第40章　分析表达式

虽然标准C++非常全面，仍然有一些东西它没有提供。在本章中，我们将讨论其中的一个：表达式分析器。我们使用表达式分析器来对代数表达式，例如(10 – 8) * 3求值。表达式分析器是相当有用的，适合于很大范围的应用。它们也是编程中用的最多的实体之一。由于各种原因，用于创建表达式分析器的过程并没有被教授。相反，许多程序员都对表达式分析的过程感到困惑。

表达式分析实际上非常直接，在许多方面比编程任务更容易。理由是：这个任务是很好定义的，可依照严格的代数规则进行。本章将开发一个经常被人称为递归下降分析器（recursive-descent parser）的东西，以及所有使你能够对数学表达式求值的必需的支持例程。我们将创建三个版本的分析器，前两个是非通用的。最后一个是通用的，可以适用于任何数字类型。然而，在开发任何分析器以前，必须简要地概述一下表达式和分析过程。

40.1　表达式

因为表达式分析器计算代数表达式的值，所以理解表达式的组成部分是非常重要的。虽然表达式可以由所有类型的信息组成，本章只讨论数字表达式。就我们的目的而言，数字表达式由下面几项组成：

- 数字
- 运算符 +, –, /, *, ^, %, =
- 括号
- 变量

对我们的分析器来说，运算符^表示求幂（在C++中表示异或XOR），=是赋值运算符。可以依据代数法则把这些项组合在表达式中。下面是一些例子。

```
10 – 8
(100 – 5) * 14/6
a + b – c
10^5
a = 10 – b
```

假定每个运算符的优先级如下所示：

最高	+ – (一元)
	^
	* / %
	+ –
最低	=

具有相同优先级的运算符从左到右求值。

在本章的例子中，所有的变量都是用单个字母表示的（换句话说，共有26个变量，即从A到Z可用）。变量不区分大小写（a和A是同一个变量）。对于第一种形式的分析器，所有的数值都被求值为double型，尽管很容易编写程序来处理其他类型的值）。最后，为了使逻辑清楚且易于理解，仅包含了最少量的错误检查。

40.2 分析表达式：问题

如果没有更多地思考表达式分析时出现的问题，你可能认为它是一项简单的工作。然而，要更好地理解这个问题，尝试一下计算下面这个表达式：

```
10 - 2 * 3
```

我们知道这个表达式的值等于4。尽管我们可以很容易地创建一个程序，用于计算具体的表达式，问题是如何创建一个程序，这个程序对任意的表达式会给出正确的答案。最初，你可能会设想像下面这样的一个例程：

```
a = 得到第一个操作数
while(操作数存在) {
op = 得到运算符
b = 得到第二个操作数
a = a op b
}
```

这个例程得到第一个操作数、运算符和第二个操作数来执行第一个操作，然后得到下一个运算符和操作数来执行下一个操作，等等。然而，如果使用这个基本方法，表达式10 - 2 * 3的结果为24（即8 * 3)而不是4，因为这个过程忽略了运算符的优先级。我们不能按从左到右的顺序取操作数和运算符，因为代数法则规定乘法必须在减法之前完成。某些初学者认为这个问题很容易解决，是的，有时，在某些情况下确实能。但是当加入括号、幂、变量、一元运算符等时，这个问题就会变得更糟了。

尽管有几种方法可以用来编写计算表达式的例程，这里开发的最易于人编写，它也是最常见的。这里所用的方法称为递归下降分析器，在本章中你会看到这个名字是怎样来的（其他一些用来编写分析器的方法采用了必须由另一个计算机程序生成的复杂表格。这些有时称为表驱动的分析器）。

40.3 分析一个表达式

可以使用若干方法来分析和计算一个表达式。当使用递归下降分析器时，应把表达式想成递归的数据结构，即，依据其自身定义的表达式。现在，如果我们假定表达式仅能使用+, -, *, /和括号，那么所有的表达式都可用下面的规则定义：

```
表达式 ->项[+ 项] [- 项]
项 ->因数[* 因数] [/ 因数]
因数 ->变量, 数字, 或(表达式)
```

方括号指明一个可选元素，->意味着生成（produces）。事实上，这些原则通常称为表达式的生成原则（production rules）。因此，可以说：依据项的定义，“项生成因数乘因数或因数除因数”。注意，在定义表达式时，运算符的优先级是隐含存在的。

表达式

```
10 + 5 * B
```

有两项：10 和 5 * B。第二项包含两个因数：5 和 B。这些因数由一个数和一个变量组成。

另一方面，表达式

```
14 * (7 - C)
```

有两个因数：14 和(7 - C)。这些因数由一个数和一个带括号的表达式组成。带括号的表达式包含两项：一个数字和一个变量。

这个过程形成了递归下降分析器的基础，递归下降分析器是一组相互递归的函数，像链一样工作且实现了生成原则。在每一步，分析器以代数上正确的序列执行具体的操作。要弄清楚用于分析表达式的生成原则，让我们研究一个使用下列表达式的例子：

```
9/3 - (100 + 56)
```

下面是将执行的顺序：

1. 得到第一项，9/3。
2. 得到每个因数并除整数。本例中结果值是 3。
3. 得到第二项，(100 + 56)。此时开始递归分析第二个子表达式。
4. 得到每一项并相加。其结果值是 156。
5. 从递归调用中返回并从 3 中减去 156。答案是 -153。

如果对此感到有点困惑，不必担心。这是一个比较复杂的概念，需要逐渐适应。关于表达式的这种递归情况，有两件事需要记住。首先，运算符的优先级是隐含在生成原则的定义中的。第二，这个分析和求值表达式的方法类似于人们对数学表达式求值的方法。本章其余部分开发了三个分析器。第一个将分析和求 double 型的浮点表达式的值，其中的浮点表达式仅由常量值组成。第二个增强这个分析器以支持使用变量。最后的第三个分析器作为模板类实现，可被用于分析任何类型的表达式。

40.4　parser 类

表达式分析器是构建在 parser 类之上的。第一种形式的分析器如下所示，其后的分析器都是建立在它之上的。

```
class parser {
  char *exp_ptr;  // points to the expression
  char token[ 80]; // holds current token
  char tok_type;  // holds token's type

  void eval_exp2(double &result);
  void eval_exp3(double &result);
```

```
  void eval_exp4(double &result);
  void eval_exp5(double &result);
  void eval_exp6(double &result);
  void atom(double &result);
  void get_token();
  void serror(int error);
  int isdelim(char c);
public:
  parser();
  double eval_exp(char *exp);
};
```

parser类包含三个私有成员变量。要求值的表达式包含在一个由exp_ptr所指的以null结束的字符串中。因此，这个分析器计算包含在标准ASCII字符串中的表达式。例如，下面的字符串包含该分析器能够求值的表达式：

```
"10-5"
"2 * 3.3 / (3.1416 * 3.3)"
```

当分析器开始执行时，exp_ptr必须指向表达式字符串中的第一个字符。随着分析器的执行，它分析完整个字符串，直到遇到一个null结束符。

其他两个成员变量token和tok_type的意义，将在下一节描述。

分析器的进入点是eval_exp( )，它必须用一个指向要分析的表达式的指针调用。函数eval_exp2( )到eval_exp6( )与atom( )一道形成了递归下降分析器。它们实现了一个前面讨论过的表达式生成原则的增强版。在接下来的分析器中，也将增加称为eval_exp1( )的函数。

serror( )处理表达式中的语法错误。使用函数get_token( )和isdelim( )把表达式分解为各个组分，如下一节所述。

40.5 剖析一个表达式

为了计算表达式的值，你需要能把表达式分成几个部分。因为这个操作是分析的基础，在检验分析器本身之前，让我们来看看它。

表达式的每个组成部分称为一项（token）。例如，表达式

```
A * B - (W + 10)
```

包含项A, *, B, –, (, W, +, 10和)。每一项表示表达式的一个不可分的单元。一般来讲，你需要一个函数，该函数顺序地返回表达式中的每一项。这个函数也必须能够跳过空格和制表符并检测到表达式的结束。我们将用来执行这项工作的函数称为get_token( )，它是parser类的一个成员函数。

除了项本身外，也需要知道正在返回的项的类型。对于本章中开发的每个分析器，仅需要三个类型：VARIABLE, NUMBER和DELIMITER (DELIMITER用于运算符和括号)。

下面显示了get_token( )函数。它从由exp_ptr所指的表达式中获得下一项并把它放到成员变量token中，它也把项的类型放到成员变量tok_type中。

```
// Obtains the next token.
```

```
void parser::get_token()
{
  register char *temp;

  tok_type = 0;
  temp = token;
  *temp = '\0';

  if(!*exp_ptr) return; // at end of expression

  while(isspace(*exp_ptr)) ++exp_ptr; // skip over white space

  if(strchr("+-*/%^=()", *exp_ptr)){
    tok_type = DELIMITER;
    // advance to next char
    *temp++ = *exp_ptr++;
  }
  else if(isalpha(*exp_ptr)) {
    while(!isdelim(*exp_ptr)) *temp++ = *exp_ptr++;
    tok_type = VARIABLE;
  }
  else if(isdigit(*exp_ptr)) {
    while(!isdelim(*exp_ptr)) *temp++ = *exp_ptr++;
    tok_type = NUMBER;
  }

  *temp = '\0';
}

// Return true if c is a delimiter.
int parser::isdelim(char c)
{
  if(strchr(" +-/*%^=()", c) || c==9 || c=='\r' || c==0)
    return 1;
  return 0;
}
```

让我们仔细看看前面的函数。在前几次初始化之后，get_token( )检查是否找到一个null结束了表达式。它是通过检查被exp_ptr所指的字符做到这一点的。因为exp_ptr是一个指向正被分析的表达式的指针，如果它指向一个null，则到达了表达式的结尾。如果仍然有项可从表达式中获得，则get_token( )首先跳过任何前导的空格。一旦跳过空格，exp_ptr指向一个数字、一个变量、一个运算符，或者如果结尾的空格结束了表达式，则指向一个null。如果下一个字符是一个运算符，在项中它将作为一个字符串返回且DELIMITER被放到tok_type中。如果下一个字符是一个字母，就假定它是变量之一。在项中它作为一个字符串返回，且tok_type被赋予了值VARIABLE。如果下一个字符是一个数字，那么读取整个数字并将之按字符串形式放到项中，它的类型为NUMBER。最后，如果下一个字符不是前面所提的任何一种，则假定到达了表达式的末尾。在此例中，项（token）是null，即标志着表达式的结束。

如前所述，为了让这个函数中的代码更简洁，省略了一些错误检查并做了一些假设。例如，任何没被识别的字符可能标志着表达式的结束。还有，在这种形式中，虽然变量可以是任意长度的，但是仅第一个字母是有效的。可以按特定应用程序所指示的那样添加更多的错误检查功

能和其他细节。

要更好地理解标志项的过程，研究一下下面的表达式中每一项返回什么及返回类型：

A + 100 - (B * C) /2

项	项的类型
A	VARIABLE
+	DELIMITER
100	NUMBER
-	DELIMITER
(	DELIMITER
B	VARIABLE
*	DELIMITER
C	VARIABLE
)	DELIMITER
/	DELIMITER
2	NUMBER
null	null

记住，项（token）总是包含一个以null结束的字符串，即使它只包含一个字符，也是如此。

40.6 一个简单的表达式分析器

下面是第一种形式的分析器。它可以计算完全由常量、运算符和括号组成的表达式。它不能接受包含变量的表达式。

```
/* This module contains the recursive descent
   parser that does not use variables.
*/

#include <iostream>
#include <cstdlib>
#include <cctype>
#include <cstring>
using namespace std;

enum types { DELIMITER = 1, VARIABLE, NUMBER};

class parser {
  char *exp_ptr;  // points to the expression
  char token[80]; // holds current token
  char tok_type;  // holds token's type

  void eval_exp2(double &result);
  void eval_exp3(double &result);
  void eval_exp4(double &result);
  void eval_exp5(double &result);
  void eval_exp6(double &result);
  void atom(double &result);
  void get_token();
  void serror(int error);
```

```
  int isdelim(char c);
public:
  parser();
  double eval_exp(char *exp);
};

// Parser constructor.
parser::parser()
{
  exp_ptr = NULL;
}

// Parser entry point.
double parser::eval_exp(char *exp)
{
  double result;

  exp_ptr = exp;

  get_token();
  if(!*token) {
    serror(2); // no expression present
    return 0.0;
  }
  eval_exp2(result);
  if(*token) serror(0); // last token must be null
  return result;
}

// Add or subtract two terms.
void parser::eval_exp2(double &result)
{
  register char op;
  double temp;

  eval_exp3(result);
  while((op = *token) == '+' || op == '-') {
    get_token();
    eval_exp3(temp);
    switch(op) {
      case '-':
        result = result - temp;
        break;
      case '+':
        result = result + temp;
        break;
    }
  }
}

// Multiply or divide two factors.
void parser::eval_exp3(double &result)
{
  register char op;
```

```
  double temp;

  eval_exp4(result);
  while((op = *token) == '*' || op == '/' || op == '%') {
    get_token();
    eval_exp4(temp);
    switch(op) {
      case '*':
        result = result * temp;
        break;
      case '/':
        result = result / temp;
        break;
      case '%':
        result = (int) result % (int) temp;
        break;
    }
  }
}

// Process an exponent.
void parser::eval_exp4(double &result)
{
  double temp, ex;
  register int t;

  eval_exp5(result);
  if(*token== '^') {
    get_token();
    eval_exp4(temp);
    ex = result;
    if(temp==0.0) {
      result = 1.0;
      return;
    }
    for(t=(int)temp-1; t>0; --t) result = result * (double)ex;
  }
}

// Evaluate a unary + or -.
void parser::eval_exp5(double &result)
{
  register char op;

  op = 0;
  if((tok_type == DELIMITER) && *token=='+' || *token == '-') {
    op = *token;
    get_token();
  }
  eval_exp6(result);
  if(op=='-') result = -result;
}

// Process a parenthesized expression.
```

```
void parser::eval_exp6(double &result)
{
  if((*token == '(')) {
    get_token();
    eval_exp2(result);
    if(*token != ')')
      serror(1);
    get_token();
  }
  else atom(result);
}

// Get the value of a number.
void parser::atom(double &result)
{
  switch(tok_type) {
    case NUMBER:
      result = atof(token);
      get_token();
      return;
    default:
      serror(0);
  }
}

// Display a syntax error.
void parser::serror(int error)
{
  static char *e[] = {
      "Syntax Error",
      "Unbalanced Parentheses",
      "No expression Present"
  };
  cout << e[error] << endl;
}

// Obtain the next token.
void parser::get_token()
{
  register char *temp;

  tok_type = 0;
  temp = token;
  *temp = '\0';

  if(!*exp_ptr) return; // at end of expression

  while(isspace(*exp_ptr)) ++exp_ptr; // skip over white space

  if(strchr("+-*/%^=()", *exp_ptr)){
    tok_type = DELIMITER;
    // advance to next char
    *temp++ = *exp_ptr++;
  }
  else if(isalpha(*exp_ptr)) {
```

```
    while(!isdelim(*exp_ptr)) *temp++ = *exp_ptr++;
    tok_type = VARIABLE;
  }
  else if(isdigit(*exp_ptr)) {
    while(!isdelim(*exp_ptr)) *temp++ = *exp_ptr++;
    tok_type = NUMBER;
  }

  *temp = '\0';
}

// Return true if c is a delimiter.
int parser::isdelim(char c)
{
  if(strchr(" +-/*%^=()", c) || c==9 || c=='\r' || c==0)
    return 1;
  return 0;
}
```

如其所示，这个分析器可以处理下面的运算符：+, –, *, /, %。此外，它可以处理整数幂(^)和一元减运算符。这个分析器也能正确地处理括号。表达式的实际计算发生在相互递归函数eval_exp2( )到eval_exp6( ), 以及atom( )函数中，后者返回一个数的值。在每个函数开始处的注释描述了在分析表达式时它起的作用。

接着的main( )函数演示了分析器的使用情况。

```
int main()
{
  char expstr[80];

  cout << "Enter a period to stop.\n";

  parser ob; // instantiate a parser

  for(;;) {
    cout << "Enter expression: ";
    cin.getline(expstr, 79);
    if(*expstr=='.') break;
    cout << "Answer is: " << ob.eval_exp(expstr) << "\n\n";
  };

  return 0;
}
```

下面是一个范例的运行情况。

```
Enter a period to stop.
Enter expression: 10-2*3
Answer is: 4

Enter expression: (10-2)*3
Answer is: 24

Enter expression: 10/3
Answer is: 3.33333

Enter expression: .
```

40.6.1 理解分析器

为了准确地理解分析器怎样计算一个表达式的值，让我们看一下下面的表达式（假定exp_ptr指向表达式的开始处）。

```
10-3*2
```

当调用eval_exp( )这个分析器的进入点时，它得到第一项。如果该项为空，函数打印信息No Expression Present并返回。然而，在这个例子中，该项包含数10。因为第一项不为空，所以调用eval_exp2( )，接着eval_exp2( )调用eval_exp3( )，eval_exp3( )调用eval_exp4( )，而后者接着调用eval_exp5( )。然后eval_exp5( )检查是否该项是一个一元加或减，因为在这个例子中该项不是一元加或减，所以调用eval_exp6( )。此时，eval_exp6( )要么递归调用eval_exp2( )(在有括号的表达式的情况下)，要么调用atom( )来查找一个数字值。因为该项不是一个左括号，所以执行atom( )且result被赋予值10。接着，获得下一项，且函数开始返回到链上。因为该项现在是一个运算符-，函数返回到eval_exp2( )。

接下来发生的事情是非常重要的。因为该项是-运算符，所以把它保存在op中。然后分析器得到下一项3，下降链开始。像前面一样，键入atom( )。值3在结果（result）中返回且读入项*。这导致返回到eval_exp3( )，在那里读入最后一项2。此时，发生第一个算术运算——2和3相乘。结果返回到eval_exp2( )，并执行减法操作。减法操作的结果是生成答案4。尽管这个过程最初看起来有点复杂，在看过其他一些例子后就会证实这个方法每次都很正确。

如前面的程序所示，这个分析器适于简单的桌面计算器使用。然而，在把它用在计算机语言、数据库或更复杂的计算器之前，它需要具有处理变量的能力。这是下一节中的主题。

40.7 向分析器中添加变量

所有的编程语言、许多计算器和电子表格都使用变量来存储值，供以后使用。在把分析器用于这样的应用程序之前，需要扩展它来包含变量。要完成这一点，需要给分析器中添加几样东西。当然，首先是变量本身。如前所述，我们将使用字母A到Z表示变量。变量将存储在parser类内的一个数组中。每个变量都使用一个含26个元素的double型数组中的一个数组位置。因此，给parser类添加下面的语句：

```
double vars[ NUMVARS] ; // holds variables' values
```

你也需要改变parser构造函数，如下所示。

```
// parser constructor
parser::parser()
{
  int i;

  exp_ptr = NULL;
  for(i=0; i<NUMVARS; i++) vars[ i] = 0.0;
}
```

可以看到，变量被初始化为0。

你也将需要一种查找所给变量值的函数。因为变量被从A到Z命名，通过从变量名中减去A的ASCII值，可以很容易地使用它们来给数组vars加下标。下面所示的成员函数find_var( )

就是完成这项工作的：

```
// Return the value of a variable.
double parser::find_var(char *s)
{
  if(!isalpha(*s)){
    serror(1);
    return 0.0;
  }
  return vars[toupper(*token)-'A'];
}
```

虽然这个函数实际上接受长变量名，但是只有第一个字母是有效的。可以修改此值以适应自己的需要。

也必须修改 atom( )函数以处理数字和变量。新形式如下所示：

```
// Get the value of a number or a variable.
void parser::atom(double &result)
{
  switch(tok_type) {
    case VARIABLE:
      result = find_var(token);
      get_token();
      return;
    case NUMBER:
      result = atof(token);
      get_token();
      return;
    default:
      serror(0);
  }
}
```

从技术上讲，要让分析器正确地使用变量，添加这些就够了，然而，无法给这些变量赋值。这经常是在分析器外完成的，但是可以把等号当做赋值运算符（在C++中就是这样处理的）并使它成为分析器的一部分。有很多方法可以做到这一点。一种方法是给 parser 类添加称为 eval_exp1( )的另一个函数。现在这个函数将开始递归下降链了。这意味着必须用 eval_exp( )调用它而不是 eval_exp2( )，以便开始分析表达式。eval_exp1( )如下所示：

```
// Process an assignment.
void parser::eval_exp1(double &result)
{
  int slot;
  char ttok_type;
  char temp_token[80];

  if(tok_type==VARIABLE) {
    // save old token
    strcpy(temp_token, token);
    ttok_type = tok_type;

    // compute the index of the variable
```

```
    slot = toupper(*token) - 'A';

    get_token();
    if(*token != '=') {
      putback(); // return current token
      // restore old token - not assignment
      strcpy(token, temp_token);
      tok_type = ttok_type;
    }
    else {
      get_token(); // get next part of exp
      eval_exp2(result);
      vars[slot] = result;
      return;
    }
  }

  eval_exp2(result);
}
```

可以看到，这个函数需要确定是否实际有一个赋值。这是因为变量名总是位于赋值前，但是单单变量名不足以保证后面跟有一个赋值表达式。即，这个分析器将接受A=100作为赋值，但是它也知道A/10不是一个赋值。为了完成这一点，eval_exp1( )从输入流中读入下一项。如果它不是等号，这一项返回到输入流，供以后调用putback( )使用。putback( )函数也必须包含于parser类中，如下所示：

```
// Return a token to the input stream.
void parser::putback()
{
  char *t;

  t = token;
  for(; *t; t++) exp_ptr--;
}
```

在做了所有必需的改变后，分析器如下面这样。

```
/* This module contains the recursive descent
   parser that recognizes variables.
*/

#include <iostream>
#include <cstdlib>
#include <cctype>
#include <cstring>
using namespace std;

enum types { DELIMITER = 1, VARIABLE, NUMBER};

const int NUMVARS = 26;

class parser {
  char *exp_ptr;  // points to the expression
  char token[80]; // holds current token
  char tok_type;  // holds token's type
```

```
  double vars[ NUMVARS]; // holds variables' values

  void eval_exp1(double &result);
  void eval_exp2(double &result);
  void eval_exp3(double &result);
  void eval_exp4(double &result);
  void eval_exp5(double &result);
  void eval_exp6(double &result);
  void atom(double &result);
  void get_token();
  void putback();
  void serror(int error);
  double find_var(char *s);
  int isdelim(char c);
public:
  parser();
  double eval_exp(char *exp);
};

// Parser constructor.
parser::parser()
{
  int i;

  exp_ptr = NULL;

  for(i=0; i<NUMVARS; i++) vars[ i] = 0.0;
}

// Parser entry point.
double parser::eval_exp(char *exp)
{
  double result;

  exp_ptr = exp;

  get_token();
  if(!*token) {
    serror(2); // no expression present
    return 0.0;
  }
  eval_exp1(result);
  if(*token) serror(0); // last token must be null
  return result;
}

// Process an assignment.
void parser::eval_exp1(double &result)
{
  int slot;
  char ttok_type;
  char temp_token[ 80];

  if(tok_type==VARIABLE) {
    // save old token
```

```
    strcpy(temp_token, token);
    ttok_type = tok_type;

    // compute the index of the variable
    slot = toupper(*token) - 'A';

    get_token();
    if(*token != '=') {
      putback(); // return current token
      // restore old token - not assignment
      strcpy(token, temp_token);
      tok_type = ttok_type;
    }
    else {
      get_token(); // get next part of exp
      eval_exp2(result);
      vars[slot] = result;
      return;
    }
  }

  eval_exp2(result);
}

// Add or subtract two terms.
void parser::eval_exp2(double &result)
{
  register char op;
  double temp;

  eval_exp3(result);
  while((op = *token) == '+' || op == '-') {
    get_token();
    eval_exp3(temp);
    switch(op) {
      case '-':
        result = result - temp;
        break;
      case '+':
        result = result + temp;
        break;
    }
  }
}

// Multiply or divide two factors.
void parser::eval_exp3(double &result)
{
  register char op;
  double temp;

  eval_exp4(result);
  while((op = *token) == '*' || op == '/' || op == '%') {
    get_token();
```

```
    eval_exp4(temp);
    switch(op) {
      case '*':
        result = result * temp;
        break;
      case '/':
        result = result / temp;
        break;
      case '%':
        result = (int) result % (int) temp;
        break;
    }
  }
}

// Process an exponent.
void parser::eval_exp4(double &result)
{
  double temp, ex;
  register int t;
  eval_exp5(result);
  if(*token== '^') {
    get_token();
    eval_exp4(temp);
    ex = result;
    if(temp==0.0) {
      result = 1.0;
      return;
    }
    for(t=(int)temp-1; t>0; --t) result = result * (double)ex;
  }
}

// Evaluate a unary + or -.
void parser::eval_exp5(double &result)
{
  register char op;

  op = 0;
  if((tok_type == DELIMITER) && *token=='+' || *token == '-') {
    op = *token;
    get_token();
  }
  eval_exp6(result);
  if(op=='-') result = -result;
}

// Process a parenthesized expression.
void parser::eval_exp6(double &result)
{
  if((*token == '(')) {
    get_token();
    eval_exp2(result);
```

```
    if(*token != ')')
      serror(1);
    get_token();
  }
  else atom(result);
}

// Get the value of a number or a variable.
void parser::atom(double &result)
{
  switch(tok_type) {
    case VARIABLE:
      result = find_var(token);
      get_token();
      return;
    case NUMBER:
      result = atof(token);
      get_token();
      return;
    default:
      serror(0);
  }
}

// Return a token to the input stream.
void parser::putback()
{
  char *t;

  t = token;
  for(; *t; t++) exp_ptr--;
}

// Display a syntax error.
void parser::serror(int error)
{
  static char *e[] = {
      "Syntax Error",
      "Unbalanced Parentheses",
      "No expression Present"
  };
  cout << e[error] << endl;
}

// Obtain the next token.
void parser::get_token()
{
  register char *temp;

  tok_type = 0;
  temp = token;
  *temp = '\0';

  if(!*exp_ptr) return; // at end of expression
```

```
  while(isspace(*exp_ptr)) ++exp_ptr; // skip over white space

  if(strchr("+-*/%^=()", *exp_ptr)){
    tok_type = DELIMITER;
    // advance to next char
    *temp++ = *exp_ptr++;
  }
  else if(isalpha(*exp_ptr)) {
    while(!isdelim(*exp_ptr)) *temp++ = *exp_ptr++;
    tok_type = VARIABLE;
  }
  else if(isdigit(*exp_ptr)) {
    while(!isdelim(*exp_ptr)) *temp++ = *exp_ptr++;
    tok_type = NUMBER;
  }

  *temp = '\0';
}

// Return true if c is a delimiter.
int parser::isdelim(char c)
{
  if(strchr(" +-/*%^=()", c)  | c==9 || c=='\r' || c==0)
    return 1;
  return 0;
}

// Return the value of a variable.
double parser::find_var(char *s)
{
  if(!isalpha(*s)){
    serror(1);
    return 0.0;
  }
  return vars[ toupper(*token)-'A'];
}
```

要试一下这个增强版的分析器，可以使用简单分析器所用的同一main( )函数。利用该增强版分析器，现在可以像下面这样键入表达式

```
A = 10/4
A- B
C = A * (F - 21)
```

40.8 递归下降分析器中的语法检查

在转到分析器的模板形式以前，让我们简短地看一下语法检查问题。在表达式分析中，语法错误是这样一种情况，即输入表达式与分析器所要求的严格的原则不一致。大多数时候，这是由人的错误引起的，通常是键入错误。例如，下面的表达式对于本章的分析器来说是无效的：

```
10 ** 8
(10 - 5) * 9)
/8
```

第一个表达式在一行中包含两个运算符，第二个有一个未匹配的括号，而最后一个在表达式的开始处有一个除号。这些条件都不被分析器所允许。因为语法错误能引起分析器给出错误的结果，应小心它们。

在研究分析器的代码时，你可能注意到了在某些环境下调用的serror( )函数。不像其他分析器，递归下降方法让语法检查变得更容易了，因为大多数情况下，它出现在 atom( ), find_var( ) 或eval_exp6( )中，在那里进行括号检查。语法检查的惟一问题是整个分析器不在检查到语法错误时终止。这可能导致多个错误信息。

实现 serror( )函数最好的方式是让它执行某种重置。例如，所有的C++ 编译器都带有两个函数 setjmp( )和 longjmp( )，这两个函数允许一个程序分支到一个不同的函数。因此，serror( ) 能够执行 longjmp( ) 到分析器外程序中的某个安全点。

取决于分析器的用途，当处理错误时，你可能会发现C++的异常处理机制（通过 try, catch 和 throw 实现）是很有益的。

如果让代码像现在这样，可以出现多条语法错误消息。在某些情况下，这很麻烦，但是在另一方面，由于可能捕捉到多个错误，这可能是好事。然而，一般来讲，在把它应用于商用程序之前，你可能想要增强语句检查功能。

40.9　构建一个通用的分析器

前面的两个分析器对数字表达式进行操作，表达式中的所有值都假定为double型的。虽然这对于使用double值的应用来说很合适，但是对于仅使用整数值的应用来说，就不必要了。还有，通过硬编码所要求的值的类型，分析器应用程序受到了不必要的限制。幸运的是，通过使用类模板，很容易创建一个通用的分析器，这种分析器可以处理所定义的代数型表达式中的各种数据类型。一旦完成了这项工作，就可以把此分析器用于我们所创建的内嵌类型和数字类型。

下面是一个通用的表达式分析器。

```
// A generic parser.

#include <iostream>
#include <cstdlib>
#include <cctype>
#include <cstring>
using namespace std;

enum types { DELIMITER = 1, VARIABLE, NUMBER};

const int NUMVARS = 26;

template <class PType> class parser {
  char *exp_ptr; // points to the expression
  char token[ 80] ; // holds current token
  char tok_type; // holds token's type
  PType vars[ NUMVARS] ; // holds variable's values

  void eval_exp1(PType &result);
  void eval_exp2(PType &result);
  void eval_exp3(PType &result);
  void eval_exp4(PType &result);
```

```
  void eval_exp5(PType &result);
  void eval_exp6(PType &result);
  void atom(PType &result);
  void get_token(), putback();
  void serror(int error);
  PType find_var(char *s);
  int isdelim(char c);
public:
  parser();
  PType eval_exp(char *exp);
};

// Parser constructor.
template <class PType> parser<PType>::parser()
{
  int i;

  exp_ptr = NULL;

  for(i=0; i<NUMVARS; i++) vars[i] = (PType) 0;
}

// Parser entry point.
template <class PType> PType parser<PType>::eval_exp(char *exp)
{
  PType result;

  exp_ptr = exp;

  get_token();
  if(!*token) {
    serror(2); // no expression present
    return (PType) 0;
  }
  eval_exp1(result);
  if(*token) serror(0); // last token must be null
  return result;
}

// Process an assignment.
template <class PType> void parser<PType>::eval_exp1(PType &result)
{
  int slot;
  char ttok_type;
  char temp_token[80];

  if(tok_type==VARIABLE) {
    // save old token
    strcpy(temp_token, token);
    ttok_type = tok_type;
    // compute the index of the variable
    slot = toupper(*token) - 'A';

    get_token();
    if(*token != '=') {
      putback(); // return current token
```

```
      // restore old token - not assignment
      strcpy(token, temp_token);
      tok_type = ttok_type;
    }
    else {
      get_token(); // get next part of exp
      eval_exp2(result);
      vars[slot] = result;
      return;
    }
  }

  eval_exp2(result);
}

// Add or subtract two terms.
template <class PType> void parser<PType>::eval_exp2(PType &result)
{
  register char op;
  PType temp;

  eval_exp3(result);
  while((op = *token) == '+' || op == '-') {
    get_token();
    eval_exp3(temp);
    switch(op) {
      case '-':
        result = result - temp;
        break;
      case '+':
        result = result + temp;
        break;
    }
  }
}

// Multiply or divide two factors.
template <class PType> void parser<PType>::eval_exp3(PType &result)
{
  register char op;
  PType temp;

  eval_exp4(result);
  while((op = *token) == '*' || op == '/' || op == '%') {
    get_token();
    eval_exp4(temp);
    switch(op) {
      case '*':
        result = result * temp;
        break;
      case '/':
        result = result / temp;
        break;
      case '%':
```

```
        result = (int) result % (int) temp;
        break;
    }
  }
}

// Process an exponent.
template <class PType> void parser<PType>::eval_exp4(PType &result)
{
  PType temp, ex;
  register int t;

  eval_exp5(result);
  if(*token== '^') {
    get_token();
    eval_exp4(temp);
    ex = result;
    if(temp==0.0) {
      result = (PType) 1;
      return;
    }
    for(t=(int)temp-1; t>0; --t) result = result * ex;
  }
}

// Evaluate a unary + or -.
template <class PType> void parser<PType>::eval_exp5(PType &result)
{
  register char op;

  op = 0;
  if((tok_type == DELIMITER) && *token=='+' || *token == '-') {
    op = *token;
    get_token();
  }
  eval_exp6(result);
  if(op=='-') result = -result;
}

// Process a parenthesized expression.
template <class PType> void parser<PType>::eval_exp6(PType &result)
{
  if((*token == '(')) {
    get_token();
    eval_exp2(result);
    if(*token != ')')
      serror(1);
    get_token();
  }
  else atom(result);
}

// Get the value of a number or a variable.
template <class PType> void parser<PType>::atom(PType &result)
```

```
{
  switch(tok_type) {
    case VARIABLE:
      result = find_var(token);
      get_token();
      return;
    case NUMBER:
      result = (PType) atof(token);
      get_token();
      return;
    default:
    serror(0);
  }
}

// Return a token to the input stream.
template <class PType> void parser<PType>::putback()
{
  char *t;

  t = token;
  for(; *t; t++) exp_ptr--;
}

// Display a syntax error.
template <class PType> void parser<PType>::serror(int error)
{
  static char *e[] = {
      "Syntax Error",
      "Unbalanced Parentheses",
      "No expression Present"
  };
  cout << e[error] << endl;
}

// Obtain the next token.
template <class PType> void parser<PType>::get_token()
{
  register char *temp;

  tok_type = 0;
  temp = token;
  *temp = '\0';

  if(!*exp_ptr) return; // at end of expression

  while(isspace(*exp_ptr)) ++exp_ptr; // skip over white space

  if(strchr("+-*/%^=()", *exp_ptr)){
    tok_type = DELIMITER;
    // advance to next char
    *temp++ = *exp_ptr++;
  }
  else if(isalpha(*exp_ptr)) {
    while(!isdelim(*exp_ptr)) *temp++ = *exp_ptr++;
```

```
    tok_type = VARIABLE;
  }
  else if(isdigit(*exp_ptr)) {
    while(!isdelim(*exp_ptr)) *temp++ = *exp_ptr++;
    tok_type = NUMBER;
  }

  *temp = '\0';
}

// Return true if c is a delimiter.
template <class PType> int parser<PType>::isdelim(char c)
{
  if(strchr(" +-/*%^=()", c) || c==9 || c=='\r' || c==0)
    return 1;
  return 0;
}

// Return the value of a variable.
template <class PType> PType parser<PType>::find_var(char *s)
{
  if(!isalpha(*s)){
    serror(1);
    return (PType) 0;
  }
  return vars[ toupper(*token)-'A'];
}
```

可以看到，分析器现在所分析的数据类型由通用类型PType指定。下面的main( )函数演示了这个通用的分析器。

```
int main()
{
  char expstr[ 80];

  // Demonstrate floating-point parser.
  parser<double> ob;

  cout << "Floating-point parser. ";
  cout << "Enter a period to stop\n";
  for(;;) {
    cout << "Enter expression: ";
    cin.getline(expstr, 79);
    if(*expstr=='.') break;
    cout << "Answer is: " << ob.eval_exp(expstr) << "\n\n";
  }
  cout << endl;

  // Demonstrate integer-based parser.
  parser<int> Iob;

  cout << "Integer parser. ";
  cout << "Enter a period to stop\n";
  for(;;) {
    cout << "Enter expression: ";
```

```
    cin.getline(expstr, 79);
    if(*expstr=='.') break;
    cout << "Answer is: " << Iob.eval_exp(expstr) << "\n\n";
  }

  return 0;
}
```

下面是一个范例运行的结果。

```
Floating-point parser. Enter a period to stop
Enter expression: a=10.1
Answer is: 10.1

Enter expression: b=3.2
Answer is: 3.2

Enter expression: a/b
Answer is: 3.15625

Enter expression: .

Integer parser. Enter a period to stop
Enter expression: a=10
Answer is: 10

Enter expression: b=3
Answer is: 3

Enter expression: a/b
Answer is: 3

Enter expression: .
```

可以看到，浮点分析器使用浮点值，整数分析器使用整数值。

40.10　需要试验的一些东西

正如本章前面所述，分析器只执行很少的错误检查。你可能想要添加详细的错误报告。例如，你可能想突出表达式中检测到错误的点。这会允许用户发现并更正一个语法错误。

虽然现在分析器只能对数字表达式求值，然面，在加了一些条件后，就可以使分析器对其他类型的表达式求值了，例如字符串、三维坐标或复数。例如，要允许分析器对字符串对象求值，必须进行下列改变：

1. 定义一个新的类型项 STRING。
2. 增强 get_token( )，以便它可以识别字符串。
3. 在 atom( )内添加处理 STRING 类型项的新情况。

在实现了所有这些步骤后，分析器可以处理像下面这样的字符串表达式：

```
a = "one"
b = "two"
c = a + b
```

c 的结果应该是 a 和 b 连接后的情况，或者 "onetwo"。

下面是分析器的一个好的应用：创建一个简单的弹出式小计算器，这个小计算器接受用户键入的表达式，然后显示结果。这可添加到几乎任何的商用应用程序中。如果正在进行Windows编程，这特别容易实现。