The Structure and Interpretation of Ruby Programs

[](https://buildingvts.com/@DeanNasseri?source=post_page-----362db0412f29--------------------------------)

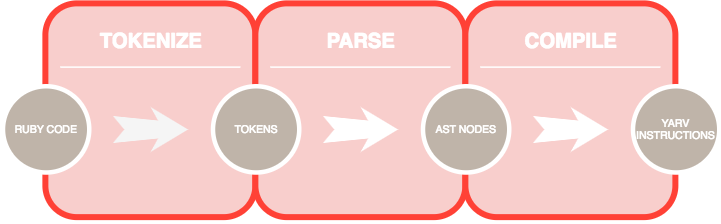
[Dean Nasseri](https://buildingvts.com/@DeanNasseri?source=post_page-----362db0412f29--------------------------------)

[Follow](https://medium.com/m/signin?actionUrl=%2F_%2Fsubscribe%2Fuser%2F72f0167b79d0&operation=register&redirect=https%3A%2F%2Fbuildingvts.com%2Fthe-structure-and-interpretation-of-ruby-programs-362db0412f29&source=-72f0167b79d0-------------------------follow_byline-----------)

[Nov 5, 2017](https://buildingvts.com/the-structure-and-interpretation-of-ruby-programs-362db0412f29?source=post_page-----362db0412f29--------------------------------) · 5 min read

Whenever a ruby program is run, the program is first lexed into tokens, then the tokens are assembled into an abstract syntax tree, and finally the AST is compiled into virtual machine instructions. In this post, we will explore each of these steps in detail.

Image for post



First, an example program

class Math  
 def add(x, y)  
 x + y  
 end  
end   
  
Math.new.add(1, 2)

Whenever you run a ruby program, ruby steps through the characters in the program one at a time and groups them into special words called “tokens”. These tokens are not guaranteed to be valid Ruby. At this point the stream of tokens could be invalid. It is the responsibility of the parser to determine whether the inputted program is valid or not.

We can actually see how Ruby’s tokenization would treat this program using the built in tool ‘ripper’. Lets take a look at how the definition of the add method is tokenized using the following code:

require 'ripper'  
require 'pp'  
  
code = <<CODE  
def add(x, y)  
 x + y  
end  
CODE  
  
pp Ripper.lex(code)

This results in the following output:

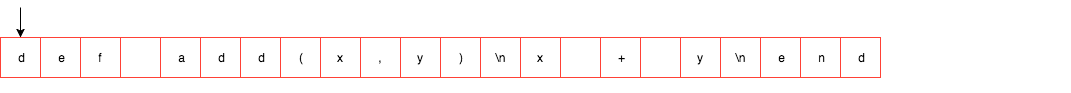
[[[1, 0], :on\_kw, "def"],  
 [[1, 3], :on\_sp, " "],  
 [[1, 4], :on\_ident, "add"],  
 [[1, 7], :on\_lparen, "("],  
 [[1, 8], :on\_ident, "x"],  
 [[1, 9], :on\_comma, ","],  
 [[1, 10], :on\_sp, " "],  
 [[1, 11], :on\_ident, "y"],  
 [[1, 12], :on\_rparen, ")"],  
 [[1, 13], :on\_ignored\_nl, "\n"],  
 [[2, 0], :on\_sp, " "],  
 [[2, 2], :on\_ident, "x"],  
 [[2, 3], :on\_sp, " "],  
 [[2, 4], :on\_op, "+"],  
 [[2, 5], :on\_sp, " "],  
 [[2, 6], :on\_ident, "y"],  
 [[2, 7], :on\_nl, "\n"],  
 [[3, 0], :on\_kw, "end"],  
 [[3, 3], :on\_nl, "\n"]]

Each nested array in above output represents a single token. The structure of the array is as follows:

[[line number, text column], token name, characters]

The token name symbols in the output map to token types defined inside of the file “parse.y” of the ruby source code. Though the symbol names are not one to one with the C definition inside parse.y, the output of ripper captures what Ruby does internally as it encounters tokens. For instance “def” is mapped to :on\_kw indicating that it is ruby keyword. Similarly “end” is also mapped to :on\_kw. However inside of ruby “def” would be treated as the token of type “keyword\_def”, and “end” would be treated as the token of type “keyword\_end”. There are some other differences in the output of ripper and the internal representation of each token, but ripper gives us a good idea of how ruby tokenization works. It is important to remember that tokenization happens in a character by character streaming fashion. This gif should help visualize how a stream of characters is turned into a stream of tokens:

Image for post



Ruby tokenization in action

**Parsing**

After ruby has finished converting the text of the program into a stream of tokens, the tokens are then grouped into logical units that ruby can understand. This is the parsing step, and it is at this step that the program is determined to be valid ruby or not. Ruby does this by determining if the stream of tokens conform the grammar rules defined in the parse.y file. If so, the stream of tokens are converted into a corresponding abstract syntax tree. The AST is a data structure representation of the syntactical meaning of your ruby program. The handy ripper tool can again help us visualize what this AST looks like to ruby, this time using the “sexp” function:

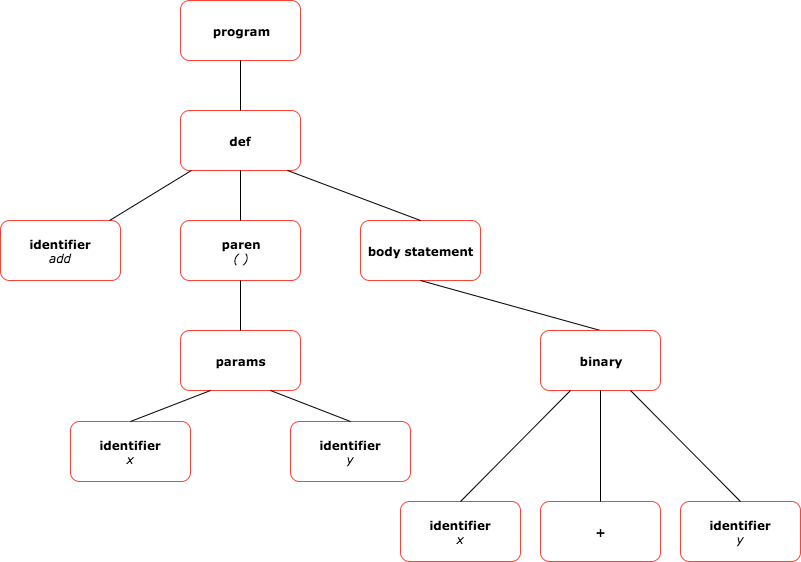
require 'ripper'  
require 'pp'code = <<CODE  
def add(x, y)  
 x + y  
end  
CODEpp Ripper.sexp(code)

Output:

[:program,  
 [[:def,  
 [:@ident, "add", [1, 4]],  
 [:paren,  
 [:params,  
 [[:@ident, "x", [1, 8]], [:@ident, "y", [1, 11]]],  
 nil,  
 nil,  
 nil,  
 nil,  
 nil,  
 nil]],  
 [:bodystmt,  
 [[:binary,  
 [:var\_ref, [:@ident, "x", [2, 2]]],  
 :+,  
 [:var\_ref, [:@ident, "y", [2, 6]]]]],  
 nil,  
 nil,  
 nil]]]]

This output is a bit terse, so I’ve included a visual reprsenation of the AST below. Each node in the tree represents a construct in the program. For instance, the addition of the identifier’s x and y is represented by a “binary” node. A binary node encodes an operation that performs on two elements of a set to derive a third element.

Image for post



The structure of the AST

The generated AST encapsulates everything ruby needs to understand your program, and with that, ruby can move on to the third and final step.

**Compilation**

Ruby is a compiled language in much the same way that Java is. While ruby is not compiled down to native machine code, it is compiled into a set of bytecode instructions that are interpreted by a virtual machine. In the case of Java the VM is JVM, in the case of Ruby it is YARV, which stands for “Yet another ruby virtual-machine”.

In order to compile your program, ruby recursively iterates over the nodes in the AST from the top down and compiles each node into corresponding YARV instructions. Once more, we can use built in tools to examine how ruby compiles our AST into YARV instructions.

code = <<CODE  
def add(x, y)  
 x + y  
end  
CODEputs RubyVM::InstructionSequence.compile(code).disasm

Output:

== disasm: #@>================================  
0000 trace 1 ( 1)  
0002 putspecialobject 1  
0004 putobject :add  
0006 putiseq add  
0008 opt\_send\_without\_block ,   
0011 leave  
== disasm: #>=======================================  
local table (size: 2, argc: 2 [opts: 0, rest: -1, post: 0, block: -1, kw: -1@-1, kwrest: -1])  
[ 2] x [ 1] y  
0000 trace 8 ( 1)  
0002 trace 1 ( 2)  
0004 getlocal\_OP\_\_WC\_\_0 4  
0006 getlocal\_OP\_\_WC\_\_0 3  
0008 opt\_plus ,   
0011 trace 16 ( 3)  
0013 leave ( 2)

YARV is a stack oriented virtual machine, so most of the instructions invlove putting an object onto the stack, and then executing an operation against the values on the stack. The top block of instructions are used to define the “add” method. Essentially the instructions put the method name on the stack, and then calls “define\_method” a C function that is used by YARV to create a new ruby method. In the second block, a local table is defined which represent the arguments our function can accept.

**Conclusion**

In this post we explored how ruby translates the text of a program, first into tokens, then into a structure called an AST, and finally into intstructions usable by the virtual machine. These three passes are what allow ruby to be interpreted by the virtual machine. Hopefully after reading this post you will have a little bit better understanding about what exactly happens when you boot up a ruby process.

This blog post was originally published on my blog here: <https://www.nasseri.io/>

The company I work for, VTS, is always on the lookout for talented people who want to be a part of what we’re building. You can find out more about us, and see all our open roles, on our [careers page](https://www.vts.com/careers).

Just-In-Time for Ruby 2.6, an explanation of compiled and interpreted languages

[](https://medium.com/@mich_berr?source=post_page-----4fd021e7a58--------------------------------)

[Michelle Berry](https://medium.com/@mich_berr?source=post_page-----4fd021e7a58--------------------------------)

Follow

[Jan 6, 2019](https://medium.com/@mich_berr/just-in-time-for-ruby-2-6-an-explanation-of-compiled-and-interpreted-languages-4fd021e7a58?source=post_page-----4fd021e7a58--------------------------------) · 7 min read

Image for post

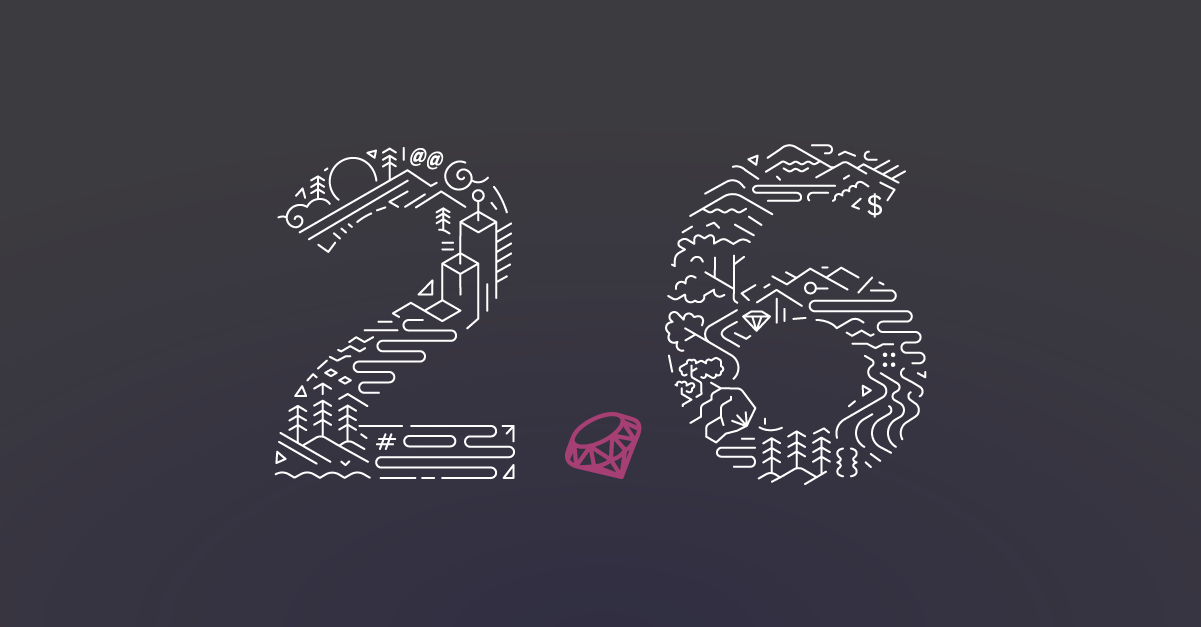


Image from <https://blog.heroku.com/ruby-just-in-time-compilation>

Ruby 2.6 was [released](https://www.ruby-lang.org/en/news/2018/12/25/ruby-2-6-0-released/) a couple of weeks ago, featuring a brand new Just-In-Time (JIT) compiler . You can read up on the details of this new feature [here](https://medium.com/square-corner-blog/rubys-new-jit-91a5c864dd10), but if you’re looking for more background, I thought it would be helpful to give an introduction to the differences between interpreted and compiled languages and how JIT fits in.

First, what is a programming language? All languages are abstractions of **machine code**. Machine code is composed of bits — 0s and 1s — that can be stored and operated on by your machine’s hardware. It would be very inefficient for a human to read and write binary, which is why we invented programming languages.

The difference between interpretation and compilation lies in how a program written by a human is converted into executable instructions. It turns out that languages are *not* explicitly compiled or interpreted; compilation and interpretation are two different strategies that can be used to translate the code you write into the code a machine reads. As we’ll see when we get to the discussion of JIT, the lines between compilation and interpretation are becoming increasingly blurred.

**Compilation**

Put simply, compilation is the translation of a higher-level programming language into a machine’s language.

Let’s use C as an example of a language that is typically compiled. To run a C program, one must use compiler software, like gcc or clang, to compile the C source code into the appropriate machine code for your computer. Importantly, different computers have different**CPU architectures,**meaning that one computer handles the computation of 0s and 1s differently than another. A compiler translates source code into machine code for a specific architecture. Once your code is compiled, it can be run as many times as you’d like on any system with the same architecture, but if you update the source code, or want to run your program on a machine with different architecture, you will need to recompile.

A good analogy for a compiler is a human language translator. Similar to how a translator would translate a Spanish book into English, a compiler translates source code written by a human into machine code. Once a book has been translated, any English speaker can read it. However, if there are changes to the Spanish version, the English version will have to get retranslated and republished as well.

**Interpretation**

Unlike compilers which pre-translate source code into machine code before a program runs, interpreters translate code as they are executing it, line-by-line. Continuing with the previous analogy, computer interpreters are like, well, interpreters. They serve as the interface between a Spanish and English speaker interacting in real time, interpreting the language sentence by sentence.

Let’s use Ruby as an example of an “interpreted” language. Ruby 1.8 and earlier versions utilized Ruby utilized Matz’s Ruby Interpreter (MRI), which behaved as described above. It read in each line of Ruby, parsed and tokenized it, and then used a tree-based data structure to execute it. However, starting in version 1.9 (ca. 2011) Ruby switched to an implementation called YARV (Yet Another Ruby Virtual Machine). In this implementation, Ruby is pre-compiled into a simpler form consisting of **bytecodes**, named because they occupy 1 byte of memory. As a very simplified example, 2 + 3 would be converted to the bytecode for addition and would accept 2 and 3 as arguments. Once Ruby is converted to bytecode, a virtual machine executes the code line by line. Converting source code to bytecode offers significant speed advantages.

Python also utilizes bytecode, and in Python you can directly observe it in the .pyc files your program spits out. These files function like a cache; if the Python program is run again without changes, it can skip the compilation step and go straight to execution. While Ruby is also compiled to bytecode, its bytecode is only stored in memory rather than printed to a file.

**Just-In-Time compilation**

I just described how some contemporary interpreted languages now involve a bytecode compilation step, but there’s yet another way that compilation and interpretation are starting to blur, called Just-In-Time compilation.

The most popular Just-In-Time compiler is the **Java Virtual Machine (JVM).**Java is a statically typed language that can be directly compiled to machine code, but is commonly interpreted via the JVM. In the latter case, Java is compiled to Java bytecode by the Javac compiler, and then interpreted by the JVM. However, the JVM doesn’t interpret the bytecode line-by-line. Instead it tries to compose meaningful chunks of code, such as functions, all at the same time. It takes slightly longer to determine how to compose these chunks of code, but the execution is more efficient. This process is called Just-In-Time compilation, because it behaves like a compiler would, except it compiles *at runtime*. The benefit of using the JVM is that it retains some of the performance features of compiled languages, while making Java portable to different machines as an interpreted language. Java is the most popular language in the world, in large part because of the success of the JVM.

In addition to the JVM, there are other VM projects that make use of JIT compilation. [PyPy](https://pypy.org/) is an example of a python interpreter with JIT, and of course Ruby now ships with an optional JIT feature.

**Tradeoffs**

Now that you have an introduction to Compiled vs Interpreted languages, what are the tradeoffs to each?

**Speed**

Compilation is generally much faster than interpretation. A compiled C program might execute a couple orders of magnitude faster than an interpreted program in Python or Ruby. However, Java’s hybrid JIT solution is quite efficient and runs can be nearly as efficient as a compiled program written in C.

**Portability**

In order to run your compiled program on a different machine’s architecture, you would need to recompile it. When a language is interpreted, it ships with it’s own instruction set that handles the specifications of your machine’s hardware architecture. The JVM is a good example of a technology that combines interpretation and compilation to maximize both the speed and portability of its language.

**Dynamic vs Static typing**

Since compilers have to translate and compose their programs directly into machine instructions, they are much more rigid. When declaring a variable, the compiler needs to know exactly what kind of variable it is and how much memory to allocate. This is why compilation generally requires static typing. In contrast, interpreters execute their programs line by line, so they can behave more flexibly.

In ruby, we can run write something like 2 + 3 or "a" + "b" and the interpreter will determine the type of the objects at runtime and call the right method for either integer addition or string concatenation.

**Bugs/Debugging**

Debugging is often easier with interpretation, because the program will execute until it encounters the error. An interpreter will notify the user exactly which line caused the runtime error, whereas bugs in compiled programs can be much more cryptic.

**FAQs**

*Why is referring to a language as “interpreted” or “compiled” a misnomer?*

Languages are defined by their syntax and data structures. Compilation and interpretation are two examples of implementations, e.g. a process to convert language syntax to a form that can be run on hardware. Languages are not inherently “interpreted” or “compiled”; the same language can be implemented using either approach.

*Why then do people refer to Python as an “interpreted language” and C as a “compiled language”?*

They are referring to the most common implementation/distribution of the language. However, Python can also be compiled and C can also be interpreted!

*What is a virtual machine?*

A virtual machine is anything that behaves abstractly like a computer, meaning it accepts as input a series of instructions, but is implemented through software as opposed to hardware. One common use of virtual machines is to run the OS of another system on a computer. For example, you own a windows laptop, but you want to emulate a linux operating system. You would utilize a software program that can covert between the OS of the system you want to run and the architecture of your physical computer. This is called a “system virtual machine”.

Not to be confused with the previous example, there are also “process virtual machines” e.g. the Java’s virtual machine (JVM) or Ruby’s virtual machine (YARV). These would be considered virtual machines because they accept an instruction set for executing bytecodes. The advantage of these VM’s are that they abstract away the hardware and provide a platform-independent programming environment.

*Why do people generally refer to Python and Ruby as having interpreters whereas Java has a virtual machine? Doesn’t an interpreter also fit the definition of a virtual machine?*

This distinction between an interpreter and virtual machine is mostly semantic or a “social construct” as this [person](https://news.ycombinator.com/item?id=8559085) phrased it. I think an interpreter fits the definition of a virtual machine. Even with a more constrictive definition, both Ruby and Python’s interpreters include a virtual machine that interprets bytecodes. Definitions aside, the JVM is fundamentally different from either the Ruby or Python interpreter. I’ve touched on a few of the reasons here, but the rest are beyond the scope of this article.

*Why does compiling a language to bytecode before interpretation make it faster?*

Bytecodes take up less memory than the full source code and are easier for the interpreter to execute. Indeed, when a language is compiled to bytecode, the interpreter has to parse everything once and convert it to bytecode, then reparse the bytecode to execute it. For simple pieces of code, this intermediate step does increase the total execution time by a small amount. However, for code that is executed repeatedly, think loops or reused functions, the bytecode step adds a significant speed advantage.

**Further Resources:**

**BaseCS**, a great blog on fundamental programming concepts: <https://medium.com/basecs/a-deeper-inspection-into-compilation-and-interpretation-d98952ebc842>

**Ruby Under a Microscope**, a great in-depth book on Ruby’s internals, accessible to programmers with limited C exposure: <http://patshaughnessy.net/ruby-under-a-microscope>

**Computer Architecture course at Bradfield Academy,** superior in-person course on computer architecture concepts: <https://bradfieldcs.com/>

[Tyler Elliot Bettilyon](https://medium.com/u/7147db7866ab?source=post_page-----4fd021e7a58--------------------------------)’s **youtube video**, Tyler was my instructor at Bradfield and one of the best humans on the planet at explaining CS concepts https://www.youtube.com/watch?v=KsZLPTRSleI

**How Ruby Interprets and Runs Your Programs**

In this post we'll follow the journey of a simple program as it's lexed, parsed and compiled into bytecode. We'll use the tools that Ruby gives us to spy on the interpreter every step of the way.

*  By [Starr Horne](https://www.honeybadger.io/blog/how-ruby-interprets-and-runs-your-programs/#authorDetails) [Author Twitter](https://twitter.com/starrhorne)

* [#ruby](https://www.honeybadger.io/blog/tags/ruby)

* Nov 3, 2015

The more you know about your tools, the better decisions you will make as a developer. It's often useful — especially when debugging performance issues — to understand what Ruby is actually doing when it runs your program.

In this post we'll follow the journey of a simple program as it's lexed, parsed and compiled into bytecode. We'll use the tools that Ruby gives us to spy on the interpreter every step of the way.

Don't worry — even if you're not an expert this post should be pretty easy to follow. It's more of a guided tour than a technical manual.

**Meet our sample program**

As an example, I'm going to use a single if/else statement. To save space, I'll write this using the ternary operator. But don't be fooled, it's just an if/else.

x > **100** ? 'foo' : 'bar'

As you'll see, even a simple program like this gets translated into quite a lot of data as it is processed.

Note: All of the examples in this post were written in Ruby (MRI) 2.2. If you're using other implementations of Ruby, they probably won't work.

**Tokenizing**

Before the Ruby interpreter can run your program it has to convert it from a somewhat free-form programming language into more structured data.

The first step might be to break the program into chunks. These chunks are called tokens.

# This is a string

"x > 1"

# These are tokens

["x", ">", "1"]

The Ruby standard library provides a module called Ripper that lets us process Ruby code in much the same way as the Ruby interpreter.

In the example below we are using the tokenize method on our Ruby code. As you can see, it returns an array of tokens.

require 'ripper'

**Ripper**.**tokenize**("x > 1 ? 'foo' : 'bar'")

# => ["x", " ", ">", " ", "1", " ", "?", " ", "'", "foo", "'", " ", ":", " ", "'", "bar", "'"]

The tokenizer is pretty stupid. You can feed it completely invalid Ruby and it will still tokenize it.

# bad code

**Ripper**.**tokenize**("1var @= **\/**foobar`")

# => ["1", "var"]

**Lexing**

Lexing is one step beyond tokenization. The string is still broken into tokens, but additional data is added to the tokens.

In the example below we are using Ripper to Lex our small program. as you can see, it's now tagging each token as being an identifier *:on\_ident*, an operator *:on\_op*, an integer *:on\_int*, etc.

require 'ripper'

require 'pp'

pp **Ripper**.**lex**("x > 100 ? 'foo' : 'bar'")

# [[[1, 0], :on\_ident, "x"],

# [[1, 1], :on\_sp, " "],

# [[1, 2], :on\_op, ">"],

# [[1, 3], :on\_sp, " "],

# [[1, 4], :on\_int, "100"],

# [[1, 5], :on\_sp, " "],

# [[1, 6], :on\_op, "?"],

# [[1, 7], :on\_sp, " "],

# [[1, 8], :on\_tstring\_beg, "'"],

# [[1, 9], :on\_tstring\_content, "foo"],

# [[1, 12], :on\_tstring\_end, "'"],

# [[1, 13], :on\_sp, " "],

# [[1, 14], :on\_op, ":"],

# [[1, 15], :on\_sp, " "],

# [[1, 16], :on\_tstring\_beg, "'"],

# [[1, 17], :on\_tstring\_content, "bar"],

# [[1, 20], :on\_tstring\_end, "'"]]

There is still no real syntax checking going on at this point. The lexer will happily process invalid code.

**Parsing**

Now that Ruby has broken up the code into more manageable chunks, it's time for parsing to begin.

During the parsing stage, Ruby transforms the text into something called an abstract syntax tree, or AST. The abstract syntax tree is a representation of your program in memory.

You might say that programming languages in general are just more user-friendly ways of describing abstract syntax trees.

require 'ripper'

require 'pp'

pp **Ripper**.**sexp**("x > 100 ? 'foo' : 'bar'")

# [:program,

# [[:ifop,

# [:binary, [:vcall, [:@ident, "x", [1, 0]]], :>, [:@int, "100", [1, 4]]],

# [:string\_literal, [:string\_content, [:@tstring\_content, "foo", [1, 11]]]],

# [:string\_literal, [:string\_content, [:@tstring\_content, "foobar", [1, 19]]]]]]]

It might not be easy to read this output, but if you stare at it for long enough you can kind of see how it maps to the original program.

# Define a progam

[:program,

# Do an "if" operation

[[:ifop,

# Check the conditional (x > 100)

[:binary, [:vcall, [:@ident, "x", [**1**, **0**]]], :>, [:@int, "100", [**1**, **4**]]],

# If true, return "foo"

[:string\_literal, [:string\_content, [:@tstring\_content, "foo", [**1**, **11**]]]],

# If false, return "bar"

[:string\_literal, [:string\_content, [:@tstring\_content, "foobar", [**1**, **19**]]]]]]]

At this point, the Ruby interpreter knows exactly what's you want it to do. It could run your program right now. And before Ruby 1.9, it would have. But now, there's one more step.

**Compiling to bytecode**

Instead of traversing the abstract syntax tree directly, nowadays Ruby compiles the abstract syntax tree into lower-level byte code.

This byte code is then run by the Ruby virtual machine.

We can take a peek into the inner workings of the virtual machine via the *RubyVM::InstructionSequence* class. In the example below, we compile our sample program and then disassemble it to make a human readable.

puts **RubyVM**::**InstructionSequence**.**compile**("x > 100 ? 'foo' : 'bar'").**disassemble**

# == disasm: <RubyVM::InstructionSequence:<compiled>@<compiled>>==========

# 0000 trace 1 ( 1)

# 0002 putself

# 0003 opt\_send\_without\_block <callinfo!mid:x, argc:0, FCALL|VCALL|ARGS\_SIMPLE>

# 0005 putobject 100

# 0007 opt\_gt <callinfo!mid:>, argc:1, ARGS\_SIMPLE>

# 0009 branchunless 15

# 0011 putstring "foo"

# 0013 leave

# 0014 pop

# 0015 putstring "bar"

# 0017 leave

Whoa! This suddenly looks a lot more like assembly language than Ruby. Let's step through it and see if we can make sense of it.

# Call the method `x` on self and save the result on the stack

**0002** putself

**0003** opt\_send\_without\_block <callinfo!mid:x, argc:**0**, **FCALL**|**VCALL**|**ARGS\_SIMPLE**>

# Put the number 100 on the stack

**0005** putobject **100**

# Do the comparison (x > 100)

**0007** opt\_gt <callinfo!mid:>, argc:**1**, **ARGS\_SIMPLE**>

# If the comparison was false, go to line 15

**0009** branchunless **15**

# If the comparison was true, return "foo"

**0011** putstring "foo"

**0013** leave

**0014** pop

# Here's line 15. We jumped here if comparison was false. Return "bar"

**0015** putstring "bar"

**0017** leave

The ruby virtual machine (YARV) then steps through these instructions and executes them. That's it!

**Conclusion**

This ends our very simplified, cartoony tour of the Ruby interpreter. With the tools I've shown you here, it's possible to take a lot of the guesswork out of how Ruby is interpreting your programs. I mean, it doesn't get more concrete than an AST. And next time you're stumped by some weird performance issue, try looking at the bytecode. It probably won't solve your problem, but it might take your mind off of it. :)