# 2

# COMPILATION

Now that Ruby has tokenized and parsed your code, is it ready to run it? Will it finally get to work and iterate through the block 10 times in my simple `10.times do` example? If not, what else could Ruby possibly have to do first?

Starting with version 1.9, Ruby compiles your code before executing it. The word *compile* means to translate your code from one programming language to another. Your programming language is easy for you to understand, while usually the target language is easy for the computer to understand.

For example, when you compile a C program, the compiler translates C code to machine language, a language your computer's microprocessor hardware understands. When you compile a Java program, the compiler translates Java code to Java bytecode, a language the Java Virtual Machine understands.

Ruby's compiler is no different. It translates your Ruby code into another language that Ruby's virtual machine understands. The only difference is that you don't use Ruby's compiler directly; unlike in C or Java,

Ruby's compiler runs automatically without you ever knowing. Here in Chapter 2, I'll explain how Ruby does this and what language it translates your code into.

**ROADMAP**



## No Compiler for Ruby 1.8

The Ruby core team introduced a compiler with version 1.9. Ruby 1.8 and earlier versions of Ruby don't contain a compiler. Instead, Ruby 1.8 immediately executes your code after the tokenizing and parsing processes are finished. It does this by walking through the nodes in the AST tree and executing each one. Figure 2-1 shows another way of looking at the Ruby 1.8 tokenizing and parsing processes.

The top of Figure 2-1 shows your Ruby code. Below this are the different internal formats Ruby converts your Ruby code into. These are the tokens and AST nodes we saw in Chapter 1—the different

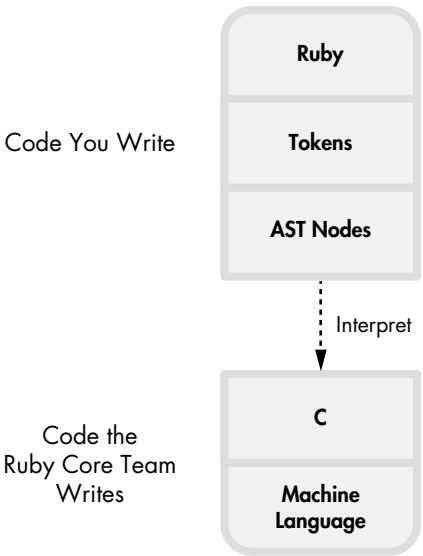


*Figure 2-1: In Ruby 1.8, your code is converted into AST nodes and then interpreted.*

forms your code takes when you run it using Ruby. The lower section of the diagram shows the code the Ruby core team wrote: the C source code for the Ruby language and the machine language it is converted into by the C compiler.

The dotted line between the two code sections indicates that Ruby interprets your code. The Ruby C code, the lower section, reads and executes your code, the top section. Ruby 1.8 doesn't compile or translate your code into any form beyond AST nodes. After converting it into AST nodes, it proceeds to iterate over the nodes in the AST, taking whatever action each node represents as it executes each node.

The gap in the middle of the diagram shows that your code is never completely compiled into machine language. If you were to disassemble and inspect the machine language that your CPU actually runs, you would not see instructions that directly map to your original Ruby code. Instead, you would find instructions that tokenize, parse, and execute your code, or, in other words, that implement the Ruby interpreter.

## Ruby 1.9 and 2.0 Introduce a Compiler

If you've upgraded to Ruby 1.9 or 2.0, Ruby is still not quite ready to run your code. It needs to compile it first.

With Ruby 1.9, Koichi Sasada and the Ruby core team introduced Yet Another Ruby Virtual Machine (YARV), which actually executes your Ruby code. At a high level, this is the same idea behind the Java Virtual Machine (JVM) used by Java and many other languages. (I'll cover YARV in more detail in Chapters 3 and 4.)

When using YARV (as with the JVM), you first compile your code into *bytecode*, a series of low-level instructions that the virtual machine understands. The only differences between YARV and the JVM are the following:

- Ruby doesn't expose the compiler to you as a separate tool. Instead, it automatically compiles your Ruby code into bytecode instructions internally.
- Ruby never compiles your Ruby code all the way to machine language. As you can see in Figure 2-2, Ruby interprets the bytecode instructions. The JVM, on the other hand, can compile some of the bytecode instructions all the way into machine language using its "hotspot" or just-in-time (JIT) compiler.

Figure 2-2 shows how Ruby 1.9 and 2.0 handle your code.

Notice that this time, unlike in the process shown in Figure 2-1, your code is translated into a third format. After parsing the tokens and producing the AST, Ruby 1.9 and 2.0 continue to compile your code into a series of low-level instructions called *YARV instructions*.

The primary reason for using YARV is speed: Ruby 1.9 and 2.0 run much faster than Ruby 1.8 due to the use of YARV instructions. Like Ruby 1.8, YARV is an interpreter—just a faster one. Your Ruby code ultimately is still not converted directly into machine language by Ruby 1.9 or 2.0. There is still a gap in Figure 2-2 between the YARV instructions and Ruby's C code.

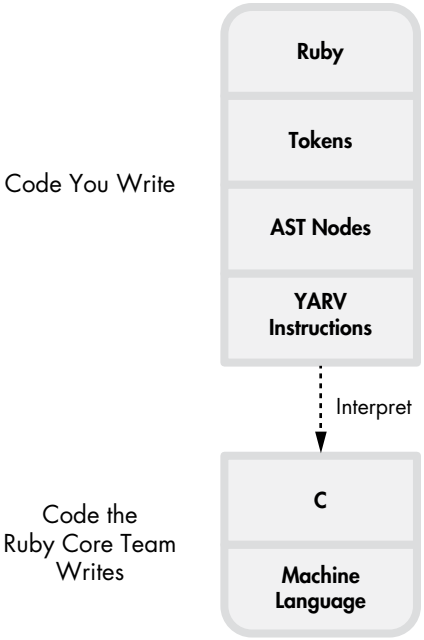


*Figure 2-2: Ruby 1.9 and 2.0 compile the AST nodes into YARV instructions before interpreting them.*

## How Ruby Compiles a Simple Script

In this section, we'll look at the last step along your code's journey through Ruby: how Ruby compiles your code into the instructions that YARV expects. Let's explore how Ruby's compiler works by stepping through an example compilation. Listing 2-1 shows a simple Ruby script that calculates 2 + 2 = 4.

```
puts 2+2
```

*Listing 2-1: A one-line Ruby program we will compile as an example*

Figure 2-3 shows the AST structure that Ruby will create after tokenizing and parsing this simple program. (This is a more detailed view of the AST than you would get from the Ripper tool that we saw in Experiment 1-2 on page 23.)

**NOTE** *The technical names shown in Figure 2-3 (`NODE_SCOPE`, `NODE_FCALL`, and so on) are taken from the actual Ruby C source code. To keep things simple, I'm omitting some AST nodes—specifically, ones that represent arrays of the arguments to each method call, which in this simple example would be arrays of only one element.*

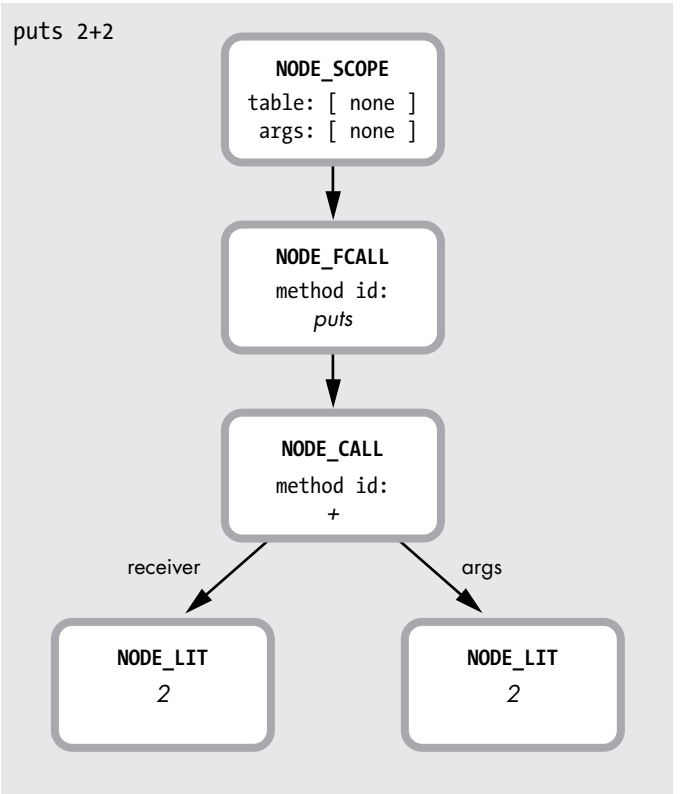


*Figure 2-3: The AST Ruby produces after parsing the code in Listing 2-1*

Before we cover the details of how Ruby compiles the `puts 2+2` script, let's look at one very important attribute of YARV: It's a *stack-oriented virtual machine*. That means when YARV executes your code, it maintains a stack of values—mainly arguments and return values for the YARV instructions. (I'll explain this in more detail in Chapter 3.) Most of YARV's instructions either push values onto the stack or operate on the values that they find on the stack, leaving a result value on the stack as well.

In order to compile the `puts 2+2` AST structure into YARV instructions, Ruby will iterate over the tree recursively from the top down, converting each AST node into instructions. Figure 2-4 shows how this works, beginning with `NODE_SCOPE`.

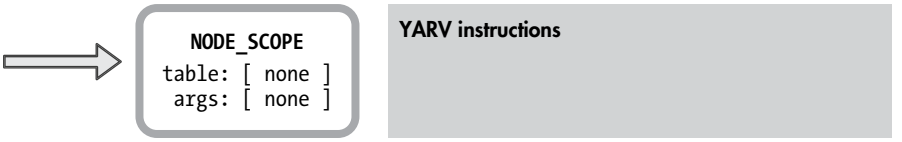


*Figure 2-4: Ruby starts the compile process at the root of the AST.*

`NODE_SCOPE` tells the Ruby compiler that it is starting to compile a new *scope*, or section of Ruby code, which, in this case, is a whole new program.

This scope is indicated on the right with an empty box. (The `table` and `args` values are both empty, so we'll ignore them for now.)

Next, the Ruby compiler steps down the AST tree and encounters `NODE_FCALL`, as shown in Figure 2-5.

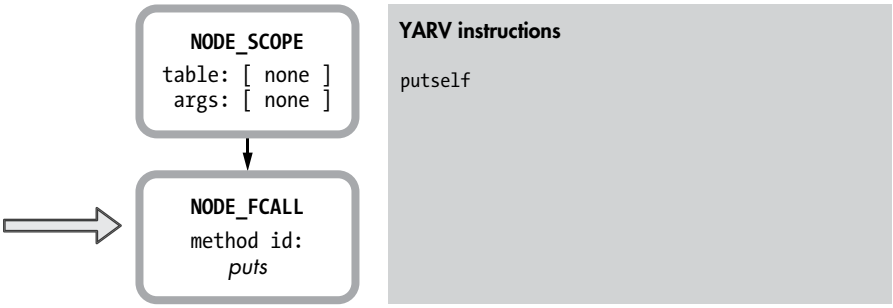


*Figure 2-5: To compile a function call, Ruby first creates an instruction to push the receiver.*

`NODE_FCALL` represents a *function call*—in this case, the call to `puts`. (Function and method calls are very important and very common in Ruby programs.) Ruby compiles function calls for YARV according to the following pattern:

- Push receiver.
- Push arguments.
- Call the method/function.

In Figure 2-5, the Ruby compiler first creates a YARV instruction called `putself` to indicate that the function call uses the current value of the `self` pointer as the receiver. Because I call `puts` from the top-level scope—that is, the top section—of this simple script, `self` is set to point to the `top self` object. (The `top self` object is an instance of the `Object` class that is automatically created when Ruby starts up. One purpose of `top self` is to serve as the receiver for function calls like this one in the top-level scope.)

**NOTE** *In Ruby all functions are actually methods. That is, functions are always associated with a Ruby class; there is always a receiver. Inside of Ruby, however, Ruby's parser and compiler distinguish between functions and methods: Method calls have an explicit receiver, while function calls assume the receiver is the current value of* `self`.

Next, Ruby needs to create instructions to push the arguments of the `puts` function call. But how? The argument to `puts` is `2+2`, which is the result of another method call. Although `2+2` is a simple expression, `puts` could instead be operating on some extremely complex Ruby expression involving many operators, method calls, and so on. How can Ruby know which instructions to create here?

The answer lies in the structure of the AST. By simply following the tree nodes down recursively, Ruby can take advantage of all the parser's earlier work. In this case, it can now just step down to the NODE_CALL node, as shown in Figure 2-6.

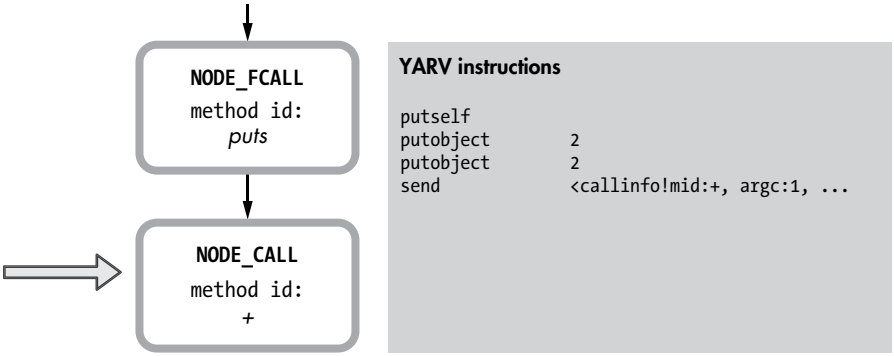


*Figure 2-6: Next, Ruby writes instructions for calculating 2+2, the argument to puts.*

Here Ruby will compile the + method call, which theoretically is the process of sending the + message to the 2 integer object. Again, following the same receiver, arguments, method call pattern, Ruby performs these actions in order:

1. Creates a YARV instruction to push the receiver onto the stack (the object 2 in this case).
2. Creates a YARV instruction to push the argument or arguments onto the stack (again, 2 in this example).
3. Creates a method call YARV instruction send <callinfo!mid:+, argc:1, ARGS_SKIP> that means "send the + message" to the receiver, which is the object previously pushed onto the YARV stack (in this case, the first Fixnum 2 object). mid:+ means "method id = +" and is the name of the method we want to call. The argc:1 parameter tells YARV there is one argument to this method call (the second Fixnum 2 object). ARGS_SKIP indicates the arguments are simple values (not blocks or arrays of unnamed arguments), allowing YARV to skip some work it would have to do otherwise.

When Ruby executes the send <callinfo!mid:+... instruction it adds 2+2, fetching those arguments from the stack, and leaves the result, 4, as a new value on top of the stack. What's fascinating about this is that YARV's stack-oriented nature also helps Ruby compile the AST nodes more easily, as you can see when it finishes compiling the NODE_FCALL, as shown in Figure 2-7.

Now Ruby can assume that the return value of the 2+2 operation—that is, 4—will be left at the top of the stack, just where it needs to be as the argument to the puts function call. Ruby's stack-oriented virtual machine goes hand in hand with the way that it recursively compiles the AST nodes! As you can see at the right of Figure 2-7, Ruby has added the send <callinfo!mid:puts, argc:1 instruction, which calls puts and indicates that there is one argument to puts.

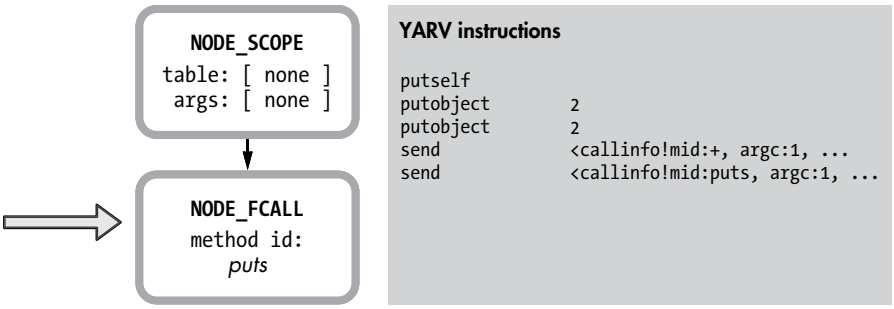


*Figure 2-7: Finally, Ruby can write an instruction for the call to `puts`.*

As it turns out, Ruby further modifies these YARV instructions before executing them as part of an optimize step. One of its optimizations is to replace some YARV instructions with *specialized instructions*, which are YARV instructions that represent commonly used, primitive operations, such as `size`, `not`, `less than`, `greater than`, and so on. One such instruction, `opt_plus`, is used for adding two numbers together. During optimization, Ruby replaces `send <callinfo!mid:+...` with `opt_plus`, as shown in Figure 2-8.

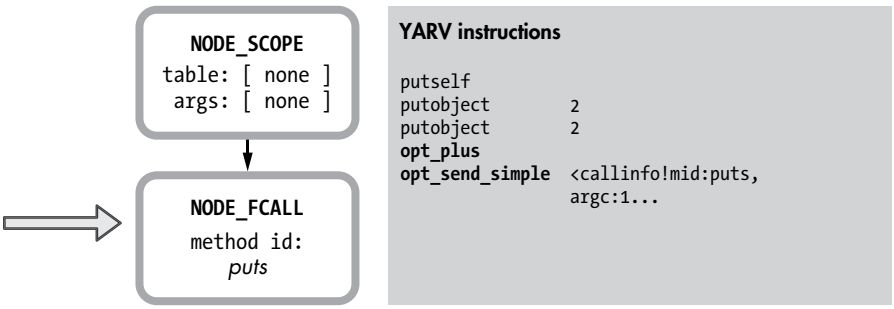


*Figure 2-8: Ruby replaces some instructions with specialized instructions.*

As you can see in Figure 2-8, Ruby also replaces the second `send` with `opt_send_simple`, which runs a bit faster when none of the arguments needs special treatment, such as expansion.

## Compiling a Call to a Block

Next, let's compile my `10.times do` example from Listing 1-1 in Chapter 1 (see Listing 2-2).

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

*Listing 2-2: A simple script that calls a block*

Notice that this example contains a block parameter to the `times` method. This is interesting because it will give us a chance to see how the Ruby compiler handles blocks. Figure 2-9 shows the AST for the `10.times do` example again, using the actual node names rather than the simplified output from Ripper.

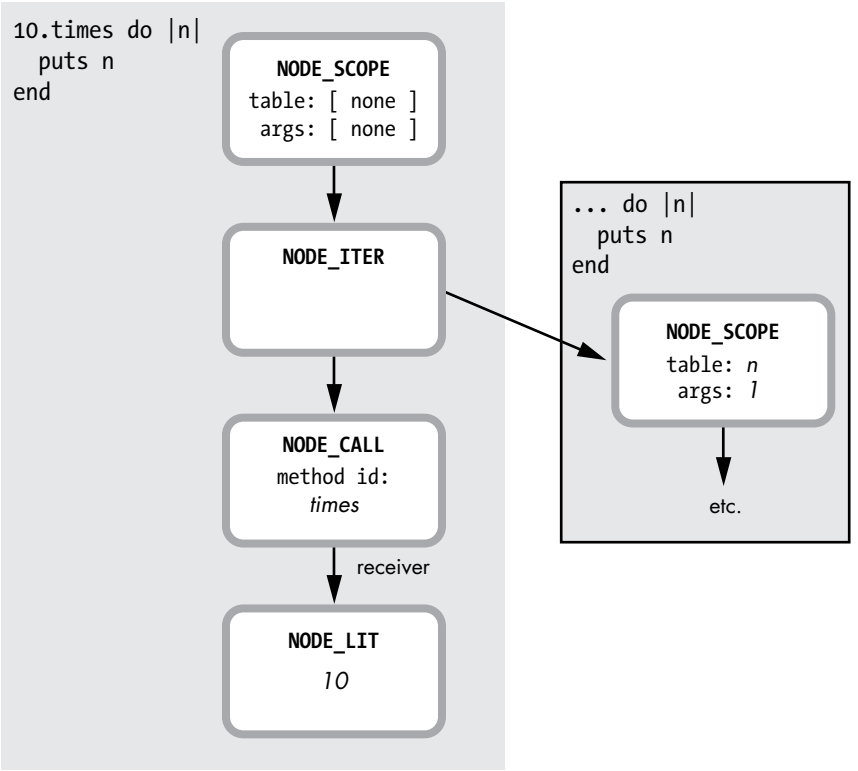


*Figure 2-9: The AST for the call to `10.times`, passing a block*

This looks very different than `puts 2+2`, mostly because of the inner block shown at the right. (Ruby handles the inner block differently, as we'll see shortly.)

Let's break down how Ruby compiles the main portion of the script shown on the left of Figure 2-9. As before, Ruby starts with the first `NODE_SCOPE` and creates a new snippet of YARV instructions, as shown in Figure 2-10.

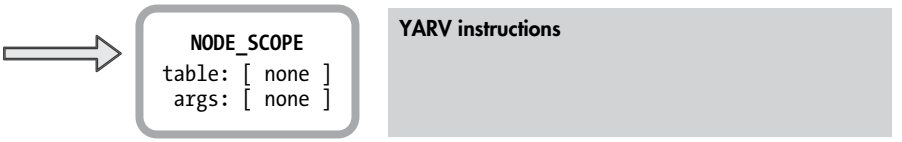


*Figure 2-10: Each `NODE_SCOPE` is compiled into a new snippet of YARV instructions.*

Next, Ruby steps down the AST nodes to NODE_ITER, as shown in Figure 2-11.

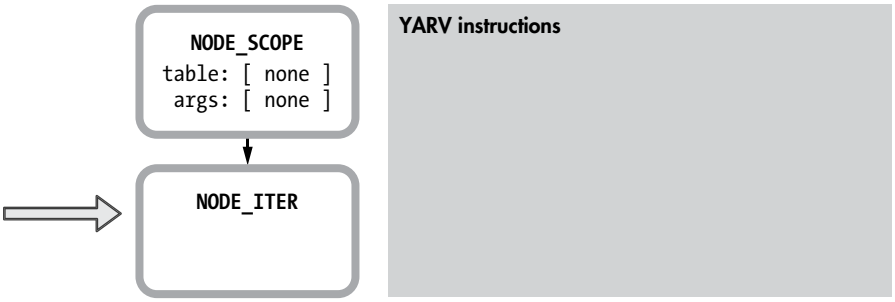


*Figure 2-11: Ruby stepping through an AST*

At this point, there is still no code generated, but notice in Figure 2-9 that two arrows lead from NODE_ITER: one to NODE_CALL, which represents the `10.times` call, and another to the inner block. Ruby will first continue down the AST and compile the nodes corresponding to the `10.times` code. The resulting YARV code, following the same receiver-arguments-message pattern we saw in Figure 2-6, is shown in Figure 2-12.

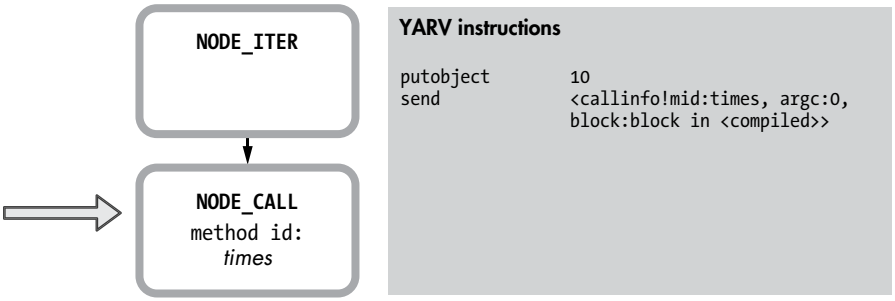


*Figure 2-12: Ruby compiles the `10.times` method call.*

Notice that the new YARV instructions shown in Figure 2-12 push the receiver (the integer object `10`) onto the stack first, after which Ruby generates an instruction to execute the `times` method call. But notice, too, the `block:block in <compiled>` argument in the `send` instruction. This indicates that the method call also contains a block argument: my `do |n| puts n end` block. In this example, NODE_ITER has caused the Ruby compiler to include this block argument because the AST above shows an arrow from NODE_ITER to the second NODE_SCOPE.

Ruby continues by compiling the inner block, beginning with the second NODE_SCOPE shown at right in Figure 2-9. Figure 2-13 shows what the AST for that inner block looks like.

This looks simple enough—just a single function call and a single argument n. But notice the value for table and args in NODE_SCOPE. These values were empty in the parent NODE_SCOPE, but they're set here in the inner NODE_SCOPE. As you might guess, these values indicate the presence of the block parameter n.

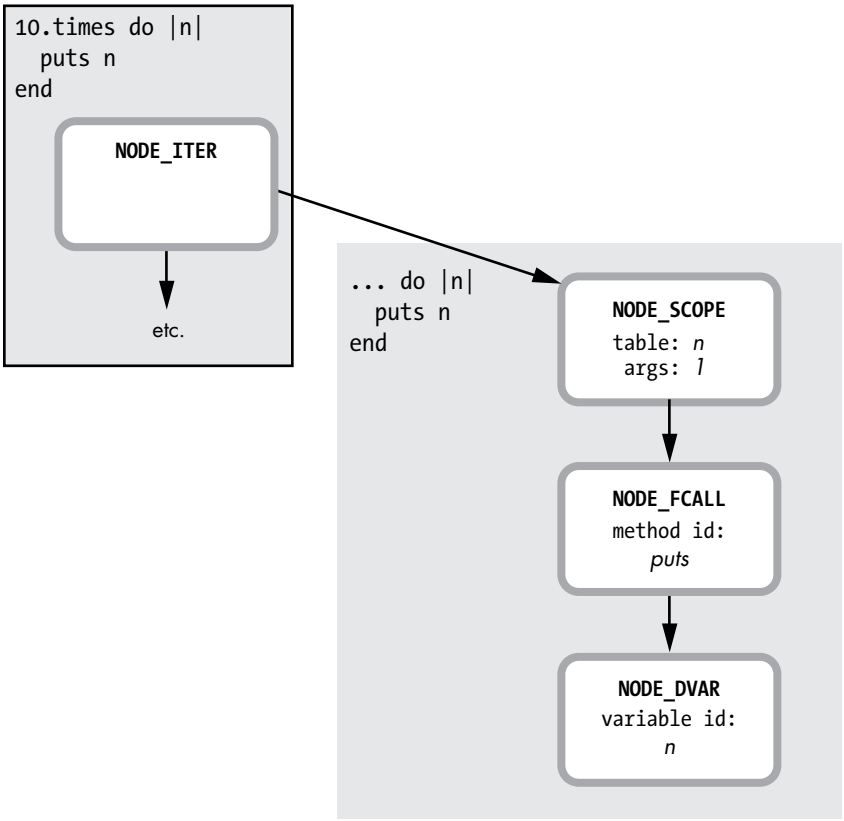


*Figure 2-13: The branch of the AST for the contents of the block*

Also notice that the Ruby parser created NODE_DVAR instead of NODE_LIT, which we saw earlier in Figure 2-9. This is the case because n is not just a literal string; it's a block parameter passed in from the parent scope.

From a relatively high level, Figure 2-14 shows how Ruby compiles the inner block.

### HOW RUBY ITERATES THROUGH THE AST

Let's look more closely at how Ruby actually iterates through the AST structure, converting each node into YARV instructions. The MRI C source code file that implements the Ruby compiler is called *compile.c*. To learn how the code in *compile.c* works, we first look for the function `iseq_compile_each`. Listing 2-3 shows the beginning of that function.

```
/**
  compile each node

  self:  InstructionSequence
  node:  Ruby compiled node
  poped: This node will be poped
 */
static int
iseq_compile_each(rb_iseq_t *iseq, LINK_ANCHOR *ret, NODE * node,
                  int poped)
{
```

*Listing 2-3: This C function compiles each node in the AST.*

This function is very long, with a very, very long `switch` statement that runs to thousands of lines! The `switch` statement branches based on the type of the current AST node and generates the corresponding YARV code. Listing 2-4 shows the start of the `switch` statement ❷.

```
❶ type = nd_type(node);
--snip--
❷ switch (type) {
```

*Listing 2-4: This C `switch` statement looks at the type of each AST node.*

In this statement, `node` ❶ is a parameter passed into `iseq_compile_each`, and `nd_type` is a C macro that returns the type from the given node structure.

Now we'll look at how Ruby compiles function or method call nodes into YARV instructions using the receiver-arguments-function call pattern. First, search *compile.c* for the C `case` statement shown in Listing 2-5.

```
case NODE_CALL:
case NODE_FCALL:
case NODE_VCALL:{               /* VCALL: variable or call */
  /*
    call:  obj.method(...)
    fcall: func(...)
    vcall: func
  */
```

*Listing 2-5: This case of the switch compiles method calls in your Ruby code.*

`NODE_CALL` represents a real method call (like `10.times`), `NODE_FCALL` is a function call (like `puts`), and `NODE_VCALL` is a variable or function call. Skipping over some of the C

code details (including the optional SUPPORT_JOKE code used for implementing the goto statement), Listing 2-6 shows what Ruby does next to compile these AST nodes.

```
/* receiver */
if (type == NODE_CALL) {
❶    COMPILE(recv, "recv", node->nd_recv);
}
else if (type == NODE_FCALL || type == NODE_VCALL) {
❷    ADD_CALL_RECEIVER(recv, nd_line(node));
}
```

*Listing 2-6: This C code compiles the receiver value for a method call.*

Here, Ruby calls either COMPILE or ADD_CALL_RECEIVER as follows:

- In the case of real method calls (like NODE_CALL), Ruby calls COMPILE ❶ to recursively call into iseq_compile_each again, processing the next AST node down the tree that corresponds to the receiver of the method call or message. This will create YARV instructions to evaluate whatever expression was used to specify the target object.
- If there is no receiver (NODE_FCALL or NODE_VCALL), Ruby calls ADD_CALL_RECEIVER ❷, which creates a putself YARV instruction.

Next, as shown in Listing 2-7, Ruby creates YARV instructions to push each argument of the method/function call onto the stack.

```
/* args */
if (nd_type(node) != NODE_VCALL) {
❶    argc = setup_args(iseq, args, node->nd_args, &flag);
}
else {
❷    argc = INT2FIX(0);
}
```

*Listing 2-7: This snippet of C code compiles the arguments to every Ruby method call.*

For NODE_CALL and NODE_FCALL, Ruby calls into the setup_args function ❶, which will recursively call into iseq_compile_each again as needed in order to compile each argument to the method/function call. For NODE_VCALL, there are no arguments, so Ruby simply sets argc to 0 ❷.

Finally, Ruby creates YARV instructions to execute the actual method or function call, as shown here:

```
ADD_SEND_R(ret, nd_line(node), ID2SYM(mid),
           argc, parent_block, LONG2FIX(flag));
```

This C macro will create the new send YARV instruction, which will cause the actual method call to occur when YARV executes it.

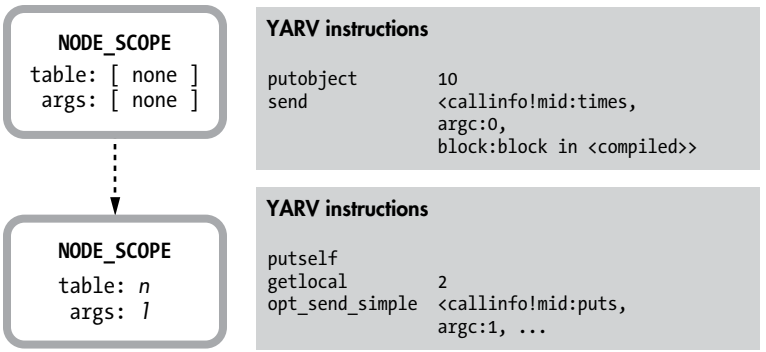


*Figure 2-14: How Ruby compiles a call to a block*

You can see the parent `NODE_SCOPE` at the top, along with the YARV code from Figure 2-12. Below that I've listed the YARV code compiled from the inner block's AST.

The key point here is that Ruby compiles each distinct scope in your Ruby program—methods, blocks, classes, or modules, for example—into a separate snippet of YARV instructions.

## Experiment 2-1: Displaying YARV Instructions

One easy way to see how Ruby compiles your code is with `RubyVM::InstructionSequence`, which gives you access to Ruby's YARV engine from your Ruby program! Like the Ripper tool, its use is very straightforward, as you can see in Listing 2-8.

```
code = <<END
puts 2+2
END
puts RubyVM::InstructionSequence.compile(code).disasm
```

*Listing 2-8: How to view the YARV instructions for `puts 2+2`*

The challenge lies in understanding what the output actually means. For example, Listing 2-9 shows the output for `puts 2+2`.

```
== disasm: <RubyVM::InstructionSequence:<compiled>@<compiled>>==========
❶ 0000 trace            1                                               (   1)
  0002 putself
  0003 putobject        2
  0005 putobject        2
  0007 opt_plus         <callinfo!mid:+, argc:1, ARGS_SKIP>
  0009 opt_send_simple  <callinfo!mid:puts, argc:1, FCALL|ARGS_SKIP>
❷ 0011 leave
```

*Listing 2-9: The YARV instructions for `puts 2+2`*

As you can see in Listing 2-9, the output contains all of the same instructions from Figures 2-5 to 2-8 and two new ones: `trace` ❶ and `leave` ❷. The `trace` instruction is used to implement the `set_trace_func` feature,[1] which will call a given function for each Ruby statement executed in your program. The `leave` function is like a return statement. The line numbers on the left show the position of each instruction in the bytecode array that the compiler actually produces.

`RubyVM::InstructionSequence` makes it easy to explore how Ruby compiles different Ruby scripts. For example, Listing 2-10 shows how to compile my `10.times do` example.

```
code = <<END
10.times do |n|
  puts n
end
END
puts RubyVM::InstructionSequence.compile(code).disasm
```

*Listing 2-10: Displaying the YARV instructions for a call to a block*

The output that I get now is shown below in Listing 2-11. Notice that the `send <callinfo!mid:times` YARV instruction shows `block:block in <compiled>` ❷, which indicates that I'm passing a block to the `10.times` method call.

```
❶ == disasm: <RubyVM::InstructionSequence:<compiled>@<compiled>>==========
== catch table
| catch type: break  st: 0002 ed: 0006 sp: 0000 cont: 0006
|------------------------------------------------------------------------
0000 trace            1                                               (   1)
0002 putobject        10
❷ 0004 send           <callinfo!mid:times, argc:0, block:block in <compiled>>
0006 leave
❸ == disasm: <RubyVM::InstructionSequence:block in <compiled>@<compiled>>=
== catch table
| catch type: redo   st: 0000 ed: 0011 sp: 0000 cont: 0000
| catch type: next   st: 0000 ed: 0011 sp: 0000 cont: 0011
|------------------------------------------------------------------------
local table (size: 2, argc: 1 [opts: 0, rest: -1, post: 0, block: -1] s3)
[ 2] n<Arg>
0000 trace            256                                             (   1)
0002 trace            1                                               (   2)
0004 putself
0005 getlocal_OP__WC__0 2
0007 opt_send_simple  <callinfo!mid:puts, argc:1, FCALL|ARGS_SKIP>
0009 trace            512                                             (   3)
0011 leave                                                            (   2)
```

*Listing 2-11: The YARV instructions for a call to a block and for the block itself*

1. For Ruby 2.x, the Ruby core team recommends using `TracePoint` instead of `set_trace_func`.

As you can see, Ruby displays the two YARV instruction snippets separately. The first corresponds to the global scope ❶ and the second to the inner block scope ❸.

## The Local Table

In Figures 2-3 through 2-14, you may have noticed that each `NODE_SCOPE` element in the AST contained information I labeled `table` and `args`. These values in the inner `NODE_SCOPE` structure contain information about the block's parameter `n` (see Figure 2-9 on page 39).

Ruby generated the information about this block parameter during the parsing process. As I discussed in Chapter 1, Ruby parses the block parameter along with the rest of my Ruby code using grammar rules. In fact, I showed the specific rule for parsing block parameters back in Figure 1-30 (page 21): `opt_block_param`.

Once Ruby's compiler runs, however, the information about the block parameter is copied out of the AST and into another data structure called the *local table*, saved nearby the newly generated YARV instructions. Each snippet of YARV instructions, each scope in your Ruby program, has its own local table.

Figure 2-15 shows the local table attached to the YARV instructions that Ruby generated for the sample block code from Listing 2-2.

```
... do |n|
  puts n
end
```

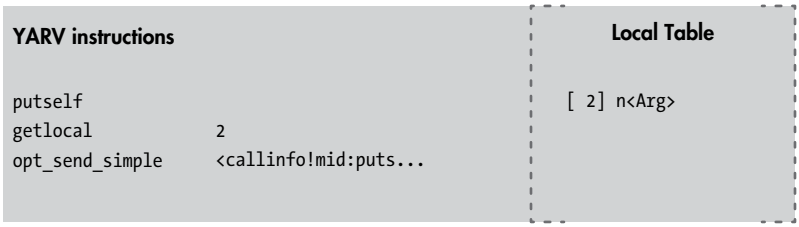


*Figure 2-15: A snippet of YARV instructions with a local table*

Notice on the right side of Figure 2-15 that Ruby has associated the number 2 with the block parameter `n`. As we'll see in Chapter 3, the YARV instructions that refer to `n` will use this index 2. The `getlocal` instruction is an example of this. The `<Arg>` notation indicates that this value is an argument to the block.

As it turns out, Ruby also saves information about local variables in this table, hence the name *local table*. Figure 2-16 shows the YARV instructions and local table Ruby will generate when compiling a method that uses one local variable and takes two arguments.

```
def add_two(a, b)
  sum = a+b
end
```

```
YARV instructions                                    Local Table

getlocal            4                                [ 2] sum
getlocal            3                                [ 3] b<Arg>
opt_plus            <callinfo!mid:+...               [ 4] a<Arg>
dup
setlocal            2
```

*Figure 2-16: This local table contains one local variable and two arguments.*

Here, you can see that Ruby lists all three values in the local table. As we'll see in Chapter 3, Ruby treats local variables and method arguments in the same way. (Notice that the local variable `sum` does not have the `<Arg>` label.)

Think of the local table as a key to help you understand what the YARV instructions do, similar to the legend on a map. As you can see in Figure 2-16, local variables have no label, but Ruby uses the following labels to describe different types of method and block arguments:

**<Arg>** A standard method or block argument

**<Rest>** An array of unnamed arguments that are passed together using a * (splat) operator

**<Post>** A standard argument that appears after the splat array

**<Block>** A Ruby proc object that is passed using the `&` operator

**<Opt=i>** A parameter defined with a default value. Internally, this value is a pointer to YARV instructions that set the default value. The local table does not contain the actual default values.

Understanding the information displayed by the local table can help you understand how Ruby's complex argument syntax works and how to take full advantage of the language.

To help you understand what I mean, let's look at how Ruby compiles a method call that uses an array of unnamed arguments, as shown Listing 2-12.

```
def complex_formula(a, b, *args, c)
  a + b + args.size + c
end
```

*Listing 2-12: A method that takes standard arguments and an array of unnamed arguments*

Here `a`, `b`, and `c` are standard arguments, and `args` is an array of other arguments that appear between `b` and `c`. Figure 2-17 shows how the local table saves all of this information.

As in Figure 2-16, `<Arg>` refers to a standard argument. But now Ruby uses `<Rest>` to indicate that value 3 contains the "rest" of the arguments and `<Post>` to indicate that value 2 contains the argument that appears after the unnamed array, the last one.

```
def complex_formula (a, b, *args, c)
  a + b + args.size + c
end
```

**YARV instructions**

```
getlocal        5
getlocal        4
opt_plus        <callinfo!mid:+...
getlocal        3
opt_size        <callinfo!mid:size...
opt_plus        <callinfo!mid:+...
getlocal        2
opt_plus        <callinfo!mid:+...
```

**Local Table**

```
[ 2] c<Post>
[ 3] args<Rest>
[ 4] b<Arg>
[ 5] a<Arg>
```

*Figure 2-17: Ruby saves information about special arguments in the local table.*

## Compiling Optional Arguments

As you probably know, you can make an argument optional by specifying a default value for it in the argument list. Later, Ruby will use the default value if you don't provide a value for that argument when you call the method or block. Listing 2-13 shows a simple example.

```
def add_two_optional(a, b = 5)
  sum = a+b
end
```

*Listing 2-13: A method that takes an optional argument*

If you provide a value for `b`, the method will use that value as follows:

```
puts add_two_optional(2, 2)
 => 4
```

But if you don't, Ruby will assign the default value of 5 to `b`:

```
puts add_two_optional(2)
 => 7
```

Ruby has a bit more work to do in this situation. Where does the default value go? Where does the Ruby compiler put it? Figure 2-18 shows how Ruby generates a few extra YARV instructions during the compile process that set the default value.

```
def add_two_optional (a, b = 5)
  sum = a+b
end
```

**YARV instructions**

```
putobject       5
setlocal        3
getlocal        4
getlocal        3
opt_plus        <callinfo!mid:+...
dup
setlocal        2
```

**Local Table**

```
[ 2] sum
[ 3] b<Opt=0>
[ 4] a<Arg>
```

*Figure 2-18: Ruby's compiler generates extra code to handle optional arguments.*

Ruby's compiler generates the bolded YARV instructions, `putobject` and `setlocal`, to set the value of `b` to 5 when you call the method. (As we'll see in Chapter 3, YARV will call these instructions if you don't provide a value for `b` but skip them if you do.) You can also see that Ruby lists the optional argument `b` in the local table as `b<Opt=0>`. The `0` here refers to YARV instructions that set the default value.

### Compiling Keyword Arguments

In Ruby 2.x, we can specify a name along with a default value for each method or block argument. Arguments written this way are known as *keyword arguments*. For example, Listing 2-14 shows the same argument `b` declared using Ruby 2.0's new keyword argument syntax.

```
def add_two_keyword(a, b: 5)
  sum = a+b
end
```

*Listing 2-14: A method that takes a keyword argument*

Now to provide a value for `b`, I need to use its name:

```
puts add_two_keyword(2, b: 2)
 => 4
```

Or, if I don't specify `b` at all, Ruby will use the default value:

```
puts add_two_keyword(2)
 => 7
```

How does Ruby compile keyword arguments? Figure 2-19 shows Ruby needs to add quite a bit of additional code to the method's YARV snippet.

```
def add_two_keyword (a, b: 5)
  sum = a+b
end
```

**YARV instructions**

```
getlocal          3
dup
putobject         :b
opt_send_simple   <callinfo!mid:key?...
branchunless      18
dup
putobject         :b
opt_send_simple   <callinfo!mid:delete...
setlocal          4
jump              22
putobject         5
setlocal          4
pop
getlocal          5
getlocal          4
opt_plus          <callinfo!mid:+...
dup
setlocal          2
```

**Local Table**

```
[ 2] sum
[ 3] ?
[ 4] b
[ 5] a<Arg>
```

*Figure 2-19: The Ruby compiler generates many more instructions to handle keyword arguments.*

The Ruby compiler generates all of the YARV instructions in bold—13 new instructions—to implement the keyword argument `b`. In Chapters 3 and 4, I'll cover how YARV works in detail and what these instructions actually mean, but for now, we can guess what's going on here:

- In the local table, we can see a new mystery value shown as `[ 3]?`.
- To the left of Figure 2-19, new YARV instructions call the `key?` and `delete` methods.

Which Ruby class contains the `key?` and `delete` methods? The `Hash`. Figure 2-19 shows evidence that Ruby must implement keyword arguments using an internal, hidden hash object. All of these additional YARV instructions automatically add some logic to my method that checks this hash for the argument `b`. If Ruby finds the value of `b` in the hash, it uses it. If not, it uses the default value of 5. The mystery element `[3]?` in the local table must be this hidden hash object.

## Experiment 2-2: Displaying the Local Table

Along with YARV instructions, `RubyVM::InstructionSequence` will also display the local table associated with each YARV snippet or scope. Finding and understanding the local table for your code will help you to understand what the corresponding YARV instructions do. In this experiment, we'll look at where the local table appears in the output generated by the `RubyVM::InstructionSequence` object.

Listing 2-15 repeats Listing 2-10 from Experiment 2-1.

```
code = <<END
10.times do |n|
  puts n
end
END

puts RubyVM::InstructionSequence.compile(code).disasm
```

*Listing 2-15: Displaying the YARV instructions for a call to a block*

And Listing 2-16 repeats the output we saw earlier in Experiment 2-1.

```
== disasm: <RubyVM::InstructionSequence:<compiled>@<compiled>>==========
== catch table
| catch type: break  st: 0002 ed: 0006 sp: 0000 cont: 0006
|------------------------------------------------------------------------
0000 trace            1                                               (   1)
0002 putobject        10
0004 send             <callinfo!mid:times, argc:0, block:block in <compiled>>
0006 leave
== disasm: <RubyVM::InstructionSequence:block in <compiled>@<compiled>>=
```

```
== catch table
| catch type: redo   st: 0000 ed: 0011 sp: 0000 cont: 0000
| catch type: next   st: 0000 ed: 0011 sp: 0000 cont: 0011
|------------------------------------------------------------------------
❶ local table (size: 2, argc: 1 [opts: 0, rest: -1, post: 0, block: -1] s3)
❷ [ 2] n<Arg>
0000 trace            256                                               (   1)
0002 trace            1                                                 (   2)
0004 putself
0005 getlocal_OP__WC__0 2
0007 opt_send_simple  <callinfo!mid:puts, argc:1, FCALL|ARGS_SKIP>
0009 trace            512                                               (   3)
0011 leave                                                              (   2)
```

*Listing 2-16: Along with the YARV instructions, `RubyVM::InstructionSequence` displays the local table.*

Just above the YARV snippet for the inner scope—the block—we see information about its local table at ❶. This displays the total size of the table (`size: 2`), the argument count (`argc: 1`), and other information about the types of parameters (`opts: 0, rest: -1, post: 0`).

The second line ❷ shows the actual contents of the local table. In this example, we have just one argument, `n`.

Listing 2-17 shows how to use `RubyVM::InstructionSequence` in the same way to compile my unnamed arguments example from Listing 2-12.

```
code = <<END
def complex_formula(a, b, *args, c)
  a + b + args.size + c
end
END

puts RubyVM::InstructionSequence.compile(code).disasm
```

*Listing 2-17: This method uses unnamed arguments with a splat operator.*

And Listing 2-18 shows the output.

```
❶ == disasm: <RubyVM::InstructionSequence:<compiled>@<compiled>>==========
0000 trace            1                                                 (   1)
0002 putspecialobject 1
0004 putspecialobject 2
0006 putobject        :complex_formula
0008 putiseq          complex_formula
❷ 0010 opt_send_simple  <callinfo!mid:core#define_method, argc:3, ARGS_SKIP>
0012 leave
== disasm: <RubyVM::InstructionSequence:complex_formula@<compiled>>=====
❸ local table (size: 5, argc: 2 [opts: 0, rest: 2, post: 1, block: -1] s0)
❹ [ 5] a<Arg>     [ 4] b<Arg>     [ 3] args<Rest> [ 2] c<Post>
0000 trace            8                                                 (   1)
0002 trace            1                                                 (   2)
```

```
0004 getlocal_OP__WC__0 5
0006 getlocal_OP__WC__0 4
0008 opt_plus         <callinfo!mid:+, argc:1, ARGS_SKIP>
0010 getlocal_OP__WC__0 3
0012 opt_size         <callinfo!mid:size, argc:0, ARGS_SKIP>
0014 opt_plus         <callinfo!mid:+, argc:1, ARGS_SKIP>
0016 getlocal_OP__WC__0 2
0018 opt_plus         <callinfo!mid:+, argc:1, ARGS_SKIP>
0020 trace            16                                              (   3)
0022 leave                                                            (   2)
```

*Listing 2-18: Displaying the YARV instructions for a call to a block*

The top YARV scope, around ❶, shows the instructions YARV uses to define a new method. Notice the call to `core#define_method` at ❷, an internal C function that YARV uses to create new Ruby methods. This corresponds to calling `def complex_formula` in my script. (I'll discuss how Ruby implements methods in more detail in Chapters 5, 6, and 9.)

Notice the local table for the lower YARV snippet at ❸. This line now shows more information about the unnamed arguments (`rest: 2`) and the last standard argument following them (`post: 1`). Finally, the line at ❹ shows the contents of the local table that I showed back in Figure 2-17.

## Summary

In this chapter, we learned how Ruby compiles our code. You may think of Ruby as a dynamic scripting language, but, in fact, it uses a compiler just like C, Java, and many other programming languages. The obvious difference is that Ruby's compiler runs automatically behind the scenes; you never need to worry about compiling your Ruby code.

We've learned that Ruby's compiler works by iterating through the AST produced by the tokenizing and parsing processes, generating a series of bytecode instructions along the way. Ruby translates your code from Ruby into a language tailored for the YARV virtual machine, and it compiles every scope or section of your Ruby program into a different snippet or set of these YARV instructions. Every block, method, lambda, or other scope in your program has a corresponding set of bytecode instructions.

We've also seen how Ruby handles different types of arguments. We were able to use the local table as a key or legend for understanding which YARV instructions accessed which arguments or local variables. And we saw how Ruby's compiler generates additional, special YARV instructions to handle optional and keyword parameters.

In Chapter 3, I'll begin to explain how YARV executes the instructions produced by the compiler—that is, how YARV executes your Ruby program.