CHAPTER 12

Binary Trees

In the previous chapter, you learned about general trees, which are trees that can have any number of branches per node. Now I'm going to show you the most popular variant of the tree structure: the *binary tree*.

In this chapter, you will learn

- What a binary tree is
- Some common traits of binary trees
- Two common implementations of binary trees
- How to program a linked binary tree
- How to perform the two tree traversals on a binary tree
- How to perform a new traversal specific to the binary tree structure
- How to build a simple arithmetic expression parser using binary trees

What Is a Binary Tree?

A binary tree is a very simple variant of the general tree structure, and it is often used in game programming. In fact, almost every tree-based structure in this book uses a binary tree as its base.

Simply put, a binary tree is a tree that can have up to two children. These two children are usually called the *left* and the *right* children of the tree. Figure 12.1 shows a binary tree node.

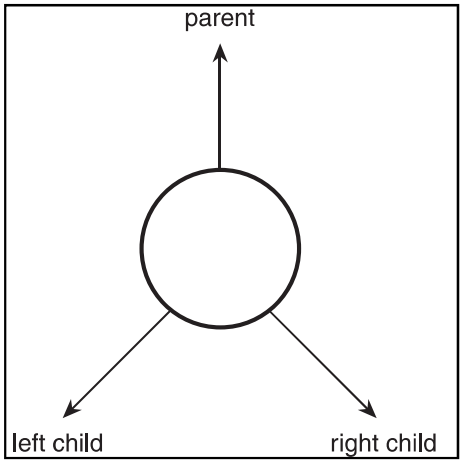

Figure 12.1

This is a binary tree node.

As you can see, there really isn't much to learn about plain binary trees because they are the simplest of all tree structures. A binary tree can have several traits that general trees cannot have, though.

Fullness

A binary tree can be *full*. Because each node can have a maximum of two child nodes, you can fill up a tree so that you cannot insert any more nodes without making the tree go down a level.

Figure 12.2 shows a full four-level binary tree.

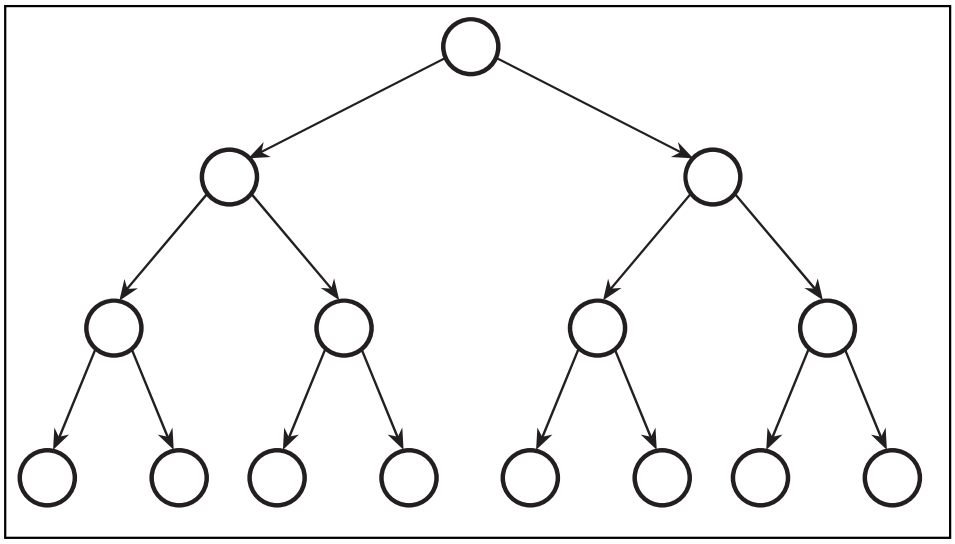

Figure 12.2

Here is a full binary tree. You cannot add more nodes to this tree without making it increase in size by another level.

In a full binary tree, every leaf node must be on the same level, and every non-leaf node must have two children.

Denseness

Another property of binary trees is called *denseness*. Sometimes this is also called *completeness* or *leftness*. A dense binary tree is similar to a full tree, except that in the bottom level of a tree, every node is packed to the left side of the tree.

Figure 12.3 shows a dense binary tree.

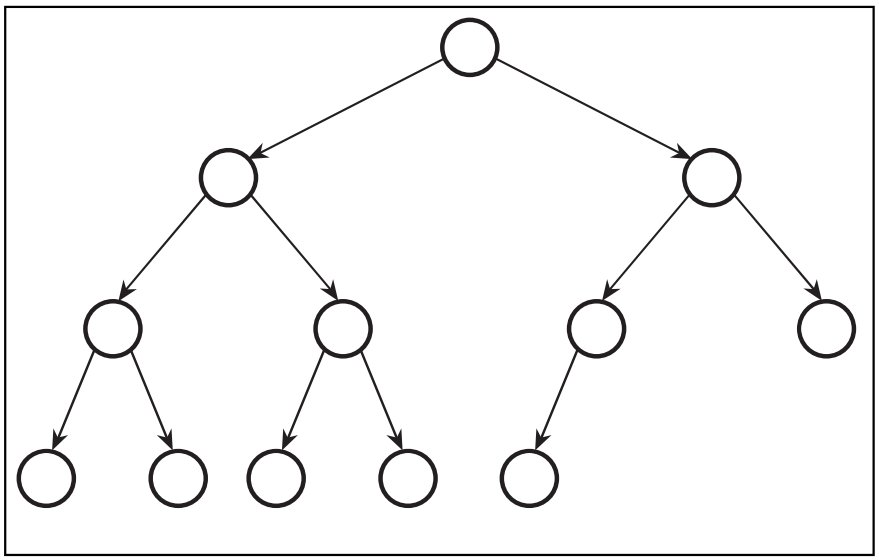

Figure 12.3
Here is a dense binary tree. Every level is full, except the last level, where the nodes are all packed to the left of the tree.

Denseness is an important trait with some variants of binary trees, as you'll see later on in this chapter and when I teach you about heaps in Chapter 14, "Priority Queues and Heaps."

Balance

Even though I don't really use this trait in this book, I feel it is important enough to mention. A *balanced* tree is a tree in which every node in the tree has approximately as many children in the left side as the right side. This property becomes important when using some of the binary search tree (*BST*) variants, such as *AVL trees* and *red-black trees (RBT)*. I discuss BSTs in Chapter 13, "Binary Search Trees," but not AVL trees or RBTs. They are fairly complex and used to solve specific problems that don't occur in most game programming situations; we will skip them because this is a game programming book.

Structure of Binary Trees

You can store a binary tree in two ways. The first method is the most common, and it's very similar to the `Tree` class. The second method is not as common, but it has its uses.

Linked Binary Trees

A *linked* binary tree is just like the regular tree structure and therefore is node-based. Instead of using a linked list of child pointers, though, the linked binary

tree node has two *fixed* pointers. The fixed pointers either point to the left or right child nodes or contain 0 if the node doesn't have a child. The structure for these kinds of nodes is shown in Figure 12.4

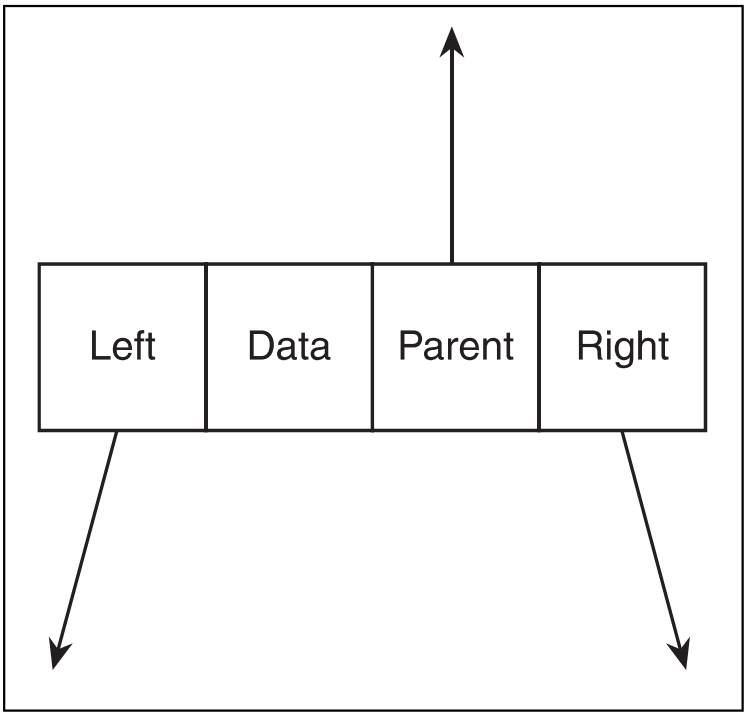

Figure 12.4

This is a linked binary tree node.

The three boxes with arrows coming out of them are all pointers that point to another node structure. Note that I included a parent pointer in the node; even though it is not necessary, I feel that it saves a lot of trouble when working with binary trees.

This method of structuring nodes is great because it allows for an effectively limitless tree size due to the linked nature of the tree.

Arrayed Binary Trees

There is another method of storing binary trees, however. You've seen how a binary tree can be full because the number of children in a binary tree is fixed at two.

Because you know that a binary tree can only have a certain number of nodes depending on the height of the tree, you can make certain assumptions. For example, imagine what would happen if you turned every node from the full binary tree in Figure 12.2 into an array cell. Figure 12.5 shows what I mean by this.

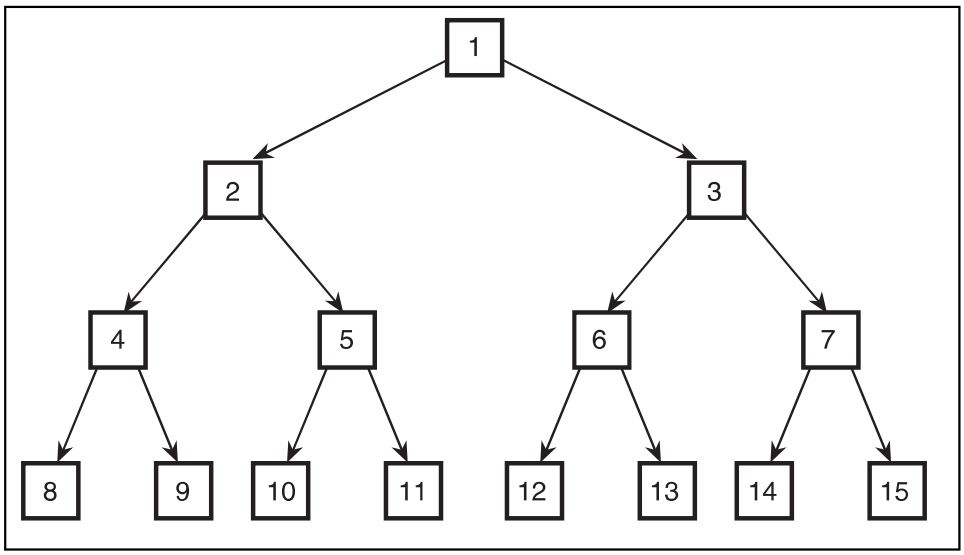

Figure 12.5

This is a full binary tree where the nodes have been turned into array cells.

Pay particular attention to the order in which I numbered the cells. The root starts at index 1 and the numbering goes from left to right all the way down to the last node on the right, 15. Now, imagine if you concatenated all of the cells into an array of cells, like Figure 12.6 shows.

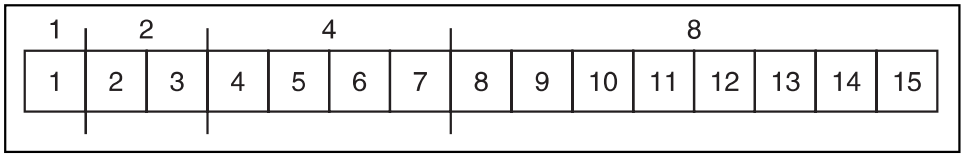

Figure 12.6

This is how you would represent a binary tree as an array.

The array is separated into four different segments, each with a number on top. The segments represent the levels of the tree. The first segment is only one cell in size because there is only one root node. The second segment contains two cells because there are two nodes on the second level of a binary tree. Likewise, the third segment has four cells, and the fourth segment has eight cells.

Size of Arrayed Binary Trees

The number of nodes on a level of a full binary tree doubles with each new level, and follows this formula: nodes for level $n = 2^{n-1}$. Therefore, the number of nodes required for level 5 would be 2^4, or 16.

The total number of cells in a binary tree of a particular depth follows this formula: cells for depth $n = 2^n-1$. For example, in the four-level tree in Figure 12.5, there are $2^4 - 1$ nodes, or 15. A binary tree with five levels requires 31 nodes.

Traversing Arrayed Binary Trees

You don't need iterators to traverse arrayed binary trees. A few easy algorithms allow you to determine the index of the left, right, and parent nodes of a binary tree cell.

Take a look back at Figure 12.5 and see if you can find a relationship between the index of any node and its left child. It is easy to see that the index of the left child of any node is twice the index of its parent. By using this knowledge, you can create a function that determines the left child of any cell in the tree:

```
left = index * 2;
```

That was easy enough, wasn't it? Now, see if you can figure out how to calculate the index of the right child of any cell. Because the right child of any node is only one index higher than the left child, you can use that formula to create the formula for finding the right child:

```
right = index * 2 + 1;
```

The last thing you need to figure out is how to get to the parent node from any node in the tree. If you look at the formula for finding the left node and reverse it, you get this:

```
parent = index / 2;
```

That works for left children, because the left children are all even numbers and are divisible by 2, but what about right children? What happens when you divide 3 in half? Although 3/2 is 1.5, the extra 0.5 is cut off because these algorithms are using integers, giving 1 as the result. So the parent algorithm works on any node.

Size Efficiency

I've said before that arrayed binary trees are not as common as linked trees. This is due to several reasons, but first, look at Figure 12.7.

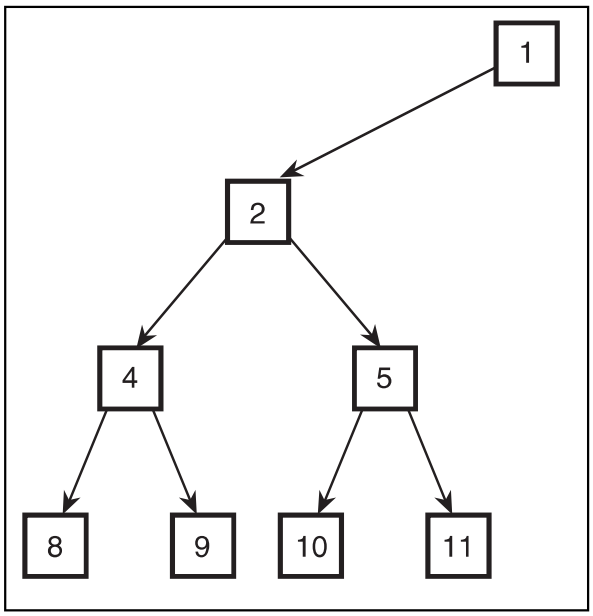

Figure 12.7

Here is another binary tree, where a lot of space is wasted.

The tree in Figure 12.7 is the same as the tree in Figure 12.5, but the entire subtree starting with index 3 has been removed. Imagine how this tree looks when stored into an array, though. Figure 12.8 shows this.

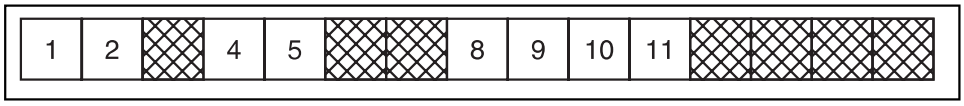

Figure 12.8

This is the tree from Figure 12.7 stored in an array.

The tree from Figure 12.7 has 8 nodes, but the array has 15 cells, which means that 7 cells are empty! That's almost half of the array!

Granted, the last 4 cells are unused, so you could chop them off the array, but what happens if you insert a left child onto node 8? Then the child would need to be stored into cell 16, requiring you to resize the array.

This example shows that using arrays to store binary trees is very inefficient if your trees aren't full or dense.

Graphical Demonstration: Binary Trees

This is Graphical Demonstration 12-1, which you can find on the CD in the directory \demonstrations\ch12\Demo01 – Binary Trees\.

Compiling the Demo

This demonstration uses the SDLGUI library that I have developed for the book. For more information about this library, see Appendix B, "The Memory Layout of a Computer Program."

To compile this demo, either open up the workspace file in the directory or create your own project using the settings described in Appendix B. If you create your own project, all of the files you need to include are in the directory.

Figure 12.9 shows a screenshot from this demonstration. The demo has eight different buttons, and Table 12.1 has a listing of what they do.

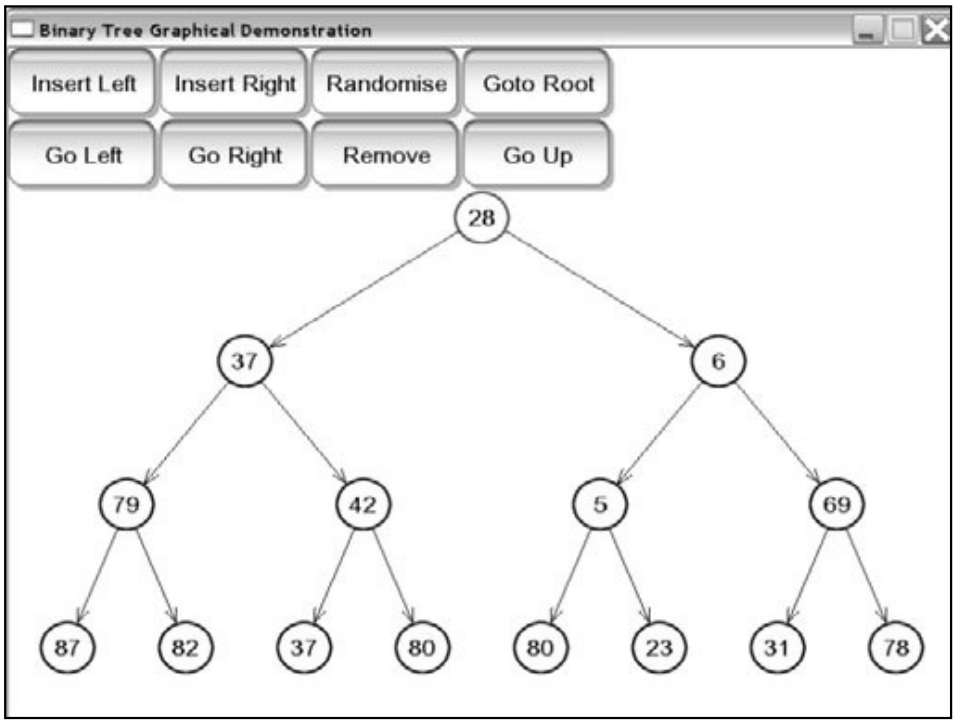

Figure 12.9

Here is a screenshot from the binary tree demonstration.

Table 12.1 Binary Tree Demonstration Commands

Command	Action
Insert Left	Inserts a new node to the left of the current node if there is none
Insert Right	Inserts a new node to the right of the current node if there is none
Go Left	Moves the current node to the left child node
Go Right	Moves the current node to the right child node
Randomize	Randomizes the tree
Remove	Removes the current node, unless it is the root
Goto Root	Moves the current node iterator to the root of the tree
Go Up	Moves the current node iterator up one level

As in the Tree graphical demonstration, the current node is highlighted in red. Play around with the demo to familiarize yourself with binary trees a bit more.

Coding a Binary Tree

All of the code for the Binary Tree structure and algorithms is located on the CD in the file \structures\BinaryTree.h.

Lucky for you, coding a binary tree isn't nearly as difficult as coding a general tree. In fact, you don't even need an iterator class with a binary tree; you can just as easily use a pointer to a node as the iterator.

Note that I'm not including an `Arrayed Binary Tree` class. Because an arrayed binary tree is essentially an array, there is no need to include one.

The Structure

As I stated before, the binary tree class has four variables:

```
template<class DataType>
class BinaryTree
{
```

```
public:
    DataType m_data;
    Node* m_parent;
    Node* m_left;
    Node* m_right;
};
```

They are the data, a pointer to the parent, and a pointer to the left and right children.

The Constructor

The constructor exists to clear the pointers so that they aren't filled with garbage data when a node is created.

```
BinaryTree()
{
    m_parent = 0;
    m_left = 0;
    m_right = 0;
}
```

The Destructor and the Destroy Function

The destructor of the `BinaryTree` class just calls the `Destroy` function, like the `Tree` class did, so there is no need to paste the code here.

However, the `Destroy` function is slightly different than before:

```
void Destroy()
{
    if( m_left )
        delete m_left;
    m_left = 0;
    if( m_right )
        delete m_right;
    m_right = 0;
}
```

This function determines if the node has a left child and deletes it if it does, and then it determines if it has a right child and deletes it if it does. As before, the function is recursive because the destructor of each child node calls `Destroy`.

The Count Function

The `Count` function is only slightly modified from the `Tree` version; instead of looping through the child list, it calls the `Count` function on each child of the node.

```
int Count()
{
    int c = 1;
    if( m_left )
        c += m_left->Count();
    if( m_right )
        c += m_right->Count();
    return c;
}
```

Note that it checks to see if each child node exists before calling the `Count` function on it.

Using the BinaryTree Class

This is Example 12-1, which can be found on the CD in the directory \examples\ch12\01 – Binary Tree\.

This example takes you through the process of building a simple three-level full binary tree.

The first step is to declare the tree root and an iterator:

```
BinaryTree<int>* root = 0;
BinaryTree<int>* itr = 0;
```

After that, you need to initialize the root of the tree:

```
root = new BinaryTree<int>;
root->m_data = 1;
```

Then you create the left and right child nodes of the root node:

```
root->m_left = new BinaryTree<int>;
root->m_left->m_data = 2;
root->m_left->m_parent = root;

root->m_right = new BinaryTree<int>;
root->m_right->m_data = 3;
root->m_right->m_parent = root;
```

Now, the iterator is put to work to create the nodes lower down in the tree:

```
itr = root;
itr = itr->m_left;
itr->m_left = new BinaryTree<int>;
itr->m_left->m_data = 4;
itr->m_left->m_parent = itr;

itr->m_right = new BinaryTree<int>;
itr->m_right->m_data = 5;
itr->m_right->m_parent = itr;
```

The iterator is first pointed at the root node and then is moved down to the left node of the root. After that, node 4 is inserted at the left of node 2, and node 5 is inserted at the right.

Now you want to go back up one level:

```
itr = itr->m_parent;
```

And now go back down to the right and do the same thing:

```
itr = itr->m_right;
itr->m_left = new BinaryTree<int>;
itr->m_left->m_data = 6;
itr->m_left->m_parent = itr;

itr->m_right = new BinaryTree<int>;
itr->m_right->m_data = 7;
itr->m_right->m_parent = itr;
```

As you can see, iterating through a binary tree is simple because you know there are only two children per node.

Traversing the Binary Tree

If you remember, the general tree structure had two simple traversal methods: the preorder and the postorder. The binary tree structure allows for another type of traversal, called the *inorder* traversal, as well.

I'll show you how to accomplish all three. The actual C++ code for these functions is in the BinaryTree.h file and is almost identical to the code for the general tree traversal functions, so I won't include it here. If you need clarification, the "Traversing a Tree" section in Chapter 11, "Trees," describes how the traversal functions work.

The Preorder Traversal

The preorder traversal for a binary tree is simple, and it is almost identical to the algorithm used for general trees:

```
Preorder( node )
    process( node )
    Preorder( node.left )
    Preorder( node.right )
End Preorder
```

It is important to note that the left node is processed before the right node; that is the general convention used by all binary trees.

The Postorder Traversal

Just like last time, the postorder traversal processes the current node after the child nodes:

```
Postorder( node )
    Postorder( node.left )
    Postorder( node.right )
    process( node )
End Postorder
```

The Inorder Traversal

So, if the *pre*order traversal processes the current node *before* the children, and the *post*order traversal processes the current node *after* the children, what do you think the *in*order traversal does?

That's right, it processes the current node in between the children nodes:

```
Inorder( node )
    Inorder( node.left)
    process( node )
    Inorder( node.right )
End Inorder
```

This traversal assures that the entire left subtree of every node is processed before the current node and the right subtree. Remember this traversal; you'll be using it for a neat trick in Chapter 20, "Sorting Data."

Figure 12.10 shows the order in which nodes are processed in a binary tree using the inorder traversal. Note the general trend of processing the nodes from left to right.

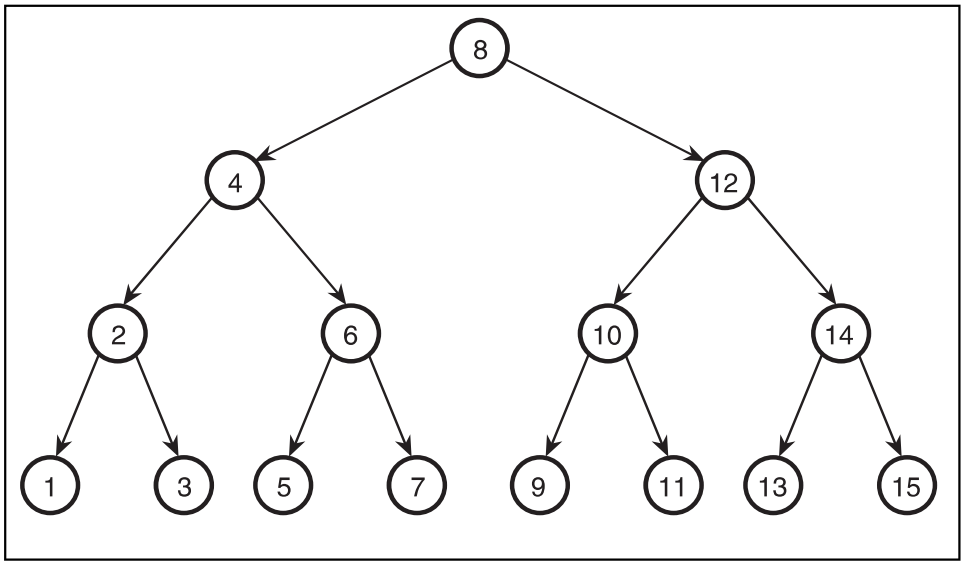

Figure 12.10

This is the order of nodes processed using the inorder traversal.

Graphical Demonstration: Binary Tree Traversals

This is Graphical Demonstration 12-2, which is located on the CD in the directory \demonstrations\ch12\Demo02 - Binary Tree Traversals\.

Compiling the Demo

This demonstration uses the SDLGUI library that I have developed for the book. For more information about this library, see Appendix B.

To compile this demo, either open up the workspace file in the directory or create your own project using the settings described in Appendix B. If you create your own project, all of the files you need to include are in the directory.

This demonstration is almost the same as Graphical Demonstration 11-2, except that it has an extra button to execute the inorder traversal. Figure 12.11 shows a screenshot of the demo in action.

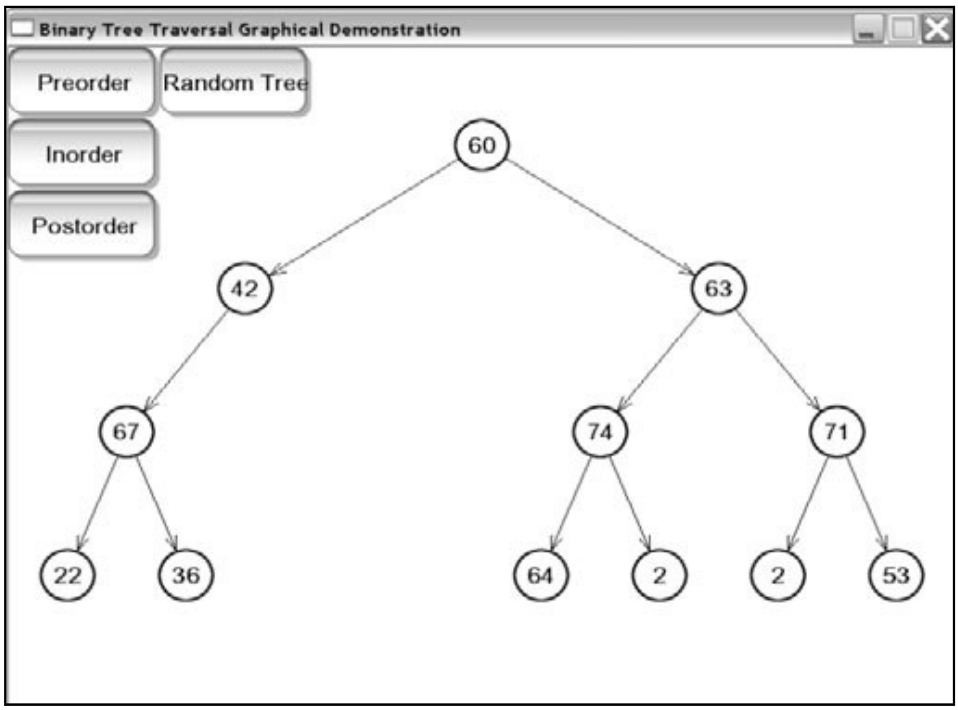

Figure 12.11

Here is a screenshot from the traversal demo.

As before, the nodes will be highlighted for 700 milliseconds while they are being processed to show you the order in which they are visited by the algorithms.

Application: Parsing

This next topic, although it's a little advanced, is a really neat application of binary trees. The code for this section is on the CD in the directory \demonstrations\ch12\Game01 - Parsing\.

Compiling the Demo

This demonstration uses the SDLGUI library that I have developed for the book. For more information about this library, see Appendix B.

To compile this demo, either open up the workspace file in the directory or create your own project using the settings described in Appendix B. If you create your own project, all of the files you need to include are in the directory.

Parsing is the act of breaking up a sentence into easy-to-understand segments. For example, when you read a sentence, your mind mentally parses it into a form that makes sense to you.

Take the following sentence, for example: "Bob runs up the hill." Your mind recognizes that sentence, and it has parsed it into several segments. I don't want to turn this into an English lecture, but a lot of computer language theory is based in concepts that English linguists invented.

The sentence can be broken up into these fragments: verb phrase, preposition, noun phrase. Bob runs, up, the hill. The two phrases can then be broken down further; the verb phrase is a combination of a noun and a verb, and the noun phrase is a combination of an article and a noun. Figure 12.12 shows the tree that is created when your mind parses the sentence.

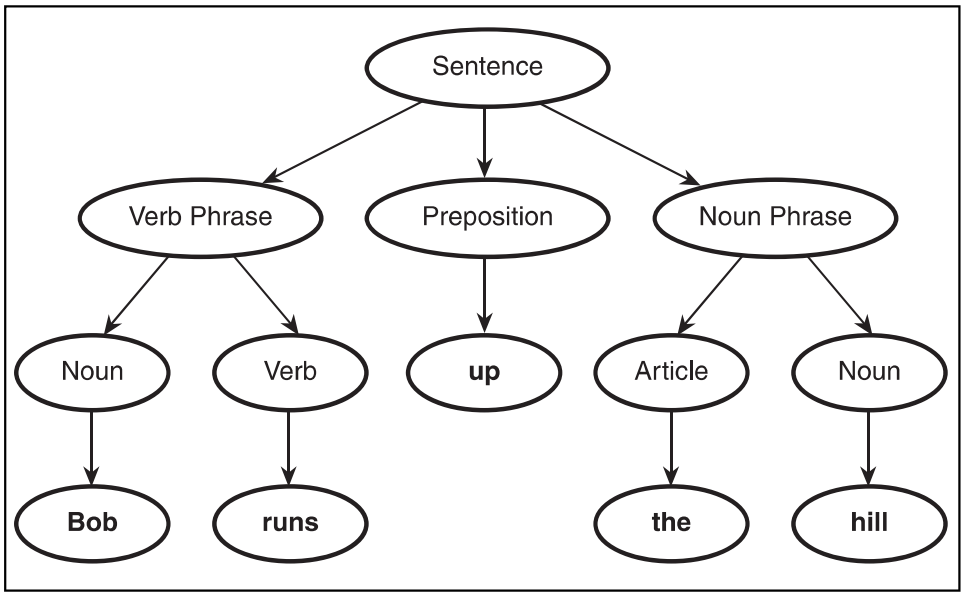

Figure 12.12

This is a parse tree for an English sentence.

Now, don't be put off if you didn't understand that; this is a complex topic in English, after all. I showed that to you so that you can begin to understand how computers parse the code that you send into your C++ compiler.

"Okay," you say, "parsing is important when you're making compilers, but what the heck does it have to do with game programming?"

I'm sure you've played *Quake* before. If you have made custom maps for *Quake*, you know that *Quake* has a *scripting system* known as *QuakeC*. This system allows you to add little bits of C code to *Quake* maps so that code is executed when the player or monsters do something on the map.

A scripting system essentially allows you to make *very* customizable maps for a game. I'm sure you've played some of the *Quake modules* (*mods*) before. One of my favorites is *Team Fortress Classic* (*TFC*). These mods allow you to drastically change the way the game operates, expanding upon the original game's capabilities.

One of the reasons games like *Quake* are so popular is because they are so modifiable.

This section introduces you to basic arithmetic parsing, which is the first step toward creating your very own scripting system.

Arithmetic Expressions

Don't be confused by that big name; arithmetic expressions are really just mathematical formulas involving numbers and variables. $x = 24 + y$ is an arithmetic expression.

The standard four operators in math are addition, subtraction, multiplication, and division. All four of these operators are *binary* operators, which means that they operate on two numbers.

Parsing an Arithmetic Expression

Look at this expression for a moment: 2 * (*y* / *z*). There are two operators in this expression: multiplication and division. Each operator has a *term* on the *left* and the *right* sides of itself. Does that remind you of anything—possibly something in this chapter? That's right—binary tree nodes have left and right children!

So you can treat the operator as a node and put the terms into the left and right nodes of a binary tree. For example, the term inside the parentheses can be viewed like the first tree in Figure 12.13. Then, if you create a node with the multiplication symbol in it and put 2 as the left child node and the subtree created inside the parentheses as the right child node, you get the second tree in Figure 12.13.

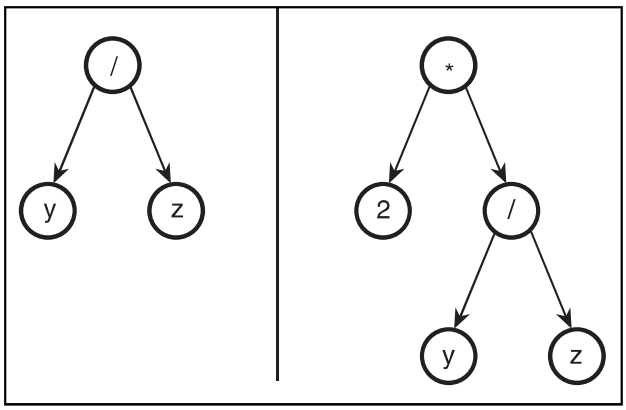

Figure 12.13

*This is the parse tree for the arithmetic expression 2 * (y / z).*

Well, now you've got a tree; what do you do with it? You can perform a postorder traversal on the tree to calculate its value!

For example, you start at the root node and tell it to return the value of the left node first. The left node just returns 2. Then, you tell the right node to return its value. Because the right node is another operator, the postorder algorithm is called again. The division node asks its left node for its value, which is *y*, and then asks the right node for its value, which is *z*. Now that both child nodes have returned their values, the division node can divide *y* by *z* and return the result back up to the multiplication node. Now that the multiplication node has the values of both of its children, it multiplies both of them together and returns that result! Whoa, that's cool.

Recursive Descent Parsing

I'm going to show you an amazingly simple demonstration of what is called *recursive descent* parsing, which you can use to parse a simple arithmetic expression and turn it into a tree that your program can then use as a simple script.

Tokens

The first thing you need to do is turn the actual arithmetic expression into a list of *tokens*. A token is basically a structure that says, "This is a number," "This is an operator," or "This is a variable."

I'll first create an enumerated type, which will help you determine the type of a token:

```
enum TOKEN
{
    NUMBER,
    VARIABLE,
    OPERATOR,
    LPAREN,
    RPAREN
};
```

After that, I create the actual `Token` class:

```
class Token
{
    TOKEN m_type;
    float m_number;
```

```
    int m_variable;
     int m_operator;
};
```

This class has a type variable that determines which of the following three variables is valid.

> **NOTE**
>
> **More-complex implementations of a token class would use the C++ `union` directive and have a different class structure for each kind of token type. If you don't know what a `union` is, don't worry; I'm not using them in this demo because this demo is simple.**

If the type of the token is `NUMBER`, then `m_number` will hold the number. If the type of the token is `VARIABLE`, then `m_variable` will hold the number of the variable (you'll see how this works in a bit). If the token is `OPERATOR`, then `m_operator` has a number from 0–3, where 0 is addition, 1 is subtraction, 2 is multiplication, and 3 is division.

Variables

This very simple demo only has four variables for now, so the only valid values of `m_value` are 0–3. More-complex systems might have more variables than this. The most complex systems don't use this method at all; instead, they store information about whether the variable is global or local and the memory offset and datatype of the variable. It gets very complex.

For this system, the only valid variables are `c`, `s`, `t`, and `l`, which stand for `cosine`, `sine`, `time`, and `life`. The `cosine` and the `sine` variables keep track of the cosine and sine of the current game time. The `time` variable keeps track of the current time of the system, and the `life` variable keeps track of the amount of life that the player has left.

Scanning

The process of converting the text string into a stream of tokens is called *scanning*, or *tokenizing*. The scanner will read each part of an expression into a string and then determine if it is an operator, variable, number, or parenthesis.

The code for this process isn't very complex, but it is long, bulky, and boring.

The scanning process for a simple system works like this:

1. Read in a character.
2. If the character is one of the four variables, create a variable token.
3. If the character is one of the four operators, create an operator token.

4. If the character is a number, read in the rest of the number and create a number token.
5. Place the token into a queue.
6. Repeat.

You can find the code in the g12-01.cpp file on the CD if you're really interested (the `Scan` function); I have decided not to include it here because it doesn't have anything to do with trees. The scanner just provides an easy way of turning a string of characters into a queue of items that the parser recognizes.

Parsing

There are basically two different forms for an arithmetic expression term:

1. It can be a single constant or variable.
2. It can be two constants or variables with an operator in between.

I established previously that the operators in this demo are all binary; they operate on two numbers. In languages like C++, you can chain operators together, like this:

```
c + s + t
```

For simplicity, the parser doesn't support statements like that. Instead, parentheses must surround two of the variables. Either of these corrections is acceptable:

```
c + (s + t)
(c + s) + t
```

So the parser's job is to view the queue of tokens and turn it into a binary tree. The parser is a recursive function, which makes your life much easier.

I'm going to show you the pseudocode algorithm in a few sections so you can understand what is going on.

The parse algorithm takes a queue of tokens and returns a tree. The algorithm also creates three tree nodes as local variables:

```
Tree Parse( Queue )
    Tree left, center, right
```

Now, the first thing to do is to check the first token.

```
    if Queue.First == LPAREN
        Queue.Dequeue
        left = Parse( Queue )
```

```
    Queue.Dequeue
else if Queue.First == VARIABLE or NUMBER
     left = VARIABLE or NUMBER
Queue.Dequeue
```

There are three valid token types for the first token of the queue. If the first token is a left parenthesis, then the parenthesis is taken off the queue and the rest of the queue is passed into the parse algorithm again. The result of the recursively called parse algorithm is placed into the left tree node. Theoretically, the parse algorithm should have removed everything after the first left parenthesis up to a matching right parenthesis, so there should be a right parenthesis at the front of the queue. That is also removed from the queue.

CAUTION

Real parsers would check to see if the queue actually contained a right parenthesis after the parse algorithm returns. If it isn't a right parenthesis, the string that is being parsed is illegal. For the purposes of the demo, I left error checking out, but you should be aware that a clean system would use error checking. I recommend using exceptions if you know how to use them.

If the first token was a variable or a constant number instead, then the left tree node is made into a leaf node that contains information about the variable or constant.

Finally, the token is removed from the queue. After the first token is processed, the algorithm decides if the term is just a single variable or number or if it is two variables or numbers separated by an operator.

If the current term is just a single variable or number, then that token has already been processed and the queue will either be empty or have a right parenthesis at the front.

```
if  Queue.Empty or Queue.Front == RPAREN
    return left
```

The function returns the left node at this point because it contains the single term.

If it isn't a single term, then the queue must contain an operator:

```
if Queue.Front == OPERATOR
    center = OPERATOR
Queue.Dequeue
```

If the queue doesn't contain an operator at the front, then the string is invalid, and the parser should handle the error by informing the user. For simplicity, this demo doesn't have that kind of error checking.

Now that you've gotten to this point, there is only one more token to process for the term. Like the first token, the only valid types it can be are variables, numbers, or left parentheses:

```
if Queue.First == LPAREN
    Queue.Dequeue
    right  = Parse( Queue )
    Queue.Dequeue
 else if Queue.First == VARIABLE or NUMBER
     right = VARIABLE or NUMBER
 Queue.Dequeue
```

And finally, attach the left and right children to the center and return it:

```
center.left = left
center.right = right
return center
```

If you can think recursively, this algorithm will appear amazingly simple for the task it does. If you don't quite understand recursion yet, I'll show you a few examples on how this algorithm works.

Using the Algorithm

First, I'll start off with the simplest example:

```
t
```

This is a single-variable term. Naturally, you should expect the parser to return a tree with one node: `t` at the root. The algorithm looks at the token, sees that it is a variable, and then sets the left node so that it is a variable node.

Now the function checks the queue and sees that it is empty, so it returns the left node, giving us a simple one-node tree with `t` in it.

Now I'll move on to a more complicated example:

```
t + ( 5 * c )
```

The first step is the same; the left node is turned into a variable node. The second step is different, however. Last time, the queue was empty; this time, an operator token is in it.

So now the algorithm creates the center node and turns it into a +.

Now it looks at the next token, which is a left parenthesis. So it strips off the parenthesis and passes the queue (which contains `5 * c )` now) into the parse algorithm again.

This time, the second parse algorithm strips off the 5 and makes the left node a constant number node. It strips off the star and turns the center node into a multiplication operator node. Finally, it strips off the c and turns the right node into a constant node. The second parse algorithm then returns the center node up to the first parse algorithm.

Now the result of the second parse algorithm is placed in the right node and the first center node is returned, resulting in the tree in Figure 12.14.

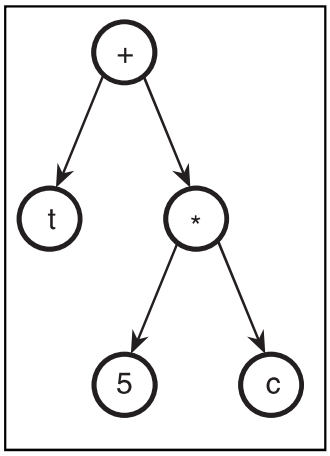

Figure 12.14

The parse tree for a simple expression.

Now you can see how recursion is your friend here: It takes care of those nasty nested parentheses automatically so you don't have to mess around with them much.

Source Listing

Here is the source code listing for the `ParseArithmetic` function used in the demo. Pay attention to where the comments are; they alert you as to where proper error checking should be inserted.

```
BinaryTree<Token>* ParseArithmetic( LQueue<Token>& p_queue )
{
    BinaryTree<Token>* left = 0;
    BinaryTree<Token>* center = 0;
    BinaryTree<Token>* right = 0;
    // make sure the queue has something in it.
    if( p_queue.Count() == 0 )
        return 0;
    // take off the first token and determine what it is
    switch( p_queue.Front().m_type )
    {
    case LPAREN:
```

```
        p_queue.Dequeue();
        left = ParseArithmetic( p_queue );
        // if( p_queue.Front().m_type != RPAREN )
            // this is where you would throw an error;
            // the string is unparsable with our language.
        p_queue.Dequeue();
        break;
    case VARIABLE:
    case NUMBER:
        left = new BinaryTree<Token>;
        left->m_data = p_queue.Front();
        p_queue.Dequeue();
        break;
    // case OPERATOR:
        // this is where you would throw an error;
        // the string is unparsable with our language.
    }
    if( p_queue.Count() == 0 )
        return left;
    if( p_queue.Front().m_type == RPAREN )
        return left;
    // if( p_queue.Front().m_type != OPERATOR )
        // this is where you would throw an error;
        // the string is unparsable with our language.
    center = new BinaryTree<Token>;
    center->m_data = p_queue.Front();
    p_queue.Dequeue();
    // make sure the queue has something in it.
    if( p_queue.Count() == 0 )
        return 0;
    // take off the third token and determine what it is
    switch( p_queue.Front().m_type )
    {
    case LPAREN:
        p_queue.Dequeue();
        right = ParseArithmetic( p_queue );
        // if( p_queue.Front().m_type != RPAREN )
            // this is where you would throw an error;
            // the string is unparsable with our language.
        p_queue.Dequeue();
        break;
```

```
    case VARIABLE:
    case NUMBER:
        right = new BinaryTree<Token>;
        right->m_data = p_queue.Front();
        p_queue.Dequeue();
        break;
    // case OPERATOR:
        // this is where you would throw an error;
        // the string is unparsable with our language.
    }
    center->m_left = left;
    center->m_right = right;
    return center;
}
```

You can probably see why I didn't just paste the code right away; pseudo-code is almost always easier to understand.

Executing the Tree

Now that the parser has built the parse tree, you need to be able to evaluate it somehow. I mentioned before that you can use a simple postorder traversal to evaluate the tree, which is what I will show you now.

The `Evaluate` function is also (take a guess!) recursive! Gee, that was surprising, wasn't it? I hope you're beginning to see a trend when using trees. Recursion really makes some things easy.

The function will evaluate a tree node, returning a float value. There are three types of nodes, so I'll split the code up into five parts: the beginning, the three node types, and the end.

Here is the beginning:

```
float Evaluate( BinaryTree<Token>* p_tree )
{
    if( p_tree == 0 )
        return 0.0f;

    float left = 0.0f;
    float right = 0.0f;
```

This sets everything up first. If the node passed into the algorithm is 0, then 0 is returned. If not, then the left and right variables are set to 0.

Now, the algorithm uses a switch statement to determine which of the three node types it is:

```
switch( p_tree->m_data.m_type )
{
case VARIABLE:
     return g_vars[p_tree->m_data.m_variable];
     break;
```

The first node type is a variable. Because the demo has four valid variables, all four variables are stored in an array, `g_vars`. The `m_variable` member of the `Token` class will contain a number from 0 to 3, so the function gets that number and returns the correct value from the variable table.

```
case NUMBER:
     return p_tree->m_data.m_number;
     break;
```

The second node type is a constant number. This case is easy; it just returns the number stored within the token.

```
case OPERATOR:
     left = Evaluate( p_tree->m_left );
     right = Evaluate( p_tree->m_right );
     switch( p_tree->m_data.m_operator )
     {
     case 0:
          return left + right;
          break;
     case 1:
          return left - right;
          break;
     case 2:
          return left * right;
          break;
     case 3:
          return left / right;
          break;
     }
}
```

The third node type is the most interesting: the operator. If the node is an operator, then it recursively calls the `Evaluate` function on its left and right children, determines which operation to execute on the two values, and returns the result.

```
    return 0.0f;
}
```

Last, in case something messed up, 0 is returned at the end. Hopefully nothing did, but it is always safe to do so anyway.

Playing the Demo

This is the most complex demo in the book so far, so it needs a fair amount of explanation. Figure 12.15 shows a screenshot from the demo in action.

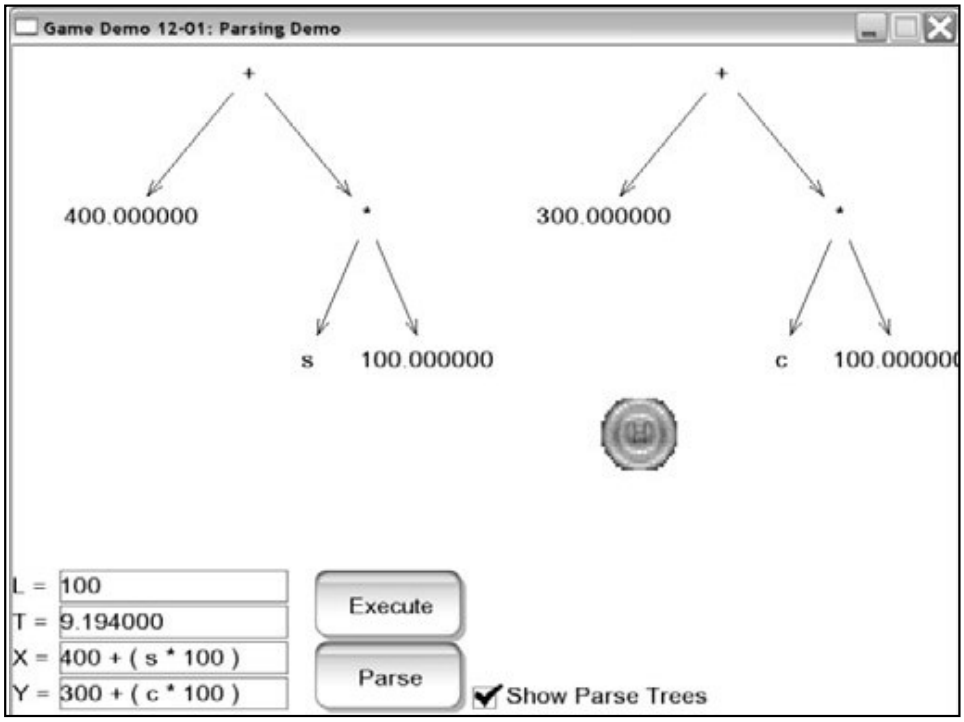

Figure 12.15

Here is a screenshot from the demo.

At the bottom are four text boxes. They represent the life of the player, the current time, and the *x* and *y* formulas for the player. You'll be using the bottom boxes to control the position of the player on-screen.

To start off, try entering these two lines into the *x* and *y* boxes:

```
t * 100
0
```

Now check the check box on the right of the screen so that it will display the parse trees. After that, click the Parse button; you should see two trees drawn on the

screen now. The *x* tree is on the left and the *y* tree is on the right. This way, you can visually see how your expression was parsed by the system.

Next, you want to set up your life variable. You can click on the *L* box and enter a life value.

You cannot modify the *T* value, though. The Execute button is a toggle that resets the time to 0 when you click it and then starts the demonstration.

Now that you've entered your formula, click the Execute button. A UFO should appear on the screen at the upper left, and it should move to the right at 100 pixels per second. It will take 8 seconds to travel off the screen, and you need to reset it when it's done. Clicking the Execute button again will stop the demo from running.

I urge you to play around with different formulas to see what you can accomplish. Table 12.2 holds some of the cool ones that I've discovered.

NOTE

A lot of the formulas in Table 12.2 use the `c` and `s` variables, which are the sine and cosine of the time. If you know *trigonometry*, then the effect of these variables should be obvious to you. This book doesn't teach trigonometry, but trig isn't a requirement for the book, so the best I can do is tell you to sit back and enjoy the pretty effects that they produce. If you don't know trigonometry, though, you're missing out on a lot. Trig is one of the most important math subjects you can use when programming games.

Table 12.2 Cool Formulas

x	y	Effect
400 + (c * 100)	300 + (s * 100)	Makes the ship fly around in circles
t * 100	300 + (s * 100)	Makes the ship fly in a sine wave pattern
400 + (c * 200)	300	Makes the ship fly back and forth rapidly
(t * t) * 10	300	Makes the ship slowly accelerate off the screen
400 + (c * (t * 10))	300 + (s * (t * 10))	Makes the ship slowly circle out of control

I made these formulas after playing around for a minute; I'm sure you can come up with some even neater ones. For example, you could make the speed of the spaceship depend on the amount of health you have left. The possibilities are endless.

Conclusion

This chapter turned out to be a lot longer than I expected, mainly due to the extensive parsing section I included. I hope you understood it, because parsing is a very neat area of game developing. Nothing beats a game that is 100 percent extendible and modifiable.

If anything, this chapter should have reinforced the idea that recursion is a very important area of programming. Some people may say that recursion is too slow for game programming, and they are sometimes right. The key is knowing when recursion is used *best*.

Binary trees aren't very exciting on their own, but I included them here to lead up to the next few chapters. BSTs (see Chapter 13), heaps (see Chapter 14), and Huffman trees (see Chapter 21, "Data Compression") all use binary trees as their base. In addition, a lot of trees that aren't covered in this book are based on binary trees, such as AVL trees and red-black trees, as I mentioned before.