1

TOKENIZATION AND PARSING

How many times do you think Ruby reads and transforms your code before running it? Once? Twice?

The correct answer is three times. Whenever you run a Ruby script—whether it's a large Rails application, a simple Sinatra website, or a background worker job—Ruby rips your code apart into small pieces and then puts them back together in a different format *three times*! Between the time you type *ruby* and the time you start to see actual output on the console, your Ruby code has a long road to take—a journey involving a variety of different technologies, techniques, and open source tools.

Figure 1-1 shows what this journey looks like at a high level.

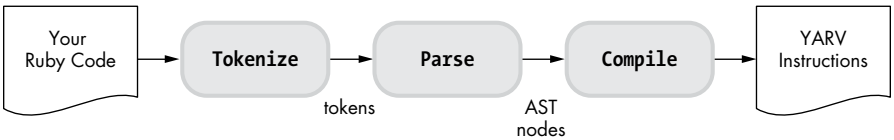

Figure 1-1: Your code's journey through Ruby

First, Ruby *tokenizes* your code, which means it reads the text characters in your code file and converts them into *tokens*, the words used in the Ruby

language. Next, Ruby *parses* these tokens; that is, it groups the tokens into meaningful Ruby statements just as one might group words into sentences. Finally, Ruby compiles these statements into low-level instructions that it can execute later using a virtual machine.

I'll cover Ruby's virtual machine, called "Yet Another Ruby Virtual Machine" (YARV), in Chapter 3. But first, in this chapter, I'll describe the tokenizing and parsing processes that Ruby uses to understand your code. After that, in Chapter 2, I'll show you how Ruby compiles your code by translating it into a completely different language.

NOTE *Throughout most of this book we'll learn about the original, standard implementation of Ruby, known as* Matz's Ruby Interpreter (MRI) *after Yukihiro Matsumoto, who invented Ruby in 1993. There are many other implementations of Ruby available in addition to MRI, including Ruby Enterprise Edition, MagLev, MacRuby, RubyMotion, mruby, and many, many others. Later, in Chapters 10, 11, and 12, we'll look at two of these alternative Ruby implementations: JRuby and Rubinius.*

ROADMAP

Tokens: The Words That Make Up the Ruby Language

Suppose you write a simple Ruby program and save it in a file called *simple.rb*, shown in Listing 1-1.

```
10.times do |n|
  puts n
end
```

Listing 1-1: A very simple Ruby program (simple.rb)

Listing 1-2 shows the output you would see after executing the program from the command line.

```
$ ruby simple.rb
0
1
2
3
--snip--
```

Listing 1-2: Executing Listing 1-1

What happens after you type `ruby simple.rb` and press ENTER? Aside from general initialization, processing your command line parameters, and so on, the first thing Ruby does is open *simple.rb* and read in all the text from the code file. Next, it needs to make sense of this text: your Ruby code. How does it do this?

After reading in *simple.rb*, Ruby encounters the series of text characters shown in Figure 1-2. (To keep things simple, I'm showing only the first line of text here.)

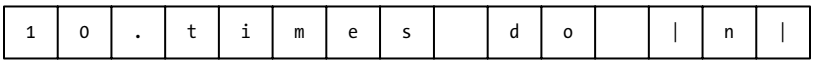

Figure 1-2: The first line of text in simple.rb

When Ruby sees these characters, it tokenizes them. That is, it converts them into a series of tokens or words that it understands by stepping through the characters one at a time. In Figure 1-3, Ruby starts scanning at the first character's position.

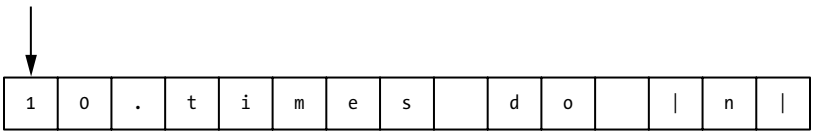

Figure 1-3: Ruby starts to tokenize your code.

The Ruby C source code contains a loop that reads in one character at a time and processes it based on what that character is.

To keep things simple, I'm describing tokenization as an independent process. In fact, the parsing engine I describe next calls this C tokenize code whenever it needs a new token. Tokenization and parsing are separate processes that actually occur at the same time. For now, let's just continue to see how Ruby tokenizes the characters in your Ruby file.

Ruby realizes that the character 1 is the start of a number and continues to iterate over the characters that follow until it finds a nonnumeric character. First, in Figure 1-4, it finds a 0.

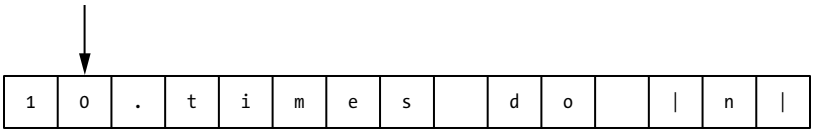

Figure 1-4: Ruby steps to the second text character.

And stepping forward again, in Figure 1-5, Ruby finds a period character.

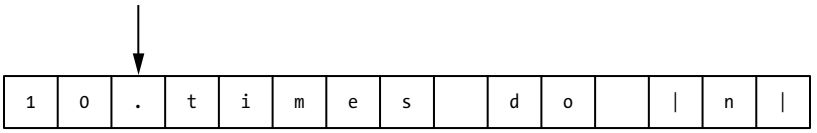

Figure 1-5: Ruby finds a period character.

Ruby actually considers the period character to be numeric because it might be part of a floating-point value. In Figure 1-6, Ruby steps to the next character, t.

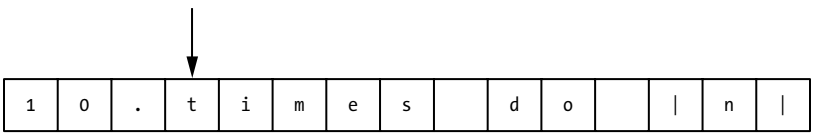

Figure 1-6: Ruby finds the first nonnumeric character.

Now Ruby stops iterating because it has found a nonnumeric character. Because there are no more numeric characters after the period, Ruby considers the period to be part of a separate token, and it steps back one, as shown in Figure 1-7.

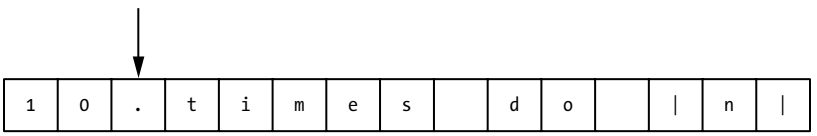

Figure 1-7: Ruby steps back one character.

Finally, in Figure 1-8, Ruby converts the numeric characters that it found into the first token from your program, called `tINTEGER`.

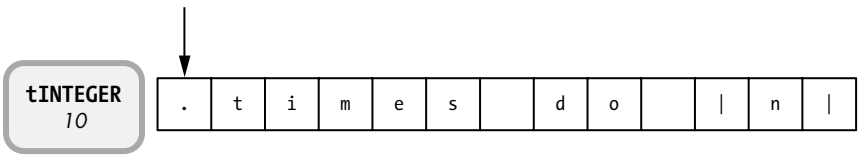

Figure 1-8: Ruby converts the first two text characters into a `tINTEGER` token.

Ruby continues to step through the characters in your code file, converting them into tokens and grouping characters as necessary. The second token, shown in Figure 1-9, is a single character: a period.

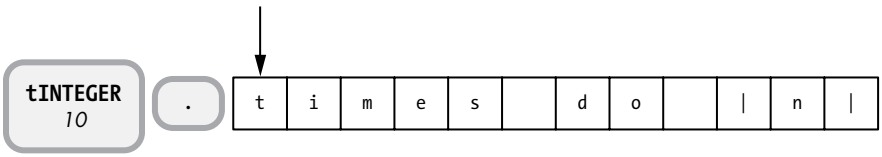

Figure 1-9: Ruby converts the period character into a token.

Next, in Figure 1-10, Ruby encounters the word *times* and creates an identifier token.

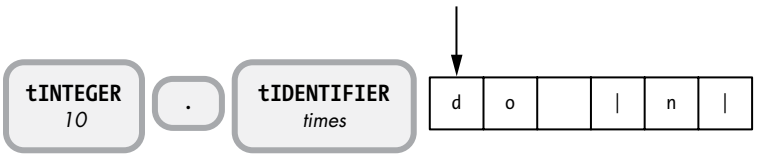

Figure 1-10: Ruby tokenizes the word times.

Identifiers are words in your Ruby code that are not reserved words. Identifiers usually refer to variable, method, or class names.

Next, Ruby sees *do* and creates a reserved word token, as indicated by `keyword_do` in Figure 1-11.

Figure 1-11: Ruby creates a reserved word token: `keyword_do`.

Reserved words are keywords that carry significant meaning in Ruby because they provide the structure, or framework, of the language. They are called *reserved words* because you can't use them as normal identifiers, although you can use them as method names, global variable names (such as `$do`), or instance variable names (for example, `@do` or `@@do`).

Internally, the Ruby C code maintains a constant table of reserved words. Listing 1-3 shows the first few, in alphabetical order.

```
alias
and
begin
break
case
class
```

Listing 1-3: The first few reserved words, listed alphabetically

THE PARSER_YYLEX FUNCTION

If you're familiar with C and are interested in learning more about the detailed way in which Ruby tokenizes your code file, see the *parse.y* file in your version of Ruby. The *.y* extension indicates that *parse.y* is a *grammar rule file*—one that contains a series of rules for the Ruby parser engine. (I'll discuss these in the next section.) *parse.y* is an extremely large and complex file with over 10,000 lines of code!

For now, ignore the grammar rules, and search for a C function called `parser_yylex`, about two-thirds of the way down the file, around line 6500. This complex C function contains the code that actually tokenizes your code. Look closely and you should see a very large `switch` statement that starts with the code shown in Listing 1-4.

```
❶ retry:
❷ last_state = lex_state;
❸ switch (c = nextc()) {
```

Listing 1-4: The C code inside Ruby that reads in each character from your code file

The `nextc()` function ❸ returns the next character in the code file text stream. Think of this function as the arrow in the previous diagrams. The `lex_state` variable ❷ keeps information about what state or type of code Ruby is processing at the moment.

The large `switch` statement inspects each character of your code file and takes a different action based on what it is. For example, the code shown in Listing 1-5 looks for whitespace characters and ignores them by jumping back up to the `retry` label ❶ just above the `switch` statement in Listing 1-4.

```
  /* white spaces */
case ' ': case '\t': case '\f': case '\r':
case '\13': /* '\v' */
  space_seen = 1;
--snip--
  goto retry;
```

Listing 1-5: This C code checks for whitespace characters in your code and ignores them.

Ruby's reserved words are defined in the file called *defs/keywords*. If you open this file, you'll see a complete list of all of Ruby's reserved words (see a partial list in Listing 1-3). The *keywords* file is used by an open source package called *gperf* to produce C code that can quickly and efficiently look up strings in a table—in this case, a table of reserved words. You can find the generated C code that looks up reserved words in *lex.c*, which defines a function named `rb_reserved_word`, called from *parse.y*.

One final detail about tokenization: Ruby doesn't use the Lex tokenization tool that C programmers commonly use in conjunction with a parser generator like Yacc or Bison. Instead, the Ruby core team wrote the Ruby tokenization code by hand for performance reasons.

Finally, as shown in Figure 1-12, Ruby converts the remaining characters to tokens.

Figure 1-12: Ruby finishes tokenizing the first line of text.

Ruby continues to step through your code until it has tokenized the entire Ruby script. At this point, it has processed your code for the first time, ripping it apart and putting it back together again in a completely different way. Your code began as a stream of text characters, and Ruby converted it to a stream of tokens, words that it will later combine into sentences.

Experiment 1-1: Using Ripper to Tokenize Different Ruby Scripts

Now that we've learned the basic idea behind tokenization, let's look at how Ruby actually tokenizes different Ruby scripts. After all, how else will you know that the previous explanation is actually correct?

As it turns out, a tool called *Ripper* makes it very easy to see what tokens Ruby creates for different code files. Shipped with Ruby 1.9 and Ruby 2.x, the `Ripper` class allows you to call the same tokenization and parsing code that Ruby uses to process text from code files. (Ripper is not available in Ruby 1.8.)

Listing 1-6 shows how simple using Ripper is.

```
require 'ripper'
require 'pp'
code = <<STR
10.times do |n|
  puts n
end
STR
puts code
❶ pp Ripper.lex(code)
```

Listing 1-6: An example of how to call `Ripper.lex` (lex1.rb)

After requiring the Ripper code from the standard library, you call it by passing some code as a string to the `Ripper.lex` method ❶. Listing 1-7 shows the output from Ripper.

```
$ ruby lex1.rb
10.times do |n|
  puts n
end
❶ [[[1, 0], :on_int, "10"],
```

```
 [[1, 2], :on_period, "."],
❷ [[1, 3], :on_ident, "times"],
 [[1, 8], :on_sp, " "],
 [[1, 9], :on_kw, "do"],
 [[1, 11], :on_sp, " "],
 [[1, 12], :on_op, "|"],
 [[1, 13], :on_ident, "n"],
 [[1, 14], :on_op, "|"],
 [[1, 15], :on_ignored_nl, "\n"],
 [[2, 0], :on_sp, "  "],
 [[2, 2], :on_ident, "puts"],
 [[2, 6], :on_sp, " "],
 [[2, 7], :on_ident, "n"],
 [[2, 8], :on_nl, "\n"],
 [[3, 0], :on_kw, "end"],
 [[3, 3], :on_nl, "\n"]]
```

Listing 1-7: The output generated by `Ripper.lex`

Each line corresponds to a single token that Ruby found in your code string. On the left, we have the line number (`1`, `2`, or `3` in this short example) and the text column number. Next, we see the token itself displayed as a symbol, such as `:on_int` ❶ or `:on_ident` ❷. Finally, Ripper displays the text characters that correspond to each token.

The token symbols that Ripper displays are somewhat different from the token identifiers I used in Figures 1-2 through 1-12 that showed Ruby tokenizing the `10.times do` code. I used the same names you would find in Ruby's internal parse code, such as `tIDENTIFIER`, while Ripper used `:on_ident` instead.

Regardless, Ripper will still give you a sense of what tokens Ruby finds in your code and how tokenization works.

Listing 1-8 shows another example of using Ripper.

```
$ ruby lex2.rb
10.times do |n|
  puts n/4+6
end
--snip--
 [[2, 2], :on_ident, "puts"],
 [[2, 6], :on_sp, " "],
 [[2, 7], :on_ident, "n"],
 [[2, 8], :on_op, "/"],
 [[2, 9], :on_int, "4"],
 [[2, 10], :on_op, "+"],
 [[2, 11], :on_int, "6"],
 [[2, 12], :on_nl, "\n"],
--snip--
```

Listing 1-8: Another example of using `Ripper.lex`

This time Ruby converts the expression `n/4+6` into a series of tokens in a very straightforward way. The tokens appear in exactly the same order they did inside the code file.

Listing 1-9 shows a third, slightly more complex example.

```
$ ruby lex3.rb
array = []
10.times do |n|
  array << n if n < 5
end
p array
--snip--
 [[3, 2], :on_ident, "array"],
 [[3, 7], :on_sp, " "],
❶ [[3, 8], :on_op, "<<"],
 [[3, 10], :on_sp, " "],
 [[3, 11], :on_ident, "n"],
 [[3, 12], :on_sp, " "],
 [[3, 13], :on_kw, "if"],
 [[3, 15], :on_sp, " "],
 [[3, 16], :on_ident, "n"],
 [[3, 17], :on_sp, " "],
❷ [[3, 18], :on_op, "<"],
 [[3, 19], :on_sp, " "],
 [[3, 20], :on_int, "5"],
--snip--
```

Listing 1-9: A third example of running `Ripper.lex`

As you can see, Ruby is smart enough to distinguish between `<<` and `<` in the following line: `array << n if n < 5`. The characters `<<` are converted to a single operator token ❶, while the single `<` character that appears later is converted into a simple less-than operator ❷. Ruby's tokenize code is smart enough to look ahead for a second `<` character when it finds one `<`.

Finally, notice that Ripper has no idea whether the code you give it is valid Ruby or not. If you pass in code that contains a syntax error, Ripper.lex will just tokenize it as usual and not complain. It's the parser's job to check syntax.

Suppose you forget the `|` symbol after the block parameter `n` ❶, as shown in Listing 1-10.

```
require 'ripper'
require 'pp'
code = <<STR
❶ 10.times do |n
  puts n
end
STR
puts code
pp Ripper.lex(code)
```

Listing 1-10: This code contains a syntax error.

Running this, you get the output shown in Listing 1-11.

```
$ ruby lex4.rb
10.times do |n
  puts n
end
--snip--
[[[1, 0], :on_int, "10"],
 [[1, 2], :on_period, "."],
 [[1, 3], :on_ident, "times"],
 [[1, 8], :on_sp, " "],
 [[1, 9], :on_kw, "do"],
 [[1, 11], :on_sp, " "],
 [[1, 12], :on_op, "|"],
 [[1, 13], :on_ident, "n"],
 [[1, 14], :on_nl, "\n"],
--snip--
```

Listing 1-11: Ripper does not detect syntax errors.

Parsing: How Ruby Understands Your Code

Once Ruby converts your code into a series of tokens, what does it do next? How does it actually understand and run your program? Does Ruby simply step through the tokens and execute each one in order?

No. Your code still has a long way to go before Ruby can run it. The next step on its journey through Ruby is called *parsing*, where words or tokens are grouped into sentences or phrases that make sense to Ruby. When parsing, Ruby takes into account the order of operations, methods, blocks, and other larger code structures.

But how can Ruby actually understand what you're telling it with your code? Like many programming languages, Ruby uses a *parser generator*. Ruby uses a parser to process tokens, but the parser itself is generated with a parser generator. Parser generators take a series of grammar rules as input that describe the expected order and patterns in which the tokens will appear.

Ruby uses an LALR parser generator called Bison.

The most widely used and well-known parser generator is Yacc (Yet Another Compiler Compiler), but Ruby uses a newer version of Yacc called *Bison*. The grammar rule file for Bison and Yacc has a *.y* extension. In the Ruby source code, the grammar rule file is *parse.y* (introduced earlier). The *parse.y* file defines the actual syntax and grammar that you have to use while writing your Ruby code; it's really the heart and soul of Ruby and where the language itself is actually defined!

Ruby uses Bison when building Ruby itself, and not to directly process tokens. In effect, there are two separate steps to the parsing process, shown in Figure 1-13.

Before you run your Ruby program, the Ruby build process uses Bison to generate the parser code (*parse.c*) from the grammar rule file (*parse.y*). Later, at run time, this generated parser code parses the tokens returned by Ruby's tokenizer code.

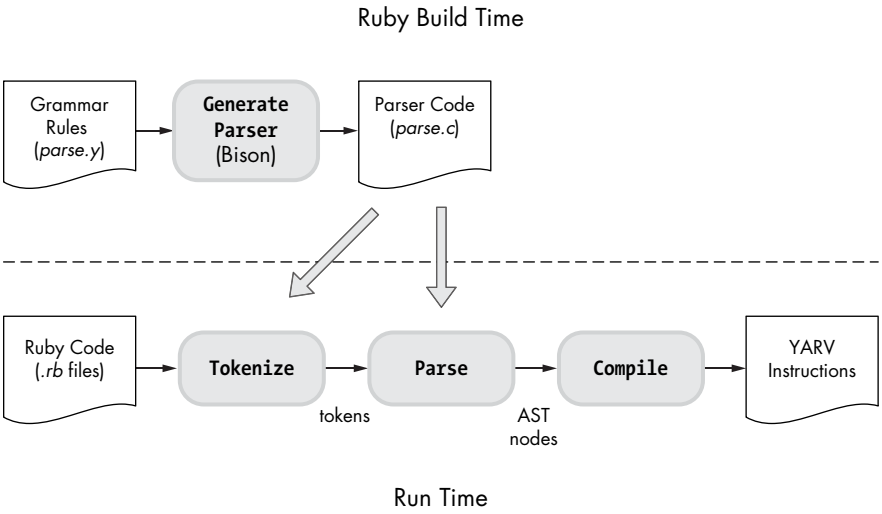

Figure 1-13: The Ruby build process runs Bison ahead of time.

Because the *parse.y* file and the generated *parse.c* file also contain the tokenization code, Figure 1-13 has a diagonal arrow from *parse.c* to the tokenize process on the lower left. (In fact, the parse engine I'm about to describe calls the tokenization code whenever it needs a new token.) The tokenization and parsing processes actually occur simultaneously.

Understanding the LALR Parse Algorithm

How does the parser code analyze and process the incoming tokens? With an algorithm known as *LALR*, or *Look-Ahead Left Reversed Rightmost Derivation*. Using the LALR algorithm, the parser code processes the token stream from left to right, trying to match their order and the pattern in which they appear against one or more of the grammar rules from *parse.y*. The parser code also "looks ahead" when necessary to decide which grammar rule to match.

The best way to become familiar with the way Ruby grammar rules work is with an example. To keep things simple for now, we'll look at an abstract example. Later on, I'll show that Ruby actually works in precisely the same way when it parses your code.

Suppose you want to translate from the Spanish:

Me gusta el Ruby. [Phrase 1]

to the English:

I like Ruby.

And suppose that to translate Phrase 1, you use Bison to generate a C language parser from a grammar file. Using the Bison/Yacc grammar rule syntax, you can write the simple grammar shown in Listing 1-12, with the rule name on the left and the matching tokens on the right.

```
SpanishPhrase : me gusta el ruby {
  printf("I like Ruby\n");
}
```

Listing 1-12: A simple grammar rule matching the Spanish Phrase 1

This grammar rule says the following: If the token stream is equal to `me`, `gusta`, `el`, and `ruby`—in that order—we have a match. If there's a match, the Bison generated parser will run the given C code, and the `printf` statement (similar to `puts` in Ruby) will print the translated English phrase.

Figure 1-14 shows the parsing process in action.

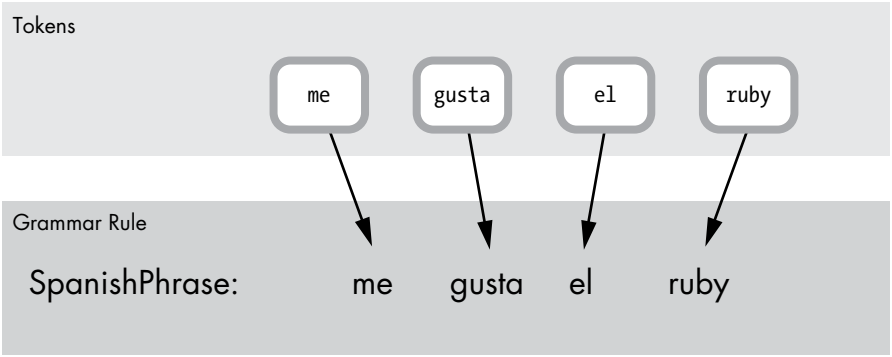

Figure 1-14: Matching tokens with a grammar rule

There are four input tokens at the top, and the grammar rule is underneath. It should be clear that there's a match because each input token corresponds directly to one of the terms on the right side of the grammar rule. We have a match on the `SpanishPhrase` rule.

Now let's improve on this example. Suppose you need to enhance your parser to match Phrase 1 and Phrase 2:

Me gusta el Ruby. [Phrase 1]

and:

Le gusta el Ruby. [Phrase 2]

In English, Phrase 2 means "She/He/It likes Ruby."

The modified grammar file in Listing 1-13 can parse both Spanish phrases.

```
SpanishPhrase: VerbAndObject el ruby {
  printf("%s Ruby\n", $1);
};
VerbAndObject: SheLikes | ILike {
  $$ = $1;
};
SheLikes: le gusta {
  $$ = "She likes";
}
ILike: me gusta {
  $$ = "I like";
}
```

Listing 1-13: These grammar rules match both Phrase 1 and Phrase 2.

As you can see, there are four grammar rules here instead of just one. Also, you're using the Bison directive `$$` to return a value from a child grammar rule to a parent and `$1` to refer to a child's value from a parent.

Unlike with Phrase 1, the parser can't immediately match Phrase 2 with any of the grammar rules.

In Figure 1-15, we can see the `el` and `ruby` tokens match the `SpanishPhrase` rule, but `le` and `gusta` do not. (Ultimately, we'll see that the child rule `VerbAndObject` does match `le gusta`, but never mind that for now.) With four grammar rules, how does the parser know which other rules to try to match against? And against which tokens?

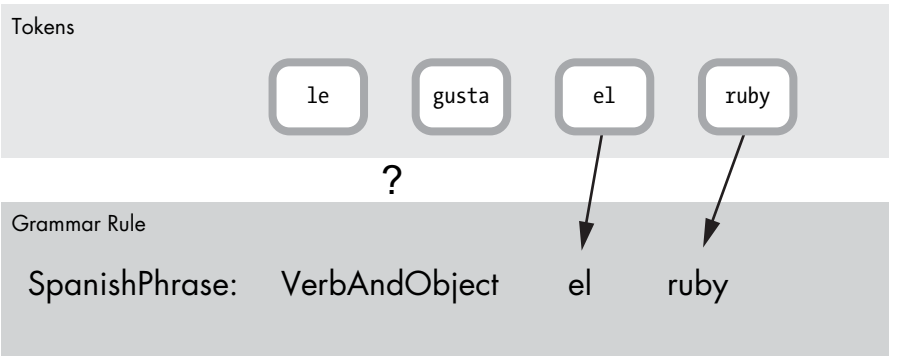

Figure 1-15: The first two tokens don't match.

This is where the intelligence of the LALR parser comes in. As I mentioned earlier, the acronym LALR stands for *Look-Ahead LR* parser, and it

describes the algorithm the parser uses to find matching grammar rules. We'll get to the *look ahead* part in a minute. For now, let's start with LR:

- **L** (left) means the parser moves from left to right while processing the token stream. In this example, that would be `le`, `gusta`, `el`, and `ruby`, in that order.
- **R** (reversed rightmost derivation) means the parser takes a bottom-up strategy, using a shift/reduce technique, to find matching grammar rules.

Here's how the algorithm works for Phrase 2. First, the parser takes the input token stream, shown again in Figure 1-16.

Figure 1-16: The input stream of tokens

Next, it shifts the tokens to the left, creating what I'll call the *grammar rule stack*, as shown Figure 1-17.

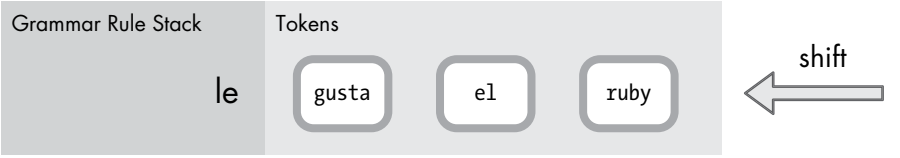

Figure 1-17: The parser moves the first token onto the grammar rule stack.

Because the parser has processed only the token `le`, it places this token in the stack alone for the moment. The term g*rammar rule stack* is a bit of an oversimplification; while the parser uses a stack, instead of grammar rules, it pushes numbers onto its stack to indicate which grammar rule it has just parsed. These numbers, or *states*, help the parser keep track of which grammar rules it has matched as it processes tokens.

Next, as shown in Figure 1-18, the parser shifts another token to the left.

Figure 1-18: The parser moves another token onto the stack.

Now there are two tokens in the stack on the left. At this point, the parser stops to search the different grammar rules for a match. Figure 1-19 shows the parser matching the `SheLikes` rule.

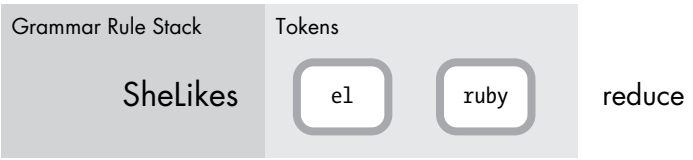

Figure 1-19: The parser matches the `SheLikes` rule and reduces.

This operation is called *reduce* because the parser is replacing the pair of tokens with a single matching rule. The parser looks through the available rules and reduces, or applies the single matching rule.

Now the parser can reduce again because there's another matching rule: `VerbAndObject`! The `VerbAndObject` rule matches because its use of the `OR` (`|`) operator matches *either* the `SheLikes` *or* `ILike` child rules.

You can see in Figure 1-20 that the parser replaces `SheLikes` with `VerbAndObject`.

Figure 1-20: The parser reduces again, matching the `VerbAndObject` rule.

But think about this: How did the parser know to reduce and not continue to shift tokens? Also, if in the real world there are actually many matching rules, how does the parser know which one to use? How does it decide whether to shift or reduce? And if it reduces, how does it decide which grammar rule to reduce with?

In other words, suppose at this point in the process multiple matching rules included `le gusta`. How would the parser know which rule to apply or whether to shift in the `el` token first before looking for a match? (See Figure 1-21.)

Figure 1-21: How does the parser know to shift or reduce?

Here's where the *look ahead* portion of LALR comes in. In order to find the correct matching rule, the parser looks ahead at the next token. The arrow in Figure 1-22 shows the parser looking ahead at the `el` token.

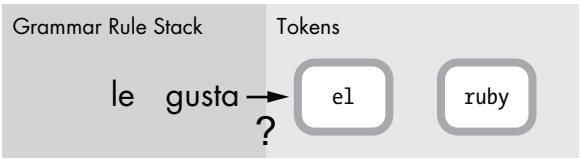

Figure 1-22: Looking ahead at the next token in the input stream

Additionally, the parser maintains a state table of possible outcomes depending on what the next token is and which grammar rule was just parsed. In this case, the table would contain a series of states, describing which grammar rules have been parsed so far and which states to move to next depending on the next token. (LALR parsers are complex state machines that match patterns in the token stream. When you use Bison to generate the LALR parser, Bison calculates what this state table should contain based on the grammar rules you provided.)

In this example, the state table would contain an entry indicating that if the next token was `el`, the parser should first reduce using the `SheLikes` rule before shifting a new token.

Rather than waste your time with the details of what a state table looks like (you'll find the actual LALR state table for Ruby in the generated *parse.c* file), let's continue the shift/reduce operations for Phrase 2, "Le gusta el Ruby." After matching the `VerbAndObject` rule, the parser would shift another token to the left, as shown in Figure 1-23.

Figure 1-23: The parser shifts another token onto the stack.

At this point, no rules would match, and the state machine would shift another token to the left (see Figure 1-24).

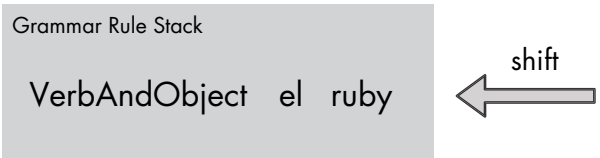

Figure 1-24: The parser shifts another token onto the stack.

Figure 1-25 shows how the parent grammar rule `SpanishPhrase` would match after a final reduce operation.

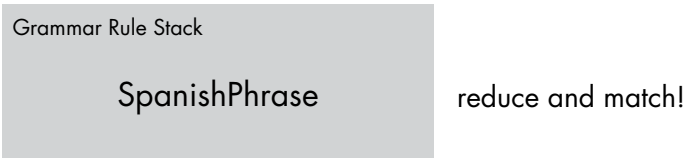

Figure 1-25: The parser matches the `SpanishPhrase` rule—and the entire input stream!

I've shown you this Spanish-to-English example because Ruby parses your program in exactly the same way! Inside the Ruby *parse.y* source code file, you'll see hundreds of rules that define the structure and syntax of the Ruby language. There are parent and child rules, and the child rules return values the parent rules can refer to in exactly the same way our `SpanishPhrase` grammar rules do, using the symbols `$$`, `$1`, `$2`, and so on. The only real difference is scale: Our `SpanishPhrase` grammar example is trivial, really. In contrast, Ruby's grammar is very complex; it's an intricate series of interrelated parent and child grammar rules, which sometimes refer to each other in circular, recursive patterns. But this complexity just means that the generated state table in *parse.c* is quite large. The basic LALR algorithm, which describes how the parser processes tokens and uses the state table, is the same in our Spanish example as it is in Ruby.

To get a sense of just how complex the state table is for Ruby, you can try using Ruby's `-y` option, which displays internal debug information every time the parser jumps from one state to another. Listing 1-14 shows a small portion of the output generated when you run the `10.times do` example from Listing 1-1.

```
$ ruby -y simple.rb
Starting parse
Entering state 0
Reducing stack by rule 1 (line 850):
-> $$ = nterm @1 ()
Stack now 0
Entering state 2
Reading a token: Next token is token tINTEGER ()
Shifting token tINTEGER ()
Entering state 41
Reducing stack by rule 498 (line 4293):
   $1 = token tINTEGER ()
-> $$ = nterm numeric ()
Stack now 0 2
Entering state 109
--snip--
```

Listing 1-14: Ruby optionally displays debug information, showing how the parser jumps from one state to another.

Some Actual Ruby Grammar Rules

Let's look at some actual Ruby grammar rules from *parse.y*. Listing 1-15 contains the simple example Ruby script from Listing 1-1 on page 4.

```
10.times do |n|
  puts n
end
```

Listing 1-15: The simple Ruby program from Listing 1-1.

Figure 1-26 shows how Ruby's parsing process works with this script.

Ruby Code

```
10.times do |n|
  puts n
end
```

Grammar Rules

```
program: top_compstmt
top_compstmt: top_stmts opt_terms
top_stmts: ... | top_stmt | ...
top_stmt: stmt | ...
stmt: ... | expr
expr: ... | arg
arg: ... | primary
primary: ... | method_call brace_block | ...
```

Figure 1-26: The grammar rules on the right match the Ruby code on the left.

On the left is the code that Ruby is trying to parse. On the right are the actual matching grammar rules from the Ruby *parse.y* file, shown simplified. The first rule, `program: top_compstmt`, is the root grammar rule that matches every Ruby program in its entirety.

As you go down the list, you see a complex series of child rules that also match the entire Ruby script: top statements, a single statement, an expression, an argument, and, finally, a primary value. Once Ruby's parse reaches the primary grammar rule, it encounters a rule with two matching child rules: `method_call` and `brace_block`. Let's look at `method_call` first (see Figure 1-27).

Ruby Code

```
10.times
```

Grammar Rules

```
method_call: ... | primary_value '.' operation2 | ...
```

Figure 1-27: `10.times` matches the `method_call` grammar rule.

The `method_call` rule matches the `10.times` portion of the Ruby code—that is, where we call the `times` method on the `10 Fixnum` object. You can

see that the `method_call` rule matches another primary value, followed by a period character, followed by an `operation2` rule.

Figure 1-28 shows that the `primary_value` rule first matches the value `10`.

Ruby Code	Grammar Rules
`10`	`primary_value: primary` `primary: literal | ...`

Figure 1-28: The value `10` matches the `primary_value` grammar rule.

Then, in Figure 1-29, the `operation2` rule matches the method name `times`.

Ruby Code	Grammar Rules
`times`	`operation2: identifier | ...`

Figure 1-29: The `times` method name matches the `operation2` grammar rule.

How does Ruby parse the contents of the `do ... puts ... end` block that's passed to the `times` method? It uses the `brace_block` rule we saw in Figure 1-26. Figure 1-30 shows the definition of the `brace_block` rule.

Ruby Code	Grammar Rules

`do |n|`
`  puts n`
`end`
	`brace_block: ... | keyword_do opt_block_param compstmt keyword_end | ...`

Figure 1-30: The entire block matches the `brace_block` rule.

I don't have space here to go through all the remaining child grammar rules, but you can see how this rule, in turn, contains a series of other matching child rules:

- `keyword_do` matches the `do` reserved keyword.
- `opt_block_param` matches the block parameter `|n|`.
- `compstmt` matches the contents of the block itself, `puts n`.
- `keyword_end` matches the `end` reserved keyword.

READING A BISON GRAMMAR RULE

To give you a taste of the actual Ruby *parse.y* source code, take a look at Listing 1-16, which shows part of the `method_call` ❶ grammar rule definition.

```
❶ method_call        :
--snip--
      primary_value '.' operation2
      {
      /*%%%*/
          $<num>$ = ruby_sourceline;
      /*% %*/
      }
    opt_paren_args
      {
      /*%%%*/
          $$ = NEW_CALL($1, $3, $5);
          nd_set_line($$, $<num>4);
      /*%
          $$ = dispatch3(call, $1, ripper_id2sym('.'), $3);
          $$ = method_optarg($$, $5);
      %*/
      }
```

Listing 1-16: Ruby's actual `method_call` grammar rule from parse.y

As with the preceding Spanish-to-English example grammar file, you can see that there are snippets of complex C code after each of the terms in the grammar rule. Listing 1-17 shows one example of this.

```
      $$ = NEW_CALL($1, $3, $5);
      nd_set_line($$, $<num>4);
```

Listing 1-17: Ruby calls this C code when the `opt_paren_args` grammar rule matches.

The Bison-generated parser will execute one of these snippets when there's a match for a rule on the tokens found in the target Ruby script. However, these C code snippets also contain Bison directives, such as `$$` and `$1`, that allow the code to create return values and to refer to values returned by other grammar rules. We end up with a confusing mix of C and Bison directives.

To make things worse, Ruby uses a trick during its build process to divide these C/Bison code snippets into separate pieces. Some of these pieces are used by Ruby, while others are used only by the Ripper tool from Experiment 1-1. Here's how that trick works:

- The C code that appears between the `/*%%%*/` line and the `/*%` line in Listing 1-16 is actually compiled into Ruby during the Ruby build process.
- The C code between `/*%` and `%*/` in Listing 1-16 is dropped when Ruby is built. This code is used only by the Ripper tool, which is built separately during the Ruby build process.

Ruby uses this very confusing syntax to allow the Ripper tool and Ruby itself to share the same grammar rules inside *parse.y*.

What are these snippets actually doing? As you might guess, Ruby uses the Ripper code snippets to allow the Ripper tool to display information about what Ruby is parsing. (We'll try that next, in Experiment 1-2.) There's also some bookkeeping code: Ruby uses the `ruby_sourceline` variable to keep track of which source code line corresponds to each portion of the grammar.

But more importantly, the snippets Ruby actually uses at run time when parsing your code create a series of *nodes*, or temporary data structures, that form an internal representation of your Ruby code. These nodes are saved in a tree structure called an *abstract syntax tree (AST)* (more about this in Experiment 1-2). You can see one example of creating an AST node in Listing 1-17, where Ruby calls the `NEW_CALL` C macro/function. This call creates a new `NODE_CALL` node, which represents a method call. (In Chapter 2 we'll see how Ruby eventually compiles this into bytecode that can be executed by a virtual machine.)

Experiment 1-2: Using Ripper to Parse Different Ruby Scripts

In Experiment 1-1, you learned how to use Ripper to display the tokens that Ruby converts your code into, and we've just seen how the Ruby grammar rules in *parse.y* are also included in the Ripper tool. Now let's learn how to use Ripper to display information about how Ruby parses your code. Listing 1-18 shows how to do it.

```
require 'ripper'
require 'pp'
code = <<STR
10.times do |n|
  puts n
end
STR
puts code
❶ pp Ripper.sexp(code)
```

Listing 1-18: An example of how to call `Ripper.sexp`

This is exactly the same code from Experiment 1-1, except that we call `Ripper.sexp` ❶ instead of `Ripper.lex`. Running this gives the output shown in Listing 1-19.

```
[:program,
  [[:method_add_block,
     [:call,
       [:@int, "10", [1, 0]], :".",
       [:@ident, "times", [1, 3]]],
```

```
[:do_block,
  [:block_var,
    [:params, [[:@ident, "n", [1, 13]]],
              nil, nil, nil, nil, nil, nil],
    false],
  [[:command,
     [:@ident, "puts", [2, 2]],
     [:args_add_block, [[:var_ref, [:@ident, "n", [2, 7]]]],
                       false]]]]]]
```

Listing 1-19: The output generated by `Ripper.sexp`

You can see some bits and pieces from the Ruby script in this cryptic text, but what do all of the other symbols and arrays mean?

It turns out that the output from Ripper is a textual representation of your Ruby code. As Ruby parses your code, matching one grammar rule after another, it converts the tokens in your code file into a complex internal data structure called an *abstract syntax tree (AST)*. (You can see some of the C code that produces this structure in "Reading a Bison Grammar Rule" on page 22.) The AST is used to record the structure and syntactical meaning of your Ruby code.

To see what I mean, look at Figure 1-31, which shows a graphical view of part of the output that Ripper generated for us: the `puts n` statement inside the block.

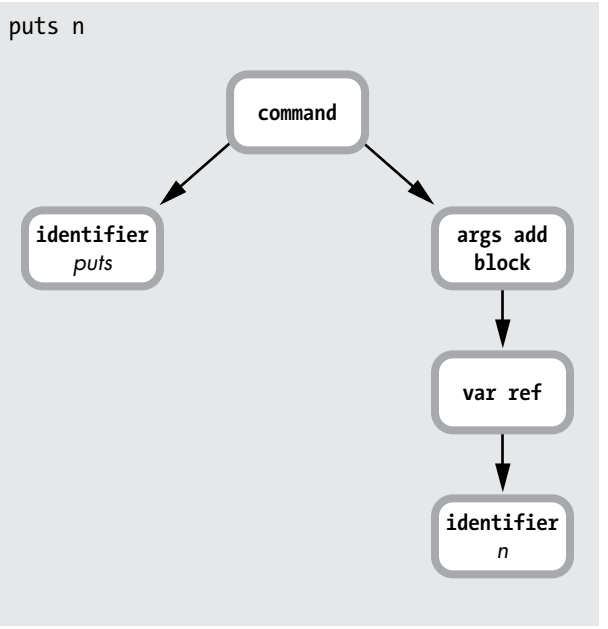

Figure 1-31: The portion of the AST corresponding to `puts n`

This diagram corresponds to the last three lines of the Ripper output, repeated here in Listing 1-20.

```
[[:command,
❶   [:@ident, "puts", [2, 2]],
    [:args_add_block, [[:var_ref, [:@ident, "n", [2, 7]]]],
                      false]]]
```

Listing 1-20: The last three lines of the `Ripper.sexp` output

As in Experiment 1-1, when we displayed token information from Ripper, you can see that the source code file line and column information are displayed as integers. For example, `[2, 2]` ❶ indicates that Ripper found the `puts` call on line 2 at column 2 of the code file. You can also see that Ripper outputs an array for each of the nodes in the AST—with `[:@ident, "puts", [2, 2]]` ❶, for example.

Now your Ruby program is beginning to "make sense" to Ruby. Instead of a simple stream of tokens, which could mean anything, Ruby now has a detailed description of what you meant when you wrote `puts n`. You see a function call (a command), followed by an identifier node that indicates which function to call.

Ruby uses the `args_add_block` node because you could pass a block to a command/function call like this. Even though you're not passing a block in this case, the `args_add_block` node is still saved into the AST. (Notice, too, how the `n` identifier is recorded as a `:var_ref`, or variable reference node, not as a simple identifier.)

Figure 1-32 represents more of the Ripper output.

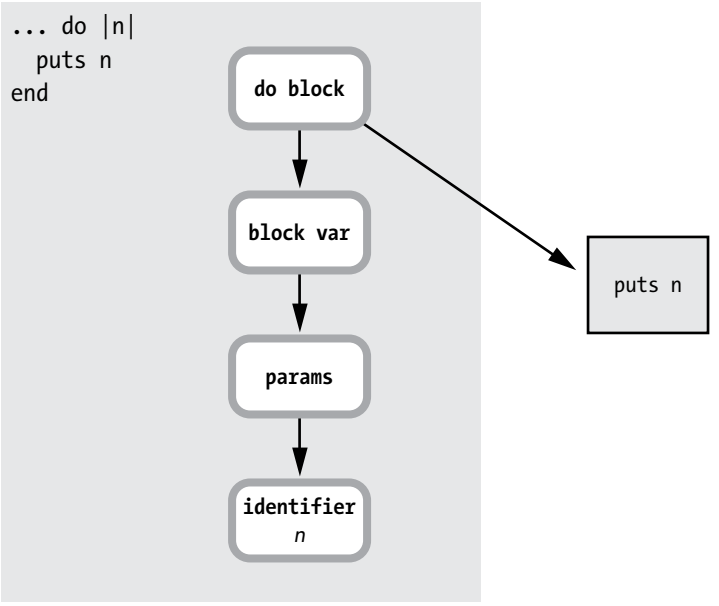

Figure 1-32: The portion of the AST corresponding to the entire block

You can see that Ruby now understands that `do |n| ... end` is a block, with a single block parameter called `n`. The `puts n` box on the right represents the other part of the AST shown earlier—the parsed version of the `puts` call.

Finally, Figure 1-33 shows the entire AST for the sample Ruby code.

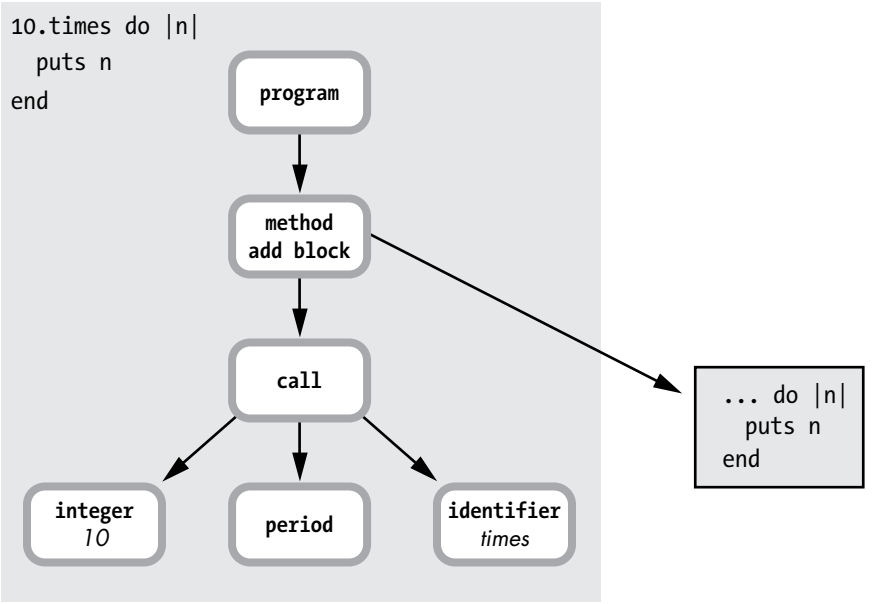

Figure 1-33: The AST for the entire Ruby program

Here, `method add block` means that you're calling a method, but with a block parameter: `10.times do`. The `call` tree node obviously represents the actual method call `10.times`. This is the `NODE_CALL` node that we saw earlier in the C code snippet. Ruby's understanding of what you meant with your code is saved in the way the nodes are arranged in the AST.

To clarify things, suppose you pass the Ruby expression `2 + 2` to Ripper, as shown in Listing 1-21.

```
require 'ripper'
require 'pp'
code = <<STR
2 + 2
STR
puts code
pp Ripper.sexp(code)
```

Listing 1-21: This code will display the AST for `2 + 2`.

Running this code gives the output in Listing 1-22.

```
[:program,
  [[:binary,
     [:@int, "2", [1, 0]],
     :+,
     [:@int, "2", [1, 4]]]]]
```

Listing 1-22: The output of `Ripper.sexp` for `2 + 2`

As you can see in Figure 1-34 below, the `+` is represented with an AST node called `binary`.

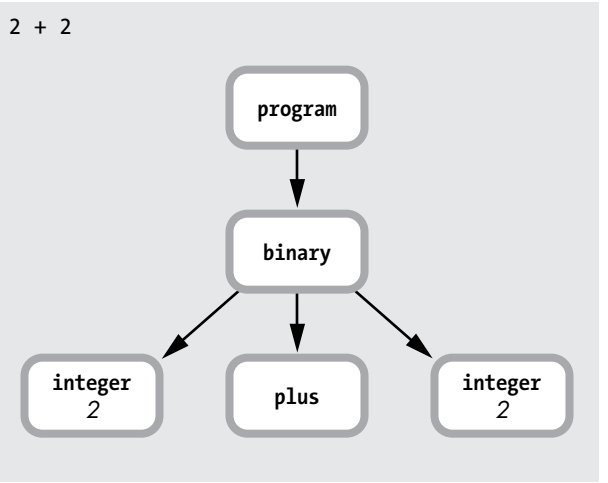

Figure 1-34: The AST for `2 + 2`

But see what happens when I pass the expression `2 + 2 * 3` into Ripper, as in Listing 1-23.

```
require 'ripper'
require 'pp'
code = <<STR
2 + 2 * 3
STR
puts code
pp Ripper.sexp(code)
```

Listing 1-23: Code to display the AST for `2 + 2 * 3`

Listing 1-24 shows that you get a second binary node for the `*` operator at ❶.

```
[:program,
 [[:binary,
   [:@int, "2", [1, 0]],
   :+,
❶  [:binary,
     [:@int, "2", [1, 4]],
     :*,
     [:@int, "3", [1, 8]]]]]]
```

Listing 1-24: The output of `Ripper.sexp` for `2 + 2 * 3`

Figure 1-35 shows what that looks like.

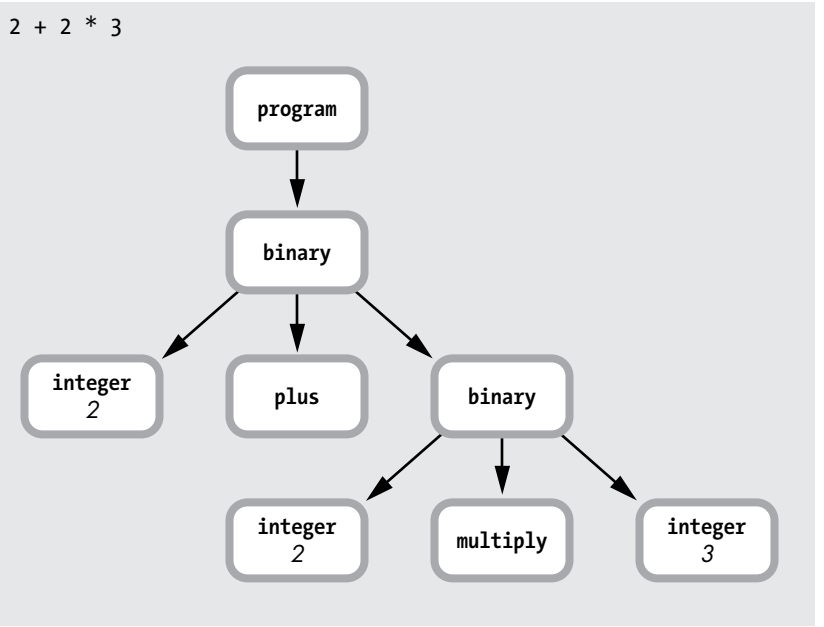

Figure 1-35: The AST for `2 + 2 * 3`

Ruby was smart enough to realize that multiplication has a higher precedence than addition, but what's really interesting is how the AST tree structure captures the information about the order of operations. The token stream `2 + 2 * 3` simply indicates what you wrote in your code file. But the parsed version that's saved to the AST structure now contains the *meaning* of your code—that is, all of the information Ruby will need later to execute it.

One final note: Ruby actually contains some debug code that can display information about the AST node structure. To use it, run your Ruby script with the `parsetree` option (see Listing 1-25).

```
$ ruby --dump parsetree your_script.rb
```

Listing 1-25: Display debug information about your code's AST using the `parsetree` option.

This will display the same information we've just seen, but instead of showing symbols, the `parsetree` option should show the actual node names from the C source code. (In Chapter 2, I'll also use the actual node names.)

Summary

In Chapter 1, we looked at one of the most fascinating areas of computer science: how Ruby can *understand* the text that you give it—your Ruby program. In order to do this, Ruby converts your code into two different formats. First, it converts the text in your Ruby program into a series of *tokens*. Next, it uses an LALR parser to convert the input stream of tokens into a data structure called an *abstract syntax tree*.

In Chapter 2, we'll see that Ruby converts your code into a third format: a series of *bytecode instructions* that are later used when your program is actually executed.