Hi, I’m Ian. I’m here to show that there’s never been a better time than now to write your own compiler.
Why would someone want to do this? That depends on the direction you’re coming from.
If, like me, you came from a hardware background, compilers are the next logical step up the ladder of abstraction from programming.
If you come from a computer science background, compilers and parsers are practical tools to get things done (such as reading crufty ad hoc data formats at work).
If you’re following the self–made path, compilers are one of many computer science topics you may come to naturally on your travels.
By the end of this talk, we’ll have seen the basics of how to use JRuby to create a compiler for a fictional programming language.
Since all the good letters for programming languages are taken (C, D, K, R, etc.), let’s use a fictional letter. “Thnad,” a letter that comes to us from Dr. Seuss, will do nicely.
Here’s a simple Thnad program. As you can see, it has only integers, functions, and conditionals. No variables, type definitions, or anything else. But this bare minimum will keep us plenty busy for the next half hour.
Before we jump into the code, let’s look at the general view of where we’re headed.
We need to take a piece of program text like “minus(n, 1)” and break it down into something like a sentence’s parts of speech: this is a function call, this is a parameter, and so on. The code in the middle is how we might represent this breakdown in Ruby, but really we should be thinking about it graphically, like the tree at the bottom.
The representation on the previous slide used plain arrays, strings, and so on. It will be helpful to transform these into custom Ruby objects that know how to emit JVM bytecode. Here's the kind of thing we'll be aiming for. Notice that the tree looks really similar to the one we just saw—the only differences are the names in the nodes.
1. parse
2. transform
3. emit

Each stage of our compiler will transform one of our program representations—original program text, generic Ruby objects, custom Ruby objects, JVM bytecode—into the next.
First, let’s look at parsing the original program text into generic Ruby objects.
describe Thnad::Parser do
  before do
    @parser = Thnad::Parser.new
  end

  it 'reads a number' do
    expected = { :number => '42' }
    @parser.number.parse('42').must_equal expected
  end
end

Here’s a unit test written with minitest, a unit testing framework that ships with Ruby 1.9 (and therefore JRuby). We want the text “42” to be parsed into the Ruby hash shown here.
How are we going to parse our programming language into that internal representation? By using Parslet, a parsing tool written in Ruby. For the curious, Parslet is a PEG (Parsing Expression Grammar) parser.
require 'parslet'

module Thnad

class Parser < Parslet::Parser
  # gotta see it all
  rule(:space)  { match('\s').repeat(1) }
  rule(:space?) { space.maybe }

  # numeric values
  rule(:number) { match('[0-9]').repeat(1).as(:number) >> space? }
end
end

Here’s a set of Parslet rules that will get our unit test to pass. Notice that the rules read a little bit like a mathematical notation: “a number is a sequence of one or more digits, possibly followed by trailing whitespace.”
The next step is to transform the Ruby arrays and hashes into more sophisticated objects that will (eventually) be able to generate bytecode.
Here’s a unit test for that behavior. We start with the result of the previous step (a Ruby hash) and expect a custom Number object.

```ruby
input = { :number => '42' }
expected = Thnad::Number.new(42)

@transform.apply(input).must_equal expected
```
How are we going to transform the data? We’ll use Parslet again.
This rule takes any simple Ruby hash with a :number key and transforms it into a Number object.
require 'parslet'

module Thnad
  class Number < Struct.new(:value)
    # more to come...
  end
end

Of course, we’ll need to define the Number class. Once we’ve done so, the unit tests will pass.
Finally, we need to emit JVM bytecode.
bytecode

ldc 42

Here’s the bytecode we want to generate when our compiler encounters a Number object with 42 as
the value. It just pushes the value right onto the stack.
describe 'Emit' do
  before do
    @builder = mock
    @context = Hash.new
  end

  it 'emits a number' do
    input = Thnad::Number.new 42
    @builder.expects(:ldc).with(42)
    input.eval @context, @builder
  end
end

And here’s the unit test that specifies this behavior. We’re using mock objects to say in effect, “Imagine a Ruby object that can generate bytecode; our compiler should call it like this.”
Our Number class will now need an “eval” method that takes a context (useful for carrying around information such as parameter names) and a bytecode builder (which we’ll get to on the next slide).
A moment ago, we saw that we’ll need a Ruby object that knows how to emit JVM bytecode. The BiteScript library for JRuby will give us just such an object.

tool #3: BiteScript

https://github.com/headius/bitescript
This, for example, is a chunk of BiteScript that writes a Java .class file containing a function call—in this case, the equivalent of the “minus(n, 1)” Thnad code we saw earlier.
If you run that through the BiteScript compiler and then use the standard Java tools to view the resulting bytecode, you can see it’s essentially the same program.
enough hand-waving
let’s see the code!

Armed with this background information, we can now jump into the code.
At this point in the talk, I fired up TextMate and navigated through the project, choosing the direction based on whim and audience questions. You can follow along at home by looking at the various commits made to this project on GitHub.

a copy of our home game

https://github.com/undees/thnad
There are two resources I found helpful during this project. The first was Charles Nutter’s primer on Java bytecode, which was the only JVM reference material necessary to write this compiler.
The second helpful resource was Marc-André Cournoyer’s e-book *How to Create Your Own Freaking Awesome Programming Language*. It’s geared more toward interpreters than compilers, but was a great way to check my progress after I’d completed the major stages of this project.
My hope for us as a group today is that, after our time together, you feel it might be worth giving one of these tools a shot—either for a fun side project, or to solve some protocol parsing need you may encounter one day in your work.