3

HOW RUBY EXECUTES YOUR CODE

Now that Ruby has tokenized, parsed, and compiled your code, it's finally ready to execute it. But just how does it do that? We've seen how the Ruby compiler creates YARV (Yet Another Ruby Virtual Machine) instructions, but how does YARV actually run them? How does it track variables and return values and arguments? How does it implement `if` statements and other control structures?

Koichi Sasada and the Ruby core team designed YARV to use a stack pointer and a program counter—that is, to function like your computer's actual microprocessor. In this chapter, I'll examine the basics of YARV instructions; namely, how they pop arguments off of and push return values onto an internal stack. We'll also see how YARV keeps track of

your Ruby call stack along with its own internal stack. I'll explain how Ruby accesses local variables and how it can find variables farther down your call stack using dynamic access. We'll finish with a look at how Ruby implements special variables. In Chapter 4 I'll continue the discussion of YARV by examining how it implements control structures and method dispatch.

ROADMAP

YARV's Internal Stack and Your Ruby Stack

As we'll see in a moment, YARV uses a stack internally to track intermediate values, arguments, and return values. YARV is a stack-oriented virtual machine.

In addition to its own internal stack, YARV keeps track of your Ruby program's *call stack*, recording which methods call which other methods, functions, blocks, lambdas, and so on. In fact, YARV is not just a stack machine—it's a double-stack machine! It has to track the arguments and return values not only for its own internal instructions but also for your Ruby program.

Figure 3-1 shows YARV's basic registers and internal stack.

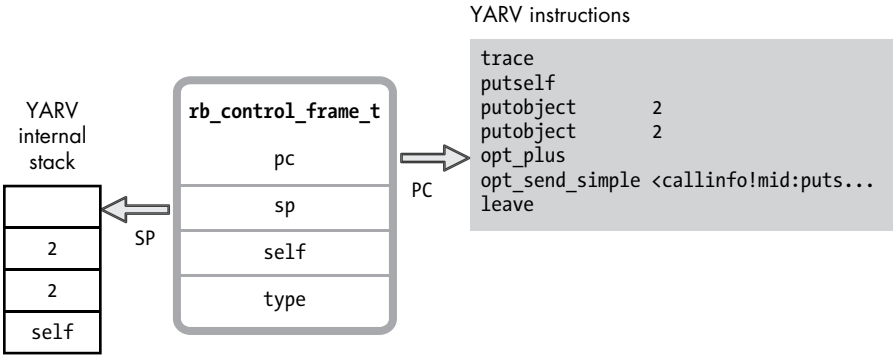

Figure 3-1: Some of YARV's internal registers, including the program counter and stack pointer

YARV's internal stack is on the left. The `SP` label is the *stack pointer*, or the location of the top of the stack. On the right are the instructions that YARV is executing. `PC` is the *program counter*, or the location of the current instruction.

You can see the YARV instructions that Ruby compiled from the `puts 2+2` example on the right side of Figure 3-1. YARV stores both the `SP` and `PC` registers in a C structure called `rb_control_frame_t`, along with a `type` field, the current value of Ruby's `self` variable, and some other values not shown here.

At the same time, YARV maintains another stack of these `rb_control_frame_t` structures, as shown in Figure 3-2.

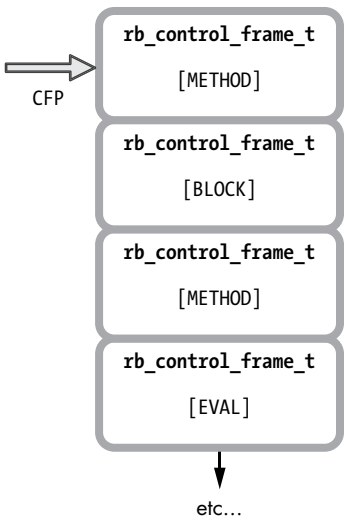

Figure 3-2: YARV keeps track of your Ruby call stack using a series of `rb_control_frame_t` structures.

This second stack of `rb_control_frame_t` structures represents the path that YARV has taken through your Ruby program, and YARV's current location. In other words, this is your Ruby call stack—what you would see if you ran `puts caller`.

The `CFP` pointer indicates the *control frame pointer*. Each stack frame in your Ruby program stack contains, in turn, a different value for the `self`, `PC`, and `SP` registers, as shown in Figure 3-1. The `type` field in each `rb_control_frame_t` structure indicates the type of code running at this level in your Ruby call stack. As Ruby calls into the methods, blocks, or other structures in your program, the type might be set to `METHOD`, `BLOCK`, or one of a few other values.

Stepping Through How Ruby Executes a Simple Script

In order to help you understand this a bit better, here are a couple of examples. I'll begin with the simple `2+2` example from Chapters 1 and 2, shown again in Listing 3-1.

```
puts 2+2
```

Listing 3-1: A one-line Ruby program that we'll execute as an example

This one-line Ruby script doesn't have a Ruby call stack, so I'll focus on the internal YARV stack for now. Figure 3-3 shows how YARV will execute this script, beginning with the first instruction, `trace`.

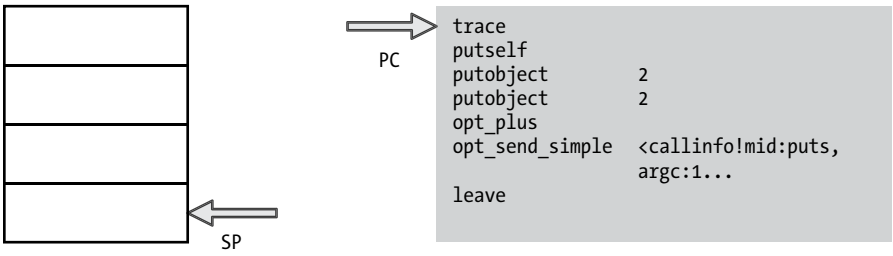

Figure 3-3: On the left is YARV's internal stack, and on the right is the compiled version of my `puts 2+2` program.

As you can see in Figure 3-3, YARV starts the program counter (`PC`) at the first instruction, and initially the stack is empty. Now YARV will execute the `trace` instruction, incrementing the `PC` register, as shown in Figure 3-4.

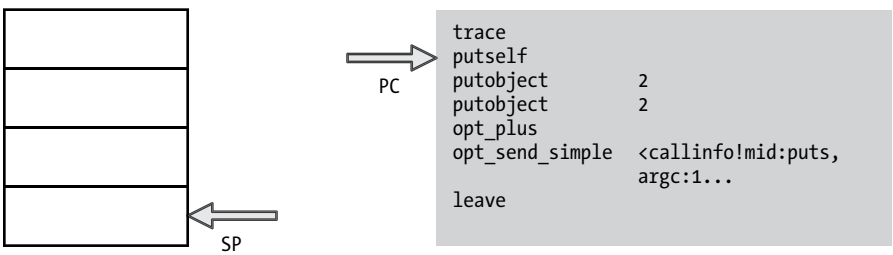

Figure 3-4: Ruby executes the first instruction, `trace`.

Ruby uses the `trace` instruction to support the `set_trace_func` feature. If you call `set_trace_func` and provide a function, Ruby will call it each time it executes a line of Ruby code.

Next, YARV executes `putself` and pushes the current value of `self` onto the stack, as shown in Figure 3-5.

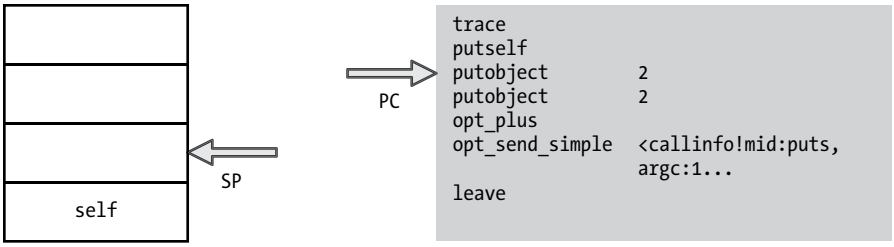

Figure 3-5: `putself` pushes the `top self` value onto the stack.

Because this simple script contains no Ruby objects or classes, the `self` pointer is set to the default `top self` object. This is an instance of the `Object` class that Ruby automatically creates when YARV starts. It serves as the receiver for method calls and the container for instance variables in the top-level scope. The `top self` object contains a single, predefined `to_s` method, which returns the string `main`. You can call this method by running the following command in the console:

```
$ ruby -e 'puts self'
 => main
```

YARV will use this `self` value on the stack when it executes the `opt_send_simple` instruction: `self` is the receiver of the `puts` method because I didn't specify a receiver for this method call.

Next, YARV executes `putobject 2`. It pushes the numeric value `2` onto the stack and increments the `PC` again, as shown in Figure 3-6.

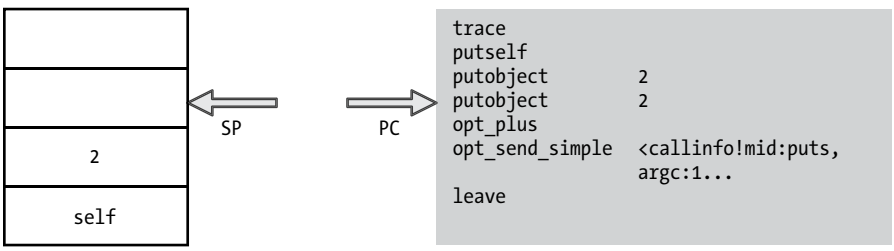

Figure 3-6: Ruby pushes the value `2` onto the stack, the receiver of the `+` method.

This is the first step of the receiver (arguments) operation pattern described in "How Ruby Compiles a Simple Script" on page 34. First, Ruby pushes the receiver onto the internal YARV stack. In this example, the `Fixnum` object `2` is the receiver of the message/method `+`, which takes a single argument, also a `2`. Next, Ruby pushes the argument `2`, as shown in Figure 3-7.

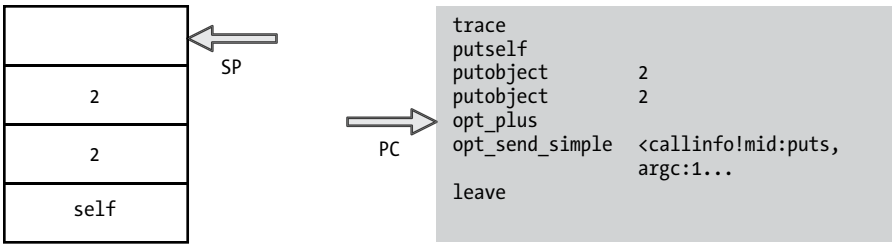

Figure 3-7: Ruby pushes another value `2` onto the stack, the argument of the + method.

Finally, Ruby executes the + operation. In this case, `opt_plus` is an optimized instruction that will add two values: the receiver and the argument, as shown in Figure 3-8.

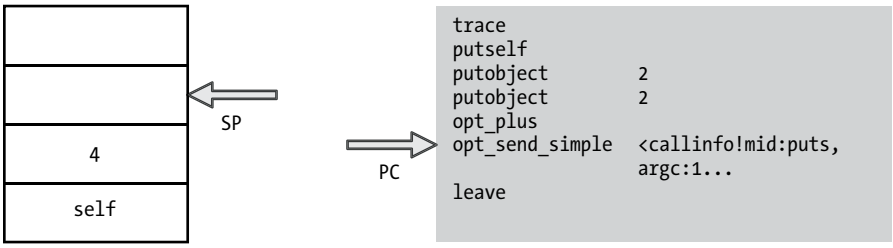

Figure 3-8: The `opt_plus` instruction calculates 2 + 2 = 4.

As you can see in Figure 3-8, the `opt_plus` instruction leaves the result, `4`, at the top of the stack. Now Ruby is perfectly positioned to execute the `puts` function call: The receiver `self` is first on the stack, and the single argument, `4`, is at the top of the stack. (I'll describe how method lookup works in Chapter 6.)

Next, Figure 3-9 shows what happens when Ruby executes the `puts` method call. As you can see, the `opt_send_simple` instruction leaves the return value, `nil`, at the top of the stack. Finally, Ruby executes the last instruction, `leave`, which finishes the execution of our simple, one-line Ruby program. Of course, when Ruby executes the `puts` call, the C code implementing the `puts` function will actually display the value `4` in the console output.

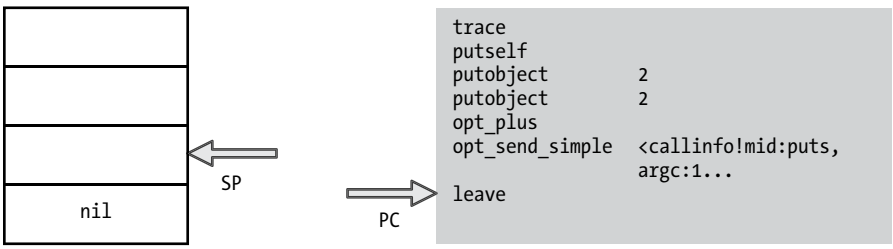

Figure 3-9: Ruby calls the `puts` method on the `top self` object.

Executing a Call to a Block

Now let's see how the Ruby call stack works. In Listing 3-2, a slightly more complicated example, you see a simple Ruby script that calls a block 10 times, printing out a string.

```
10.times do
  puts "The quick brown fox jumps over the lazy dog."
end
```

Listing 3-2: This example program calls a block 10 times.

Let's skip over a few steps and start where YARV is about to call the `times` method, as shown in Figure 3-10.

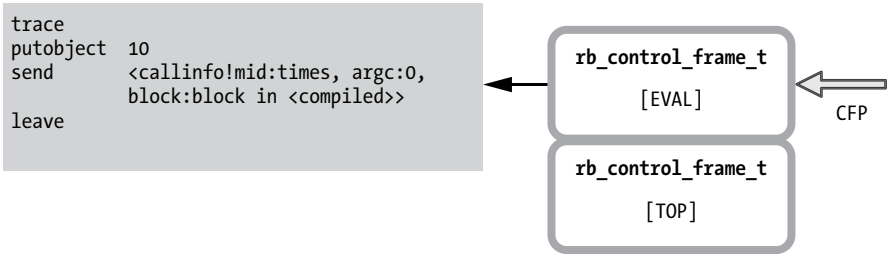

Figure 3-10: Every Ruby program starts with these two control frames.

On the left side of the diagram are the YARV instructions that Ruby is executing. On the right, you see two control frame structures.

At the bottom of the stack, you see a control frame with the type set to `TOP`. Ruby always creates this frame first when starting a new program. At the top of the stack, at least initially, a frame of type `EVAL` corresponds to the top level or main scope of the Ruby script.

Next, Ruby calls the `times` message on the `Fixnum` object `10`—the receiver of the `times` message. When it does so, it adds a new level to the control frame stack, as shown in Figure 3-11.

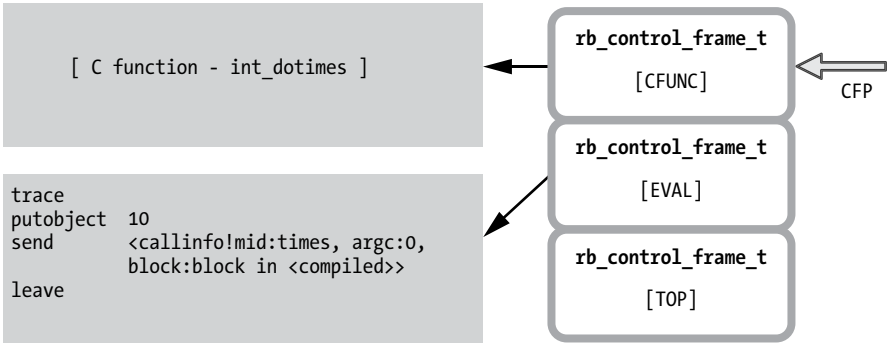

Figure 3-11: Ruby uses the `CFUNC` frame when you call built-in functions implemented in C.

This new entry (at the right of Figure 3-11) represents a new level in the program's Ruby call stack, and the CFP pointer has moved up to point at the new control frame structure. Also, notice that because the Integer#times method is built into Ruby, there are no YARV instructions for it. Instead, Ruby will call some internal C code to pop the argument 10 off the stack and call the provided block 10 times. Ruby gives this control frame a type of CFUNC.

Finally, Figure 3-12 shows what the YARV and control frame stacks will look like if we interrupt the program inside the inner block.

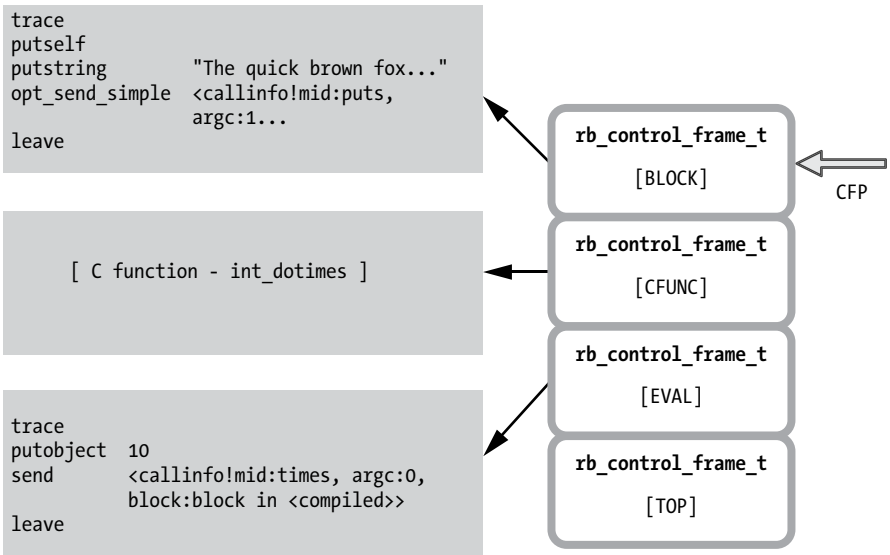

Figure 3-12: The CFP stack when we pause the code from Listing 3-2 inside the block

There will now be four entries, as follows, in the control frame stack on the right:

- The TOP and EVAL frames that Ruby always starts with
- The CFUNC frame for the call to `10.times`
- A BLOCK frame at the top of the stack that corresponds to the code running inside the block

TAKING A CLOSE LOOK AT A YARV INSTRUCTION

As it does with most other things, Ruby implements all YARV instructions, like `putobject` or `send`, using C code that is then compiled into machine language and executed directly by your hardware. Strangely, however, you won't find the C source code for each YARV instruction in a C source file. Instead, the Ruby core team put the YARV-instruction C code in a single large file called *insns.def*. Listing 3-3 shows a small snippet from *insns.def*, where Ruby implements the `putself` YARV instruction internally.

```
/**
  @c put
  @e put self.
  @j スタックに self をプッシュする。
 */
DEFINE_INSN
putself
()
()
(VALUE val)
{
❶   val = GET_SELF();
}
```

Listing 3-3: The definition of the `putself` YARV instruction

This doesn't look like C at all and, in fact, most of it is not. Instead, what you see here is a bit of C code (`val = GET_SELF()`) at ❶ that appears below a call to `DEFINE_INSN`.

It's not hard to figure out that `DEFINE_INSN` stands for *define instruction*. In fact, Ruby processes and converts the *insns.def* file into real C code during the Ruby build process, similar to the way that Bison converts the *parse.y* file into *parse.c*, as shown in Figure 3-13.

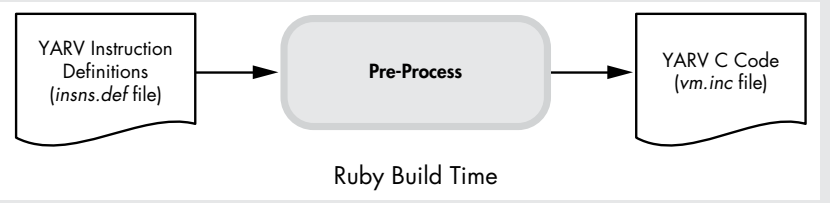

Figure 3-13: Ruby compiles the YARV-instruction definition script insns.def *into C code during the Ruby build process.*

continued

Ruby processes the *insns.def* file using Ruby: The build process first uses Ruby[1] to generate *vm.inc* and a few similar files. Then it uses these C source code files to compile *Miniruby*, a small version of Ruby, which later helps compile the complete version of Ruby. The other generated C files are related to encodings and C extension libraries.

Listing 3-4 shows what the snippet for `putself` looks like in *vm.inc* once Ruby has processed it.

```
INSN_ENTRY(putself){
{
  VALUE val;
  DEBUG_ENTER_INSN("putself");
❶ ADD_PC(1+0);
  PREFETCH(GET_PC());
  #define CURRENT_INSN_putself 1
  #define INSN_IS_SC()     0
  #define INSN_LABEL(lab)  LABEL_putself_##lab
  #define LABEL_IS_SC(lab) LABEL_##lab##_##t
  COLLECT_USAGE_INSN(BIN(putself));
{
#line 282 "insns.def"
❷     val = GET_SELF();
#line 408 "vm.inc"
  CHECK_VM_STACK_OVERFLOW(REG_CFP, 1);
❸ PUSH(val);
#undef CURRENT_INSN_putself
#undef INSN_IS_SC
#undef INSN_LABEL
#undef LABEL_IS_SC
  END_INSN(putself);}}}
```

Listing 3-4: The definition of `putself` is transformed into this C code during the Ruby build process.

The single line `val = GET_SELF()` appears in the middle of the listing at ❷. Above and below this line, Ruby calls a few different C macros to do various things, like add `1` to the program counter (`PC`) register at ❶ and push the `val` value onto the YARV internal stack at ❸. If you look through *vm.inc*, you'll see this same C code repeated over and over again for the definition of each YARV instruction.

The *vm.inc* C source code file, in turn, is included by the *vm_exec.c* file, which contains the primary YARV instruction loop that steps through the YARV instructions in your program one after another and calls the C code corresponding to each one.

1. That is, Ruby is required to build Ruby. This design is based on the assumption that Ruby developers have Ruby in their development environments. The public source distribution includes the generated *vm.inc*, so you do not need Ruby if you use it.

Experiment 3-1: Benchmarking Ruby 2.0 and Ruby 1.9 vs. Ruby 1.8

The Ruby core team introduced the YARV virtual machine with Ruby 1.9. Earlier versions of Ruby executed programs by directly stepping through the nodes of the *abstract syntax tree (AST)*. There was no compile step: Ruby just tokenized, parsed, and then immediately executed your code.

Ruby 1.8 worked just fine. In fact, for years it was the most commonly used version. Then why did the Ruby core team do all of the extra work required to write a compiler and a new virtual machine? Speed. Executing a compiled Ruby program using YARV is much faster than walking through the AST directly.

How much faster is YARV? Let's take a look! In this experiment, we'll measure how much faster Ruby 2.0 and 1.9 are compared to Ruby 1.8 by executing the very simple Ruby script shown in Listing 3-5.

```
i = 0
while i < ARGV[0].to_i
  i += 1
end
```

Listing 3-5: A simple test script for benchmarking Ruby 2.0 and Ruby 1.9 vs. Ruby 1.8

This script receives a count value from the command line via the `ARGV` array and then just iterates in a `while` loop counting up to that value. This Ruby script is very, very simple: By measuring the time it takes to execute this script for different values of `ARGV[0]`, we should get a good sense of whether executing YARV instructions is actually faster than iterating over AST nodes. (There are no database calls or other external code involved.)

We can use the Unix `time` command to measure how long it takes Ruby to iterate one time:

```
$ time ruby benchmark1.rb 1
ruby benchmark1.rb 1  0.02s user 0.00s system 92% cpu 0.023 total
```

ten times:

```
$ time ruby benchmark1.rb 10
ruby benchmark1.rb 10  0.02s user 0.00s system 94% cpu 0.027 total
```

and so on.

Figure 3-14 shows a plot of the measured times on a logarithmic scale for Ruby 1.8.7, 1.9.3, and 2.0.

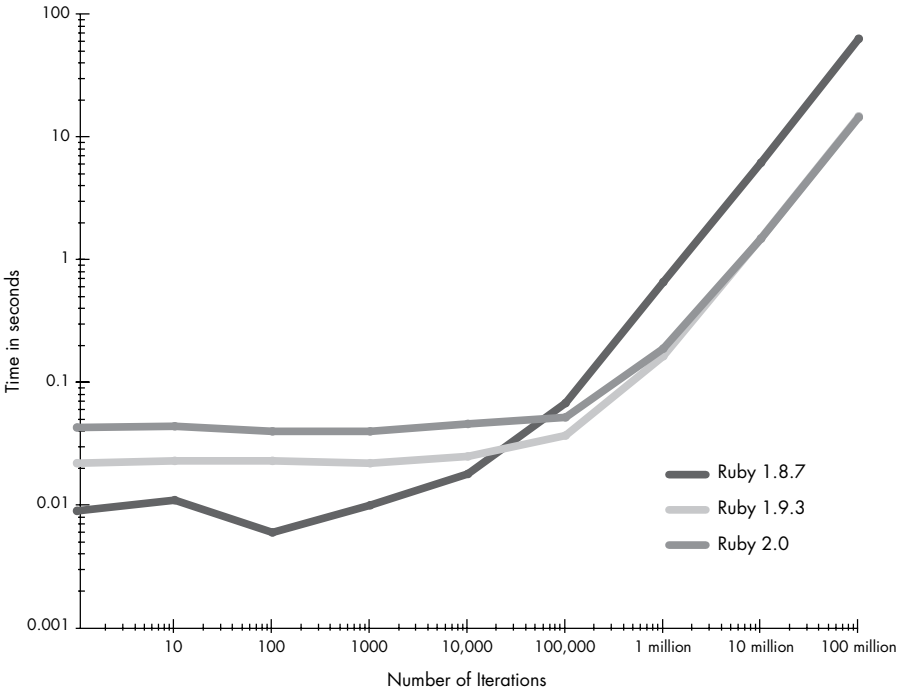

Figure 3-14: Performance of Ruby 1.8.7 vs. Ruby 1.9.3 and Ruby 2.0; time (in seconds) vs. number of iterations on a logarithmic scale

Looking at the chart, you can see that for short-lived processes, such as loops with a small number of iterations (see the left side of Figure 3-14), Ruby 1.8.7 is actually faster than Ruby 1.9.3 and 2.0 because there is no need to compile the Ruby code into YARV instructions. Instead, after tokenizing and parsing the code, Ruby 1.8.7 immediately executes it. The time difference between Ruby 1.8.7 and Ruby 1.9.3 and 2.0 at the left side of the chart, about 0.01 seconds, tells us how long it takes Ruby 1.9.3 or 2.0 to compile the script into YARV instructions. You can also see that Ruby 2.0 is actually a bit slower than Ruby 1.9.3 for short loops.

However, after about 11,000 iterations, Ruby 1.9.3 and 2.0 are faster. This crossover occurs when the additional speed provided by executing YARV instructions begins to pay off and make up for the additional time spent compiling. For long-lived processes, such as loops with a large number of iterations (see the right side of Figure 3-14), Ruby 1.9 and 2.0 are about 4.25 times faster! Also, we can see that Ruby 2.0 and 1.9.3 execute YARV instructions at exactly the same speed for many iterations.

This speed up doesn't look like much on the logarithmic chart in Figure 3-14, but notice what happens if we redraw the right side of this chart using a linear scale instead, as shown in Figure 3-15.

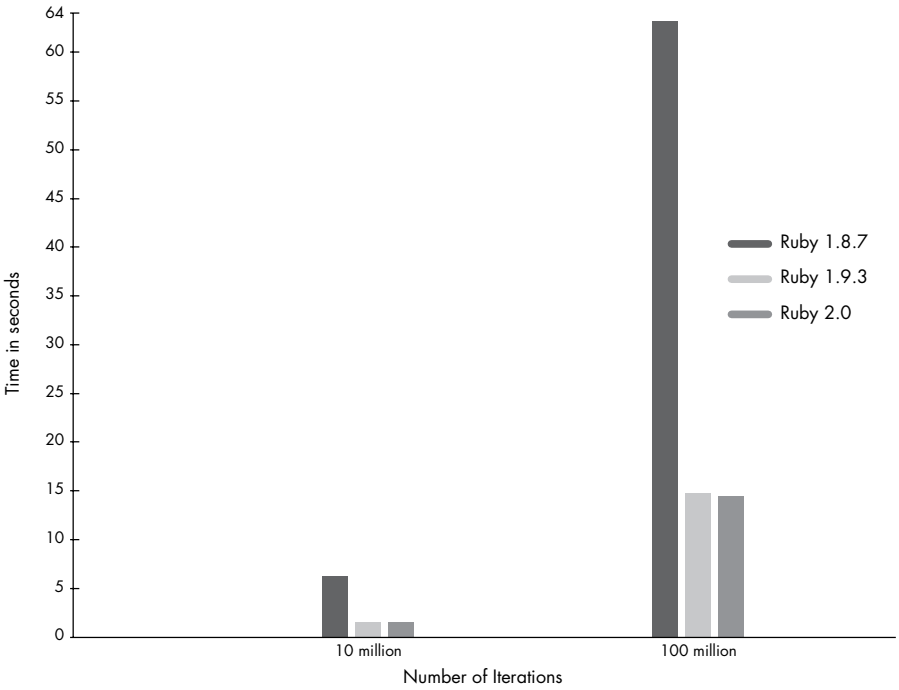

Figure 3-15: Performance of Ruby 1.8.7 vs. Ruby 1.9.3 vs. Ruby 2.0; time (in seconds) for 10 or 100 million iterations on a linear scale

The difference is dramatic! Executing this simple Ruby script using Ruby 1.9.3 or Ruby 2.0 with YARV is about 4.25 times faster than it is using Ruby 1.8.7 without YARV.

Local and Dynamic Access of Ruby Variables

In the previous section, we saw how Ruby maintained an internal stack used by YARV as well as your Ruby program's call stack. But something obvious was missing from both of the code examples: variables. Neither script used any Ruby variables. A more realistic example program would have used variables many times. How does Ruby handle variables internally? And where are they stored?

Ruby stores all of the values you save in variables on YARV's stack, along with the parameters to and return values from the YARV instructions. However, accessing these variables is not so simple. Internally, Ruby uses two very different methods for saving and retrieving a value you save in a variable: *local access* and *dynamic access*.

Local Variable Access

Whenever you make a method call, Ruby sets aside some space on the YARV stack for any local variables declared inside the method you are calling.

Ruby knows how many variables you are using by consulting the *local table* created for each method during the compilation step discussed in "The Local Table" on page 46.

For example, suppose we write the silly Ruby function you see in Figure 3-16.

```
def display_string
  str = "Local access."
  puts str
end
```

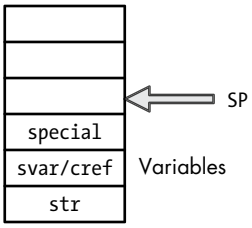

Figure 3-16: An example Ruby script that uses a local variable

The Ruby code is at the left of the figure; on the right is a diagram showing the YARV stack and stack pointer. You can see that Ruby stores the variables on the stack just under the stack pointer. (Notice that a space is reserved for the `str` value on the stack, three slots under `SP`, at `SP-3`.)

Ruby uses `svar/cref` to contain one of two things: either a pointer to a table of the special variables in the current method (values such as `$!` for *last exception message* or `$&` for *last regular expression match*) or to the current lexical scope. In this context, *lexical scope* indicates which class or module you are currently adding methods to. (In Experiment 3-2 we'll explore special variables in more detail, and I'll discuss lexical scope further in Chapter 6.) Ruby uses the first slot—the `special` variable—to track information related to blocks. (More in a moment when we discuss dynamic variable access.)

When the example code saves a value into `str`, Ruby just needs to write the value into that space on the stack, as shown in Figure 3-17.

```
def display_string
  str = "Local access."
  puts str
end
```

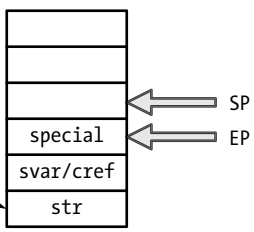

Figure 3-17: Ruby saves local variables on its stack near the environment pointer (EP).

To implement this internally, YARV uses another pointer similar to the stack pointer, called the `EP` or *environment pointer*. This points to where the local variables for the current method are located on the stack. Initially, `EP` is set to `SP-1`. Later on, the value of `SP` will change as YARV executes instructions, while the `EP` value will normally remain constant.

Figure 3-18 shows the YARV instructions that Ruby compiled my `display_string` function into.

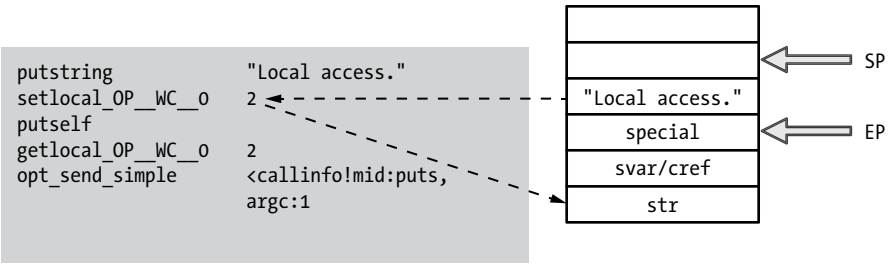

Figure 3-18: The `display_string` method compiled into YARV instructions

Ruby uses the `setlocal` YARV instruction to set the value of a local variable. However, instead of `setlocal` in Figure 3-18, I show an instruction called `setlocal_OP__WC__0`.

As it turns out, beginning with version 2.0, Ruby uses an optimized instruction with this confusing name instead of the simple `setlocal`. The difference is that Ruby 2.0 includes one of the parameters of the instruction, `0`, in the instruction name itself.

Internally, Ruby 2.0 calls this the *operand* optimization. (In the optimized instruction name, *OP* stands for *operand* and *WC* for *wildcard*.) In other words, `getlocal_OP__WC__0` is equivalent to `getlocal *, 0`, and `setlocal_OP__WC__0` is the same as `setlocal *, 0`. The instruction now requires only one parameter, as indicated by `*`. This trick allows Ruby 2.0 to save a bit of time because it doesn't need to pass the `0` argument separately.

But to keep things simple, let's ignore the operand optimization. Figure 3-19 repeats the YARV instructions for my example but shows `getlocal` and `setlocal` with the second operand listed normally.

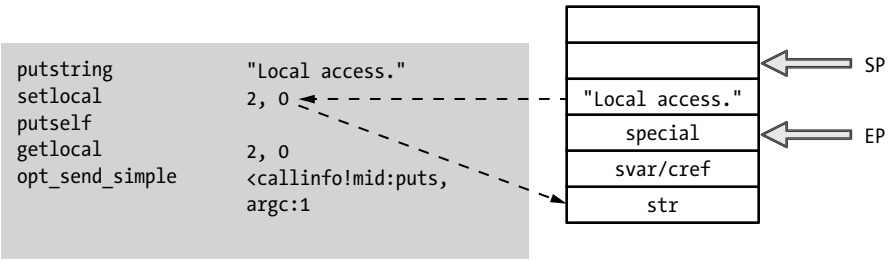

Figure 3-19: The compiled version of `display_string` shown without operand optimization

This a bit easier to understand. As you can see, first the `putstring` instruction saves the `Local access` string on top of the stack, incrementing the `SP` pointer. Then, YARV uses the `setlocal` instruction to get the value at the top of the stack and save it in the space allocated on the stack for the `str` local variable. The two dashed arrows on the left side of Figure 3-19 show the `setlocal` instruction copying the value. This type of operation is called *local variable access*.

To determine which variable to set, `setlocal` uses the `EP` pointer and the numerical index provided as the first parameter. In this example, that would be `address of str = EP-2`. We'll discuss what the second parameter, `0`, means in "Dynamic Variable Access" on page 71.

Next, for the call to `puts str`, Ruby uses the `getlocal` instruction, as shown in Figure 3-20.

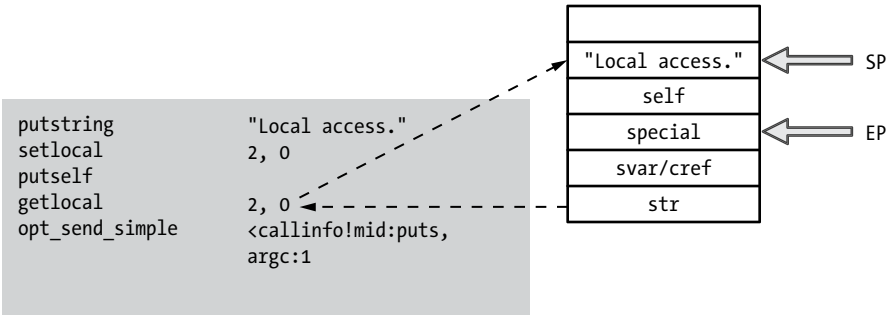

Figure 3-20: Getting the value of a local variable using `getlocal`

Here, Ruby has pushed the string value back onto the top of the stack, where it can be used as an argument for the call to the `puts` function. Again, the first parameter to `getlocal`, `2`, indicates which local variable to access. Ruby uses the local table for this snippet at compile time to find out `2` corresponds to the variable `str`.

Method Arguments Are Treated Like Local Variables

Passing in a *method argument* works the same way as accessing a local variable, as shown in Figure 3-21.

```
def display_string(str)
  puts str
end
```

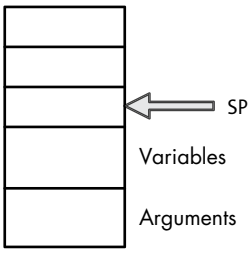

Figure 3-21: Ruby stores method arguments on the stack just like local variables.

Method arguments are essentially the same as local variables. The only difference between the two is that the calling code pushes the arguments onto the stack before the method call even occurs. In this example there are no local variables, but the single argument appears on the stack just like a local variable, as shown in Figure 3-22.

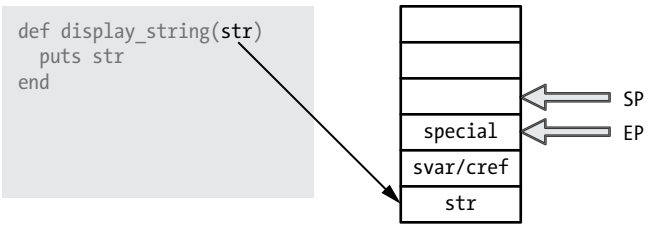

Figure 3-22: The calling code saves the argument values before the method is called.

Dynamic Variable Access

Now let's see how dynamic variable access works and what that `special` value is. Ruby uses dynamic access when you use a variable that's defined in a different scope—for example, when you write a block that references values in the surrounding code. Listing 3-6 shows an example.

```
def display_string
  str = "Dynamic access."
  10.times do
    puts str
  end
end
```

Listing 3-6: The code inside the block accesses `str` in the surrounding method.

Here, `str` is a local variable in `display_string`. As you can see in Figure 3-23, Ruby will save `str` using the `setlocal` instruction in just the same way we saw in Figure 3-18.

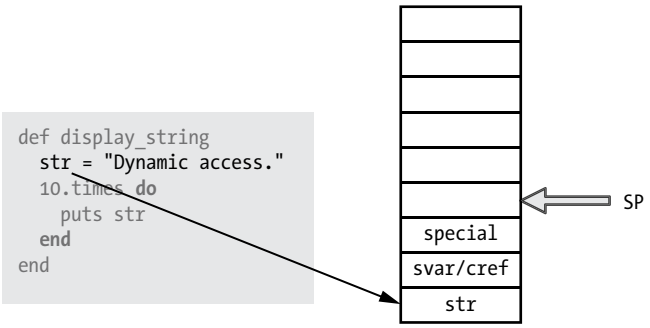

Figure 3-23: Ruby saves the value of the `str` local variable on the stack as usual.

Next, Ruby will call the `10.times` method, passing a block in as an argument. Let's step through the process of calling a method with a block.

Figure 3-24 shows the same process we saw in Figures 3-10, 3-11, and 3-12 but with more details about YARV's internal stack.

```
def display_string
  str = "Dynamic access."
  10.times do
    puts str
  end
end
```

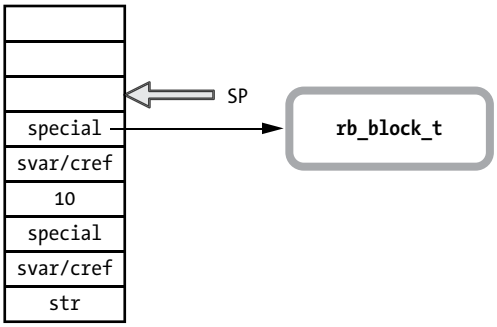

Figure 3-24: When Ruby calls a method passing in a block, it saves a pointer to a new `rb_block_t` structure as the special value in the new stack frame.

Notice the value `10` on the stack: This is the actual receiver of the `times` method. Notice too that Ruby has created a new stack frame with `svar/cref` and `special` above the value `10` for the C code that implements `Integer#times` to use. Because we passed a block into the method call, Ruby saves a pointer to this block in the `special` variable in the new stack frame. Each frame on the YARV stack corresponding to a method call tracks whether there was a block argument using this `special` variable. (I'll discuss blocks and the `rb_block_t` structure in more detail in Chapter 8.)

Now the `Integer#times` method yields to or calls the block's code 10 times. Figure 3-25 shows how the YARV stack appears when Ruby is executing the code inside the block.

```
def display_string
  str = "Dynamic access."
  10.times do
    puts str
  end
end
```

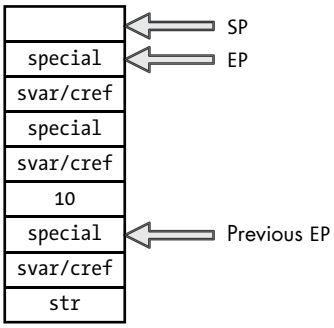

Figure 3-25: How YARV's stack would appear if we halted execution inside the block

Just as we saw in Figures 3-17 through 3-22, Ruby sets `EP` to point to the location of the `special` value in each stack frame. Figure 3-25 shows one

value of EP for the new stack frame used by the block near the top of the stack and a second value of EP in the original method's stack frame near the bottom. In Figure 3-25 this second pointer is labeled *Previous EP*.

Now, what happens when Ruby executes the `puts str` code inside the block? Ruby needs to obtain the value of the local variable `str` and pass it to the `puts` function as an argument. But notice in Figure 3-25 that `str` is located farther down the stack. It's not a local variable inside the block; rather, it's a variable in the surrounding method, `display_string`. How does Ruby obtain the value from farther down the stack while executing code inside the block?

This is where dynamic variable access comes in and why Ruby needs those `special` values in each stack frame. Figure 3-26 shows how dynamic variable access works.

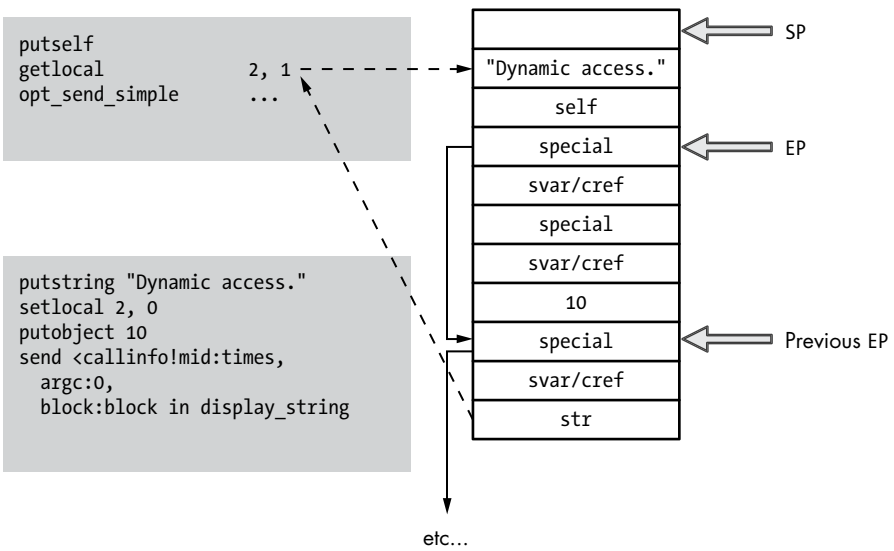

Figure 3-26: Ruby using dynamic variable access to obtain the value of `str` from farther down the stack

The dashed arrows indicate dynamic variable access: The `getlocal` YARV instruction copies the value of `str` from the lower stack frame (from the parent or outer Ruby scope) to the top of the stack, where the block can access it. Notice how the `EP` pointers form a kind of ladder that Ruby can climb to access the local variables in the parent scope, the grandparent scope, and so on.

In the `getlocal 2, 1` call in Figure 3-26, the second parameter, `1`, tells Ruby where to find the variable. In this example, Ruby will follow the ladder of `EP` pointers one level down the stack to find `str`. That is, `1` means step once from the block's scope to the surrounding method's scope.

Listing 3-7 shows another example of dynamic variable access.

```
def display_string
  str = "Dynamic access."
  10.times do
    10.times do
      puts str
    end
  end
end
```

Listing 3-7: In this example, Ruby would step two levels down the stack to find `str` using dynamic variable access.

If I had two nested blocks, as in Listing 3-7, Ruby would have used `getlocal 2, 2` instead of `getlocal 2, 1`.

CLIMBING THE ENVIRONMENT POINTER LADDER IN C

Let's look at the actual C implementation of `getlocal`. As it does with most YARV instructions, Ruby implements `getlocal` in the *insns.def* code file, using the code shown in Listing 3-8.

```
/**
  @c variable
  @e Get local variable (pointed by `idx' and `level').
     'level' indicates the nesting depth from the current block.
  @j level, idx で指定されたローカル変数の値をスタックに置く。
     level はブロックのネストレベルで、何段上かを示す。
 */
DEFINE_INSN
getlocal
(lindex_t idx, rb_num_t level)
()
(VALUE val)
{
    int i, lev = (int)level;
❶   VALUE *ep = GET_EP();

    for (i = 0; i < lev; i++) {
❷       ep = GET_PREV_EP(ep);
    }
❸   val = *(ep - idx);
}
```

Listing 3-8: The C implementation of the `getlocal` YARV instruction

First, the `GET_EP` macro ❶ returns the `EP` from the current scope. (This macro is defined in the *vm_insnhelper.h* file along with a number of other macros related to

YARV instructions.) Next, Ruby iterates over the EP pointers, moving from the current to the parent scope and then from the parent to the grandparent scope by repeatedly dereferencing the EP pointers. Ruby uses the GET_PREV_EP macro at ❷ (also defined in *vm_insnhelper.h*) to move from one EP to another. The level parameter tells Ruby how many times to iterate or how many rungs of the ladder to climb.

Finally, Ruby obtains the target variable using the idx parameter at ❸, which is the index of the target variable. As a result, this line of code gets the value from the target variable.

```
val = *(ep - idx);
```

This code means the following:

- Start from the address of the EP for the target scope ep, obtained previously from the GET_PREV_EP iterations.
- Subtract idx from this address. The integer value idx gives getlocal the index of the local variable that you want to load from the local table. In other words, it tells getlocal how far down the stack the target variable is located.
- Get the value from the YARV stack at the adjusted address.

Therefore, in the call to getlocal in Figure 3-26, YARV will take the EP from the scope one level down on the YARV stack and subtract the index value str (in this case, 2) to obtain a pointer to the str variable.

```
getlocal 2, 1
```

Experiment 3-2: Exploring Special Variables

In Figures 3-16 through 3-26, I showed you a value called svar/cref in the EP-1 position on the stack. What are these two values, and how can Ruby save two values in one location on the stack? For that matter, why does it do this? Let's find out.

Usually, the EP-1 slot in the stack will contain the svar value, which is a pointer to a table of any special variables defined in this stack frame. In Ruby the term *special variables* refers to values that Ruby automatically creates as a matter of convenience, based on the environment or on recent operations. For example, Ruby sets $* to the ARGV array and $! to the last exception raised.

All special variables begin with the dollar sign ($) character, which usually indicates a global variable. Does that mean that special variables are global variables? If so, then why does Ruby save a pointer to them on the stack?

To answer this question, let's create a simple Ruby script to match a string using a regular expression.

```
/fox/.match("The quick brown fox jumped over the lazy dog.\n")
puts "Value of $& in the top level scope: #{$&}"
```

Here I match the word `fox` in the string using a regex, and then I print the matching string using the `$&` special variable. Here's the output I get running this at the console.

```
$ ruby regex.rb
Value of $& in the top level scope: fox
```

Listing 3-9 shows another example, this time searching for the same string twice: first in the top-level scope and then again from inside a method call.

```
str = "The quick brown fox jumped over the lazy dog.\n"
❶ /fox/.match(str)

def search(str)
❷   /dog/.match(str)
❸   puts "Value of $& inside method: #{$&}"
end
search(str)

❹ puts "Value of $& in the top level scope: #{$&}"
```

Listing 3-9: Referring to $& from two different scopes

This is simple Ruby code, but it still may be a bit confusing. Here's how this works:

- We search the string in the top scope for `fox` at ❶. This matches and saves `fox` into the `$&` special variable.
- We call the `search` method and search for `dog` at ❷. I immediately print the match using the same `$&` variable inside the method at ❸.
- Finally, we return to the top-level scope and print the value of `$&` again at ❹.

Running this test gives the following output.

```
$ ruby regex_method.rb
Value of $& inside method: dog
Value of $& in the top level scope: fox
```

This is what we expect, but consider the following for a moment. The `$&` variable is obviously not global because it has different values at different places in my Ruby script. Ruby preserves the value of `$&` from the top-level scope when executing the `search` method, which allows me to print the matching word `fox` from the original search. Ruby provides for this behavior by saving a separate set of special variables at each level of the stack using the `svar` value, as shown in Figure 3-27.

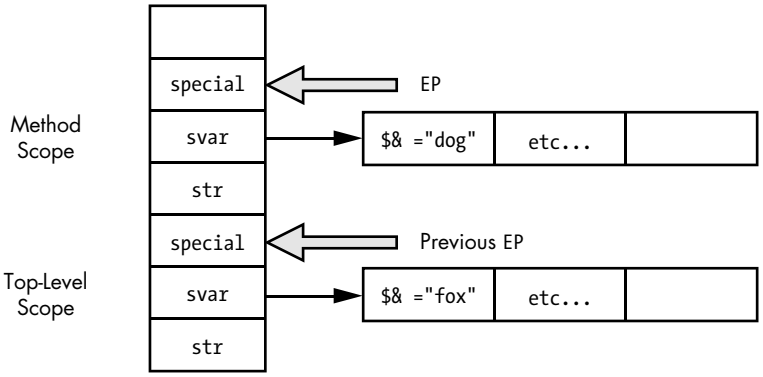

Figure 3-27: Each stack frame has its own set of special variables.

Notice that Ruby saved the `fox` string in a table referred to by the `svar` pointer for the top-level scope and saved the `dog` string in a different table for the inner-method scope. Ruby finds the proper special variable table using the `EP` pointer for each stack frame.

Ruby saves actual global variables (variables you define using a dollar sign prefix) in a single, global hash table. Regardless of where you save or retrieve the value of a normal global variable, Ruby accesses the same global hash table.

Now for one more test: What if I perform the search inside a block and not a method? Listing 3-10 shows this new search.

```
str = "The quick brown fox jumped over the lazy dog.\n"
/fox/.match(str)

2.times do
  /dog/.match(str)
  puts "Value of $& inside block: #{$&}"
end

puts "Value of $& in the top-level scope: #{$&}"
```

Listing 3-10: Displaying the value of $& from inside a block

Here's the output I get at the console this time.

```
$ ruby regex_block.rb
Value of $& inside block: dog
Value of $& inside block: dog
Value of $& in the top-level scope: dog
```

Notice that now Ruby has overwritten the value of `$&` in the top scope with the matching word `dog` from the search I performed inside the block! This is by design: Ruby considers the top-level and inner-block scope to be the same with regard to special variables. This is similar to how dynamic variable access works; we expect variables inside the block to have the same values as those in the parent scope.

Figure 3-28 shows how Ruby implements this behavior.

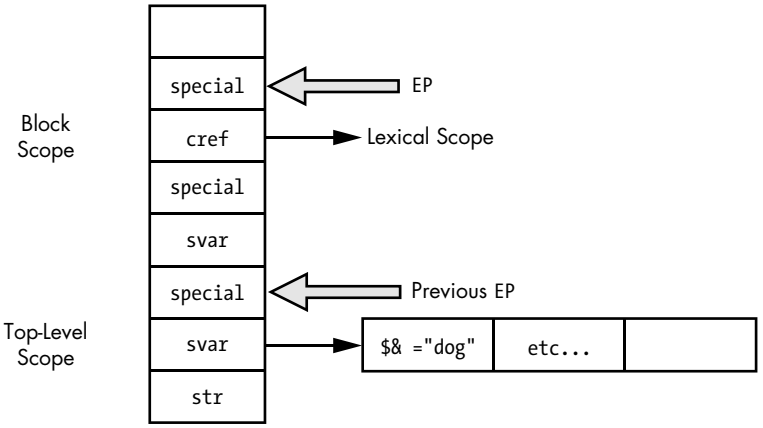

Figure 3-28: Ruby uses the `EP-1` stack position for `cref` in blocks and for `svar` otherwise.

As you can see in Figure 3-28, Ruby has just a single special variable table for the top-level scope. It finds the special variables using the previous `EP` pointer, which points to the top-level scope. Inside the block scope (because there is no need for a separate copy of the special variables), Ruby takes advantage of the `EP-1` open slot and saves the value `cref` there instead. Ruby uses the `cref` value to keep track of which lexical scope this block belongs to. *Lexical scope* refers to a section of code within the syntactical structure of your program and is used by Ruby to look up constant values. (See Chapter 6 for more on lexical scope.) Specifically, Ruby uses the `cref` value here to implement metaprogramming API calls, such as `eval` and `instance_eval`. The `cref` value indicates whether the given block should be executed in a different lexical scope compared to the parent scope. (See "instance_eval Creates a Singleton Class for a New Lexical Scope" on page 243.)

A DEFINITIVE LIST OF SPECIAL VARIABLES

One place to find an accurate list of all the special variables that Ruby supports is the C source itself. For example, Listing 3-11 is a piece of Ruby's C source code that tokenizes your Ruby program, as snipped from the `parser_yylex` function located in *parse.y*:

```
❶ case '$':
  lex_state = EXPR_END;
  newtok();
  c = nextc();
❷ switch (c) {
❸   case '_':             /* $_: last read line string */
      c = nextc();
      if (parser_is_identchar()) {
          tokadd('$');
          tokadd('_');
          break;
      }
      pushback(c);
      c = '_';
      /* fall through */
❹   case '~':             /* $~: match-data */
    case '*':             /* $*: argv */
    case '$':             /* $$: pid */
    case '?':             /* $?: last status */
    case '!':             /* $!: error string */
    case '@':             /* $@: error position */
    case '/':             /* $/: input record separator */
    case '\\':            /* $\: output record separator */
    case ';':             /* $;: field separator */
    case ',':             /* $,: output field separator */
    case '.':             /* $.: last read line number */
    case '=':             /* $=: ignorecase */
    case ':':             /* $:: load path */
    case '<':             /* $<: reading filename */
    case '>':             /* $>: default output handle */
    case '\"':            /* $": already loaded files */
      tokadd('$');
      tokadd(c);
      tokfix();
      set_yylval_name(rb_intern(tok()));
      return tGVAR;
```

Listing 3-11: Consulting parse.y *is a good way to find a definitive list of Ruby's many special variables.*

Notice at ❶ that Ruby matches a dollar sign character (`$`). This is part of the large C `switch` statement that tokenizes your Ruby code—the process I discussed in "Tokens: The Words That Make Up the Ruby Language" on page 4. This is

continued

followed by an inner `switch` statement at ❷ that matches on the following character. Each of these characters and each of the `case` statements that follow (at ❸ and after ❹) correspond to one of Ruby's special variables.

Just a bit farther down in the function, more C code (see Listing 3-12) parses other special variable tokens that you write in your Ruby code, such as `$&` and related special variables.

```
❶ case '&':                /* $&: last match */
  case '`':                /* $`: string before last match */
  case '\'':               /* $': string after last match */
  case '+':                /* $+: string matches last paren. */
    if (last_state == EXPR_FNAME) {
        tokadd('$');
        tokadd(c);
        goto gvar;
    }
    set_yylval_node(NEW_BACK_REF(c));
    return tBACK_REF;
```

Listing 3-12: These `case` statements correspond to Ruby's regex-related special variables.

At ❶ you can see four more `case` statements corresponding to the special variables `$&`, `$``, `$/`, and `$+`, all related to regular expressions.

Finally, the code in Listing 3-13 tokenizes `$1`, `$2`, and so on, producing the special variables that return the nth back reference from the last regular expression operation.

```
❶ case '1': case '2': case '3':
  case '4': case '5': case '6':
  case '7': case '8': case '9':
    tokadd('$');
❷   do {
        tokadd(c);
        c = nextc();
    } while (c != -1 && ISDIGIT(c));
    pushback(c);
    if (last_state == EXPR_FNAME) goto gvar;
    tokfix();
    set_yylval_node(NEW_NTH_REF(atoi(tok()+1)));
    return tNTH_REF;
```

Listing 3-13: This C code tokenizes Ruby's nth back reference special variables: `$1`, `$2`, and so forth.

The `case` statements at ❶ match the numerical digits 1 through 9, while the C `do...while` loop at ❷ continues to process digits until an entire number is read in. This allows you to create special variables with multiple digits, such as `$12`.

Summary

We've covered a lot of ground in this chapter. We began by looking at how Ruby keeps track of two stacks: an internal stack YARV uses and your Ruby call stack. Next, we saw how YARV executed two simple Ruby programs: one that calculated 2 + 2 = 4 and another that called a block 10 times. In Experiment 3-1 we learned that executing YARV instructions in Ruby 2.0 and 1.9 is almost four times faster than in Ruby 1.8, which executes your program directly from the AST.

We moved on to look at how Ruby saves variables on the internal YARV stack using two methods: local and dynamic variable access. We also saw how method arguments are handled by Ruby in just the same way as local variables. In Experiment 3-2 we finished with a look at how Ruby handles special variables.

When you run any Ruby program, you are actually using a virtual machine designed specifically to execute Ruby programs. By examining how this machine works on a detailed level, we've acquired a deeper understanding of how the Ruby language works and, for example, what happens when you call a method or save a value in a local variable. In Chapter 4 we'll continue to explore this virtual machine by looking at how control structures work and at YARV's method dispatch process.