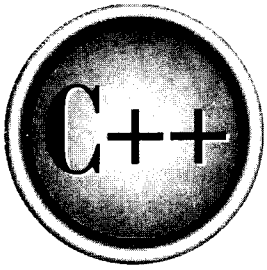
The Complete Reference



Chapter 40

Parsing Expressions

While Standard C++ is quite extensive, there are still a few things that it does not provide. In this chapter we will examine one of them: the *expression parser*. An expression parser is used to evaluate an algebraic expression, such as (10 – 8) * 3. Expression parsers are quite useful and are applicable to a wide range of applications. They are also one of programming's more elusive entities. For various reasons, the procedures used to create an expression parser are not widely taught or disseminated. Indeed, many otherwise accomplished programmers are mystified by the process of expression parsing.

Expression parsing is actually very straightforward, and in many ways easier than other programming tasks. The reason for this is that the task is well defined and works according to the strict rules of algebra. This chapter will develop what is commonly referred to as a *recursive-descent parser* and all the necessary support routines that enable you to evaluate numeric expressions. Three versions of the parser will be created. The first two are nongeneric versions. The final one is generic and may be applied to any numeric type. However, before any parser can be developed, a brief overview of expressions and parsing is necessary.

Expressions

Since an expression parser evaluates an algebraic expression, it is important to understand what the constituent parts of an expression are. Although expressions can be made up of all types of information, this chapter deals only with numeric expressions. For our purposes, numeric expressions are composed of the following items:

- Numbers
- The operators +, –, /, *, ^, %, =
- Parentheses
- Variables

For our parser, the operator ^ indicates exponentiation (not the XOR as it does in C++), and = is the assignment operator. These items can be combined in expressions according to the rules of algebra. Here are some examples:

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

Assume this precedence for each operator:

highest	+ – (unary)
	^
	* / %
	+ –
lowest	=

Operators of equal precedence evaluate from left to rigl

In the examples in this chapter, all variables are single letters (in other words, 26 variables, **A** through **Z**, are available). The variables are not case sensitive (**a** and **A** are treated as the same variable). For the first version of the parser, all numeric values are elevated to **double**, although you could easily write the routines to handle other types of values. Finally, to keep the logic clear and easy to understand, only a minimal amount of error checking is included.

Parsing Expressions: The Problem

If you have not thought much about the problem of expression parsing, you might assume that it is a simple task. However, to better understand the problem, try to evaluate this sample expression:

```
10 - 2 * 3
```

You know that this expression is equal to the value 4. Although you could easily create a program that would compute that *specific* expression, the question is how to create a program that gives the correct answer for any *arbitrary* expression. At first you might think of a routine something like this:

```
a = get first operand
while(operands present) {
    op = get operator
    b = get second operand
    a = a op b
}
```

This routine gets the first operand, the operator, and the second operand to perform the first operation and then gets the next operator and operand to perform the next operation, and so on. However, if you use this basic approach, the expression 10 – 2 * 3 evaluates to 24 (that is, 8 * 3) instead of 4 because this procedure neglects the precedence of the operators. You cannot just take the operands and operators in order from left to right because the rules of algebra dictate that multiplication must be done before subtraction. Some beginners think that this problem can be easily overcome, and sometimes, in very restricted cases, it can. But the problem only gets worse when you add parentheses, exponentiation, variables, unary operators, and the like.

Although there are a few ways to write a routine that evaluates expressions, the one developed here is the one most easily written by a person. It is also the most common. The method used here is called a *recursive-descent parser*, and in the course of this chapter you will see how it got its name. (Some of the other methods used to write parsers employ complex tables that must be generated by another computer program. These are sometimes called *table-driven parsers*.)

Parsing an Expression

There are a number of ways to parse and evaluate an expression. For use with a recursive-descent parser, think of expressions as *recursive data structures*—that is, expressions that are defined in terms of themselves. If, for the moment, we assume that expressions can only use +, –, *, /, and parentheses, all expressions can be defined with the following rules:

expression –> term [+ term] [– term]

term –> factor [* factor] [/ factor]

factor –> variable, number, or (expression)

The square brackets designate an optional element, and the –> means *produces*. In fact, the rules are usually called the *production rules* of the expression. Therefore, you could say: "Term produces factor times factor or factor divided by factor" for the definition of *term*. Notice that the precedence of the operators is implicit in the way an expression is defined.

The expression

10 + 5 * B

has two terms: 10, and 5 * B. The second term contains two factors: 5 and B. These factors consist of one number and one variable.

On the other hand, the expression

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

has two factors: 14 and (7 – C). The factors consist of one number and one parenthesized expression. The parenthesized expression contains two terms: one number and one variable.

This process forms the basis for a recursive-descent parser, which is a set of mutually recursive functions that work in a chainlike fashion and implement the production rules. At each appropriate step, the parser performs the specified operations in the algebraically correct sequence. To see how the production rules are used to parse an expression, let's work through an example using this expression:

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

Here is the sequence that you will follow:

1. Get the first term, 9/3.
2. Get each factor and divide the integers. The resulting value is 3.
3. Get the second term, (100 + 56). At this point, start recursively analyzing the second subexpression.
4. Get each term and add. The resulting value is 156.
5. Return from the recursive call, and subtract 156 from 3. The answer is –153.

If you are a little confused at this point, don't feel bad. This is a fairly complex concept that takes some getting used to. There are two basic things to remember about this recursive view of expressions. First, the precedence of the operators is implicit in the way the production rules are defined. Second, this method of parsing and evaluating expressions is very similar to the way humans evaluate mathematical expressions.

The remainder of this chapter develops three parsers. The first will parse and evaluate floating-point expressions of type **double** that consist only of constant values. Next, this parser is enhanced to support the use of variables. Finally, in the third version, the parser is implemented as a template class that can be used to parse expressions of any type.

The Parser Class

The expression parser is built upon the **parser** class. The first version of **parser** is shown here. Subsequent versions of the parser build upon it.

```
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);
};
```

The **parser** class contains three private member variables. The expression to be evaluated is contained in a null-terminated string pointed to by **exp_ptr**. Thus, the parser evaluates expressions that are contained in standard ASCII strings. For example, the following strings contain expressions that the parser can evaluate:

"10 - 5"

"2 * 3.3 / (3.1416 * 3.3)"

When the parser begins execution, **exp_ptr** must point to the first character in the expression string. As the parser executes, it works its way through the string until the null-terminator is encountered.

The meaning of the other two member variables, **token** and **tok_type**, are described in the next section.

The entry point to the parser is through **eval_exp( )**, which must be called with a pointer to the expression to be analyzed. The functions **eval_exp2( )** through **eval_exp6( )** along with **atom( )** form the recursive-descent parser. They implement an enhanced set of the expression production rules discussed earlier. In subsequent versions of the parser, a function called **eval_exp1( )** will also be added.

The **serror( )** handles syntax errors in the expression. The functions **get_token( )** and **isdelim( )** are used to dissect the expression into its component parts, as described in the next section.

Dissecting an Expression

In order to evaluate expressions, you need to be able to break an expression into its components. Since this operation is fundamental to parsing, let's look at it before examining the parser itself.

Each component of an expression is called a *token*. For example, the expression

A * B – (W + 10)

contains the tokens A, *, B, –, (, W, +, 10, and). Each token represents an indivisible unit of the expression. In general, you need a function that sequentially returns each token in the expression individually. The function must also be able to skip over spaces and tabs and detect the end of the expression. The function that we will use to perform this task is called **get_token( )**, which is a member function of the **parser** class.

Besides the token, itself, you will also need to know what type of token is being returned. For the parser developed in this chapter, you need only three types: **VARIABLE**, **NUMBER**, and **DELIMITER**. (**DELIMITER** is used for both operators and parentheses.)

The **get_token( )** function is shown here. It obtains the next token from the expression pointed to by **exp_ptr** and puts it into the member variable **token**. It puts the type of the token into the member variable **tok_type**.

```
// 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;
}
```

Look closely at the preceding functions. After the first few initializations, **get_token( )** checks to see if the null terminating the expression has been found. It does so by checking the character pointed to by **exp_ptr**. Since **exp_ptr** is a pointer to the expression being analyzed, if it points to a null, the end of the expression has been reached. If there are still more tokens to retrieve from the expression, **get_token( )** first skips over any leading spaces. Once the spaces have been skipped, **exp_ptr** is pointing to either a number, a variable, an operator, or if trailing spaces end the expression, a null. If the next character is an operator, it is returned as a string in **token**, and **DELIMITER** is placed in **tok_type**. If the next character is a letter instead, it is assumed to be one of the variables. It is returned as a string in **token**, and **tok_type** is assigned the value **VARIABLE**. If the next character is a digit, the entire number is read and placed in its string form in **token** and its type is **NUMBER**. Finally, if the next character is none of the preceding, it is assumed that the end of the expression has been reached. In this case, **token** is null, which signals the end of the expression.

As stated earlier, to keep the code in this function clean, a certain amount of error checking has been omitted and some assumptions have been made. For example, any unrecognized character may end an expression. Also, in this version, variables may be of any length, but only the first letter is significant. You can add more error checking and other details as your specific application dictates.

To better understand the tokenization process, study what it returns for each token and type in the following expression:

A + 100 – (B * C) /2

Token	Token type
A	VARIABLE
+	DELIMITER
100	NUMBER
–	DELIMITER
(	DELIMITER
B	VARIABLE
*	DELIMITER
C	VARIABLE
)	DELIMITER
/	DELIMITER
2	NUMBER
null	null

Remember that **token** always holds a null-terminated string, even if it contains just a single character.

A Simple Expression Parser

Here is the first version of the parser. It can evaluate expressions that consist solely of constants, operators, and parentheses. It cannot accept expressions that contain variables.

```
/* 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;
}
```

The parser as it is shown can handle the following operators: +, –, *, /, %. In addition, it can handle integer exponentiation (^) and the unary minus. The parser can also deal with parentheses correctly. The actual evaluation of an expression takes place in the mutually recursive functions **eval_exp2( )** through **eval_exp6( )**, plus the **atom( )** function, which returns the value of a number. The comments at the start of each function describe what role it plays in parsing the expression.

The simple **main( )** function that follows demonstrates the use of the parser.

```
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;
}
```

Here is a sample run.

```
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: .
```

Understanding the Parser

To understand exactly how the parser evaluates an expression, work through the following expression. (Assume that **exp_ptr** points to the start of the expression.)

10 – 3 * 2

When **eval_exp( )**, the entry point into the parser, is called, it gets the first token. If the token is null, the function prints the message **No Expression Present** and returns. However, in this case, the token contains the number **10**. Since the first token is not null, **eval_exp2( )** is called. As a result, **eval_exp2( )** calls **eval_exp3( )**, and **eval_exp3( )** calls

eval_exp4( ), which in turn calls **eval_exp5( )**. Then **eval_exp5( )** checks whether the token is a unary plus or minus, which in this case it is not, so **eval_exp6( )** is called. At this point **eval_exp6( )** either recursively calls **eval_exp2( )** (in the case of a parenthesized expression) or calls **atom( )** to find the value of a number. Since the token is not a left parentheses, **atom( )** is executed and **result** is assigned the value 10. Next, another token is retrieved, and the functions begin to return up the chain. Since the token is now the operator –, the functions return up to **eval_exp2( )**.

What happens next is very important. Because the token is –, it is saved in **op**. The parser then gets the next token, which is **3**, and the descent down the chain begins again. As before, **atom( )** is entered. The value 3 is returned in **result**, and the token * is read. This causes a return back up the chain to **eval_exp3( )**, where the final token 2 is read. At this point, the first arithmetic operation occurs—the multiplication of 2 and 3. The result is returned to **eval_exp2( )**, and the subtraction is performed. The subtraction yields the answer 4. Although the process may at first seem complicated, work through some other examples to verify that this method functions correctly every time.

This parser would be suitable for use by a simple desktop calculator, as is illustrated by the previous program. Before it could be used in a computer language, database, or in a sophisticated calculator, however, it would need the ability to handle variables. This is the subject of the next section.

Adding Variables to the Parser

All programming languages, many calculators, and spreadsheets use variables to store values for later use. Before the parser can be used for such applications, it needs to be expanded to include variables. To accomplish this, you need to add several things to the parser. First, of course, are the variables themselves. As stated earlier, we will use the letters **A** through **Z** for variables. The variables will be stored in an array inside the **parser** class. Each variable uses one array location in a 26-element array of **doubles**. Therefore, add the following to the **parser** class:

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

You will also need to change the **parser** constructor, as shown here.

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

  exp_ptr = NULL;
```

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

As you can see, the variables are initialized to 0 as a courtesy to the user.

You will also need a function to look up the value of a given variable. Because the variables are named **A** through **Z**, they can easily be used to index the array **vars** by subtracting the ASCII value for **A** from the variable name. The member function **find_var( )**, shown here, accomplishes this:

```
// 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'];
}
```

As this function is written, it will actually accept long variable names, but only the first letter is significant. You may modify this to fit your needs.

You must also modify the **atom( )** function to handle both numbers and variables. The new version is shown here:

```
// 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);
  }
}
```

Technically, these additions are all that is needed for the parser to use variables correctly; however, there is no way for these variables to be assigned a value. Often this is done outside the parser, but you can treat the equal sign as an assignment operator (which is how it is handled in C++) and make it part of the parser. There are various ways to do this. One method is to add another function, called **eval_exp1( )**, to the **parser** class. This function will now begin the recursive-descent chain. This means that it, not **eval_exp2( )**, must be called by **eval_exp( )** to begin parsing the expression. **eval_exp1( )** is shown here:

```
// 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);
}
```

As you can see, the function needs to look ahead to determine whether an assignment is actually being made. This is because a variable name always precedes an assignment, but a variable name alone does not guarantee that an assignment expression follows. That is, the parser will accept A = 100 as an assignment, but is also smart enough to know that A/10 is not. To accomplish this, **eval_exp1( )** reads the next token from the input stream. If it is not an equal sign, the token is returned to the input stream for later use by calling **putback( )**. The **putback( )** function must also be included in the **parser** class. It is shown here:

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

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

After making all the necessary changes, the parser will now look like this.

```
/* 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'];
}
```

To try the enhanced parser, you may use the same **main( )** function that you used for the simple parser. With the enhanced parser, you can now enter expressions like

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

Syntax Checking in a Recursive-Descent Parser

Before moving on to the template version of the parser, let's briefly look at syntax checking. In expression parsing, a syntax error is simply a situation in which the input expression does not conform to the strict rules required by the parser. Most of the time, this is caused by human error, usually typing mistakes. For example, the following expressions are not valid for the parsers in this chapter:

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

The first contains two operators in a row, the second has unbalanced parentheses, and the last has a division sign at the start of an expression. None of these conditions is allowed by the parsers. Because syntax errors can cause the parser to give erroneous results, you need to guard against them.

As you studied the code of the parsers, you probably noticed the **serror( )** function, which is called under certain situations. Unlike many other parsers, the recursive-descent method makes syntax checking easy because, for the most part, it occurs in **atom( )**, **find_var( )**, or **eval_exp6( )**, where parentheses are checked. The only problem with the syntax checking as it now stands is that the entire parser is not terminated on syntax error. This can lead to multiple error messages.

The best way to implement the **serror( )** function is to have it execute some sort of reset. For example, all C++ compilers come with a pair of companion functions called **setjmp( )** and **longjmp( )**. These two functions allow a program to branch to a *different* function. Therefore, **serror( )** could execute a **longjmp( )** to some safe point in your program outside the parser.

Depending upon the use you put the parser to, you might also find that C++'s exception handling mechanism (implemented through **try**, **catch**, and **throw**) will be beneficial when handling errors.

If you leave the code the way it is, multiple syntax-error messages may be issued. This can be an annoyance in some situations but a blessing in others because multiple errors may be caught. Generally, however, you will want to enhance the syntax checking before using it in commercial programs.

Building a Generic Parser

The two preceding parsers operated on numeric expressions in which all values were assumed to be of type **double**. While this is fine for applications that use **double** values, it is certainly excessive for applications that use only integer values, for example. Also, by hard-coding the type of values being evaluated, the application of the parser is unnecessarily restricted. Fortunately, by using a class template, it is an easy task to create a generic version of the parser that can work with any type of data for which algebraic-style expressions are defined. Once this has been done, the parser can be used both with built-in types and with numeric types that you create.

Here is the generic version of the expression parser.

```
// 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'];
}
```

As you can see, the type of data now operated upon by the parser is specified by the generic type **PType**. The following **main( )** function demonstrates the generic parser.

```
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;
}
```

Here is a sample run.

```
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: .
```

As you can see, the floating-point parser uses floating-point values, and the integer parser uses integer values.

Some Things to Try

As mentioned early on in this chapter, only minimal error checking is performed by the parser. You might want to add detailed error reporting. For example, you could highlight the point in the expression at which an error was detected. This would allow the user to find and correct a syntax error.

As the parser now stands it can evaluate only numeric expressions. However, with a few additions, it is possible to enable the parser to evaluate other types of expressions, such as strings, spatial coordinates, or complex numbers. For example, to allow the parser to evaluate string objects, you must make the following changes:

1. Define a new token type called STRING.
2. Enhance **get_token( )** so that it recognizes strings.
3. Add a new case inside **atom( )** that handles STRING type tokens.

After implementing these steps, the parser could handle string expressions like these:

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

The result in **c** should be the concatenation of **a** and **b**, or "onetwo".

Here is one good application for the parser: create a simple, pop-up mini-calculator that accepts an expression entered by the user and then displays the result. This would make an excellent addition to nearly any commercial application. If you are programming for Windows, this would be especially easy to do.