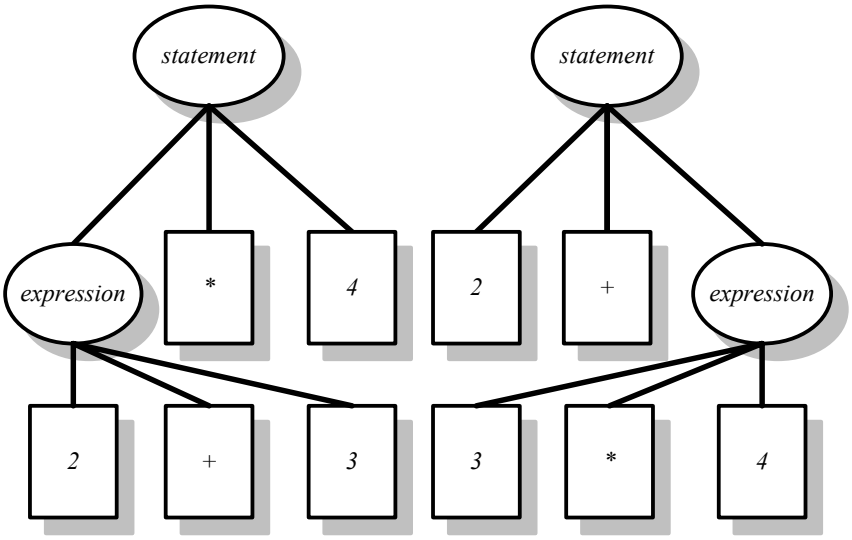

图 3-3　输入 2+3*4 时的歧义性

如果编译上面那样的语法，yacc 将告知存在 16 个移进/归约冲突，表明它无法决定到底是先从堆栈上移进标记，还是先归约规则。

例如，当分析“ 2+3*4 ”时，语法分析程序要经过这些步骤（这里将表达式简写为 *E*）：

```
2                       shift NUMBER
E                       reduce E→NUMBER
E+                      shift+
E+3                     shift NUMBER
E+E                     reduce E→NUMBER
```

这时，语法分析程序查看“ * ”，并且可以用下面的规则将“ 2+3 ”归约为一个表达式：

```
   expression:            expression ‘+’ expression
```

或者移进“ * ”，期望稍后能归约：

```
   expressione:           expresion ‘*’ expression
```

问题是我们不能告诉 yacc 有关优先级和操作的组合规则。优先级控制表达式中哪个操作符要先执行。数学和编程惯例（回溯至 1956 年的第一个 Fortan 编译程序）规定乘法和除法优先于加法和减法，所以“ a+b*c ”意思是 a+(b*c）而且 d/e-f 意思是(d/e)-f。在任何表达式语法中，操作符都被从低到高地归组到优先级中。级别的数目取决于语言。C 语言因为有太多的优先级而出名，一共 15 个级别。

结合规则（associativity）控制同一优先级上的操作符组合顺序。操作符可以向左侧归组，例如在 C 语言中 a-b-c 意思是(a-b)-c；或者向右侧归组，例如 a=b=c 意思是 a=(b=c)。在一些情况下，操作符根本不归组，例如，在 Fortran 中 A.LE.B.LE.C 是无效的。

有两种方式可以指定语法中的优先级和结合规则：隐式地和显式地。要隐式地指定它们，使用单独的非终结符号为每个优先级重新编写语法。假设使用通常的优先级和左结合，这样重新编写表达式规则如下：

```
expression:  expression ‘+’ mulexp
       |     expression ‘-’ mulexp
       |     mulexp
       ;
mulexp:      mulexp ‘*’ primary
       |     mulexp ‘/’ primary
       |     primary
       ;
primary:     ‘(’ expression ‘)’
       |     ‘-’ primary
       |     NUMBER
       ;
```

这是编写一个语法的非常合理的方式，并且如果 yacc 没有明确的优先级规则，那么它就是惟一的方式。

但是 yacc 也可以让你显式地指定优先级。可以在定义部分添加这些行，结果成为例 3-1 中的语法。

```
% left ‘+’ ‘-’
```

```
% left ‘*’ ‘/’
% nonassoc UMINUS
```

这些描述定义了优先级，它们告诉 yacc"+ ”和“ - ”是左结合而且处于最低的优先级，“ * ”和“ / ”是左结合而且处于较高的优先级，UMINUS（是代表一元减号的伪标记）没有结合规则而且处于最高的优先级。（这里没有任何右结合操作符。如果有右结合，它们将使用%right。）yacc 指定规则右侧最右边标记的优先级作为该规则的优先级；如果这条规则不包含具有优先级的标记，那么这条规则就没有自己的优先级。当 yacc 遇到歧义语法引起的移进/归约冲突时，它就参考优先级表，并且如果所有冲突的规则都包含一个出现在优先级声明中的标记时，它使用优先级来解决冲突。

在我们的语法中，所有的冲突发生在 *expression OPERATOR expression* 形式的规则中，所以为四个操作符设置优先级将解决所有的冲突。使用优先级的这个语法分析程序比外加隐式优先级规则的语法分析程序小而且快，因为需要归约的规则较少。

例 3-l：具有表达式和优先级的计算器语法 ch3-02.y

```
%token NAME NUMBER
%left '-' '+'
%left '*' '/'
%nonassoc UMINUS

%%

statement:   NAME '=' expression
   |  expression       { printf("= %d\n", $1); }
   ;

expression:  expression '+' expression { $$ = $1 + $3; }
   |  expression '-' expression { $$ = $1 - $3; }
   |  expression '*' expression { $$ = $1 * $3; }
   |  expression '/' expression
            {  if($3 == 0)
                  yyerror("divide by zero");
               else
                  $$ = $1 / $3;
            }
   |  '-' expression %prec UMINUS   { $$ = -$2; }
   |  '(' expression ')'  { $$ = $2; }
   |  NUMBER       { $$ = $1; }
   ;
%%
```

负号规则包括“ %prec UMINUS ”。这条规则包括的惟一操作符是“ - ”，这个操作符具有很低的优先级，但是我们想让一元减号具有比乘法更高的优先级而不是较低的优先级。%prec 告诉 yacc 为这条规则使用 UMINUS 的优先级。

何时不使用优先规则

可以使用优先规则调整发生在语法中的任何移进/归约冲突。这通常是一个可怕的想法。在表达式语法中，冲突的原因容易理解而且优先规则的作用很明了。在其他情况下，优先规则也可以解决移进/归约问题，但是通常很难正确地理解它们在语法中所具有的效果。

我们建议只在两种情况下使用优先级：在表达式语法中，以及为了解决 if-then-else 语言结构语法中的“ 虚挂的 else ”冲突（参见第七章后面的例子）。或者，如果可以的话，应该调整语法来删除冲突。记住冲突意味着 yacc 不能正确地分析语法，可能因为它是歧义的，也就是说同一个输入有多个可能的分析。除了以上两种情况以外，在语言设计中，冲突就是指一个错误。如果语法对 yacc 来说是歧义的，那么它当然对人也是歧义的。参见第八章得到有关寻找和纠正冲突的更多信息。

变量和有类型的标记

下一步扩展计算器来处理具有单个字母名字的变量。因为只有 26 个字母（目前只关心小写字母），所以我们能在 26 个条目的数组（称它为 **vbltable**）中存储变量。为了使计算器更加有用，也可以扩展它来处理多个表达式（每行一个）和使用浮点值，如例 3-2 和例 3-3 所示。

例 3-2：具有变量和实值的计算器语法 ch3-03.y

```
%{
double vbltable[26];
%}

%union {
    double dval;
    int vblno;
}

%token <vblno> NAME
%token <dval> NUMBER
%left '-' '+'
%left '*' '/'
%nonassoc UMINUS
```

```
%type <dval> expression
%%
statement_list:  statement '\n'
       |         statement_list statement '\n'
       ;

statement:       NAME '=' expression { vbltable[$1] = $3; }
       |         expression       { printf("= %g\n", $1); }
       ;

expression:   expression '+' expression { $$ = $1 + $3; }
       |      expression '-' expression { $$ = $1 - $3; }
       |      expression '*' expression { $$ = $1 * $3; }
       |      expression '/' expression
              {   if($3 == 0.0)
                     yyerror("divide by zero");
                  else
                     $$ = $1 / $3;
              }
       |      '-' expression %prec UMINUS   { $$ = -$2; }
       |      '(' expression ')'  { $$ = $2; }
       |      NUMBER
       |      NAME          { $$ = vbltable[$1]; }
       ;
%%
```

例 3-3：适用于具有变量和实值的计算器的词法分析程序 ch3-03.l

```
%{
#include "y.tab.h"
#include <math.h>
extern double vbltable[26];
%}

%%
([0-9]+|([0-9]*\.[0-9]+)([eE][-+]?[0-9]+)?) {
   yylval.dval = atof(yytext); return NUMBER;
   }

[ \t]     ;       /* ignore white space */

[a-z]     { yylval.vblno = yytext[0] - 'a'; return NAME; }

"$"       { return 0; /* end of input */ }

\n  |
```

```
.         return yytext[0];
%%
```

符号值和%union

现在，我们已经有了多种符号值类型。表达式具有 *double*（双精度型）值，变量引用和 **NAME** 符号的值是对应于 **vbltable** 中的从 0 到 25 之间的整数。为了使分析程序更加简单，为什么不使词法分析程序将变量的值作为 *double* 型返回呢？问题是有两种环境可以出现变量名：一种情况是作为表达式的一部分，在这种情况下，我们要用 *double* 值；另一种情况是在等号的左边，在这种情况下需要记住它是哪个变量，这样就能更新 **vbltable**。

为了定义可能的符号类型，在定义部分，添加一个%**union** 声明：

```
%union{
   double dval;
   int vblno;
}
```

声明的内容被逐字拷贝到输出文件，成为 C 的 typedef 语句中定义 **YYSTYPE** 类型的 **union** 声明。产生的头文件 *y.tab.h* 包括定义的拷贝，以便可以在词法分析程序中使用它。下面是从这个语法中生成的 *y.tab.h*:

```
#define NAME 257
#define NUMBER 258
#define UMINUS 259
typedef union {
       double dval;
       int vblno;
}YYSTYPE;
extern YYSTYPE yylval;
```

生成的文件还声明了变量 **yylval**，并且在语法中为符号标记定义了标记号。

现在，我们必须告诉分析程序符号使用的数值类型。通过在符号的定义部分的角括号中放置来自 union 数据类型的适当的字段名，可以完成这个任务：

```
%token <vblno> NAME
%token <dval> NUMBER

%type <dval> expression
```

新的声明%**type** 可以为非终结符号设置类型（否则不需要声明）。也可以在%**left**、

%**right** 或%**nonassoc** 中设置括号类型。在动作代码中，yacc 自动根据适当的字段名确定符号的引用值，例如，如果第三个符号是 **NUMBER**，引用**$3** 的动作类似**$3.dval**。

例 3-2 展示了新的、被扩展的分析程序。增加一个新的起始符号，**statement_list**，以便分析程序能接受一系列语句，每条语句的结尾都有一个换行符号，而不是只有一条语句。还为设置变量的规则增加了一个动作，在尾部还增加了一条将 **NAME** 变换为 **expression** 的新规则，它获取变量的值。

必须修改一下词法分析程序（见例 3-3）。词法分析程序中的文字块不再声明 yylval，因为它的声明目前位于 *y.tab.h* 中。词法分析程序不能自动将类型和标记联系在一起，所以当设置 yylval 时必须在明确的字段引用中插入声明。从第二章以来，我们使用实数模式匹配浮点值。动作代码使用 **atof()**读取数字，然后将值分配给 **yylval.dval**，因为分析程序期待 **dval** 字段中的数字的值。对于变量，在 **yylval.dval** 中返回变量表中变量的下标。最后，使“ \n ”成为了一个普通标记，因此使用美元符号指示输入的结束。

```
% ch3-03
2/3
-0.666667
a = 2/7
a
= 0.285714
z = a + 1
z
= 1.28571
a/z
= 0.222222
$
```

符号表

很少有用户能满意于单个字符变量名，所以现在要增加使用较长变量名的能力．这意味着需要一个符号表——用于跟踪使用中的名字的结构。每次词法分析程序读取输入中的名字时，它都在符号表中查找这个名字，并且得到一个对应符号表条目的指针。在程序的其他地方，使用符号表指针而不是名字串，因为每次需要时指针比查找名字更容易更快速。

因为符号表要求词法分析程序和语法分析程序共享的数据结构，所以我们创建一个头文件。*ch3hdr.h*（见例 3-4）。这个符号表是一个结构数组，每个结构都包含变量的名字和它的数值。我们还声明了一个例程 **symlook()**,它以文本字符串形式的名字为参数，并且返回适当的符号表条目的指针，如果它不存在，就添加它。

例 3-4：具有符号表的语法分析程序的头文件 ch3hdr.h

```
/*
 *  Header for calculator program
 */

#define NSYMS 20 /* maximum number of symbols */

struct symtab {
    char *name;
    double value;
} symtab[NSYMS];

struct symtab *symlook();
```

分析程序只稍做改变以使用符号表，如例 3-5 所示。**NAME** 标记的值是指向符号表的指针而不是如前所述的下标。我们改变%**union** 并且将指针字段称做 **symp**。**NAME** 的%**token** 声明适当地有所改变，而且给变量赋值和读取变量的动作现在将标记值作为指针来使用，所以它们能读取或编写符号表条目的值字段。

新的程序 **symlook()**在 yacc 规范的用户子例程部分定义，如例 3-6 所示。（这里没有引人注目的理由，它可以很容易地存在于 lex 文件中或一个单独的文件中。）它顺序地搜索字符表来寻找与作为参数来传入的名字对应的条目。如果某条目有一个 **name** 字符串并且它匹配 **symlook()**正在搜索的字符串，它就返回指向该条目的指针，因为名字已经被放进了表中。如果 **name** 字段为空，并且已经寻找了使用中的所有表条目且没有找到这个符号，那么我们就把名字输入到至今为止还是空的表条目中。

使用 **strdup()**生成一个名字字符串的永久拷贝。当词法分析程序调用 **symlook()** 时，它传递标记缓冲区 **yytext** 中的名字。因为每个后来的标记都会重写 **yytext**, 所以这里需要生成一个拷贝（这是 lex 扫描程序中常见的错误根源，如果需要在扫描程序继续到下一个标记后使用 yytext 的内容，那么总要生成一个拷贝）。最后，如果当前的表条目正在使用但是没有匹配，那么 **symlook()**就继续搜索下一个条目。

这个符号表程序对于这个简单的例子完全足够用了，但越实用的符号表代码会越复杂。连续搜索对于大尺寸的符号表来说太慢了，所以要使用散列法或一些其他较快的搜索函数。实际的符号表趋向于每个条目装载相当多的信息，例如，变量的类型，无论它是一个简单的变量、数组还是结构，以及如果它是一个数组它有多少维数。

例 3-5：具有符号表的语法分析程序的规则 ch3-04,y

```
%{
#include "ch3hdr.h"
#include <string.h>
```

```
%}

%union {
    double dval;
    struct symtab *symp;
}
%token <symp> NAME
%token <dval> NUMBER
%left '-' '+'
%left '*' '/'
%nonassoc UMINUS

%type <dval> expression
%%
statement_list:  statement '\n'
       |         statement_list statement '\n'
       ;

statement:    NAME '=' expression { $1->value = $3; }
       |      expression       { printf("= %g\n", $1); }
       ;

expression:   expression '+' expression { $$ = $1 + $3; }
       |      expression '-' expression { $$ = $1 - $3; }
       |      expression '*' expression { $$ = $1 * $3; }
       |      expression '/' expression
              {   if($3 == 0.0)
                     yyerror("divide by zero");
                  else
                     $$ = $1 / $3;
              }
       |      '-' expression %prec UMINUS   { $$ = -$2; }
       |      '(' expression ')'  { $$ = $2; }
       |      NUMBER
       |      NAME          { $$ = $1->value; }
       ;
%%
```

例 3-6：符号表程序 ch3-04.pgm

```
/* look up a symbol table entry, add if not present */
struct symtab *
symlook(s)
char *s;
{
    char *p;
```

```
    struct symtab *sp;

    for(sp = symtab; sp < &symtab[NSYMS]; sp++) {
        /* is it already here? */
        if(sp->name && !strcmp(sp->name, s))
            return sp;

        /* is it free */
        if(!sp->name) {
            sp->name = strdup(s);
            return sp;
        }
        /* otherwise continue to next */
    }
    yyerror("Too many symbols");
    exit(1);  /* cannot continue */
} /* symlook */
```

词法分析程序也只是稍微改变以适应符号表（见例 3-7）。它不是直接声明符号表，而是包含 *ch3hdr.h*。识别变量名的规则匹配 " [A-Za-z][A-Za-z0-9]* "，任何字母和数字串都以字母开头。它的动作调用 **symlook()**得到指向符号表条目的指针，并且将它存储在 **yylval.symp**（标记的值）中。

例 3-7：具有符号表的词法分析程序 ch3-04,l

```
%{
#include "y.tab.h"
#include "ch3hdr.h"
#include <math.h>
%}

%%
([0-9]+|([0-9]*\.[0-9]+)([eE][-+]?[0-9]+)?) {
       yylval.dval = atof(yytext);
       return NUMBER;
    }

[ \t]  ;       /* ignore white space */

[A-Za-z][A-Za-z0-9]*    {  /* return symbol pointer */
       yylval.symp = symlook(yytext);
       return NAME;
    }

"$"{ return 0; }
```

```
\n  |
.   return yytext[0];
```

符号表程序在一个很小的方面要优于大多数编程语言中 :因为我们动态地分配字符小空间，所以对于变量名的长度没有固定的限制[1]：

```
% ch3-04
foo = 12
foo /5
= 2.4
thisisanextremelylongvarlablenamewhichnobodywouldwanttotype = 42
3 * thisisanextremelylongvarlablenamewhichnobodywouldwanttotype
= 126
$
%
```

函数和保留字

为计算器生成的另一个指令是添加用于平方根、指数和对数的数学函数。以如下方式处理输入：

```
s2 = sqrt(2)
s2
= 1.41421
s2 * s2
= 2
```

强制的办法是使函数名分隔标记，并且为每个函数添加单独的规则：

```
%token SQRT LOG EXP
    . . .
%%
expression: . . .
       |      SQRT '(' expression ')'    { $$ = sqrt($3);}
       |      LOG '(' expression ')'     { $$ = log($3); }
       |      EXP '(' expression ')'     { $$ = exp($3); }
```

在扫描程序中，必须为 " sqrt " 输入返回 **SQRT** 标记，诸如此类：

```
sqrt   return SQRT;
```

[1]实际上，由于 lex 所能处理的最大标记尺寸而有所限制，但是可以使变量名非常长。见第六章 " lex 规范参考 " 中的 " yytext "。

```
log    return LOG
exp    return EXP;

[A-Za-z][A-Za-z0-9]*   {   . . .
```

（特定模式在前，所以它们比一般的符号模式匹配得早。）

这样可以工作，但是存在问题。问题之一是必须将每个函数硬编码到语法分析程序和词法分析程序中，这样会很冗长，而且很难添加更多的函数。另一个问题是函数名是保留字。也就是说，不能将 **sqrt** 用做变量名。这是不是一个问题取决于你的意图。

符号表中的保留字

首先，我们从词法分析程序中拿出函数名采用的特定模式并将它们放入符号表。给每个符号表条目添加新的字段 **funcptr**，如果这个条目是函数名，那么它就是调用 C 函数的指针。

```
struct symtab {
       char *name;
       double (*funcptr)();
       double value;
}symtab[NSYMS];
```

在开始分析程序前必须将函数名放进符号表，所以编写自己的 **main()**，它调用新的例程 **addfunc()**将每个函数名添加到符号表，然后调用 **yyparse()**。**addfunc()** 的代码仅仅得到名字的符号表条目并设置 **funcptr** 字段。

```
main()
{
   extern double sqrt(), exp(), log();

   addfunc(“sqrt”, sqrt);
   addfunc(“exp”, exp);
   addfunc(“log”, log);
   yyparse();
}

addfunc(name, func)
char *name;
double (*func)();
{
   struct symtab *sp = symlook(name);
```

```
    sp->funcptr = func;
}
```

定义 **FUNC** 标记代表函数名。词法分析程序看到函数名时返回 **FUNC**，看到变量名时返回 **NAME**。两者的值都是符号表指针。

在语法分析程序中，用一个通用的函数规则取代每个函数的单独规则：

```
%token <symp> NAME FUNC
%%
expression: . . .
       |      FUNC '(' expression ')' { $$ = ($1->funcptr)($3); }
```

当语法分析程序看到函数引用时，它可以从该函数在符号表中的条目找到真正的内部函数引用。

在词法分析程序中，针对匹配函数名的模式，如果符号表条目表明名字是函数名就改变名字的动作代码来返回 **FUNC**：

```
[A-Za-z][A-Za-z0-9]* {
       struct symtab *sp = symlook(yytext);

       yylval.symp = sp;
       if(sp->funcptr)
          return FUNC;
       else
          return NAME;
}
```

这些改变产生和前一个程序一样工作的程序，但是函数名位于符号表中。例如，这个程序在分析过程中可以输入新的函数名。

可互换的函数和变量名

最后的改变是技术卜的“ 小儿科 ”，但是可以有效地改变语言。函数和变量必须分开没有理由！语法分析程序根据语法可以区分函数调用和变量引用。

所以让词法分析程序返回它原有的方式，即总是为任何一种名字返回 **NAME**。然后改变语法分析程序以在函数处接受 **NAME**:

```
%token <symp> NAME
%%
expression: . . .
```

```
      |      NAME '(' expression ')' { . . . }
```

整个程序包含在例 3-8 至例 3-11 中。正如你在例 3-9 中所着到的那样，必须添加错误检测以确定用户调用函数时，它是一个真正的函数。

现在，除了函数和变量名可以重叠以外，计算器如前所述进行操作。

```
% ch3-05
sqrt(3)
= 1.73205
foo(3)
foo not a function
= 0
sqrt = 5
sqrt(sqrt)
= 2.23607
```

你是否想让用户在同一程序中为两件事情使用相同的名字是有争议的。一方面它使程序变得难以理解，但是另一方面可以不再强制用户只使用不与保留字发生冲突的名字。

两者都会走向极端。COBOL 有 300 多个保留字，没有人能都记住它们，编程人员可能采用了不常用的约定，例如每个变量名以数字开始以确保它们不发生冲突。另一方面，PL/I 根本没有保留字，所以可以写：

```
   IF IF = THEN THEN ELSE = THEN; ELSE ELSE = IF;
```

例 3-8：最终的计算器头文件 ch3hdr2.h

```
/*
 *  Header for calculator program
 */

#define NSYMS 20 /* maximum number of symbols */

struct symtab {
   char *name;
   double (*funcptr)();
   double value;
} symtab[NSYMS];

struct symtab *symlook();
```

例 3-9：最终的计算器语法分析程序的规则 ch3-05.y

```
%{
#include "ch3hdr2.h"
```

```
#include <string.h>
#include <math.h>
%}

%union {
    double dval;
    struct symtab *symp;
}
%token <symp> NAME
%token <dval> NUMBER
%left '-' '+'
%left '*' '/'
%nonassoc UMINUS

%type <dval> expression
%%
statement_list:  statement '\n'
    |   statement_list statement '\n'
    ;

statement:    NAME '=' expression { $1->value = $3; }
    |   expression       { printf("= %g\n", $1); }
    ;

expression:  expression '+' expression { $$ = $1 + $3; }
    |   expression '-' expression { $$ = $1 - $3; }
    |   expression '*' expression { $$ = $1 * $3; }
    |   expression '/' expression
              {   if($3 == 0.0)
                     yyerror("divide by zero");
                  else
                     $$ = $1 / $3;
              }
    |   '-' expression %prec UMINUS   { $$ = -$2; }
    |   '(' expression ')'  { $$ = $2; }
    |   NUMBER
    |   NAME          { $$ = $1->value; }
    |   NAME '(' expression ')'{
           if($1->funcptr)
              $$ = ($1->funcptr)($3);
           else {
              printf("%s not a function\n", $1->name);
              $$ = 0.0;
           }
```

```
        }
    ;
%%
```

例 3-l：最终计算器语法分析程序的用户子例程 ch3-05.y

```
/* look up a symbol table entry, add if not present */
struct symtab *
symlook(s)
char *s;
{
    char *p;
    struct symtab *sp;

    for(sp = symtab; sp < &symtab[NSYMS]; sp++) {
        /* is it already here? */
        if(sp->name && !strcmp(sp->name, s))
            return sp;

        /* is it free */
        if(!sp->name) {
            sp->name = strdup(s);
            return sp;
        }
        /* otherwise continue to next */
    }
    yyerror("Too many symbols");
    exit(1);  /* cannot continue */
} /* symlook */

addfunc(name, func)
char *name;
double (*func)();
{
    struct symtab *sp = symlook(name);
    sp->funcptr = func;
}

main()
{
    extern double sqrt(), exp(), log();

    addfunc("sqrt", sqrt);
    addfunc("exp", exp);
    addfunc("log", log);
    yyparse();
```

```
}
```

例 3-11：最终的计算器词法分析程序 ch3-05.l

```
%{
#include "y.tab.h"
#include "ch3hdr2.h"
#include <math.h>
%}

%%
([0-9]+|([0-9]*\.[0-9]+)([eE][-+]?[0-9]+)?) {
       yylval.dval = atof(yytext);
       return NUMBER;
   }

[ \t]  ;       /* ignore white space */

[A-Za-z][A-Za-z0-9]*   {  /* return symbol pointer */
       struct symtab *sp = symlook(yytext);

       yylval.symp = sp;
       return NAME;
   }

"$"{ return 0; }

\n  |
.   return yytext[0];
%%
```

用 make 构建语法分析程序

我们第三次重新编译这个例子，在重新编译过程中使用一些自动操作是适宜的，即使用 UNIX *make* 程序。控制这个过程的 *Makefile* 如例 3-12 所示。

例 3-12：计算器的 Makefile

```
#LEX = flex -I
#YACC = byacc

CC = cc -DYYDEBUG=1

ch3-05: y.tab.o lex.yy.o
   $(CC) -o ch3-05 y.tab.h lex.yy.o -ly -ll -lm
```

```
lex.yy.o: lex.yy.c y.tab.h

lex.yy.o y.tab.o: ch3hdr2.h

y.tab.c y.tab.h: ch3-05.y
   $(YACC) -d ch3-05.y

lex.yy.c: ch3-05.l
   $(LEX) ch3lex.l
```

在顶端是两个被注释的赋值语句，即用 flex 代替 lex，用 Berkeley yacc 代替 AT&T yacc。flex 需要*-I* 标志以表明它产生一个交互式扫描程序，这个交互式扫描程序不试图向前查看越过一行。**CC** 宏设置预处理程序符号 YYDEBUG，用于插入测试语法分析程序时有用的调试代码。

将所有的事情汇编到 *ch3* 的规则涉及 3 个库：yacc 库*-ly*、lex 库*-ll* 和数学库*-lm*。yacc 库提供 **yyerror()**（在计算器的早期版本中，还有 **main()**）。lex 库提供一些内部的 lex 扫描程序需要的支持例程（由 flex 产生的扫描程序不需要库，但是保留它没有害处）。数学库提供 **sqrt()**、**exp()**和 **log()**。

如果必须使用 bison(yacc 的 GNU 版本)，那么就必须改变产生 y.tab.c 的规则，因为 bison 使用不同的默认文件名。

```
y.tab.c y.tab.h: ch3yac.y
   bison -d ch3yac.y
   mv ch3yac.tab.c y.tab.c
   mv ch3yac.tab.h y.tab.h
```

（或者将 *Makefile* 和代码的其余部分改变为使用 bison 所采用的更容易记忆的名字，另外还可以使用*-y* 告诉 bison 使用通常的 yacc 文件名。）

要得到有关 *make*。的更多信息，参见 Steve Talbott 编写的《Managing Projects with Make》，由 O'Reilly&Associates 出版。

小结

本章已经看到了如何创建 yacc 语法规范，将它与词法分析程序结合起来生成可工作的计算器，并且扩展这些计算器以处理符号变量和函数名。在下面的两章中，将讨沦更庞大的而且更加实际的应用程序，即 SQL 数据库语言的菜单产生程序和处理程序。

练习

1. 给计算器添加更多的函数。例如，使用下面的规则，尝试添加两个参数函数，例如取模和反正切：

```
expression: NAME '(' expression ',' expression ')'
```

2. 可以在符号表中为两个参数的函数放置一个独立的字段，所以可以根据一个参数还是两个参数调用适当的 **atan()**版本。

3. 添加字符串数据类型，以便将字符串分配给变量并且在表达式或函数调用中使用它们。为引用文字字符串添加 **STRING** 标记。将 **expression** 的值变为包含值类型标签以及值的结构．另外，用 **stringexp** 非终结符号为具有字符串（char *）值的字符串表达式扩展语法。

4. 如果添加一个 **stringexp** 非终结符号，当用户键入 42 + "grapefruit"时会发生什么？修改语法以允许混合类型的表达式有多困难？

5. 将字符串值赋给变量时必须做什么？

6. 重载操作符有多困难（例如，如果参数是字符串，使用 " + " 意味着拼接）?

7. 给计算器添加命令以保存和恢复来自和去往磁盘文件的变量。

8. 向计算器添加用户定义的函数。困难的部分是以用户调用函数时能重新执行的方式存储该函数的定义。一种可能性是保存定义函数的标记流。例如：

```
statement: NAME '(' NAME ')' '=' { start_save($1, $3); }
   expression
   { end_save(); define_func($1, $3); }
```

   函数 **start_save()**和 **end_save()**告诉词法分析程序保存表达式的所有标记的列表。在定义表达式中需要标识对哑变量**$3** 的引用。

   当用户调用这个函数时，重放这个标记：

```
expression: USERFUNC '(' expression ')' { start_replay($1, $3); }
   expression /* replays the function */
   { $$ = $6; } /* use its value */
```

   当重放这个函数时，将参数值**$3** 插入重放的表达式代替哑变量。

9. 如果继续向计算器中添加特征，最终以你的独特的程序语言结束。那是个好主意吗？为什么是或者为什么不是？