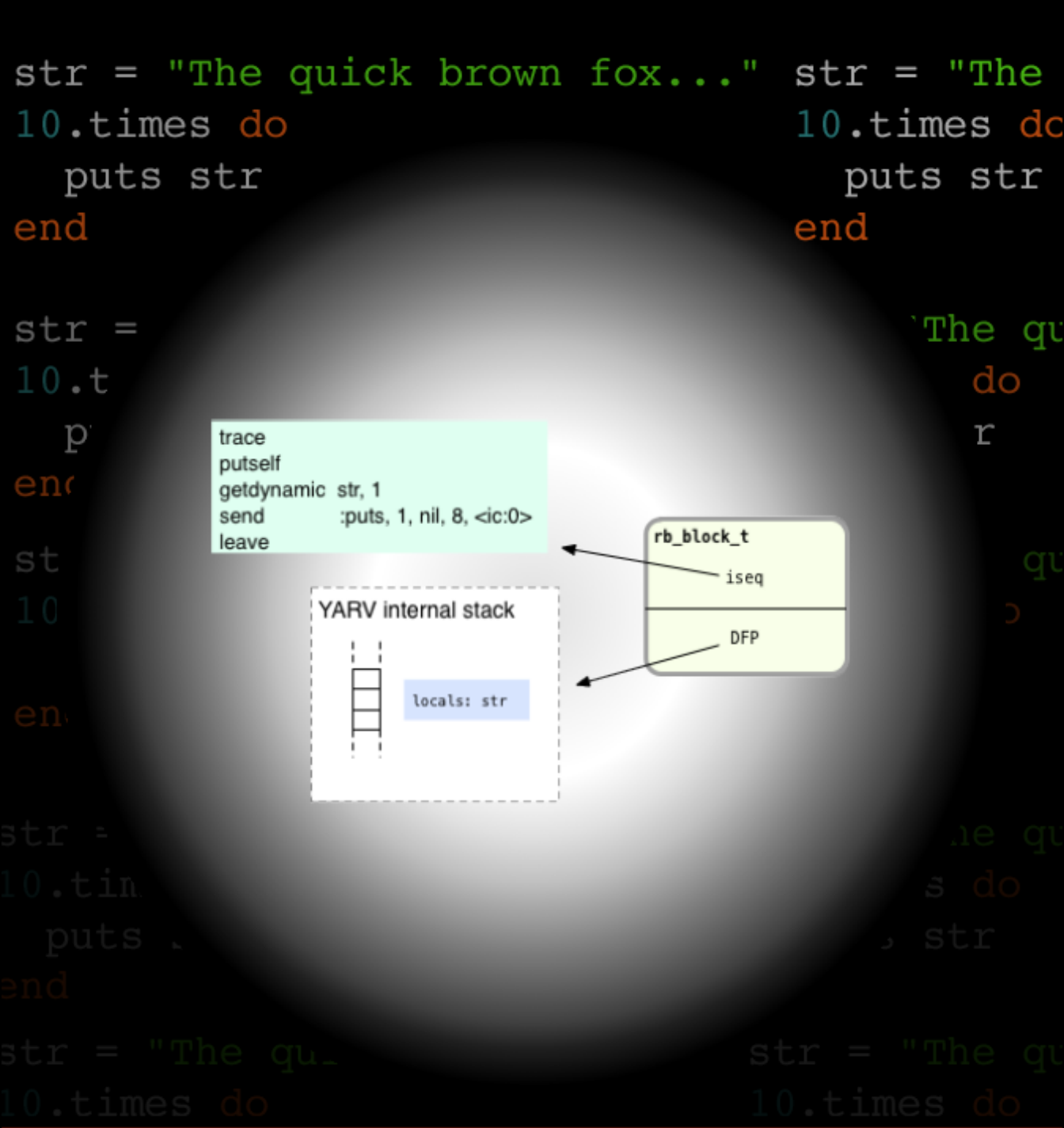


Ruby Under a Microscope

Learning Ruby Internals Through Experiment

By Pat Shaughnessy

Ruby Under a Microscope

Learning Ruby Internals Through Experiment

Version 1.1 - November 11, 2012

Pat Shaughnessy

Acknowledgments

I could never have finished a project like this without the support of many different people!

First of all, thanks to Satty Bhens and everyone at McKinsey for giving me the flexibility to write a book and keep my day job at a great company. Alex Rothenberg and Daniel Higginbotham gave me invaluable advice, suffered through reading many early drafts and helped me throughout the process. Alex also proofread the rough draft, and Daniel put together a fantastic landing page for the book… thank you guys! Special thanks to Xavier Noria who took an interest in the project early on, gave me the inspiration behind Experiment 3-3 and then gave me fantastic feedback on the entire rough draft. Thank you also to Santiago Pastorino who reviewed the rough draft as well. Also thank you to Jill Caporrimo, Prajakta Thakur, Yvannova Montalvo, Divya Ganesh and Yanwing Wong – my "proofreading SWAT team" – self publishing would have been much harder without your help. Finally, without the constant encouragement and support Peter Cooper has given me this year, I probably never would have even attempted to write this book… thank you Peter.

Dedicated to my wife Cristina, my daughter Ana and my son Liam – thanks for supporting me all along. Special thanks to Ana for the cover art!

Table of Contents

Experiments:

Preface

My approach in this book: theory and experiment

> *"It doesn't matter how beautiful your theory is, it doesn't matter how smart you are. If it doesn't agree with experiment, it's wrong." – Richard Feynman*

In *Ruby Under a Microscope* I'm going to teach you how Ruby works internally. I'll use a series of simple, easy to understand diagrams that will show you what is happening on the inside when you run a Ruby program. Like a physicist or chemist, I've developed a theory about how things actually work based on many hours of research and study. I've done the hard work of reading and understanding Ruby's internal C source code so you don't have to.

But, like any scientist, I know that theory is worthless without some hard evidence to back it up. Therefore after explaining some aspect of Ruby internals, some feature or behavior of the language, I'll perform an experiment to prove that my theory was correct. To do this I'll use Ruby to test itself! I'll run some small Ruby test scripts to see whether they produce the expected output or whether they run as fast or as slowly as I expect. We'll find out if Ruby actually behaves the way my theory says it should. Since these experiments are written in simple Ruby, you can try them yourself.

A travel journal

Before teaching you how Ruby works internally, I had to learn it myself. To do this, I went on a six month journey through MRI Ruby's internal implementation. I also took a few side trips to visit JRuby and Rubinius. I started by simply reading and studying Ruby's C source code; later I continued by looking at how it actually functions by setting breakpoints and stepping through the code using the GDB debugger. Finally, I modified and recompiled the C source code to be sure I thoroughly understood how it worked. Often I added "printf" statements to write out debug information; occasionally I changed the code directly to see what would happen if it were written differently.

Ruby Under a Microscope is my travel journal; I've written everything I've learned during this long journey. I hope the following pages give you the same sense of beauty and excitement which I found and felt as I discovered one amazing thing after another.

Ruby's internal C source code is like a foreign country where people speak a language you don't understand. At first it's difficult to find your way and understand the people around you, but if you take the time to learn something of the local language you can eventually come to know the fascinating people, places, food and culture from what was previously uncharted territory.

What this book is not

Ruby Under a Microscope is not a beginner's guide to learning Ruby. Instead, in *Ruby Under a Microscope* I assume you already know Ruby and use it on a daily basis. There are many great books on the market that teach Ruby far better than I ever could.

Ruby Under a Microscope is also not a newer, updated version of the Ruby Hacking Guide. As the name implies, the Ruby Hacking Guide is a guide for C programmers who want to understand Ruby's internal C implementation at a detailed level. It's an invaluable resource for the Ruby community and required reading for those who want to read and work on the MRI source code. *Ruby Under a Microscope*, on the other hand, is intended to give Ruby developers a high level, conceptual understanding of how Ruby works internally. No knowledge of C programming is required.

> For those people familiar with C, however, I will show a few vastly simplified snippets of C code to give you a more concrete sense of what's going on inside Ruby. I'll also indicate which MRI C source code file I found the snippet in; this will make it easier for you to get started studying the MRI C code yourself if you ever decide to. Like this paragraph, I'll display this information on a yellow background.
>
> If you're not interested in the C code details, just skip over these yellow sections.

Why bother to study Ruby internals?

Everyday you need to use your car to drive to work, drop your kids off at school, etc., but how often have you ever thought about how your car actually works internally? When you stopped at that red light on your way to the grocery store last weekend were you thinking about the theory and engineering behind the internal combustion engine? No, of course not! All you need to know about your car is which pedal is which, how to turn the steering wheel and a few other important details like shifting gears, turn indicator lights, etc.

At first glance, studying how Ruby is implemented internally is no different. Why bother to learn how the language was implemented when all you need to do is use it? Well, in my opinion, there are a few good reasons why you should take the time to study the internal implementation of Ruby:

- You'll become a better Ruby developer. By studying how Ruby works internally, you can become more aware of how Yukihiro Matsumoto and the rest of the Ruby core team intended the language to be used. You'll be a better Ruby developer by using the language as it was intended to be used, and not just in the way you prefer to use it.
- You can learn a lot about computer science. Beyond just appreciating the talent and vision of the Ruby core team, you'll be able to learn from their work. While implementing the Ruby language, the core team had to solve many of the same computer science problems that you might have to solve in your job or open source project.
- It's fun! I find learning about the algorithms and data structures Ruby uses internally absolutely fascinating, and I hope you will too.

Roadmap

The journey I took through Ruby's internal implementation was a long one, and I covered a lot of ground. Here's a quick summary of what I will teach you about in *Ruby Under a Microscope*:

I start in Chapter 1, Tokenization, Parsing and Compilation, by describing how Ruby reads in and processes your Ruby program. When you type "ruby my_script.rb" at the console and press ENTER, what happens? How does Ruby make sense of the text characters you typed into your Ruby program? How and why does Ruby transform your Ruby code from one format to another and another? I find this to be one of the most fascinating areas of Ruby internals and of computer science.

Next in Chapter 2, How Ruby Executes Your Code, I pick up the story from where Chapter 1 left off and describe how Ruby's virtual machine, called "Yet Another Ruby Virtual Machine" (YARV) executes your program. How does YARV actually execute your Ruby code? How does it keep track of local variables? How does YARV execute "if" statements and other control structures?

In Chapter 3, Objects, Classes and Modules, I switch gears and explain how Ruby's object model works internally. What is a Ruby object? What would I see if I could slice one open? How do Ruby classes work? How are Ruby modules and classes related? How does Ruby's method lookup algorithm work?

Chapter 4, Hash Tables, thoroughly explains how hash tables work in Ruby. Ruby uses hash tables not only to implement the hash object you use in your programs, but also for many of its own internal data structures. Methods, instance variables, constant values - Ruby stores all of these and many other things internally in hash tables. Hash tables are central to Ruby internals.

Finally in Chapter 5, How Ruby Borrowed a Decades Old Idea from Lisp, I take a trip back to the 1960s to learn more about closures, first introduced by the Lisp programming language. Are Ruby blocks really closures? What happens when you call a block? How are lambdas, procs, bindings and blocks related? And what about metaprogramming - what does this have to do with closures?

Along the way in each of these five chapters I compare and contrast the MRI implementation with how JRuby and Rubinius work. While most Ruby developers still use MRI, it isn't the only game in town and there's a lot to learn from the alternative implementations as well. In fact, there are even more versions of Ruby that I didn't have time to cover here at all: mruby, MacRuby, RubyMotion, among others.

What is missing?

Ruby's internal implementation is a vast, foreign territory that I have just started to describe. Completely covering all of Ruby's internal implementation would require many trips - many books similar to *Ruby Under a Microscope*. To name just a few examples: I didn't cover how Ruby implements many of the core classes, such as strings, arrays or files. I also didn't cover garbage collection and memory management, and I never said a word about threads or concurrency.

Instead, I decided to cover the real "guts" of the language: how does the Ruby language work at its core? I felt it would be better to cover a few important topics well, in great detail, rather than to touch on a larger set of topics at a surface level.

If I have time someday, and if *Ruby Under a Microscope* turns out to be a useful resource for the Ruby community, I may try to write a second book. I might call it *Ruby Under a Microscope Part 2*, which would pick up where this book left off. However, I'm not making any promises!

A word about my diagrams

As you'll see, *Ruby Under a Microscope* is an illustrated travel journal. While I'm not an artist, I tried my best to describe Ruby's internals visually. Obviously, a picture is worth a thousands words. My goal is that some of these diagrams come back into your mind the next time you use a particular feature of Ruby. I want you to be able to imagine what

is happening inside Ruby when you call a block, include a module in a class or save a value in a hash, for example.

However, my diagrams are not intended to be either definitive or exhaustive. Instead, view them as visual aids that can help you understand something. On many occasions I left out fields, pointers and other details that I felt would be confusing or just wouldn't fit. The only way to get completely accurate information about a particular structure, object or algorithm is to read the C source code yourself.

Feedback please

Please send feedback to:

- http://patshaughnessy.net/ruby-under-a-microscope#disqus_thread
- Twitter: @pat_shaughnessy
- Email: pat@patshaughnessy.net
- https://github.com/patshaughnessy/ruby-under-a-microscope/issues

Chapter 1
Tokenization, Parsing and Compilation

Your code has a long road to take before Ruby ever runs it.

How many times do you think Ruby reads and transforms your code before running it? Once? Twice? Whenever you run a Ruby script – whether it's a large Rails application, a simple Sinatra web site, 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 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.

At a high level, here's what this journey looks like:

First, Ruby tokenizes your code. During this first step, Ruby reads the text characters in your code file and converts them into tokens. Think of tokens as the words that are used in the Ruby language. In the next step, Ruby parses these tokens; "parsing" means to group the tokens into meaningful Ruby statements. This is analogous to grouping words into sentences. Finally, Ruby compiles these statements or sentences into low level instructions that Ruby can execute later using a virtual machine.

I'll get to Ruby's virtual machine, called "Yet Another Ruby Virtual Machine" (YARV), next in Chapter 2, but first in this chapter I'll describe the tokenizing, parsing and

compiling processes which Ruby uses to understand the code you give it. Join me as I follow a Ruby script on its journey!

Chapter 1 Roadmap

Tokens: the words that make up the Ruby language

Tokens are the words that make up the Ruby language.

Let's suppose you write this very simple Ruby program:

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

… and then execute it from the command line like this:

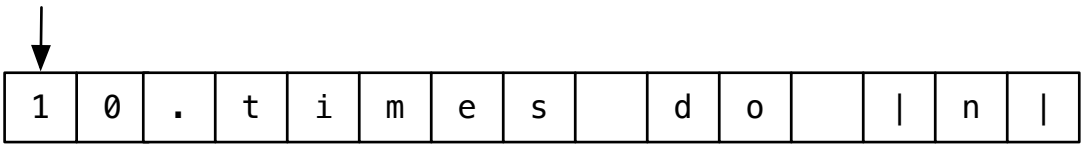
```
$ ruby simple.rb
0
1
2
3
etc...
```

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

After reading in simple.rb, Ruby encounters a series of text characters that looks like this:

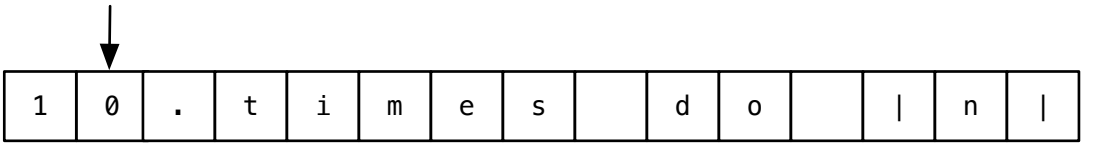
1	0	.	t	i	m	e	s		d	o		|	n	|

To keep things simple I'm only showing the first line of text here. When Ruby sees all of these characters it first "tokenizes" them. As I said above, tokenization refers to the process of converting this stream of text characters into a series of tokens or words that Ruby understands. Ruby does this by simply stepping through the text characters one at a time, starting with the first character, "1:"

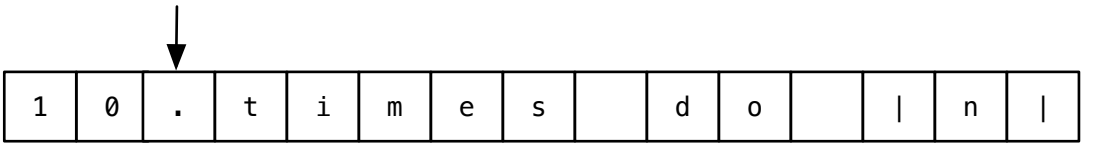


Inside the Ruby C source code, there's a loop that reads in one character at a time and processes it based on what character it is. As a simplification I'm describing tokenization as an independent process; in fact, the parsing engine I describe in the next section calls this C tokenize code whenever it needs a new token. Tokenization and parsing are two separate processes that actually happen at the same time. For now let's just continue to see how Ruby tokenizes the characters in my Ruby file.

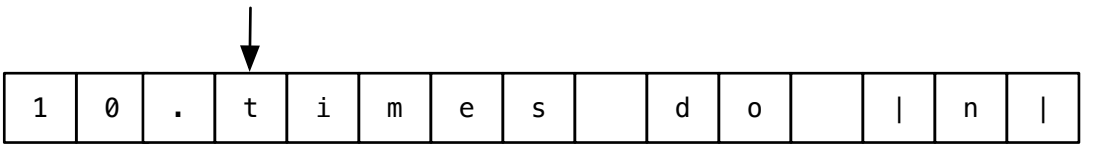
In this example, Ruby realizes that the character "1" is the start of a number, and continues to iterate over all of the following characters until it finds a non-numeric character – next it finds a "0:"



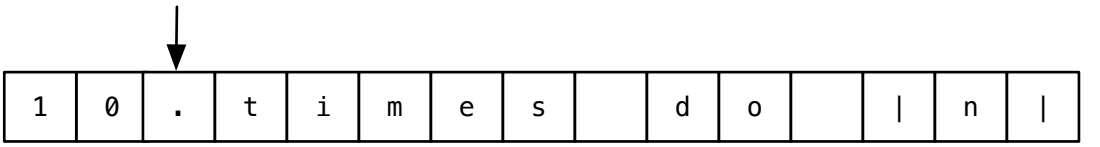
And stepping forward again it finds a period character:



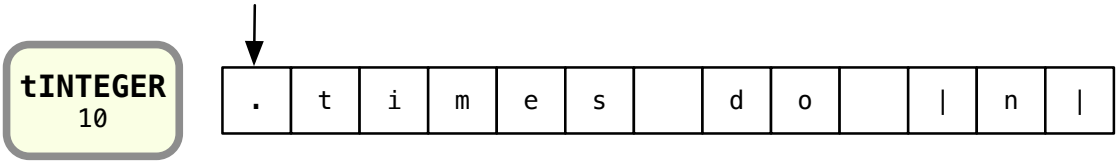
Ruby actually considers the period character to be numeric also, since it might be part of a floating point value. So now Ruby continues and steps to the next character:



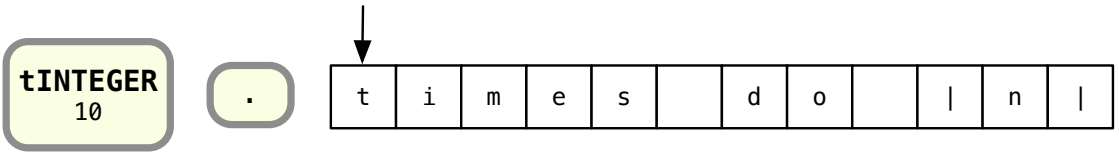
Here Ruby stops iterating since it found a non-numeric character. Since there were no more numeric characters after the period, Ruby considers the period to be part of a separate token and steps back one:



And finally Ruby converts the numeric characters that it found into a new token called `tINTEGER`:



This is the first token Ruby creates from your program. Now Ruby continues to step through the characters in your code file, converting each of them to tokens, grouping the characters together as necessary:



The second token is a period, a single character. Next, Ruby encounters the word "times" and creates an `identifier` token:

Identifiers are words that you use in your Ruby code that are not reserved words; usually they refer to variable, method or class names. Next Ruby sees "do" and creates a reserved word token, indicated by `keyword_do`:

Reserved words are the special keywords that have some important meaning in the Ruby language – the words that provide the structure or framework of the language. They are called reserved words since you can't use them as normal identifiers, although you can use them as method names, global variable names (e.g. `$do`) or instance variable names (e.g. `@do` or `@@do`). Internally, the Ruby C code maintains a constant table of reserved words; here are the first few in alphabetical order:

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

Finally, Ruby converts the remaining characters on that line of code to tokens also:

I won't show the entire program here, but Ruby continues to step through your code in a similar way, until it has tokenized your entire Ruby script. At this point, Ruby has processed your code for the first time – it has ripped your code apart and put it back together again in a completely different way. Your code started as a stream of text characters, and Ruby converted it to a stream of tokens, words that Ruby will later put together into sentences.

If you're familiar with C and are interested in learning more about the detailed way in which Ruby tokenizes your code file, take a look at the parse.y file in your version of Ruby. The ".y" extension indicates parse.y is a grammar rule file – a file that contains a series of rules for the Ruby parser engine which I'll cover in the next section. Parse.y is an extremely large and complex code file; it contains over 10,000 lines of code! But don't be intimidated; there's a lot to learn here and this file is worth becoming familiar with.

For now, ignore the grammar rules and search for a C function called `parser_yylex`, which you'll find about two thirds of the way down the file, around line 6500. This complex C function contains the code that does the actual work of tokenizing your code. If you look closely, you should see a very large switch statement that starts like this:

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

The `nextc()` function returns the next character in the code file text stream – think of this as the arrow in my diagrams above. And the `lex_state` variable 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 this code:

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


...


  goto retry;
```

… looks for whitespace characters and ignores them by jumping back up to the `retry` label just above the switch statement.

One other interesting detail here is that Ruby's reserved words are defined in a code file called defs/keywords – if you open up the keywords file you'll see a complete list of all of Ruby's reserved words, the same list I showed above. The keywords file is used by an open source package called gperf to produce C code that can quickly and efficiently lookup strings in a table, a table of reserved words in this case. You can find the generated reserved word lookup C code in lex.c, which defines a function named `rb_reserved_word`, called from parse.y.

One final detail I'll mention about tokenization is that Ruby doesn't use the Lex tokenization tool, which C programmers commonly use in conjunction with a parser generator like Yacc or Bison. Instead, the Ruby core wrote the Ruby tokenization code by hand. They may have done this for performance reasons, or because Ruby's tokenization rules required special logic Lex couldn't provide.

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 do I know the explanation above is actually correct? It turns out it is very easy to see what tokens Ruby creates for different code files, using a tool called Ripper. Shipped with Ruby 1.9 and Ruby 2.0, the Ripper class allows you to call the same tokenize and parse code that Ruby itself uses to process the text from code files. It's not available in Ruby 1.8.

Using it is simple:

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

After requiring the Ripper code from the standard library, I call it by passing some code as a string to the `Ripper.lex` method. In this example, I'm passing the same example code from earlier. Running this I get:

```
$ ruby lex1.rb
10.times do |n|
  puts n
end
[[[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_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"]]
```

Each line corresponds to a single token Ruby found in my 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` or `:on_ident`. Finally Ripper displays the text characters it found that correspond to each token.

The token symbols Ripper displays are somewhat different than the token identifiers I showed in the diagrams above. Above I used the same names you would find in Ruby's internal parse code, such as `tIDENTIFIER`, while Ripper used `:on_ident` instead. Regardless, it's easy to get a sense of what tokens Ruby finds in your code and how tokenization works by running Ripper for different code snippets.

Here's another example:

```
$ ruby lex2.rb
10.times do |n|
  puts n/4+6
end


...


 [[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"],


...
```

This time we see that 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.

Here's a third, slightly more complex example:

```
$ ruby lex3.rb
array = []
10.times do |n|
  array << n if n < 5
end
p array


...


 [[3, 2], :on_ident, "array"],
 [[3, 7], :on_sp, " "],
 [[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, " "],
 [[3, 18], :on_op, "<"],
 [[3, 19], :on_sp, " "],
 [[3, 20], :on_int, "5"],


...
```

Here you can see that Ruby was smart enough to distinguish between << and < in the line: "`array << n if n < 5`." The characters << were converted to a single operator token, while the single < character that appeared later was converted into a simple less-than operator. 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 I pass in code that contains a syntax error, Ripper will just tokenize it as usual and not complain. It's the parser's job to check syntax, which I'll get to in the next section.

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

Here I forgot the | symbol after the block parameter `n`. Running this, I get:

```
$ ruby lex4.rb
10.times do |n
  puts n
end


...


[[[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"],


...
```

Parsing: how Ruby understands the code you write

Ruby uses an LALR parser generator called Bison.

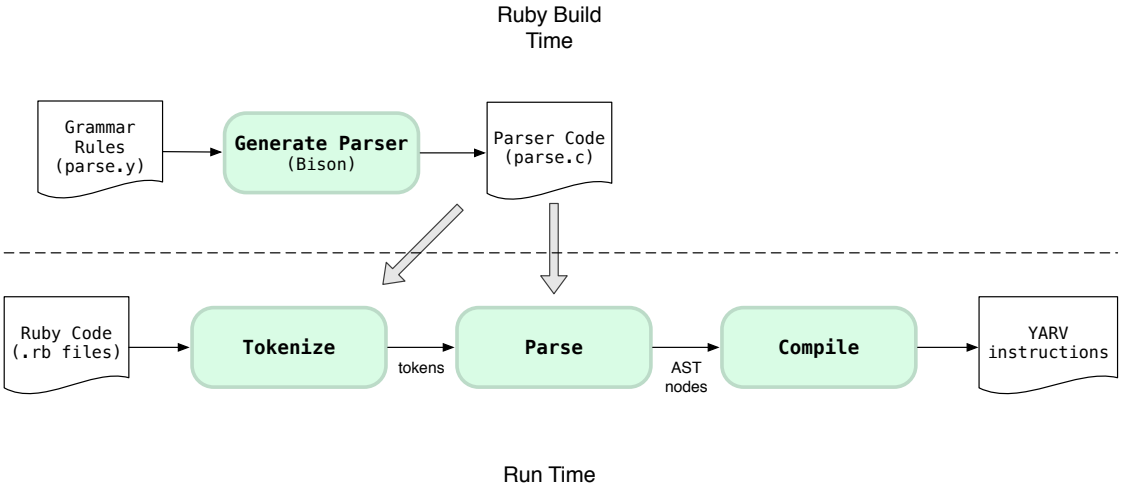
Now that Ruby has converted your code into a series of tokens, what does it do next? How does it actually understand and run your program? Does Ruby simply step through the tokens and execute each one in order?

No, it doesn't… your code still has a long way to go before Ruby can run it. As I said above, the next step on your code's journey through Ruby is called "parsing," which is the process for grouping the words or tokens into sentences or phrases that make sense to Ruby. It is during the parsing process that Ruby takes order of operations, methods and arguments, blocks and other larger code structures into account. But how does Ruby do this? How can Ruby or any language actually "understand" what you're telling it with your code? For me, this is one of the most fascinating areas of computer science… endowing a computer program with intelligence.

Ruby, like many programming languages, uses something called an "LALR parser generator" to process the stream of tokens that we just saw above. Parser generators were invented back in the 1960s; like the name implies, parser generators take a series of grammar rules and generate code that can later parse and understand tokens that follow those rules. The most widely used and well known parser generator is called Yacc ("Yet Another Compiler Compiler"), but Ruby instead uses a newer version of Yacc called Bison, part of the GNU project. The term "LALR" describes how the generated parser actually works internally – more on that below.

Bison, Yacc and other parser generators require you to express your grammar rules using "Backus–Naur Form" (BNF) notation. For Bison and Yacc, this grammar rule file will have a ".y" extension, named after "Yacc." The grammar rule file in the Ruby source code is called parse.y – the same file I mentioned earlier that contains the tokenize code. It is in this parse.y file that Ruby defines the actual syntax and grammar that you have to use while writing your Ruby code. The parse.y file is really the heart and soul of Ruby – it is where the language itself is actually defined!

Ruby doesn't use Bison to actually process the tokens, instead Ruby runs Bison ahead of time during Ruby's build process to create the actual parser code. There are really two separate steps to the parsing process, then:



Ahead of time, before you ever run your Ruby program, the Ruby build process uses Bison to generate the parser code (parse.c) from the grammar rules file (parse.y). Then later at run time this generated parser code actually parses the tokens returned by Ruby's tokenizer code. You might have built Ruby yourself from source manually or automatically on your computer by using a tool like Homebrew. Or someone else may have built Ruby ahead of time for you, if you installed Ruby with a prepared install kit.

As I explained at the end of the last section, the parse.y file, and therefore the generated parse.c file, also contains the tokenization code. This is why I show the diagonal arrow from parse.c to the "Tokenize" process on the lower left. In fact, the parse engine I am 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

Now let's take a look at how grammar rules work – the best way to become familiar with grammar rules is to take a close look at one simple example. Suppose I want to translate from the Spanish:

Me gusta el Ruby

…to the English:

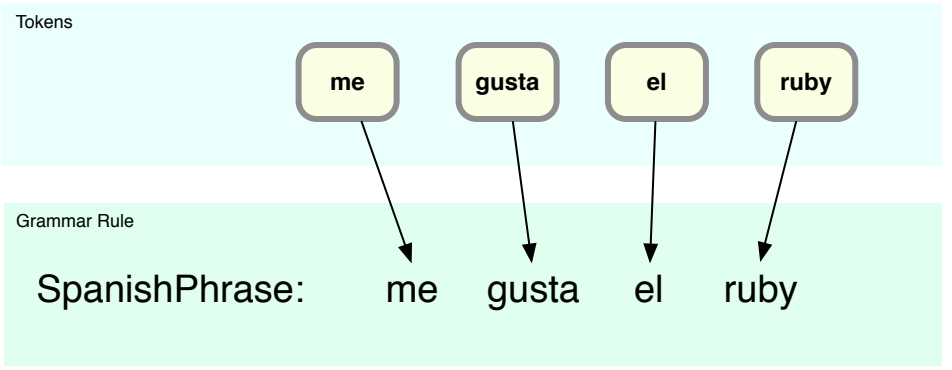
I like Ruby

… and that to do this suppose I use Bison to generate a C language parser from a grammar file. Using the Bison/Yacc grammar rule syntax – the "Backus–Naur" notation – I can write a simple grammar rule like this with the rule name on the left, and the matching tokens on the right:[1]

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

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

How does this work? Here's a conceptual picture of the parsing process in action:



At the top I show the four input tokens, and the grammar rule right underneath it. It's obvious in this case there's a match since each input token corresponds directly to one of the terms on the right side of the grammar rule. In this example we have a match on the `SpanishPhrase` rule.

Now let's change the example to be a bit more complex: suppose I need to enhance my parser to match both:

1. This is actually a slightly modified version of BNF that Bison uses – the original BNF syntax would have used '::=' instead of a simple ':' character.

Me gusta el Ruby

and:

Le gusta el Ruby

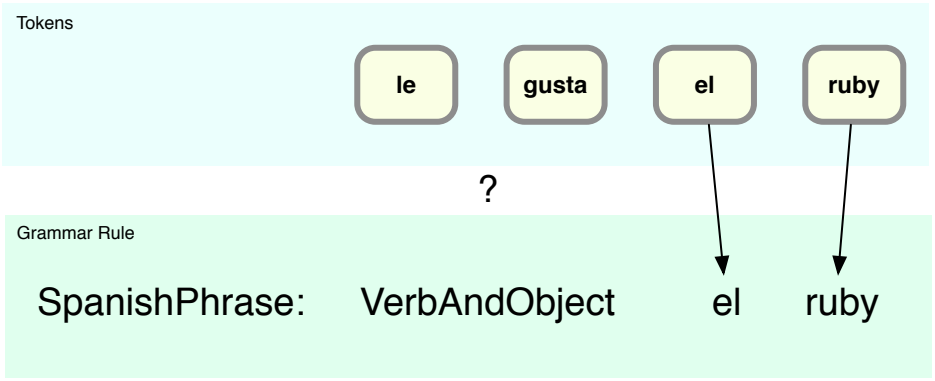
…which means “She/He/It likes Ruby.” Here’s a more complex grammar file that can parse both Spanish phrases:[2]

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

There’s a lot more going on here; you can see four grammar rules instead of just one. Also, I’m 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.

Now things aren’t so obvious – the parser can’t immediately match any of the grammar rules like in the previous, trivial example:

2. again this is a modified version of BNF used by Bison – the original syntax from the 1960s would use < > around the child rule names, like this for example: “VerbAndObject ::= <SheLikes> | <ILike>”)

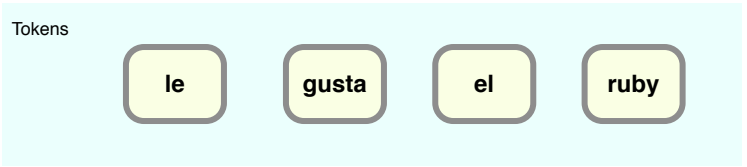


Here using the `SpanishPhrase` rule the `el` and `ruby` tokens match, but `le` and `gusta` do not. Ultimately we'll see how the child rule `VerbAndObject` does match "le gusta" but for now there is no immediate match. And now that there are four grammar rules, how does the parser know which one to try to match against? …and against which tokens?

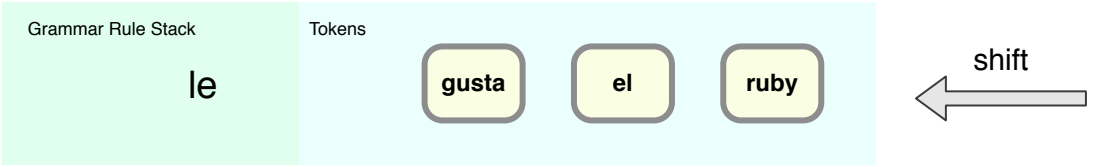
This is where the real intelligence of the LALR parser comes into play. This acronym describes the algorithm the parser uses to find matching grammar rules, and means "Look Ahead LR parser." We'll get to the "Look Ahead" part in a minute, but let's start with "LR:"

- "L" (left) means the parser moves from left to right while processing the token stream; in my example this would be: le, gusta, el, ruby.
- "R" (reversed rightmost derivation) means the parser uses a bottom up strategy for finding matching grammar rules, by using a shift/reduce technique.

Here's how the algorithm works for this example. First, the parser takes the input token stream:



… and shifts the tokens to the left, creating something I'll call the "Grammar Rule Stack:"



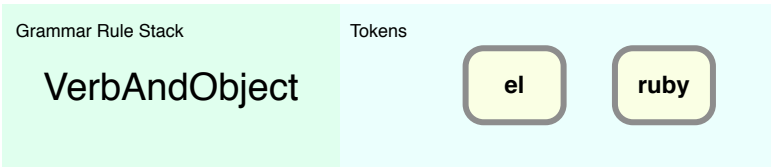
Here since the parser has processed just one token, `le`, this is kept in the stack alone for the moment. "Grammar Rule Stack" is a simplification; in reality the parser does use a stack, but instead of grammar rules it pushes numbers on to its stack that indicate which grammar rule it just parsed. These numbers – or states from a state machine – help the parser keep track of where it is as it processes the tokens.

Next, the parser shifts another token to the left:

Now there are two tokens in the stack on the left. At this point the parser stops to examine all of the different grammar rules and looks for one that matches. In this case, it finds that the `SheLikes` rule matches:

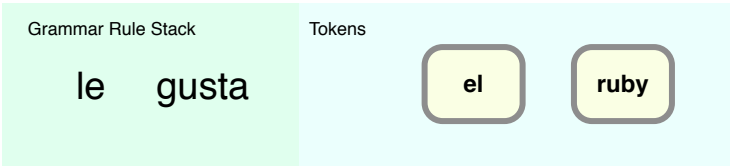
This operation is called "reduce," since the parser is replacing the pair of tokens with a single, matching rule. This seems very straightforward… the parser just has to look through the available rules and reduce or apply the single, matching rule.

Now the parser in our example can reduce again – now there is another matching rule: `VerbAndObject`! This rule matches because of the OR (vertical bar) operator: it matches *either* the `SheLikes` *or* `ILike` child rules. The parser can next replace `SheLikes` with `VerbAndObject`:



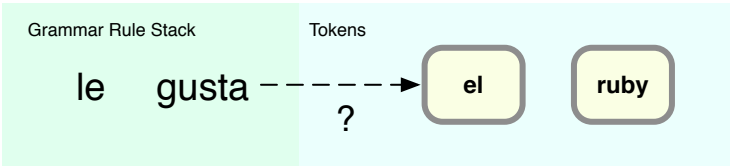
But let's stop for a moment and think about this a bit more carefully. How did the parser know to reduce and not continue to shift tokens? Also, in the real world there might actually be many matching rules the parser could reduce with – how does it know which rule to use? This is the crux of the algorithm that LALR parsers use… that Ruby uses… 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…



… there were multiple matching rules that 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?

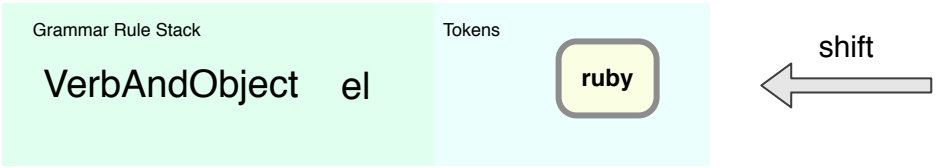
Here's where the "LA" (Look Ahead) portion of LALR comes in: in order to find the proper matching rule it looks ahead at the next token:



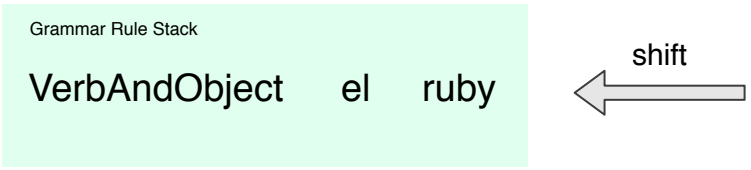
Additionally, the parser maintains a state table of possible outcomes depending on what the next token was and which grammar rule was just parsed. This table would contain a series of states, describing which grammar rules have been parsed so far, and which states to move to next depending on the next 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.

I won't show the details of what a state table looks like; if you're interested, the actual LALR state table for Ruby can be found in the generated parse.c file. Instead let's just continue the shift/reduce operations for my simple example. After matching the `VerbAndObject` rule, the parser would shift another token to the left:



At this point no rules would match, and the state machine would shift another token to the left:



And finally, the parent grammar rule `SpanishPhrase` would match after a final reduce operation:

Why have I 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, using the `$$`, `$1`, `$2`, etc. symbols. The only real difference is scale – my Spanish phrase grammar is extremely simple, trivial really. On the other hand, Ruby's grammar is very complex, an intricate series of interrelated parent and child grammar rules, which sometimes even refer to each other in circular, recursive patterns. But this complexity just means that the generated state table in the parse.c file

is larger. The basic LALR algorithm – how the parser processes tokens and uses the state table – is the same.

Some actual Ruby grammar rules

Let's take a quick look at some of the actual Ruby grammar rules from parse.y. Here's my example Ruby script from the last section on tokenization:

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

This is a very simple Ruby script, right? Since this is so short, it should't be too difficult to trace the Ruby parser's path through its grammar rules. Let's take a look at how it works:

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 | …
```

On the left I show the code that Ruby is trying to parse. On the right are the actual matching grammar rules from the Ruby parse.y file, shown in a simplified manner. The first rule, "program: top_compstmt," is the root grammar rule which matches every Ruby program in its entirety. As you follow the list down, you can see a complex series of child rules that also match my 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 that has two matching child rules: "method_call" and "brace_block." Let's take the "method_call" rule first:

Ruby Code

```
10.times
```

Grammar Rules

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

The “method_call” rule matches the “10.times” portion of my Ruby code - i.e. where I call the `times` method on the 10 `Fixnum` object. You can see the rule matches another primary value, followed by a period character, followed, in turn, by an “operation2.” The period is simple enough, and here’s how the “primary_value” and “operation2” child rules work: first the “primary_value” rule matches the literal “10:”

Ruby Code

```
10
```

Grammar Rules

```
primary_value: primary
primary: literal | …
```

And then the “operation2” rule matches the method name `times`:

Ruby Code

```
times
```

Grammar Rules

```
operation2: identifier | …
```

What about the rest of my Ruby code? How does Ruby parse the contents of the “do … puts… end” block I passed to the `times` method? Ruby handles that using the “brace_block” rule from above:

Ruby Code

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

Grammar Rules

```
brace_block: … | keyword_do opt_block_param compstmt keyword_end | …
```

I won’t go through all the remaining child grammar rules here, 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,” and
- “keyword_end” matches the `end` reserved keyword

To give you a taste of what the actual Ruby parse.y source code looks like, here's a small portion of the file containing part of the "method_call" grammar rule I showed above:

```
method_call        :


...


  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);
  %*/
  }
```

Like my Spanish to English example grammar file above, here you can see there are snippets of complex C code that appear after each of the terms in the grammar rule. The way this works is that the Bison generated parser will execute these snippets if and when there's a match for this 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. What we end up with is a confusing mix of C and Bison directives.

And to make things even worse, Ruby uses a trick during the Ruby build process to divide these C/Bison code snippets into separate pieces – some that are actually used by Ruby and others that are only used by the Ripper tool which we tried out in Experiment 1-1. Here's how that works:

- The C code that appears between the `/*%%%*/` line and the `/*%` line is actually compiled into Ruby during the Ruby build process, and:

- The C code between `/*%` and `%*/` is dropped when Ruby is built. Instead, this code is only used 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 of parse.y.

And 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 what 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 (or AST)… more on that in a minute. You can see one example of creating an AST node above, where Ruby calls the `NEW_CALL` C macro/function. This creates a new `NODE_CALL` node, which represents a method call. We'll see later how Ruby eventually compiles this into byte code that can be executed by a virtual machine.

Experiment 1-2: Using Ripper to parse different Ruby scripts

In Experiment 1-1 I showed you how to use Ripper to display the tokens Ruby converts your code into, and we just saw how all of 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. Here's how to do it:

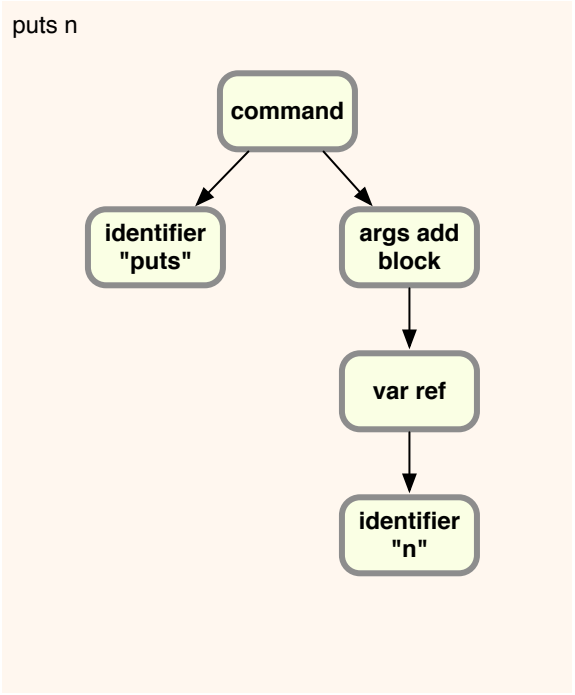
```
require 'ripper'
require 'pp'
code = <<STR
10.times do |n|
  puts n
end
STR
puts code
pp Ripper.sexp(code)
```

This is exactly the same code I showed in the first experiment, except that here I call `Ripper.sexp` instead of `Ripper.lex`. Running this I get:

```
[: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]]]]]]
```

What the heck does this mean? I can see some bits and pieces from my Ruby script in this cryptic text, but what do all of the other symbols and arrays mean here?

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 found 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 the previous yellow section. The purpose of the AST is to record the structure and syntactical meaning of your Ruby code. To see what I mean, here's a small piece of the AST structure Ripper just displayed for my sample Ruby script:



This is how Ruby represents that single call `puts n` internally. This corresponds to the last three lines of the Ripper output:

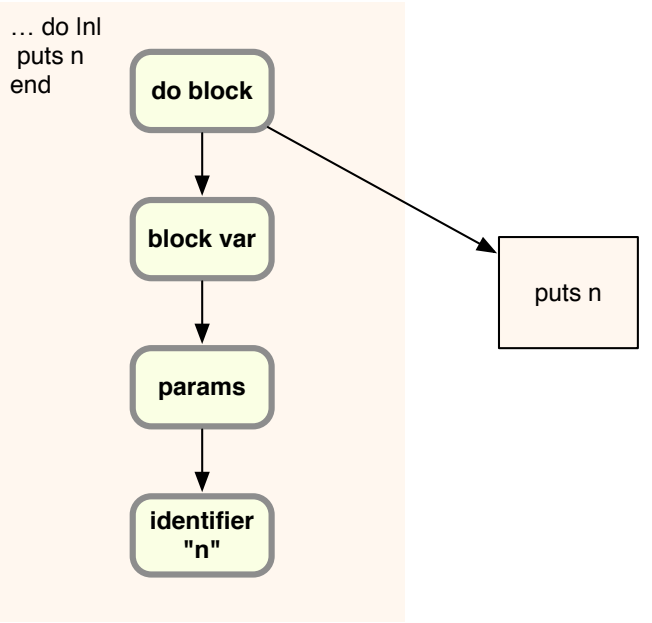
```
[[:command,
   [:@ident, "puts", [2, 2]],
   [:args_add_block, [[:var_ref, [:@ident, "n", [2, 7]]]],
                     false]]]
```

Like in Experiment 1-1 when we displayed token information from Ripper, you can see the source code file line and column information are displayed as integers. For example [2, 2] indicates that Ripper found the `puts` call on line 2 at column 2 of my code file.

Aside from that, you can see that Ripper outputs an array for each of the nodes in the AST, "`[:@ident, "puts", [2, 2]]`" for example.

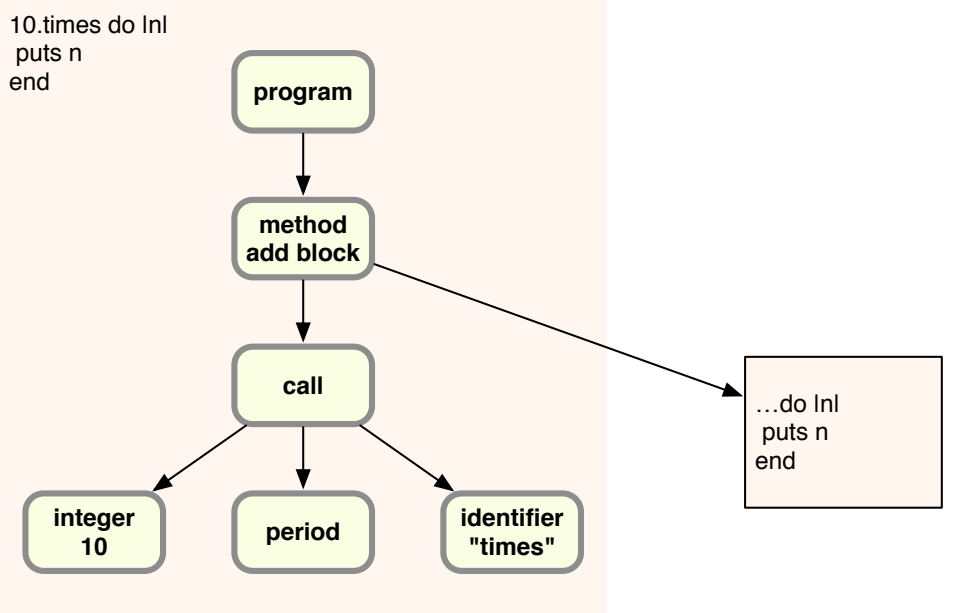
What's interesting and important about this is that now my Ruby program is beginning to "make sense" to Ruby. Instead of a simple stream of tokens, which could mean anything, Ruby now has a detailed description of what I meant when I wrote `puts n`. We have a function call, "a command," followed by an identifier node which indicates what function to call. Ruby uses the `args_add_block` node since you might optionally pass a block to a command/function call like this. Even though we are not passing a block in this case, the `args_add_block` node is still saved into the AST. Another interesting detail is how the `n` identifier is recorded as a `:var_ref` or variable reference node, and not as a simple identifier.

Let's take a look at more of the Ripper output:



Here you can see 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 I showed above, the parsed version of the `puts` call.

And finally here's the entire AST for my sample Ruby code:



Here you can see "method add block" means we're calling a method, but also adding 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.

Again, the key point here is that now your Ruby program is no longer a simple series of tokens – Ruby now "understands" what you meant with your code. Ruby's knowledge of your code is saved in the way the nodes are arranged in the AST.

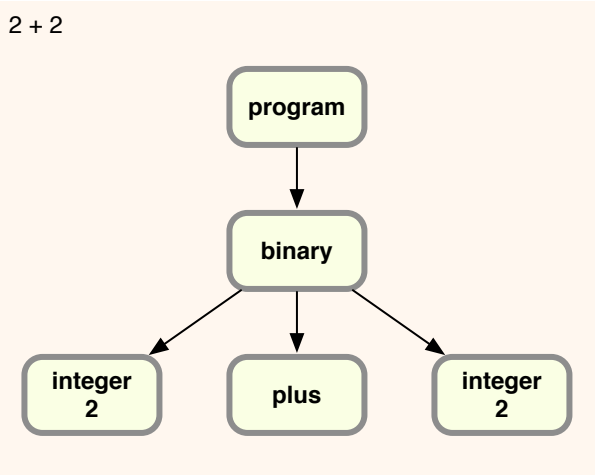
To make this point even more clear, suppose I pass the Ruby expression "`2+2`" to Ripper like this:

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

And running it I get:

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

Here you can see the + is represented with an AST node called “binary:”



Not very surprising, but look what happens when I pass the expression “`2 + 2 * 3`” into Ripper:

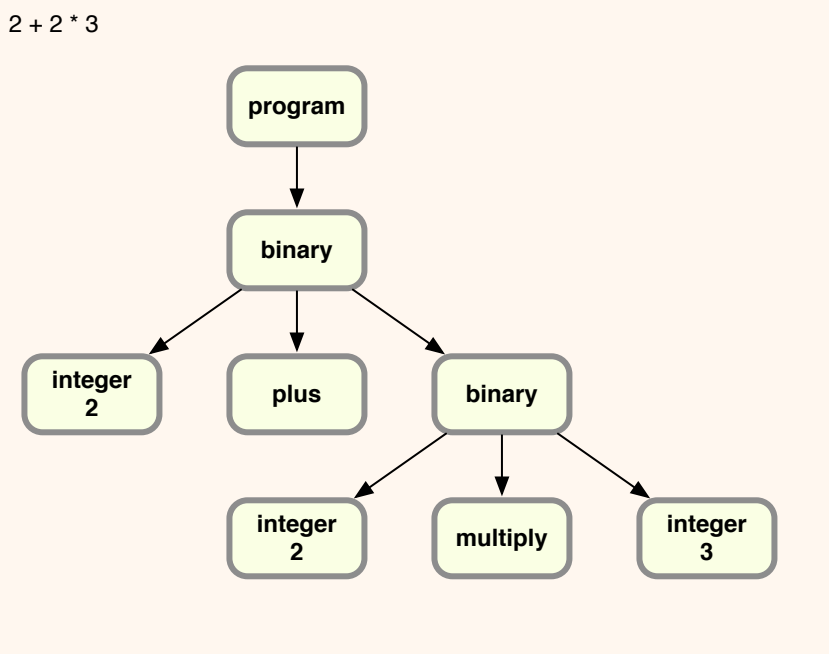
```
require 'ripper'
require 'pp'
code = <<STR
2 + 2 * 3
STR
puts code
pp Ripper.sexp(code)
```

Now I get:

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

```
    :*,
    [:@int, "3", [1, 8]]]]]]
```

And here's what that looks like:



Note how Ruby was smart enough to realize that multiplication has a higher precedence than addition does. We all knew this, of course… this isn't very interesting. But what is interesting to me here is how the AST tree structure itself inherently captures the information about the order of operations. The simple token stream: `2 + 2 * 3` just indicates what I wrote in my code file, while the parsed version saved to the AST structure now contains the *meaning* of my code – all the information Ruby will need later to execute it.

One final note: Ruby itself actually contains some debug code that can also display information about the AST node structure. To use it, just run your Ruby script with the "parsetree" option:

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

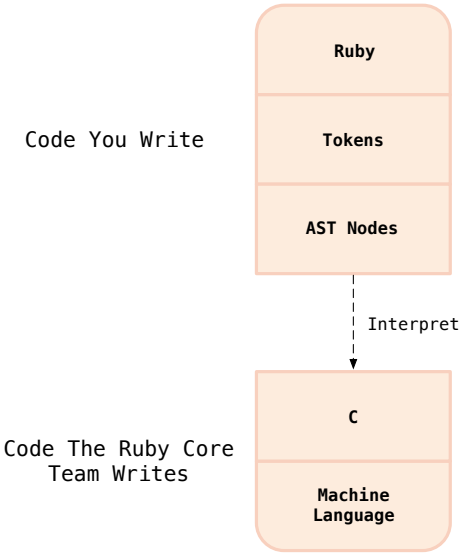
This will display the same information we've just seen, but in a different format. Instead of showing symbols, the "parsetree" option will show the actual node names from the C source code. In the next section, about how Ruby compiles your code, I'll also use the actual node names.

Compilation: how Ruby translates your code into a new language

The code Ruby actually runs looks nothing like your original code.

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

The answer depends on which version of Ruby you have. If you're still using Ruby 1.8, then yes: Ruby will now simply walk through the nodes in the AST and execute each one. Here's another way of looking at the Ruby 1.8: tokenizing and parsing processes:



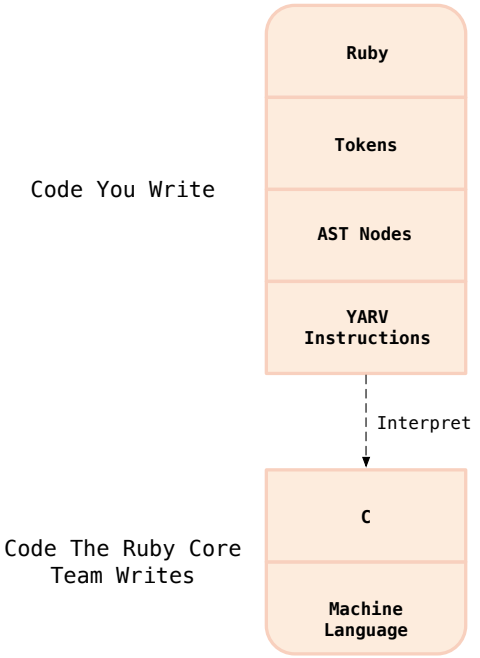
At the top as you move down you can see how Ruby translates your Ruby code into tokens and AST nodes, as I described above. At the bottom I show the Ruby 1.8 interpreter itself – written in C and, of course, compiled into native machine language code.

I show a dotted line between the two code sections to indicate that Ruby 1.8 simply interprets your code – it doesn't compile or translate your code into any other form past AST nodes. After converting your code into AST nodes, Ruby 1.8 proceeds to iterate over the nodes in the AST, taking whatever action each node represents – executing each node. The break in the diagram between "AST nodes" and "C" means your code is never completely compiled into machine language. If you were to disassemble and inspect the machine language your computer's CPU actually runs, you would never find instructions that directly map to your original Ruby code. Instead, you would find instructions that tokenize, parse and execute your code… instructions that implement the Ruby interpreter.

However if you have upgraded to Ruby 1.9 or Ruby 2.0, then Ruby is still not quite ready to run your code. There's one final step on your code's journey through Ruby: compilation. With Ruby 1.9, the Ruby core team introduced something called "Yet Another Ruby Virtual Machine" (or YARV), which actually executes your Ruby code. At a high level, this is the same idea behind the much more famous Java Virtual Machine (or JVM) used by Java and many other languages. To use the JVM, you first compile your Java code into "byte code," a series of low level instructions that the JVM understands. Starting with version 1.9, Ruby works the same way! The only differences are that:

- Ruby doesn't expose the compiler to you as a separate tool; instead, it automatically compiles your Ruby code into byte code instructions internally without you ever realizing it.
- MRI Ruby also never compiles your Ruby code all the way to machine language. As you can see in the next diagram, Ruby interprets the byte code instructions. The JVM, however, can compile some of the byte code instructions all the way into machine language using its "hotspot" or JIT compiler.

Here's the same diagram again, this time showing how Ruby 1.9 and Ruby 2.0 handle your code:



This time your Ruby code is translated into no less than three different formats or intermediate languages! After parsing the tokens and producing the AST, Ruby 1.9 and 2.0 continue to compile your code to a series of low level instructions called "YARV instructions" for lack of a better name.

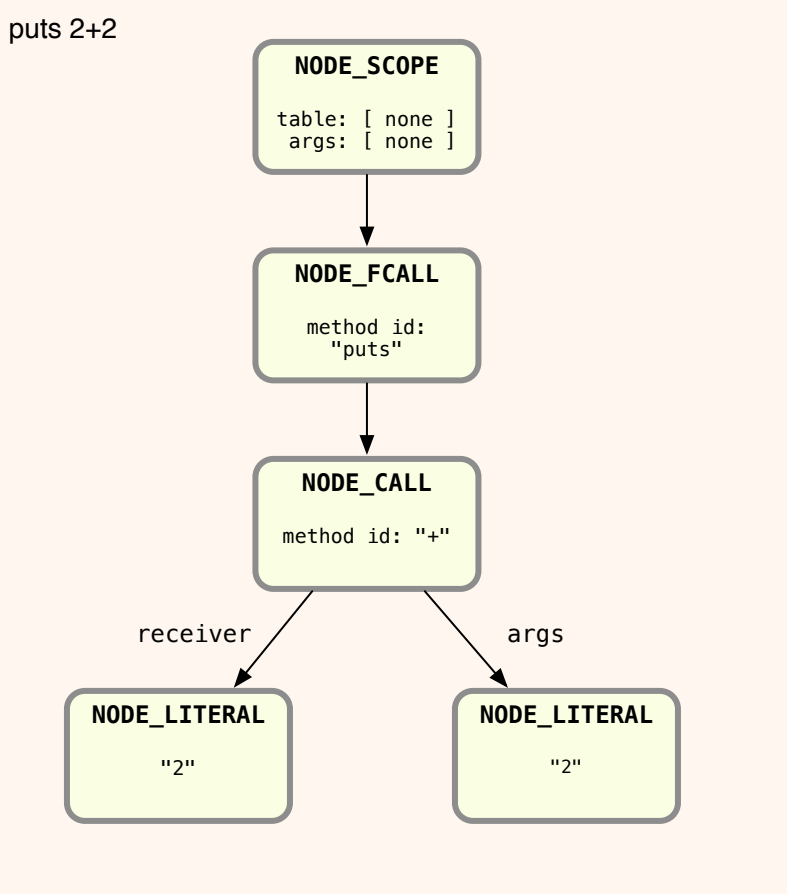
I'll cover YARV in more detail in the next chapter: what the instructions are and how they work, etc. I'll also look at how much faster Ruby 1.9 and Ruby 2.0 are compared to Ruby 1.8. The primary reason for all of the work that the Ruby core team put into YARV is speed: Ruby 1.9 and 2.0 run much faster than Ruby 1.8 primarily because of the use of the YARV instructions. Like Ruby 1.8, YARV is still an interpreter, although 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 the diagram between the YARV instructions and Ruby's C code.

Stepping through how Ruby compiles a simple script

But now let's take a look at how Ruby compiles your code into the instructions that YARV expects – the last step along your code's journey through Ruby. Here's an example Ruby script that calculates 2+2 = 4:

```
puts 2+2
```

And here's the AST structure Ruby will create after tokenizing and parsing this simple program – note this is a more technical, detailed view of the AST than you would get from the Ripper tool… what we saw above in Experiment 2:

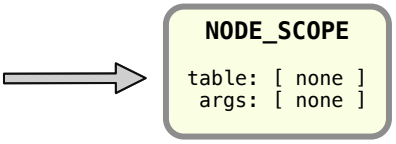


The technical names I show here, `NODE_SCOPE`, `NODE_FCALL`, etc., are taken from the actual MRI Ruby C source code. To keep this simple, I'm also omitting some AST nodes that aren't important in this example: nodes that represent arrays of the arguments to each method call, which in this simple example would be arrays of only one element.

Before we get into the details of how Ruby compiles this program, let me mention one very important attribute of YARV: it is a stack oriented virtual machine. As I'll explain in the next chapter, that means when YARV executes your code it maintains a stack of values, mainly arguments and return values for the YARV instructions. 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.

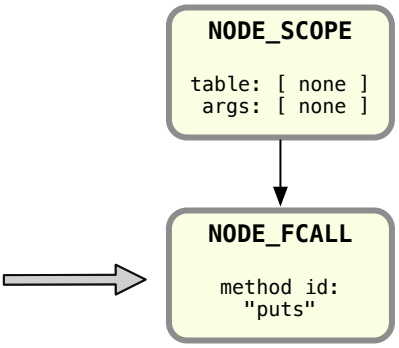
Now to compile the "`puts 2+2`" AST structure into YARV instructions, Ruby will iterate over the tree in a recursive manner from the top down, converting each AST node into one or more instructions. Here's how it works, starting with the top node, `NODE_SCOPE`:



YARV instructions

`NODE_SCOPE` tells the Ruby compiler it is now starting to compile a new scope or section of Ruby code – in this case a whole new program. Conceptually I'm indicating this scope on the right with the empty green box. The "table" and "args" values are both empty, so we'll ignore those for now.

Next the Ruby compiler will step down the AST tree and encounter `NODE_FCALL`:



YARV instructions

putself

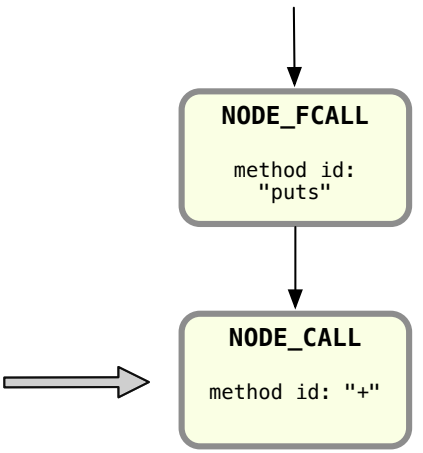
`NODE_FCALL` represents a function call, in this case the call to `puts`. Function and method calls are very important and very common in Ruby programs; Ruby compiles them for YARV using this pattern:

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

So in this example, the Ruby compiler first creates a YARV instruction called `putself` – this indicates that the function call uses the current value of the "self" pointer as the receiver. Since I call `puts` from the top level scope of this simple, one line Ruby script, "self" will be set to point to the "top self" object. The "top self" object is an instance of the "Object" class automatically created when Ruby starts up. It's sole purpose is to serve as the receiver for function calls like this one in the top level scope.

Next Ruby needs to create instructions to push the arguments of the `puts` function call. But how can it do this? The argument to `puts` is `2+2` – in other words the result of some other method call. Although `2+2` is a very simple expression in this example, `puts` could instead be operating on some extremely complex Ruby expression involving many operators, method calls, etc. How can Ruby possibly know what instructions to create here?

The answer lies in the structure of the AST: by simply following the tree nodes down in a recursive manner, Ruby can take advantage of all the work the parser did earlier. In this case, Ruby can now just step down to the `NODE_CALL` node:



YARV instructions

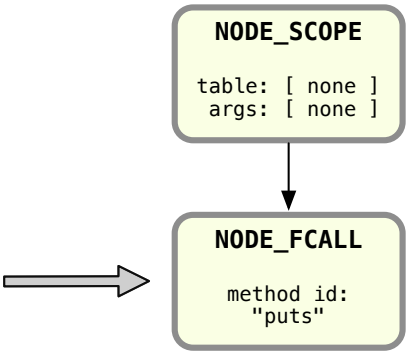
```
putself
putobject  2
putobject  2
send       :+, 1
```

Here Ruby will compile the + method call, which theoretically is really the process of sending the + message to the `2 Integer` object. Again, following the same receiver – arguments – method call format I explained above:

- First Ruby creates a YARV instruction to push the receiver onto the stack, the object `2` in this case.
- Then Ruby creates a YARV instruction to push the argument or arguments onto the stack, again `2` in this example.
- Finally Ruby creates a method call YARV instruction – "`send :+, 1`". This means "send the '+' message" to the receiver: whatever object was previously pushed onto the YARV stack, in this case the first `Fixnum 2` object. The `1` parameter tells YARV there is one argument to this method call, the second `Fixnum 2` object.

What you have to imagine here – and what we'll go through more carefully in the next chapter – is how YARV will execute these instructions. What will happen when Ruby executes the "`send :+`" instruction is that it will add 2+2, fetching those arguments from the stack, and leave the result 4 as a new value on the top of the stack.

What I find fascinating about this is that YARV's stack oriented nature also helps Ruby to compile the AST nodes more easily. You can see how this is the case when Ruby continues to finish compiling the `NODE_FCALL` from above:



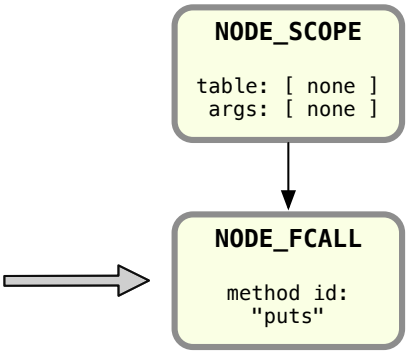
YARV instructions

```
putself
putobject  2
putobject  2
send       :+, 1
send       :puts, 1
```

Now Ruby can assume the return value of the "2+2" operation, 4, will be left at the top of the stack, just where Ruby needs it 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! On the right you can see Ruby has added the "`send :puts, 1`" instruction. This last instruction will call the `puts` function, and as before the value `1` indicates there is one argument to the `puts` function.

It turns out Ruby later modifies these YARV instructions one more time before executing them: the Ruby compiler has an optimize step, and one of Ruby's optimizations is to

replace some YARV instructions with “specialized instructions.” These are special YARV instructions that represent commonly used operations such as “size,” “not,” “less-than,” “greater-than,” etc. One of these special instructions is for adding two numbers together: the `opt_plus` YARV instruction. So during this optimization step Ruby changes the YARV program to:



YARV instructions

```
putself
putobject   2
putobject   2
opt_plus
send        :puts, 1
```

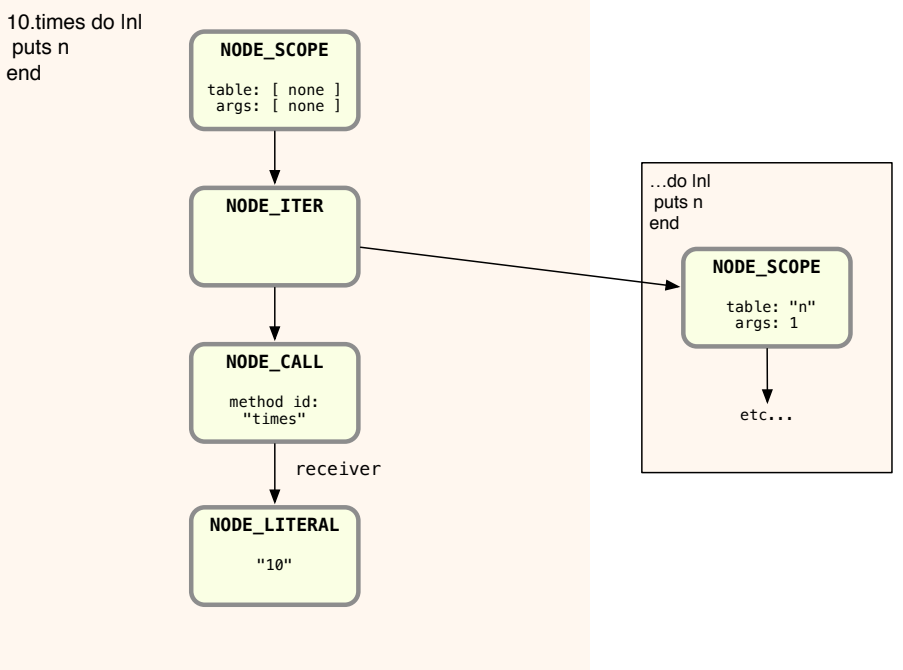
You can see here that Ruby replaced “`send :+, 1`” with `opt_plus` – a specialized instruction which will run a bit faster.

Compiling a call to a block

Now let’s take a somewhat more complex example and compile my “`10.times do`” example from before:

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

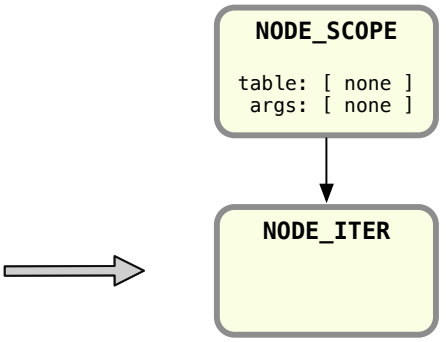
What really makes this example interesting is the fact that I’ve introduced a block as a parameter to the `times` method. Let’s see how the Ruby compiler handles blocks. Here is the AST for the the “`10.times do`” example – again using the actual node names and not the simplified output from Ripper:



This looks very different than "`puts 2+2`," mostly because of the inner block shown on the right side. I did this to keep the diagram simpler, but also because Ruby handles the inner block differently, as we'll see in a moment. But first, let's break down how Ruby compiles the main portion on the script, on the left. Ruby starts with the top `NODE_SCOPE` as before, and creates a new snippet of YARV instructions:

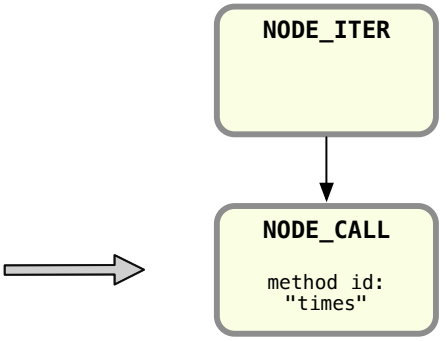
YARV instructions

Now Ruby steps down the AST nodes on the left, to `NODE_ITER`:



YARV instructions

Here there is still no code generated, but notice that above in the AST there are two arrows leading from `NODE_ITER`: one continues down to the `NODE_CALL`, which represents the `10.times` call, and a second to the inner block on the right. First Ruby will continue down the AST and compile the nodes corresponding to the "`10.times`" code. I'll save some space and skip over the details; here's the resulting YARV code following the same receiver-arguments-message pattern we saw above:

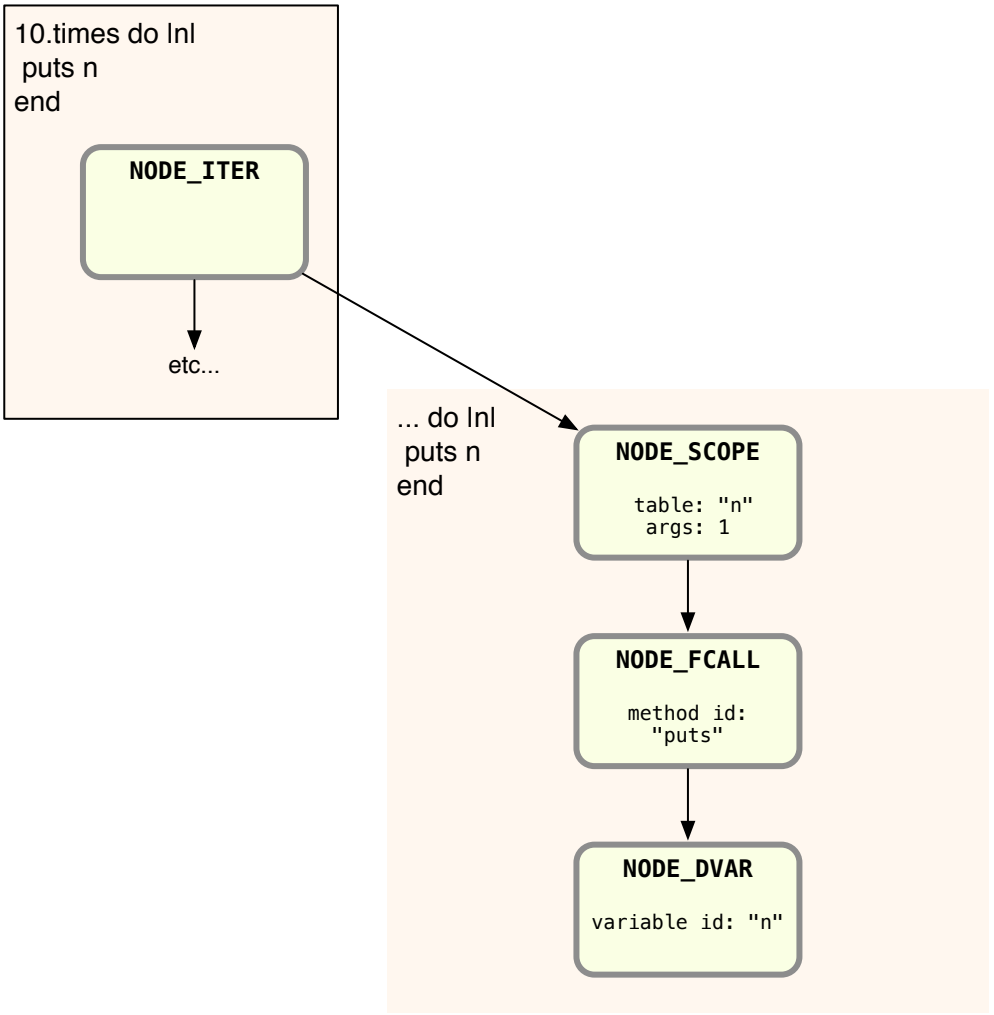


YARV instructions

```
putobject   10
send        :times, 0, block in <main>
```

You can see here that the new YARV instructions push the receiver, the `Integer` object `10`, onto the stack first. Then Ruby generates an instruction to execute the `times` method call. But note how the `send` instruction also contains an argument "block in <main>." 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, since in the AST above there's an arrow from `NODE_ITER` over to the second `NODE_SCOPE` node.

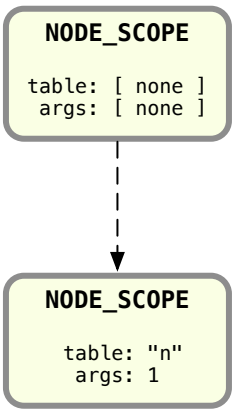
Now Ruby will continue by compiling the inner block, starting with the second `NODE_SCOPE` I showed on the right in the AST diagram above. Here's what the AST for the inner block looks like:



This looks simple enough – just a single function call and a single argument `n`. But notice I show a value for "table" and "args" in `NODE_SCOPE`. These values were empty in the parent `NODE_SCOPE` but are set here for the inner `NODE_SCOPE`. As you might guess, these values indicate the presence of the block parameter `n`. Also notice that the Ruby parser created `NODE_DVAR` instead of `NODE_LITERAL` which we saw before. This

is because `n` is actually not just a literal string or local variable; instead it is a "dynamic variable" – a reference to the block parameter passed in from the parent scope. There are also a lot of other details that I'm not showing here.

Skipping a few steps again, here's how Ruby compiles the inner block:



YARV instructions

```
putobject  10
send       :times, 0, block in <main>
```

YARV instructions

```
putself
getdynamic   n
send         :puts, 1
```

On the top I've shown the parent `NODE_SCOPE`, and the YARV code we saw above. Below that I've displayed a second green box containing 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, whether it's a block, lambda, method, class or module definition, etc., into a separate snippet of YARV instructions. Again, in the next chapter I'll take a look at how YARV actually executes these instructions, including how it jumps from one scope to another.

Now let's take a look at some of the internal code details of how Ruby actually iterates through the AST structure, converting each AST node into YARV instructions. The MRI C source code file which implements the Ruby compiler is called compile.c, not surprisingly. To learn how the code in compile.c works, you should start by looking for a function called `iseq_compile_each`. Here's what the beginning of this function looks like:

```
/**
  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)
{
```

This function is very long and again consists of a very, very long switch statement… the switch statement alone is 1000s of lines long! The switch statement branches based on the type of the current AST node and generates the corresponding YARV code. Here's the start of the switch statement:

```
type = nd_type(node);


...


switch (type) {
```

Here `node` was a parameter passed into `iseq_compile_each`, and `nd_type` is a C macro that returns the type from the given node structure.

Now let's take a quick look at how Ruby compiles function or method calls nodes into YARV instructions using the receiver/arguments/function call pattern from earlier. First search in compile.c for this case in the large switch statement:

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

Here as the comment explains `NODE_CALL` represents a real method call (like `10.times`), `NODE_FCALL` a function call (like `puts`) and `NODE_VCALL` a "variable" or function call. Skipping over some of the C code details –

including optional SUPPORT_JOKE code used for implementing the goto statement – here's what Ruby does next to compile these AST nodes:

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

Here Ruby calls either COMPILE or ADD_CALL_RECEIVER:

- In the first case, for real method calls (NODE_CALL), Ruby calls COMPILE 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) then Ruby calls ADD_CALL_RECEIVER which creates a pushself YARV instruction.

Next Ruby creates YARV instructions to push each argument of the method/function call onto the stack:

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

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

Finally Ruby creates YARV instructions to execute the actual method or function call:

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

This C macro will create the new `send` YARV instruction.

Experiment 1-3: Using the RubyVM class to display YARV instructions

It turns out there's an easy way to see how Ruby compiles your code: the RubyVM object gives you access to Ruby's YARV engine from your Ruby program! Just like the Ripper tool, using it is very straightforward:

```
code = <<END
puts 2+2
END

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

The challenge is understanding what the output actually means. Here's the output you'll get for "`puts 2+2`:"

```
== disasm: <RubyVM::InstructionSequence:<compiled>@<compiled>>==========
0000 trace            1                                               (   1)
0002 putself
0003 putobject        2
0005 putobject        2
0007 opt_plus         <ic:2>
0009 send             :puts, 1, nil, 8, <ic:1>
0015 leave
```

You can see the same instructions that I showed earlier in my diagrams, with some additional technical details that I omitted above for sake of clarity. There are also two new instructions that I dropped completely: `trace` and `leave`. `trace` is used to implement the `set_trace_func` feature, which will call a given function for each Ruby statement executed in your program, and `leave` is similar to a return statement. The line numbers on the left show the position of each instruction in the byte code array the compiler actually produces.

The "<ic:1>" and "<ic:2>" notation shown with both `opt_plus` and `send` indicates these two method calls will use an inline method lookup cache to speed things up later when Ruby executes the YARV instructions.

The other values shown with the `send` instruction - "`send :puts, 1, nil, 8`" indicate that:

- `puts` takes one argument,
- there is no block parameter (nil), and
- This is a function call, and not a normal method call (8).

Using RubyVM it's easy to explore how Ruby compiles different Ruby scripts; for example, here's my "`10.times do`" example:

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

Here's the output I get now - notice that the "`send :times`" YARV instruction now shows "`block in <compiled>`" which indicates that I am passing a block to the "`10.times`" method call...

```
== disasm: <RubyVM::InstructionSequence:<compiled>@<compiled>>==========
== catch table
| catch type: break  st: 0002 ed: 0010 sp: 0000 cont: 0010
|------------------------------------------------------------------------
0000 trace            1                                               (   1)
0002 putobject        10
0004 send             :times, 0, block in <compiled>, 0, <ic:0>
0010 leave
== disasm: <RubyVM::InstructionSequence:block in <compiled>@<compiled>>=
== catch table
| catch type: redo   st: 0000 ed: 0012 sp: 0000 cont: 0000
| catch type: next   st: 0000 ed: 0012 sp: 0000 cont: 0012
|------------------------------------------------------------------------
local table (size: 2, argc: 1 [opts: 0, rest: -1, post: 0, block: -1] s3)
[ 2] n<Arg>
0000 trace            1                                               (   2)
0002 putself
0003 getdynamic       n, 0
0006 send             :puts, 1, nil, 8, <ic:0>
0012 leave
```

Now you can see that Ruby has displayed the two YARV instruction snippets separately: the first one corresponds to the global scope, and the second to the inner block scope.

Another important detail to learn about here is the "local table." This shows a listing of the variables that are available in each scope. In my "`10.times do`" example, the local table for the inner scope contains a single variable: "n<Arg>" - the block parameter. The "<Arg>" text indicates that n is a parameter to this block. The text "argc: 1 [opts: 0, rest: -1, post: 0, block: -1]" describes what kind of arguments were passed to the method or block that this YARV code snippet corresponds to. Here's how it works:

- "argc" indicates the total number of arguments.
- "opts" shows the count of optional variables that were passed in, e.g. "var=1, var2=2."
- "rest" shows the number of arguments included by the splat operator, e.g. "*args."
- "post" shows the number of arguments that appear after the splat operator, e.g. "*args, y, z", and
- "block" is true or false indicating whether or not a block was passed in.

It's easier to see how the local table works by creating a few local variables in a Ruby script and then compiling it:

```
code = <<END
a = 2
b = 3
c = a+b
END
puts RubyVM::InstructionSequence.compile(code).disasm
```

Running, I get:

```
== disasm: <RubyVM::InstructionSequence:<compiled>@<compiled>>==========
local table (size: 4, argc: 0 [opts: 0, rest: -1, post: 0, block: -1] s1)
[ 4] a          [ 3] b          [ 2] c
0000 trace            1                                               (   1)
0002 putobject        2
0004 setlocal         a
0006 trace            1                                               (   2)
0008 putobject        3
0010 setlocal         b
```

```
0012 trace            1                                               (   3)
0014 getlocal         a
0016 getlocal         b
0018 opt_plus         <ic:1>
0020 dup
0021 setlocal         c
0023 leave
```

Notice that the local table now contains three variables: “a,” “b,” and “c:”

```
local table (size: 4, argc: 0 [opts: 0, rest: -1, post: 0, block: -1] s1)
[ 4] a          [ 3] b          [ 2] c
```

These are the three local variables created by my Ruby code. You should also note YARV uses the instructions `setlocal` and `getlocal` to set and get local variables. One confusing detail here is that the local table size is shown as 4, even though I have only defined three variables. YARV uses the extra space in the locals table when it executes your code – I’ll cover this in detail in Chapter 2.

Another important detail about the RubyVM output worth learning about are “catch tables.” These have to do with how YARV implements program control features such as `redo`, `next`, `break`, `throw/catch`, `raise/rescue`, etc. Let’s try adding a `redo` statement to the inner block in my example program…

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

… and see how Ruby compiles that:

```
== disasm: <RubyVM::InstructionSequence:<compiled>@<compiled>>==========
== catch table
| catch type: break  st: 0002 ed: 0010 sp: 0000 cont: 0010
|------------------------------------------------------------------------
0000 trace            1                                               (   1)
0002 putobject        10
0004 send             :times, 0, block in <compiled>, 0, <ic:0>
0010 leave
```

```
== disasm: <RubyVM::InstructionSequence:block in <compiled>@<compiled>>=
== catch table
| catch type: redo   st: 0000 ed: 0020 sp: 0000 cont: 0000
| catch type: next   st: 0000 ed: 0020 sp: 0000 cont: 0020
|------------------------------------------------------------------------
local table (size: 2, argc: 1 [opts: 0, rest: -1, post: 0, block: -1] s3)
[ 2] n<Arg>
0000 trace            1                                               (   2)
0002 putself
0003 getdynamic       n, 0
0006 send             :puts, 1, nil, 8, <ic:0>
0012 pop
0013 trace            1                                               (   3)
0015 jump             17
0017 jump             0
0019 putnil
0020 leave
```

You can see the output “catch type: redo” and “catch type:next” at the start of the block’s YARV code snippet. These indicate where the control should jump to if a redo or next statement is compiled inside the block. Since the “catch type: redo” line ended with “cont: 0000” the jump statement on line 17 is “jump 0”. Curiously, Ruby adds an extra, unnecessary “jump 17” instruction on line 15; this must be due to a minor inefficiency or bug in the compiler.

Finally, if we use `break` instead of `redo`:

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

… then we get this output:

```
== disasm: <RubyVM::InstructionSequence:<compiled>@<compiled>>==========
== catch table
| catch type: break  st: 0002 ed: 0010 sp: 0000 cont: 0010
|------------------------------------------------------------------------
0000 trace            1                                               (   1)
0002 putobject        10
```

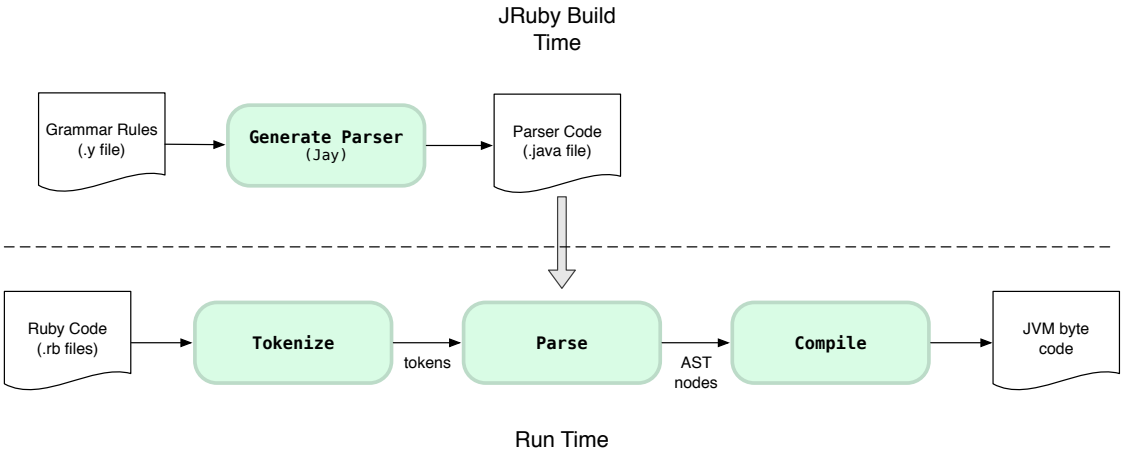
```
0004 send              :times, 0, block in <compiled>, 0, <ic:0>
0010 leave
== disasm: <RubyVM::InstructionSequence:block in <compiled>@<compiled>>=
== catch table
| catch type: redo   st: 0000 ed: 0018 sp: 0000 cont: 0000
| catch type: next   st: 0000 ed: 0018 sp: 0000 cont: 0018
|------------------------------------------------------------------------
local table (size: 2, argc: 1 [opts: 0, rest: -1, post: 0, block: -1] s3)
[ 2] n<Arg>
0000 trace             1                                               (   2)
0002 putself
0003 getdynamic        n, 0
0006 send              :puts, 1, nil, 8, <ic:0>
0012 pop
0013 trace             1                                               (   3)
0015 putnil
0016 throw             2
0018 leave
```

This looks similar, but now Ruby has created a `throw` instruction at the end of the inner block, which will cause YARV to jump out of the block and back up to the top scope, since that scope contains a “catch type: break” line. Since the line shows “cont: 0010” Ruby will continue from line 0010 after executing the throw statement.

I’ll explain how this works in more detail next in Chapter 2.

Tokenization, parsing and compilation in JRuby

Although JRuby uses a completely different technical platform than MRI Ruby does - JRuby uses Java while MRI Ruby uses C - it tokenizes and parses your code in much the same way. Once your code is parsed, JRuby and MRI both continue to compile your code into byte code instructions. As I explained above, Ruby 1.9 and Ruby 2.0 compile your Ruby code into byte code instructions that Ruby's custom YARV virtual machine executes. JRuby, however, instead compiles your Ruby code into Java byte code instructions that are interpreted end executed by the Java Virtual Machine (JVM):

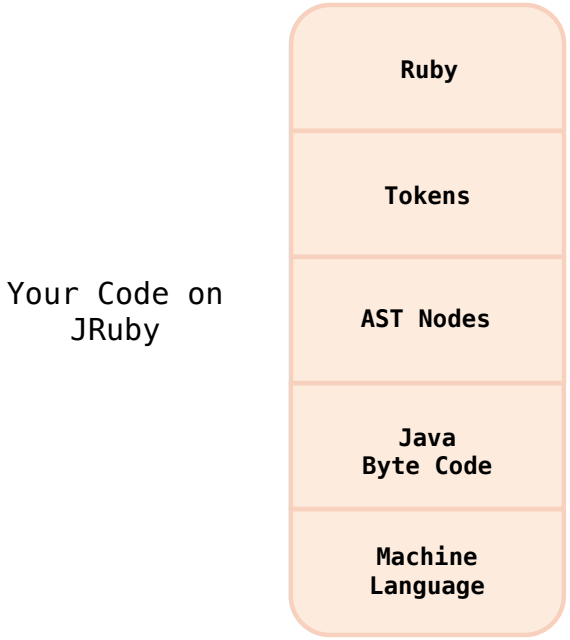


Just like with Ruby 2.0, 1.9 and 1.8, JRuby uses a two step process for tokenizing and parsing. First, ahead of time during the JRuby build process a tool called Jay generates LALR parser code based on a grammar file, in just the same way that MRI Ruby uses Bison. In fact, Jay is really just a rewrite of Bison that generates a parser that uses Java or C# code instead of C. For JRuby the grammar file is called DefaultRubyParser.y instead of parse.y and the generated parser code is saved in a file called DefaultRubyParser.java instead of parse.c. Note: if you run JRuby in 1.9 compatibility mode, the new default for the JRuby head/master build, JRuby will use a different file called Ruby19Parser.y instead. The JRuby team more or less copied over the grammar rules from MRI Ruby's parse.y into DefaultRubyParser.y and Ruby19Parser.y - this is not a surprise since JRuby aims to implement Ruby in a completely compatible way.

Then, once you have installed JRuby on your machine including the generated parser, JRuby will run the parser to tokenize and parse your Ruby script. First JRuby will read the text from your Ruby file and generate a stream of tokens, and next the generated

LALR parser will convert this stream of tokens into an AST structure. This all works essentially the same as it does in MRI Ruby.

Here's a high level view of the different forms your Ruby code takes as you run a JRuby process:



At the top you can see JRuby converts your Ruby code into a token stream and then, in turn, into an AST structure. Next, JRuby compiles these AST nodes into Java byte code, which are later interpreted and executed by the JVM - the same VM that runs Java programs along with many other programming languages such as Clojure and Scala.

I didn't include the "interpret" dotted line in this diagram that appears in the analogous Ruby 1.8 and Ruby 1.9 diagrams, because the JVM's JIT ("Just In Time") compiler actually converts some of that Java byte code - the compiled version of your Ruby program - into machine language. The JVM will take the time to do this for "hotspots" or frequently called Java byte code functions. For this reason, JRuby can often run faster than MRI Ruby even though it's implemented in Java and not C, especially for long running processes. What this means is that it's possible for JRuby and the JVM working together to convert part of the Ruby code you write all the way into native machine language code!

Taking a look at the JRuby tokenizing and parsing code details, the similarity to MRI is striking. The only real difference is that JRuby is written in Java instead of C. For example, here's some of the code that JRuby uses to tokenize the stream of characters read in from the target Ruby code file - you can find this in RubyYaccLexer.java in the src/jruby/org/jruby/lexer/yacc folder.

```
loop: for(;;) {
    c = src.read();
    switch(c) {


...


    case ',':
        return comma(c);
```

Just like the `parser_yylex` function in MRI Ruby, the RubyYaccLexer Java class uses a giant switch statement to branch based on what character is read in. Above is the start of this switch statement, which calls `src.read()` each time it needs a new character, and one case of the switch statement that looks for comma characters. The JRuby code is somewhat simpler and cleaner than the corresponding MRI Ruby code, since it uses object oriented Java vs. standard C. For example, tokens are represented by Java objects - here's the `comma` function called from above which returns a new comma token:

```
private int comma(int c) throws IOException {
    setState(LexState.EXPR_BEG);
    yaccValue = new Token(",", getPosition());
    return c;
}
```

It's a similar story for parsing: the same idea using a different programming language. Here's a snippet from the DefaultRubyParser.y file - this implements the same `method_call` grammar rule that I discussed in detail earlier for MRI Ruby:

```
method_call   :
```

```
...

                | primary_value tDOT operation2 opt_paren_args {
                    $$ = support.new_call($1, $3, $4, null);
                }
```

Since JRuby uses Jay instead of Bison, the code that JRuby executes when there's a matching rule is Java and not C. But you can see Jay uses the same "`$$, $1, $2`, etc." syntax to specify the return value for the grammar rule, and to allow the matching code to access the values of each of the child rules.

Again, since the matching code is written in Java and not C, it's generally cleaner and easier to understand compared to the same code you would find in MRI Ruby. In the snippet above, you can see JRuby creates a new `call` AST node to represent this method call. In this case the `support` object, an instance of the `ParserSupport` class, actually creates the AST node. Instead of C structures, JRuby uses actual Java objects to represent the nodes in the AST tree.

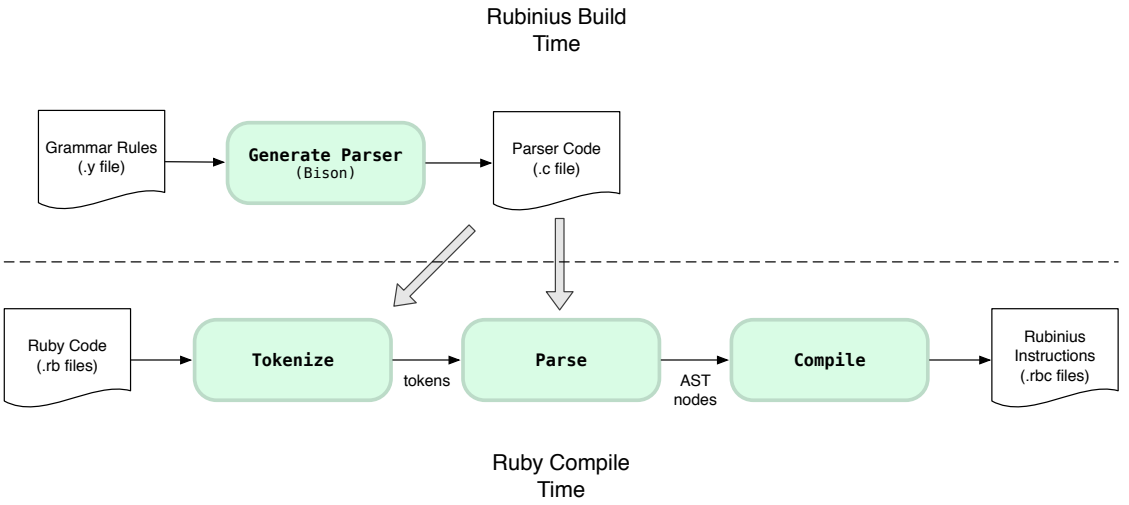
JRuby's Ruby to JVM byte code compiler, however, doesn't resemble the YARV compiler code I explained earlier in Chapter 1 very much. Instead, the JRuby team implemented a new, custom compiler - it walks the AST node tree in a similar way, but outputs JVM byte code instructions instead of YARV instructions. Generally these byte code instructions are more granular and low-level compared to the YARV instructions - i.e. each instruction does less and there are more of them. This is due to the nature of the JVM, which was designed to run not only Java but also many other languages. The YARV instructions, as we've seen, are designed specifically for Ruby. If you're interested in exploring JRuby's compiler, look in the org.jruby.compile package in your copy of the JRuby source tree.

The JRuby core team is also currently working on a new higher-level and less granular instruction set called "IR," which will be specifically designed to represent Ruby programs. To learn more about the new IR instruction set see the article OSS Grant Roundup: JRuby's New Intermediate Representation.

Tokenization, parsing and compilation in Rubinius

Now let's take a look at Rubinius and how it parses your Ruby code. You may have heard that Rubinius is a version of Ruby implemented with Ruby, but did you know this also applies to the compiler that Rubinius uses? That's right: as hard as it is to imagine, when you run a Ruby script using Rubinius, it compiles your Ruby code using Ruby.

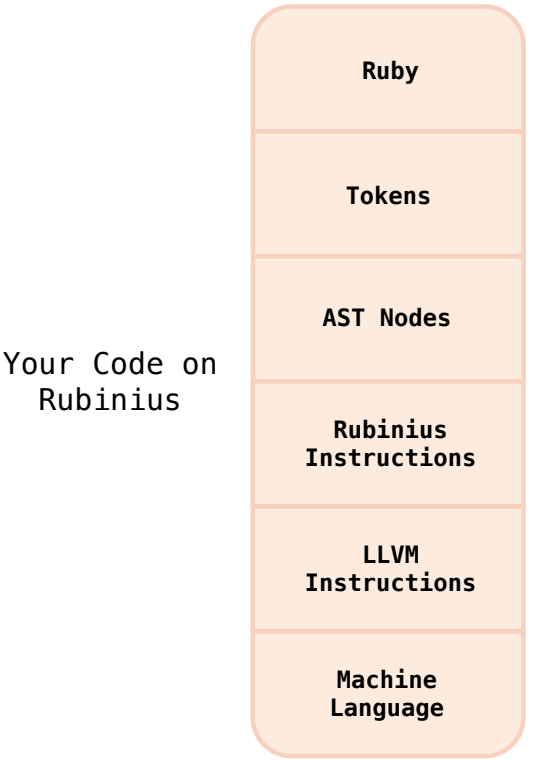
At a high level the process looks very similar to MRI and JRuby:



Again at build time, before you ever run your Ruby program, Rubinius generates an LALR parser using Bison – the same tool that MRI Ruby uses. Just like JRuby, the Rubinius team has more or less copied the same grammar rules over from the original MRI parse.y file. In Rubinius the grammar file is called either "grammar18.y" or "grammar19.y" – just like JRuby, Rubinius maintains two copies of the grammar rules for its 1.8 and 1.9 compatibility modes.

Later when you run your Rubinius process, it converts your code again into a token stream, an AST structure, and later into high level instructions called "Rubinius instructions." One nice feature of Rubinius is that it allows you to save these compiled instructions into special ".rbc" files. That is, Rubinius exposes a compile command, and allows you to precompile your Ruby code before you actually run it, if you prefer, saving some time later. Remember that MRI didn't provide this feature: Ruby 1.9 and 2.0 always compile your code every time you run it.

But what makes Rubinius fascinating is the way that it implements Ruby using Ruby, or more precisely a combination of C, C++ and Ruby. I'll have more examples of this later in other chapters, but for now let's take a look at how Rubinius parses and compiles your code. Here's the same diagram I had for MRI and JRuby showing all the different forms your code takes internally inside of Rubinius when you run it:



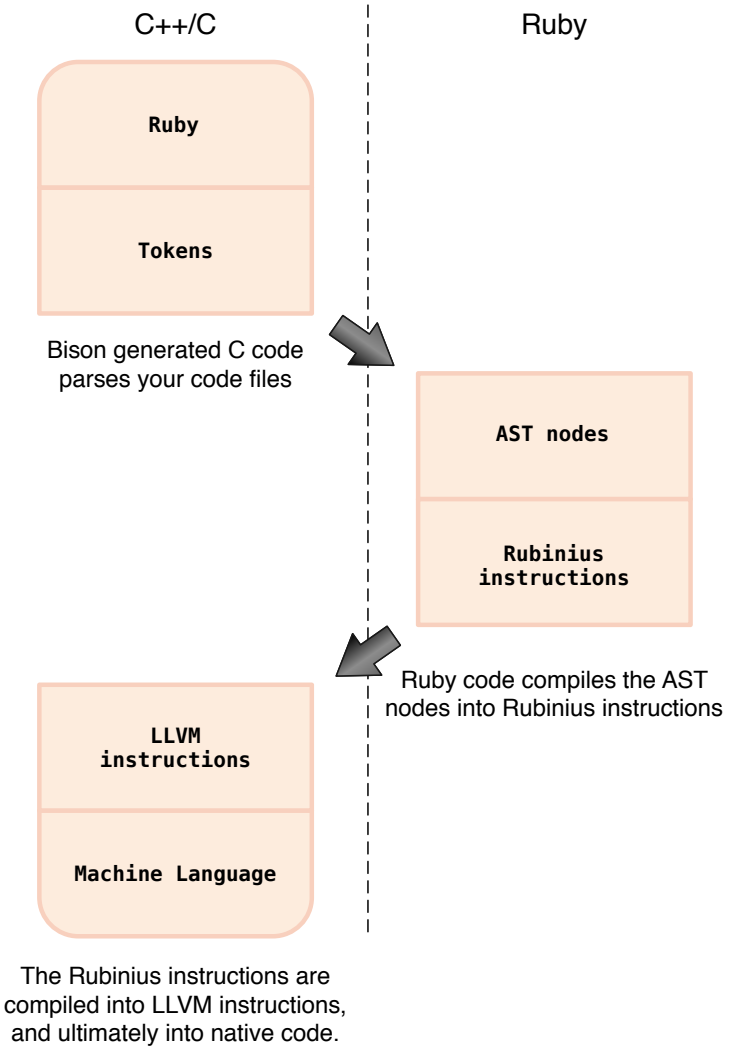
When you run a Ruby script using Rubinius your code is converted into all of these different formats, and ultimately into machine language! At the top, the picture is the same: your Ruby script is once again tokenized and parsed, and converted into a similar AST structure. Next, Rubinius iterates through the AST nodes, compiling them into high level instructions which I'll call "Rubinius instructions." These are similar to the YARV instructions that Ruby 1.9 and 2.0 use internally, except as I mentioned above they can optionally be saved into .RBC files for later use.

Then in order to execute these instructions, Rubinius uses a well known and very powerful open source framework called the "Low Level Virtual Machine" or LLVM. The LLVM framework includes a number of different, powerful tools that make it easy – or at least easier – to write a language compiler. LLVM provides a low-level instruction set, a

virtual machine to execute these instructions along with optimizers, a C/C++ compiler (Clang), a debugger and more.

Rubinius primarily leverages the LLVM virtual machine itself by converting the high level Rubinius instructions into low level LLVM instructions using a JIT ("just in time") compiler written by the Rubinius team. That is, first your Ruby code is parsed and compiled into Rubinius instructions; later Rubinius converts these high level instructions into their equivalent low level LLVM instructions using a background thread as your Rubinius process runs.

As we'll continue to see in later chapters, Rubinius's implementation is a *tour de force* – it's an innovative, creative implementation of Ruby that at the same time leverages some of the best open source software available to provide fantastic performance. For me one of the most elegant aspects of Rubinius internals is the way that it seamlessly combines C++, C and Ruby code together – the parsing/compiling process is a good example of this. Here's a closer look at the way Rubinius processes your code:



Inside of Rubinius, parsing and compiling your code is a team effort:

- First, as I mentioned above, Rubinius uses the same Bison generated LALR parser that MRI Ruby does. Rubinius also uses similar C code to first tokenize your code file's text.
- But next, the C code triggered by the matching grammar rules in the parser create AST nodes… that are implemented by Ruby classes! Every type of AST

node has a corresponding Ruby class, all of which have a common Ruby super class: `Rubinius::AST::Node`.

- Next each of these AST node Ruby classes contains code that compiles that type of AST node into Rubinius instructions.
- Finally, once your Rubinius process is running a JIT compiler written in C++ converts these high Rubinius instructions into low level LLVM instructions.

The Rubinius Ruby compiler, itself written in Ruby, is very readable and straightforward to understand. In fact, the fact that much of Rubinius is implemented in Ruby is one of its most important features. To see what I mean, take a look at how the `send` AST node – or method call – is compiled into high level Rubinius instructions:

```
module Rubinius
  module AST
    class Send < Node


...


      def bytecode(g)
        pos(g)
        if @vcall_style and reference = check_local_reference(g)
          return reference.get_bytecode(g)
        end
        @receiver.bytecode(g)
        if @block
          @block.bytecode(g)
          g.send_with_block @name, 0, @privately
        elsif @vcall_style
          g.send_vcall @name
        else
          g.send @name, 0, @privately
        end
      end


...
```

This is a snippet from the lib/compiler/ast/sends.rb Rubinius source code file. This class, `Rubinius::AST::Send`, implements the `Send` Rubinius AST node that the parser creates when it encounters a method or function call in your Ruby script. You can see the reference to the `Rubinius::AST::Node` super class.

I won't explain every detail, but at a high level the way this works is:

- When Rubinius compiles the AST nodes into Rubinius instructions, it visits every AST node object and calls their `bytecode` methods, passing in a generator object or "g" here. The generator object provides a DSL for creating Rubinius instructions, e.g. `send_with_block` or `send`.
- After checking for the case where the function call might actually be a reference to a local variable, Rubinius calls `@receiver.bytecode` – this compiles the receiver object first.
- Then Rubinius creates either a `send_with_block`, `send_vcall` or `send` method depending on various attributes of the node object.

To save space I'm glossing over some details here but it's real pleasure reading the Ruby compiler code inside Rubinius since it's so easy to understand and follow. Again, you can find all of the AST node Ruby classes in the lib/compiler/ast folder in your Rubinius source tree.

Chapter 2
How Ruby Executes Your Code

Ruby 1.9 and later use a virtual machine known as "YARV"

Now that Ruby has tokenized, parsed and compiled your code, Ruby is finally ready to execute it. But exactly how does it do this? 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 keep track of variables, return values and arguments? How does it implement `if` statements and other control structures?

Just like your computer's actual microprocessor hardware, Koichi Sasada and the Ruby core team designed YARV to use a stack pointer and a program counter. In this chapter, I'll start by looking at the basics of YARV instructions: how they pop arguments off the stack and push return values onto the stack. I'll continue by explaining how Ruby accesses variables in two different ways: locally and dynamically. Then I'll show you how YARV implements Ruby control structures – including a look at how Ruby implements the `break` keyword internally by raising an exception! Finally, I'll compare the instruction sets used by the JRuby and Rubinius virtual machines to YARV's instruction set.

Chapter 2 Roadmap

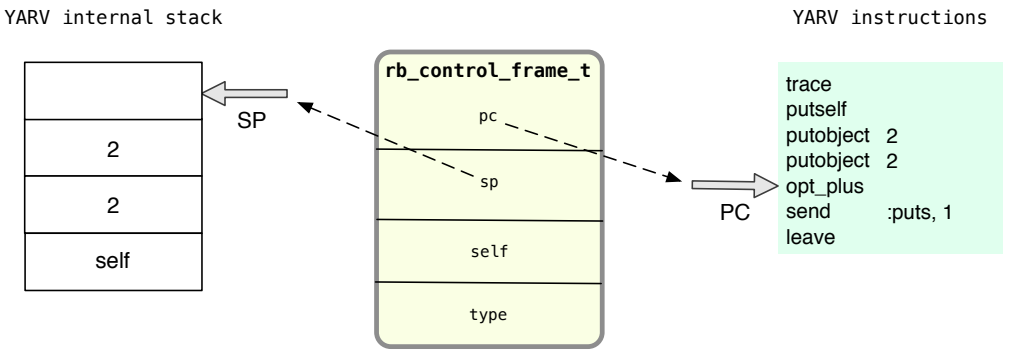
YARV's internal stack and your Ruby stack

Aside from it's own stack, YARV keeps track of your Ruby call stack.

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

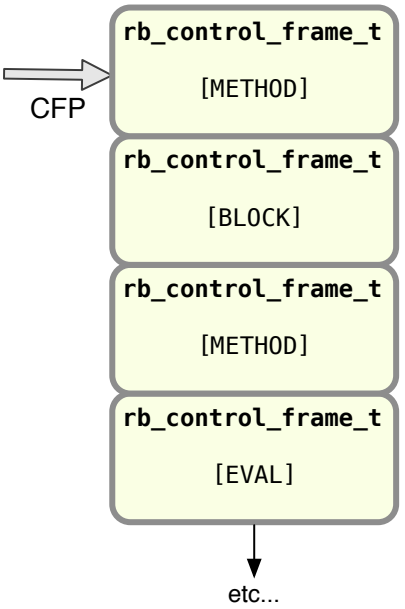
But alongside YARV's internal stack Ruby also keeps track of your Ruby program's call stack: which methods called which other methods, functions, blocks, lambdas, etc. In fact, YARV is not just a stack machine – it's a "double stack machine!" It not only has to track the arguments and return values for it's own internal instructions; it has to do it for your Ruby arguments and return values as well.

First let's take a look at YARV's basic registers and internal stack:



On the left I show YARV's internal stack – `SP` is the "stack pointer" or location of the top of the stack. On the right are the instructions that YARV is currently executing. `PC` is the program counter or location of the current instruction. You can see the YARV instructions that Ruby compiled from my "`puts 2+2`" example from Chapter 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 I'm not showing here.

At the same time YARV maintains another stack of these `rb_control_frame` structures, like this:



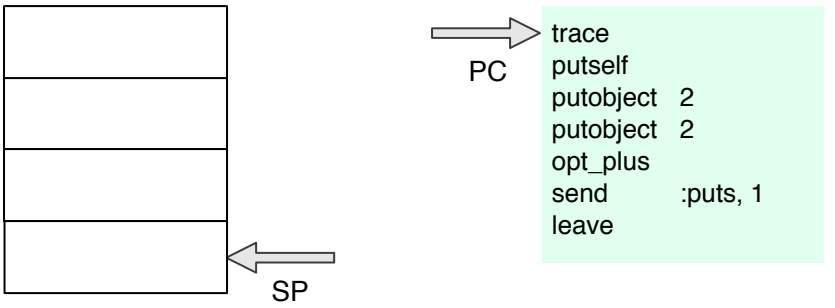
This second stack represents the path through your Ruby program YARV has taken and it's current location. In other words, this is your Ruby call stack – what you would see if you ran "`puts caller`." The CFP pointer indicates the "current frame pointer." Each stack frame in your Ruby program stack contains, in turn, a different value for the `self`, `PC` and `SP` registers we saw above. The `type` field in each `rb_control_frame_t` structure indicates what type of code is 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

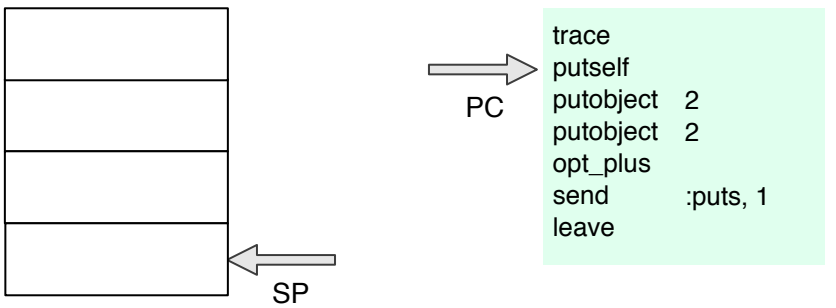
To understand all of this better, let's run through a couple examples. I'll start with my simple 2+2 example:

```
puts 2+2
```

This one line Ruby script doesn't have any Ruby call stack, so I'll focus on the internal YARV stack only for now. Here's how YARV will execute this script, starting with the first instruction, `trace`:

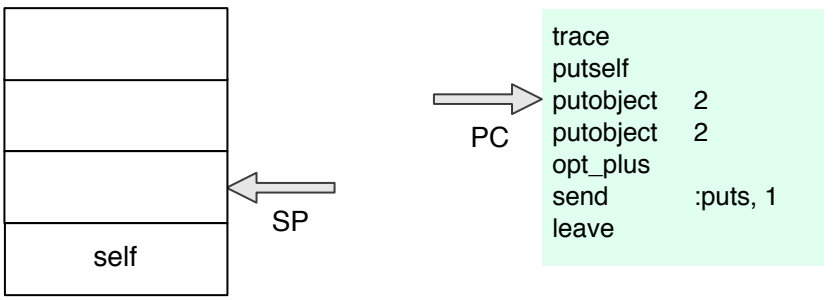


You can see here YARV starts the PC or program counter at the first instruction, and initially the stack is empty. Now YARV will execute the `trace` instruction, incrementing the PC register:



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, or when a few other events occur.

Next YARV will execute `putself` and push the current value of `self` onto the stack:

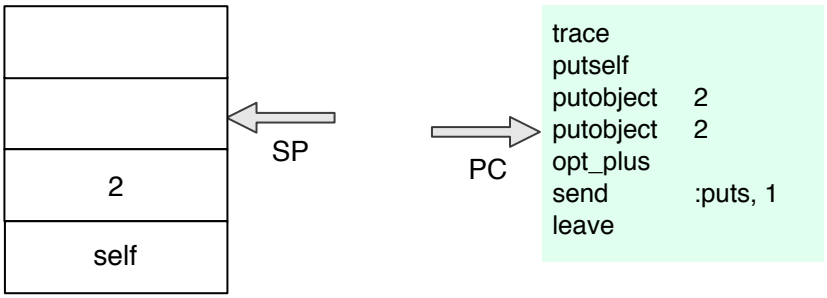


Since this simple script contains no Ruby objects or classes the `self` pointer will be set to the default "top self" object. This is an instance of the `Object` class Ruby automatically creates when YARV starts up. It serves as the receiver for method calls and the container for instance variables in 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 this command at your console:

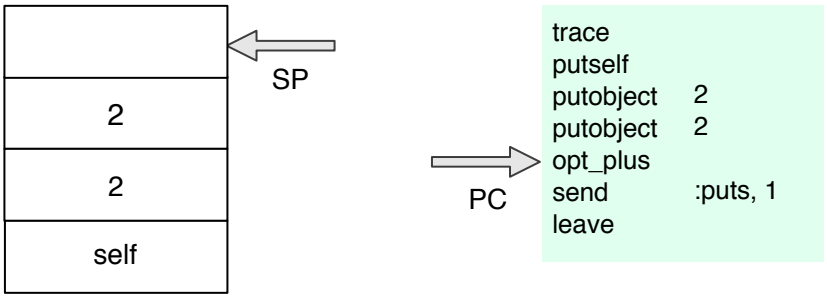
```
$ ruby -e 'puts self'
```

Later YARV will use this `self` value on the stack when it executes the `send` instruction – `self` is the receiver of the `puts` method, since I didn't specify a receiver for this method call.

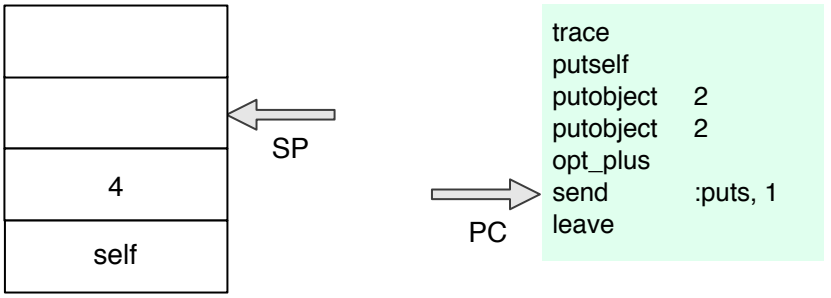
Next YARV will execute "`pushobject 2`" and push the numeric value `2` onto the stack, and increment the PC again:



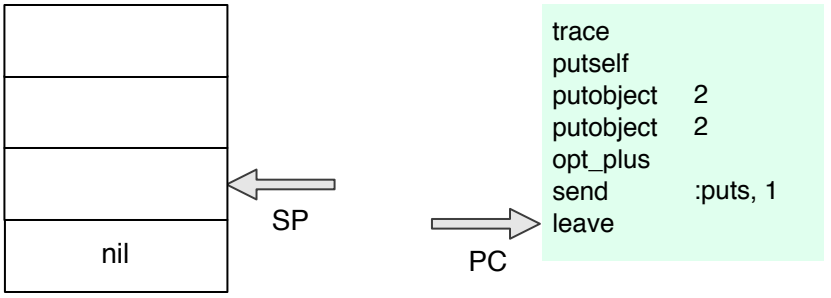
This is the first step of the receiver – arguments – operation pattern I described in Chapter 1. First Ruby pushes the receiver onto the internal YARV stack; in this example the `Integer` object `2` is the receiver of the message/method `plus` which takes a single argument, also a `2`. Next Ruby will push the argument `2`:



And finally it will execute the operation – in this case `opt_plus` is an special, optimized instruction that will add two values: the receiver and argument.



You can see the `opt_plus` instruction leaves the result, 4, at the top of the stack. And now, as I explained in Chapter 1, 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 3, but for now let's just step ahead:



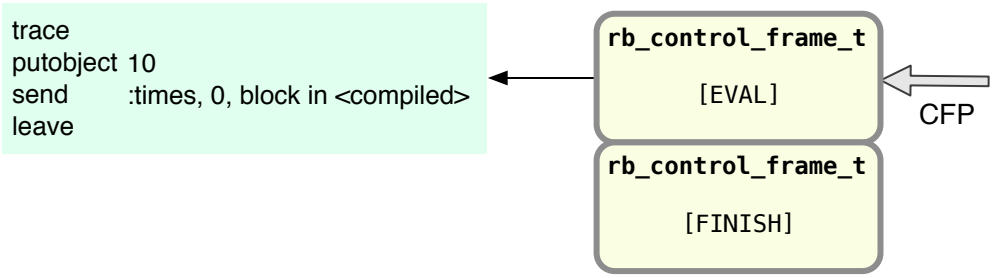
Here the `send` instruction has left the return value, nil, at the top of the stack. Finally Ruby executes the last instruction `leave`, which finishes up executing our simple, one line Ruby program.

Executing a call to a block

Now let's take a slightly more complicated example and see how the other stack – your Ruby program stack – works. Here's a simple Ruby script that calls a block 10 times, printing out a string:

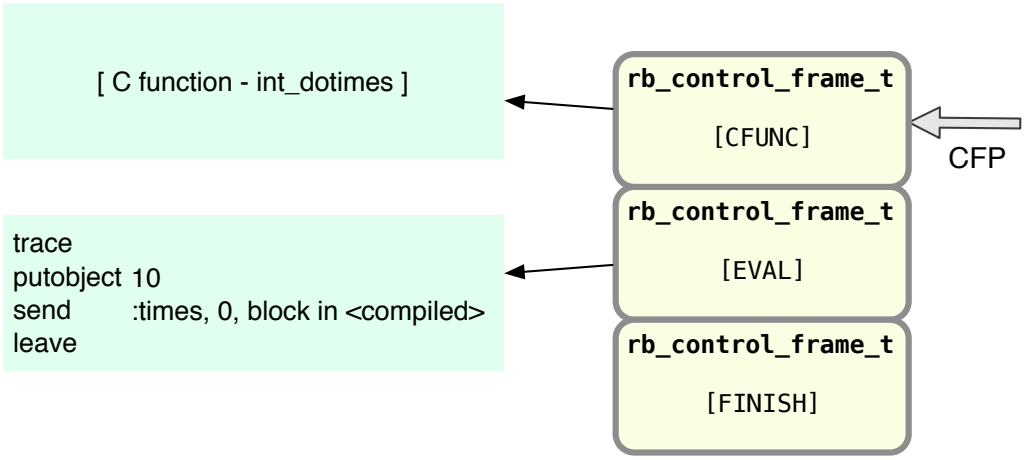
```
10.times do
  puts "The quick brown fox jumps over the lazy dog."
end
```

Let's skip over a few steps and start off where YARV is about to call the `times` method:



On the left are the YARV instructions Ruby is executing, and now on the right I'm showing two control frame structures. At the bottom of the stack is a control frame with the type set to `FINISH` – Ruby always creates this frame first when starting a new program. At the top of the stack initially is a frame of type `EVAL` – this corresponds to the top level or main scope of your Ruby script. Internally, Ruby uses the `FINISH` frame to catch any exceptions that your Ruby code might throw, or to catch exceptions generated by a `break` or `return` keyword. I'll have more on this in section 2.3.

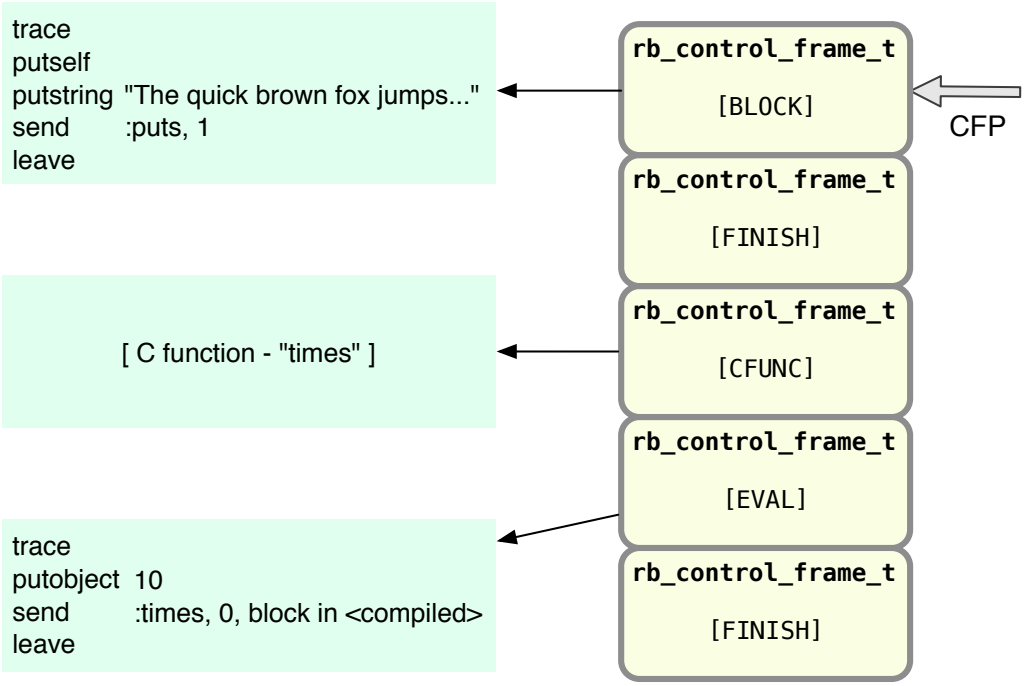
Next when Ruby calls the `times` message on the Integer object `10` the receiver of the `times` message, it will add a new level to the control frame stack:



This new entry on the right represents a new level in your program's Ruby call stack, and the CFP pointer has moved up to point at the new control frame structure. Also since the `times Integer` method is built into Ruby there are no YARV instructions for it. Instead, Ruby will call some internal C code that will pop the argument "10" off the

stack and call the provided block 10 times. Ruby gives this control frame a type of `CFUNC`.

Finally, if we interrupt the program inside the inner block here's what the YARV and control frame stacks will look like:



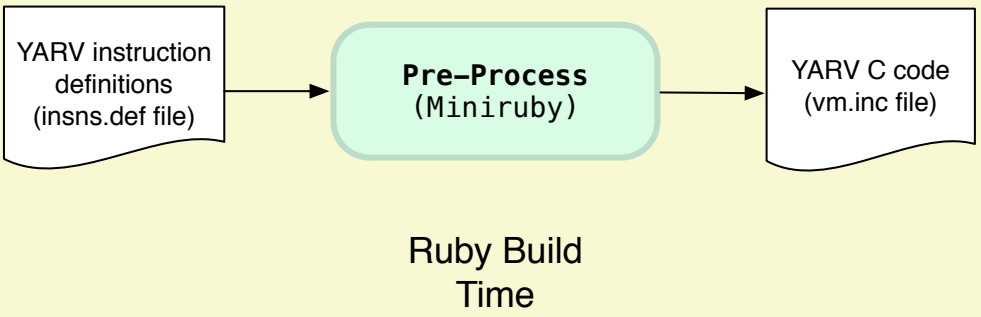
You can see there will now be five entries in the control frame stack on the right:

- the `FINISH` and `EVAL` frames that Ruby always starts up with,
- the `CFUNC` frame for the call to `10.times`,
- another `FINISH` frame; Ruby uses this one to catch and exceptions or calls to return or break that might occur inside the block, and
- a `BLOCK` frame; This frame at the top of the stack corresponds to the code running inside the block.

Like most other things, Ruby implements all of the YARV instructions like `putobject` or `send` using C code which 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. For example, here's a small snippet from insns.def showing how 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)
{
    val = GET_SELF();
}
```

This doesn't look like C at all – in fact, most of it is not. Instead, what you see here is a bit of C code ("`val = GET_SELF()`") 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 how Bison converts the parse.y file into parse.c:



Ruby processes the insns.def file using Ruby: the Ruby build process first compiles a smaller version of Ruby called “Miniruby,” and then uses this to run some Ruby code that processes insns.def and converts it into a C source code file called vm.inc. Later the Ruby build process hands vm.inc to the C compiler which includes the generated C code in the final, compiled version of Ruby.

Here’s what the snippet above for `putself` looks like in vm.inc after Ruby has processed it:

```
INSN_ENTRY(putself){
{
  VALUE val;
  DEBUG_ENTER_INSN("putself");
  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
  USAGE_ANALYSIS_INSN(BIN(putself));
{
#line 323 "insns.def"
    val = GET_SELF();
#line 474 "vm.inc"
  CHECK_STACK_OVERFLOW(REG_CFP, 1);
  PUSH(val);
#undef CURRENT_INSN_putself
#undef INSN_IS_SC
#undef INSN_LABEL
#undef LABEL_IS_SC
  END_INSN(putself);}}}
```

The single line “`val = GET_SELF()`” appears in the middle, while above and below this Ruby calls a few different C macros to do various things, like adding one to the program counter (`PC`) register, and pushing the `val` value onto the YARV internal stack. The vm.inc C source code file, in turn, is included by the vm_exec.c file, which contains the primary YARV instruction loop: the loop that steps through the YARV instructions in your program one after another and calls the C code corresponding to each one.

Experiment 2-1: Benchmarking Ruby 1.9 vs. Ruby 1.8

The Ruby core team introduced the YARV virtual machine with Ruby 1.9; before that Ruby 1.8 and earlier versions of ruby executed your program by directly stepping through the nodes of the Abstract Syntax Tree (AST). There was no compile step at all; Ruby just tokenized, parsed and then immediately executed your code. Ruby 1.8 worked just fine; in fact, for years Ruby 1.8 was the most commonly used version of Ruby. Why did the Ruby core team do all of the extra work required to write a compiler and a new virtual machine? The answer is simple: speed. Executing a compiled Ruby program using YARV is much faster than walking around the AST directly.

How much faster is YARV? Let's take a look… in this experiment I'll measure how much faster Ruby 1.9 is compared to Ruby 1.8 by executing this very simple Ruby script:

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

Here I'm passing in a count value on the command line via the `ARGV` array, and then just iterating 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]` I 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.

By using the `time` Unix command I can measure how long it takes Ruby to iterate 1 time:

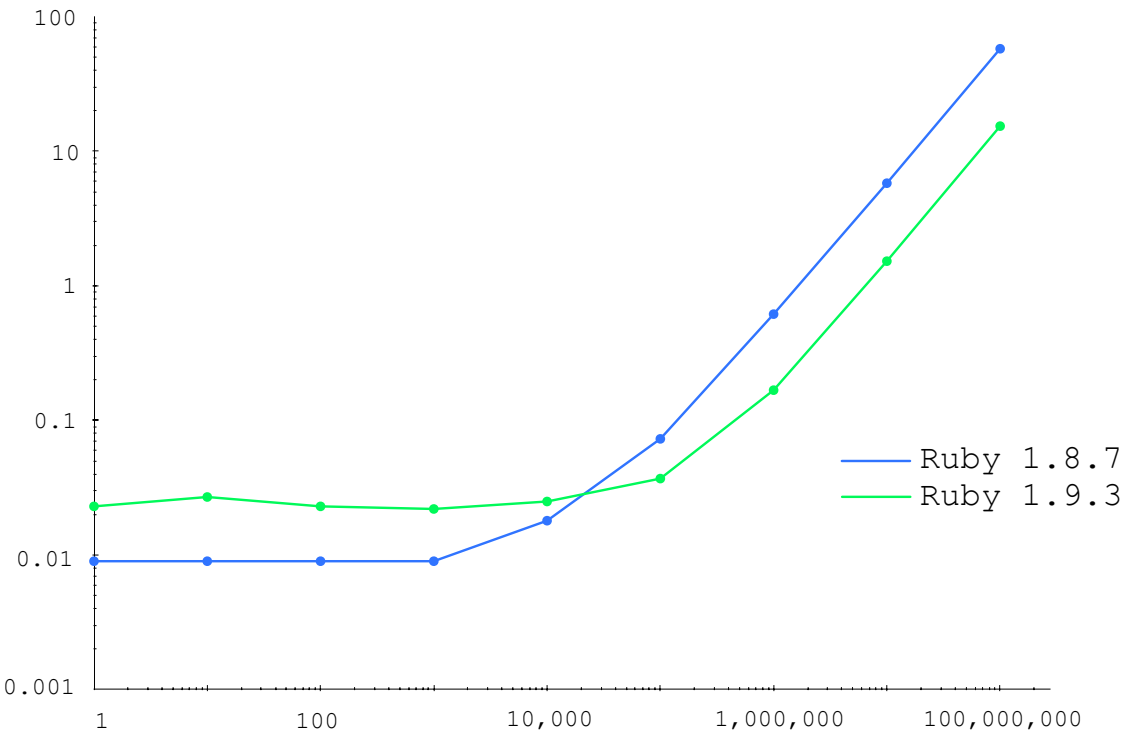
```
$ time ruby benchmark1.rb 1
ruby benchmark1.rb 1  0.02s user 0.00s system 92% cpu 0.023 total
```

…or 10 times:

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

…etc…

Plotting the times on a logarithmic scale for Ruby 1.8.7 and Ruby 1.9.3, I get:



Time (sec) vs. number of iterations

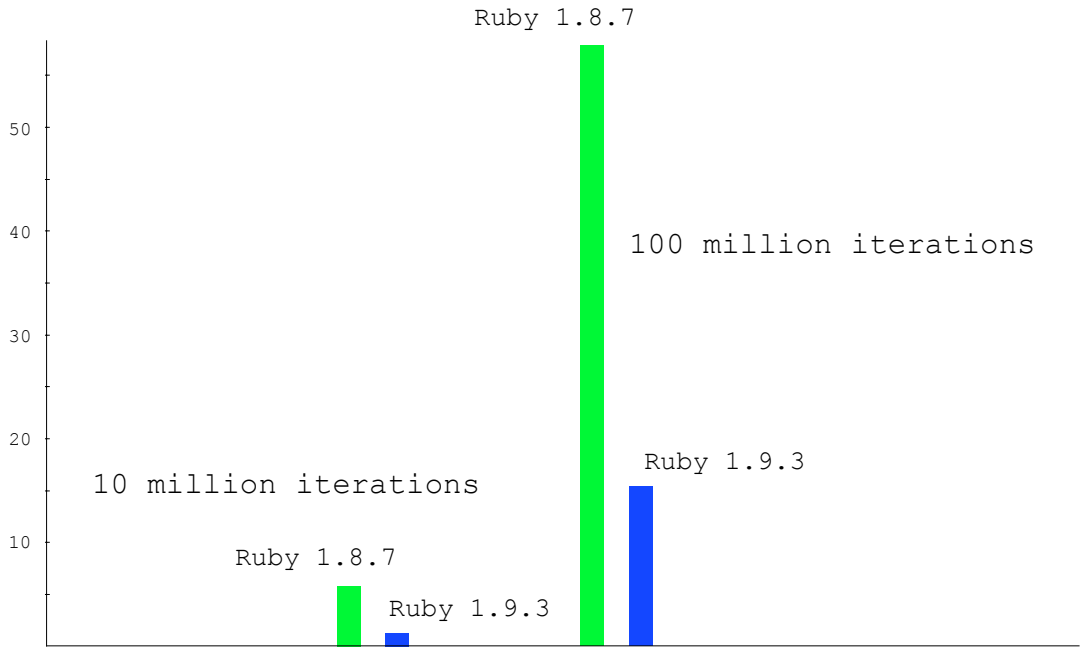
Looking at the chart, you can see that:

- For short lived processes, i.e. loops with a small number of iterations shown on the left, Ruby 1.8.7 is actually faster than Ruby 1.9.3, since there is no need to compile the Ruby code into YARV instructions at all. 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 at the left side of the chart, about 0.01 seconds, is how long it takes Ruby 1.9.3 to compile the script into YARV instructions.
- However, after a certain point – after about 11,000 iterations – Ruby 1.9.3 is 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, i.e. loops with a large number of iterations shown on the right, Ruby 1.9 is about 3.75 times faster!

This speed up doesn't look like much on the logarithmic chart above, but if I redraw the right side of this chart using a linear scale:



Time (sec) for 10 or 100 million iterations

…you can see the difference is dramatic! Executing this simple Ruby script using Ruby 1.9.3 with YARV is about 3.75 times faster than it using Ruby 1.8.7 without YARV!

Local and dynamic access of Ruby variables

Using dynamic access, Ruby can climb up to access values in the parent scope

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

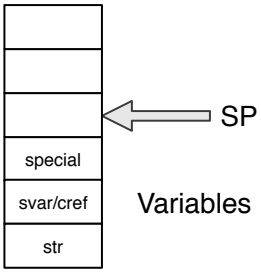
Storing variables is straightforward: Ruby stores all of the values you save in variables on YARV's stack, along with the parameters to and return values from the YARV instructions. However, accessing these variables is not so simple. Internally Ruby uses two very different methods for saving and retrieving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

Local variable access

Let's start with local access first, since that's simpler. Whenever you make a method call, Ruby sets aside some space on the YARV stack for any local variables that are declared inside the method you are calling. Ruby knows how many variables you are using by consulting the "local table" that was created for each method during the compilation step I covered in Chapter 1.

For example, suppose I write a very silly Ruby function to display a string:

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



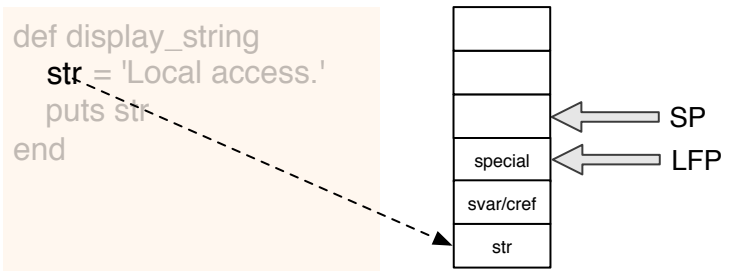
On the left is my Ruby code, and on the right is a diagram showing the YARV stack and stack pointer. You can see that Ruby stores the variables on the stack just under the

stack pointer. Notice there's a space reserved for the `str` value on the stack, three slots under where the `SP` is, in other words at `SP-3`.

Ruby uses the `svar/cref` slot for two different purposes: it might contain a pointer to a table of the "special variables" that exist in the current method. These are values such as `$!` (last exception message) or `$&` (last regular expression match). Or it might contain a pointer to the current lexical scope. Lexical scope indicates which class or module you are currently adding methods to. In Experiment 2-2 I'll explore what special variables are and how they work.

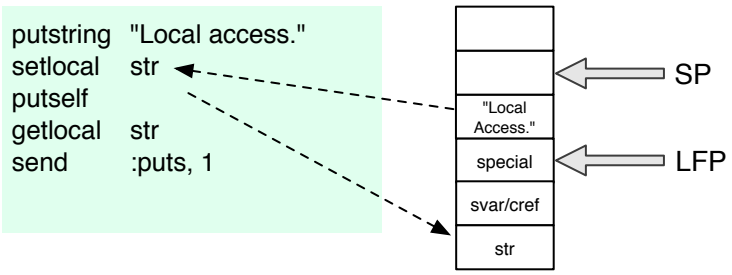
Ruby uses the first slot – the "special" variable – to keep track of information related to blocks. I'll have more about this in a moment when I discuss dynamic variable access.

When my example code saves a value into `str`, Ruby just needs to write the value into that space on the stack:



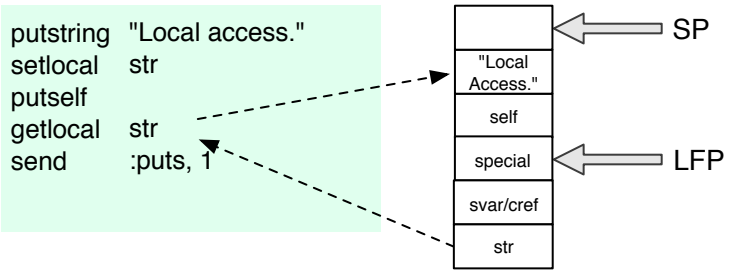
Internally YARV uses another pointer similar to the stack pointer called the `LFP` or "Local Frame Pointer." This points to where the local variables for the current method are located on the stack. Initially it is set to `SP-1`. Later the value of `SP` will change as YARV executes instructions, while the `LFP` value will normally remain constant.

Here are the YARV instructions that Ruby compiled my `display_string` function into:



First the `putstring` instruction saves the "Local access" string on the top of the stack, incrementing the `SP` pointer. Then you can see 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. Internally, `setlocal` uses the `LFP` pointer and a numerical index indicating which variable to set – in this example that would be: "`address of str = LFP-2`."

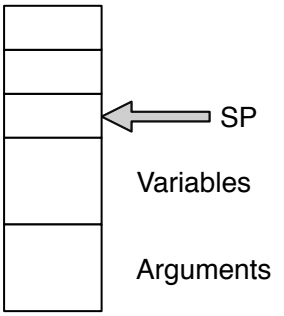
Next for the call to "`puts str`" Ruby uses the `getlocal` instruction:



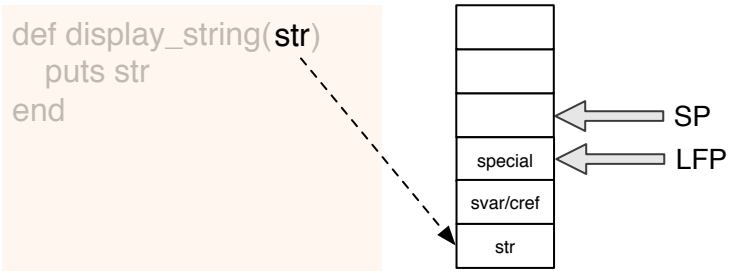
Here Ruby has pushed the string value back onto the top of the stack, where it can be used as an argument for the call to the `puts` function.

The works the same way if I instead pass the string in as a method parameter – method arguments are essentially the same as local variables:

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



The only difference between method arguments and local variables is that the calling code pushes the arguments onto the stack before the method calls even occurs. In this example, there are no local variables, but the single argument appears on the stack just like a local variable:

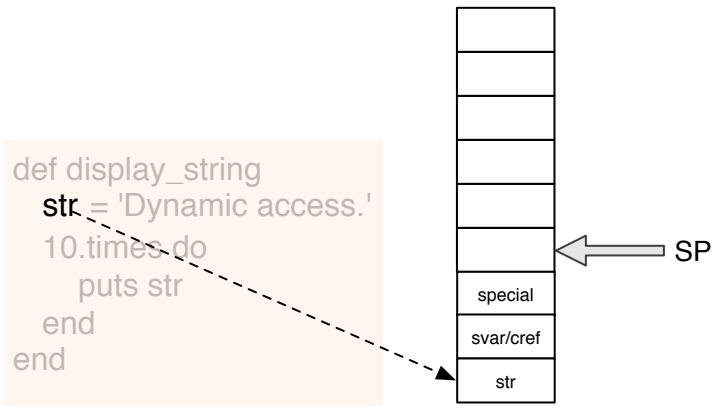


Dynamic variable access

Now let's take a look at 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 parent scope. Here's an example:

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

Here `str` is again a local variable in `display_string`, and Ruby will save it using the `setlocal` instruction we saw above.

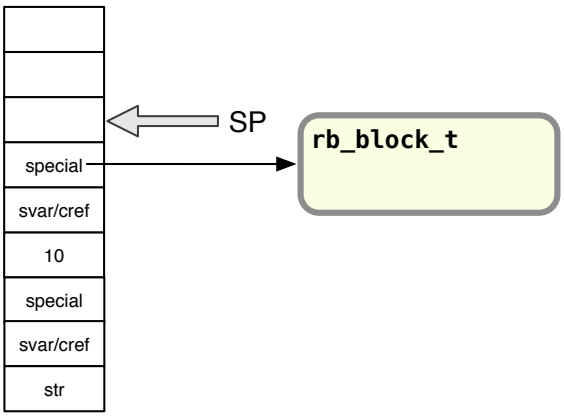


However, now I'm calling "`puts str`" from inside a block. To access the `str` local variable from the block, Ruby will have to use dynamic access to reach the stack frame

for the parent scope. Before explaining exactly how dynamic access works, let's first step through the process of calling the block to see how Ruby sets up the stack.

First Ruby will call the `10.times` method, passing a block in as an argument:

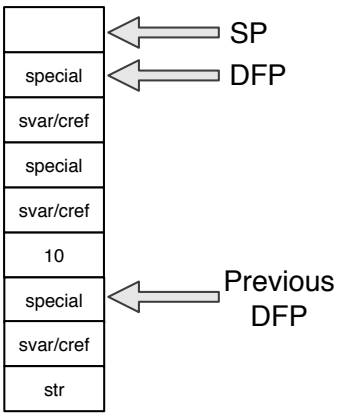
```
def display_string
  str = 'Dynamic access.'
  10.times do
    puts str
  end
end
```



First, notice the value 10 on the stack – this is the actual receiver of the method `times`. You can also see just above that Ruby has created a new stack frame on the right for the C code that implements `Integer#times` to use. Since I passed a block into the method call, Ruby saves a pointer to this block in the "special" variable on the stack. Each frame on the YARV stack corresponding to a method call keeps track of whether or not there was a block argument using this "special" variable. I'll cover blocks and the `rb_block_t` structure in much more detail in Chapter 5.

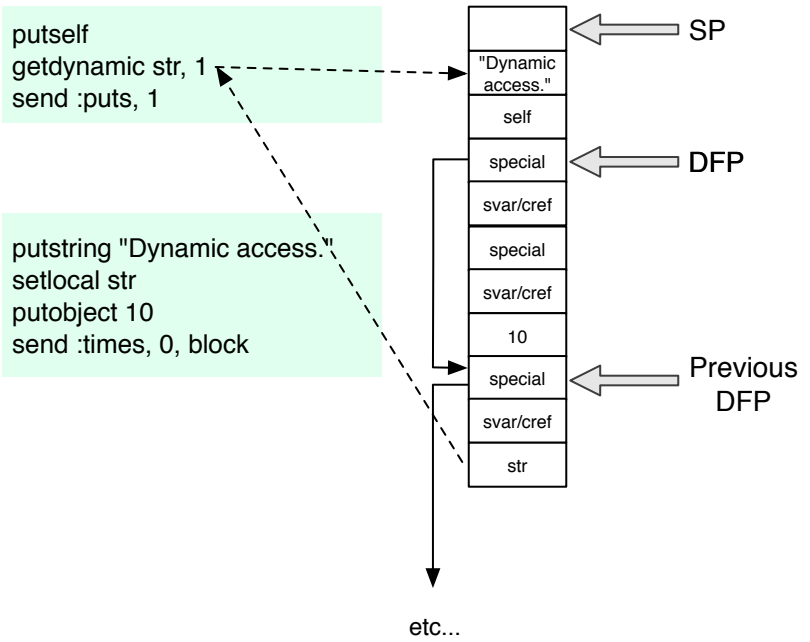
Now Ruby will call the block's code over and over again, 10 times:

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



You can see here that, as I explained in section 2.1, Ruby actually creates two new stack frames when you call a block: a `FINISH` frame and a `BLOCK` frame. The first `FINISH` frame is more or less a copy of the previous stack frame, holding the block as a parameter in the "special" variable. But when Ruby starts to execute the block itself, it changes the "special" variable to become something else: a pointer to the parent scope's stack frame. This is known as the DFP or Dynamic Frame Pointer.

Ruby uses the DFP to enable dynamic variable access. Here are the YARV code instructions Ruby compiled my block into:



The dashed arrows indicate Ruby's dynamic variable access: the `getdynamic` YARV instruction copies the value of `str` from the lower stack frame, from the parent or outer Ruby scope, up to the top of the stack, where the YARV instructions in the block can access it. Note how the `DFP` pointers, in a sense, form a ladder that Ruby can climb to access the local variables in the parent scope, or the grandparent scope, etc.

In the "`getdynamic str, 1`" call above, the second parameter 1 indicates which stack frame or Ruby scope to look for the variable `str` in. Ruby implements this by iterating through the `DFP` pointers that number of times. In this case Ruby moves up one scope before looking for `str`. If I had two nested blocks like this:

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

… then Ruby would have used "`getdynamic str, 2`" instead.

Let's take a look at the actual C implementation of `getdynamic`. Like most of the other YARV instructions, Ruby implements `getdynamic` in the insns.def code file:

```
/**
  @c variable
  @e Get value of block local variable (pointed to by idx
     'level' indicates the nesting depth from the current
  @j level, idx で指定されたブロックローカル変数の値をスタックに
     level はブロックのネストレベルで、何段上か
 */
DEFINE_INSN
getdynamic
(dindex_t idx, rb_num_t level)
()
(VALUE val)
{
    rb_num_t i;
    VALUE *dfp2 = GET_DFP();
    for (i = 0; i < level; i++) {
        dfp2 = GET_PREV_DFP(dfp2);
    }
    val = *(dfp2 - idx);
}
```

Here the `GET_DFP` macro returns the `DFP` from the current scope. This macro is defined in the vm_insnhelper.h file along with a number of other YARV instruction related macros. Then Ruby iterates over the `DFP` pointers, moving from the current scope to the parent scope, and then from the parent scope to the grandparent scope, by repeatedly dereferencing the DFP pointers. Ruby

uses the `GET_PREV_DFP` macro, also defined in vm_insnhelper.h, to move from one `DFP` to another. The `level` parameter indicates how many times to iterate, or how many rungs of the ladder to climb.

Finally, Ruby obtains the target variable using the `idx` parameter; this is the index of the target variable. Therefore, this line of code:

```
val = *(dfp2 - idx);
```

…gets the value from the target variable. It means:

- Start from the address of the `DFP` for the target scope, `dfp2`, obtained previously from the `GET_PREV_DFP` iterations.
- Subtract `idx` from this address. `idx` tells `getdynamic` the index of the local variable you want to load, or in other words how far down the stack the target variable is located.
- Get the value from the YARV stack at this adjusted address.

So in my example above:

```
getdynamic str, 2
```

YARV will take the `DFP` from the scope two levels up on the YARV stack, and subtract the index value `str` (this might be 2 or 3 for example) from it to obtain a pointer to the `str` variable.

Experiment 2-2: Exploring special variables

In the diagrams above I showed a value called `svar/cref` in the `LFP-1` position on the stack. What are these two values? And how can Ruby save two values in one location on the stack? Why does it do this? Let's take a look….

Most often the `LFP-1` slot in the stack will contain the `svar` value - this is a pointer to a table of any special variables that might exist in this stack frame. In Ruby the term "special variables" refers to values that Ruby automatically creates for you as a convenience based on the environment or on recent operations. For example, Ruby sets `$*` to the ARGV array and `$!` to the last exception raised.

Notice that all of the special variables begin with the dollar sign character, which usually indicates a global variable. This begs the question: are special variables global variables? If so, then why does Ruby save a pointer to them on the stack? To find out, 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'm matching the word `fox` in the string using a regex. Then I print out the matching string using the `$&` special variable. Running this I get:

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

Now I'll search 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"
/fox/.match(str)


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

```
end
search(str)

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

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

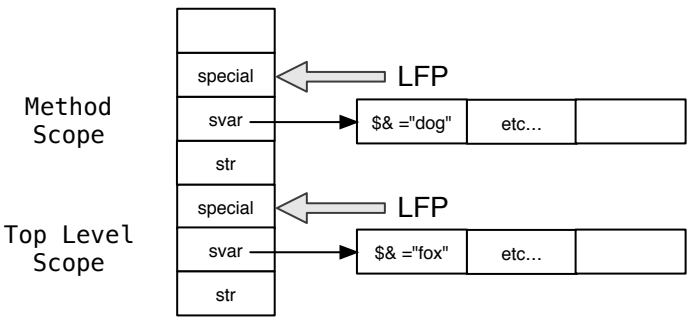
- First I search the string in the top scope for `fox`. This matches the word and saves `fox` into the `$&` special variable.
- Then I call the `search` method and search for the word `dog`. I immediately print out the match using the same `$&` variable inside the method.
- Finally I return to the top level scope and print out the value of `$&` again.

Running this test, I get:

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

This is what we expect, but think carefully about this for a moment. The `$&` variable is obviously not global since it has different values at different places in my Ruby script. Ruby preserves the value of `$&` from the top level scope during the execution of the `search` method, allowing me to print out 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:



Here you can see 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 `LFP` pointer for each stack frame. Depending on exactly which special variable you use, the table in this diagram might be a hash table or just a simple C structure. I’ll discuss hash tables in Chapter 4.

Ruby saves actual global variables - these are 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 let’s try one more test - what happens if I perform the search inside a block and not a method?

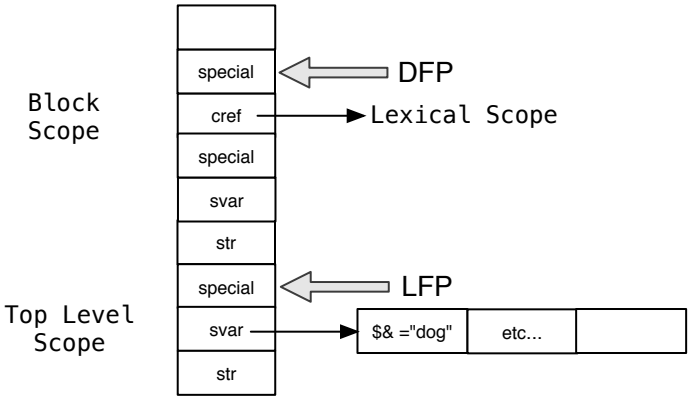
```
str = "The quick brown fox jumped over the lazy dog.\n"
/fox/.match(str)


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


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

Running this last test, I get:

```
$ 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 scope and the 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.

Here is how Ruby implements this behavior:



Now Ruby has just a single special variable table, for the top level scope. Ruby finds the special variables using the `LFP` pointer, which points only to the top level scope. Inside the block scope, since there is no need for a separate copy of the special variables, Ruby takes advantage of the `DFP-1` open slot and saves a value called the `cref` there instead.

What does the `cref` value mean? Unfortunately, I don't have space in this book to explain this carefully, but in a nutshell `cref` indicates whether the given block should be executed in a different lexical scope compared to the parent frame. Lexical scope refers to the class or module the you are currently defining methods for. Ruby uses the `cref` value to implement metaprogramming API calls such as `eval` and `instance_eval` - the `cref` value is a pointer to the location on the lexical scope stack this block should be evaluated in. I'll touch on these advanced concepts in Chapter 5, but you'll have to wait for *Ruby Under a Microscope - Part 2* to read a complete explanation of lexical scope and how Ruby implements it.

The best way to get an accurate list of all the special variables Ruby supports is to look right at the MRI C source; here's a snippet of the C code that tokenizes your Ruby program. I've taken this from the parser_yylex function located in parse.y:

```
case '$':
  lex_state = EXPR_END;
  newtok();
  c = nextc();
  switch (c) {
```

```
    case '_':                /* $_: last read line string */
      c = nextc();
      if (parser_is_identchar()) {
          tokadd('$');
          tokadd('_');
          break;
      }
      pushback(c);
      c = '_';
      /* fall through */
    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;
```

At the top of this code snippet you can see Ruby matches a dollar sign “$” character - this is part of the large C switch statement that tokenizes your Ruby code, the process I discussed back at the beginning of Chapter 1. This is followed by an inner switch statement that matches on the following character; each of these characters corresponds to a special variable.

Just a bit farther down in the function is more C code that parses other special variable tokens you write in your Ruby code - these are the “regex last match” and related special variables:

```
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;
```

Finally, this last snippet parses $1, $2, etc., producing the special variables that return the “nth back reference” from the last regular expression operation:

```
case '1': case '2': case '3':
case '4': case '5': case '6':
case '7': case '8': case '9':
  tokadd('$');
  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;
```

How YARV controls your program's execution flow

YARV uses its own internal set of control structures, similar to the structures you use in Ruby.

We've seen how YARV uses a stack while executing its instruction set and how it can access variables locally or dynamically, but what about control structures? Controlling the flow of execution is a fundamental requirement for any programming language, and Ruby has a rich set of control structures. How does YARV implement it?

Just like Ruby itself, YARV has it own control structures, albeit at a much lower level. Instead of `if` or `unless` statements, YARV uses two low level instructions called `branchif` and `branchunless`. And instead of using control structures such as "`while…end`" or "`until…end`" loops, YARV has a single low level function called `jump` that allows it to change the program counter and move from one place to another in your compiled program. By combining the `branchif` or `branchunless` instruction with the `jump` instruction YARV is able to execute most of Ruby's simple control structures.

How Ruby executes an if statement

A good way to understand how YARV controls execution flow is to take a look at how the `if/else` statement works. Here's a simple Ruby script that uses both `if` and `else`:

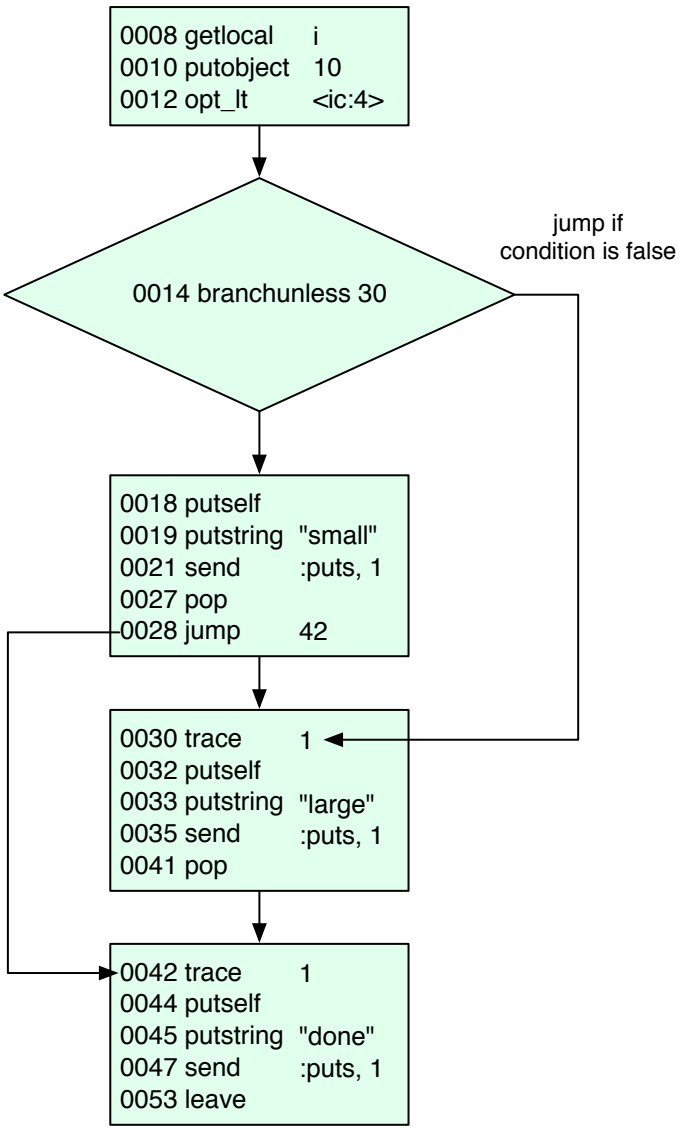
```
i = 0
if i < 10
  puts "small"
else
  puts "large"
end
puts "done"
```

```
0000 trace            1
0002 putobject        0
0004 setlocal         i
0006 trace            1
0008 getlocal         i
0010 putobject        10
0012 opt_lt           <ic:4>
0014 branchunless     30
0016 trace            1
0018 putself
0019 putstring        "small"
0021 send             :puts, 1
0027 pop
0028 jump             42
0030 trace            1
0032 putself
0033 putstring        "large"
0035 send             :puts, 1
0041 pop
0042 trace            1
0044 putself
0045 putstring        "done"
0047 send             :puts, 1
0053 leave
```

On the right you can see the corresponding snippet of compiled YARV instructions. Reading the YARV instructions, you can see Ruby follows a pattern for implementing the if/else statement:

- evaluate condition
- jump to false code if condition is false
- true code; jump to end
- false code

This is a bit easier to follow if I paste the instructions into a flowchart:



You can see how the `branchunless` instruction in the center is the key to how Ruby implements `if` statements; here's how it works:

- First at the top Ruby evaluates the condition of my `if` statement, "`i < 10`," using the `opt_lt` (optimized less-than) instruction. This will leave either a true or false value on the stack.

- Then `branchunless` will jump down to the false/else condition if the condition is false. That is, it "branches unless" the condition is true. Ruby uses `branchunless` and not `branchif` for `if/else` conditions since the positive case, the code that immediately follows the if statement, is compiled to appear right after the condition code. Therefore YARV needs to jump if the condition is false.
- Or if the condition is true Ruby will not branch and will just continue to execute the positive case code. After finishing the positive code Ruby will then jump down to the instructions following the `if/else` statement using the `jump` instruction.
- Finally either way Ruby will continue to execute the subsequent code.

YARV implements the `unless` statement in a similar way using the same `branchunless` instruction, except the positive and negative code snippets are in reverse order. For looping control structures like "`while`…`end`" and "`until`…`end`" YARV uses the `branchif` instruction instead. But the idea is the same: calculate the loop condition, then execute `branchif` to jump as necessary, and finally use `jump` statements to implement the loop.

Jumping from one scope to another

One of the challenges YARV has implementing some control structures is that, similar to dynamic variable access, Ruby sometimes can jump from one scope to another. The simplest example of this is the `break` statement. `break` can be used both to exit a simple loop like this:

```
i = 0
while i<10
  puts i
  i += 1
  break
end
```

…or from a block iteration like this:

```
10.times do |n|
  puts n
  break
end
puts "continue from here"
```

In the first case, YARV can exit the while loop using simple `jump` instructions like we saw above in the `if/else` example. However, exiting a block is not so simple: in this case YARV needs to jump to the parent scope and continue execution after the call to `10.times`. How does it do this? How does it know where to jump to? And how does it adjust both its internal stack and your Ruby call stack to be able to continue execution properly in the parent scope?

To implement jumping from one place to another in the Ruby call stack – that is, outside of the current scope – Ruby uses the `throw` YARV instruction. YARV's `throw` instruction resembles the Ruby `throw` keyword: it sends or throws the execution path back up to a higher scope. It also resembles the `throw` keyword from C++ or Java – it's similar to raising an exception, except there is no exception object here.

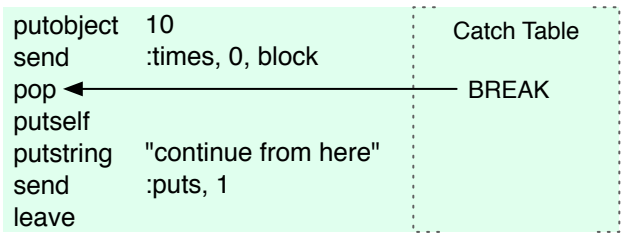
Let's take a look at how that works; here's the compiled code for the block above containing the `break` statement:

```
10.times do |n|
  puts n
  break
end
puts "continue from here"
```

```
putself
getdynamic  n, 0
send        :puts, 1
pop
putnil
throw       2
leave
```

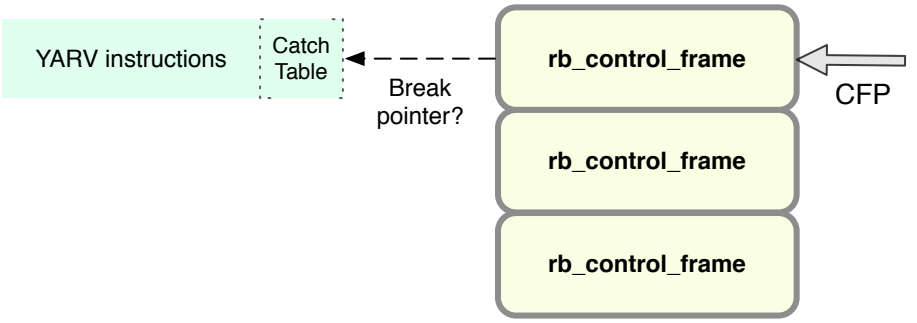
```
putobject   10
send        :times, 0, block
pop
putself
putstring   "continue from here"
send        :puts, 1
leave
```

You can see a "`throw 2`" instruction appears in the compiled code for the block. `throw` implements throwing an exception at the YARV instruction level by using something called a "catch table" A catch table is a table of pointers optionally attached to any YARV code snippet. Conceptually, a catch table might look like this:

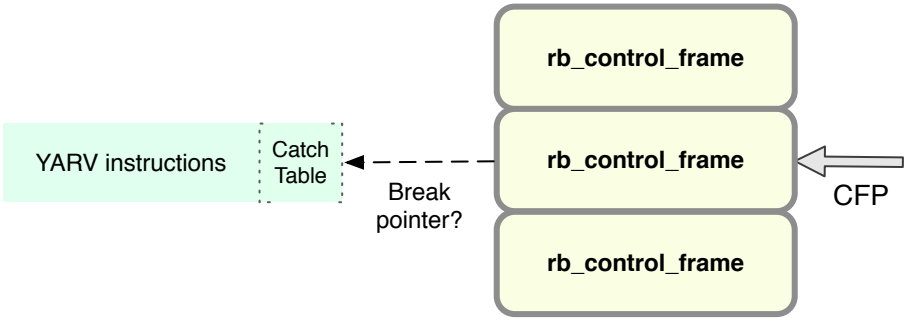


Here, the catch table from my example contains just a single pointer to the `pop` statement, which is where execution would continue after an exception. Whenever you use a break statement in a block, Ruby not only compiles the `throw` instruction into the block's code, but it also adds the `BREAK` entry into the catch table of the parent scope. For a `break` within a series of nested blocks, Ruby would add the `BREAK` entry to a catch table even farther down the `rb_control_frame` stack.

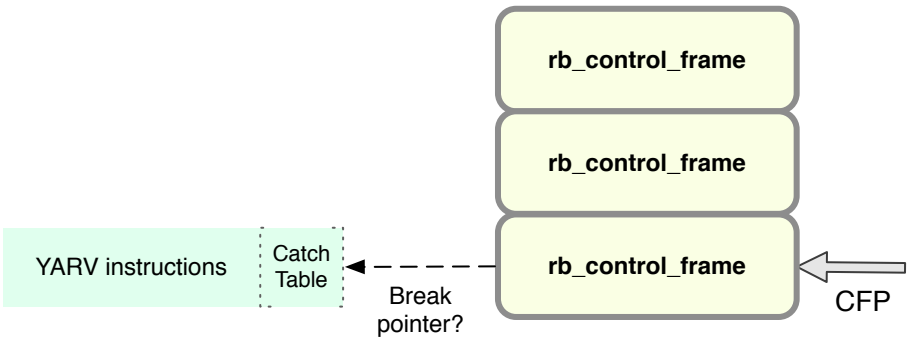
Later, when YARV executes the `throw` instruction it checks to see whether there's a catch table containing a `BREAK` pointer for the current YARV instruction sequence:



If there isn't, Ruby will start to iterate down through the stack of `rb_control_frame` structures looking for a catch table containing a break pointer…



…and continue to iterate until it finds one:



In my simple example, there is only one level of block nesting, so Ruby will find the catch table and `BREAK` pointer after just one iteration:

```
10.times do |n|
   puts n
   break
end
puts "continue from here"
```

```
putself
getdynamic  n, 0
send        :puts, 1
pop
putnil
throw       2
leave
```

```
putobject   10
send        :times, 0, block
pop
putself
putstring   "continue from here"
send        :puts, 1
leave
```

Catch Table

BREAK

Once Ruby finds the catch table pointer, it resets both the Ruby call stack (the `CFP` pointer) and the internal YARV stack to reflect the new program execution point. Then YARV continues to execute your code from there. That is, YARV resets the internal `PC` and `SP` pointers as needed.

What is interesting to me about this is how Ruby uses a process similar to raising and rescuing an exception internally to implement a very commonly used control structure:

the `break` keyword. In other words, what in more verbose languages is an exceptional occurrence becomes in Ruby a common, everyday action. Ruby has wrapped up a confusing, unusual syntax – raising/rescuing of exceptions – into a simple keyword, `break`, and made it very easy to understand and use. Of course, Ruby needs to use exceptions because of the way blocks work: they are on one hand like separate functions or subroutines, but on the other hand just part of the surrounding code. For this reason Ruby needs a keyword like `break` that seems simple at first glance but internally is quite complex.

Another commonplace, ordinary Ruby control structure that also uses catch tables is the `return` keyword. Whenever you call `return` from inside a block, Ruby internally raises an exception and rescues it with a catch table pointer like this. In fact, `break` and `return` are implemented with exactly the same YARV instructions; the only difference is that for `return` Ruby passes a 1 to the `throw` instruction (e.g. `throw 1`), while for `break` it passes a 2 (`throw 2`) as we saw above. The `return` and `break` keywords are really two sides of the same coin.

Finally, besides `BREAK` there are other types of pointers that Ruby can use in the catch table. The others are used to implement different control structures: `RESCUE`, `ENSURE`, `RETRY`, `REDO` and `NEXT`. For example, when you explicitly raise an exception in your Ruby code using the `raise` keyword, Ruby implements the `rescue` block in a similar way by using the catch table, but this time with a `RESCUE` pointer. The catch type is simply a list of event types that can be caught and handled by that sequence of YARV instructions, similar to how you would use a rescue block in your Ruby code.

Experiment 2-3: Testing how Ruby implements for loops internally

I always knew that Ruby's for loop control structure worked essentially the same way as using a block with the `each` `Enumerable` method. That is to say, I knew this code:

```
for i in 0..5
   puts i
end
```

… worked the same way as this code:

```
(0..5).each do |i|
   puts i
end
```

But I never suspected that internally Ruby actually implements for loops using `each`! That is, there really is no for loop control structure in Ruby at all; instead, the `for` keyword is really just syntactical sugar for calling `each` with a range.

To prove this is the case, all you have to do is inspect the YARV instructions that are produced by Ruby when you compile a for loop. Let's use the same `RubyVM::InstructionSequence.compile` method that I did in Chapter 1:

```
code = <<END
for i in 0..5
   puts i
end
END
puts RubyVM::InstructionSequence.compile(code).disasm
```

Running this I get:

```
== disasm: <RubyVM::InstructionSequence:<compiled>@<compiled>>==========
== catch table
| catch type: break  st: 0002 ed: 0010 sp: 0000 cont: 0010
```

```
|-------------------------------------------------------------------------
local table (size: 2, argc: 0 [opts: 0, rest: -1, post: 0, block: -1] s1)
[ 2] i
0000 trace            1                                               (   1)
0002 putobject        0..5
0004 send             :each, 0, block in <compiled>, 0, <ic:0>
0010 leave
== disasm: <RubyVM::InstructionSequence:block in <compiled>@<compiled>>=
== catch table
| catch type: redo   st: 0005 ed: 0016 sp: 0000 cont: 0005
| catch type: next   st: 0005 ed: 0016 sp: 0000 cont: 0016
|-------------------------------------------------------------------------
local table (size: 2, argc: 1 [opts: 0, rest: -1, post: 0, block: -1] s3)
[ 2] ?<Arg>
0000 getdynamic       *, 0                                            (   3)
0003 setlocal         i                                               (   1)
0005 trace            1                                               (   2)
0007 putself
0008 getlocal         i
0010 send             :puts, 1, nil, 8, <ic:0>
0016 leave
```

To make this a bit easier to follow, I'll repeat these YARV instructions in a diagram, and remove some of the technical details like the `trace` statements:

```
for i in 0..5
  puts i
end
```

```
putobject    0..5
send         :each, 0, block
leave
```

```
getdynamic   *, 0
setlocal     i
putself
getlocal     i
send         :puts, 1
leave
```

You should notice right away there are two separate YARV code blocks: the outer scope which calls `each` on the range `0..5` and then an inner block that makes the `puts i` call. The "`getdynamic *, 0`" instruction in the inner block loads the implied block parameter value - `i` in my Ruby code - and the following `setlocal` instruction saves it into a local variable also called `i`.

Taking a step back and thinking about this, what Ruby has done here is:

- Automatically converted the "`for i in 0..5`" code into "`(0..5).each do`"
- Automatically created a block parameter to hold each value in the range, and:
- Automatically created a local variable in the block with the same name as the for loop variable, and saved the block parameter's value into it.

How JRuby executes your code

As I explained in Chapter 1, JRuby tokenizes and parses your Ruby code in almost the same way that MRI Ruby does. And, like Ruby 1.9 and Ruby 2.0, JRuby continues to compile your Ruby code into byte code instructions before actually running your program using a virtual machine.

However, this is where the similarity ends: MRI and JRuby use two very different virtual machines to execute your code. As I showed earlier in Chapter 2, MRI Ruby 1.9 and higher use YARV, which was custom designed to run Ruby programs. JRuby, however, uses the Java Virtual Machine to execute your Ruby program. Despite it's name, many different programming languages run on the JVM. In fact, this really is JRuby's *raison d'être* - the whole point of building a Ruby interpreter with Java is to be able to execute Ruby programs using the JVM. There are two important reasons to do this:

- Environmental: Using the JVM opens new doors for Ruby and allows you to use Ruby on servers, in applications and in IT organizations where previously you could not run Ruby at all.
- Technical: The JVM is the product of almost 20 years of intense research and development. It contains sophisticated solutions for many difficult computer science problems such as garbage collection, multithreading, and much more. By running on the JVM, Ruby runs faster and more reliably!

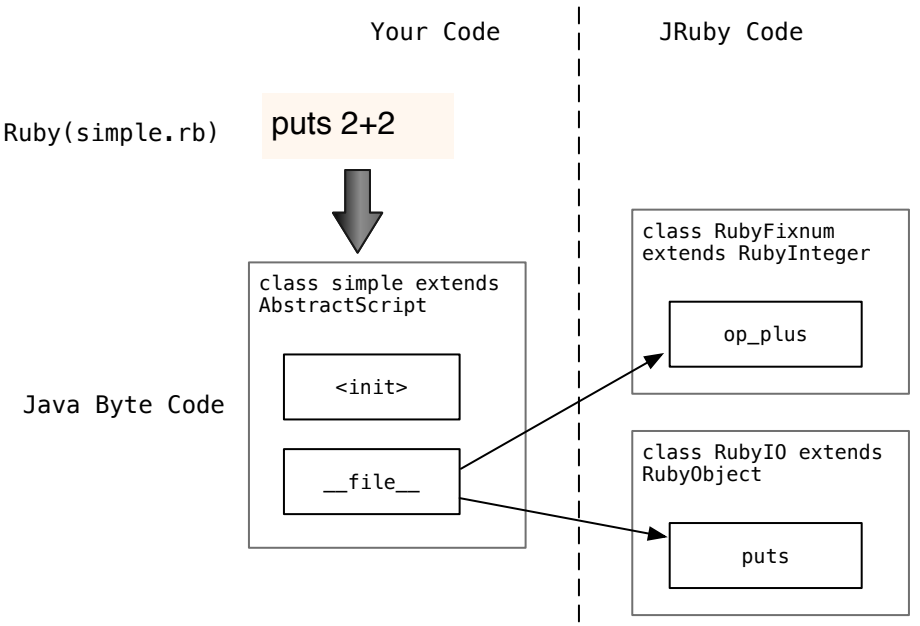
To get a better sense of how this works, let's take a look at how JRuby would execute the same one line Ruby script I used as an example earlier:

```
puts 2+2
```

The first thing JRuby does is tokenize and parse this Ruby code into an AST node structure. Once this is finished, JRuby will iterate through the AST nodes and convert your Ruby into Java byte code. Using the `bytecode` command line option you can actually see this byte code for yourself:

```
$ cat simple.rb
puts 2+2
$ jruby --bytecode simple.rb
```

The output is complex and confusing and I don't have the space to explain it here, but here's a diagram summarizing how JRuby compiles and executes this one line program:



Here's how this works:

- On the top left I show the "`puts 2+2`" Ruby source code from simple.rb.
- The downward arrow indicates that JRuby translates this into a Java class, named "simple" after my Ruby file name, and derived from the `AbstractScript` base class.
- The JVM later calls the second method in this class, `__file__`, in order to execute my compiled Ruby script. The `__file__` method contains the compiled version of the top level Ruby code in simple.rb - in this example the entire program.
- The `__file__` method, in turn, calls the `op_plus` method in the `RubyFixnum` Java class.
- Once JRuby's `RubyFixnum` Java class has added 2+2 for me and returned 4, `__file__` will call the `puts` method in the `RubyIO` Java class to display the result.

There are a couple of important ideas to notice in all of this: First, as I said above, your Ruby code is compiled into Java byte code. It's both alarming and amazing at the same

time to imagine one of my Ruby programs converted into Java! However, remember we're talking about Java byte code here, not an actual Java program. Java byte code instructions are very low level in nature and can be used to represent code originally written in any language, not just Java.

Second, JRuby implements all of the built in Ruby classes such as `Fixnum` and `IO` using Java classes; these classes are named `RubyFixnum`, `RubyIO`, etc. Of course, JRuby also implements all of the Ruby language's intrinsic behavior as a series of other Java classes, including: objects, modules, blocks, lambdas, etc. I'll touch on a few of these implementations in the following chapters.

Internally, the JVM uses a stack to save arguments, return values and local variables just like YARV does. However, explaining how the JVM works is beyond the scope of this book.

To get a feel for what the JRuby source code looks like, let's take a quick look at the `op_plus` method in the `org.jruby.RubyFixnum` Java class:

```
public IRubyObject op_plus(ThreadContext context,
                           IRubyObject other) {
  if (other instanceof RubyFixnum) {
    return addFixnum(context, (RubyFixnum)other);
  }
  return addOther(context, other);
}
```

First of all, remember this is a method of the `RubyFixnum` Java class, which represents the Ruby Fixnum class, the receiver of the `op_plus` operation. Thinking about this for a moment, this means that each instance of a Ruby object, such as the `Fixnum` receiver "2" in my example, is represented by an instance of a Java class. This is one of the key concepts behind how JRuby's implementation works: for every Ruby object instance there is an underlying Java object instance. I'll have more about this in Chapter 3.

Next, note the arguments to `op_plus` are something called a `ThreadContext` and the operand of the addition operation, a Java object called `other` which implements the `IRubyObject` interface. Reading the code above, we can see that if the `other` operand is also an instance of `RubyFixnum` then JRuby will call the `addFixnum` method; here is that code:

```
private IRubyObject addFixnum(ThreadContext context,
                              RubyFixnum other) {
  long otherValue = other.value;
  long result = value + otherValue;
  if (additionOverflowed(value, otherValue, result)) {
    return addAsBignum(context, other);
  }
  return newFixnum(context.getRuntime(), result);
}
```

Here you can see the Java code calculates the actual result of the “2+2” operation: “`result = value + otherValue`.” If the result were too large to fit into a `Fixnum` object, JRuby would call the `addAsBignum` method instead. Finally JRuby creates a new `Fixnum` instance, sets its value to `result` or 4 and returns it.

How Rubinius executes your code

In Chapter 1 I explained how Rubinius uses a combination of C++ and Ruby to tokenize, parse and compile your Ruby code. The same is true when it comes time to actually execute your code: Rubinius combines a virtual machine (the "Rubinius VM") implemented in C++ with a library of the basic core Ruby classes written in Ruby itself. Called the "kernel," this Ruby library allows you to see how all of the core Ruby classes actually work… without having to understand C or Java! Where is it not possible to implement Ruby with Ruby, Rubinius's implementation uses C++ code in the virtual machine instead.

Similar to MRI and JRuby, Rubinius first compiles your Ruby code into a series of VM instructions. Along with implementing the portions of the Ruby basic object library that couldn't be built with Ruby, the Rubinius VM also interprets and executes these instructions. In addition, the C++ Rubinius VM also implements a garbage collector and contains support for threads, for interacting with the operating system and many other things.

I don't have space here in this book to explain Rubinius internals in complete detail, but let's see how Rubinius executes my one line sample Ruby program:

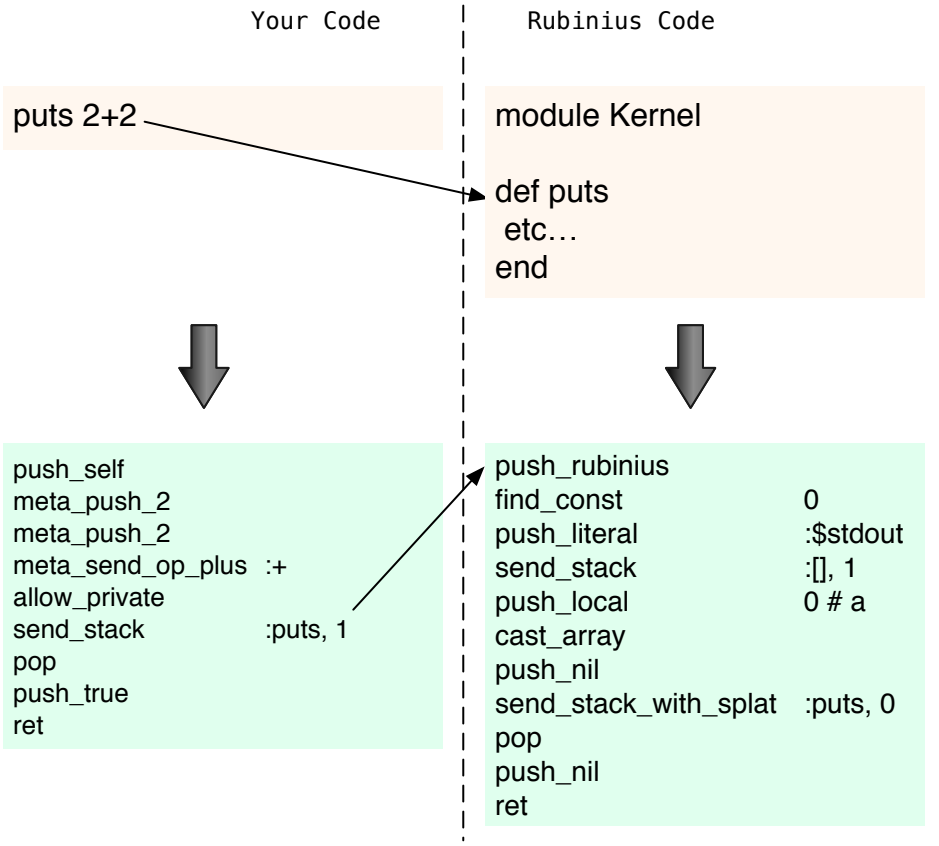
```
puts 2+2
```

Just like JRuby, Rubinius has a command line option that allows you to see the VM instructions your code is compiled into:

```
$ cat simple.rb
puts 2+2
$ rbx compile simple.rb -B


============= :__script__ ==============
Arguments:   0 required, 0 post, 0 total
Arity:       0
Locals:      0
Stack size:  3
Lines to IP: 1: 0..11
0000:  push_self
0001:  meta_push_2
0002:  meta_push_2
0003:  meta_send_op_plus          :+
```

```
0005:  allow_private
0006:  send_stack                  :puts, 1
0009:  pop
0010:  push_true
0011:  ret
---------------------------------------
```

Unlike JVM byte code, the Rubinius VM instructions are very high level and easy to understand; in fact, they very closely resemble the YARV instructions we saw earlier in this chapter. For example, `push_self` will push the `self` pointer on the top of the stack and `send_stack` will call the specified method with the given number of arguments.



This diagram shows what happens when I run my simple Ruby program:

- First, on the left Rubinius compiles my "`puts 2+2`" code into Rubinius VM instructions.
- On the right, Rubinius compiles it's own Ruby code, in this case the `Kernel` module `puts` method, into VM instructions in the same way. This compilation actually happens ahead of time during the Rubinius build process.
- Later the Rubinius VM starts to interpret and execute these instructions. Depending on how long my process continues to run, a JIT compiler might further compile these instructions into LLVM byte code and ultimately into machine language.
- When my code makes the call to `puts` to print out the result of 4, the Rubinius VM `send_stack` instruction finds and calls the `Kernel.puts` method.

Let's see what happens next. Since Rubinius's implementation of `puts` is written in Ruby we can just take a look at it! Here is a snippet of that code, taken from kernel/common/kernel.rb:

```
module Kernel

  ...

  def puts(*a)
    $stdout.puts(*a)
    nil
  end
```

As you can see, here Rubinius simply calls into the `puts` method of the underlying global `IO` object that represents the `stdout` stream. This is also written in Ruby, this time inside the kernel/common/io19.rb file:

```
class IO

  ...

  def puts(*args)

  ...
```

```
  write str
  write DEFAULT_RECORD_SEPARATOR unless str.suffix?(DEFAULT_RECORD_SEPARATOR)
```

I've removed some of the code in the `IO.puts` method to keep things simple, but you can see `IO.puts` calls a `write` method to actually write out the string. It turns out Rubinius implements this using C++ code inside the Rubinius VM itself.

Taking a step back, let's review the overall process Rubinius uses to execute your Ruby code:

- First, it compiles your code into VM instructions.
- Then it executes these instructions using the Rubinius VM.
- Since Rubinius implements all of the Ruby core classes using Ruby itself, any calls you make to `String`, `Array`, `Fixnum`, etc., are all simple Ruby calls into the Ruby portion of the Rubinius kernel.
- Finally, portions of the Rubinius core library that can't be implemented in Ruby – either for performance reasons or because the code needs to interact with the operating system at a low level – are written in C++ directly inside the Rubinius VM.

Let's continue down the call stack and see how exactly Rubinius calls into the `write` method in the C++ code. Here's a snippet from the vm/builtin/io.hpp C++ source file:

```
namespace rubinius {

  class IO : public Object {

    ...

    // Rubinius.primitive :io_write
    Object* write(STATE, String* buf, CallFrame* calling_environment);
```

```
    ...
  }
}
```

You can see the `write` method is a member of the `IO` C++ class. The `IO` C++ class corresponds to the `IO` Ruby class – each core Ruby class in Rubinius's Ruby kernel has as corresponding C++ class that handles things the Ruby class cannot. An important detail here is the comment: "`Rubinius.primitive :io_write`" – this is actually the glue that holds Ruby and C++ together inside of Rubinius. This is not just a comment, but is also a directive that tells the Rubinius VM to call the `IO::write` C++ method when the Ruby IO code calls the `IO.write` Ruby method.

I won't show the actual implementation of `IO::write`, but if you're interested you can find it in vm/builtin/io.cpp. As you might guess, it takes the string data and passes it into an operating system call.

Chapter 3
Objects, Classes and Modules

I always think about object oriented programming in the supermarket.

We all learn very early on that Ruby is an object oriented language, descended from languages like Smalltalk and Simula. Everything is an object and all Ruby programs consist of a set of objects and the messages that are sent back and forth among them. Typically, we learn about object oriented programming by looking at how to use objects and what they can do: how they can group together data values and behavior related to those values, how each class should have a single responsibility or purpose or how different classes can be related to each other through encapsulation or inheritance.

But what are Ruby objects, exactly? What information does an object contain? If I were to look at a Ruby object through a microscope, what would I see? Are there any moving parts inside? And what about Ruby classes? All of us know how to create and use Ruby classes, but what exactly is a class? Finally, what are modules in Ruby? How are modules and classes related? What happens when I include a module into a class? How does Ruby determine which class or module implements a given method?

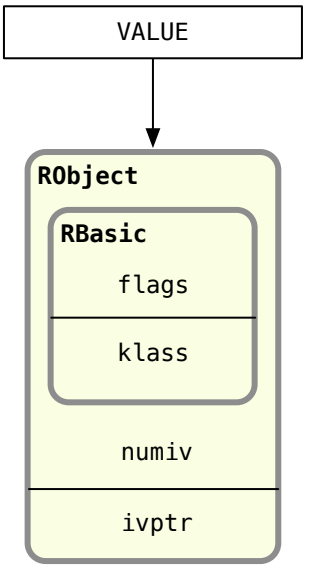
In this chapter I am going to answer these questions by exploring how Ruby works internally. Looking at exactly how Ruby implements objects, classes and modules can give you some insight into how they were intended to be used, and into how to write object oriented programs using Ruby.

Chapter 3 Roadmap

What's inside a Ruby object?

If I could slice open a Ruby object, what would I see?

Ruby saves all of your custom objects inside a C structure called `RObject`, which looks like this in Ruby 1.9 and 2.0:



On the top is a pointer to the `RObject` structure. Internally Ruby always refers to any value using these `VALUE` pointers. Below you can see the `RObject` value is divided into two halves: `RBasic` and `RObject`. The `RBasic` section contains information that all values use, not only objects: a set of boolean values called `flags` which store a variety of internal, technical values and also a class pointer, called `klass`. The class pointer indicates which class this object is an instance of. At the bottom in the `RObject` specific portion, Ruby saves an array of instance variables that this object instance contains, using two values: `numiv`, the instance variable count, and `ivptr`, a pointer to an array of values.

Summarizing the contents of the `RObject` structure, we can write a very technical definition of what a Ruby object is:

> *Every Ruby object is the combination of a class pointer and an array of instance variables.*

At first glance, this definition doesn't seem that useful at all. It doesn't help me understand the meaning or purpose behind objects, or how to use them in a Ruby program. Why does Ruby implement objects in this way? The answer is simple: Ruby saves this information in `RObject` to support the basic features of the language.

For example, suppose I have a simple Ruby class:

```
class Mathematician
  attr_accessor :first_name
  attr_accessor :last_name
end
```

Ruby needs to save the class pointer in `RObject` because every object has to keep track of the class you used to create it:

```
> euler = Mathematician.new
 => #<Mathematician:0x007fbd738608c0>
```

In the above example by displaying the class name, "`#<Mathematician...`," Ruby is displaying the value of the class pointer for the "euler" object when I inspect it. The hex string that follows is actually the `VALUE` pointer for the object. This will be different for every instance of Mathematician.

Ruby also has to keep track of any values you save in it – Ruby uses the instance variable array to do this:

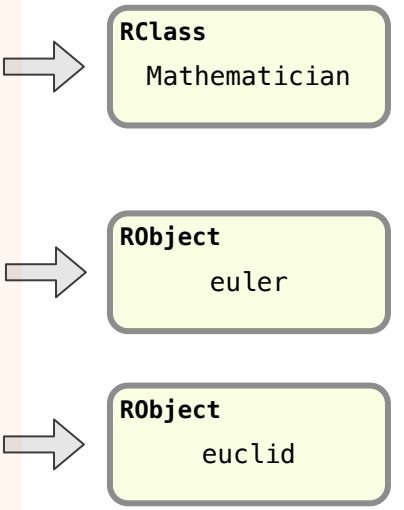
```
> euler.first_name = 'Leonhard'
 => "Leonhard"
> euler.last_name  = 'Euler'
 => "Euler"
> euler
 => #<Mathematician:0x007fbd738608c0 @first_name="Leonhard", @last_name="Euler">
```

As you can see here, Ruby also displays the instance variable array for `euler` when I inspect it again. Ruby needs to save this array of values in each object since every object instance can have different values for the same instance variables. For example:

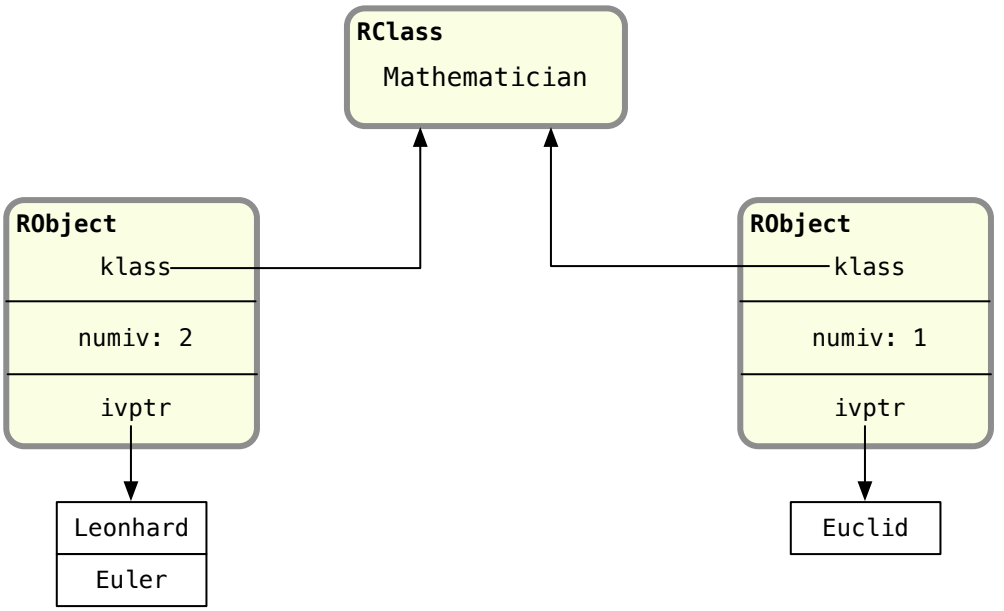
```
> euclid = Mathematician.new
> euclid.first_name = 'Euclid'
> euclid
 => #<Mathematician:0x007fabdb850690 @first_name="Euclid">
```

Now let's take a look at Ruby's C structures in a bit more detail – when you run this simple script, Ruby will create one `RClass` structure and two `RObject` structures:

```
class Mathematician
  attr_accessor :first_name
  attr_accessor :last_name
end


euler = Mathematician.new
euler.first_name = 'Leonhard'
euler.last_name = 'Euler'


euclid = Mathematician.new
euclid.first_name = 'Euclid'
```



I will cover how Ruby implements classes with the `RClass` structure in the next section, but here is an example of how Ruby saves the mathematician information in the two `RObject` structures in more detail:

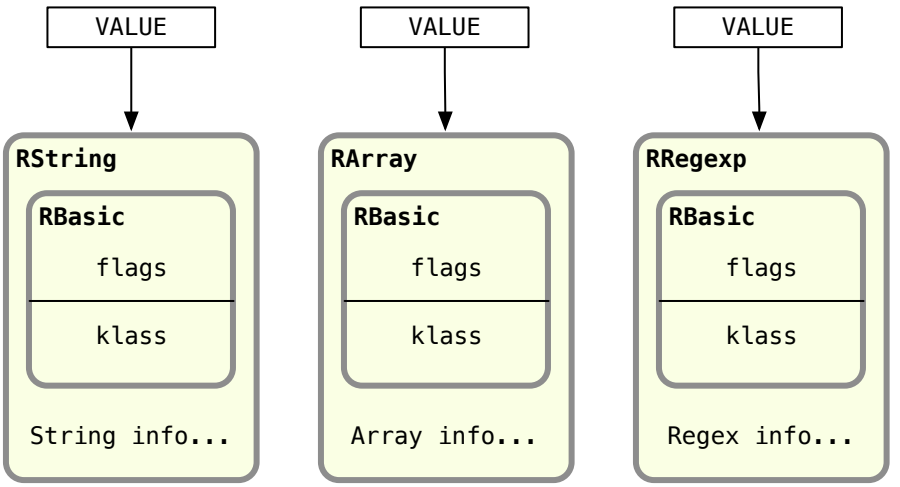


You can see each of the `klass` values points to the `Mathematician` `RClass` structure, and each `RObject` structure has a separate array of instance variables. Both arrays contain `VALUE` pointers, the same pointer that Ruby uses to refer to the `RObject` structure. As you can see from the example above, one of the objects contains two instance variables, while the other contains only one.

Generic objects

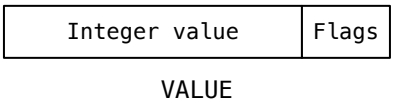
This is how Ruby saves custom classes, like my `Mathematician` class, in `RObject` structures. But we all know that every Ruby value, including basic data types, such as integers, strings or symbols, are also objects. The Ruby source code internally refers to these built in types as “generic” types. How does Ruby store these generic objects? Do they also use the `RObject` structure? The answer is no. Internally Ruby uses a different C structure to save values for each of its generic data types, and not `RObject`. For example, Ruby saves string values in `RString` structures, arrays in `RArray` structures and regular expressions in `RRegexp` structures, etc. Ruby only uses `RObject` to save instances of custom object classes that you create, and for a few custom object classes Ruby creates internally as well.

However, all of these different structures share the same `RBasic` information that we saw in `RObject`:

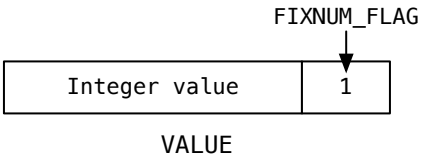


Since the `RBasic` structure contains the class pointer, each of these generic data types is also an object. They are all instances of some Ruby class, indicated by the class pointer saved inside of `RBasic`.

As a performance optimization, Ruby saves small integers, symbols and a few other simple values without any structure at all. Ruby saves these values right inside the `VALUE` pointer:



That is, these `VALUE`s are not pointers at all; instead they are the values themselves. For these simple data types, there is no class pointer. Instead, Ruby remembers the class using a series of bit flags saved in the first few bits of the `VALUE`. For example, all integers have the `FIXNUM_FLAG` bit set, like this:



Whenever the `FIXNUM_FLAG` is set, Ruby knows this `VALUE` is really a small integer, an instance of the `Fixnum` class, and not a pointer to a value structure. There is also a

similar bit flag to indicate if the `VALUE` is a symbol, and values such as `nil`, `true` and `false` also have special values.

It's easy to see that integers, strings and other generic values are all objects using IRB:

```
$ irb
> "string".class
 => String
> 1.class
 => Fixnum
> :symbol.class
 => Symbol
```

Here we can see Ruby saves a class pointer or the equivalent bit flag for all of these values by calling the `class` method on each of them. The `class` method returns the class pointer, or at least the name of the class the `klass` pointer refers to.

Do generic objects have instance variables?

Now let's reread our definition of a Ruby object from above:

> *Every Ruby object is the combination of a class pointer and an array of instance variables.*

What about instance variables for generic objects? According to our definition, all Ruby objects are a class pointer combined with an array of instance variables. Do integers, strings and other generic data values have instance variables? That would seem a bit odd. But if integers and strings are objects, then this must be true! And if this is true, where does Ruby save these values, if it doesn't use the `RObject` structure?

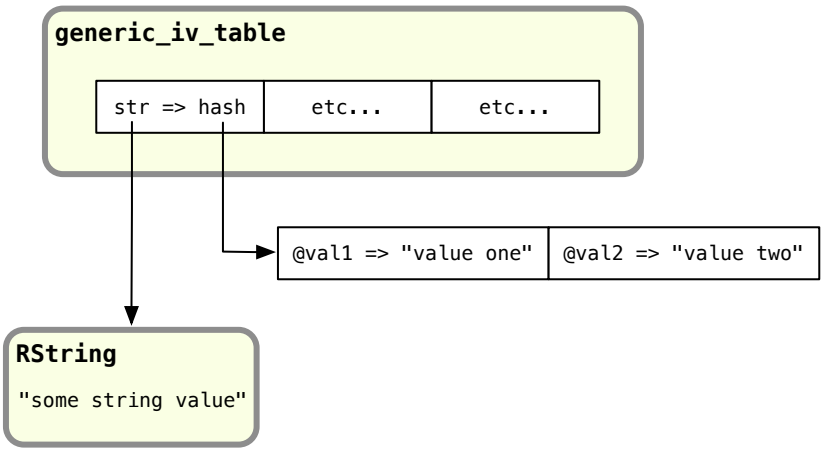
Using the `instance_variables` method you can see that each of these basic values can also contain an array of instance variables, as strange as that might seem at first:

```
$ irb
> str = "some string value"
 => "some string value"
> str.instance_variables
 => []
> str.instance_variable_set("@val1", "value one")
 => "value one"
> str.instance_variables
 => [:@val1]
> str.instance_variable_set("@val2", "value two")
```

```
 => "value two"
> str.instance_variables
 => [:@val1, :@val2]
```

You can repeat the same exercise using symbols, arrays, or any Ruby value you select whatsoever. Every Ruby value is an object, and every object contains a class pointer and an array of instance variables.

Internally, Ruby uses a bit of a hack to save instance variables for generic objects; that is, for objects that don't use an `RObject` structure. When you save an instance variable in a generic object, Ruby saves it in a special hash called the `generic_iv_table`. This hash maintains a map between generic objects and pointers to other hashes that contain each object's instance variables. For my `str` string example above, this would look like this:



For more information about hashes and how Ruby implements them, please refer to Chapter 4.

Here are the actual definitions of the `RBasic` and `RObject` C structures; you can find this code in the include/ruby/ruby.h header file:

```
struct RBasic {
    VALUE flags;
    VALUE klass;
};


#define ROBJECT_EMBED_LEN_MAX 3
struct RObject {
    struct RBasic basic;
    union {
        struct {
            long numiv;
            VALUE *ivptr;
            struct st_table *iv_index_tbl;
        } heap;
        VALUE ary[ROBJECT_EMBED_LEN_MAX];
    } as;
};
```

First at the top you'll see the definition of `RBasic`. This contains the two values I described earlier: `flags` and `klass`. Below, you'll see the `RObject` definition. Notice that it contains a copy of the `RBasic` structure as I described above. Following this is a `union` keyword, which contains a structure called `heap` followed by an array called `ary`. I'll have more on this in a moment.

The `heap` structure contains the values I discussed earlier:

- First is the value `numiv` which tracks the number of instance variables contained in this object.
- After that is `ivptr` – a pointer to an array containing the values of the instance variables this object contains. Note the names or id's of the instance variables are not stored here, only the values. Instead, Ruby uses the next value, `iv_index_tbl`, to keep track of which instance variable is which.

- `iv_index_tbl` points to a hash table that maps between the id or name of each instance variable and its location in the `ivptr` array. This value is actually stored once in the `RClass` structure for this object's class, and this pointer is simply a cache or shortcut Ruby uses to obtain that hash table quickly. The `st_table` type refers to Ruby's implementation of hash tables, which are covered in Chapter 4.

Finally, the last member of the `RObject` structure, called `ary`, occupies the same memory space as all of the previous values because of the `union` keyword at the top. Using this `ary` value, Ruby can save all of the instance variables right inside the `RObject` structure if there's enough room for them to fit. This avoids the need to call `malloc` to allocate extra memory to hold the instance variable value array. Ruby also uses this sort of optimization for the `RString` and `RArray` structures.

Experiment 3-1: How long does it take to save a new instance variable?

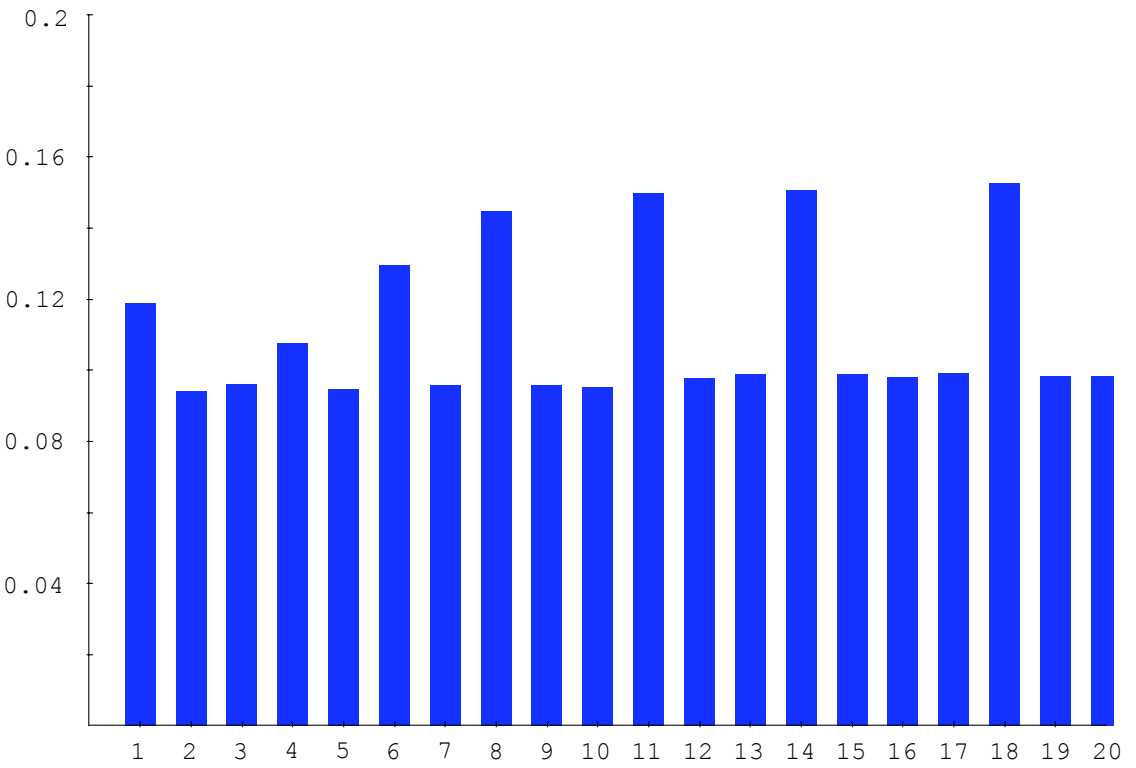
To learn more about how Ruby saves instance variables internally let's measure how long it takes Ruby to save one in an object. To do this, I'll create a large number of test objects:

```
obj = []
ITERATIONS.times do |n|
  obj[n] = Class.new.new
end
```

Here I'm using `Class.new` to create a unique class for each new object so they are all independent. Then I'll add instance variables to each of them:

```
20.times do |count|
  bench.report("adding instance variable number #{count+1}") do
    ITERATIONS.times do |n|
      obj[n].instance_variable_set("@var#{count}", "value")
    end
  end
end
```

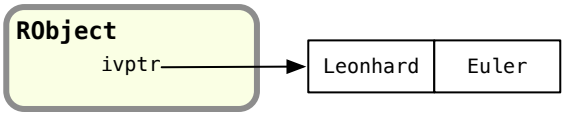
This code will iterate 20 times, repeatedly saving one more new instance variable to each of the objects I created above. Here's a graph showing the time it takes Ruby 1.9.3 to add each variable – on the left is the time it takes to save the first instance variable in all the objects, and moving to the right the additional time taken to save one more instance variable in each object:



Time to add one more instance variable (seconds x100,000) vs. instance variable count

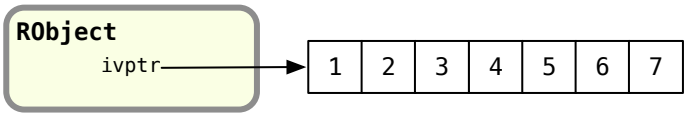
Looking at this bar chart, you can see a strange pattern. Sometimes it takes Ruby longer to add a new instance variable, and sometimes Ruby is able to save one faster. What's going on here?

The reason for this behavior has to do with that array I showed above where Ruby stores the instance variables:

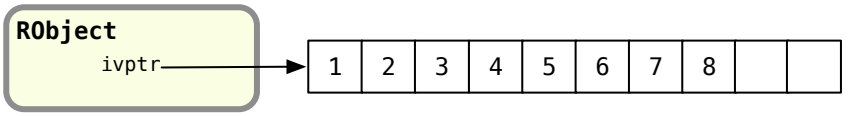


In Ruby 1.8 this array is actually a hash table containing both the variable names (the hash keys) and the actual values, which will automatically expand to accommodate any number of elements. Stay tuned for Chapter 4 to learn more about hash tables.

However, Ruby 1.9 and 2.0 speed things up a bit by saving the values in a simple array – the instance variable names are saved in the object's class instead, since the names are the same for all instances of a class. What this means, however, is that Ruby 1.9 and 2.0 need to either preallocate a large array to handle any number of instance variables, or repeatedly increase the size of this array as you save more variables. From the graph, you can see Ruby 1.9 and 2.0 repeatedly increase the array size. For example, suppose I have seven instance variables in a given object:



When I add the eighth variable, bar number 8 in the graph, Ruby 1.9 and 2.0 increase the array size by three, anticipating that you will soon add more variables:



Allocating more memory takes some extra time, which is why bar number 8 is higher. Now if I add a ninth and tenth instance variable Ruby 1.9 and 2.0 won't need to reallocate memory for this array, the space will already be available. This explains the shorter times for bars numbered 9 and 10.

Deducing what's inside the Class structure

Two objects, one class

We saw above that every object remembers its class by saving a pointer to an `RClass` structure. What information does each `RClass` structure contain? What would I see if I could look inside a Ruby class? Let's build up a model of what information must be present in `RClass`, and therefore, a technical definition of what a Ruby class is, based on what we know classes can do.

Every Ruby developer knows how to write a class: you type the `class` keyword, specify a name for the new class, and then type in the class's methods. In fact, I already wrote a Ruby class this way in the previous section:

```
class Mathematician
  attr_accessor :first_name
  attr_accessor :last_name
end
```

As you probably know, `attr_accessor` is just shorthand for defining get and set methods for an attribute.[3] Here's the more verbose way of defining the same Mathematician class:

```
class Mathematician
  def first_name
    @first_name
  end
  def first_name=(value)
    @first_name = value
  end
  def last_name
```

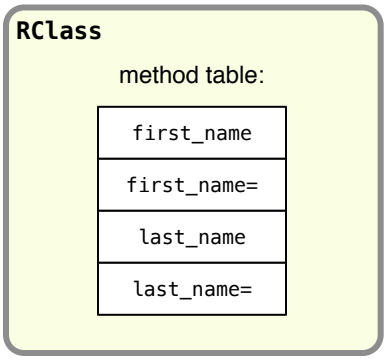
3. The methods defined by `attr_accessor` also check for `nil` values. I don't show this here.

```
    @last_name
  end
  def last_name=(value)
    @last_name = value
  end
end
```

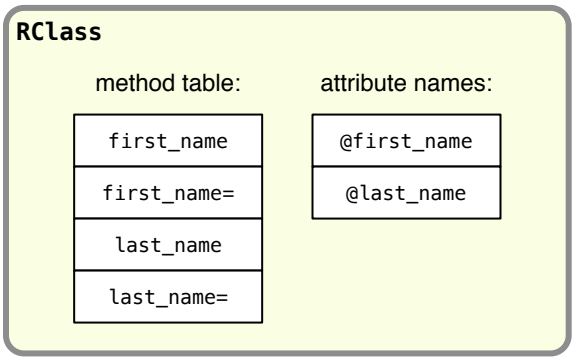
When taking a step back, and looking at this class, or any Ruby class, it looks like it is just a group of method definitions. I can assign behavior to an object by adding methods to its class, and when I call a method on an object, Ruby looks for the method in the object's class. This leads me to my first definition of what a Ruby class is:

A Ruby class is a group of method definitions.

Therefore, I know that the `RClass` structure for `Mathematician` must save a list of all the methods I defined in the class:



While reviewing my Ruby code above, notice that I've also created two instance variables called `@first_name` and `@last_name`. We saw earlier how Ruby stores these values in each `RObject` structure, but you may have noticed that the names of these variables were not stored in `RObject`, just the values were. (As I mentioned above, Ruby 1.8 actually stores the names in `RObject` as well.) Instead, Ruby must store the attribute names in `RClass`; this makes sense since the names will be the same for every `Mathematician` instance. Let's redraw `RClass` again, including a table of attribute names as well this time:



Now my definition of a Ruby class is:

> *A Ruby class is a group of method definitions and a table of attribute names.*

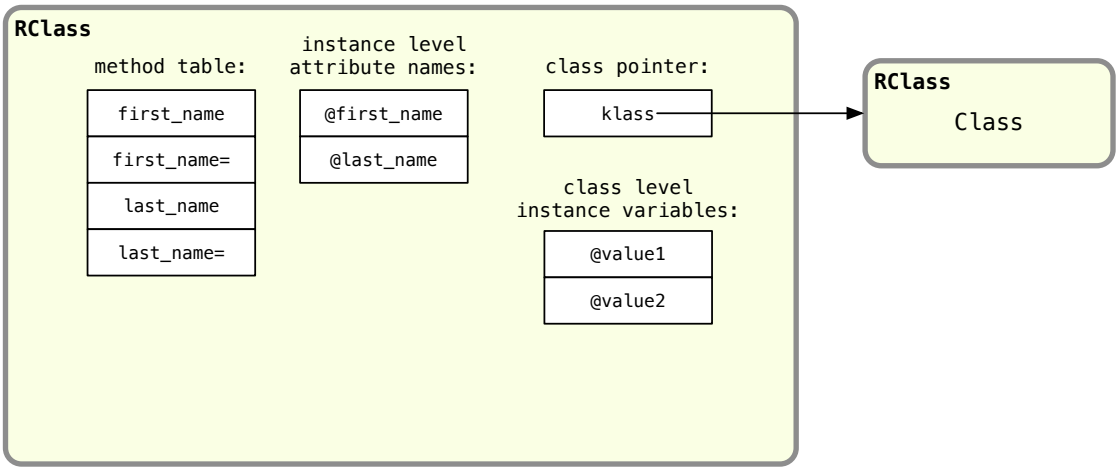
At the beginning of this chapter I mentioned that everything in Ruby is an object. This might be true for classes too. It's easy to prove this is, in fact, the case using IRB:

```
> p Mathematician.class
 => Class
```

You can see Ruby classes are all instances of the `Class` class; therefore, classes are also objects. Let's update our definition of a Ruby class again:

> *A Ruby class is a Ruby object that also contains method definitions and attribute names.*

Since Ruby classes are objects, we know that the `RClass` structure must also contain a class pointer and an instance variable array, the values that we know every Ruby object contains:



You can see I've added a pointer to the `Class` class, in theory the class of every Ruby class object. However, in Experiment 3-2 below I'll show that actually this diagram is not accurate, that `klass` actually points to something else! I've also added a table of instance variables. Note: these are the class level instance variables. Don't confuse this with the table of attribute names for the object level instance variables.

As you can see, this is rapidly getting out of control; the `RClass` structure seems to be much more complex than the `RObject` structure was! But, don't worry, we're almost done. In a moment I'll show you what the actual `RClass` structure looks like. But first, there are still two more important types of information we need to consider that each Ruby class contains.

Another essential feature of object oriented programming that we all know Ruby also implements is inheritance. Ruby implements single inheritance by allowing us to optionally specify one superclass when we create a class, or if we don't specify a superclass then Ruby assigns the `Object` class to be the superclass. For example, I could rewrite my Mathematician class using a superclass like this:

```
class Mathematician < Person
...
```

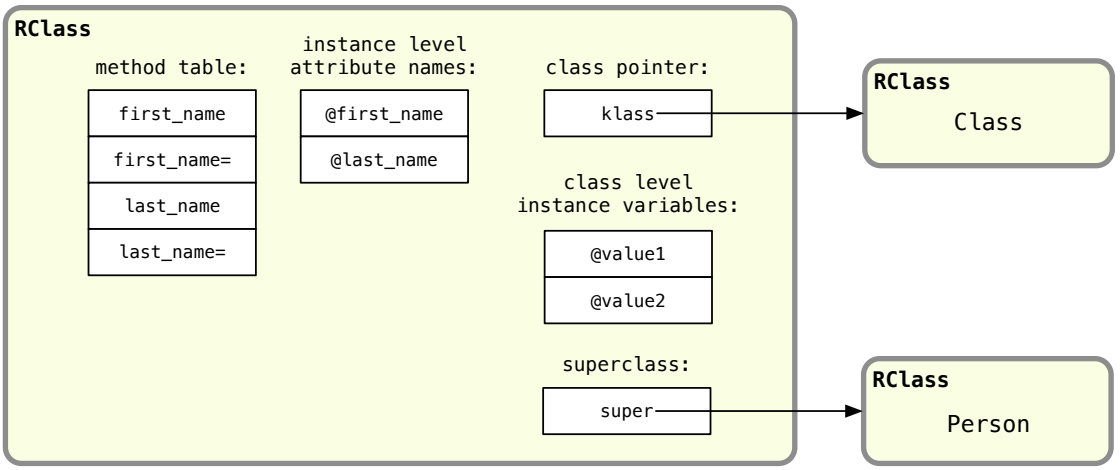
Now every instance of `Mathematician` will include the same methods that instances of `Person` have. In this example, I might want to move the `first_name` and `last_name` accessor methods into `Person`. I could also move the `@first_name` and `@last_name` attributes into the `Person` class, all instances of `Mathematician` would also share these attributes. Somehow the `Mathematician` class must contain a

reference to the Person class (its superclass) so that Ruby can find any methods or attributes that actually were defined in a superclass.

Let's update my definition again, assuming that Ruby tracks the superclass using another pointer similar to `klass`:

> *A Ruby class is a Ruby object that also contains method definitions, attribute names and a superclass pointer.*

And let's redraw the `RClass` structure including the new superclass pointer:



At this point it is critical to understand the difference between the `klass` pointer and the `super` pointer. The `klass` pointer indicates which class the Ruby class object is an instance of. This will always be the `Class` class:

```
> p Mathematician.class
 => Class
```

Ruby uses the `klass` pointer to find the methods of the `Mathematician` class, such as the `new` method which every Ruby class implements.

However, the `super` pointer records which class is the superclass of this class:

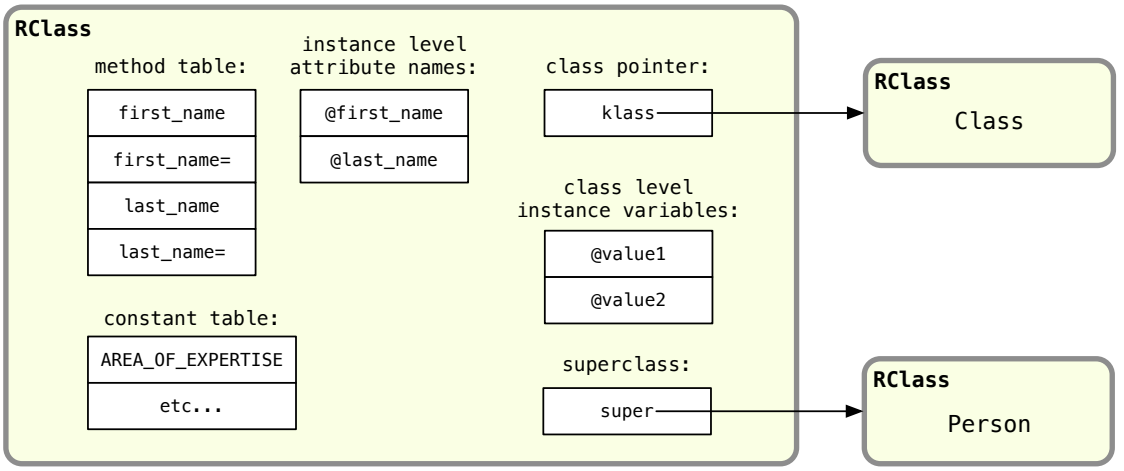
```
> p Mathematician.superclass
 => Person
```

Ruby uses the "super" pointer to help find methods that each `Mathematician` instance has, such as `first_name=` or `last_name`. I'll cover method lookup in the next section.

Now we have just one more feature of Ruby classes to cover: constants. As you probably know, Ruby allows you to define constant values inside of a class, like this:

```
class Mathematician < Person
  AREA_OF_EXPERTISE = "Mathematics"
  etc...
```

Constant values must start with a capital letter, and are valid within the scope of the current class. Curiously, Ruby actually allows you to change a constant value but will display a warning when you do so. Let's add a constant table to our `RClass` structure, since Ruby must save these values inside each class:



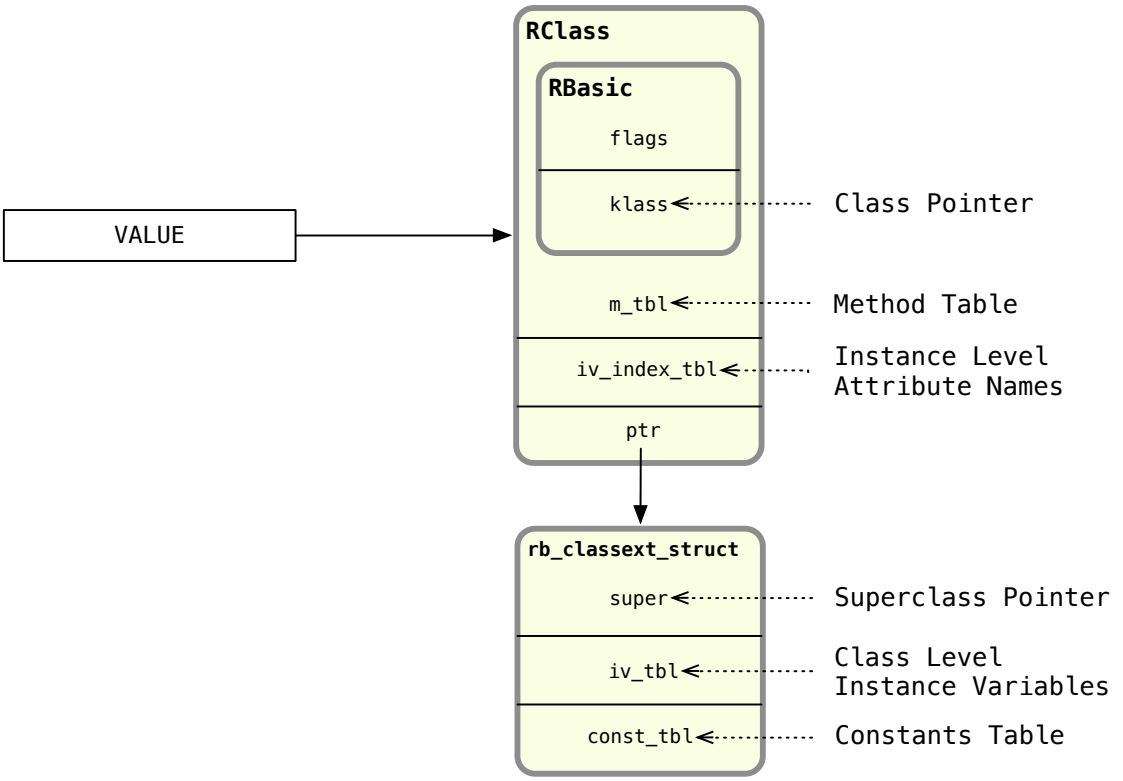
That's it – so now we can write a complete, technical definition of what a Ruby class is:

> *A Ruby class is a Ruby object that also contains method definitions, attribute names, a superclass pointer and a constants table.*

This isn't as concise as the simple definition we had for what a Ruby object is, but each Ruby class does actually contain much more information than each Ruby object does. Ruby classes are obviously fundamental to the language.

The actual RClass structure

Now that we have built up a conceptual model for what information must be stored in `RClass`, let's look at the actual C structure that Ruby uses to represent classes:



As you can see, Ruby actually uses two separate structures to represent each class: `RClass` and `rb_classext_struct`. But, these act as one large structure since each `RClass` always contains a pointer (`ptr`) to a corresponding `rb_classext_struct`. You might guess that the Ruby core team decided to use two different structures since there are so many different values to save, but actually they likely created `rb_classext_struct` to save internal values they didn't want to expose in the public Ruby C extension API.

Like I did for `RObject`, on the left I show a `VALUE` pointer. Ruby always accesses classes using these `VALUE` pointers. On the right, you can see the technical names for all of the fields we just discussed:

- `flags` and `klass` are the same `RBasic` values that every Ruby value contains.
- `m_tbl` is the method table, a hash whose keys are the names or id's of each method and whose values are pointers to the definition of each method, including the compiled YARV instructions.
- `iv_index_tbl` is the attribute names table, a hash that maps each instance variable name to the index of the attribute's value in each `RObject` instance variable array.
- `super` is a pointer to the `RClass` structure for this class's superclass.
- `iv_tbl` contains the class level instance variables – both their names and values.
- And finally `const_tbl` is a hash containing all of the constants – names and values – defined in this class's scope. You can see that Ruby implements `iv_tbl` and `const_tbl` in the same way; that is, class level instance variables and constants are almost the same thing.

Now let's take a quick look at the actual `RClass` structure definition:

```
typedef struct rb_classext_struct rb_classext_t;
struct RClass {
    struct RBasic basic;
    rb_classext_t *ptr;
    struct st_table *m_tbl;
    struct st_table *iv_index_tbl;
};
```

Like the `RObject` definition we saw earlier, you can find this structure definition in the include/ruby/ruby.h file. You can see all of the values I showed in the previous diagram.

The `rb_classext_struct` structure definition, on the other hand, can be found in the internal.h C header file:

```
struct rb_classext_struct {
    VALUE super;
    struct st_table *iv_tbl;
```

```
    struct st_table *const_tbl;
};
```

Once again, you can see the values I showed in the diagram. In Chapter 4 I'll cover hash tables in detail, the `st_table` type here, which Ruby uses to save all of these values: the method table, the constant table, the instance variables for the class and also the instance variable names/id's for object instances of this class.

Experiment 3-2: Where does Ruby save class methods?

Above we saw how each `RClass` structure saves all the methods defined in a certain class; in my example:

```
class Mathematician
  def first_name
    @first_name
  end
```

Ruby stores information about the `first_name` method inside the `RClass` structure for `Mathematician` using the method table.

But what about class methods? It's a common idiom in Ruby to save methods in a class directly, using this syntax:

```
class Mathematician
  def self.class_method
    puts "This is a class method."
  end
```

Or this syntax:

```
class Mathematician
  class << self
    def class_method
      puts "This is a class method."
    end
  end
```

Are they saved in the `RClass` structure along with the normal methods for each class, maybe with a flag to indicate they are class methods and not normal methods? Or are they saved somewhere else? Let's find out!

It's easy to see where class methods are *not* saved. They are obviously not saved in the `RClass` method table along with normal methods, since instances of `Mathematician` cannot call them:

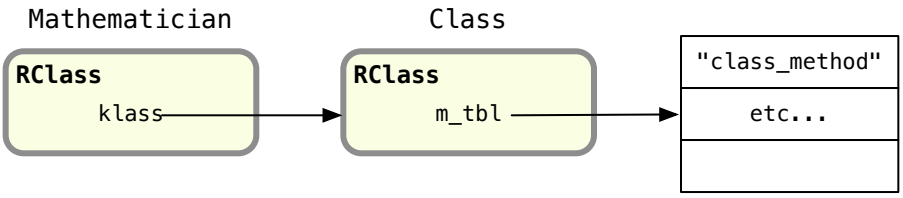
```
obj = Mathematician.new
obj.class_method
```

```
=> undefined method `class_method' for
#< Mathematician:0x007fdd8384d1c8 (NoMethodError)
```

Thinking about this some more, since `Mathematician` is also a Ruby object – remember my definition from above:

> *A Ruby class is a Ruby object that also contains method definitions, attribute names, a superclass pointer and a constants table.*

…then Ruby should save methods for `Mathematician` in the same way it saves them for any object: in the method table for the object's class. That is, Ruby should get `Mathematician`'s class using the `klass` pointer and save the method in the method table in that `RClass` structure:



But actually Ruby doesn't do this – you can prove this is the case by creating another class and trying to call the new method:

```
> class AnotherClass; end
> AnotherClass.class_method
=> undefined method `class_method' for AnotherClass:Class (NoMethodError)
```

If Ruby had added the class method to the method table in the `Class` class, then all classes in your application would have the method. Obviously this isn't what we intended by writing a class method, and thankfully Ruby doesn't implement class methods this way.

Then where do the class methods go? You can find a clue by using the `ObjectSpace.count_objects` method, as follows:

```
$ irb
> ObjectSpace.count_objects[:T_CLASS]
 => 859
> class Mathematician; end
```

```
 => nil
> ObjectSpace.count_objects[:T_CLASS]
 => 861
```

`ObjectSpace.count_objects` returns the number of objects of a given type that currently exist. In this test, I'm passing the `T_CLASS` symbol to get the count of class objects that exist in my IRB session. Before I create `Mathematician`, there are 859 classes. After I declare `Mathematician`, there are 861 – two more. This seems a bit odd… I declared one new class but Ruby actually created two! What is the second one for? Where is it?

It turns out whenever you create a new class internally Ruby always creates two classes! The first class is your new class: Ruby creates a new `RClass` structure to represent your class as I have described above. But internally Ruby also creates a second, hidden class called the "metaclass." Why? Just for this reason: to save any class methods you might later create for your new class. In fact, Ruby sets the metaclass to be the `class` of your new class – it sets the `klass` pointer of your new `RClass` structure to point to the metaclass.

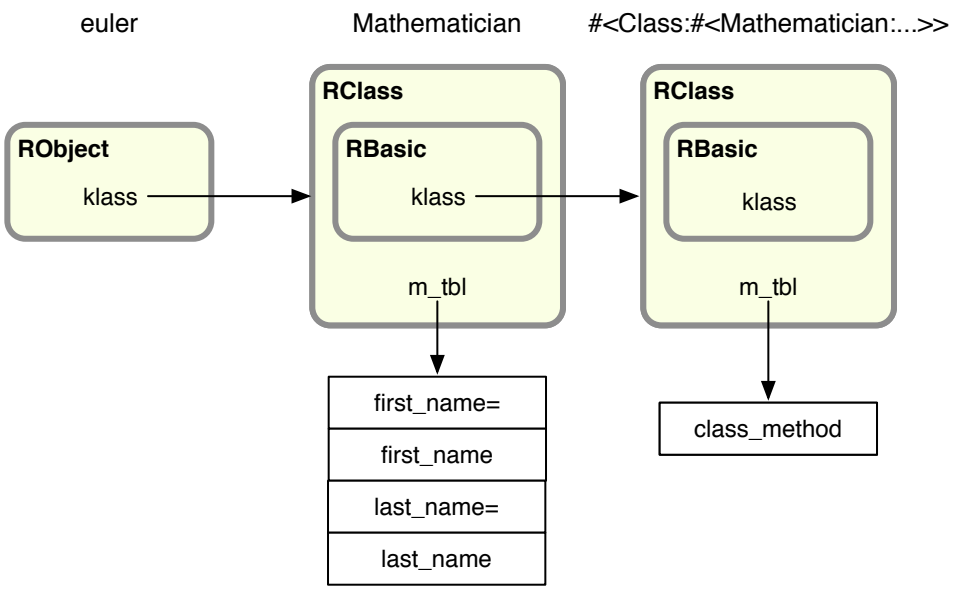
Without writing C code, there's no easy way to see the metaclass or the `klass` pointer value, but you can obtain the metaclass as a Ruby object as follows:

```
class Mathematician
end
obj = Mathematician.new
p obj.class
p obj.singleton_class
```

Running this I get:

```
$ ruby metaclass.rb
Mathematician
#<Class:#< Mathematician:0x007fb6228856c8>>
```

The first line displays the object's class, while the second line displays the object's metaclass; the odd "`#<Class:#< Mathematician…`" syntax indicates that the second class is the metaclass for `Mathematician`. This is the second `RClass` structure that Ruby automatically created for me when I declared the `Mathematician` class. And this second `RClass` structure is where Ruby saves my class method:



If I now display the methods for the metaclass, I'll see all the usual Ruby `Class` methods, along with my new class method for `Mathematician`:

```
p obj.singleton_class.methods
=> [ ... :class_method, ...  ]
```

How Ruby implements modules and method lookup

You can mix multiple modules into one class.

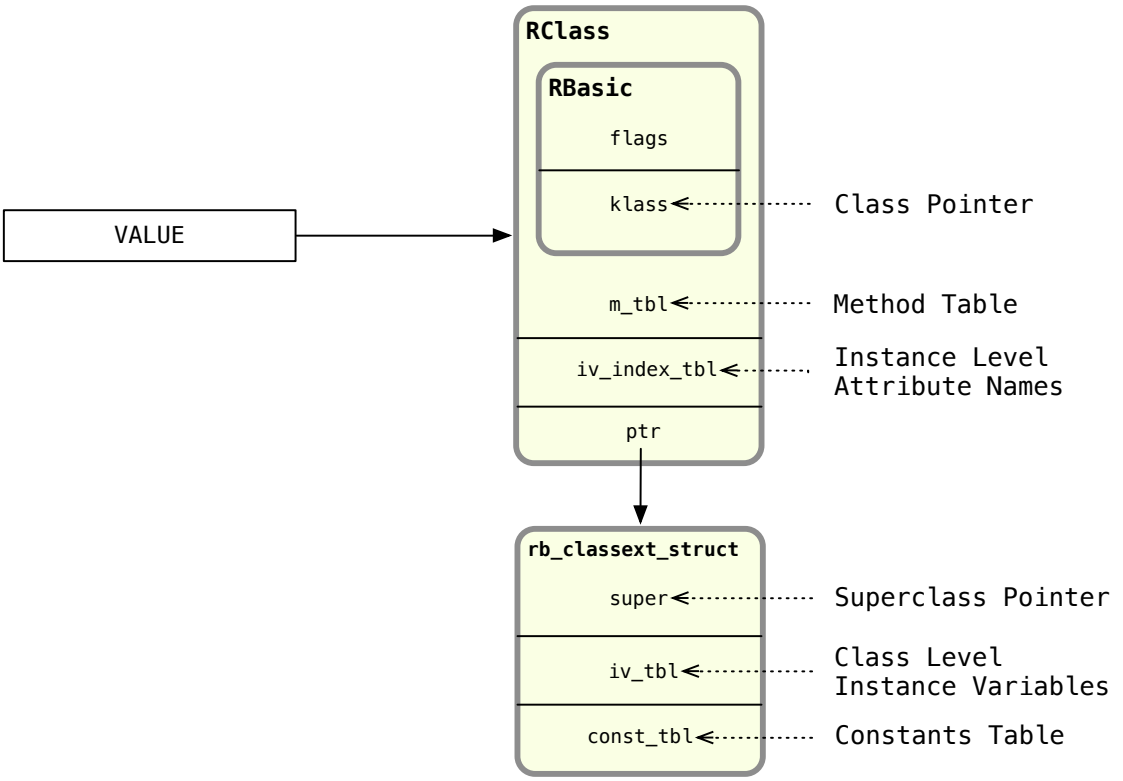
Most Ruby developers are used to the idea that Ruby only supports single inheritance; unlike C++ for example, you can only specify one superclass for each class. However, Ruby does allow for multiple inheritance in an indirect way using modules. You can include as many different modules into a class as you wish, each of them adding new methods and behavior.

How do modules work? Following the same pattern we've seen with `RObject` and `RClass`, is there also an `RModule` structure that defines a Ruby module? And how does Ruby keep track of which modules have been included in which classes? Finally, how does Ruby lookup methods? How does it know whether to search for a certain method in a class or a module?

It turns out that Ruby doesn't use an `RModule` structure. Internally Ruby implements modules as classes. Whenever you create a module, Ruby actually creates another `RClass` — `rb_classext_struct` structure pair, just like it would for a new class. For example, when I define a new module like this:

```
module Professor
end
```

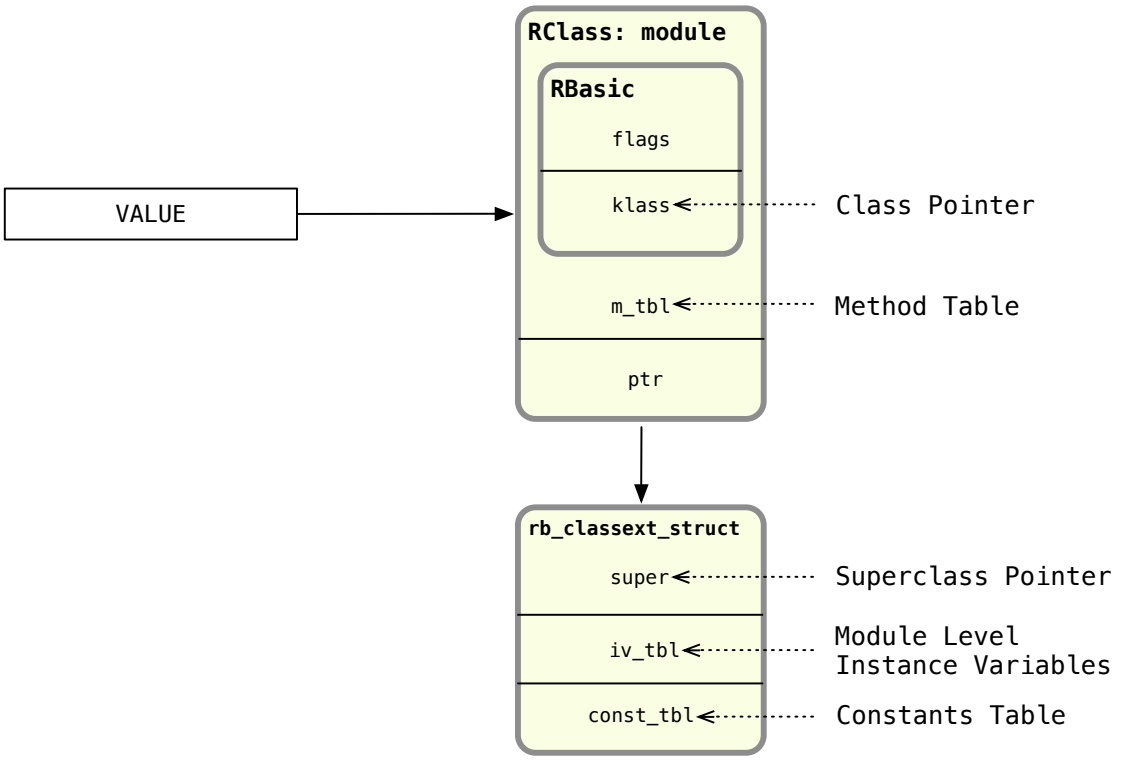
…internally Ruby will create a class, not a module! Here are the class structures again:



However, while internally modules are really classes they are still different from classes in two important ways:

- Ruby doesn't allow you to create objects directly from modules – this means you can't call the `new` method on a module. `new` is a method of `Class`, and not of `Module`.
- Ruby doesn't allow you to specify a superclass for a module.

So in fact modules don't use the `iv_index_tbl` value, since there are no object level attributes to keep track of. Modules don't have object instances. Therefore, we can imagine modules using a slightly smaller version of the `RClass` structures:



Following the same train of thought, we can write a technical definition of a Ruby module as follows:

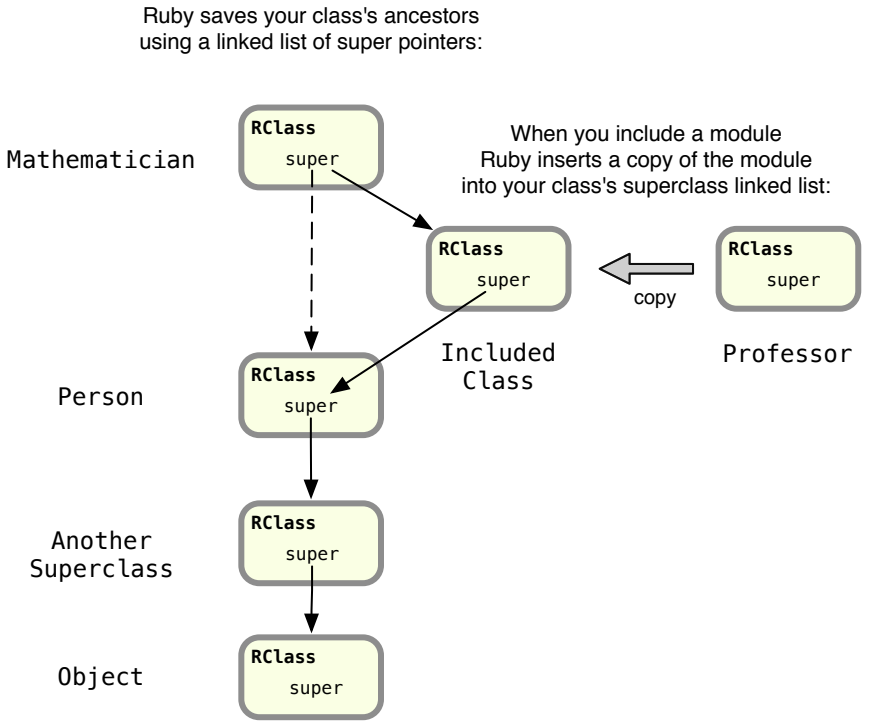
> *A Ruby module is a Ruby object that also contains method definitions, a superclass pointer and a constants table.*

What happens when you include a module in a class?

The real magic behind modules happens when you include one into a class. At the moment you include a module into a class, for example:

```
module Professor
end


class Mathematician
  include Professor
end
```

Ruby creates a copy of the `RClass` structure for the `Professor` module and inserts it as the new superclass for `Mathematician`. Ruby's C source code refers to this copy of the module as an "included class." The superclass of the new copy of `Professor` is set to the original superclass of `Mathematician`, preserving the superclass or "ancestor chain:"

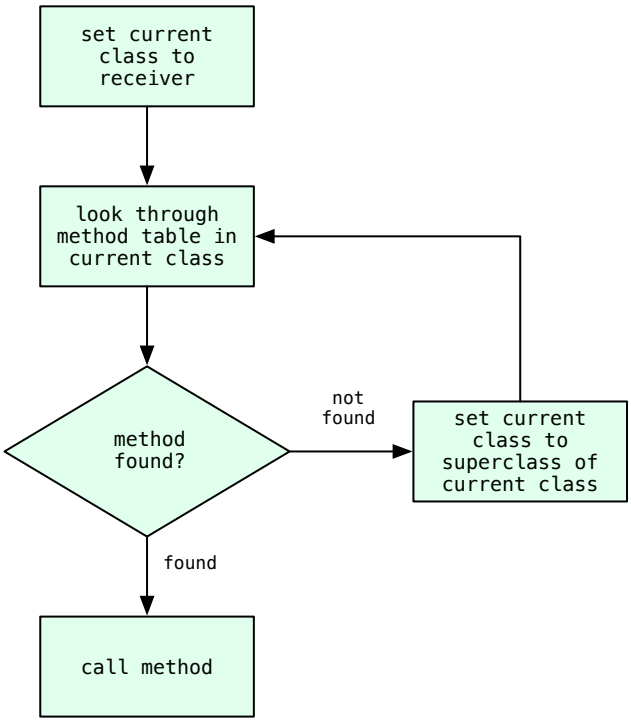


Here I've kept things simple by only displaying the `RClass` structures and not the `rb_classext_struct` structures, which actually hold the `super` pointers.

Ruby's method lookup algorithm

Why go to all of this trouble? Why does Ruby bother to change all of the super pointers to make included modules behave as if they were superclasses? Ruby does this to allow its method lookup algorithm to work properly, taking both superclasses and modules into account.

Understanding Ruby's method lookup algorithm thoroughly is essential for every Ruby developer, so let's take a close look at it:



What surprised me about this algorithm is how simple it is; Ruby simply follows the `super` pointers until it finds the class or module containing the target method. I had always imagined this would be a much more complex process: that Ruby would have to distinguish between modules and classes using some special logic, that it would have to handle the case when there were multiple included modules with some special code, and more. But no, it's very simple, just a simple loop on the `super` pointer linked list.

Let's take an example and walk through the method lookup process. Suppose I decide to move my first and last name attributes out of `Mathematician` and into the `Person` superclass like this:

```
class Person
  attr_accessor :first_name
  attr_accessor :last_name
end
```

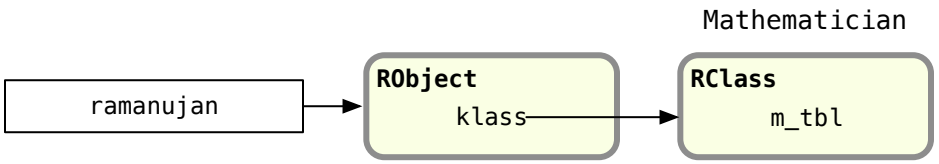
Remember my `Mathematician` class uses `Person` as the superclass and also now includes the `Professor` module:

```
module Professor
  def lectures; ...etc... end
end


class Mathematician < Person
  include Professor
end
```

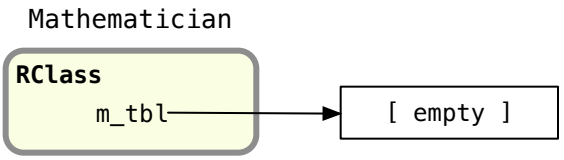
Now, suppose I set the first name of a mathematician:

```
ramanujan = Mathematician.new
ramanujan.first_name = "Srinivasa"
```

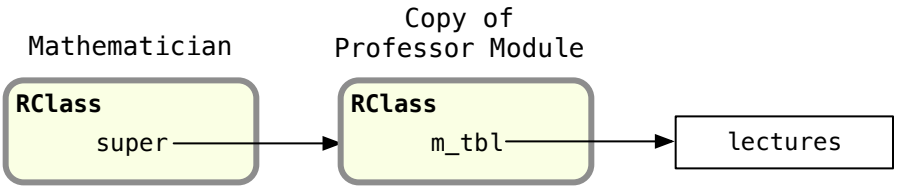
To execute this code, Ruby needs to find the `first_name=` method. To do this, Ruby will start by taking the `ramanujan` object and getting it's class via the `klass` pointer:



Then Ruby will look to see if `Mathematician` implements `first_name=` directly by looking through its method table:



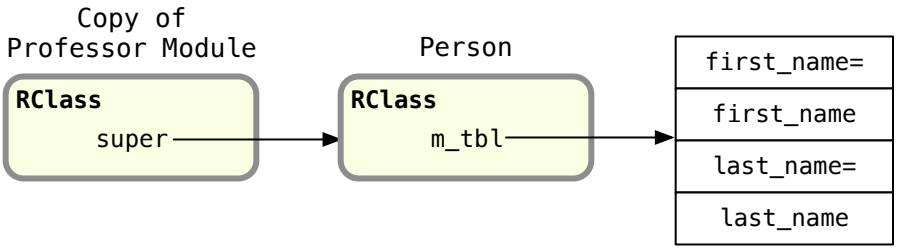
Since I moved all of the methods down into the `Person` superclass, the `first_name=` method is no longer there. Instead Ruby will get the superclass of Mathematician using the `super` pointer:



Remember, this is not the `Person` class but instead is the "included class," or copy of the `Professor` module. Ruby will now look through the method table for `Professor`, but will only find the `lectures` method, and not `first_name=`.

An important detail here is that, because of the way Ruby inserts modules above the original superclass in the superclass chain, methods in an included module will override methods present in a superclass. In this example, if `Professor` also had a `first_name=` method, Ruby would call it and not the method in `Person`.

Since in this example Ruby doesn't find `first_name=` in `Professor`, it will continue to iterate over the `super` pointers – this time using the `super` pointer in `Professor`:



Note the superclass of the `Professor` module, or more precisely, the superclass of the included class copy of the `Professor` module, is now actually the `Person` class. This was the original superclass of `Mathematician`. Finally, Ruby can find the `first_name=` method and call it.

What is interesting here is that internally Ruby implements module inclusion using class inheritance. Saying that in a different way, there is no difference at all between including a module and specifying a superclass. Both make new methods available to the target class, and internally both use the class's `super` pointer. Including multiple modules in a Ruby class really is equivalent to specifying multiple superclasses.

However, Ruby keeps things simple by enforcing a single list of ancestors. While including multiple modules does create multiple superclasses internally, Ruby maintains them in a single list for you. As a Ruby developer, you get the benefits of multiple inheritance – adding new behavior to class from as many different modules as you would like – while keeping the simplicity of the single inheritance model. Ruby itself benefits from this simplicity as well! By enforcing this single list of superclass ancestors, Ruby's method lookup algorithm can be very simple. Whenever you call a method on an object, all Ruby has to do is iterate through the superclass linked list until it finds the class or module that contains the target method.

Including two modules in one class

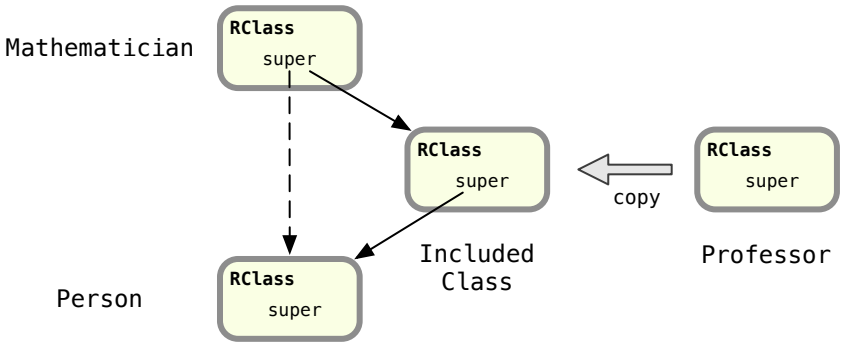
Ruby's method lookup algorithm is simple, but the code it uses to include modules is not. As we saw above, when you include a module in a class, Ruby inserts a copy of the module into the class's ancestor chain. This also means if you include two modules, one after the other, the second module will appear first in the ancestor chain… and will be found first by Ruby's method lookup logic.

For example, suppose I include two modules into `Mathematician`:

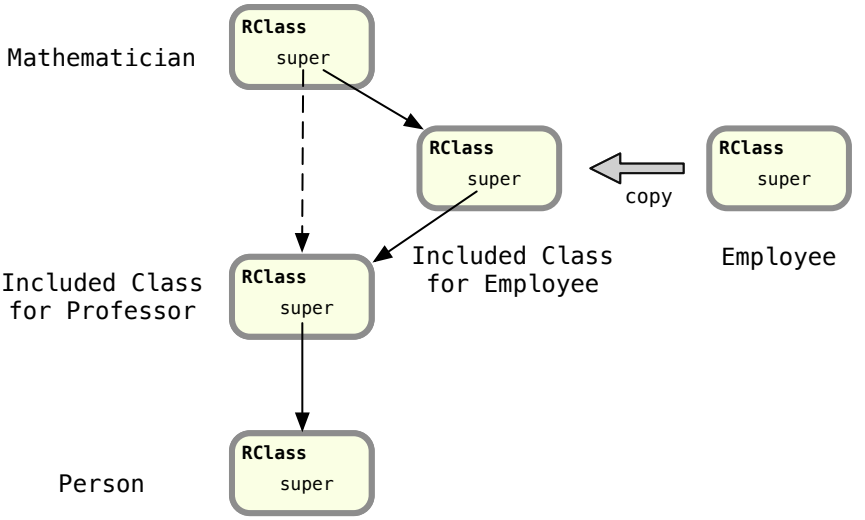
```
class Mathematician < Person
  include Professor
  include Employee
end
```

Now `Mathematician` objects have methods from the `Professor` module, the `Employee` module and the `Person` class. But which methods will Ruby find first? Which methods override which?

Using a diagram, it's easy to see the order: since I include the `Professor` module first, Ruby inserts its copy as a superclass first:



And now when I include the `Employee` module, its copy will be inserted above the `Professor` module's copy using the same process:



This means that methods from `Employee` will override methods from `Professor`, which in turn will override methods from `Person`, the actual superclass.

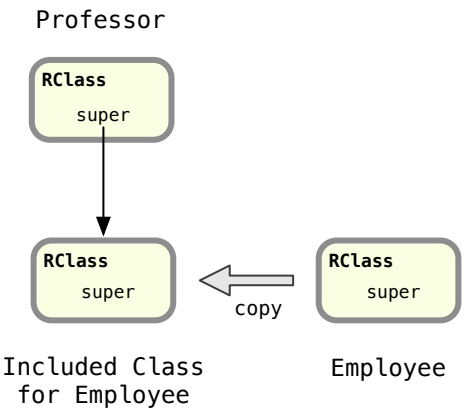
Finally, modules don't allow you to specify superclasses; i.e., I can't write:

```
module Professor < Employee
end
```

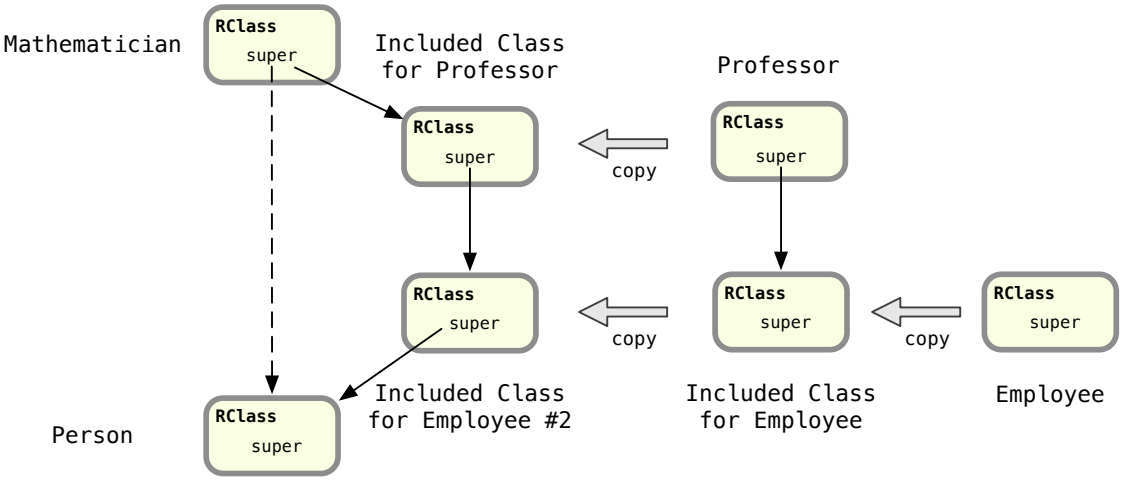
But I can include one module into another like this:

```
module Professor
  include Employee
end
```

Now what happens when I include `Professor`, a module with other modules included in it, into `Mathematician`? Which methods will Ruby find first? Here's what happens: first, when I include `Employee` into `Professor`, Ruby will create a copy of `Employee` and set it as the superclass of `Professor` internally:



That's right: modules can't have a superclass in your code, but inside of Ruby they can! This is because Ruby represents modules with classes internally. And now, finally, when I include `Professor` into `Mathematician`, Ruby iterates over the two modules and inserts them both as superclasses of `Mathematician`:



Now Ruby will find the methods in `Professor` first, and `Employee` second.

Experiment 3-3: Modifying a module after including it

Following a suggestion by Xavier Noria, this experiment will look at what happens when you modify a module after it's been included into a class. Let's reuse the same `Mathematician` class and the `Professor` module:

```
module Professor
  def lectures; end
end
```

```
class Mathematician
  attr_accessor :first_name
  attr_accessor :last_name
  include Professor
end
```

This time the `Mathematician` class contains the accessor methods for `@first_name` and `@last_name`, and I've also included the `Professor` module. If I inspect the methods of a mathematician object, I should see both the attribute methods, `first_name=`, etc., and the `lectures` method which came from `Professor`:

```
fermat = Mathematician.new
fermat.first_name = 'Pierre'
fermat.last_name = 'de Fermat'


p fermat.methods.sort
 => [ … :first_name, :first_name=, … :last_name, :last_name=, :lectures … ]
```

No surprise; I see all the methods.

Now let's try adding some new methods to the `Professor` module after including it in the `Mathematician` class. Is Ruby smart enough to know the new methods should be added to `Mathematician` as well? Let's find out.

```
module Professor
  def primary_classroom; end
end
```

```
p fermat.methods.sort
=> [ ... :first_name, :first_name=, ... :last_name, :last_name=, :lectures,
... :primary_classroom, ... ]
```

As you can see, I get all the methods, including the new `primary_classroom` method added to `Professor` after it was included in `Mathematician`. No surpise again – Ruby is one step ahead of me.

Now let's try one more test. What will happen if I re-open the `Professor` module and include yet another module into to it:

```
module Employee
  def hire_date; end
end


module Professor
  include Employee
end
```

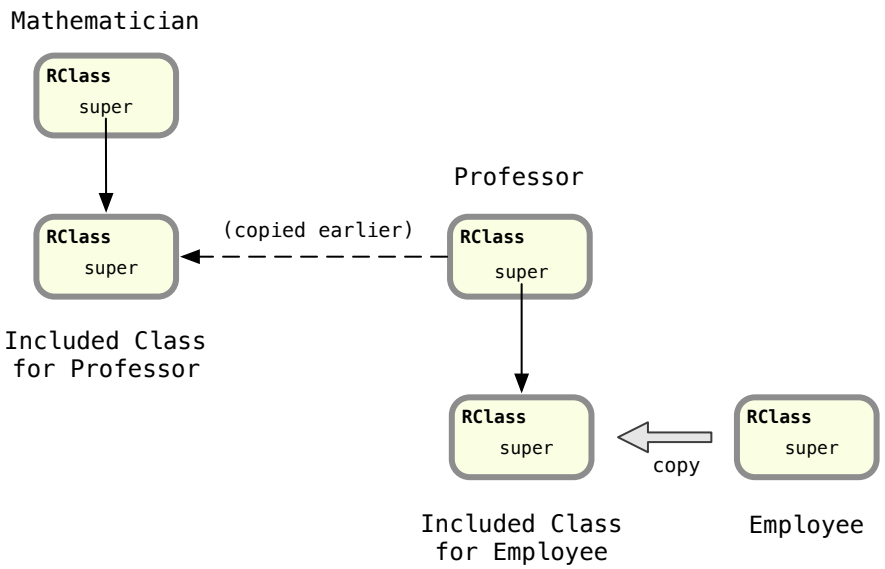
This is getting somewhat confusing now, so let me summarize what I've done so far:

- First I included the `Professor` module in the `Mathematician` class.
- Then I included the `Employee` module in the `Professor` module.
- Therefore, methods of the `Employee` module should now be available on a mathematician object.

Let's see if Ruby works as I expect:

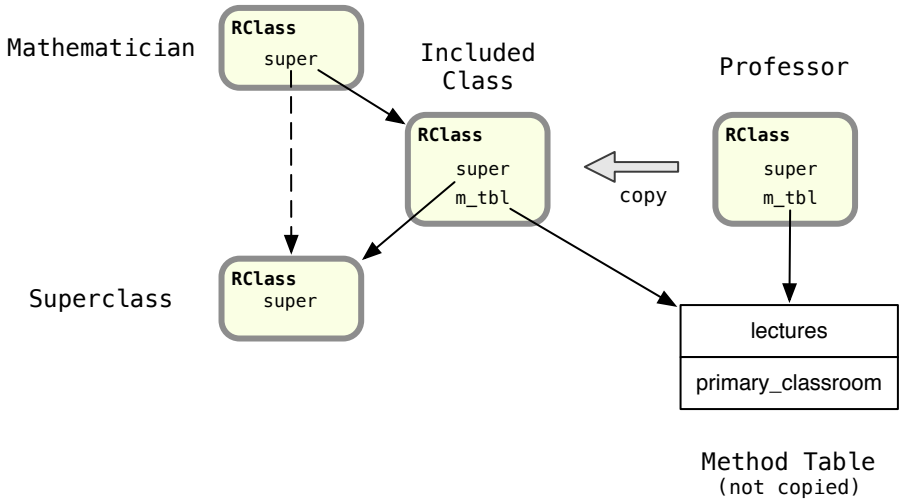
```
p fermat.methods.sort
 => [ … :first_name, :first_name=, … :last_name, :last_name=, :lectures … ]
```

The `hire_date` method is *not* available in the `fermat` object. Including a module into a module that was already included into a class *does not* effect that class. After learning about how Ruby implements modules this shouldn't be too hard to understand. Including `Employee` into `Professor` does change the `Professor` module, but not the copy of `Professor` that Ruby created when I included it in `Mathematician`:



But what about the `primary_classroom` method? How was Ruby able to include `primary_classroom` in `Mathematician`, even though I added it to `Professor` after I included `Professor` in `Mathematician`? Looking at the diagram above, it's clear Ruby created a copy of the `Professor` module before I added the new method to it. But the `fermat` object gets the new method… how?

To understand this, we need to take a closer look at how Ruby copies modules when you include them into a class. It turns out that Ruby copies the `RClass` structure, but not the underlying module table! Here's what I mean:



Ruby doesn't copy the method table for `Professor`. Instead, it simply sets `m_tbl` in the new copy of `Professor`, the "included class," to point to the same method table. This means that modifying the method table, reopening the module and adding new methods, will change both the module and any classes it was already included in.

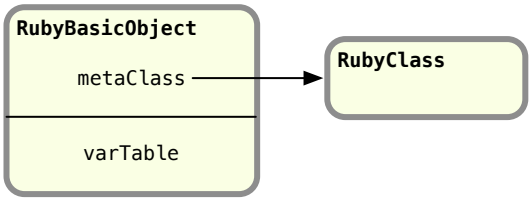
Objects, classes and modules in JRuby

In Chapter 2 I showed how JRuby executes your code at a very, very high level:

- JRuby compiles your Ruby code into a Java class containing byte code instructions.
- Meanwhile, JRuby provides a series of Java classes that implement Ruby's intrinsic behavior and the built in Ruby classes, such as `String`, `Array` or `Fixnum`.

To learn what I mean by "Ruby's intrinsic behavior" better let's take a look now at how JRuby implements Ruby's object model: objects, classes and modules. We saw earlier that MRI Ruby uses the `RObject` and `RClass` C structures to represent these concepts - does JRuby use similar structures?

As you might guess, instead of C structures JRuby uses a series of Java objects. In fact, JRuby creates one Java object for each Ruby object you create in your application. The common superclass for all of these Java objects is called `RubyBasicObject`, named after the Ruby `BasicObject` class.



In the diagram I've shown two of the instance variables present in this Java class:

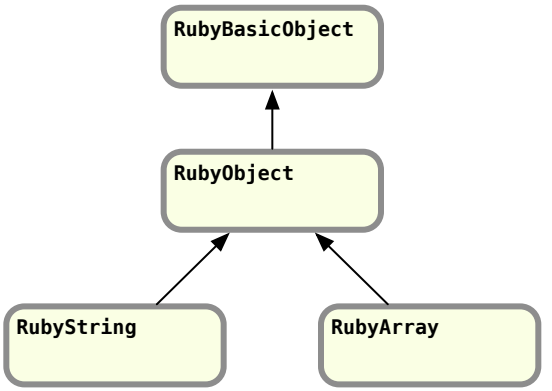
- `metaClass` is a reference to an instance of the `RubyClass` Java class; this indicates which Ruby class this Ruby object is an instance of.
- `varTable[]` is an array containing the instance variables stored inside of this Ruby object.

You can see that the Java `RubyBasicObject` class meets the requirements of our definition of a Ruby object from earlier:

> *Every Ruby object is the combination of a class pointer and an array of instance variables.*

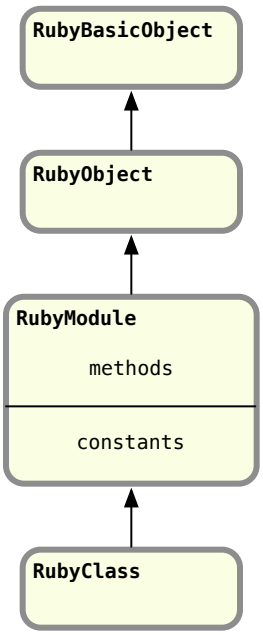
Here `metaClass` is the class pointer and `varTable[]` is the instance variable array.

Since JRuby uses an object oriented implementation in Java to implement Ruby objects, it able to take advantage of Java object inheritance to create a series of subclasses representing different types of Ruby objects:



This elegant design allows JRuby to organize the code that implements all of these classes, while sharing a base class that provides the common behavior. This is analogous to MRI including the `RBasic` C structure inside each `RObject` and other Ruby object structures.

Next let's look at how JRuby represents Ruby classes; I mentioned above that JRuby has a Java class called `RubyClass`. It turns out that this Java class is actually a subclass of the `RubyModule` Java class:



What's interesting about this is that JRuby's internal implementation of modules and classes actually reflects their meaning and usage in the Ruby language:

- Both classes and modules are also Ruby objects - they are both derived from `RubyObject` and `RubyBasicObject`.
- The superclass of the `RubyClass` Java class is the `RubyModule` Java class, just as the superclass of the Ruby `Class` class is the Ruby `Module` class. Instead of using a structure like `RClass` to implement both classes and modules like MRI does, JRuby saves the method definitions and constant table in the `RubyModule` Java class where they belong.

Reviewing our definition of a Ruby class:

> *A Ruby class is a Ruby object that also contains method definitions, attribute names, a superclass pointer and a constants table.*

- It's derived from `RubyBasicObject`, which contains all of the Ruby object information, and

- It's derived from `RubyModule`, which contains the superclass pointer, method definitions and the constants table.

Above you can see in JRuby the `RubyModule` class contains the method definition table, which in MRI was saved in the `RClass` structure. Let's take a look now at how JRuby's `RubyModule` class implements the method lookup algorithm we saw earlier. Here's a method from org/jruby/RubyModule.java that looks up a method given a name:

```
public DynamicMethod searchMethod(String name) {
  return searchWithCache(name).method;
}
```

The first thing you'll notice here is that JRuby implements some sort of a cache. If we follow along and look at the `searchWithCache` method, we see:

```
public CacheEntry searchWithCache(String name) {
  CacheEntry entry = cacheHit(name);
  if (entry != null) return entry;


...
```

I won't try to explain how the cache actually works, but here you can see if JRuby finds the requested method in the cache it will return it immediately. If not, JRuby continues to actually lookup the method without a cache using a method called `searchMethodInner`:

```
DynamicMethod method = searchMethodInner(name);
```

... and here's the implementation of `seachMethodInner`:

```
public DynamicMethod searchMethodInner(String name) {
  DynamicMethod method = getMethods().get(name);

  if (method != null) return method;

  return superClass == null ? null :
                              superClass.searchMethodInner(name);
}
```

Here's how this works: first, JRuby calls `getMethods()`. This returns the actual method table which JRuby implements using a Java map:

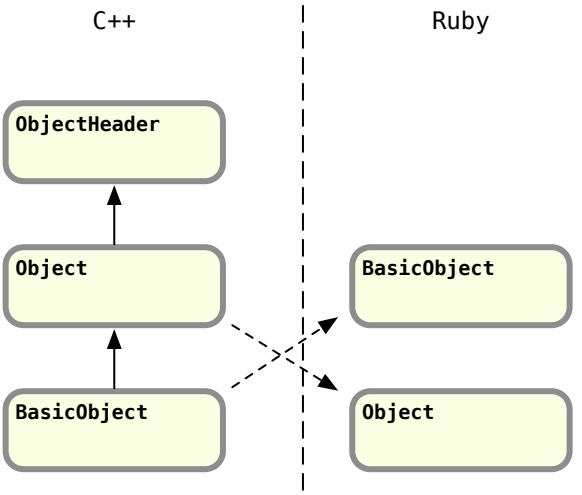
```
private volatile Map<String, DynamicMethod> methods =
                                              Collections.EMPTY_MAP;
```

Next, the `get(name)` call will lookup the method name in the map. If it's found, JRuby returns it. If it's not found, then JRuby recursively calls the `searchMethodInner` method on the super class `RubyModule`, if there is one.

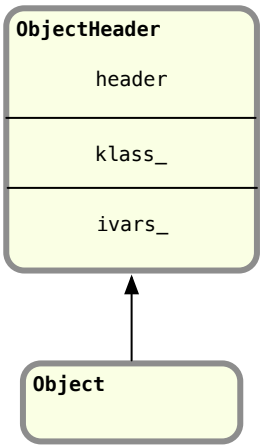
Objects, classes and modules in Rubinius

In Chapter 2 I explained how Rubinius executes your code using a combination of Ruby and C++. We saw how Rubinius's kernel contains pure Ruby implementations for all of the core classes, such as `Array`, `String` and `Hash`. For the portions of these core classes that cannot be implemented directly in Ruby, Rubinius uses native code written in a corresponding C++ class, compiled into the Rubinius VM.

This applies to the core classes behind Ruby's object model as well. Here is how Rubinius represents Ruby objects internally:



On the left are three C++ classes: `ObjectHeader`, `Object`, and `BasicObject`. These are related using C++ class inheritance, indicated by the arrows. The `Object` and `BasicObject` C++ classes correspond to the Ruby core classes with the same name shown on the right. However, inside the Rubinius C++ VM, the `Object` class is the common base class for all Rubinius objects, while `BasicObject` is actually a subclass of `Object`. This is the opposite of what we have in Ruby, where `BasicObject` is the superclass of `Object`. The `ObjectHeader` class, similar to the `RBasic` structure in MRI, contains some basic technical information Rubinius keeps track of for every object:

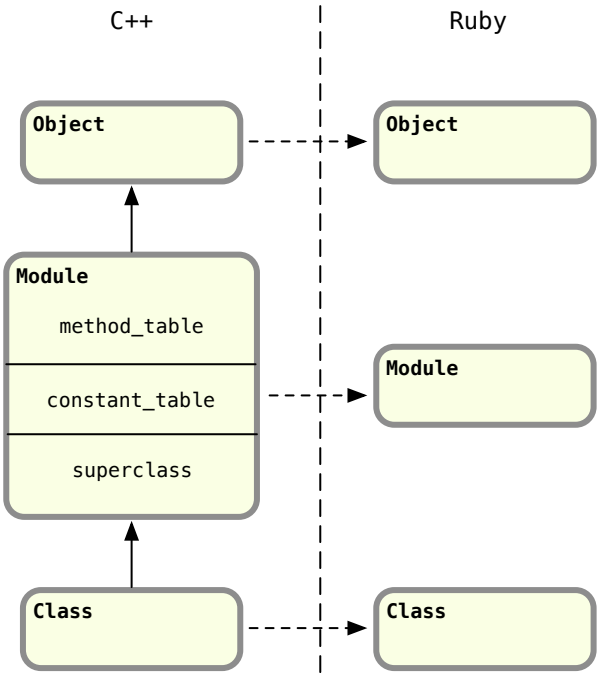


- `header` is an instance of the `HeaderWord` C++ class. This contains some technical flags Rubinius keeps track of for each object.
- `klass_` is a pointer to a C++ class called `Class`. This is the class of this Ruby object.
- `ivars_` is a pointer to a C++ object that contains a table of the instance variables stored in this object. Rubinius stores the variable names in this table also, like Ruby 1.8.

Since the Rubinius `Object` C++ class is a subclass of `ObjectHeader`, you can see it also meets our definition of a Ruby object:

> *Every Ruby object is the combination of a class pointer and an array of instance variables.*

Next let's briefly look at how Rubinius implements classes and modules:



Again you can see a one to one correspondence between Ruby and C++ classes. This time the C++ class inheritance model reflects the Ruby object model; the `Class` class is a subclass of the `Module` class. Finally you can see Rubinius classes meet our previous definition since they contain all of the same information:

> *A Ruby class is a Ruby object that also contains method definitions, attribute names, a superclass pointer and a constants table.*

Rubinius stores the attribute names in the instance variable table, part of `Object`, and not in the `Module` object.

Chapter 4
Hash Tables

Ruby stores much of its own internal data in hash tables.

Back in Experiment 3-1 we saw how in Ruby 1.9 and 2.0 the `ivptr` member of the `RObject` structure pointed to a simple array of instance variable values. We saw adding a new value was usually very fast, but that while saving every third or fourth instance variable Ruby was somewhat slower, since it had to allocate a larger array. Looking more broadly across the MRI C source code base, it turns out this technique of repeatedly allocating larger and larger arrays to save data values is unusual. This should't be a surprise, since repeatedly increasing the size of an array over and over by a small amount is probably inefficient.

In fact, Ruby instead saves much of its own internal data in a memory structure called a "hash table." Unlike the simple array we saw in Experiment 3-1, hash tables can automatically expand to accommodate more values. There's no need for the user or client of a hash table to worry about how much space is available or about allocating more memory for it.

As you might guess, Ruby uses a hash table to hold the data you save in the `Hash` objects you create in your Ruby script. However, Ruby uses hash tables for many other reasons as well: it saves much of its own internal data in hash tables. Every time you create a method, Ruby inserts a new value in a hash table. Every time you create a constant, Ruby inserts a new value in a hash table. Ruby saves many of the special variables we saw in Experiment 2-2 in hash tables. As we saw in Chapter 3, Ruby saves instance variables for generic objects, such as integers or symbols, in a hash table. Hash tables are the work horse of Ruby internals.

In Chapter 4 I'll start by explaining how hash tables work: what happens inside the table when you save a new value with a key, and when you later retrieve the value again

using the same key. Later I'll explain how hash tables automatically expand to accommodate more values. Finally, we'll look at how hash functions work in Ruby.

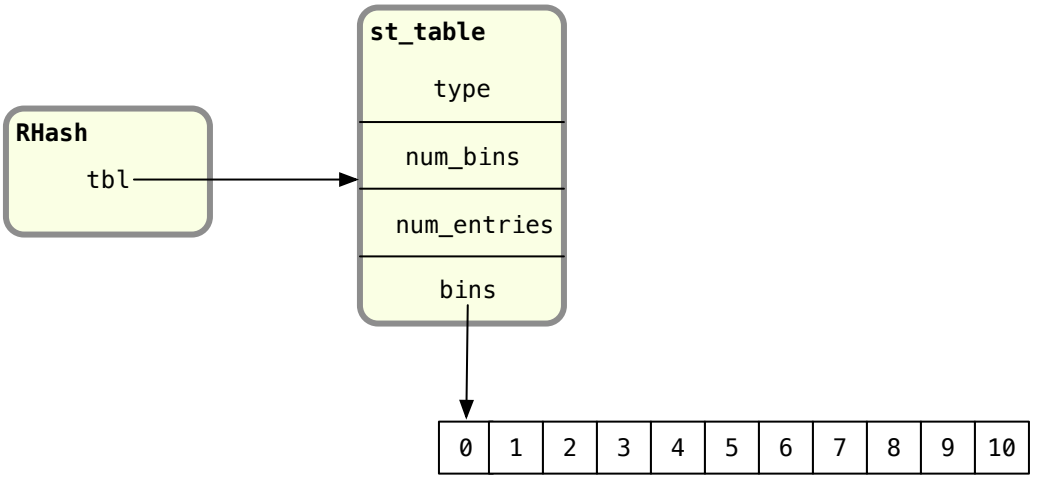
Chapter 4 Roadmap

Hash tables in Ruby

Every time you write a method,
Ruby creates an entry in a hash table.

Hash tables are a commonly used, well known, old concept in computer science. They organize values into groups or "bins" based on an integer value calculated from each value called a "hash." Later when you need to search for and find a value, by recalculating the hash value you can figure out which bin the value is contained in, to speed up the search.

Here's a high level diagram showing a single hash object and its hash table:



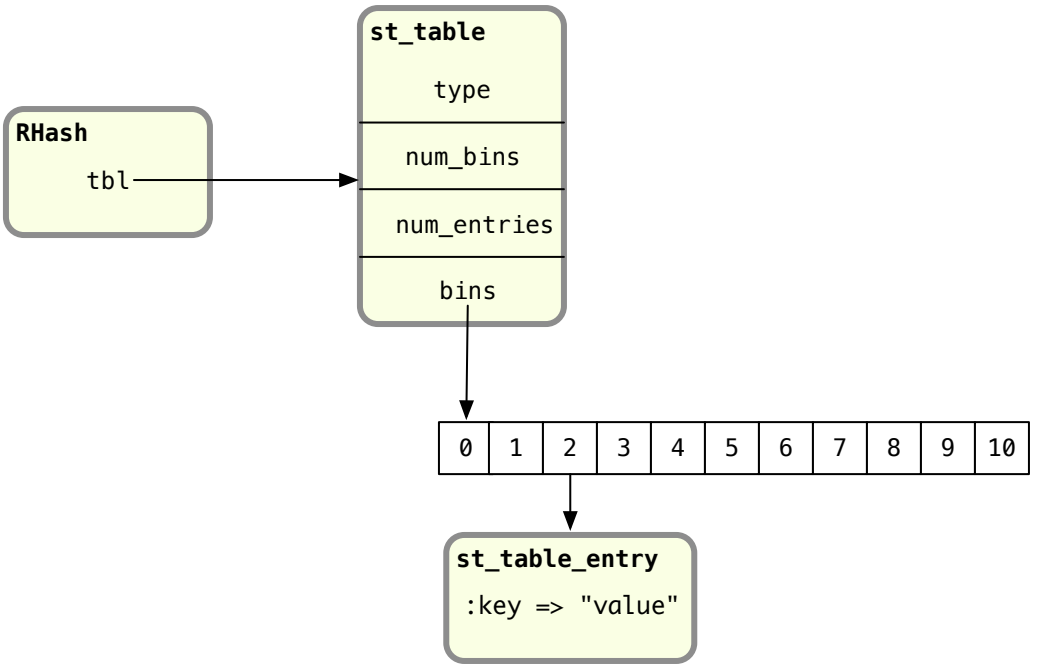
On the left is the `RHash` structure; this is short for "Ruby Hash." On the right, I show the hash table used by this hash, represented by the `st_table` structure. This C structure contains the basic information about the hash table, such as the number of entries saved in the table, the number of bins and a pointer to the bins. Each `RHash` structure contains a pointer to a corresponding `st_table` structure. Finally, I show some empty

bins on the lower right. Ruby 1.8 and Ruby 1.9 initially create 11 bins for a new, empty hash.

The best way to understand how a hash table works is by stepping through an example. Let's suppose I add a new key/value to a hash called `my_hash`:

```
my_hash[:key] = "value"
```

While executing this line of code, Ruby will create a new structure called an `st_table_entry` and will save it into the hash table for `my_hash`:



Here you can see Ruby saved the new key/value pair under the third bucket, #2. Ruby did this by taking the given key, the symbol `:key` in this example, and passing it to an internal hash function that returns a pseudorandom integer:

```
some_value = internal_hash_function(:key)
```

Next, Ruby takes the hash value, `some_value` in this example, and calculates the modulus by the number of bins… i.e. the remainder after dividing by the number of bins:

```
some_value % 11 = 2
```

In this diagram I imagine that the actual hash value for `:key` divided by 11 leaves a remainder of 2. Later in this chapter I'll explore the hash functions that Ruby actually uses in more detail.

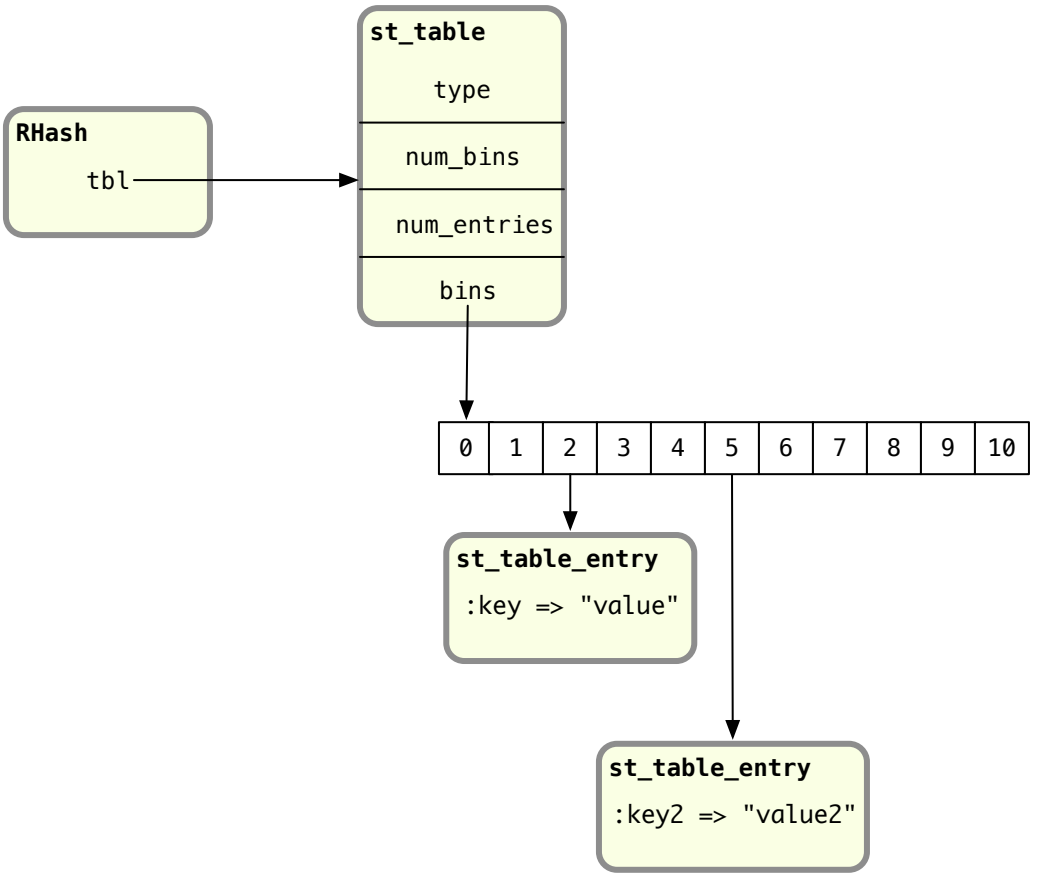
Now let's add a second element to the hash:

```
my_hash[:key2] = "value2"
```

And this time let's imagine that the hash value of `:key2` divided by 11 yields a remainder of 5:

```
internal_hash_function(:key2) % 11 = 5
```

Now you can see Ruby places a second `st_table_entry` structure under bin #5, the sixth bin:



The benefit of using a hash table comes later, when you ask Ruby to retrieve the value for a given key:

```
p my_hash[:key]
=> "value"
```

If Ruby had saved all of the keys and values in an array or linked list, then it would have to iterate over all the elements in that array or list, looking for `:key`. This might take a very long time, depending on how many elements there were. But using a hash table Ruby can jump straight to the key it needs to find by recalculating the hash value for that key. It simply calls the hash function again:

```
some_value = internal_hash_function(:key)
```

…redivides the hash value by the number of bins and obtaining the remainder, the modulus:

```
some_value % 11 = 2
```

…and now Ruby knows to look in bin #2 for the entry with a key of :key. In a similar way, Ruby can later find the value for `:key2` by repeating the same hash calculation:

```
internal_hash_function(:key2) % 11 = 5
```

Believe it or not, the C library used by Ruby to implement hash tables was originally written back in the 1980's by Peter Moore from the University of California at Berkeley, and later modified by the Ruby core team. You can find Peter Moore's hash table code in the C code files st.c and include/ruby/st.h. All of the function and structure names use the naming convention `st_` in Peter's hash table code.

Meanwhile, the definition of the `RHash` structure that represents every Ruby Hash object can be found in the include/ruby/ruby.h file. Along with `RHash`, here you'll find all of the other primary object structures used in the Ruby source code: `RString`, `RArray`, `RValue`, etc.

Experiment 4-1: Retrieving a value from hashes of varying sizes

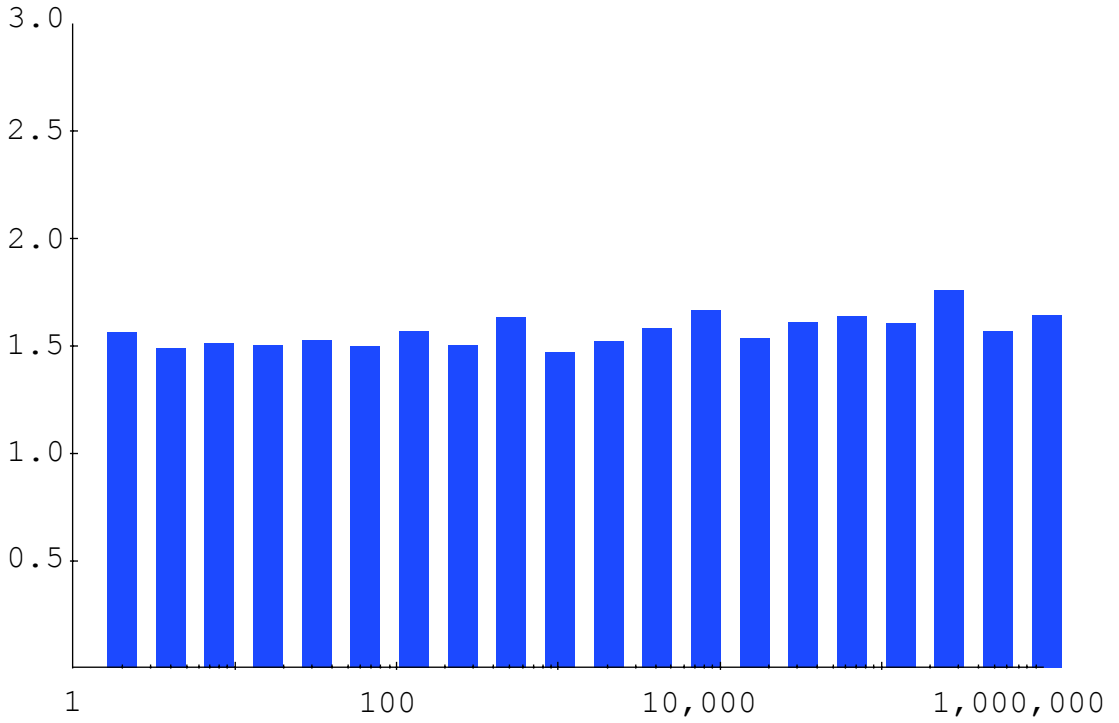
My first experiment will create hashes of wildly different sizes, from 1 element to 1 million elements and then measure how long it takes to find and return a value from each of these hashes. First, I create hashes of different sizes, based on powers of two, by running this code for different values of `exponent`:

```
size = 2**exponent
hash = {}
(1..size).each do |n|
  index = rand
  hash[index] = rand
end
```

Here both the keys and values are random floating values. Then I measure how long it takes to find one of the keys, the `target_key` 10,000 times using the benchmark library:

```
Benchmark.bm do |bench|
bench.report("retrieving an element
              from a hash with #{size} elements 10000 times") do
    10000.times do
      val = hash[target_key]
    end
  end
end
```

By using a hash table internally, Ruby is able to find and return value from a hash containing over a million elements just as fast as it takes to return one from a small hash:



Time to retrieve 10,000 values (ms) vs. hash size

Clearly the hash function Ruby uses is very fast, and once Ruby identifies the bin containing the target key, it is able to very quickly find the corresponding value and return it. What's remarkable about this is that the values in this chart are more or less flat.

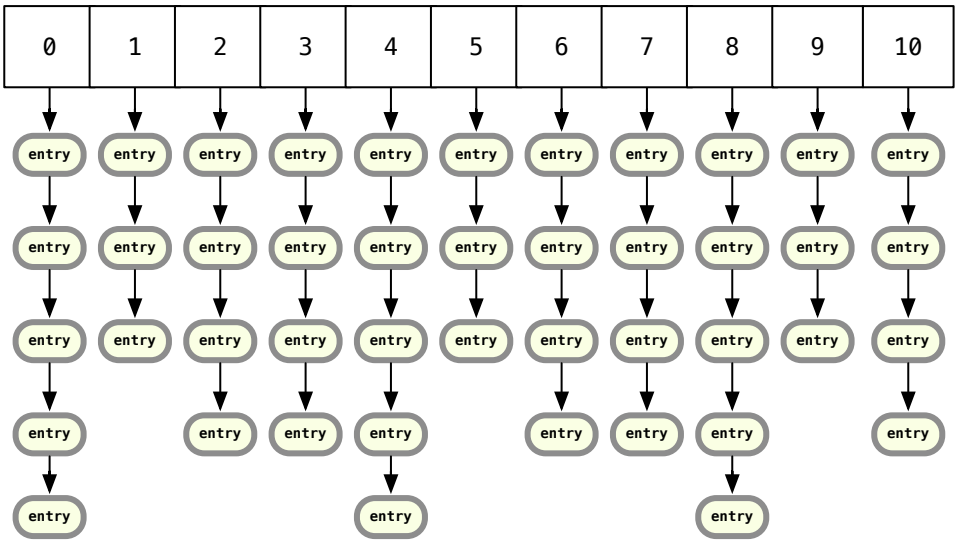
How hash tables expand to accommodate more values

You might be thinking ahead at this point by asking yourself: "If there are millions of `st_table_entry` structures, why does distributing them among 11 bins help Ruby search quickly?" Even if the hash function is fast, and even if Ruby distributes the values evenly among the 11 bins in the hash table, Ruby will still have to search among almost 100,000 elements in each bin to find the target key if there are a million elements overall.

Something else must be going on here. It seems to me that Ruby must add more bins to the hash table as more and more elements are added. Let's take another look at how Ruby's internal hash table code works. Continuing with the example from above, suppose I keep adding more and more elements to my hash:

```
my_hash[:key3] = "value3"
my_hash[:key4] = "value4"
my_hash[:key5] = "value5"
my_hash[:key6] = "value6"
...
```

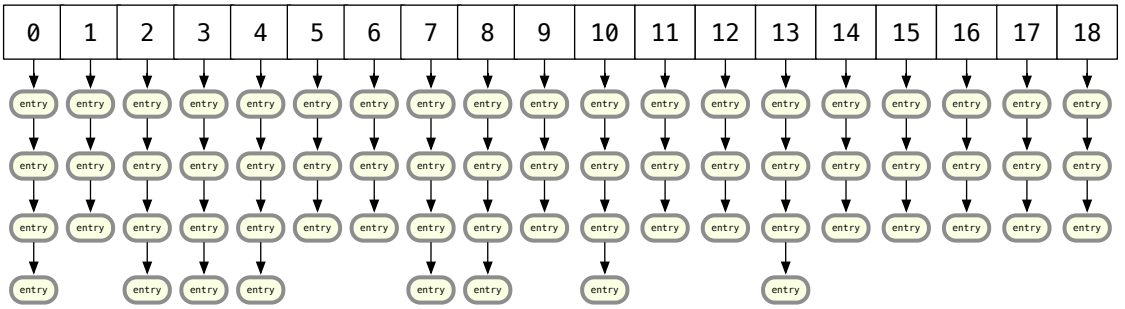
As I add more and more elements, Ruby will continue to create more `st_table_entry` structures and add them to different bins. The additional bins depend on the modulus of the hash value for each key.



Ruby uses a linked list to keep track of the entries in each bin: each `st_table_entry` structure contains a pointer to the next entry in the same bin. As you add more entries to the hash, the linked list for each bin gets longer and longer.

To keep these linked lists from getting out of control, Ruby measures something called the "density" or the average number of entries per bin. In my diagram above, you can see that the average number of entries per bin has increased to about 4. What this means is that the hash value modulus 11 has started to return repeated values for different keys and hash values. Therefore, when searching for a target key, Ruby might have to iterate through a small list, after calculating the hash value and finding which bin contains the desired entry.

Once the density exceeds 5, a constant value in the MRI C source code, Ruby will allocate more bins and then "rehash", or redistribute, the existing entries among the new bin set. For example, if I keep adding more key/value pairs, after a while Ruby will discard the array of 11 bins, allocate an array of 19 bins, and then rehash all the existing entries:



Now in this diagram the bin density has dropped to about 3.

By monitoring the bin density in this way, Ruby is able to guarantee that the linked lists remain short, and that retrieving a hash element is always fast. After calculating the hash value Ruby just needs to step through 1 or 2 elements to find the target key.

You can find the `rehash` function - the code that loops through the `st_table_entry` structures and recalculates which bin to put the entry into - in the st.c source file. This snippet is from Ruby 1.8.7:

```
static void
rehash(table)
    register st_table *table;
{
  register st_table_entry *ptr, *next, **new_bins;
  int i, old_num_bins = table->num_bins, new_num_bins;
  unsigned int hash_val;
  new_num_bins = new_size(old_num_bins+1);
  new_bins = (st_table_entry**)Calloc(new_num_bins,
                                      sizeof(st_table_entry*));
  for(i = 0; i < old_num_bins; i++) {
    ptr = table->bins[i];
    while (ptr != 0) {
      next = ptr->next;
      hash_val = ptr->hash % new_num_bins;
      ptr->next = new_bins[hash_val];
      new_bins[hash_val] = ptr;
      ptr = next;
    }
  }
  free(table->bins);
```

```
  table->num_bins = new_num_bins;
  table->bins = new_bins;
}
```

The `new_size` method call here returns the new bin count, for example 19. Once Ruby has the new bin count, it allocates the new bins and then iterates over all the existing `st_table_entry` structures (all the key/value pairs in the hash). For each `st_table_entry` Ruby recalculates the bin position using the same modulus formula: `hash_val = ptr->hash % new_num_bins`. Then it saves each entry in the linked list for that new bin. Finally Ruby updates the `st_table` structure and frees the old bins.

In Ruby 1.9 and Ruby 2.0 the rehash function is implemented somewhat differently, but works essentially the same way.

Experiment 4-2: Inserting one new element into hashes of varying sizes

One way to test whether this rehashing or redistribution of entries really occurs is to measure the amount of time Ruby takes to save one new element into an existing hash of different sizes. As I add more and more elements to the same hash, at some point I should see some evidence that Ruby is taking extra time to rehash the elements.

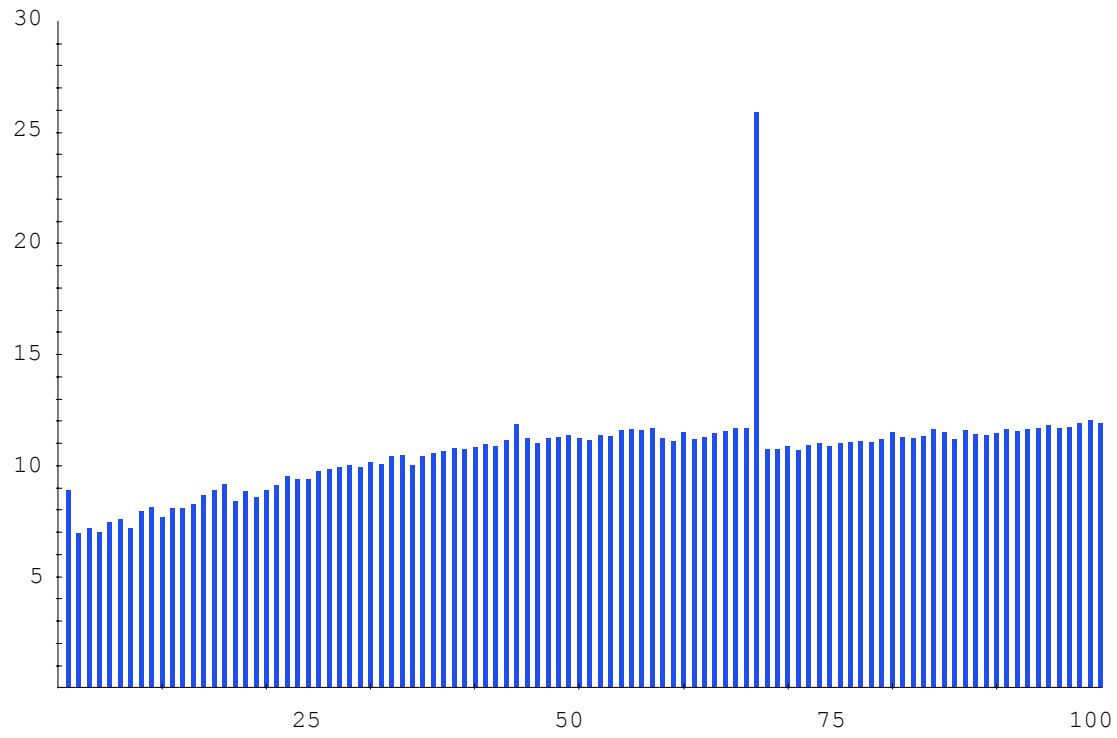
I'll do this by creating 10,000 hashes, all of the same size, indicated by the variable `size`:

```
hashes = []
10000.times do
  hash = {}
  (1..size).each do
    hash[rand] = rand
  end
  hashes << hash
end
```

Once these are all setup, I can measure how long it takes to add one more element to each hash - element number `size+1`:

```
Benchmark.bm do |bench|
  bench.report("adding element number #{size+1}") do
    10000.times do |n|
      hashes[n][size] = rand
    end
  end
end
```

What I found was surprising! Here's the data for Ruby 1.8:



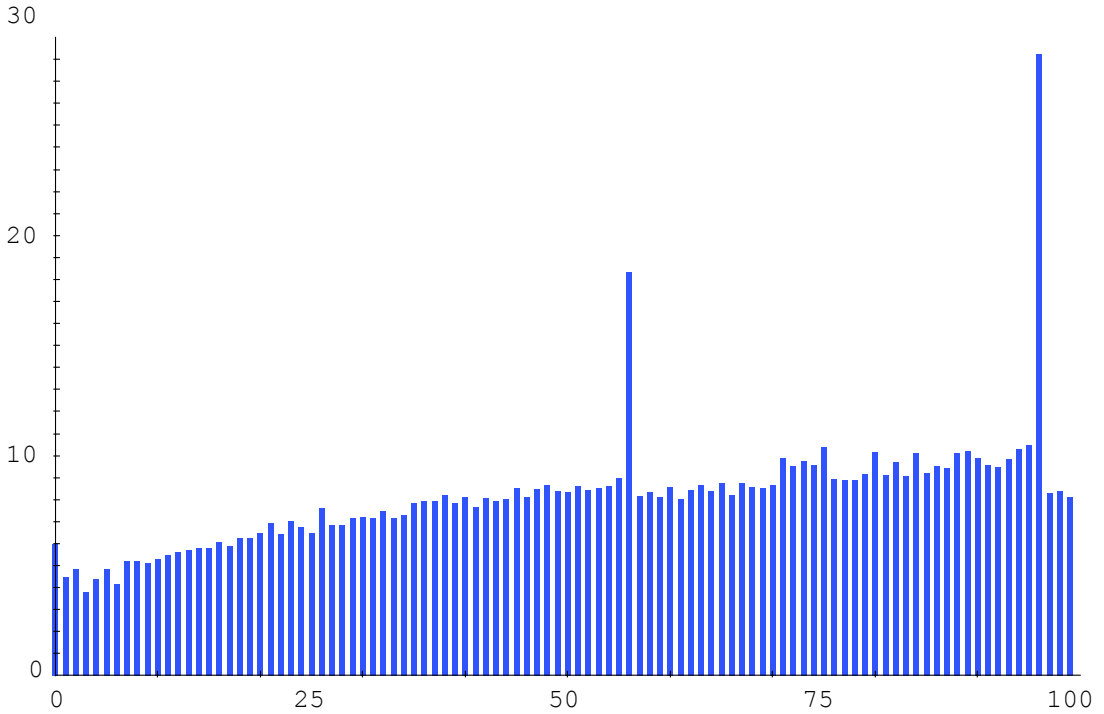
Time to add a new key/value pair (ms) vs. hash size

Interpreting these data values from left to right:

- It takes about 9ms to insert the first element into an empty hash (10000 times).
- It takes about 7ms to insert the second element into a hash containing one value (10000 times).
- As the hash size increases from 2, 3, up to about 60 or 65 the amount of time required to insert a new element slowly increases.
- We see it takes around 11ms or 12ms to insert each new key/value pair into a hash that contains 64, 65 or 66 elements (10000 times).
- Later, we see a huge spike! Inserting the 67th key/value pair takes over twice as much time: about 26ms instead of 11ms for 10000 hashes!
- After inserting the 67th element, the time required to insert additional elements drops to about 10ms or 11ms, and then slowly increases again from there.

What's going on here? Well, the extra time required to insert that 67th key/value pair is spent by Ruby reallocating the bin array from 11 bins to 19 bins, and then reassigning the `st_table_entry` structures to the new bin array.

Here's the same graph for Ruby 1.9 - you can see this time the bin density threshold is different. Instead of taking extra time to reallocate the elements into bins on the 67th insert, Ruby 1.9 does it when the 57th element is inserted. Later you can see Ruby 1.9 performs another reallocation after the 97th element is inserted.



Time to add a new key/value pair (ms) for 10,000 hashes vs. hash size

If you're wondering where these magic numbers come from, 57, 97, etc., then take a look at the top of the "st.c" code file for your version of Ruby. You should find a list of prime numbers like this:

```
/*
Table of prime numbers 2^n+a, 2<=n<=30.
*/
```

```
static const unsigned int primes[] = {
  8 + 3,
  16 + 3,
  32 + 5,
  64 + 3,
  128 + 3,
  256 + 27,
  512 + 9,
...
```

This C array lists some prime numbers that occur near powers of two. Peter Moore's hash table code uses this table to decide how many bins to use in the hash table. For example, the first prime number in the list above is 11, which is why Ruby hash tables start with 11 bins. Later as the number of elements increases, the number of bins is increased to 19, and later still to 37, etc.

Ruby always sets the number of hash table bins to be a prime number to make it more likely that the hash values will be evenly distributed among the bins, after calculating the modulus - after dividing by the prime number and using the remainder. Mathematically, prime numbers help here since they are less likely to share a common factor with the hash value integers, in case a poor hash function often returned values that were not entirely random. If the hash values and bin counts shared a factor, or if the hash values were a multiple of the bin count, then the modulus might always be the same. This leads to the table entries being unevenly distributed among the bins.

Elsewhere in the st.c file, you should be able to find this C constant:

```
#define ST_DEFAULT_MAX_DENSITY 5
```

… which defines the maximum allowed density, or average number of elements per bin. Finally, you should also be able to find the code that decides when to perform a bin reallocation by searching for where that ST_DEFAULT_MAX_DENSITY constant is used in st.c. For Ruby 1.8 you'll find this code:

```
if (table->num_entries/(table->num_bins) > ST_DEFAULT_MAX_DENSITY) {
  rehash(table);
```

So Ruby 1.8 rehashes from 11 to 19 bins when the value `num_entries/11` is greater than 5… i.e. when it equals 66. Since this check is performed before a new element is added, the condition becomes true when you add the 67th element, because `num_entries` would be 66 then.

For Ruby 1.9 and Ruby 2.0 you'll find this code instead:

```
if ((table)->num_entries >
    ST_DEFAULT_MAX_DENSITY * (table)->num_bins) {
  rehash(table);
```

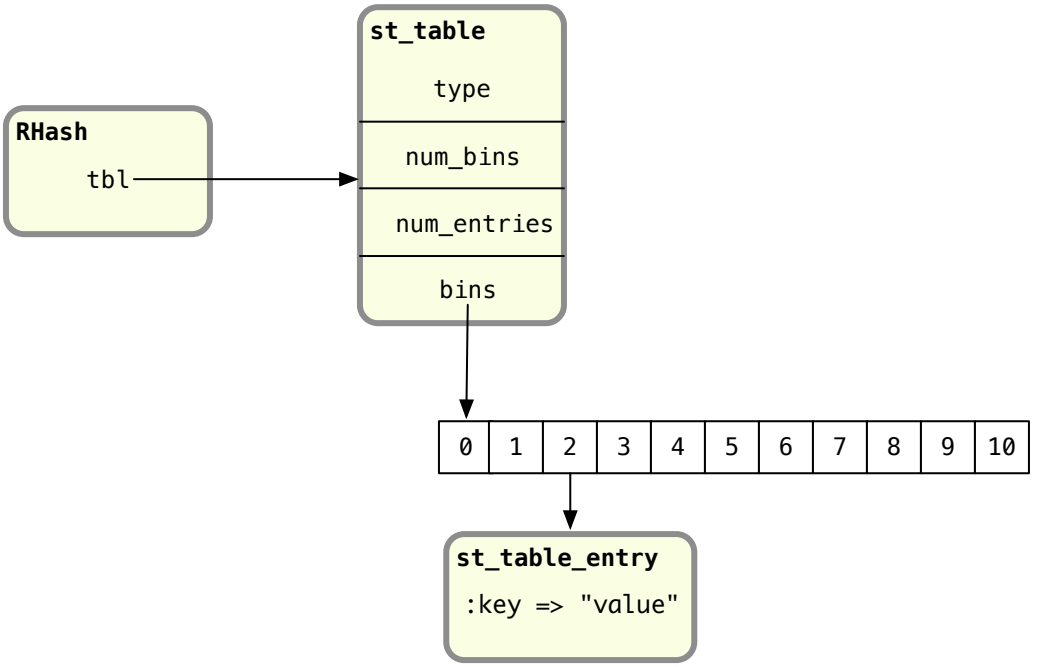
You can see Ruby 1.9 rehashes for the first time when `num_entries` is greater than 5*11, or when you insert the 57th element.

How Ruby implements hash functions

Hash functions allow Ruby to find which bin contains a given key and value.

Now let's take a closer look at the actual hash function Ruby uses to assign keys and values to bins in hash tables. If you think about it, this function is central to the way the Hash object is implemented – if this function works well then Ruby hashes will be fast, but a poor hash function would in theory cause severe performance problems. And not only that, as I mentioned above, Ruby uses hash tables internally to store its own information, in addition to the data values you save in hash objects. Clearly having a good hash function is very important!

First let's review again how Ruby uses hash values. Remember that when you save a new element – a new key/value pair – in a hash, Ruby assigns it to a bin inside the internal hash table used by that hash object:



Again, the way this works is that Ruby calculates the modulus of the key's hash value by the number of bins:

```
bin_index = internal_hash_function(key) % bin_count
```

Or in this example:

```
2 = hash(:key) % 11
```

The reason this works well for Ruby is that Ruby's hash values are more or less random integers for any given input data. You can get a feel for how Ruby's hash function works by calling the `hash` method for any object like this:

```
$ irb
> "abc".hash
 => 3277525029751053763
> "abd".hash
 => 234577060685640459
> 1.hash
 => -3466223919964109258
```

```
> 2.hash
 => -2297524640777648528
```

Here even similar values have very different hash values. Note that if I call `hash` again I always get the same integer value for the same input data:

```
> "abc".hash
 => 3277525029751053763
> "abd".hash
 => 234577060685640459
```

Here's how Ruby's hash function actually works for most Ruby objects:

- When you call `hash` Ruby finds the default implementation in the `Object` class. You, of course, are free to override this if you really want to.
- The C code used by the Object class's implementation of the hash method gets the C pointer value for the target object – i.e. the actual memory address of that object's RValue structure. This is essentially a unique id for that object.
- Ruby then passes it through a complex C function – the hash function – that mixes up and scrambles the bits in the value, producing a pseudo-random integer in a repeatable way.

For string and arrays it works differently. In this case, Ruby actually iterates through all of the characters in the string or elements in the array and calculates a cumulative hash value. This guarantees that the hash value will always be the same for any instance of a string or array, and will generally change if any of the values in that string or array change.

Finally, integers and symbols are another special case. For them Ruby just passes their values right to the hash function.

> Ruby 1.9 and 2.0 actually use something called the "MurmurHash" hash function, which was invented by Austin Appleby in 2008. The name "Murmur" comes from the machine language operations used in the algorithm: "multiply" and "rotate." If you're interested in the details of how the Murmur algorithm actually works, you can find the C code for it in the st.c Ruby source code file. Or you can read Austin's web page on Murmur: http://sites.google.com/site/murmurhash/.

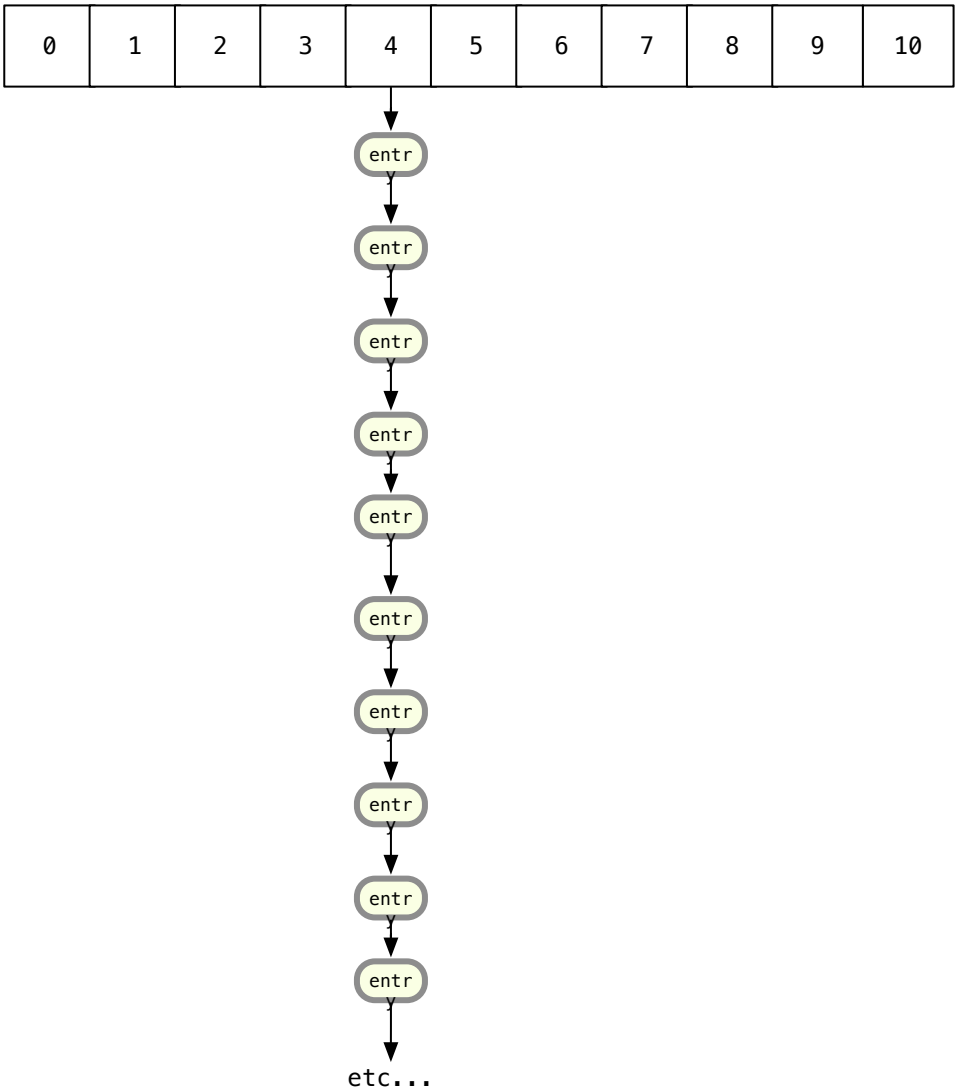
Also, Ruby 1.9 and Ruby 2.0 initialize MurmurHash using a random seed value which is reinitialized each time you restart Ruby. This means that if you stop and restart Ruby you'll get different hash values for the same input data. It also means if you try this yourself you'll get different values than I did above. However, the hash values will always be the same within the same Ruby process.

Experiment 4-3: Using objects as keys in a hash

Since hash values are pseudo-random numbers, once Ruby divides them by the bin count, e.g. 11, the remainder values left over (the modulus values) will be a random number between 0 and 10. This means that the `st_table_entry` structures will be evenly distributed over the available bins as they are saved in the hash table. Evenly distributing the entries ensures that Ruby will be able to quickly search for and find any given key. The number of entries per bin will always be small.

But imagine if Ruby's hash function didn't return random integers - imagine if instead it returned the same integer for every input data value. What would happen?

In that case, every time you added any key/value to a hash it would always be assigned to the same bin. Then Ruby would end up with all of the entries in a single, long list under that one bin, and with no entries in any other bin:



Now when you tried to retrieve some value from this hash, Ruby would have to look through this long list, one element at a time, trying to find the requested key. In this scenario loading a value from a Ruby hash would be very, very slow.

```
size = 2**exponent
hash = {}
(1..size).each do
```

```
  index = rand
  hash[index] = rand
end
```

Now I'm going to prove this is the case – and illustrate just how important Ruby's hash function really is – by using objects with a poor hash function as keys in a hash. Let's repeat Experiment 1 and create many hashes that have different numbers of elements, from 1 to a million:

```
size = 2**exponent
hash = {}
(1..size).each do
  index = rand
  hash[index] = rand
end
```

But instead of calling `rand` to calculate random key values, this time I'll create a new, custom object class called `KeyObject` and use instances of that class as my key values:

```
class KeyObject
end


size = 2**exponent
hash = {}
(1..size).each do
  index = KeyObject.new
  hash[index] = rand
end
```

This works essentially the same way as Experiment 1 did, except that Ruby will have to calculate the hash value for each of these `KeyObject` objects instead of the random floating point values I used earlier.

After re-running the test with this KeyObject class, I'll then proceed to change the KeyObject class and override the `hash` method, like this:

```
class KeyObject
  def hash
    4
```

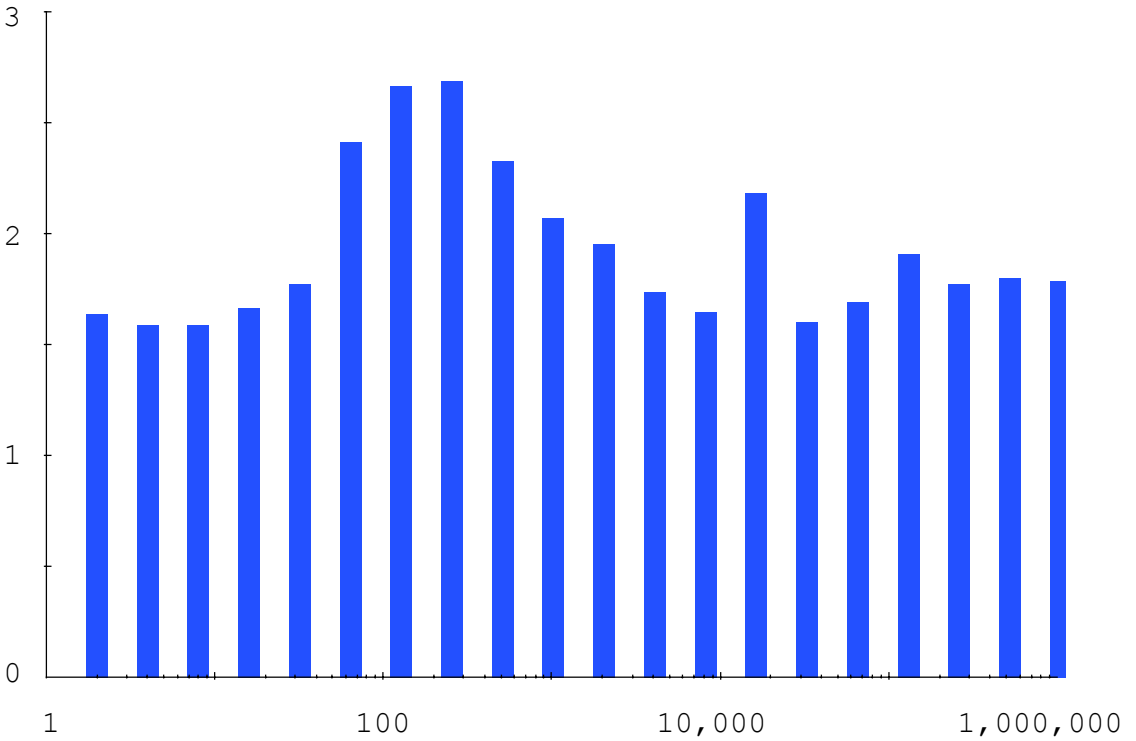
```
  end
end
```

I've purposefully written a very poor hash function – instead of returning a pseudo-random integer, this hash function always returns the integer 4, regardless of which `KeyObject` object instance you call it on. Now Ruby will always get 4 when it calculates the hash value. It will have to assign all of the hash elements to bin #4 in the internal hash table, like in the diagram above. Let's see what happens….

Running the test with an empty `KeyObject` class:

```
class KeyObject
end
```

…I get results similar to Experiment 1:



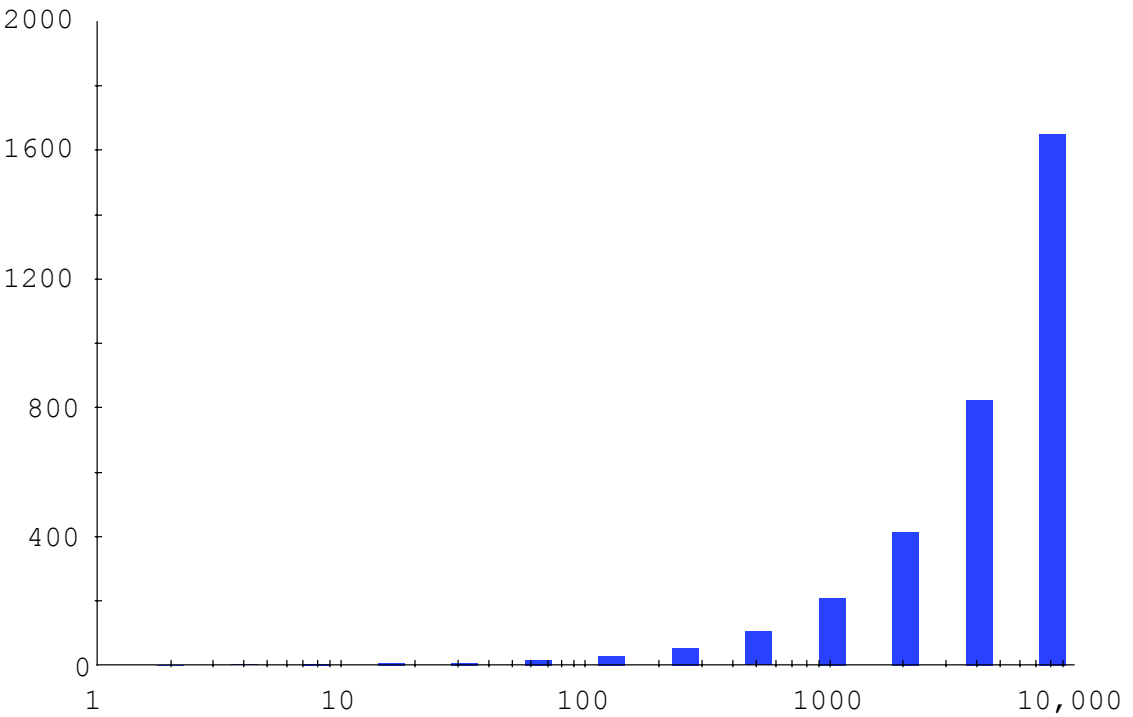
Time to retrieve 10000 values (ms) vs. hash size

Using Ruby 1.9 I again see that Ruby takes about 1.5ms to 2ms to retrieve 10,000 elements from a hash, this time using instances of the `KeyObject` class as the keys.

Now let's run the same code, but this time with the poor hash function in `KeyObject`:

```
class KeyObject
  def hash
    4
  end
end
```

Here are the results:



Time to retrieve 10000 values (ms) vs. hash size

Wow – very different! Pay close attention to the scale of the graph. On the y-axis I show milliseconds and on the x-axis again the number of elements in the hash, shown on a logarithmic scale. But this time, notice that I have 1000s of milliseconds, or actual seconds, on the y-axis! With 1 or a small number of elements, I can retrieve the 10,000

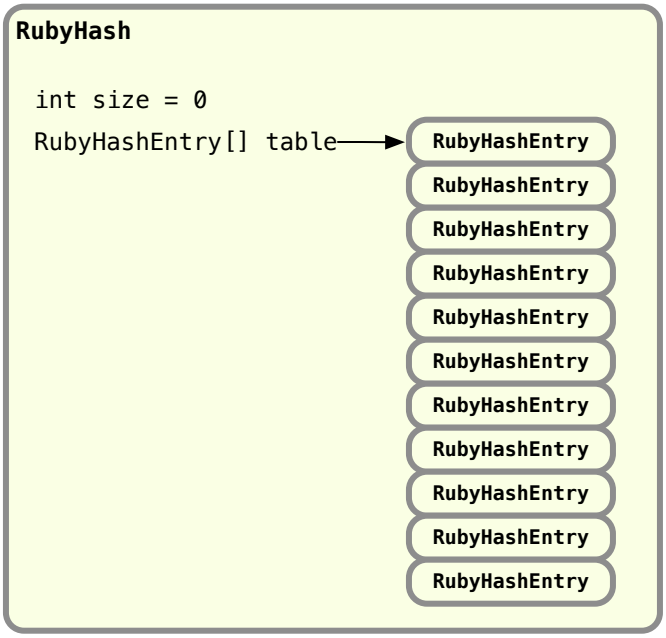
values very quickly – so quickly that the time is too small to appear on this graph. In fact it takes about the same 1.5ms time.

When the number of elements increases past 100 and especially 1000, the time required to load the 10,000 values increases linearly with the hash size. For a hash containing about 10,000 elements it takes over 1.6 full seconds to load the 10,000 values. If I continue the test with larger hashes it would take minutes or even hours to load the values.

Again what's happening here is that all of the hash elements are saved into the same bin, forcing Ruby to search through the list one key at a time.

Hash tables in JRuby

It turns out JRuby implements hashes more or less the same way MRI Ruby does. Of course, the JRuby source code is written in Java and not C, but the JRuby team chose to use the same underlying hash table algorithm that MRI uses. Since Java is an object oriented language, unlike C, JRuby is able to use actual Java objects to represent the hash table and hash table entries, instead of C structures. Here's what a hash table looks like internally inside of a JRuby process:



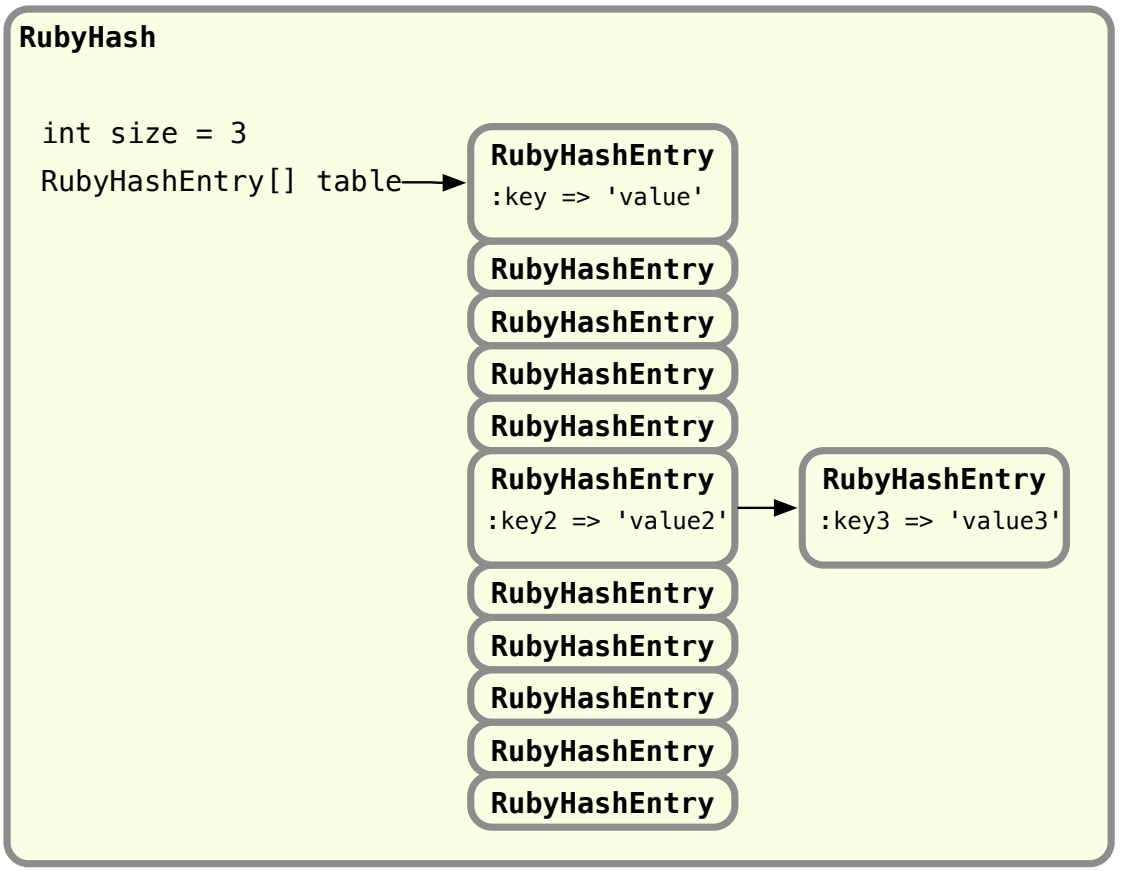
Instead of the C `RHash` and `st_table` structures, we have a Java object which is an instance of `RubyHash`. And instead of the bin array and `st_table_entry` structures we have an array of Java objects of type `RubyHashEntry`. The `RubyHash` object contains an instance variable called `size` which keeps track of the number of elements in the hash, and another instance variable called `table`, which is the `RubyHashEntry` array.

JRuby allocates 11 empty `RubyHashEntry` objects or hash table bins when you create a new hash. As you insert elements into the hash, JRuby fills in these objects them with keys and values. Inserting and retrieving elements works the same was as in MRI:

JRuby uses the same formula to divide the hash value of the key by the bin count, and uses the modulus to find the proper bin:

```
bin_index = internal_hash_function(key) % bin_count
```

As you add more and more elements to the hash, JRuby forms a linked list of `RubyHashEntry` objects as necessary when two keys fall into the same bin - just like MRI:



JRuby also tracks the density of entries, the average number of `RubyHashEntry` objects per bin, and allocates a larger table of `RubyHashEntry` objects as necessary to rehash the entries.

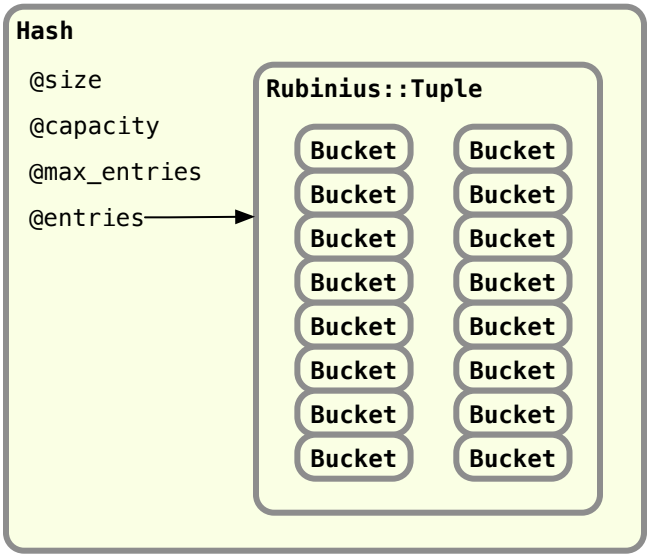
If you're interested, you can find the Java code JRuby uses to implement hashes in the src/org/jruby/RubyHash.java source code file. I found it easier to understand than the original C code from MRI, mostly because in general Java is a bit more readable and easier to understand than C is, and because it's object oriented. The JRuby team was able to separate the hash code into different Java classes, primarily `RubyHash` and `RubyHashEntry`.

The JRuby team even used the same identifier names as MRI in some cases; for example you'll find the same `ST_DEFAULT_MAX_DENSITY` value of 5, and JRuby uses the same table of prime numbers that MRI does: 11, 19, 37, etc., that fall near powers of two. This means that JRuby will show the same performance pattern MRI does for reallocating bins and redistributing the entries.

Hash tables in Rubinius

At a high level, Rubinius uses the same hash table algorithm as MRI and JRuby - but using Ruby instead of C or Java. This means the Rubinius source code is about 10 times easier to understand than either the MRI or JRuby code, and is a great way to learn more about hash tables if you're interested in getting your hands dirty without learning C or Java.

Here's how hashes look inside of Rubinius:



Since this is just plain Ruby, in Rubinius your Ruby objects are actually implemented with a real Ruby class called `Hash`. You'll see it has a few integer attributes, such as `@size`, `@capacity` and `@max_entries`, and also an instance variable called `@entries` which is the bin array that actually contains the hash data. Rubinius implements the bin array using a Ruby class called `Rubinius::Tuple`, a simple storage class similar to an array. Rubinius saves each hash element inside a Ruby object called `Bucket`, saved inside of the `@entries Rubinius::Tuple` array.

One difference you'll see in the Rubinius hash table implementation is that it uses simple powers of two to decide how many hash bins to create, instead of prime numbers. Initially Rubinius uses 16 Bucket objects. Whenever Rubinius needs to allocate more bins, it just doubles the size of the bin array, `@entries`, in the code

above. While theoretically this is less ideal than using prime numbers, it simplifies the code substantially and also allows Rubinius to use bitwise arithmetic to calculate the bin index, instead of having to divide and take the remainder/modulus.

You'll find the Rubinius hash implementation in source code files called kernel/common/ hash18.rb and kernel/common/hash19.rb - Rubinius has entirely different implementations of hashes depending on whether you start in Ruby 1.8 or Ruby 1.9 compatibility mode. Here's a snippet from hash18.rb, showing how Rubinius finds a value given a key:

```
def [](key)
  if item = find_item(key)
    item.value
  else
    default key
  end
end
...etc...
# Searches for an item matching +key+. Returns the item
# if found. Otherwise returns +nil+.
def find_item(key)
  key_hash = key.hash
  item = @entries[key_index(key_hash)]
  while item
    if item.match? key, key_hash
      return item
    end
    item = item.link
  end
end
...etc...
# Calculates the +@entries+ slot given a key_hash value.
def key_index(key_hash)
  key_hash & @mask
end
```

You can see the key_index method uses bitwise arithmetic to calculate the bin index, since the bin count will always be a power of two for Rubinius, and not a prime number.

Chapter 5
How Ruby Borrowed a Decades Old Idea From Lisp

The IBM 704, above, was the first computer to run Lisp in the early 1960s.

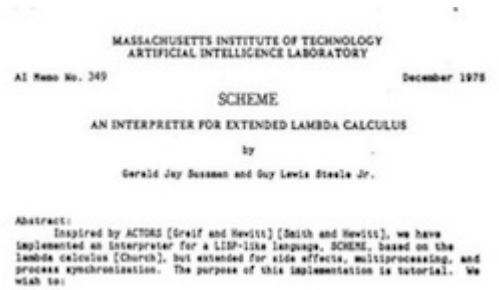
Blocks are one of the most commonly used and powerful features of Ruby – as you probably know, they allow you to pass a code snippet to iterators such as `each`, `detect` or `inject`. In Ruby you can also write your own custom iterators or functions that call blocks for other reasons using the `yield` keyword. Ruby code containing blocks is often more succinct, elegant and expressive than the equivalent code would appear in older languages such as C.

However, don't jump to the conclusion that blocks are a new idea! In fact, blocks are not new to Ruby at all; the computer science concept behind blocks, called "closures," was first invented by Peter J. Landin in 1964, a few years after the original version of Lisp was created by John McCarthy in 1958. Closures were later adopted by Lisp – or more precisely a dialect of Lisp called Scheme, invented by Gerald Sussman and Guy Steele in 1975. Sussman and Steele's use of closures in Scheme brought the idea to many programmers for the first time starting in the 1970s.

But what does "closure" actually mean? In other words, exactly what are Ruby blocks? Are they as simple as they appear? Are they just the snippet of Ruby code that appears between the `do` and `end` keywords? Or is there more to Ruby blocks than meets the eye? In this chapter I'll review how Ruby implements blocks internally, and show how they meet the definition of "closure" used by Sussman and Steele back in 1975. I'll also show how blocks, lambdas, procs and bindings are all different ways of looking at closures, and how these objects are related to Ruby's metaprogramming API.

Chapter 5 Roadmap

Blocks: Closures in Ruby



MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

AI Memo No. 349 December 1975

SCHEME

AN INTERPRETER FOR EXTENDED LAMBDA CALCULUS

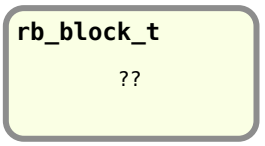
by

Gerald Jay Sussman and Guy Lewis Steele Jr.

Abstract:
Inspired by ACTORS [Greif and Hewitt] [Smith and Hewitt], we have implemented an interpreter for a LISP-like language, SCHEME, based on the lambda calculus [Church], but extended for side effects, multiprocessing, and process synchronization. The purpose of this implementation is tutorial. We wish to:

Sussman and Steele gave a useful definition of the term "closure" in this 1975 academic paper, one of the so-called "Lambda Papers."

Internally Ruby represents each block using a C structure called `rb_block_t`:

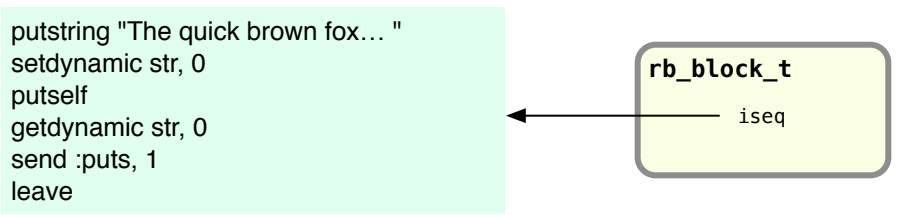


Exactly what are blocks? One way to answer this question would be to take a look at the values Ruby stores inside this structure. Just as we did in Chapter 3 with the `RClass` structure, let's deduce what the contents of the `rb_block_t` structure must be based on what we know blocks can do in our Ruby code.

Starting with the most obvious attribute of blocks, we know each block must consist of a piece of Ruby code, or internally a set of compiled YARV byte code instructions. For example, if I call a method and pass a block as a parameter:

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

…it's clear that when executing the `10.times` call, Ruby needs to know what code to iterate over. Therefore, the `rb_block_t` structure must contain a pointer to that code:



In this diagram, you can see a value called `iseq` which is a pointer to the YARV instructions for the Ruby code in my block.

Another obvious but often overlooked behavior of blocks is that they can access variables in the surrounding or parent Ruby scope. For example:

```
str = "The quick brown fox"
10.times do
  str2 = "jumps over the lazy dog."
  puts "#{str} #{str2}"
end
```

Here the `puts` function call refers equally well to the `str` variable located inside the block and the `str2` variable from the surrounding code. We often take this for granted – obviously blocks can access values from the code surrounding them. This ability is one of the things that makes blocks useful.

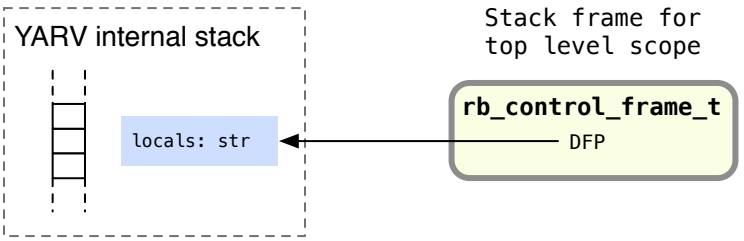
If you think about this for a moment, you'll realize blocks have in some sense a dual personality. On the one hand, they behave like separate functions: you can call them and pass them arguments just as you would with any function. On the other hand, they are part of the surrounding function or method. As I wrote the sample code above I didn't think of the block as a separate function – I thought of the block's code as just part of the simple, top level script that printed a string 10 times.

Stepping through how Ruby calls a block

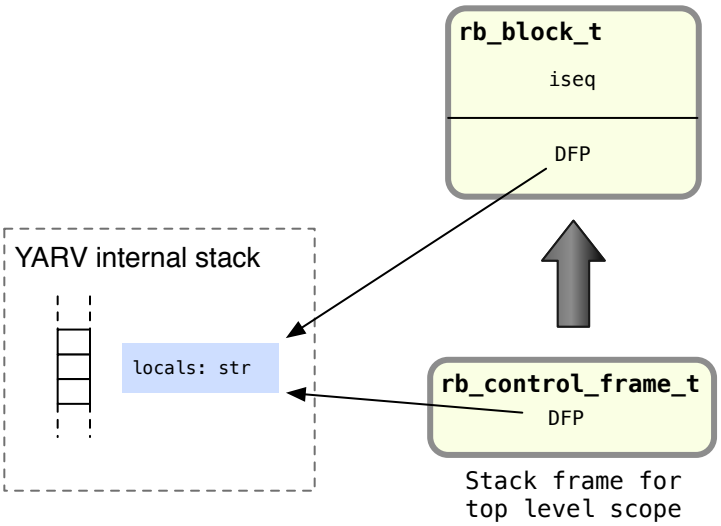
How does this work internally? Does Ruby internally implement blocks as separate functions? Or as part of the surrounding function? Let's step through the example above, slowly, and see what happens inside of Ruby when you call a block.

In this example when Ruby executes the first line of code, as I explained in Chapter 2, YARV will store the local variable `str` on its internal stack, and save its location in the `DFP` pointer located in the current `rb_control_frame_t` structure.[4]

4. If the outer code was located inside a function or method then the `DFP` would point to the stack frame as shown, but if the outer code was located in the top level scope of your Ruby program, then Ruby would use dynamic access to save the variable in the `TOPLEVEL_BINDING` environment instead – more on this in section 5.3. Regardless the `DFP` will always indicate the location of the `str` variable.

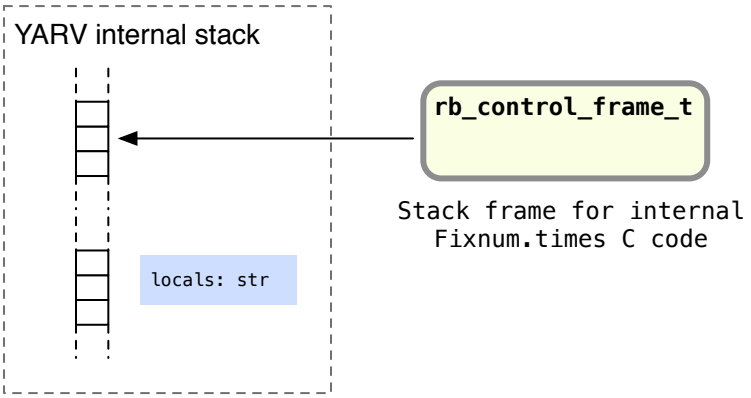


Next Ruby will come to the "`10.times do`" call. Before executing the actual iteration – before calling the `times` method – Ruby will create and initialize a new `rb_block_t` structure to represent the block. Ruby needs to create the block structure now, since the block is really just another argument to the `times` method:



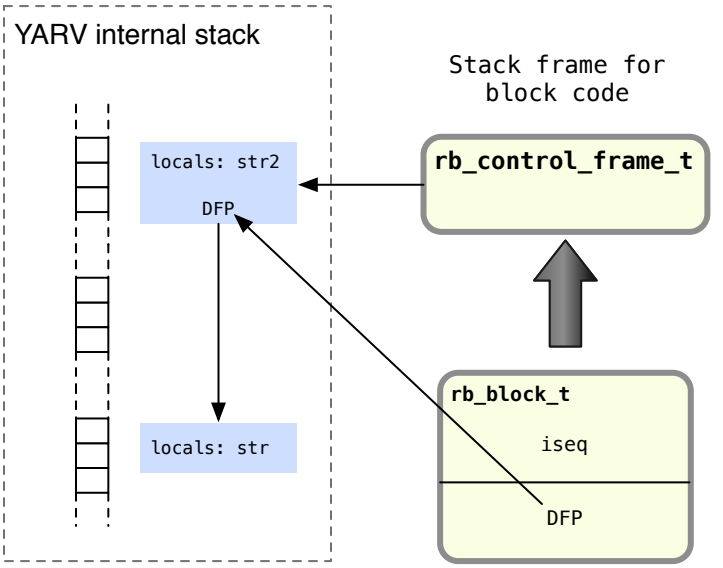
To do this, Ruby copies the current value of the `DFP`, the dynamic frame pointer, into the new block. In other words, Ruby saves away the location of the current stack frame in the new block.

Next Ruby will proceed to call the `times` method on the object `10`, an instance of the `Fixnum` class. While doing this, YARV will create a new frame on its internal stack. Now we have two stack frames: on the top is the new stack frame for the `Fixnum.times` method, and below is the original stack frame used by the top level function:



Ruby implements the `times` method internally using its own C code. It's a built in method, but Ruby implements it the same way you probably would in Ruby. Ruby starts to iterate over the numbers 0, 1, 2, etc., up to 9, and calls `yield`, calling the block once for each of these integers. Finally, the code that implements `yield` internally actually calls the block each time through the loop, pushing a third[5] frame onto the top of the stack for the code inside the block to use:

5. Ruby actually pushes an extra, internal stack frame whenever you call yield before actually calling the block, so strictly speaking there should be four stack frames in this diagram. I only show three for the sake of clarity.



Above, on the left, we now have three stack frames:

- On the top is the new stack frame for the block, containing the `str2` variable.
- In the middle is the stack frame used by the internal C code that implements the `Fixnum.times` method.
- And at the bottom is the original function's stack frame, containing the `str` variable from the outer scope.

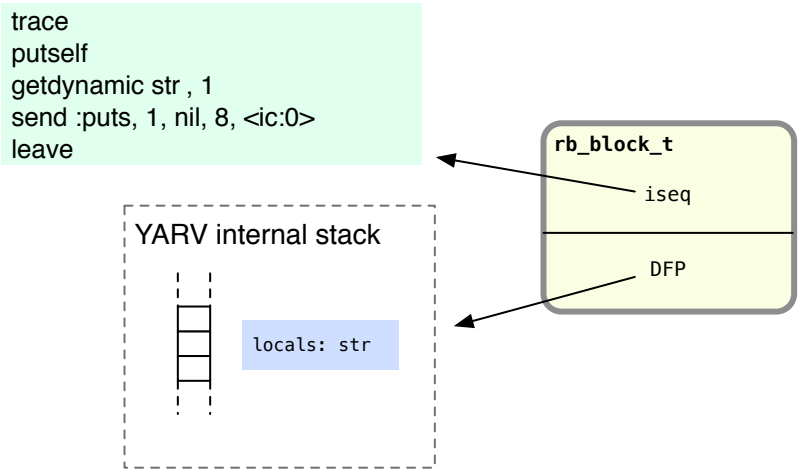
While creating the new stack frame at the top, Ruby's internal `yield` code copies the `DFP` from the block into the new stack frame. Now the code inside the block can access both its own local variables, via the `rb_control_frame` structure as usual, and indirectly the variables from the parent scope, via the `DFP` pointer using dynamic variable access, as I explained in Chapter 2. Specifically, this allows the `puts` statement to access the `str2` variable from the parent scope.

Borrowing an idea from 1975

To summarize, we have seen now that Ruby's `rb_block_t` structure contains two important values:

- a pointer to a snippet of YARV code instructions, and

- a pointer to a location on YARV's internal stack, the location that was at the top of the stack when the block was created:



At first glance, this seems like a very technical, unimportant detail. This is obviously a behavior we expect Ruby blocks to exhibit, and the `DFP` seems to be just another minor, uninteresting part of Ruby's internal implementation of blocks.

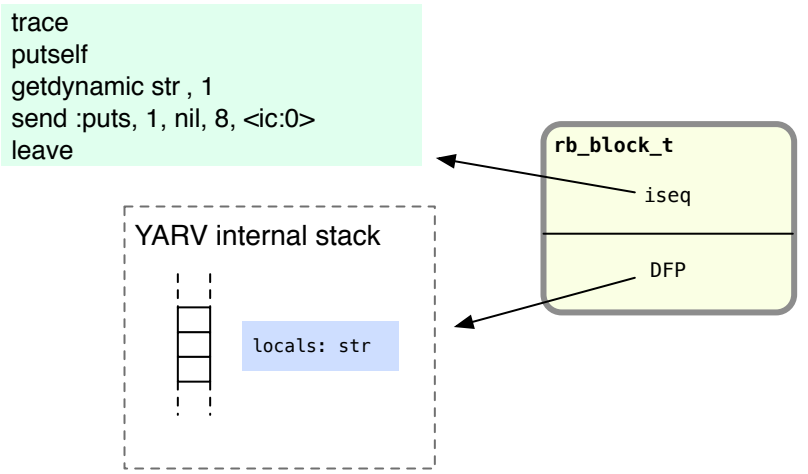
Or is it? I believe the `DFP` is actually a profound, important part of Ruby internals. The `DFP` is the basis for Ruby's implementation of "closures," a computer science concept invented long before Ruby was created in the 1990s. In fact, the Scheme programming language, a dialect of Lisp invented by Gerald Sussman and Guy Steele in 1975, was one of the first languages to formally implement closures – almost twenty years earlier! Here's how Sussman and Steele defined the term "closure" in their 1975 paper *Scheme: An Interpreter for Extended Lambda Calculus*:

> *In order to solve this problem we introduce the notion of a closure [11, 14] which is a data structure containing a lambda expression, and an environment to be used when that lambda expression is applied to arguments.*

Reading this again, a closure is defined to be the combination of:

- A "lambda expression," i.e. a function that takes a set of arguments, and
- An environment to be used when calling that lambda or function.

I'll have more context and information about "lambda expressions" and how Ruby's borrowed the "lambda" keyword from Lisp in section 5-2, but for now take another look at the internal `rb_block_t` structure:



Notice that this structure meets the definition of a closure Sussman and Steele wrote back in 1975:

- `iseq` is a pointer to a lambda expression – i.e. a function or code snippet, and
- `DFP` is a pointer to the environment to be used when calling that lambda or function – i.e. a pointer to the surrounding stack frame.

Following this train of thought, we can see that blocks are Ruby's implementation of closures. Ironically blocks, one of the features that in my opinion makes Ruby so elegant and natural to read – so modern and innovative – is based on research and work done at least 20 years before Ruby was ever invented!

In Ruby 1.9 and later you can find the actual definition of the `rb_block_t` structure in the vm_core.h file. Here it is:

```
typedef struct rb_block_struct {
    VALUE self;
    VALUE *lfp;
    VALUE *dfp;
    rb_iseq_t *iseq;
```

```
    VALUE proc;
} rb_block_t;
```

You can see the `iseq` and `dfp` values I described above, along with a few other values:

- `self`: As we'll see in the next section when I cover the lambdas, procs and bindings, the value the `self` pointer had when the block was first referred to is also an important part of the closure's environment. Ruby executes block code inside the same object context the code outside the block had.
- `lfp`: It turns out blocks also contain a local frame pointer, along with the dynamic frame pointer. However, Ruby doesn't use local variable access inside of blocks; it doesn't use the `set/getlocal` YARV instructions inside of blocks. Instead, Ruby uses this `LFP` for internal, technical reasons and not to access local variables.
- `proc`: Finally, Ruby uses this value when it creates a proc object from a block. As we'll see in the next section, procs and blocks are closely related.

Right above the definition of `rb_block_t` in vm_core.h you'll see the `rb_control_frame_t` structure defined:

```
typedef struct {
    VALUE *pc;            /* cfp[0] */
    VALUE *sp;            /* cfp[1] */
    VALUE *bp;            /* cfp[2] */
    rb_iseq_t *iseq;      /* cfp[3] */
    VALUE flag;           /* cfp[4] */
    VALUE self;           /* cfp[5] / block[0] */
    VALUE *lfp;           /* cfp[6] / block[1] */
    VALUE *dfp;           /* cfp[7] / block[2] */
    rb_iseq_t *block_iseq;/* cfp[8] / block[3] */
    VALUE proc;           /* cfp[9] / block[4] */
    const rb_method_entry_t *me;/* cfp[10] */
} rb_control_frame_t;
```

Notice that this C structure also contains all of the same values the `rb_block_t` structure did: everything from `self` down to `proc`. The fact that

these two structures share the same values is actually one of the interesting, but confusing, optimizations Ruby uses internally to speed things up a bit. Whenever you refer to a block for the first time by passing it into a method call, as I explained above, Ruby creates a new `rb_block_t` structure and copies values such as the `LFP` from the current `rb_control_frame_t` structure into it. However, by making the members of these two structures similar – `rb_block_t` is a subset of `rb_control_frame_t`; they contain the same values in the same order – Ruby is able to avoid creating a new `rb_block_t` structure and instead sets the pointer to the new block to refer to the common portion of the `rb_control_frame` structure. In other words, instead of allocating new memory to hold the new `rb_block_t` structure, Ruby simply passes around a pointer to the middle of the `rb_control_frame_t` structure. This is very confusing, but does avoid unnecessary calls to `malloc`, and speeds up the process of creating blocks.

Experiment 5-1: Which is faster: a while loop or passing a block to each?

I said earlier that in my opinion Ruby code containing blocks is often more elegant and succinct than the equivalent code would be using an older language such as C. For example, in C I would write a simple while loop to add up the numbers 1 through 10 like this:

```
#include <stdio.h>
main()
{
  int i, sum;
  i = 1;
  sum = 0;
  while (i <= 10) {
    sum = sum + i;
    i++;
  }
  printf("Sum: %d\n", sum);
}
```

…and I could use a while loop in Ruby in the same manner:

```
sum = 0
i = 1
while i <= 10
  sum += i
  i += 1
end
puts "Sum: #{sum}"
```

However, most Rubyists would write this loop using an iterator with a block, like this:

```
sum = 0
(1..10).each do |i|
  sum += i
```

```
end
puts "Sum: #{sum}"
```

Aesthetics aside, is there any performance penalty for using a block here? Does Ruby slow down significantly in order to create the new `rb_block_t` structure, copy the `DFP` value, and create new stack frames – everything I discussed above?

I won't benchmark the C code – clearly that will be faster than either option using Ruby. Instead, let's measure how long it takes Ruby to add up the integers 1 through 10 to obtain 55, using a simple while loop:

```
require 'benchmark'
ITERATIONS = 1000000
Benchmark.bm do |bench|
  bench.report("iterating from 1 to 10, one million times") do
    ITERATIONS.times do
      sum = 0
      i = 1
      while i <= 10
        sum += i
        i += 1
      end
    end
  end
end
```

Here I am using the benchmark library to measure the time required to run the while loop one million times. Admittedly I'm using a block to control the million iterations (`ITERATIONS.times do`) but I'll use the same block in the next test as well.

On my laptop with Ruby 1.9.2, I can run through this code in just over a half second:

```
$ ruby while.rb
      user     system      total        real
      iterating from 1 to 10, one million times  0.520000   0.000000
                                                 0.520000 (  0.514112)
```

Now let's measure the time required using the `each` iterator with a block:

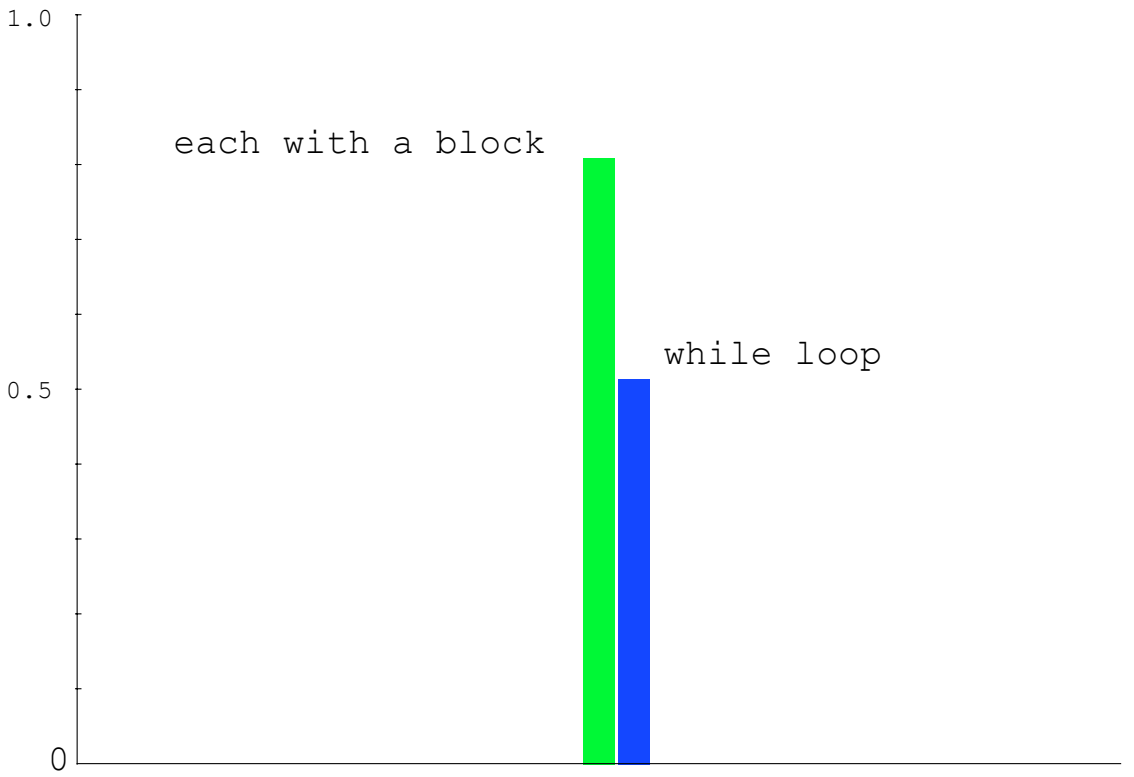
```
require 'benchmark'
ITERATIONS = 1000000
Benchmark.bm do |bench|
  bench.report("iterating from 1 to 10, one million times") do
```

```
    ITERATIONS.times do
      sum = 0
      (1..10).each do |i|
        sum += i
      end
    end
  end
end
```

This time it takes somewhat longer to run through the loop a million times, about 0.8 seconds:

```
$ ruby each.rb
      user     system      total        real
      iterating from 1 to 10, one million times  0.800000   0.000000
                                                 0.800000 (  0.808752)
```

Ruby requires about 57% more time to call the block 10 times, compared to iterating through the simple while loop 10 times.

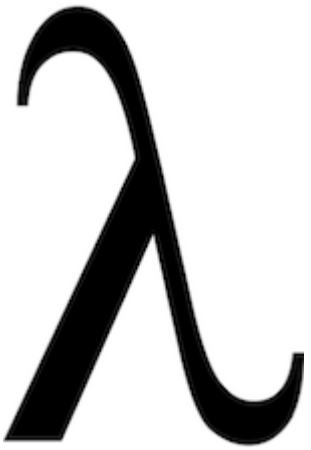


Time for 1,000,000 iterations (sec)

At first glance, 57% more time seems like a large performance penalty… just for making your Ruby code somewhat more readable and pleasant to look at. Depending on your work and the context of this while loop this may or may not be an important difference. If this loop were part of a time-sensitive, critical operation that your end users were waiting for – and if there weren't other expensive operations inside the loop – then writing the iteration using an old-fashioned C style while loop might be worthwhile.

However, the performance of most Ruby applications, and certainly Ruby on Rails web sites, is usually limited by database queries, network connections and other factors – and not by Ruby execution speed. It's rare that Ruby's execution speed has an immediate, direct impact on your application's overall performance. Of course, if you are using a large framework such as Ruby on Rails then your own Ruby code is a very small piece of a very large system. I imagine that Rails uses blocks and iterators many, many times while processing a simple HTTP request, apart from the Ruby code you write yourself.

Lambdas and Procs: treating functions as a first class citizen



In the 1930s Alonzo Church introduced "λ-notation" in his research on Lambda Calculus

Next let's look at a different, convoluted way of printing the same string to the console:

```
def message_function
  str = "The quick brown fox"
  lambda do |animal|
    puts "#{str} jumps over the lazy #{animal}."
  end
end
function_value = message_function
function_value.call('dog')
```

Here you can see I'm using a `lambda` keyword to return a block, which I call later after `message_function` returns. This is an example of "treating a function as a first class citizen," to paraphrase a commonly used computer science expression. Here I use the block as just another type of data – I return it from `message_function`, I save it in `code_value` and finally I call it explicitly using the `call` method. With the `lambda` keyword – or with the equivalent `proc` keyword or `Proc` object – Ruby allows you to convert a block into a data value like this.

In a moment, I'll take a look inside of Ruby to see what happens when I call `lambda` and how Ruby converts code into data. But first: where does the word "lambda" come from? Why in the world did Ruby choose to use a Greek letter as a language keyword? Once again, Ruby has borrowed an idea from Lisp. `Lambda` is also a reserved keyword in Lisp; it allows Lisp programmers to create an anonymous function like this:

```
(lambda (arg) (/ arg 2))
```

Like in my Ruby example, Lisp developers can treat anonymous functions like the one above as data, passing them into other functions as arguments.

Taking a look even farther back in history, however, it turns out that the term "lambda" was introduced well before John McCarthy invented Lisp in 1958. You may have noticed that Sussman and Steele's 1975 paper was titled *Scheme: An Interpreter for Extended Lambda Calculus*. Here they are referring to an area of mathematical study called "Lambda Calculus" invented by Alonzo Church in the 1930s. As part of his research, Church formalized the mathematical study of functions, and introduced the convention of using the Greek letter λ to refer to a function along with an ordered list of arguments the function uses. Later in his 1960 paper introducing Lisp, *Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I* , John McCarthy references Church's work directly while discussing functions and function definitions.

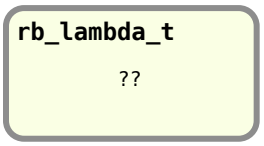
Stack memory vs. heap memory

Now let's return to my example:

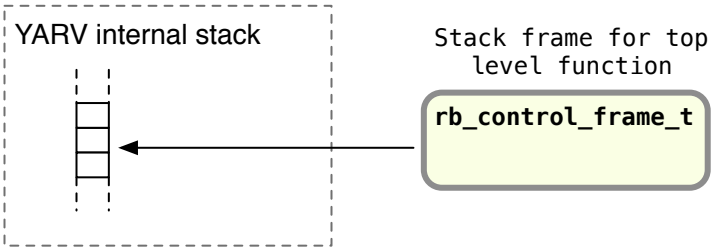
```
def message_function
  str = "The quick brown fox"
  lambda do |animal|
    puts "#{str} jumps over the lazy #{animal}."
  end
end
function_value = message_function
function_value.call('dog')
```

What happens when I call `lambda`? How does Ruby convert the block into a data value? What does it really mean to treat this function as a first class citizen?

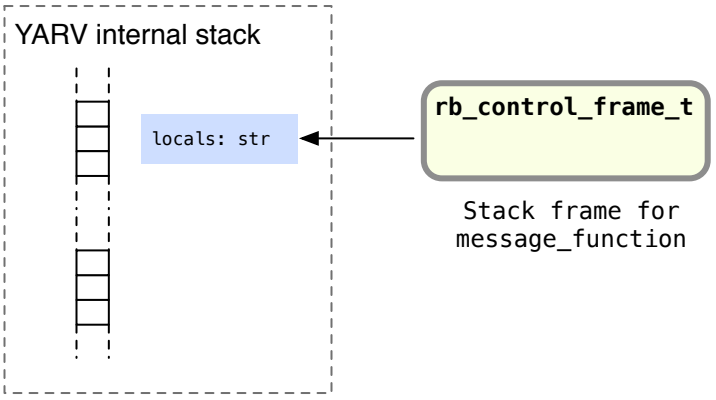
Does the `message_function` function return an `rb_block_t` structure directly? Or does it return an `rb_lambda_t` structure? If we could look inside this, what would we see? How does it work?



Before trying to understand what Ruby does internally when you call `lambda`, let's first review how Ruby handles the string value `str` more carefully. First imagine that YARV has a stack frame for the outer function scope but hasn't called `message_function` yet:

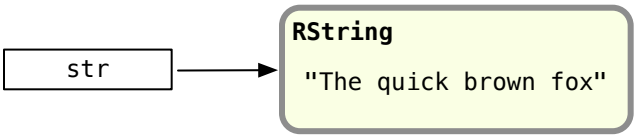


As usual you can see YARV's internal stack on the left, and the `rb_control_frame` structure on the right. Now suppose Ruby executes the `message_function` function call:

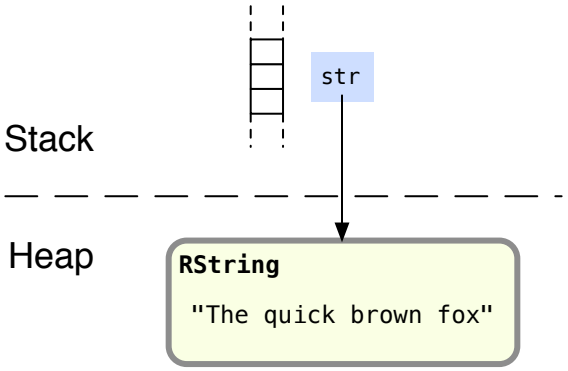


Here again we have the `str` variable saved in the top level stack frame used by `message_function`. Before going farther, let's take a closer look at that `str` variable and how Ruby stores the "quick brown fox" string in it. Recall from Chapter 3 that Ruby stores each of your objects in a C structure called `RObject`, each of your arrays in an

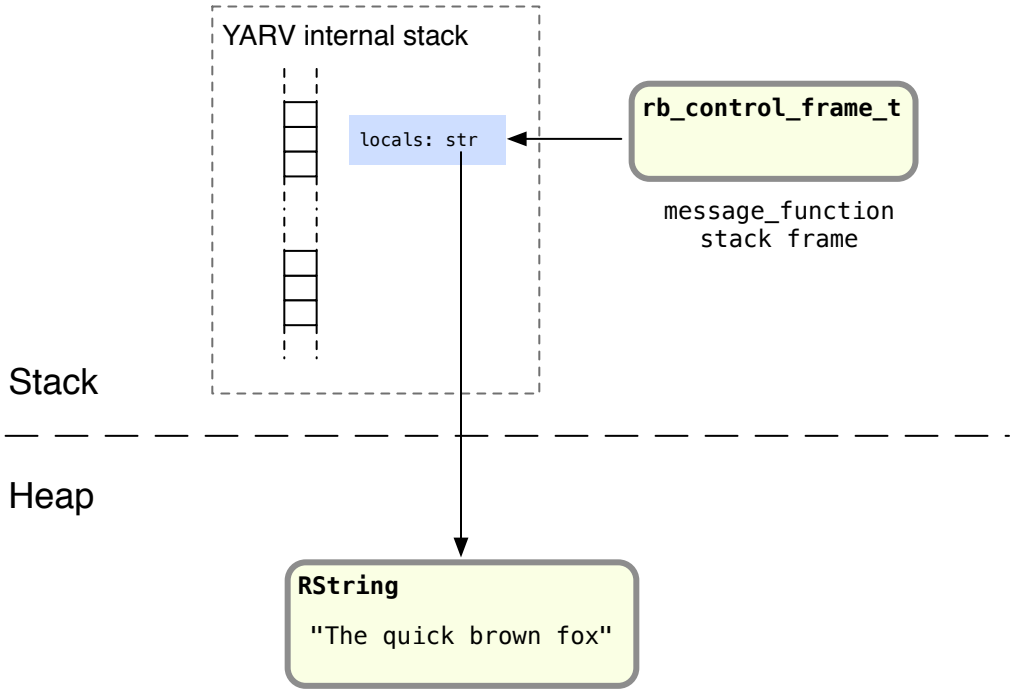
`RArray` structure, and similarly each of your strings in a structure called `RString`. For example, Ruby saves the quick brown fox string like this:



We have the actual string structure on the right, and a reference or pointer to the string on the left. When Ruby saves a string value on the YARV stack – or any object value for that matter – it actually only places the reference to the string on the stack. The actual string structure is saved in the "heap" instead:



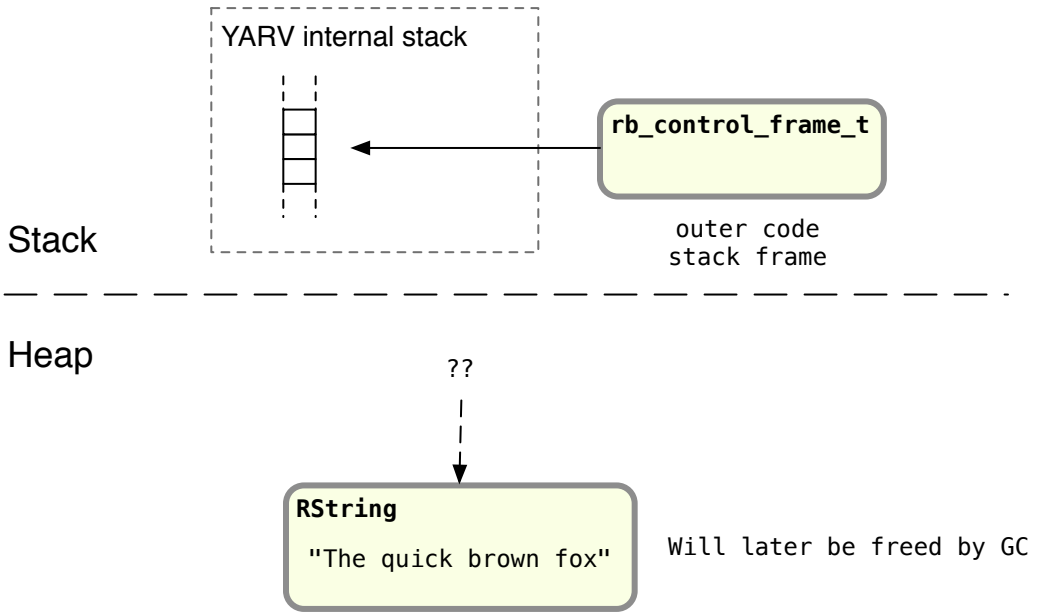
The heap is a large area of memory that Ruby or any other C program can allocate memory from. Objects or other values that Ruby saves in the heap remain valid for as long as there is a reference to them, the `str` pointer in this example. Here's a more accurate picture of what Ruby does when we create the `str` local variable inside of `message_function`:



After there are no longer any pointers referencing a particular object or value in the heap, Ruby later can free it during the next run of the garbage collection system. To show this happening, let's suppose for a moment that my example code didn't call `lambda` at all, that instead it immediately returned `nil` after saving the `str` variable:

```
def message_function
  str = "The quick brown fox"
  nil
end
```

After this call to `message_function` finishes, YARV will simply pop the `str` value and any other temporary values saved there off the stack and return to the original stack frame:



You can see there no longer is any reference to the `RString` structure containing the "quick brown fox" string and that Ruby will free it later.

Stepping through how Ruby creates a lambda

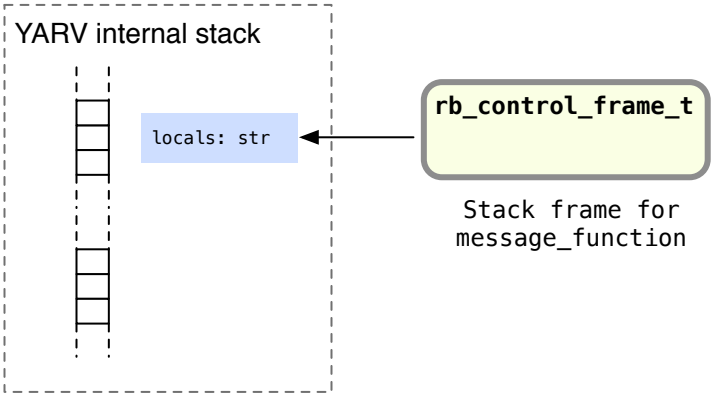
Now let's return to my original example code that returns the `lambda` expression instead of `nil`:

```
def message_function
  str = "The quick brown fox"
  lambda do |animal|
    puts "#{str} jumps over the lazy #{animal}."
  end
end
function_value = message_function
function_value.call('dog')
```

Notice that later when I actually call the lambda – the block – the `puts` statement inside the block is somehow able to access the `str` string variable from inside `message_function`. How can this be possible? We've just seen how the `str` reference to the `RString` structure is popped off the stack when `message_function`

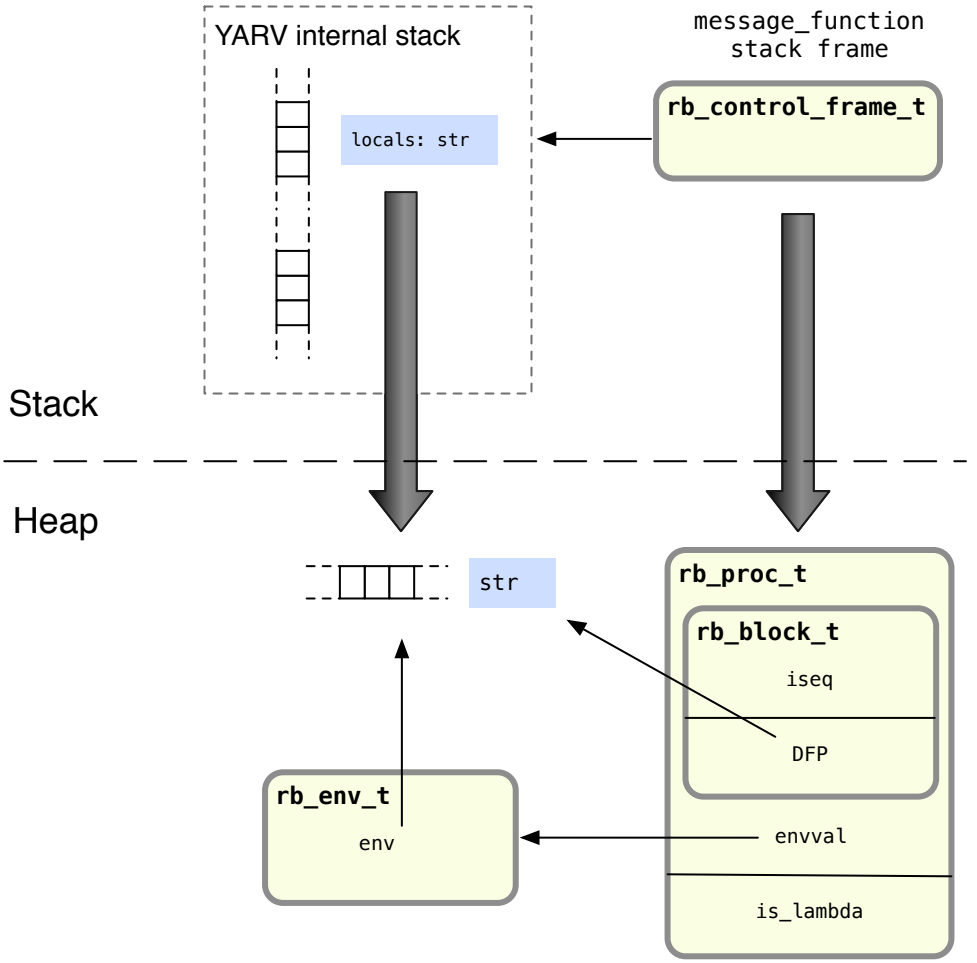
returns! Obviously, Ruby must do something special when you call `lambda`; somehow after calling `lambda` the value of `str` lives on so the block can later access it.

When you call `lambda`, internally Ruby copies the entire contents of the current YARV stack frame into the heap – the same place the `RString` structure is located. For example, here is the YARV stack again just after we call `message_function`:



To keep things simple, I don't show the `RString` structure here, but remember the `RString` structure will be saved in the heap.

Next, Ruby will call `lambda`; here's what happens internally:



You can see Ruby has created a new copy of the stack frame for `message_function` in the heap. I indicate that with the horizontal stack icon that appears below the dotted line. Now there is a second reference to the `str RString` structure, which means Ruby won't free it when `message_function` returns.

Along with the copy of the stack frame, Ruby creates two other new objects in the heap:

- An internal `environment` object, represented by the `rb_env_t` C structure at the lower left. This object only exists internally inside of Ruby; you can't access

this environment directly from your Ruby code. It is essentially a wrapper for the heap copy of the stack.

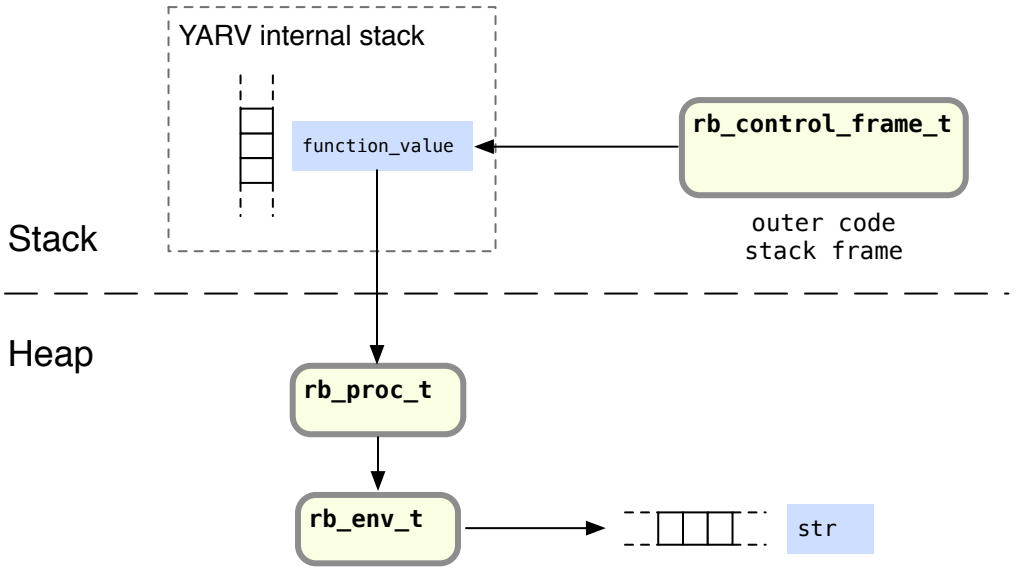
- A Ruby `Proc` object, represented by the `rb_proc_t` structure. As you may know, this is the actual return value from the `lambda` keyword; this object is what the `message_function` function returns.

Note the new Proc object structure, `rb_proc_t`, actually contains an `rb_block_t` structure, including the `iseq` and `DFP` pointers. Just as with a normal block, these keep track of the block's code and the referencing environment for the block's closure. Ruby sets the `DFP` inside this block to point to the new heap copy of the stack frame. You can think of a `Proc` as a Ruby object that wraps up a block; technically speaking, this is exactly what it is.

Also, notice the `Proc` object – the `rb_proc_t` structure – contains an internal value called `is_lambda`. This will be set to `true` for my example since I used the `lambda` keyword to create the `Proc`. If I had created the `Proc` using the `proc` keyword instead, or by just calling `Proc.new`, then `is_lambda` would be set to `false`. Ruby uses this flag to produce the slight behavior differences between procs and lambdas; however, it's best to think of procs and lambdas as essentially the same thing.

Stepping through how Ruby calls a lambda

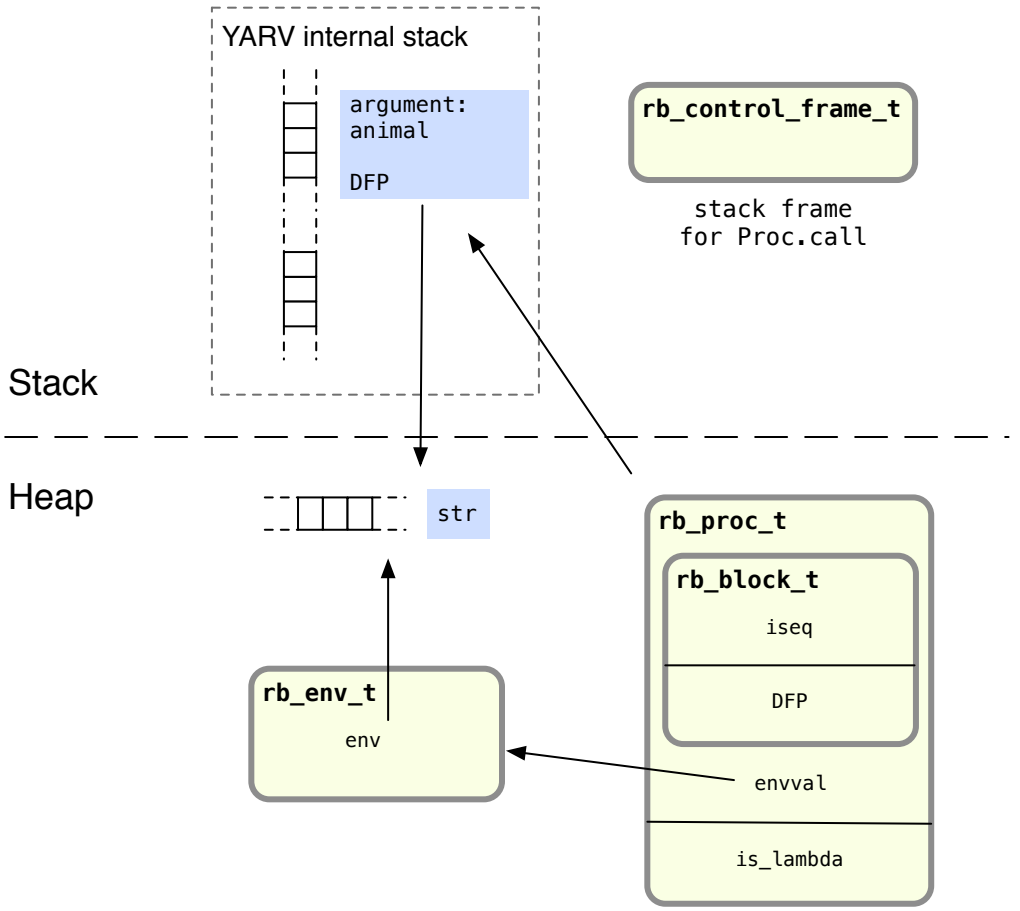
What happens when `message_function` returns? Since the lambda or proc object is the return value of `message_function`, a reference or pointer to the lambda is saved in the stack frame for the outer function in the `function_value` local variable. This prevents Ruby from freeing the proc, the internal environment object and the `str` variable; there are now pointers referring to all of these values in the heap:



Finally, when Ruby executes the `call` method on the proc object returned by `message_function`:

```
function_value = message_function
function_value.call('dog')
```

…it finally executes the block contained in the proc:

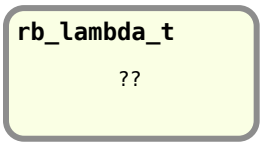


When Ruby calls a normal block, it creates a new stack frame for the block to use, and saves the `animal` value – the argument passed to the block – into this new stack frame. And just as with calling a normal block, Ruby also copies the `DFP` pointer from the `rb_block_t` structure into the new stack frame.

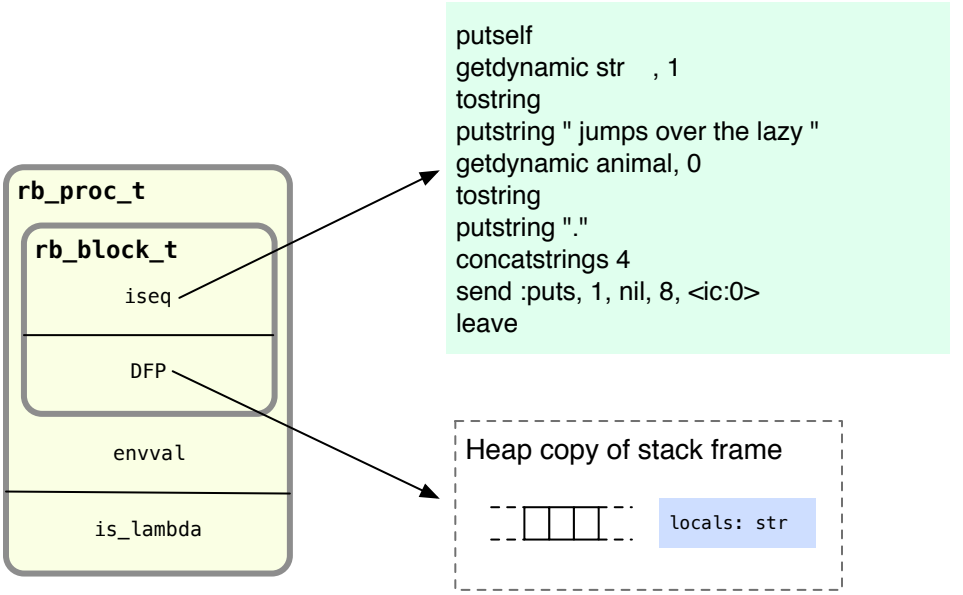
The only real difference here is that the `DFP` points to the copy of the stack frame Ruby created in the heap earlier when it executed `lambda`. This `DFP` allows the code inside the block, the call to `puts`, to access the `str` value.

The Proc object

Stepping back for a moment to review, we've just seen that inside of Ruby there really is no structure called `rb_lambda_t`:

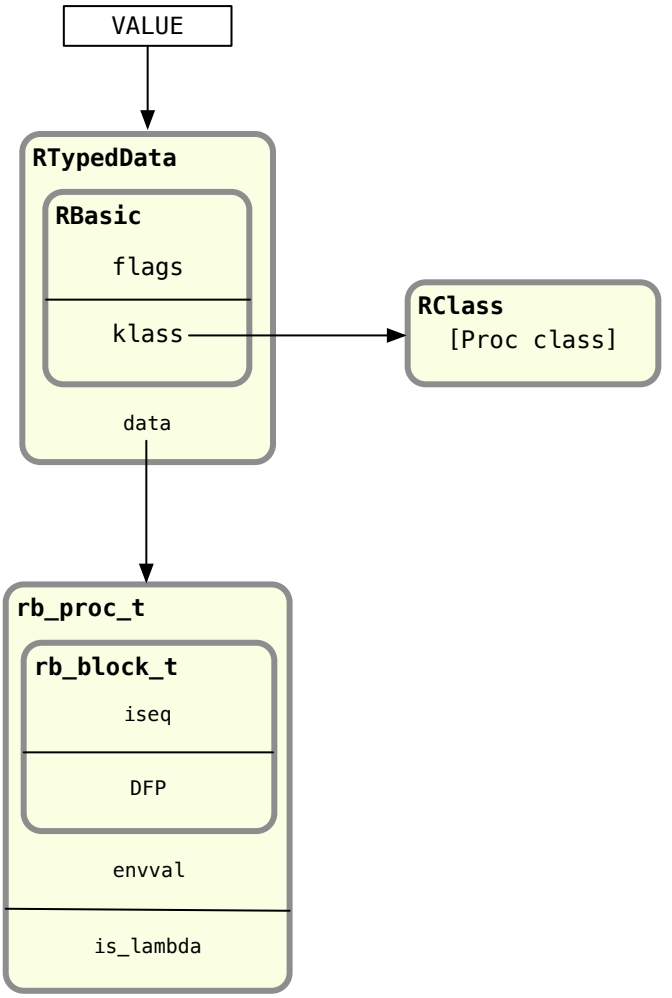


Instead, Ruby's `lambda` keyword created a proc object, which internally is a wrapper for a block – the block you pass to the `lambda` or `proc` keyword:



Just like a normal block, this is a closure: it contains a function along with the environment that function was referred to or created in. You can see that in this case the environment is a persistent copy of the stack frame saved in the heap.

There's an important difference between a normal block and a proc: procs are Ruby objects. Internally, they contain the same information that other Ruby objects contain, including the `RBasic` structure I discussed in Chapter 3. Above I mentioned that the `rb_proc_t` structure represents the Ruby proc object; it turns out this isn't exactly the case. Internally, Ruby uses another data type called `RTypedData` to represent instances of the proc object:



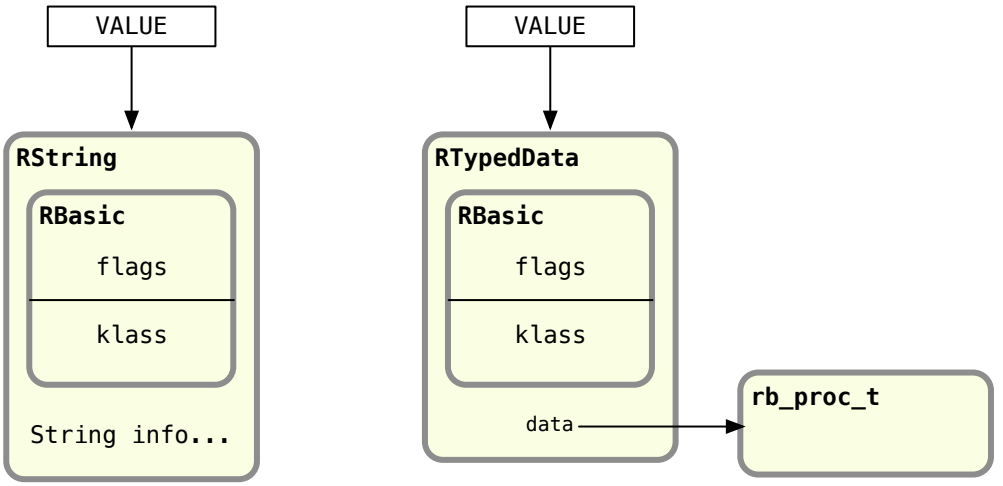
You can think of `RTypedData` as a trick that Ruby's C code uses internally to create a Ruby object wrapper around some C data structure. In this case, Ruby uses `RTypedData` to create an instance of the `Proc` Ruby class that represents a single copy of the `rb_proc_t` structure.

Here, just as we saw in Chapter 3, the `RTypedData` structure contains the same `RBasic` information that all Ruby objects contain:

- `flags`: some internal technical information Ruby needs to keep track of

- `klass`: a pointer to the Ruby class this object is an instance of, in this case the `Proc` class.

Here's a another look at how Ruby represents a proc object, this time shown on the right next to a `RString` structure:



Notice how Ruby handles the string value and the proc value in a very similar way. Just like strings, for example, procs can be saved into variables or passed as arguments to a function call. Ruby uses the `VALUE` pointer to the proc whenever you do this.

Experiment 5-2: Changing local variables after calling lambda

My previous code example showed how calling `lambda` makes a copy of the current stack frame in the heap.

```
def message_function
  str = "The quick brown fox"
  lambda do |animal|
    puts "#{str} jumps over the lazy #{animal}."
  end
end
function_value = message_function
function_value.call('dog')
```

Specifically, the `str` string value is valid here even after `message_function` returns. But what happens if I modify this value in `message_function` after calling lambda?

```
def message_function
  str = "The quick brown fox"
  func = lambda do |animal|
    puts "#{str} jumps over the lazy #{animal}."
  end
  str = "The sly brown fox"
  func
end
function_value = message_function
function_value.call('dog')
```

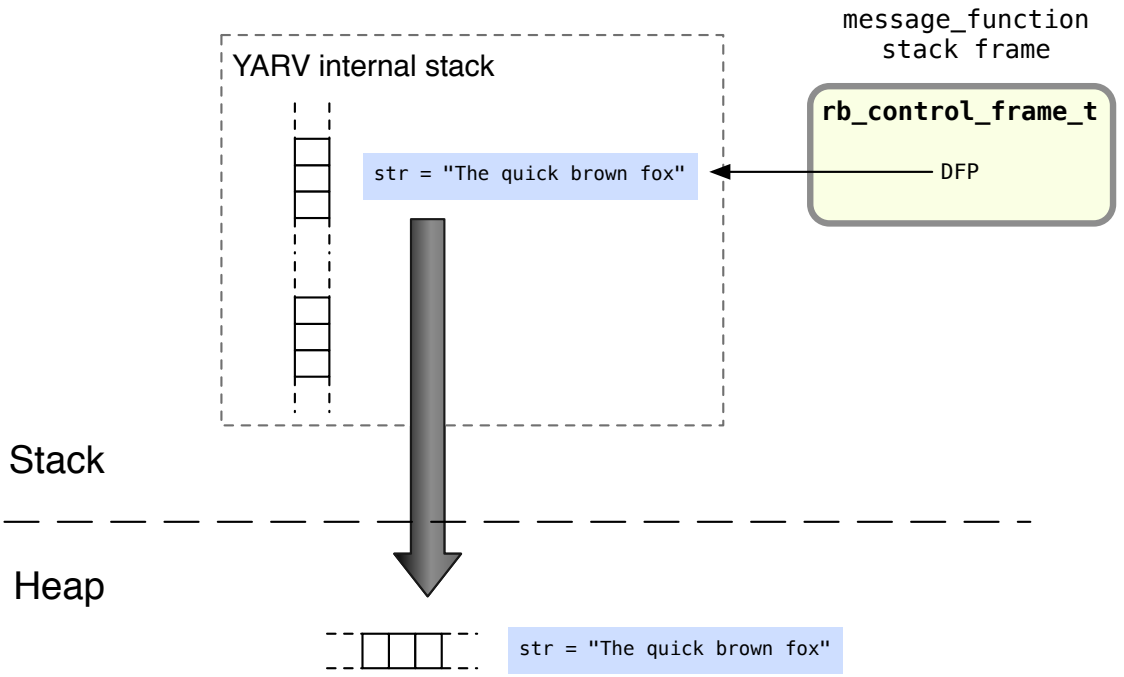
Notice now I change the value of `str` after I create the lambda. Running this code I get:

```
$ ruby modify_after_lambda.rb
The sly brown fox jumps over the lazy dog.
```

How is this possible? In the previous section I discussed how Ruby makes a copy of the current stack frame when I call `lambda`. In other words, Ruby copies the stack frame after running this code:

```
str = "The quick brown fox"
func = lambda do | animal |
```

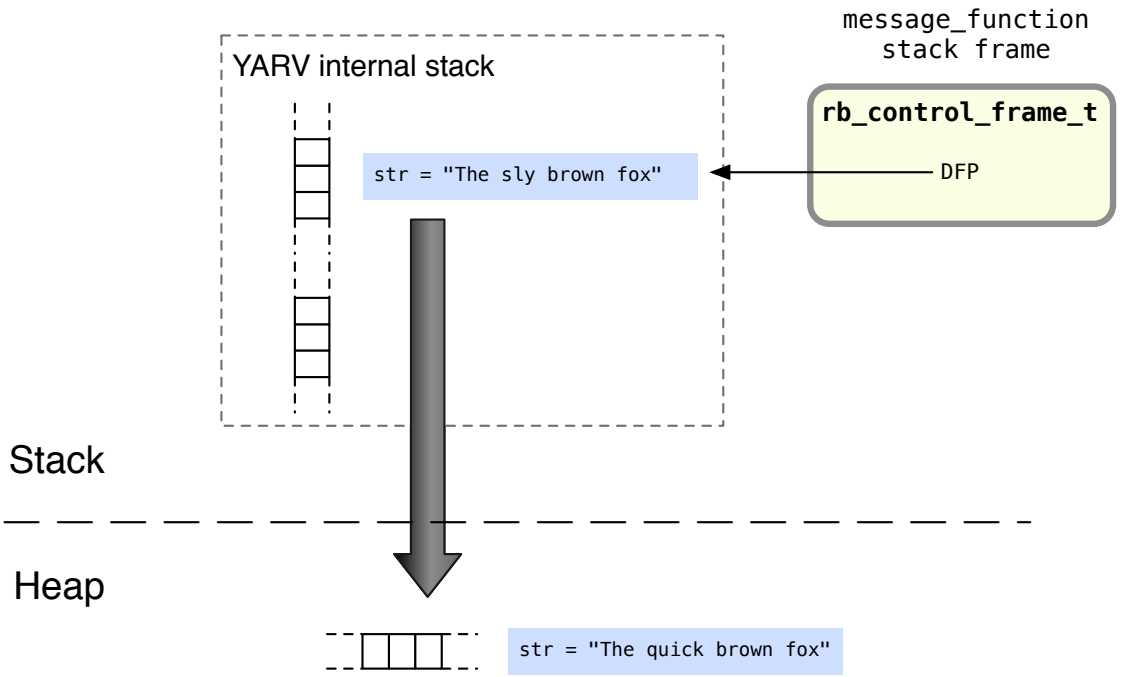
```
  puts "#{str} jumps over the lazy #{animal}."
end
```



Then after this copy is made, I change `str` to the “sly fox” string:

```
str = "The sly brown fox"
```

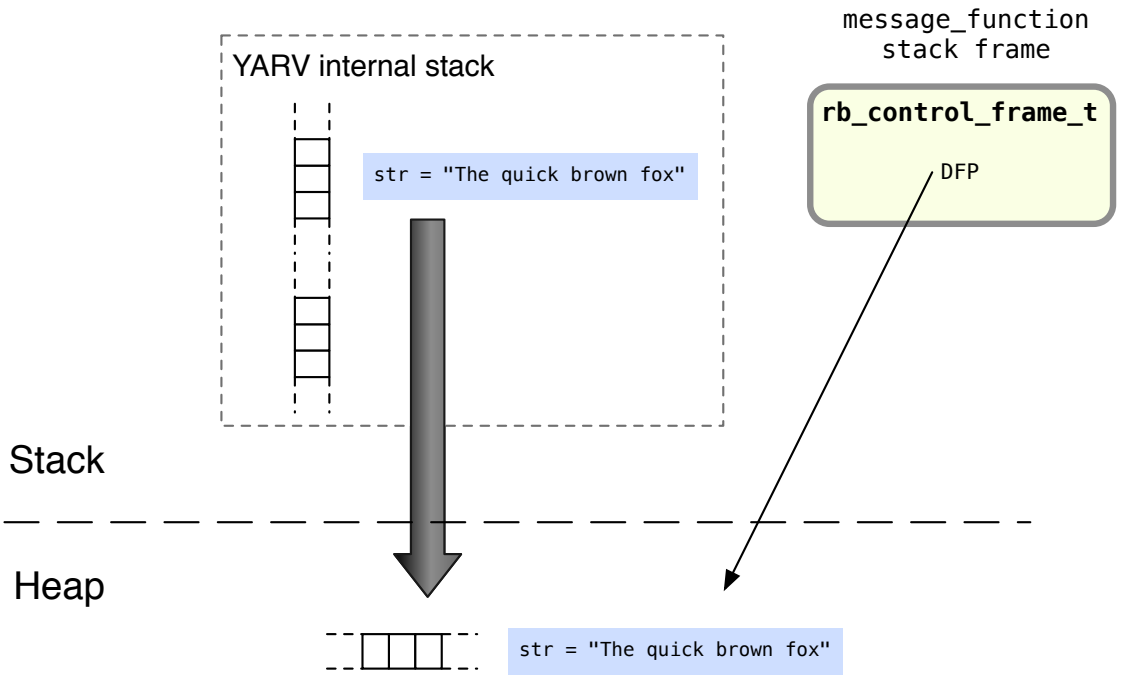
Since Ruby copied the stack frame above when I called `lambda`, now I should be modifying the original copy of `str` and not the new lambda copy:



This means that the new, lambda copy of the string should have remained unmodified. Calling the lambda later I should have gotten the original “quick fox” string, not the modified “sly fox” string.

What happened here? How does Ruby support this behavior? How does Ruby allow me to modify the new, persistent copy of the stack after it’s been created by `lambda`?

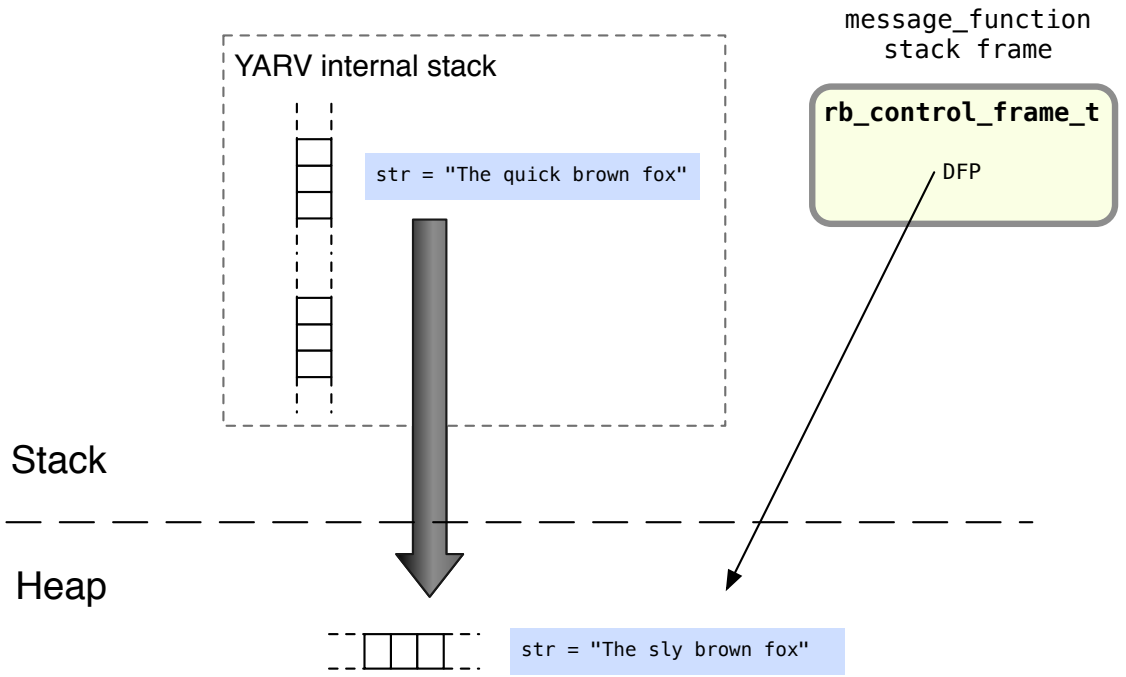
An important detail that I left out of my diagrams in the previous section is that after Ruby creates the new heap copy of the stack – the new `rb_env_t` structure or internal “environment” object – Ruby also resets the `DFP` in the `rb_control_frame_t` structure to point to the copy. Here’s another view of Ruby copying the local stack frame into the heap:



The only difference here is that the `DFP` now points down to the heap. This means that when my code accesses or changes any local variables after calling `lambda`, Ruby will use the new `DFP` and access the value in the heap, not the original value on the stack. In this code, for example:

str = “The sly brown fox”

Ruby actually modifies the new, heap copy that will later be used when I call the lambda:



Another interesting behavior of the `lambda` keyword is that Ruby avoids making copies of the stack frame more than once. For example, suppose that I call `lambda` twice in the same scope:

```
i = 0
increment_function = lambda do
  puts "Incrementing from #{i} to #{i+1}"
  i += 1
end
decrement_function = lambda do
  i -= 1
  puts "Decrementing from #{i+1} to #{i}"
end
```

Here I expect both lambda functions to operate on the local variable `i` in the main scope. Thinking about this for a moment, if Ruby made a separate copy of the stack frame for each call to `lambda`, then each function would operate on a separate copy of `i` – and would call my lambdas like this…

```
increment_function.call
decrement_function.call
increment_function.call
increment_function.call
decrement_function.call
```

…would yield these results:

```
Incrementing from 0 to 1
Decrementing from 0 to -1
Incrementing from 1 to 2
Incrementing from 2 to 3
Decrementing from -1 to -2
```

But instead I actually get this:

```
Incrementing from 0 to 1
Decrementing from 1 to 0
Incrementing from 0 to 1
Incrementing from 1 to 2
Decrementing from 2 to 1
```

Most of the time this is what you expect: each of the blocks you pass to the lambdas access the same variable in the parent scope. Ruby achieves this simply by checking whether the `DFP` already points to the heap. If it does, as it would in this example the second time I call `lambda`, then Ruby won't create a second copy again. It would simply reuse the same `rb_env_t` structure in the second `rb_proc_t` structure. Both lambdas would use the same heap copy of the stack.

Metaprogramming and closures: eval, instance_eval and binding

During the 1980s, computers designed specifically for running Lisp were built and commercialized.

Metaprogramming literally means to write a program that can, in turn, write another program. In Ruby, the `eval` method is metaprogramming in its purest form: you pass a string to Ruby and it immediately compiles it and executes it. Here's an example:

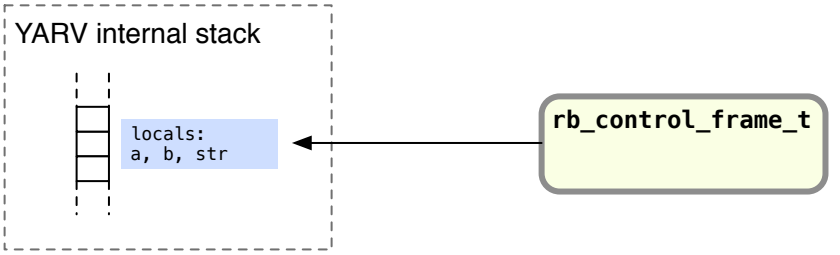
```
str = "puts"
str += " 2"
str += " +"
str += " 2"
eval(str)
```

As you can see here, I dynamically construct a string, "`puts 2+2`," and then pass it to `eval`. Ruby then evaluates the string – it tokenizes it, parses it, and compiles it using the algorithms I discussed in Chapter 1. Ruby uses exactly the same Bison grammar rules and parse engine that it did when it first processed your primary Ruby script. Once this process is finished and Ruby has another new set of YARV byte code instructions, it then executes your new code.

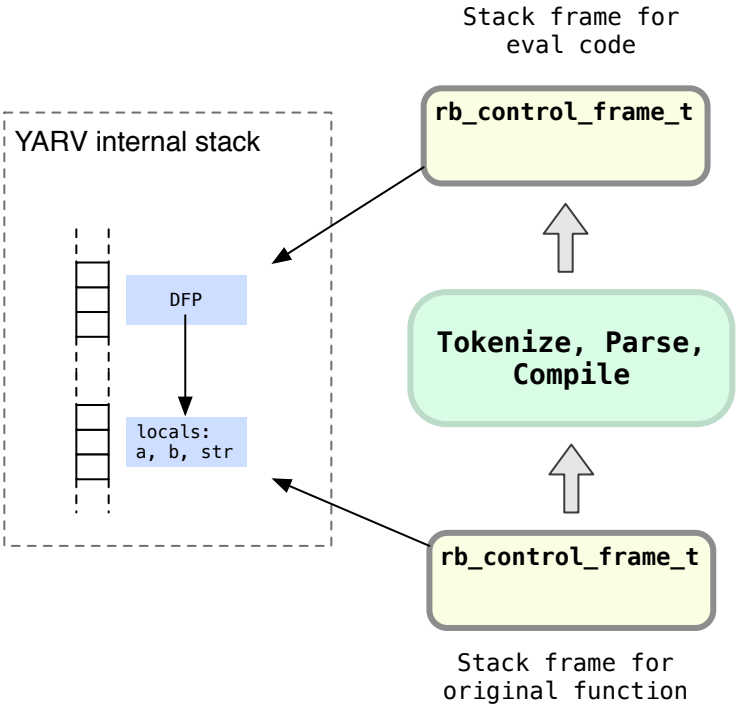
However, one very important detail about the `eval` method, which isn't obvious in the example above, is that Ruby evaluates the new code string in the same context where you called `eval` from. To see what I mean, consider this similar example:

```
a = 2
b = 3
str = "puts"
str += " a"
str += " +"
str += " b"
eval(str)
```

Running this code I get the result you would expect: 5. But notice the difference between this example and the previous one; in this second example I refer to the local variables `a` and `b` from the surrounding scope and Ruby is able to access their values without a problem. Internally, this works in the same way a block would; let's take a look:



Here, as usual, you can see Ruby has saved the values of `a`, `b` and `str` on the stack to the left; on the right we have the `rb_control_frame_t` structure representing the outer or main scope of my Ruby script. Next, when I call the `eval` method, here's what happens:



Above the green icon indicates that calling `eval` invokes the parser and compiler on the text I pass it. When the compiler finishes, Ruby creates a new stack frame for running the new, compiled code; you can see this on the top. But notice that Ruby sets the `DFP` in this new stack frame to point to the lower stack frame where the variables `a` and `b` are. This allows the code passed to `eval` to access these values.

This should look familiar; aside from parsing and compiling the code dynamically, this functions in precisely the same way as if I had instead passed a block to some function:

```
a = 2
b = 3
10.times do
  puts a+b
end
```

In other words, the `eval` method creates a closure: the combination of a function and the environment where that function was referenced. In this case, the function is the newly compiled code, and the environment is the place where I called `eval` from.

Calling eval with binding

As an option, the `eval` method can take a second parameter: the `binding`. Passing a binding value to Ruby indicates that you don't want to use the current context as the closure's environment, but instead some other environment. Here's an example:

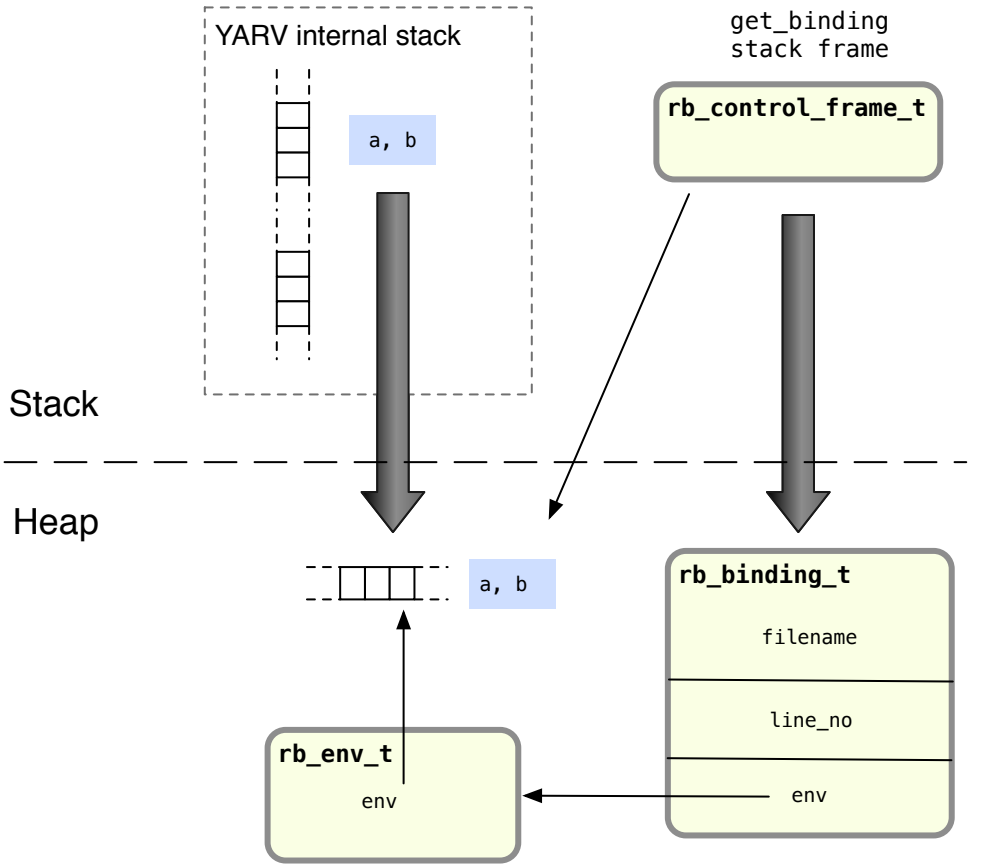
```
def get_binding
  a = 2
  b = 3
  binding
end
eval("puts a+b", get_binding)
```

Here I have written a function called `get_binding` which contains two local variables `a` and `b` – but note the function also returns a binding. At the bottom, I once again want Ruby to dynamically compile and execute the code string and print out the expected result of 5. The difference is that, by passing the binding returned by `get_binding` to `eval`, I want Ruby to evaluate "`puts a+b`" inside the context of the `get_binding` function.

A binding is a closure without a function; that is, it's just the environment. Just as it does when you call the `lambda` keyword, Ruby makes a persistent copy of this environment in the heap because you might call `eval` long after the current frame has

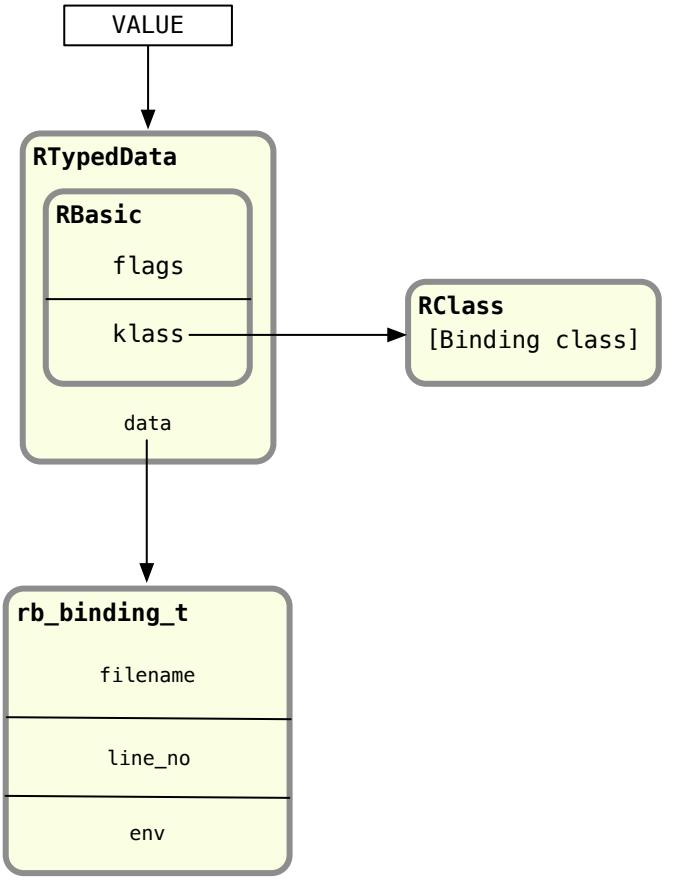
been popped off the stack. In my example, even though `get_binding` has already returned, when Ruby executes the code parsed and compiled by `eval` it still is able to access the values of `a` and `b`.

Below is what happens internally when I call `binding`:



This looks very similar to what Ruby does when you call `lambda` – the only difference is that Ruby creates an `rb_binding_t` C structure instead of an `rb_proc_t` structure. The binding structure is simply a wrapper around the internal environment structure, i.e. around the heap copy of the stack frame. The binding structure also contains the file name and line number of the location where you called `binding` from.

Just like the `Proc` object, Ruby uses the `RTypedData` structure to wrap a Ruby object around the `rb_binding_t` C structure:



With the `Binding` object, Ruby allows you to create a closure, and then obtain and treat the closure's environment as a data value. The closure created by the binding, however, doesn't contain any code – it's a closure without a function. You can also think of the `Binding` object as an indirect way of accessing, saving and passing around Ruby's internal `rb_env_t` structure.

Stepping through a call to instance_eval

Finally, let's look at a variation on the `eval` method: `instance_eval`. Here's yet another way of printing out the same "quick brown fox" string to the console:

```
class Quote
  def initialize
    @str = "The quick brown fox"
```

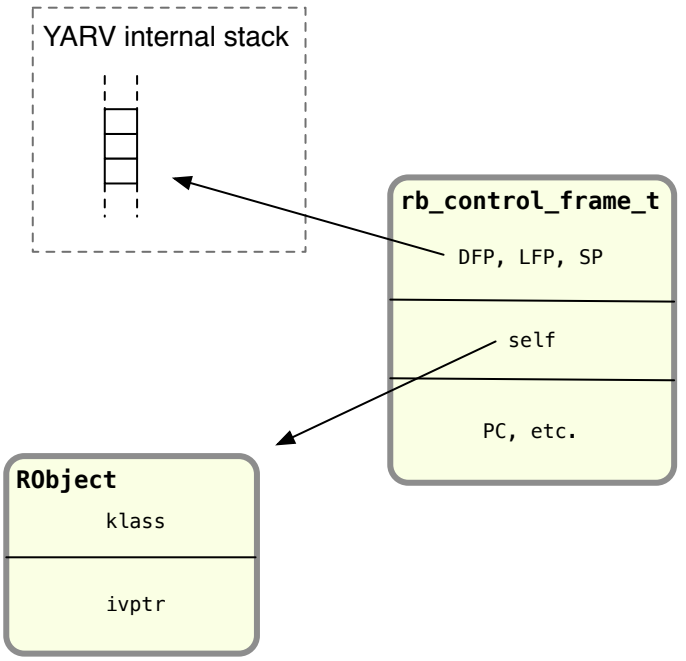
```
  end
end
str2 = "jumps over the lazy dog."
obj = Quote.new
obj.instance_eval do
  puts "#{@str} #{str2}"
end
```

This example is even more complicated, so let me take a moment to explain how it works:

- First, I create a Ruby class call `Quote`, which saves the first half of my string in an instance variable, `@str`, when I initialize any new `Quote` instance.
- At the bottom I actually create an instance of the `Quote` class, and then call the `instance_eval` method, passing a block to it. `instance_eval` is similar to `eval`, except that it evaluates the given string in the context of the receiver, or the object I call `instance_eval` on. Also, as in this example, I can pass a block to `instance_eval` instead of a string if I don't want to dynamically parse and compile code.
- The block I pass to `instance_eval` here prints out the string, accessing the first half of the string from the quote object's instance variable, and the second half from the surrounding scope or environment.

How can this possibly work? It appears the block passed to `instance_eval` has two environments: the quote instance and also the surrounding code scope. In other words, the `@str` variable comes from one place, and the `str2` variable from another.

This example points out another important part of closure environments in Ruby: the current value of `self`. Recall from Chapter 2 that the `rb_control_frame_t` structure for each stack frame or level in your Ruby call stack contains a `self` pointer, along with the `PC`, `SP`, `DFP` and `LFP` pointers and other values. The `self` pointer records indicates the current value of `self` at that point in your Ruby project; it indicates which object is the owner of the method Ruby is currently executing at that time. Here's another, more accurate view of the `rb_control_frame_t` structure:

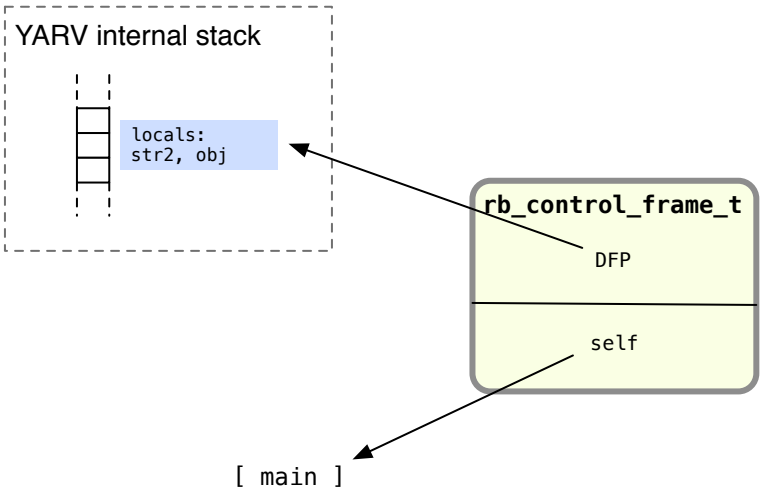


You can see `self` points to the `RObject` structure corresponding to the Ruby object to which the method Ruby is currently executing belongs. `self` might instead point at an `RString`, `RArray` or other type of built in Ruby object structure instead.

I mentioned above the `self` pointer is also part of closure environments; let's take a look at what Ruby does internally when you call `instance_eval`. First, suppose I have already declared the `Quote` class, and Ruby has just executed these two lines of code:

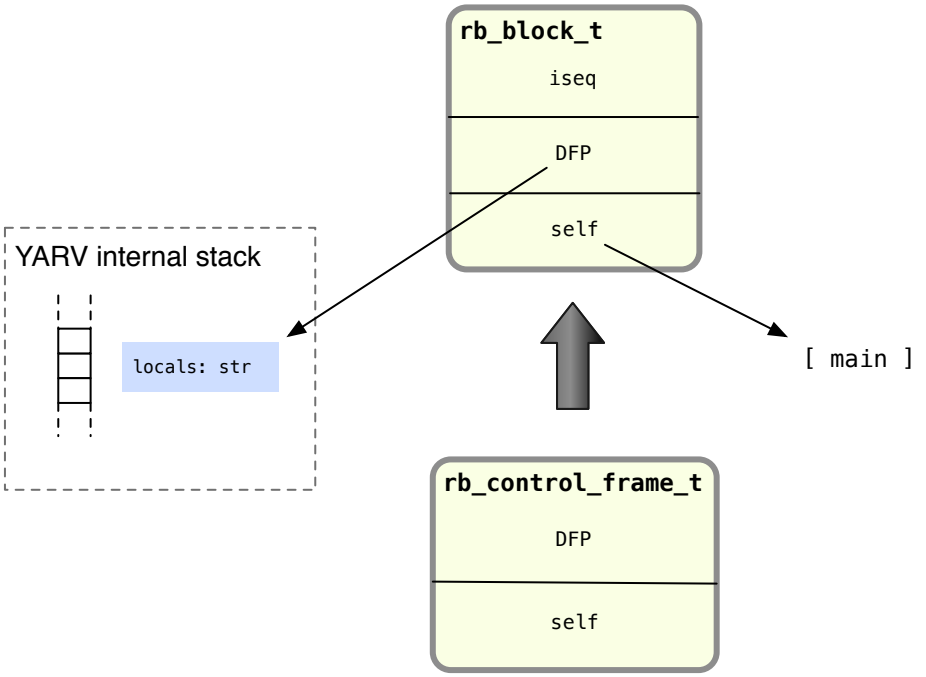
```
str2 = "jumps over the lazy dog."
obj = Quote.new
```

Here is the YARV stack with two local variables `str2` and `obj` saved on it:



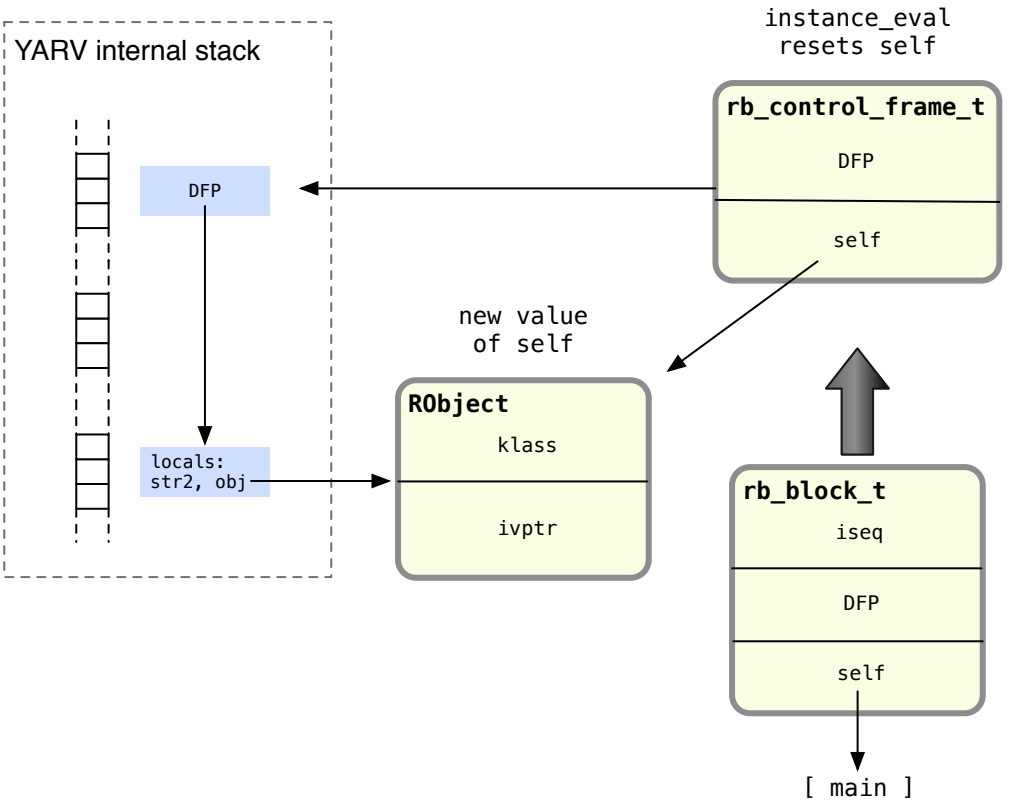
The `self` pointer here is initially set to `main`, the value of self Ruby uses in your top level scope. This is actually the "top self" object, an instance of the `Object` class Ruby creates when it starts up, just for this purpose.

Next Ruby executes the `instance_eval` method. Given I pass a block to `instance_eval` instead of a string, there's no need to startup the parser and compiler again. Ruby, however, does need to initialize a new `rb_block_t` structure to represent the new block:



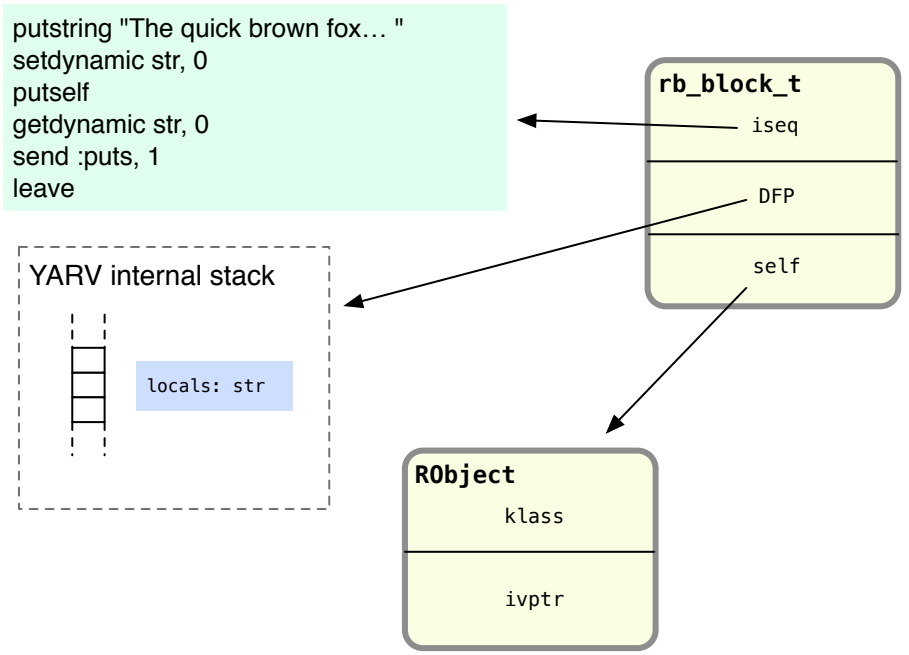
This works exactly the same way as if I had called a normal block without `instance_eval`. In fact, Ruby always saves the `self` pointer in `rb_block_t` when you pass a block into a function call.

Calling the block using `instance_eval` on `obj` tells Ruby you want to reset the `self` pointer from the value it would normally have – whatever `self` was in the code calling the block – to the receiver of `instance_eval` or `obj` in this example:



Another important part of Ruby closures

Stepping back for a moment, we've discovered another important member of the `rb_block_t` structure:



We know now, therefore, that closures in Ruby consist of:

- A function, referenced by the `iseq` pointer,
- A stack environment, referenced by the dynamic frame pointer, and
- A object environment, referenced by the `self` pointer.

Experiment 5-3: Using a closure to define a method

Another common metaprogramming pattern in Ruby is to dynamically define methods in a class using the `define_method` method. For example, here's a simple Ruby class that will print out a string when you call `display_message`:

```
class Quote
  def initialize
    @str = "The quick brown fox jumps over the lazy dog"
  end
  def display_message
    puts @str
  end
end
Quote.new.display_message
=> The quick brown fox jumps over the lazy dog
```

But using metaprogramming I could have defined `display_message` in a more verbose but dynamic way like this:

```
class Quote
  def initialize
    @str = "The quick brown fox jumps over the lazy dog"
  end
  define_method :display_message do
    puts @str
  end
end
```

You can see here I call `define_method` instead of the normal `def` keyword. Notice that the name of the new method is passed as an argument: `:display_message`. This allows you dynamically construct the method name from some data values or to iterate over an array of method names, calling `define_method` for each one.

However, notice there is another subtle difference between `def` and `define_method`. For `define_method` I provide the body of the method as a block… that is, I have to use a `do` keyword. This may seem like a minor syntax difference, but remember that blocks are actually closures. Adding that simple `do` keyword has introduced a closure, meaning that the code inside the new method has access to the environment outside. This is not the case with the simple `def` keyword.

In this example above there aren't any interesting values in the location where I call `define_method`. But suppose there was another location in my application that did have interesting values that I wanted my new method to be able to access and use – by using a closure Ruby will internally make a copy of that environment on the heap my new method will be able to use.

Let's repeat the same example, but this time only store the first half of the string in the instance variable:

```
class Quote
  def initialize
    @str = "The quick brown fox"
  end
end
```

Now I can define a method using a closure like this:

```
def create_method_using_a_closure
  str2 = "jumps over the lazy dog."
  Quote.send(:define_method, :display_message) do
    puts "#{@str} #{str2}"
  end
end
```

Note that since `define_method` is a private method in the `Module` class, I needed to use the confusing `send` syntax here. Earlier I was able to call `define_method` directly since I used it inside a class/module definition, but that isn't possible from other places in my application. By using `send` the `create_method_using_a_closure` method is able to call a private method it wouldn't normally have access to.

More importantly, you can see the `str2` variable, the second half of my example string, is preserved in the heap for my new method to use – even after `create_method_using_a_closure` returns:

```
create_method_using_a_closure
Quote.new.display_message
=> The quick brown fox jumps over the lazy dog.
```

Internally, Ruby treats this as a call to `lambda` – that is, this code functions exactly the same way as if I had written:

```
class Quote
  def initialize
    @str = "The quick brown fox"
  end
end
def create_method_using_a_closure
  str2 = "jumps over the lazy dog."
  lambda do
    puts "#{@str} #{str2}"
  end
end
Quote.send(:define_method, :display_message, create_method_using_a_closure)
Quote.new.display_message
```

Here I've separated the code that creates the closure and defines the method. If you pass 3 arguments to `define_method` Ruby expects the third to be a `Proc` object. While this is even more verbose, it's a bit less confusing since calling the `lambda` makes it clear Ruby will create a closure.

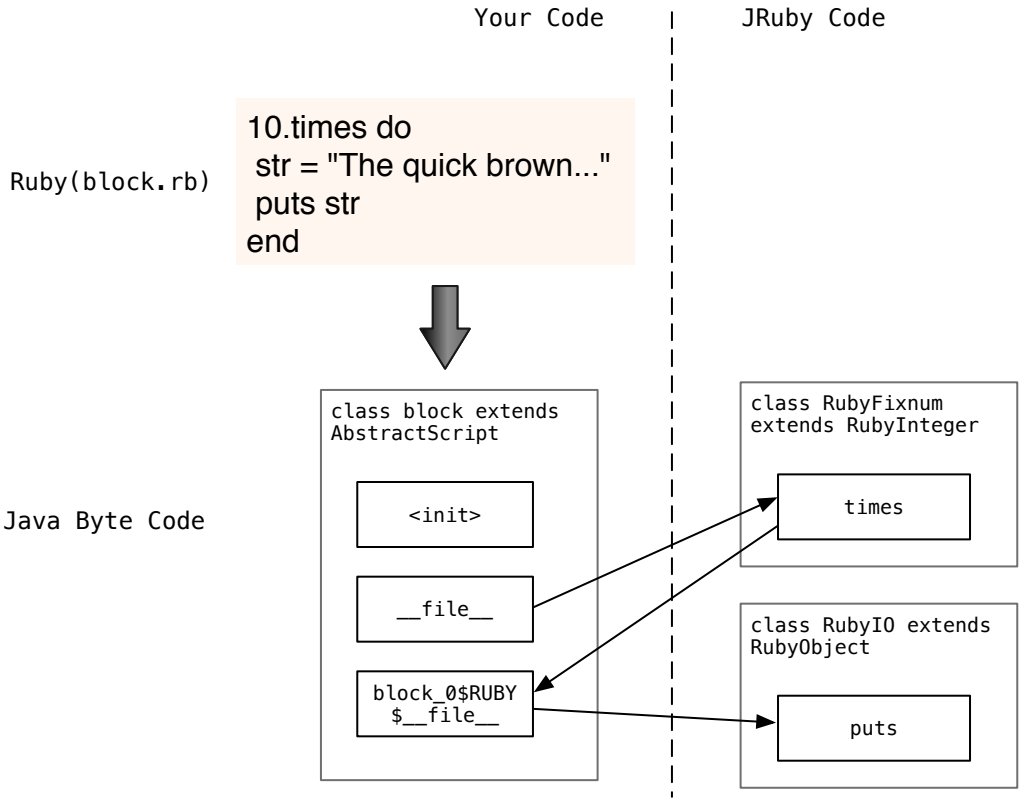
Finally, when I call the new method Ruby will reset the `self` pointer from the closure to receiver object, similar to how `instance_eval` works. That is, whatever object context the call to `create_method_using_a_closure` occurred in – maybe `create_method_using_a_closure` is a method defined in some other class, for example – Ruby uses `Quote.new` as the `self` pointer while executing `display_message`. This allows the new method to access `@str` as you would expect.

Closures in JRuby

Back in Chapter 2 I showed how JRuby executes a simple "`puts 2+2`" script. Now let's take a look at how JRuby executes some Ruby code that is just a bit more complex, for example:

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

Here's a conceptual diagram showing how JRuby will execute this script:



Let's walk through what's going on in this diagram:

- On the top left is my Ruby script, this time called `block.rb`. JRuby will compile this into a Java class called `block`, named after the Ruby source file `block.rb`.
- When it's time to start executing my script the JVM will call the **`file`** method, which corresponds to the top level code in my script. This works the same way my "`puts 2+2`" JRuby example did back in Chapter 2. However, notice now there is another Java method in the generated `block` class called `block_0$RUBY$__file__`". This oddly named method contains the compiled JVM byte code for the contents of my Ruby block.
- Next while executing the call to `10.times`, JRuby will call the `RubyFixnum.times` method, passing in the block as a parameter.
- Now `RubyFixnum.times` will iterate 10 times, calling the `block_0$RUBY$__file__` method each time through the loop.
- Finally, the block's code will in turn call the JRuby `RubyIO` Java class to print out the string.

The important detail to learn here is that JRuby's byte code compiler generates a separate Java method for each block or other scope in my Ruby script. Also note how JRuby passes control back and forth between the compiled version of my Ruby script and the Java classes that implement the Ruby's built in classes such as `Fixnum`.

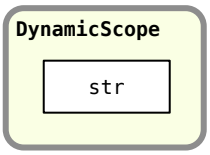
Now let's take a second example that uses a closure:

```
str = "The quick brown fox"
10.times do
  str2 = "jumps over the lazy dog."
  puts "#{str} #{str2}"
end
```

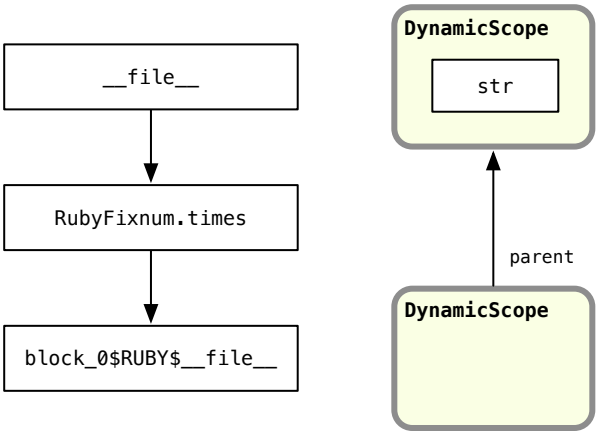
Here the block's code refers to the `str` variable in the parent scope. JRuby compiles this script in same way as the previous example, generating a Java class that contains three methods. But how does JRuby allow the block to access the `str` from the parent scope? Does it use a `DFP` pointer and dynamic variable access like MRI does?

The answer is no. One of the important differences between JRuby and MRI is that JRuby does not use the JVM stack to keep track of different scopes or closure environments in your Ruby program, in the same way that MRI Ruby uses the YARV stack. There is no equivalent to the Dynamic Frame Pointer in JRuby. Instead, JRuby

implements the same behavior using a series of Java classes, most importantly one called `DynamicScope`. When you reference variables from a different scope, i.e. when you use closures, JRuby saves the referenced variables inside a `DynamicScope` object. (Actually, there are a series of different Java classes that share `DynamicScope` as a common superclass that hold your closure variables.) In this example, JRuby saves my `str` variable inside a `DynamicScope` like this:



Later when JRuby executes the block, it creates a second `DynamicScope` object that refers to the parent scope like this:



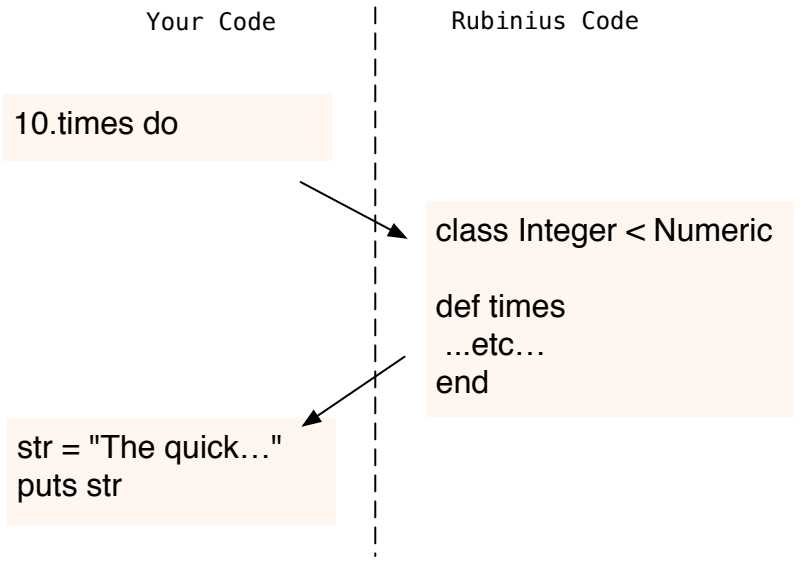
Each `DynamicScope` object contains a `parent` pointer that indicates which other dynamic scope is the enclosing or parent scope for this closure. This is JRuby's implementation of dynamic variable access. By iterating over these `parent` pointers, the JRuby Java code can get or set a variable that is present in a parent scope, or in the referencing environment of the closure.

Closures in Rubinius

Now let's take a look at how Rubinius handles blocks and closures. What happens inside of Rubinius when I call a block? Let's use the same two examples we just did with JRuby - first a simple call to a block:

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

Here's how Rubinius handles this:



Since in Rubinius the `Integer.times` method is implemented in Ruby, the call to `10.times` is a simple Ruby call. The `Integer.times` method, in turn, yields to my block code directly. All of this is implemented with Ruby!

Internally, Ruby compiles the `10.times` do" call into these byte code instructions:

10.times do

```
push_int                10
create_block            #<Rubinius::CompiledCode __block__ ...>
send_stack_with_block   :times, 0
pop
push_true
ret
```

Here the text `Rubinius::CompiledCode` refers to the block I'm passing to the `10.times` method call. Behind the scenes the Rubinius VM creates a C++ object to represent the block using the `create_block` instruction, and then passes that object along to the method call with the `send_stack_with_block` instruction.

Now let's take another example and see how Rubinius handles closures:

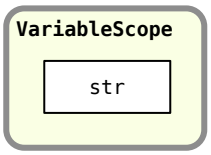
```
str = "The quick brown fox"
10.times do
  str2 = "jumps over the lazy dog."
  puts "#{str} #{str2}"
end
```

Again, this time my block code refers to a variable defined in the parent scope. How does Rubinius implement this? To find out, let's look at how Rubinius compiles the code inside the block into VM instructions:

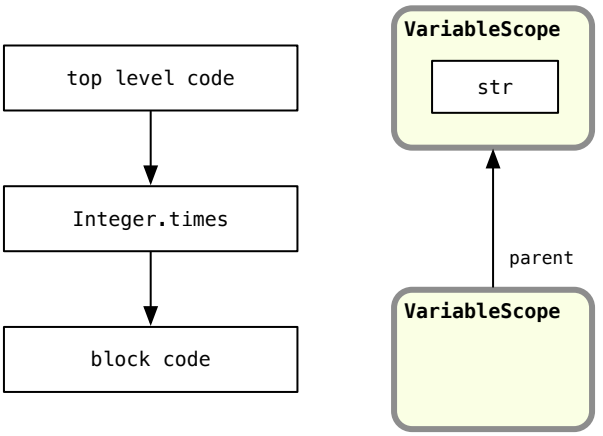
str2 = "jumps over…"
puts "#{str} #{str2}"

```
push_literal            "jumps over…"
string_dup
set_local               0
pop
push_self
push_local_depth        1, 0
allow_private
meta_to_s               :to_s
push_literal            " "
push_local              0
allow_private
meta_to_s               :to_s
string_build            3
allow_private
send_stack              :puts, 1
ret
```

I've shown the key VM instruction here in bold: Rubinius uses the `push_local_depth` instruction to walk up the stack and get the value of `str` from the parent scope. Internally Rubinius implements this instruction using a C++ object called `VariableScope`:



Just like the Java `DynamicScope` object did in JRuby, this C++ object saves an array of values - in other words the closure environment. Rubinius doesn't use tricks with the VM stack to save pointers the same way that YARV does. There is no dynamic frame pointer; instead, Rubinius represents closure environments, blocks, lambdas, procs and bindings with a set of different C++ classes. I don't have the space here to explain how all of that works in detail, but let's take a quick look at how Rubinius uses the `VariableScope` object to obtain the `str` value from the parent scope:



This should look familiar - in fact, Rubinius functions in exactly the same way that JRuby does when accessing a parent scope. At a high level, the only difference is that `VariableScope` is written in C++ while JRuby's `DynamicScope` object is written in Java. Of course, at a more detailed level the two implementations are very different.

Like JRuby, Rubinius stores the outer `str` variable in an instance of the `VariableScope` class and later creates a second `VariableScope` object when executing the code inside the block. The two `VariableScope` objects are connected

with a `parent` pointer. When the Rubinius VM executes the `push_local_depth` instruction from inside the block, it follows the `parent` pointer up to obtain the value of `str` from the outer scope.

The most important and impressive feature of Rubinius is that it does implement many of the methods in Ruby's core classes, such as `Integer.times`, in pure Ruby code. This means you can take a look right inside of Rubinius yourself to learn how something works. For example, here's Rubinius's implementation of the `Integer.times` method, taken from the kernel/common/integer.rb source code file:

```
def times
  return to_enum(:times) unless block_given?
  i = 0
  while i < self
    yield i
    i += 1
  end
  self
end
```

On the first line, Rubinius calls `to_enum` to return an enumerator object if a block is not given. But in the most common case when you do provide a block you want to iterate over, as in my `10.times` example, Rubinius uses a simple Ruby while-loop that calls `yield` each time through the loop. This is exactly how you or I would probably implement the `Integer.times` method if Ruby didn't provide it for us.

Conclusion

This is the end of my journey through Ruby internals. I hope I've given you some sense of how Ruby works, of what happens when you type "ruby my_awesome_script.rb" and press ENTER. The next time you include a module in a class, call a block or use the `break` keyword to jump back up the call stack, I hope some of the diagrams you've seen here come to mind.

More to come?

As I said in the preface, there are many, many areas of Ruby's internal implementation I didn't have time to cover in this book. For starters, except for hash tables I didn't describe how Ruby's core classes such as arrays, strings, files or integers work. I also didn't explore garbage collection, the regular expression engine, threads, or the extension API, just to name a few topics.

This year in *Ruby Under a Microscope* I've tried to cover the core of the language implementation. I've also covered the topics I was most interested in. Someday I may try to write a second book, *Ruby Under a Microscope - Part 2*, that would cover portions of Ruby's implementation I didn't have time for here.

Feedback please

I hope you found this useful, that you've gained a deeper understanding of Ruby and become a more knowledgable Ruby developer. Please let me know what you think - I plan to post an updated version of the text with technical corrections and updates. I'd also love to hear what you think should be covered in *Ruby Under a Microscope - Part 2* if I ever write it.

Please send feedback to:

- http://patshaughnessy.net/ruby-under-a-microscope#disqus_thread
- Twitter: @pat_shaughnessy
- Email: pat@patshaughnessy.net
- https://github.com/patshaughnessy/ruby-under-a-microscope/issues