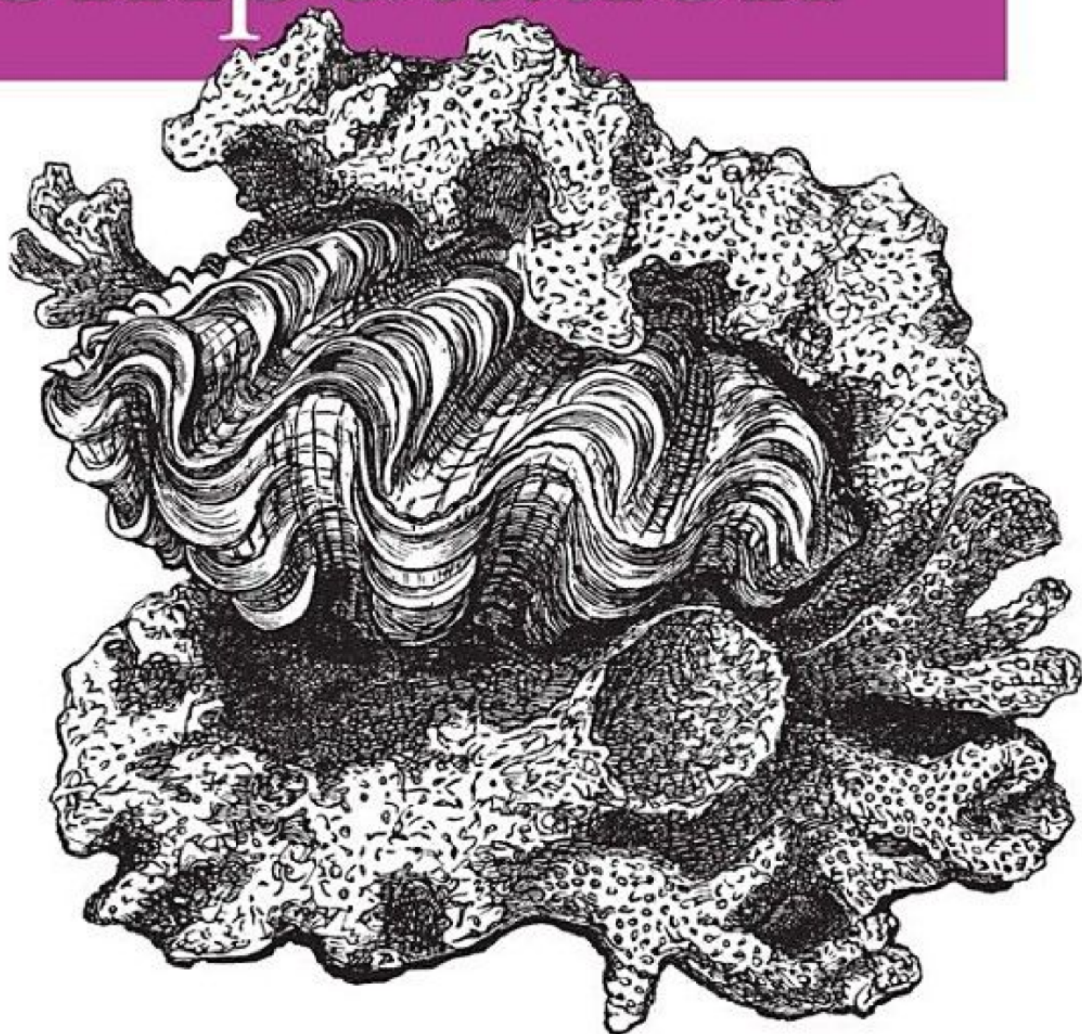
From Simple Machines to Impossible Programs

Understanding Computation



O'REILLY®

Tom Stuart

Copyrighted material

Understanding Computation

Tom Stuart

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Copyrighted material

Preface

Who Should Read This Book?

This book is for programmers who are curious about programming languages and the theory of computation, especially those who don't have a formal background in mathematics or computer science.

If you're interested in the mind-expanding parts of computer science that deal with programs, languages, and machines, but are discouraged by the mathematical language that's often used to explain them, this book is for you. Instead of complex notation we'll use working code to illustrate theoretical ideas and turn them into interactive experiments that you can explore at your own pace.

This book assumes that you know at least one modern programming language like Ruby, Python, JavaScript, Java, or C#. All of the code examples are in Ruby, but if you know another language you should still be able to follow along. However, this book *isn't* a guide to best practices in Ruby or object-oriented design. The code is intended to be clear and concise, but not necessarily to be easy to maintain; the goal is always to use Ruby to illustrate the computer science, not vice versa. It's also not a textbook or an encyclopedia, so instead of presenting formal arguments or watertight proofs, this book tries to break the ice on some interesting ideas and inspire you to learn about them in more depth.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`

Copyrighted material

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`

Shows commands or other text that should be typed literally by the user.

`Constant width italic`

Shows text that should be replaced with user-supplied values or by values determined by context.

> **TIP**
>
> This icon signifies a tip, suggestion, or general note.

> **CAUTION**
>
> This icon indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, if this book includes code examples, you may use the code in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes

Copyrighted material

the title, author, publisher, and ISBN. For example: "*Understanding Computation* by Tom Stuart (O'Reilly). Copyright 2013 Tom Stuart, 978-1-4493-2927-3."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

Safari® Books Online

> **NOTE**
>
> Safari Books Online (www.safaribooksonline.com) is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of product mixes and pricing programs for organizations, government agencies, and individuals. Subscribers have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and dozens more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Copyrighted material

Sebastopol, CA 95472

800-998-9938 (in the United States or Canada)

707-829-0515 (international or local)

707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at http://oreil.ly/understanding-computation.

To comment or ask technical questions about this book, send email to bookquestions@oreilly.com.

For more information about our books, courses, conferences, and news, see our website at http://www.oreilly.com.

Find us on Facebook: http://facebook.com/oreilly

Follow us on Twitter: http://twitter.com/oreillymedia

Watch us on YouTube: http://www.youtube.com/oreillymedia

Acknowledgments

I'm grateful for the hospitality of Go Free Range, who provided me with office space, friendly conversation, and tea throughout the writing of this book. Without their generous support, I'd definitely have gone a bit Jack Torrance.

Thank you to James Adam, Paul Battley, James Coglan, Peter Fletcher, Chris Lowis, and Murray Steele for their feedback on early drafts, and to Gabriel Kerneis and Alex Stangl for their technical reviews. This book has been immeasurably improved by their thoughtful contributions. I'd also like to thank Alan Mycroft from the University of Cambridge for all the knowledge and encouragement he supplied.

Many people from O'Reilly helped shepherd this project to completion, but I'm especially grateful to Mike Loukides and Simon St.Laurent for their early enthusiasm and faith in the idea, to Nathan Jepson for his advice on how to turn the idea into an actual book, and to Sanders Kleinfeld for humoring my

Copyrighted material

relentless quest for perfect syntax highlighting.

Thank you to my parents for giving an annoying child the means, motive, and opportunity to spend all his time mucking about with computers; and to Leila, for patiently reminding me, every time I forgot how the job should be done, to keep putting one damn word after another. I got there in the end.

Copyrighted material

Chapter 1. Just Enough Ruby

The code in this book is written in Ruby, a programming language that was designed to be simple, friendly, and fun. I've chosen it because of its clarity and flexibility, but nothing in the book relies on special features of Ruby, so you should be able to translate the code examples into whatever language you prefer—especially another dynamic language like Python or JavaScript—if that helps to make the ideas clearer.

All of the example code is compatible with both Ruby 2.0 and Ruby 1.9. You can find out more about Ruby, and download an official implementation, at the official Ruby website.

Let's take a quick tour of Ruby's features. We'll concentrate on the parts of the language that are used in this book; if you want to learn more, O'Reilly's *The Ruby Programming Language* is a good place to start.

> **NOTE**
>
> If you already know Ruby, you can safely skip to Chapter 2 without missing anything.

Interactive Ruby Shell

One of Ruby's friendliest features is its interactive console, *IRB*, which lets us enter pieces of Ruby code and immediately see the results. In this book, we'll use IRB extensively to interact with the code we're writing and explore how it works.

You can run IRB on your development machine by typing **`irb`** at the command line. IRB shows a >> prompt when it expects you to provide a Ruby expression. After you type an expression and hit Enter, the code gets evaluated, and the

Copyrighted material

result is shown at a => prompt:

```
$ irb --simple-prompt
>> 1 + 2
=> 3
>> 'hello world'.length
=> 11
```

Whenever we see these >> and => prompts in the book, we're interacting with IRB. To make longer code listings easier to read, they'll be shown without the prompts, but we'll still assume that the code in these listings has been typed or pasted into IRB. So once the book has shown some Ruby code like this...

```
x = 2
y = 3
z = x + y
```

...then we'll be able to play with its results in IRB:

```
>> x * y * z
=> 30
```

Values

Ruby is an *expression-oriented* language: every valid piece of code produces a value when it's executed. Here's a quick overview of the different kinds of Ruby value.

Basic Data

As we'd expect, Ruby supports Booleans, numbers, and strings, all of which come with the usual operations:

```
>> (true && false) || true
=> true
```

Copyrighted material

```
>> (3 + 3) * (14 / 2)
=> 42
>> 'hello' + ' world'
=> "hello world"
>> 'hello world'.slice(6)
=> "w"
```

A Ruby *symbol* is a lightweight, immutable value representing a name. Symbols are widely used in Ruby as simpler and less memory-intensive alternatives to strings, most often as keys in hashes (see Data Structures). Symbol literals are written with a colon at the beginning:

```
>> :my_symbol
=> :my_symbol
>> :my_symbol == :my_symbol
=> true
>> :my_symbol == :another_symbol
=> false
```

The special value `nil` is used to indicate the absence of any useful value:

```
>> 'hello world'.slice(11)
=> nil
```

Data Structures

Ruby array literals are written as a comma-separated list of values surrounded by square brackets:

```
>> numbers = ['zero', 'one', 'two']
=> ["zero", "one", "two"]
>> numbers[1]
=> "one"
>> numbers.push('three', 'four')
=> ["zero", "one", "two", "three", "four"]
>> numbers
=> ["zero", "one", "two", "three", "four"]
```

Copyrighted material

```
>> numbers.drop(2)
=> ["two", "three", "four"]
```

A *range* represents a collection of values between a minimum and a maximum. Ranges are written by putting a pair of dots between two values:

```
>> ages = 18..30
=> 18..30
>> ages.entries
=> [18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30]
>> ages.include?(25)
=> true
>> ages.include?(33)
=> false
```

A *hash* is a collection in which every value is associated with a key; some programming languages call this data structure a "map," "dictionary," or "associative array." A hash literal is written as a comma-separated list of *`key`* `=>` *`value`* pairs inside curly brackets:

```
>> fruit = { 'a' => 'apple', 'b' => 'banana', 'c' => 'coconut' }
=> {"a"=>"apple", "b"=>"banana", "c"=>"coconut"}
>> fruit['b']
=> "banana"
>> fruit['d'] = 'date'
=> "date"
>> fruit
=> {"a"=>"apple", "b"=>"banana", "c"=>"coconut", "d"=>"date"}
```

Hashes often have symbols as keys, so Ruby provides an alternative *`key`*`:` *`value`* syntax for writing key-value pairs where the key is a symbol. This is more compact than the *`key`* `=>` *`value`* syntax and looks a lot like the popular JSON format for JavaScript objects:

```
>> dimensions = { width: 1000, height: 2250, depth: 250 }
=> {:width=>1000, :height=>2250, :depth=>250}
```

Copyrighted material

```
>> dimensions[:depth]
=> 250
```

Procs

A *proc* is an unevaluated chunk of Ruby code that can be passed around and evaluated on demand; other languages call this an “anonymous function” or “lambda.” There are several ways of writing a proc literal, the most compact of which is the `-> arguments { body }` syntax:

```
>> multiply = -> x, y { x * y }
=> #<Proc (lambda)>
>> multiply.call(6, 9)
=> 54
>> multiply.call(2, 3)
=> 6
```

As well as the `.call` syntax, procs can be called by using square brackets:

```
>> multiply[3, 4]
=> 12
```

Control Flow

Ruby has `if`, `case`, and `while` expressions, which work in the usual way:

```
>> if 2 < 3
     'less'
   else
     'more'
   end
=> "less"
>> quantify =
     -> number {
       case number
       when 1
```

Copyrighted material

```
      'one'
    when 2
      'a couple'
    else
      'many'
    end
  }
=> #<Proc (lambda)>
>> quantify.call(2)
=> "a couple"
>> quantify.call(10)
=> "many"
>> x = 1
=> 1
>> while x < 1000
     x = x * 2
   end
=> nil
>> x
=> 1024
```

Objects and Methods

Ruby looks like other dynamic programming languages but it's unusual in an important way: every value is an *object*, and objects communicate by sending *messages* to each other.[1] Each object has its own collection of *methods* that determine how it responds to particular messages.

A message has a name and, optionally, some arguments. When an object receives a message, its corresponding method is executed with the arguments from the message. This is how all work gets done in Ruby; even `1 + 2` means "send the object `1` a message called `+` with the argument `2`," and the object `1` has a `#+` method for handling that message.

We can define our own methods with the `def` keyword:

```
>> o = Object.new
=> #<Object>
```

Copyrighted material

```
>> def o.add(x, y)
     x + y
   end
=> nil
>> o.add(2, 3)
=> 5
```

Here we're making a new object by sending the new message to a special built-in object called Object; once the new object's been created, we define an #add method on it. The #add method adds its two arguments together and returns the result—an explicit return isn't necessary, because the value of the last expression to be executed in a method is automatically returned. When we send that object the add message with 2 and 3 as arguments, its #add method is executed and we get back the answer we wanted.

We'll usually send a message to an object by writing the receiving object and the message name separated by a dot (e.g., o.add), but Ruby always keeps track of the *current object* (called self) and will allow us to send a message to that object by writing a message name on its own, leaving the receiver implicit. For example, inside a method definition the current object is always the object that received the message that caused the method to execute, so within a particular object's method, we can send other messages to the same object without referring to it explicitly:

```
>> def o.add_twice(x, y)
     add(x, y) + add(x, y)
   end
=> nil
>> o.add_twice(2, 3)
=> 10
```

Notice that we can send the add message to o from within the #add_twice method by writing add(x, y) instead of o.add(x, y), because o is the object that the add_twice message was sent to.

Outside of any method definition, the current object is a special top-level object called main, and any messages that don't specify a receiver are sent to it;

Copyrighted material

similarly, any method definitions that don't specify an object will be made available through `main`:

```
>> def multiply(a, b)
     a * b
   end
=> nil
>> multiply(2, 3)
=> 6
```

Classes and Modules

It's convenient to be able to share method definitions between many objects. In Ruby, we can put method definitions inside a *class*, then create new objects by sending the `new` message to that class. The objects we get back are *instances* of the class and incorporate its methods. For example:

```
>> class Calculator
     def divide(x, y)
       x / y
     end
   end
=> nil
>> c = Calculator.new
=> #<Calculator>
>> c.class
=> Calculator
>> c.divide(10, 2)
=> 5
```

Note that defining a method inside a `class` definition adds the method to instances of that class, not to `main`:

```
>> divide(10, 2)
NoMethodError: undefined method `divide' for main:Object
```

Copyrighted material

One class can bring in another class's method definitions through *inheritance*:

```
>> class MultiplyingCalculator < Calculator
     def multiply(x, y)
       x * y
     end
   end
=> nil
>> mc = MultiplyingCalculator.new
=> #<MultiplyingCalculator>
>> mc.class
=> MultiplyingCalculator
>> mc.class.superclass
=> Calculator
>> mc.multiply(10, 2)
=> 20
>> mc.divide(10, 2)
=> 5
```

A method in a subclass can call a superclass method of the same name by using the super keyword:

```
>> class BinaryMultiplyingCalculator < MultiplyingCalculator
     def multiply(x, y)
       result = super(x, y)
       result.to_s(2)
     end
   end
=> nil
>> bmc = BinaryMultiplyingCalculator.new
=> #<BinaryMultiplyingCalculator>
>> bmc.multiply(10, 2)
=> "10100"
```

Another way of sharing method definitions is to declare them in a *module*, which can then be included by any class:

Copyrighted material

```
>> module Addition
     def add(x, y)
       x + y
     end
   end
=> nil
>> class AddingCalculator
     include Addition
   end
=> AddingCalculator
>> ac = AddingCalculator.new
=> #<AddingCalculator>
>> ac.add(10, 2)
=> 12
```

Miscellaneous Features

Here's a grab bag of useful Ruby features that we'll need for the example code in this book.

Local Variables and Assignment

As we've already seen, Ruby lets us declare local variables just by assigning a value to them:

```
>> greeting = 'hello'
=> "hello"
>> greeting
=> "hello"
```

We can also use *parallel assignment* to assign values to several variables at once by breaking apart an array:

```
>> width, height, depth = [1000, 2250, 250]
=> [1000, 2250, 250]
>> height
=> 2250
```

Copyrighted material

String Interpolation

Strings can be single- or double-quoted. Ruby automatically performs *interpolation* on double-quoted strings, replacing any `#{expression}` with its result:

```
>> "hello #{'dlrow'.reverse}"
=> "hello world"
```

If an interpolated expression returns an object that isn't a string, that object is automatically sent a `to_s` message and is expected to return a string that can be used in its place. We can use this to control how interpolated objects appear:

```
>> o = Object.new
=> #<Object>
>> def o.to_s
     'a new object'
   end
=> nil
>> "here is #{o}"
=> "here is a new object"
```

Inspecting Objects

Something similar happens whenever IRB needs to display an object: the object is sent the `inspect` message and should return a string representation of itself. All objects in Ruby have sensible default implementations of `#inspect`, but by providing our own definition, we can control how an object appears on the console:

```
>> o = Object.new
=> #<Object>
>> def o.inspect
     '[my object]'
   end
```

Copyrighted material

```
=> nil
>> o
=> [my object]
```

Printing Strings

The #puts method is available to every Ruby object (including main), and can be used to print strings to standard output:

```
>> x = 128
=> 128
>> while x < 1000
     puts "x is #{x}"
     x = x * 2
   end
x is 128
x is 256
x is 512
=> nil
```

Variadic Methods

Method definitions can use the * operator to support a variable number of arguments:

```
>> def join_with_commas(*words)
     words.join(', ')
   end
=> nil
>> join_with_commas('one', 'two', 'three')
=> "one, two, three"
```

A method definition can't have more than one variable-length parameter, but normal parameters may appear on either side of it:

```
>> def join_with_commas(before, *words, after)
     before + words.join(', ') + after
```

Copyrighted material

```
  end
=> nil
>> join_with_commas('Testing: ', 'one', 'two', 'three', '.')
=> "Testing: one, two, three."
```

The * operator can also be used to treat each element of an array as a separate argument when sending a message:

```
>> arguments = ['Testing: ', 'one', 'two', 'three', '.']
=> ["Testing: ", "one", "two", "three", "."]
>> join_with_commas(*arguments)
=> "Testing: one, two, three."
```

And finally, * works in parallel assignment too:

```
>> before, *words, after = ['Testing: ', 'one', 'two', 'three', '.']
=> ["Testing: ", "one", "two", "three", "."]
>> before
=> "Testing: "
>> words
=> ["one", "two", "three"]
>> after
=> "."
```

Blocks

A *block* is a piece of Ruby code surrounded by do/end or curly brackets. Methods can take an implicit block argument and call the code in that block with the yield keyword:

```
>> def do_three_times
    yield
    yield
    yield
  end
=> nil
```

Copyrighted material

```
>> do_three_times { puts 'hello' }
hello
hello
hello
=> nil
```

Blocks can take arguments:

```
>> def do_three_times
     yield('first')
     yield('second')
     yield('third')
   end
=> nil
>> do_three_times { |n| puts "#{n}: hello" }
first: hello
second: hello
third: hello
=> nil
```

`yield` returns the result of executing the block:

```
>> def number_names
     [yield('one'), yield('two'), yield('three')].join(', ')
   end
=> nil
>> number_names { |name| name.upcase.reverse }
=> "ENO, OWT, EERHT"
```

Enumerable

Ruby has a built-in module called `Enumerable` that's included by `Array`, `Hash`, `Range`, and other classes that represent collections of values. `Enumerable` provides helpful methods for traversing, searching, and sorting collections, many of which expect to be called with a block. Usually the code in the block will be run against some or all values in the collection as part of whatever job the method does. For example:

Copyrighted material

```
>> (1..10).count { |number| number.even? }
=> 5
>> (1..10).select { |number| number.even? }
=> [2, 4, 6, 8, 10]
>> (1..10).any? { |number| number < 8 }
=> true
>> (1..10).all? { |number| number < 8 }
=> false
>> (1..5).each do |number|
     if number.even?
       puts "#{number} is even"
     else
       puts "#{number} is odd"
     end
   end
1 is odd
2 is even
3 is odd
4 is even
5 is odd
=> 1..5
>> (1..10).map { |number| number * 3 }
=> [3, 6, 9, 12, 15, 18, 21, 24, 27, 30]
```

It's common for the block to take one argument and send it one message with no arguments, so Ruby provides a `&:`*`message`* shorthand as a more concise way of writing the block `{ |object| object.`*`message`*` }`:

```
>> (1..10).select(&:even?)
=> [2, 4, 6, 8, 10]
>> ['one', 'two', 'three'].map(&:upcase)
=> ["ONE", "TWO", "THREE"]
```

One of `Enumerable`'s methods, `#flat_map`, can be used to evaluate an array-producing block for every value in a collection and concatenate the results:

```
>> ['one', 'two', 'three'].map(&:chars)
```

Copyrighted material

```
=> [["o", "n", "e"], ["t", "w", "o"], ["t", "h", "r", "e", "e"]]
>> ['one', 'two', 'three'].flat_map(&:chars)
=> ["o", "n", "e", "t", "w", "o", "t", "h", "r", "e", "e"]
```

Another useful method is `#inject`, which evaluates a block for every value in a collection and accumulates a final result:

```
>> (1..10).inject(0) { |result, number| result + number }
=> 55
>> (1..10).inject(1) { |result, number| result * number }
=> 3628800
>> ['one', 'two', 'three'].inject('Words:') { |result, word| "#{result} #
{word}" }
=> "Words: one two three"
```

Struct

`Struct` is a special Ruby class whose job is to generate other classes. A class generated by `Struct` contains getter and setter methods for each of the attribute names passed into `Struct.new`. The conventional way to use a `Struct`-generated class is to subclass it; the subclass can be given a name, and it provides a convenient place to define any additional methods. For example, to make a class called `Point` with attributes called x and y, we can write:

```
class Point < Struct.new(:x, :y)
  def +(other_point)
    Point.new(x + other_point.x, y + other_point.y)
  end

  def inspect
    "#<Point (#{x}, #{y})>"
  end
end
```

Now we can create instances of `Point`, inspect them in IRB, and send them messages:

Copyrighted material

```
>> a = Point.new(2, 3)
=> #<Point (2, 3)>
>> b = Point.new(10, 20)
=> #<Point (10, 20)>
>> a + b
=> #<Point (12, 23)>
```

As well as whatever methods we define, a `Point` instance responds to the messages `x` and `x=` to get and set the value of its `x` attribute, and similarly for `y` and `y=`:

```
>> a.x
=> 2
>> a.x = 35
=> 35
>> a + b
=> #<Point (45, 23)>
```

Classes generated by `Struct.new` have other useful functionality, like an implementation of the equality method `#==`, which compares the attributes of two `Struct`s to see if they're equal:

```
>> Point.new(4, 5) == Point.new(4, 5)
=> true
>> Point.new(4, 5) == Point.new(6, 7)
=> false
```

Monkey Patching

New methods can be added to an existing class or module at any time. This is a powerful feature, usually called *monkey patching*, which lets us extend the behavior of existing classes:

```
>> class Point
     def -(other_point)
       Point.new(x - other_point.x, y - other_point.y)
```

Copyrighted material

```
    end
  end
=> nil
>> Point.new(10, 15) - Point.new(1, 1)
=> #<Point (9, 14)>
```

We can even monkey patch Ruby's built-in classes:

```
>> class String
    def shout
      upcase + '!!!'
    end
  end
=> nil
>> 'hello world'.shout
=> "HELLO WORLD!!!"
```

Defining Constants

Ruby supports a special kind of variable, called a *constant*, which should not be reassigned once it's been created. (Ruby won't prevent a constant from being reassigned, but it will generate a warning so we know we're doing something bad.) Any variable whose name begins with a capital letter is a constant. New constants can be defined at the top level or within a class or module:

```
>> NUMBERS = [4, 8, 15, 16, 23, 42]
=> [4, 8, 15, 16, 23, 42]
>> class Greetings
    ENGLISH = 'hello'
    FRENCH  = 'bonjour'
    GERMAN  = 'guten Tag'
  end
=> "guten Tag"
>> NUMBERS.last
=> 42
>> Greetings::FRENCH
=> "bonjour"
```

Copyrighted material

Class and module names always begin with a capital letter, so class and module names are constants too.

Removing Constants

When we're exploring an idea with IRB it can be useful to ask Ruby to forget about a constant altogether, especially if that constant is the name of a class or module that we want to redefine from scratch instead of monkey patching its existing definition. A top-level constant can be removed by sending the `remove_const` message to `Object`, passing the constant's name as a symbol:

```
>> NUMBERS.last
=> 42
>> Object.send(:remove_const, :NUMBERS)
=> [4, 8, 15, 16, 23, 42]
>> NUMBERS.last
NameError: uninitialized constant NUMBERS
>> Greetings::GERMAN
=> "guten Tag"
>> Object.send(:remove_const, :Greetings)
=> Greetings
>> Greetings::GERMAN
NameError: uninitialized constant Greetings
```

We have to use `Object.send(:remove_const, :NAME)` instead of just `Object.remove_const(:NAME)`, because `remove_const` is a *private* method that ordinarily can only be called by sending a message from inside the `Object` class itself; using `Object.send` allows us to bypass this restriction temporarily.

---

[1] This style comes from the Smalltalk programming language, which had a direct influence on the design of Ruby.

Copyrighted material

Part I. Programs and Machines

What is *computation*? The word itself means different things to different people, but everyone can agree that when a computer reads a program, runs that program, reads some input, and eventually produces some output, then some kind of computation has definitely happened. That gives us a decent starting point: computation is a name for *what a computer does*.

To create an environment where this familiar sort of computation can occur, we need three basic ingredients:

- A *machine* capable of performing the computation
- A *language* for writing instructions that the machine can understand
- A *program* written in that language, describing the exact computation that the machine should perform

So this part of the book is about machines, languages, and programs—what they are, how they behave, how we can model and study them, and how we can exploit them to get useful work done. By investigating these three ingredients, we can develop a better intuition for what computation is and how it happens.

In Chapter 2, we'll design and implement a toy programming language by exploring several different ways to specify its meaning. Understanding the meaning of a language is what allows us to take a lifeless piece of source code and animate it as a dynamic, executing process; each specification technique gives us a particular strategy for running a program, and we'll end up with several different ways of implementing the same language.

We'll see that programming is the art of assembling a precisely defined structure that can be dismantled, analyzed, and ultimately interpreted by a machine to create a computation. And more important, we'll discover that implementing programming languages is easy and fun: although parsing, interpretation, and compilation can seem intimidating, they're actually quite simple and enjoyable to play around with.

Copyrighted material

Programs aren't much use without machines to run them on, so in Chapter 3, we'll design very simple machines capable of performing basic, hardcoded tasks. From that humble foundation, we'll work our way up to more sophisticated machines in Chapter 4, and in Chapter 5, we'll see how to design a general-purpose computing device that can be controlled with software.

By the time we reach Part II, we'll have seen the full spectrum of computational power: some machines with very limited capabilities, others that are more useful but still frustratingly constrained, and finally, the most powerful machines that we know how to build.

Copyrighted material

Chapter 2. The Meaning of Programs

> *Don't think, feel! It is like a finger pointing away to the moon. Don't concentrate on the finger or you will miss all that heavenly glory.*
>
> —Bruce Lee

Programming languages, and the programs we write in them, are fundamental to our work as software engineers. We use them to clarify complex ideas to ourselves, communicate those ideas to each other, and, most important, implement those ideas inside our computers. Just as human society couldn't operate without natural languages, so the global community of programmers relies on programming languages to transmit and implement our ideas, with each successful program forming part of a foundation upon which the next layer of ideas can be built.

Programmers tend to be practical, pragmatic creatures. We often learn a new programming language by reading documentation, following tutorials, studying existing programs, and tinkering with simple programs of our own, without giving much thought to what those programs *mean*. Sometimes the learning process feels a lot like trial and error: we try to understand a piece of a language by looking at examples and documentation, then we try to write something in it, then everything blows up and we have to go back and try again until we manage to assemble something that mostly works. As computers and the systems they support become increasingly complex, it's tempting to think of programs as opaque incantations that represent only themselves and work only by chance.

But computer programming isn't really about *programs*, it's about *ideas*. A program is a frozen representation of an idea, a snapshot of a structure that once existed in a programmer's imagination. Programs are only worth writing because they have *meaning*. So what connects code to its meaning, and how can we be more concrete about the meaning of a program than saying "it just does whatever it does"? In this chapter, we're going to look at a few techniques for

Copyrighted material

nailing down the meanings of computer programs and see how to bring those dead snapshots to life.

The Meaning of "Meaning"

In linguistics, *semantics* is the study of the connection between words and their meanings: the word "dog" is an arrangement of shapes on a page, or a sequence of vibrations in the air caused by someone's vocal cords, which are very different things from an actual dog or the idea of dogs in general. Semantics is concerned with how these concrete signifiers relate to their abstract meanings, as well as the fundamental nature of the abstract meanings themselves.

In computer science, the field of *formal semantics* is concerned with finding ways of nailing down the elusive meanings of programs and using them to discover or prove interesting things about programming languages. Formal semantics has a wide spectrum of uses, from concrete applications like specifying new languages and devising compiler optimizations, to more abstract ones like constructing mathematical proofs of the correctness of programs.

To completely specify a programming language, we need to provide two things: a *syntax*, which describes what programs look like, and a semantics,[2] which describes what programs mean.

Plenty of languages don't have an official written specification, just a working interpreter or compiler. Ruby itself falls into this "specification by implementation" category: although there are plenty of books and tutorials about how Ruby is supposed to work, the ultimate source of all this information is Matz's Ruby Interpreter (MRI), the language's reference implementation. If any piece of Ruby documentation disagrees with the actual behavior of MRI, it's the documentation that's wrong; third-party Ruby implementations like JRuby, Rubinius, and MacRuby have to work hard to imitate the exact behavior of MRI so that they can usefully claim to be compatible with the Ruby language. Other languages like PHP and Perl 5 share this implementation-led approach to language definition.

Another way of describing a programming language is to write an official prose specification, usually in English. C++, Java, and ECMAScript (the standardized version of JavaScript) are examples of this approach: the languages are

Copyrighted material

standardized in implementation-agnostic documents written by expert committees, and many compatible implementations of those standards exist. Specifying a language with an official document is more rigorous than relying on a reference implementation—design decisions are more likely to be the result of deliberate, rational choices, rather than accidental consequences of a particular implementation—but the specifications are often quite difficult to read, and it can be very hard to tell whether they contain any contradictions, omissions, or ambiguities. In particular there's no formal way to reason about an English-language specification; we just have to read it thoroughly, think about it a lot, and hope we've understood all the consequences.

> **NOTE**
>
> A prose specification of Ruby 1.8.7 does exist, and has even been accepted as an ISO standard (ISO/IEC 30170).[3] MRI is still regarded as the canonical specification-by-implementation of the Ruby language, although the mruby project is an attempt to build a lightweight, embeddable Ruby implementation that explicitly aims for compliance with the ISO standard rather than MRI compatibility.

A third alternative is to use the mathematical techniques of formal semantics to precisely describe the meaning of a programming language. The goal here is to be completely unambiguous, as well as to write the specification in a form that's suited to methodical analysis, or even *automated* analysis, so that it can be comprehensively checked for consistency, contradiction, or oversight. We'll look at these formal approaches to semantic specification after we've seen how syntax is handled.

Syntax

A conventional computer program is a long string of characters. Every programming language comes with a collection of rules that describe what kind of character strings may be considered valid programs in that language; these rules specify the language's *syntax*.

Copyrighted material

A language's syntax rules allow us to distinguish potentially valid programs like `y = x + 1` from nonsensical ones like `>/;x:1@4`. They also provide useful information about how to read ambiguous programs: rules about operator precedence, for example, can automatically determine that `1 + 2 * 3` should be treated as though it had been written as `1 + (2 * 3)`, not as `(1 + 2) * 3`.

The intended use of a computer program is, of course, to be read by a computer, and reading programs requires a *parser*: a program that can read a character string representing a program, check it against the syntax rules to make sure it's valid, and turn it into a structured representation of that program suitable for further processing.

There are a variety of tools that can automatically turn a language's syntax rules into a parser. The details of how these rules are specified, and the techniques for turning them into usable parsers, are not the focus of this chapter—see Implementing Parsers for a quick overview—but overall, a parser should read a string like `y = x + 1` and turn it into an *abstract syntax tree* (AST), a representation of the source code that discards incidental detail like whitespace and focuses on the hierarchical structure of the program.

In the end, syntax is only concerned with the surface appearance of programs, not with their meanings. It's possible for a program to be syntactically valid but not mean anything useful; for example, it might be that the program `y = x + 1` doesn't make sense on its own because it doesn't say what `x` is beforehand, and the program `z = true + 1` might turn out to be broken when we run it because it's trying to add a number to a Boolean value. (This depends, of course, on other properties of whichever programming language we're talking about.)

As we might expect, there is no "one true way" of explaining how the syntax of a programming language corresponds to an underlying meaning. In fact there are several different ways of talking concretely about what programs mean, all with different trade-offs between formality, abstraction, expressiveness, and practical efficiency. In the next few sections, we'll look at the main formal approaches and see how they relate to each other.

Operational Semantics

The most practical way to think about the meaning of a program is *what it does*—

Copyrighted material

when we run the program, what do we expect to happen? How do different constructs in the programming language behave at run time, and what effect do they have when they're plugged together to make larger programs?

This is the basis of *operational semantics*, a way of capturing the meaning of a programming language by defining rules for how its programs execute on some kind of device. This device is often an *abstract machine*: an imaginary, idealized computer that is designed for the specific purpose of explaining how the language's programs will execute. Different kinds of programming language will usually require different designs of abstract machine in order to neatly capture their runtime behavior.

By giving an operational semantics, we can be quite rigorous and precise about the purpose of particular constructs in the language. Unlike a language specification written in English, which might contain hidden ambiguities and leave important edge cases uncovered, a formal operational specification will need to be explicit and unambiguous in order to convincingly communicate the language's behavior.

Small-Step Semantics

So, how can we design an abstract machine and use it to specify the operational semantics of a programming language? One way is to imagine a machine that evaluates a program by operating on its syntax directly, repeatedly *reducing* it in small steps, with each step bringing the program closer to its final result, whatever that turns out to mean.

These small-step reductions are similar to the way we are taught in school to evaluate algebraic expressions. For example, to evaluate (1 × 2) + (3 × 4), we know we should:

1. Perform the left-hand multiplication (1 × 2 becomes 2) and reduce the expression to 2 + (3 × 4)

2. Perform the right-hand multiplication (3 × 4 becomes 12) and reduce the expression to 2 + 12

3. Perform the addition (2 + 12 becomes 14) and end up with 14

We can think of 14 as the result because it can't be reduced any further by this

Copyrighted material

process—we recognize 14 as a special kind of algebraic expression, a *value*, which has its own meaning and doesn't require any more work on our part.

This informal process can be turned into an operational semantics by writing down formal rules about how to proceed with each small reduction step. These rules themselves need to be written in some language (the *metalanguage*), which is usually mathematical notation.

In this chapter, we're going to explore the semantics of a toy programming language—let's call it SIMPLE.[4] The mathematical description of SIMPLE's small-step semantics looks like this:

Copyrighted material

$$\frac{\langle e_1, \sigma \rangle \rightsquigarrow_e e_1'}{\langle e_1 \texttt{ + } e_2, \sigma \rangle \rightsquigarrow_e e_1' \texttt{ + } e_2} \qquad \frac{\langle e_2, \sigma \rangle \rightsquigarrow_e e_2'}{\langle v_1 \texttt{ + } e_2, \sigma \rangle \rightsquigarrow_e v_1 \texttt{ + } e_2'}$$

$$\frac{}{\langle n_1 \texttt{ + } n_2, \sigma \rangle \rightsquigarrow_e n} \text{ if } n = n_1 + n_2$$

$$\frac{\langle e_1, \sigma \rangle \rightsquigarrow_e e_1'}{\langle e_1 \texttt{ * } e_2, \sigma \rangle \rightsquigarrow_e e_1' \texttt{ * } e_2} \qquad \frac{\langle e_2, \sigma \rangle \rightsquigarrow_e e_2'}{\langle v_1 \texttt{ * } e_2, \sigma \rangle \rightsquigarrow_e v_1 \texttt{ * } e_2'}$$

$$\frac{}{\langle n_1 \texttt{ * } n_2, \sigma \rangle \rightsquigarrow_e n} \text{ if } n = n_1 \times n_2$$

$$\frac{\langle e_1, \sigma \rangle \rightsquigarrow_e e_1'}{\langle e_1 \texttt{ < } e_2, \sigma \rangle \rightsquigarrow_e e_1' \texttt{ < } e_2} \qquad \frac{\langle e_2, \sigma \rangle \rightsquigarrow_e e_2'}{\langle v_1 \texttt{ < } e_2, \sigma \rangle \rightsquigarrow_e v_1 \texttt{ < } e_2'}$$

$$\frac{}{\langle n_1 \texttt{ < } n_2, \sigma \rangle \rightsquigarrow_e \texttt{true}} \text{ if } n_1 < n_2 \qquad \frac{}{\langle n_1 \texttt{ < } n_2, \sigma \rangle \rightsquigarrow_e \texttt{false}} \text{ if } n_1 \geq n_2$$

$$\frac{}{\langle x, \sigma \rangle \rightsquigarrow_e \sigma(x)} \text{ if } x \in \mathrm{dom}(\sigma)$$

$$\frac{\langle e, \sigma \rangle \rightsquigarrow_e e'}{\langle x \texttt{ = } e, \sigma \rangle \rightsquigarrow_s \langle x \texttt{ = } e', \sigma \rangle} \qquad \frac{}{\langle x \texttt{ = } v, \sigma \rangle \rightsquigarrow_s \langle \texttt{do-nothing}, \sigma[x \mapsto v] \rangle}$$

$$\frac{\langle e, \sigma \rangle \rightsquigarrow_e e'}{\langle \texttt{if (}e\texttt{) \{ } s_1 \texttt{ \} else \{ } s_2 \texttt{ \}}, \sigma \rangle \rightsquigarrow_s \langle \texttt{if (}e'\texttt{) \{ } s_1 \texttt{ \} else \{ } s_2 \texttt{ \}}, \sigma \rangle}$$

$$\frac{}{\langle \texttt{if (true) \{ } s_1 \texttt{ \} else \{ } s_2 \texttt{ \}}, \sigma \rangle \rightsquigarrow_s \langle s_1, \sigma \rangle} \qquad \frac{}{\langle \texttt{if (false) \{ } s_1 \texttt{ \} else \{ } s_2 \texttt{ \}}, \sigma \rangle \rightsquigarrow_s \langle s_2, \sigma \rangle}$$

$$\frac{\langle s_1, \sigma \rangle \rightsquigarrow_s \langle s_1', \sigma' \rangle}{\langle s_1 \texttt{; } s_2, \sigma \rangle \rightsquigarrow_s \langle s_1' \texttt{; } s_2, \sigma' \rangle} \qquad \frac{}{\langle \texttt{do-nothing; } s_2, \sigma \rangle \rightsquigarrow_s \langle s_2, \sigma \rangle}$$

$$\frac{}{\langle \texttt{while (}e\texttt{) \{ } s \texttt{ \}}, \sigma \rangle \rightsquigarrow_s \langle \texttt{if (}e\texttt{) \{ } s\texttt{; while (}e\texttt{) \{ } s \texttt{ \} \} else \{ do-nothing \}}, \sigma \rangle}$$

Mathematically speaking, this is a set of *inference rules* that defines a *reduction relation* on Simple's abstract syntax trees. Practically speaking, it's a bunch of weird symbols that don't say anything intelligible about the meaning of computer programs.

Instead of trying to understand this formal notation directly, we're going to investigate how to write the same inference rules in Ruby. Using Ruby as the metalanguage is easier for a programmer to understand, and it gives us the added advantage of being able to execute the rules to see how they work.

Copyrighted material

> **WARNING**
>
> We are *not* trying to describe the semantics of SIMPLE by giving a "specification by implementation." Our main reason for describing the small-step semantics in Ruby instead of mathematical notation is to make the description easier for a human reader to digest. Ending up with an executable implementation of the language is just a nice bonus.
>
> The big disadvantage of using Ruby is that it explains a simple language by using a more complicated one, which perhaps defeats the philosophical purpose. We should remember that the mathematical rules are the authoritative description of the semantics, and that we're just using Ruby to develop an understanding of what those rules mean.

Expressions

We'll start by looking at the semantics of SIMPLE expressions. The rules will operate on the abstract syntax of these expressions, so we need to be able to represent SIMPLE expressions as Ruby objects. One way of doing this is to define a Ruby class for each distinct kind of element from SIMPLE's syntax—numbers, addition, multiplication, and so on—and then represent each expression as a tree of instances of these classes.

For example, here are the definitions of `Number`, `Add`, and `Multiply` classes:

```ruby
class Number < Struct.new(:value)
end

class Add < Struct.new(:left, :right)
end

class Multiply < Struct.new(:left, :right)
end
```

We can instantiate these classes to build abstract syntax trees by hand:

```ruby
>> Add.new(
     Multiply.new(Number.new(1), Number.new(2)),
```

Copyrighted material

```
    Multiply.new(Number.new(3), Number.new(4))
  )
=> #<struct Add
    left=#<struct Multiply
      left=#<struct Number value=1>,
      right=#<struct Number value=2>
    >,
    right=#<struct Multiply
      left=#<struct Number value=3>,
      right=#<struct Number value=4>
    >
  >
```

> **NOTE**
>
> Eventually, of course, we want these trees to be built automatically by a parser. We'll see how to do that in Implementing Parsers.

The `Number`, `Add`, and `Multiply` classes inherit `Struct`'s generic definition of `#inspect`, so the string representations of their instances in the IRB console contain a lot of unimportant detail. To make the content of an abstract syntax tree easier to see in IRB, we'll override `#inspect` on each class[5] so that it returns a custom string representation:

```
class Number
  def to_s
    value.to_s
  end

  def inspect
    "«#{self}»"
  end
end

class Add
```

Copyrighted material

```
  def to_s
    "#{left} + #{right}"
  end

  def inspect
    "«#{self}»"
  end
end

class Multiply
  def to_s
    "#{left} * #{right}"
  end

  def inspect
    "«#{self}»"
  end
end
```

Now each abstract syntax tree will be shown in IRB as a short string of SIMPLE source code, surrounded by «guillemets» to distinguish it from a normal Ruby value:

```
>> Add.new(
     Multiply.new(Number.new(1), Number.new(2)),
     Multiply.new(Number.new(3), Number.new(4))
   )
=> «1 * 2 + 3 * 4»
>> Number.new(5)
=> «5»
```

Copyrighted material

> **WARNING**
>
> Our rudimentary #to_s implementations don't take operator precedence into account, so sometimes their output is incorrect with respect to conventional precedence rules (e.g., * usually binds more tightly than +). Take this abstract syntax tree, for example:
>
> ```
> >> Multiply.new(
>      Number.new(1),
>      Multiply.new(
>        Add.new(Number.new(2), Number.new(3)),
>        Number.new(4)
>      )
>    )
> => «1 * 2 + 3 * 4»
> ```
>
> This tree represents «1 * (2 + 3) * 4», which is a different expression (with a different meaning) than «1 * 2 + 3 * 4», but its string representation doesn't reflect that.
>
> This problem is serious but tangential to our discussion of semantics. To keep things simple, we'll temporarily ignore it and just avoid creating expressions that have an incorrect string representation. We'll implement a proper solution for another language in Syntax.

Now we can begin to implement a small-step operational semantics by defining methods that perform reductions on our abstract syntax trees—that is, code that can take an abstract syntax tree as input and produce a slightly reduced tree as output.

Before we can implement reduction itself, we need to be able to distinguish expressions that can be reduced from those that can't. `Add` and `Multiply` expressions are always reducible—each of them represents an operation, and can be turned into a result by performing the calculation corresponding to that operation—but a `Number` expression always represents a value, which can't be reduced to anything else.

In principle, we could tell these two kinds of expression apart with a single `#reducible?` predicate that returns `true` or `false` depending on the class of its

Copyrighted material

argument:

```
def reducible?(expression)
  case expression
  when Number
    false
  when Add, Multiply
    true
  end
end
```

> **NOTE**
>
> In Ruby `case` statements, the control expression is matched against the cases by calling each case value's `#===` method with the control expression's value as an argument. The implementation of `#===` for class objects checks to see whether its argument is an instance of that class or one of its subclasses, so we can use the `case *object* when *classname*` syntax to match an object against a class.

However, it's generally considered bad form to write code like this in an object-oriented language;[6] when the behavior of some operation depends upon the class of its argument, the typical approach is to implement each per-class behavior as an instance method for that class, and let the language implicitly handle the job of deciding which of those methods to call instead of using an explicit `case` statement.

So instead, let's implement separate `#reducible?` methods for `Number`, `Add`, and `Multiply`:

```
class Number
  def reducible?
    false
  end
end
```

Copyrighted material

```
class Add
  def reducible?
    true
  end
end

class Multiply
  def reducible?
    true
  end
end
```

This gives us the behavior we want:

```
>> Number.new(1).reducible?
=> false
>> Add.new(Number.new(1), Number.new(2)).reducible?
=> true
```

We can now implement reduction for these expressions; as above, we'll do this by defining a `#reduce` method for `Add` and `Multiply`. There's no need to define `Number#reduce`, since numbers can't be reduced, so we'll just need to be careful not to call `#reduce` on an expression unless we know it's reducible.

So what are the rules for reducing an addition expression? If the left and right arguments are already numbers, then we can just add them together, but what if one or both of the arguments needs reducing? Since we're thinking about small steps, we need to decide which argument gets reduced first if they are both eligible for reduction.[7] A common strategy is to reduce the arguments in left-to-right order, in which case the rules will be:

- If the addition's left argument can be reduced, reduce the left argument.
- If the addition's left argument can't be reduced but its right argument can, reduce the right argument.
- If neither argument can be reduced, they should both be numbers, so add

Copyrighted material

them together.

The structure of these rules is characteristic of small-step operational semantics. Each rule provides a pattern for the kind of expression to which it applies—an addition with a reducible left argument, with a reducible right argument, and with two irreducible arguments respectively—and a description of how to build a new, reduced expression when that pattern matches. By choosing these particular rules, we're specifying that a Simple addition expression uses left-to-right evaluation to reduce its arguments, as well as deciding how those arguments should be combined once they've been individually reduced.

We can translate these rules directly into an implementation of `Add#reduce`, and almost the same code will work for `Multiply#reduce` (remembering to multiply the arguments instead of adding them):

```
class Add
  def reduce
    if left.reducible?
      Add.new(left.reduce, right)
    elsif right.reducible?
      Add.new(left, right.reduce)
    else
      Number.new(left.value + right.value)
    end
  end
end

class Multiply
  def reduce
    if left.reducible?
      Multiply.new(left.reduce, right)
    elsif right.reducible?
      Multiply.new(left, right.reduce)
    else
      Number.new(left.value * right.value)
    end
  end
end
```

Copyrighted material

> **NOTE**
>
> #reduce always builds a new expression rather than modifying an existing one.

Having implemented #reduce for these kinds of expressions, we can call it repeatedly to fully evaluate an expression via a series of small steps:

```
>> expression =
     Add.new(
       Multiply.new(Number.new(1), Number.new(2)),
       Multiply.new(Number.new(3), Number.new(4))
     )
=> «1 * 2 + 3 * 4»
>> expression.reducible?
=> true
>> expression = expression.reduce
=> «2 + 3 * 4»
>> expression.reducible?
=> true
>> expression = expression.reduce
=> «2 + 12»
>> expression.reducible?
=> true
>> expression = expression.reduce
=> «14»
>> expression.reducible?
=> false
```

Copyrighted material

> **NOTE**
>
> Notice that #reduce always turns one expression into another expression, which is exactly how the rules of small-step operational semantics should work. In particular, `Add.new(Number.new(2), Number.new(12)).reduce` returns `Number.new(14)`, which represents a SIMPLE expression, rather than just 14, which is a Ruby number.
>
> This separation between the SIMPLE language, whose semantics we are *specifying*, and the Ruby metalanguage, in which we are *writing the specification*, is easier to maintain when the two languages are obviously different—as is the case when the metalanguage is mathematical notation rather than a programming language—but here we need to be more careful because the two languages look very similar.

By maintaining a piece of state—the current expression—and repeatedly calling `#reducible?` and `#reduce` on it until we end up with a value, we're manually simulating the operation of an abstract machine for evaluating expressions. To save ourselves some effort, and to make the idea of the abstract machine more concrete, we can easily write some Ruby code that does the work for us. Let's wrap up that code and state together in a class and call it a *virtual machine*:

```
class Machine < Struct.new(:expression)
  def step
    self.expression = expression.reduce
  end

  def run
    while expression.reducible?
      puts expression
      step
    end

    puts expression
  end
end
```

Copyrighted material

This allows us to instantiate a virtual machine with an expression, tell it to `#run`, and watch the steps of reduction unfold:

```
>> Machine.new(
     Add.new(
       Multiply.new(Number.new(1), Number.new(2)),
       Multiply.new(Number.new(3), Number.new(4))
     )
   ).run
1 * 2 + 3 * 4
2 + 3 * 4
2 + 12
14
=> nil
```

It isn't difficult to extend this implementation to support other simple values and operations: subtraction and division; Boolean `true` and `false`; Boolean `and`, `or`, and `not`; comparison operations for numbers that return Booleans; and so on. For example, here are implementations of Booleans and the less-than operator:

```
class Boolean < Struct.new(:value)
  def to_s
    value.to_s
  end

  def inspect
    "«#{self}»"
  end

  def reducible?
    false
  end
end

class LessThan < Struct.new(:left, :right)
  def to_s
```

Copyrighted material

```
    "#{left} < #{right}"
  end

  def inspect
    "«#{self}»"
  end

  def reducible?
    true
  end

  def reduce
    if left.reducible?
      LessThan.new(left.reduce, right)
    elsif right.reducible?
      LessThan.new(left, right.reduce)
    else
      Boolean.new(left.value < right.value)
    end
  end
end
```

Again, this allows us to reduce a boolean expression in small steps:

```
>> Machine.new(
     LessThan.new(Number.new(5), Add.new(Number.new(2), Number.new(2)))
   ).run
5 < 2 + 2
5 < 4
false
=> nil
```

So far, so straightforward: we have begun to specify the operational semantics of a language by implementing a virtual machine that can evaluate it. At the moment the state of this virtual machine is just the current expression, and the behavior of the machine is described by a collection of rules that govern how that state changes when the machine runs. We've implemented the machine as a

Copyrighted material

program that keeps track of the current expression and keeps reducing it, updating the expression as it goes, until no more reductions can be performed.

But this language of simple algebraic expressions isn't very interesting, and doesn't have many of the features that we expect from even the simplest programming language, so let's build it out to be more sophisticated and look more like a language in which we could write useful programs.

First off, there's something obviously missing from SIMPLE: variables. In any useful language, we'd expect to be able to talk about values using meaningful names rather than the literal values themselves. These names provide a layer of indirection so that the same code can be used to process many different values, including values that come from outside the program and therefore aren't even known when the code is written.

We can introduce a new class of expression, `Variable`, to represent variables in SIMPLE:

```
class Variable < Struct.new(:name)
  def to_s
    name.to_s
  end

  def inspect
    "«#{self}»"
  end

  def reducible?
    true
  end
end
```

To be able to reduce a variable, we need the abstract machine to store a mapping from variable names onto their values, an *environment*, as well as the current expression. In Ruby, we can implement this mapping as a hash, using symbols as keys and expression objects as values; for example, the hash `{ x: Number.new(2), y: Boolean.new(false) }` is an environment that associates the variables `x` and `y` with a SIMPLE number and Boolean, respectively.

Copyrighted material

> **NOTE**
>
> For this language, the intention is for the environment to only map variable names onto irreducible values like `Number.new(2)`, not onto reducible expressions like `Add.new(Number.new(1), Number.new(2))`. We'll take care to respect this constraint later when we write rules that change the contents of the environment.

Given an environment, we can easily implement `Variable#reduce`: it just looks up the variable's name in the environment and returns its value.

```
class Variable
  def reduce(environment)
    environment[name]
  end
end
```

Notice that we're now passing an `environment` argument into `#reduce`, so we'll need to revise the other expression classes' implementations of `#reduce` to both accept and provide this argument:

```
class Add
  def reduce(environment)
    if left.reducible?
      Add.new(left.reduce(environment), right)
    elsif right.reducible?
      Add.new(left, right.reduce(environment))
    else
      Number.new(left.value + right.value)
    end
  end
end

class Multiply
  def reduce(environment)
    if left.reducible?
```

Copyrighted material

```
      Multiply.new(left.reduce(environment), right)
    elsif right.reducible?
      Multiply.new(left, right.reduce(environment))
    else
      Number.new(left.value * right.value)
    end
  end
end

class LessThan
  def reduce(environment)
    if left.reducible?
      LessThan.new(left.reduce(environment), right)
    elsif right.reducible?
      LessThan.new(left, right.reduce(environment))
    else
      Boolean.new(left.value < right.value)
    end
  end
end
```

Once all the implementations of #reduce have been updated to support environments, we also need to redefine our virtual machine to maintain an environment and provide it to #reduce:

```
Object.send(:remove_const, :Machine)

class Machine < Struct.new(:expression, :environment)
  def step
    self.expression = expression.reduce(environment)
  end

  def run
    while expression.reducible?
      puts expression
      step
    end
```

Copyrighted material

```
    puts expression
  end
end
```

The machine's definition of `#run` remains unchanged, but it has a new `environment` attribute that is used by its new implementation of `#step`.

We can now perform reductions on expressions that contain variables, as long as we also supply an environment that contains the variables' values:

```
>> Machine.new(
     Add.new(Variable.new(:x), Variable.new(:y)),
     { x: Number.new(3), y: Number.new(4) }
   ).run
x + y
3 + y
3 + 4
7
=> nil
```

The introduction of an environment completes our operational semantics of expressions. We've designed an abstract machine that begins with an initial expression and environment, and then uses the current expression and environment to produce a new expression in each small reduction step, leaving the environment unchanged.

Statements

We can now look at implementing a different kind of program construct: *statements*. The purpose of an expression is to be evaluated to produce another expression; a statement, on the other hand, is evaluated to make some change to the state of the abstract machine. Our machine's only piece of state (aside from the current program) is the environment, so we'll allow SIMPLE statements to produce a new environment that can replace the current one.

The simplest possible statement is one that does nothing: it can't be reduced, so it can't have any effect on the environment. That's easy to implement:

Copyrighted material

```ruby
class DoNothing ❶
  def to_s
    'do-nothing'
  end

  def inspect
    "«#{self}»"
  end

  def ==(other_statement) ❷
    other_statement.instance_of?(DoNothing)
  end

  def reducible?
    false
  end
end
```

❶ All of our other syntax classes inherit from a `Struct` class, but `DoNothing` doesn't inherit from anything. This is because `DoNothing` doesn't have any attributes, and unfortunately, `Struct.new` doesn't let us pass an empty list of attribute names.

❷ We want to be able to compare any two statements to see if they're equal. The other syntax classes inherit an implementation of #== from `Struct`, but `DoNothing` has to define its own.

A statement that does nothing might seem pointless, but it's convenient to have a special statement that represents a program whose execution has completed successfully. We'll arrange for other statements to eventually reduce to «`do-nothing`» once they've finished doing their work.

The simplest example of a statement that actually does something useful is an *assignment* like «`x = x + 1`», but before we can implement assignment, we need to decide what its reduction rules should be.

An assignment statement consists of a variable name (x), an equals symbol, and

Copyrighted material

an expression («x + 1»). If the expression within the assignment is reducible, we can just reduce it according to the expression reduction rules and end up with a new assignment statement containing the reduced expression. For example, reducing «x = x + 1» in an environment where the variable x has the value «2» should leave us with the statement «x = 2 + 1», and reducing it again should produce «x = 3».

But then what? If the expression is already a value like «3», then we should just perform the assignment, which means updating the environment to associate that value with the appropriate variable name. So reducing a statement needs to produce not just a new, reduced statement but also a new environment, which will sometimes be different from the environment in which the reduction was performed.

Copyrighted material

NOTE

Our implementation will update the environment by using `Hash#merge` to create a new hash without modifying the old one:

```
>> old_environment = { y: Number.new(5) }
=> {:y=>«5»}
>> new_environment = old_environment.merge({ x: Number.new(3) })
=> {:y=>«5», :x=>«3»}
>> old_environment
=> {:y=>«5»}
```

We could choose to destructively modify the current environment instead of making a new one, but avoiding destructive updates forces us to make the consequences of `#reduce` completely explicit. If `#reduce` wants to change the current environment, it has to communicate that by returning an updated environment to its caller; conversely, if it doesn't return an environment, we can be sure it hasn't made any changes.

This constraint helps to highlight the difference between expressions and statements. For expressions, we pass an environment into `#reduce` and get a reduced expression back; no new environment is returned, so reducing an expression obviously doesn't change the environment. For statements, we'll call `#reduce` with the current environment and get a new environment back, which tells us that reducing a statement can have an effect on the environment. (In other words, the structure of SIMPLE's small-step semantics shows that its expressions are *pure* and its statements are *impure*.)

So reducing «`x = 3`» in an empty environment should produce the new environment `{ x: Number.new(3) }`, but we also expect the statement to be reduced somehow; otherwise, our abstract machine will keep assigning «3» to `x` forever. That's what «`do-nothing`» is for: a completed assignment reduces to «`do-nothing`», indicating that reduction of the statement has finished and that whatever's in the new environment may be considered its result.

To summarize, the reduction rules for assignment are:

- If the assignment's expression can be reduced, then reduce it, resulting in a reduced assignment statement and an unchanged environment.

Copyrighted material

- If the assignment's expression can't be reduced, then update the environment to associate that expression with the assignment's variable, resulting in a «`do-nothing`» statement and a new environment.

This gives us enough information to implement an `Assign` class. The only difficulty is that `Assign#reduce` needs to return both a statement and an environment—Ruby methods can only return a single object—but we can pretend to return two objects by putting them into a two-element array and returning that.

```
class Assign < Struct.new(:name, :expression)
  def to_s
    "#{name} = #{expression}"
  end

  def inspect
    "«#{self}»"
  end

  def reducible?
    true
  end

  def reduce(environment)
    if expression.reducible?
      [Assign.new(name, expression.reduce(environment)), environment]
    else
      [DoNothing.new, environment.merge({ name => expression })]
    end
  end
end
```

> **NOTE**
>
> As promised, the reduction rules for `Assign` ensure that an expression only gets added to the environment if it's irreducible (i.e., a value).

Copyrighted material

As with expressions, we can manually evaluate an assignment statement by repeatedly reducing it until it can't be reduced any more:

```
>> statement = Assign.new(:x, Add.new(Variable.new(:x), Number.new(1)))
=> «x = x + 1»
>> environment = { x: Number.new(2) }
=> {:x=>«2»}
>> statement.reducible?
=> true
>> statement, environment = statement.reduce(environment)
=> [«x = 2 + 1», {:x=>«2»}]
>> statement, environment = statement.reduce(environment)
=> [«x = 3», {:x=>«2»}]
>> statement, environment = statement.reduce(environment)
=> [«do-nothing», {:x=>«3»}]
>> statement.reducible?
=> false
```

This process is even more laborious than manually reducing expressions, so let's reimplement our virtual machine to handle statements, showing the current statement and environment at each reduction step:

```
Object.send(:remove_const, :Machine)

class Machine < Struct.new(:statement, :environment)
  def step
    self.statement, self.environment = statement.reduce(environment)
  end

  def run
    while statement.reducible?
      puts "#{statement}, #{environment}"
      step
    end

    puts "#{statement}, #{environment}"
  end
end
```

Copyrighted material

Now the machine can do the work for us again:

```
>> Machine.new(
     Assign.new(:x, Add.new(Variable.new(:x), Number.new(1))),
     { x: Number.new(2) }
   ).run
x = x + 1, {:x=>«2»}
x = 2 + 1, {:x=>«2»}
x = 3, {:x=>«2»}
do-nothing, {:x=>«3»}
=> nil
```

We can see that the machine is still performing expression reduction steps («x + 1» to «2 + 1» to «3»), but they now happen inside a statement instead of at the top level of the syntax tree.

Now that we know how statement reduction works, we can extend it to support other kinds of statements. Let's begin with conditional statements like «if (x) { y = 1 } else { y = 2 }», which contain an expression called the *condition* («x»), and two statements that we'll call the *consequence* («y = 1») and the *alternative* («y = 2»).[8] The reduction rules for conditionals are straightforward:

- If the condition can be reduced, then reduce it, resulting in a reduced conditional statement and an unchanged environment.
- If the condition is the expression «true», reduce to the consequence statement and an unchanged environment.
- If the condition is the expression «false», reduce to the alternative statement and an unchanged environment.

In this case, none of the rules changes the environment—the reduction of the condition expression in the first rule will only produce a new expression, not a new environment.

Here are the rules translated into an If class:

Copyrighted material

```
class If < Struct.new(:condition, :consequence, :alternative)
  def to_s
    "if (#{condition}) { #{consequence} } else { #{alternative} }"
  end

  def inspect
    "«#{self}»"
  end

  def reducible?
    true
  end

  def reduce(environment)
    if condition.reducible?
      [If.new(condition.reduce(environment), consequence, alternative),
environment]
    else
      case condition
      when Boolean.new(true)
        [consequence, environment]
      when Boolean.new(false)
        [alternative, environment]
      end
    end
  end
end
```

And here's how the reduction steps look:

```
>> Machine.new(
     If.new(
       Variable.new(:x),
       Assign.new(:y, Number.new(1)),
       Assign.new(:y, Number.new(2))
     ),
     { x: Boolean.new(true) }
   ).run
```

Copyrighted material

```
if (x) { y = 1 } else { y = 2 }, {:x=>«true»}
if (true) { y = 1 } else { y = 2 }, {:x=>«true»}
y = 1, {:x=>«true»}
do-nothing, {:x=>«true», :y=>«1»}
=> nil
```

That all works as expected, but it would be nice if we could support conditional statements with no «`else`» clause, like «`if (x) { y = 1 }`». Fortunately, we can already do that by writing statements like «`if (x) { y = 1 } else { do-nothing }`», which behave as though the «`else`» clause wasn't there:

```
>> Machine.new(
     If.new(Variable.new(:x), Assign.new(:y, Number.new(1)),
DoNothing.new),
     { x: Boolean.new(false) }
   ).run
if (x) { y = 1 } else { do-nothing }, {:x=>«false»}
if (false) { y = 1 } else { do-nothing }, {:x=>«false»}
do-nothing, {:x=>«false»}
=> nil
```

Now that we've implemented assignment and conditional statements as well as expressions, we have the building blocks for programs that can do real work by performing calculations and making decisions. The main restriction is that we can't yet connect these blocks together: we have no way to assign values to more than one variable, or to perform more than one conditional operation, which drastically limits the usefulness of our language.

We can fix this by defining another kind of statement, the *sequence*, which connects two statements like «`x = 1 + 1`» and «`y = x + 3`» to make one larger statement like «`x = 1 + 1; y = x + 3`». Once we have sequence statements, we can use them repeatedly to build even larger statements; for example, the sequence «`x = 1 + 1; y = x + 3`» and the assignment «`z = y + 5`» can be combined to make the sequence «`x = 1 + 1; y = x + 3; z = y + 5`».[9]

The reduction rules for sequences are slightly subtle:

- If the first statement is a «`do-nothing`» statement, reduce to the second

Copyrighted material

statement and the original environment.

- If the first statement is not «do-nothing», then reduce it, resulting in a new sequence (the reduced first statement followed by the second statement) and a reduced environment.

Seeing the code may make these rules clearer:

```
class Sequence < Struct.new(:first, :second)
  def to_s
    "#{first}; #{second}"
  end

  def inspect
    "«#{self}»"
  end

  def reducible?
    true
  end

  def reduce(environment)
    case first
    when DoNothing.new
      [second, environment]
    else
      reduced_first, reduced_environment = first.reduce(environment)
      [Sequence.new(reduced_first, second), reduced_environment]
    end
  end
end
```

The overall effect of these rules is that, when we repeatedly reduce a sequence, it keeps reducing its first statement until it turns into «do-nothing», then reduces to its second statement. We can see this happening when we run a sequence in the virtual machine:

Copyrighted material

```
>> Machine.new(
     Sequence.new(
       Assign.new(:x, Add.new(Number.new(1), Number.new(1))),
       Assign.new(:y, Add.new(Variable.new(:x), Number.new(3)))
     ),
     {}
   ).run
x = 1 + 1; y = x + 3, {}
x = 2; y = x + 3, {}
do-nothing; y = x + 3, {:x=>«2»}
y = x + 3, {:x=>«2»}
y = 2 + 3, {:x=>«2»}
y = 5, {:x=>«2»}
do-nothing, {:x=>«2», :y=>«5»}
=> nil
```

The only really major thing still missing from SIMPLE is some kind of unrestricted looping construct, so to finish off, let's introduce a «while» statement so that programs can perform repeated calculations an arbitrary number of times.[10] A statement like «while (x < 5) { x = x * 3 }» contains an expression called the *condition* («x < 5») and a statement called the *body* («x = x * 3»).

Writing the correct reduction rules for a «while» statement is slightly tricky. We could try treating it like an «if» statement: reduce the condition if possible; otherwise, reduce to either the body or «do-nothing», depending on whether the condition is «true» or «false», respectively. But once the abstract machine has completely reduced the body, what next? The condition has been reduced to a value and thrown away, and the body has been reduced to «do-nothing», so how do we perform another iteration of the loop? Each reduction step can only communicate with future steps by producing a new statement and environment, and this approach doesn't give us anywhere to "remember" the original condition and body for use on the next iteration.

The small-step solution[11] is to use the sequence statement to *unroll* one level of the «while», reducing it to an «if» that performs a single iteration of the loop and then repeats the original «while». This means we only need one reduction rule:

Copyrighted material

- Reduce «`while (`*condition*`) { `*body*` }`» to «`if (`*condition*`) { `*body*`; while (`*condition*`) { `*body*` } } else { do-nothing }`» and an unchanged environment.

And this rule is easy to implement in Ruby:

```
class While < Struct.new(:condition, :body)
  def to_s
    "while (#{condition}) { #{body} }"
  end

  def inspect
    "«#{self}»"
  end

  def reducible?
    true
  end

  def reduce(environment)
    [If.new(condition, Sequence.new(body, self), DoNothing.new),
environment]
  end
end
```

This gives the virtual machine the opportunity to evaluate the condition and body as many times as necessary:

```
>> Machine.new(
     While.new(
       LessThan.new(Variable.new(:x), Number.new(5)),
       Assign.new(:x, Multiply.new(Variable.new(:x), Number.new(3)))
     ),
     { x: Number.new(1) }
   ).run
while (x < 5) { x = x * 3 }, {:x=>«1»}
if (x < 5) { x = x * 3; while (x < 5) { x = x * 3 } } else { do-nothing },
```

Copyrighted material

```
{:x=>«1»}
if (1 < 5) { x = x * 3; while (x < 5) { x = x * 3 } } else { do-nothing },
{:x=>«1»}
if (true) { x = x * 3; while (x < 5) { x = x * 3 } } else { do-nothing },
{:x=>«1»}
x = x * 3; while (x < 5) { x = x * 3 }, {:x=>«1»}
x = 1 * 3; while (x < 5) { x = x * 3 }, {:x=>«1»}
x = 3; while (x < 5) { x = x * 3 }, {:x=>«1»}
do-nothing; while (x < 5) { x = x * 3 }, {:x=>«3»}
while (x < 5) { x = x * 3 }, {:x=>«3»}
if (x < 5) { x = x * 3; while (x < 5) { x = x * 3 } } else { do-nothing },
{:x=>«3»}
if (3 < 5) { x = x * 3; while (x < 5) { x = x * 3 } } else { do-nothing },
{:x=>«3»}
if (true) { x = x * 3; while (x < 5) { x = x * 3 } } else { do-nothing },
{:x=>«3»}
x = x * 3; while (x < 5) { x = x * 3 }, {:x=>«3»}
x = 3 * 3; while (x < 5) { x = x * 3 }, {:x=>«3»}
x = 9; while (x < 5) { x = x * 3 }, {:x=>«3»}
do-nothing; while (x < 5) { x = x * 3 }, {:x=>«9»}
while (x < 5) { x = x * 3 }, {:x=>«9»}
if (x < 5) { x = x * 3; while (x < 5) { x = x * 3 } } else { do-nothing },
{:x=>«9»}
if (9 < 5) { x = x * 3; while (x < 5) { x = x * 3 } } else { do-nothing },
{:x=>«9»}
if (false) { x = x * 3; while (x < 5) { x = x * 3 } } else { do-nothing },
{:x=>«9»}
do-nothing, {:x=>«9»}
=> nil
```

Perhaps this reduction rule seems like a bit of a dodge—it's almost as though we're perpetually postponing reduction of the «`while`» until later, without ever actually getting there—but on the other hand, it does a good job of explaining what we really mean by a «`while`» statement: check the condition, evaluate the body, then start again. It's curious that reducing «`while`» turns it into a syntactically larger program involving conditional and sequence statements instead of directly reducing its condition or body, and one reason why it's useful to have a technical framework for specifying the formal semantics of a language

Copyrighted material

is to help us see how different parts of the language relate to each other like this.

Correctness

We've completely ignored what will happen when a syntactically valid but otherwise incorrect program is executed according to the semantics we've given. The statement «`x = true; x = x + 1`» is a valid piece of SIMPLE syntax—we can certainly construct an abstract syntax tree to represent it—but it'll blow up when we try to repeatedly reduce it, because the abstract machine will end up trying to add «`1`» to «`true`»:

```
>> Machine.new(
     Sequence.new(
       Assign.new(:x, Boolean.new(true)),
       Assign.new(:x, Add.new(Variable.new(:x), Number.new(1)))
     ),
     {}
   ).run
x = true; x = x + 1, {}
do-nothing; x = x + 1, {:x=>«true»}
x = x + 1, {:x=>«true»}
x = true + 1, {:x=>«true»}
NoMethodError: undefined method `+' for true:TrueClass
```

One way to handle this is to be more restrictive about when expressions can be reduced, which introduces the possibility that evaluation will get *stuck* rather than always trying to reduce to a value (and potentially blowing up in the process). We could have implemented `Add#reducible?` to only return `true` when both arguments to «`+`» are either reducible or an instance of `Number`, in which case the expression «`true + 1`» would get stuck and never turn into a value.

Ultimately, we need a more powerful tool than syntax, something that can "see the future" and prevent us from trying to execute any program that has the potential to blow up or get stuck. This chapter is about *dynamic semantics*—what a program does when it's executed—but that's not the only kind of meaning that a program can have; in Chapter 9, we'll investigate *static semantics* to see how we can decide whether a syntactically valid program has a useful meaning

Copyrighted material

according to the language's dynamic semantics.

Applications

The programming language we've specified is very basic, but in writing down all the reduction rules, we've still had to make some design decisions and express them unambiguously. For example, unlike Ruby, SIMPLE is a language that makes a distinction between expressions, which return a value, and statements, which don't; like Ruby, SIMPLE evaluates expressions in a left-to-right order; and like Ruby, SIMPLE's environments associate variables only with fully reduced values, not with larger expressions that still have some unfinished computation to perform.[12] We could change any of these decisions by giving a different small-step semantics which would describe a new language with the same syntax but different runtime behavior. If we added more elaborate features to the language—data structures, procedure calls, exceptions, an object system—we'd need to make many more design decisions and express them unambiguously in the semantic definition.

The detailed, execution-oriented style of small-step semantics lends itself well to the task of unambiguously specifying real-world programming languages. For example, the latest R6RS standard for the Scheme programming language uses small-step semantics to describe its execution, and provides a reference implementation of those semantics written in PLT Redex, "a domain-specific language designed for specifying and debugging operational semantics." The OCaml programming language, which is built as a series of layers on top of a simpler language called Core ML, also has a small-step semantic definition of the base language's runtime behavior.

See Semantics for another example of using small-step operational semantics to specify the meaning of expressions in an even simpler programming language called the lambda calculus.

Big-Step Semantics

We've now seen what small-step operational semantics looks like: we design an abstract machine that maintains some execution state, then define reduction rules that specify how each kind of program construct can make incremental progress toward being fully evaluated. In particular, small-step semantics has a

Copyrighted material

mostly *iterative* flavor, requiring the abstract machine to repeatedly perform reduction steps (the Ruby `while` loop in `Machine#run`) that are themselves constructed to produce as output the same kind of information that they require as input, making them suitable for this kind of repeated application.[13]

The small-step approach has the advantage of slicing up the complex business of executing an entire program into smaller pieces that are easier to explain and analyze, but it does feel a bit indirect: instead of explaining how a whole program construct works, we just show how it can be reduced slightly. Why can't we explain a statement more directly, by telling a complete story about how its execution works? Well, we can, and that's the basis of *big-step semantics*.

The idea of big-step semantics is to specify how to get from an expression or statement straight to its result. This necessarily involves thinking about program execution as a *recursive* rather than an iterative process: big-step semantics says that, to evaluate a large expression, we evaluate all of its smaller subexpressions and then combine their results to get our final answer.

In many ways, this feels more natural than the small-step approach, but it does lack some of its fine-grained attention to detail. For example, our small-step semantics is explicit about the order in which operations are supposed to happen, because at every point, it identifies what the next step of reduction should be, but big-step semantics is often written in a looser style that just says which subcomputations to perform without necessarily specifying what order to perform them in.[14] Small-step semantics also gives us an easy way to observe the intermediate stages of a computation, whereas big-step semantics just returns a result and doesn't produce any direct evidence of how it was computed.

To understand this trade-off, let's revisit some common language constructs and see how to implement their big-step semantics in Ruby. Our small-step semantics required a `Machine` class to keep track of state and perform repeated reductions, but we won't need that here; big-step rules describe how to compute the result of an entire program by walking over its abstract syntax tree in a single attempt, so there's no state or repetition to deal with. We'll just define an `#evaluate` method on our expression and statement classes and call it directly.

Expressions

Copyrighted material

With small-step semantics we had to distinguish reducible expressions like «1 + 2» from irreducible expressions like «3» so that the reduction rules could tell when a subexpression was ready to be used as part of some larger computation, but in big-step semantics every expression can be evaluated. The only distinction, if we wanted to make one, is that some expressions immediately evaluate to themselves, while others perform some computation and evaluate to a different expression.

The goal of big-step semantics is to model the same runtime behavior as the small-step semantics, which means we expect the big-step rules for each kind of program construct to agree with what repeated application of the small-step rules would eventually produce. (This is exactly the sort of thing that can be formally proved when an operational semantics is written mathematically.) The small-step rules for values like `Number` and `Boolean` say that we can't reduce them at all, so their big-step rules are very simple: values immediately evaluate to themselves.

```
class Number
  def evaluate(environment)
    self
  end
end

class Boolean
  def evaluate(environment)
    self
  end
end
```

`Variable` expressions are unique in that their small-step semantics allow them to be reduced exactly once before they turn into a value, so their big-step rule is the same as their small-step one: look up the variable name in the environment and return its value.

```
class Variable
  def evaluate(environment)
    environment[name]
```

Copyrighted material

```
  end
end
```

The binary expressions `Add`, `Multiply`, and `LessThan` are slightly more interesting, requiring recursive evaluation of their left and right subexpressions before combining both values with the appropriate Ruby operator:

```
class Add
  def evaluate(environment)
    Number.new(left.evaluate(environment).value +
right.evaluate(environment).value)
  end
end

class Multiply
  def evaluate(environment)
    Number.new(left.evaluate(environment).value *
right.evaluate(environment).value)
  end
end

class LessThan
  def evaluate(environment)
    Boolean.new(left.evaluate(environment).value <
right.evaluate(environment).value)
  end
end
```

To check that these big-step expression semantics are correct, here they are in action on the Ruby console:

```
>> Number.new(23).evaluate({})
=> «23»
>> Variable.new(:x).evaluate({ x: Number.new(23) })
=> «23»
>> LessThan.new(
     Add.new(Variable.new(:x), Number.new(2)),
```

Copyrighted material

```
      Variable.new(:y)
    ).evaluate({ x: Number.new(2), y: Number.new(5) })
=> «true»
```

Statements

This style of semantics shines when we come to specify the behavior of statements. Expressions reduce to other expressions under small-step semantics, but statements reduce to «do-nothing» and leave a modified environment behind. We can think of big-step statement evaluation as a process that always turns a statement and an initial environment into a final environment, avoiding the small-step complication of also having to deal with the intermediate statement generated by #reduce. Big-step evaluation of an assignment statement, for example, should fully evaluate its expression and return an updated environment containing the resulting value:

```
class Assign
  def evaluate(environment)
    environment.merge({ name => expression.evaluate(environment) })
  end
end
```

Similarly, DoNothing#evaluate will clearly return the unmodified environment, and If#evaluate has a pretty straightforward job on its hands: evaluate the condition, then return the environment that results from evaluating either the consequence or the alternative.

```
class DoNothing
  def evaluate(environment)
    environment
  end
end

class If
  def evaluate(environment)
    case condition.evaluate(environment)
    when Boolean.new(true)
```

Copyrighted material

```
        consequence.evaluate(environment)
      when Boolean.new(false)
        alternative.evaluate(environment)
      end
    end
  end
```

The two interesting cases are sequence statements and «while» loops. For sequences, we just need to evaluate both statements, but the initial environment needs to be "threaded through" these two evaluations, so that the result of evaluating the first statement becomes the environment in which the second statement is evaluated. This can be written in Ruby by using the first evaluation's result as the argument to the second:

```
class Sequence
  def evaluate(environment)
    second.evaluate(first.evaluate(environment))
  end
end
```

This threading of the environment is vital to allow earlier statements to prepare variables for later ones:

```
>> statement =
     Sequence.new(
       Assign.new(:x, Add.new(Number.new(1), Number.new(1))),
       Assign.new(:y, Add.new(Variable.new(:x), Number.new(3)))
     )
=> «x = 1 + 1; y = x + 3»
>> statement.evaluate({})
=> {:x=>«2», :y=>«5»}
```

For «while» statements, we need to think through the stages of completely evaluating a loop:

- Evaluate the condition to get either «true» or «false».

Copyrighted material

- If the condition evaluates to «`true`», evaluate the body to get a new environment, then repeat the loop within that new environment (i.e., evaluate the whole «`while`» statement again) and return the resulting environment.
- If the condition evaluates to «`false`», return the environment unchanged.

This is a recursive explanation of how a «`while`» statement should behave. As with sequence statements, it's important that the updated environment generated by the loop body is used for the next iteration; otherwise, the condition will never stop being «`true`», and the loop will never get a chance to terminate.[15]

Once we know how big-step «`while`» semantics should behave, we can implement `While#evaluate`:

```
class While
  def evaluate(environment)
    case condition.evaluate(environment)
    when Boolean.new(true)
      evaluate(body.evaluate(environment)) ❶
    when Boolean.new(false)
      environment
    end
  end
end
```

❶ This is where the looping happens: `body.evaluate(environment)` evaluates the loop body to get a new environment, then we pass that environment *back into the current method* to kick off the next iteration. This means we might stack up many nested invocations of `While#evaluate` until the condition eventually becomes «`false`» and the final environment is returned.

Copyrighted material

WARNING

As with any recursive code, there's a risk that the Ruby call stack will overflow if the nested invocations become too deep. Some Ruby implementations have experimental support for *tail call optimization*, a technique that reduces the risk of overflow by reusing the same stack frame when possible. In the official Ruby implementation (MRI) we can enable tail call optimization with:

```
RubyVM::InstructionSequence.compile_option = {
  tailcall_optimization: true,
  trace_instruction: false
}
```

To confirm that this works properly, we can try evaluating the same «while» statement we used to check the small-step semantics:

```
>> statement =
     While.new(
       LessThan.new(Variable.new(:x), Number.new(5)),
       Assign.new(:x, Multiply.new(Variable.new(:x), Number.new(3)))
     )
=> «while (x < 5) { x = x * 3 }»
>> statement.evaluate({ x: Number.new(1) })
=> {:x=>«9»}
```

This is the same result that the small-step semantics gave, so it looks like `While#evaluate` does the right thing.

Applications

Our earlier implementation of small-step semantics makes only moderate use of the Ruby call stack: when we call `#reduce` on a large program, that might cause a handful of nested `#reduce` calls as the message travels down the abstract syntax tree until it reaches the piece of code that is ready to reduce.[16] But the virtual machine does the work of tracking the overall progress of the computation by maintaining the current program and environment as it repeatedly performs

Copyrighted material

small reductions; in particular, the depth of the call stack is limited by the depth of the program's syntax tree, since the nested calls are only being used to traverse the tree looking for what to reduce next, not to perform the reduction itself.

By contrast, this big-step implementation makes much greater use of the stack, relying entirely on it to remember where we are in the overall computation, to perform smaller computations as part of performing larger ones, and to keep track of how much evaluation is left to do. What looks like a single call to `#evaluate` actually turns into a series of recursive calls, each one evaluating a subprogram deeper within the syntax tree.

This difference highlights the purpose of each approach. Small-step semantics assumes a simple abstract machine that can perform small operations, and therefore includes explicit detail about how to produce useful intermediate results; big-step semantics places the burden of assembling the whole computation on the machine or person executing it, requiring her to keep track of many intermediate subgoals as she turns the entire program into a final result in a single operation. Depending on what we wish to do with a language's operational semantics—perhaps build an efficient implementation, prove some properties of programs, or devise some optimizing transformations—one approach or the other might be more appropriate.

The most influential use of big-step semantics for specifying real programming languages is Chapter 6 of the original definition of the Standard ML programming language, which explains all of the runtime behavior of ML in big-step style. Following this example, OCaml's core language has a big-step semantics to complement its more detailed small-step definition.

Big-step operational semantics is also used by the W3C: the XQuery 1.0 and XPath 2.0 specification uses mathematical inference rules to describe how its languages should be evaluated, and the XQuery and XPath Full Text 3.0 spec includes a big-step semantics written in XQuery.

It probably hasn't escaped your attention that, by writing down Simple's small- and big-step semantics in Ruby instead of mathematics, we have implemented two different Ruby *interpreters* for it. And this is what operational semantics really is: explaining the meaning of a language by describing an interpreter. Normally, that description would be written in simple mathematical notation, which makes everything very clear and unambiguous as long as we can

Copyrighted material

understand it, but comes at the price of being quite abstract and distanced from the reality of computers. Using Ruby has the disadvantage of introducing the extra complexity of a real-world programming language (classes, objects, method calls...) into what's supposed to be a simplifying explanation, but if we already understand Ruby, then it's probably easier to see what's going on, and being able to execute the description as an interpreter is a nice bonus.

Denotational Semantics

So far, we've looked at the meaning of programming languages from an operational perspective, explaining what a program means by showing what will happen when it's executed. Another approach, *denotational semantics*, is concerned instead with translating programs from their native language into some other representation.

This style of semantics doesn't directly address the question of executing a program at all. Instead, it concerns itself with leveraging the established meaning of another language—one that is lower-level, more formal, or at least better understood than the language being described—in order to explain a new one.

Denotational semantics is necessarily a more abstract approach than operational, because it just replaces one language with another instead of turning a language into real behavior. For example, if we needed to explain the meaning of the English verb "walk" to a person with whom we had no spoken language in common, we could communicate it operationally by actually walking back and forth. On the other hand, if we needed to explain "walk" to a French speaker, we could do so denotationally just by telling them the French verb "*marcher*"—an undeniably higher level form of communication, no messy exercise required.

Unsurprisingly, denotational semantics is conventionally used to turn programs into mathematical objects so they can be studied and manipulated with mathematical tools, but we can get some of the flavor of this approach by looking at how to denote SIMPLE programs in some other way.

Let's try giving a denotational semantics for SIMPLE by translating it into Ruby.[17] In practice, this means turning an abstract syntax tree into a string of Ruby code

Copyrighted material

that somehow captures the intended meaning of that syntax.

But what is the “intended meaning”? What should Ruby denotations of our expressions and statements look like? We’ve already seen operationally that an expression takes an environment and turns it into a value; one way to express this in Ruby is with a proc that takes some argument representing an environment argument and returns some Ruby object representing a value. For simple constant expressions like «5» and «false», we won’t be using the environment at all, so we only need to worry about how their eventual result can be represented as a Ruby object. Fortunately, Ruby already has objects specifically designed to represent these values: we can use the Ruby value 5 as the result of the SIMPLE expression «5», and likewise, the Ruby value false as the result of «false».

Expressions

We can use this idea to write implementations of a #to_ruby method for the Number and Boolean classes:

```
class Number
  def to_ruby
    "-> e { #{value.inspect} }"
  end
end

class Boolean
  def to_ruby
    "-> e { #{value.inspect} }"
  end
end
```

Here is how they behave on the console:

```
>> Number.new(5).to_ruby
=> "-> e { 5 }"
>> Boolean.new(false).to_ruby
=> "-> e { false }"
```

Copyrighted material

Each of these methods produces a string that happens to contain Ruby code, and because Ruby is a language whose meaning we already understand, we can see that both of these strings are programs that build procs. Each proc takes an environment argument called e, completely ignores it, and returns a Ruby value.

Because these denotations are strings of Ruby source code, we can check their behavior in IRB by using Kernel#eval to turn them into real, callable Proc objects:[18]

```
>> proc = eval(Number.new(5).to_ruby)
=> #<Proc (lambda)>
>> proc.call({})
=> 5
>> proc = eval(Boolean.new(false).to_ruby)
=> #<Proc (lambda)>
>> proc.call({})
=> false
```

WARNING

At this stage, it's tempting to avoid procs entirely and use simpler implementations of #to_ruby that just turn Number.new(5) into the string '5' instead of '-> e { 5 }' and so on, but part of the point of building a denotational semantics is to capture the essence of constructs from the source language, and in this case, we're capturing the idea that expressions *in general* require an environment, even though these specific expressions don't make use of it.

To denote expressions that do use the environment, we need to decide how environments are going to be represented in Ruby. We've already seen environments in our operational semantics, and since they were implemented in Ruby, we can just reuse our earlier idea of representing an environment as a hash. The details will need to change, though, so beware the subtle difference: in our operational semantics, the environment lived inside the virtual machine and associated variable names with SIMPLE abstract syntax trees like Number.new(5), but in our denotational semantics, the environment exists in the language we're

Copyrighted material

translating our programs into, so it needs to make sense in that world instead of the "outside world" of a virtual machine.

In particular, this means that our denotational environments should associate variable names with native Ruby values like `5` rather than with objects representing SIMPLE syntax. We can think of an operational environment like `{ x: Number.new(5) }` as having a denotation of `'{ x: 5 }'` in the language we're translating into, and we just need to keep our heads straight because both the implementation metalanguage and the denotation language happen to be Ruby.

Now we know that the environment will be a hash, we can implement `Variable#to_ruby`:

```
class Variable
  def to_ruby
    "-> e { e[#{name.inspect}] }"
  end
end
```

This translates a variable expression into the source code of a Ruby proc that looks up the appropriate value in the environment hash:

```
>> expression = Variable.new(:x)
=> «x»
>> expression.to_ruby
=> "-> e { e[:x] }"
>> proc = eval(expression.to_ruby)
=> #<Proc (lambda)>
>> proc.call({ x: 7 })
=> 7
```

An important aspect of denotational semantics is that it's *compositional*: the denotation of a program is constructed from the denotations of its parts. We can see this compositionality in practice when we move onto denoting larger expressions like `Add`, `Multiply`, and `LessThan`:

Copyrighted material

```
class Add
  def to_ruby
    "-> e { (#{left.to_ruby}).call(e) + (#{right.to_ruby}).call(e) }"
  end
end

class Multiply
  def to_ruby
    "-> e { (#{left.to_ruby}).call(e) * (#{right.to_ruby}).call(e) }"
  end
end

class LessThan
  def to_ruby
    "-> e { (#{left.to_ruby}).call(e) < (#{right.to_ruby}).call(e) }"
  end
end
```

Here we're using string concatenation to compose the denotation of an expression out of the denotations of its subexpressions. We know that each subexpression will be denoted by a proc's Ruby source, so we can use them as part of a larger piece of Ruby source that calls those procs with the supplied environment and does some computation with their return values. Here's what the resulting denotations look like:

```
>> Add.new(Variable.new(:x), Number.new(1)).to_ruby
=> "-> e { (-> e { e[:x] }).call(e) + (-> e { 1 }).call(e) }"
>> LessThan.new(Add.new(Variable.new(:x), Number.new(1)),
Number.new(3)).to_ruby
=> "-> e { (-> e { (-> e { e[:x] }).call(e) + (-> e { 1 }).call(e)
}).call(e) < (-> e { 3 }).call(e) }"
```

These denotations are now complicated enough that it's difficult to see whether they do the right thing. Let's try them out to make sure:

```
>> environment = { x: 3 }
=> {:x=>3}
```

Copyrighted material

```
>> proc = eval(Add.new(Variable.new(:x), Number.new(1)).to_ruby)
=> #<Proc (lambda)>
>> proc.call(environment)
=> 4
>> proc = eval(
     LessThan.new(Add.new(Variable.new(:x), Number.new(1)),
Number.new(3)).to_ruby
   )
=> #<Proc (lambda)>
>> proc.call(environment)
=> false
```

Statements

We can specify the denotational semantics of statements in a similar way, although remember from the operational semantics that evaluating a statement produces a new environment rather than a value. This means that `Assign#to_ruby` needs to produce code for a proc whose result is an updated environment hash:

```
class Assign
  def to_ruby
    "-> e { e.merge({ #{name.inspect} => (#{expression.to_ruby}).call(e)
}) }"
  end
end
```

Again, we can check this on the console:

```
>> statement = Assign.new(:y, Add.new(Variable.new(:x), Number.new(1)))
=> «y = x + 1»
>> statement.to_ruby
=> "-> e { e.merge({ :y => (-> e { (-> e { e[:x] }).call(e) + (-> e { 1
}).call(e) }).call(e) }) }"
>> proc = eval(statement.to_ruby)
=> #<Proc (lambda)>
>> proc.call({ x: 3 })
```

Copyrighted material

```
=> {:x=>3, :y=>4}
```

As always, the semantics of `DoNothing` is very simple:

```
class DoNothing
  def to_ruby
    '-> e { e }'
  end
end
```

For conditional statements, we can translate SIMPLE's «`if (…) { … } else { … }`» into a Ruby `if … then … else … end`, making sure that the environment gets to all the places where it's needed:

```
class If
  def to_ruby
    "-> e { if (#{condition.to_ruby}).call(e)" +
      " then (#{consequence.to_ruby}).call(e)" +
      " else (#{alternative.to_ruby}).call(e)" +
      " end }"
  end
end
```

As in big-step operational semantics, we need to be careful about specifying the sequence statement: the result of evaluating the first statement is used as the environment for evaluating the second.

```
class Sequence
  def to_ruby
    "-> e { (#{second.to_ruby}).call((#{first.to_ruby}).call(e)) }"
  end
end
```

And lastly, as with conditionals, we can translate «`while`» statements into procs that use Ruby `while` to repeatedly execute the body before returning the final

Copyrighted material

environment:

```
class While
  def to_ruby
    "-> e {" +
      " while (#{condition.to_ruby}).call(e); e = (#
{body.to_ruby}).call(e); end;" +
      " e" +
      " }"
  end
end
```

Even a simple «while» can have quite a verbose denotation, so it's worth getting the Ruby interpreter to check that its meaning is correct:

```
>> statement =
     While.new(
       LessThan.new(Variable.new(:x), Number.new(5)),
       Assign.new(:x, Multiply.new(Variable.new(:x), Number.new(3)))
     )
=> «while (x < 5) { x = x * 3 }»
>> statement.to_ruby
=> "-> e { while (-> e { (-> e { e[:x] }).call(e) < (-> e { 5 }).call(e)
}).call(e); e = (-> e { e.merge({ :x => (-> e { (-> e { e[:x] }).call(e) *
(-> e { 3 }).call(e) }).call(e) }) }).call(e); end; e }"
>> proc = eval(statement.to_ruby)
=> #<Proc (lambda)>
>> proc.call({ x: 1 })
=> {:x=>9}
```

Copyrighted material

COMPARING SEMANTIC STYLES

«while» is a good example of the difference between small-step, big-step, and denotational semantics.

The small-step operational semantics of «while» is written as a reduction rule for an abstract machine. The overall looping behavior isn't part of the rule's action—reduction just turns a «while» statement into an «if» statement—but it emerges as a consequence of the future reductions performed by the machine. To understand what «while» does, we need to look at all of the small-step rules and work out how they interact over the course of a SIMPLE program's execution.

«while»'s big-step operational semantics is written as an evaluation rule that shows how to compute the final environment directly. The rule contains a recursive call to itself, so there's an explicit indication that «while» will cause a loop during evaluation, but it's not quite the kind of loop that a SIMPLE programmer would recognize. Big-step rules are written in a recursive style, describing the complete evaluation of an expression or statement in terms of the evaluation of other pieces of syntax, so this rule tells us that the result of evaluating a «while» statement may depend upon the result of evaluating the same statement in a different environment, but it requires a leap of intuition to connect this idea with the iterative behavior that «while» is supposed to exhibit. Fortunately the leap isn't too large: a bit of mathematical reasoning can show that the two kinds of loop are equivalent in principle, and when the metalanguage supports tail call optimization, they're also equivalent in practice.

The denotational semantics of «while» shows how to rewrite it in Ruby, namely by using Ruby's `while` keyword. This is a much more direct translation: Ruby has native support for iterative loops, and the denotation rule shows that «while» can be implemented with that feature. There's no leap required to understand how the two kinds of loop relate to each other, so if we understand how Ruby `while` loops work, we understand SIMPLE «while» loops too. Of course, this means we've just converted the problem of understanding SIMPLE into the problem of understanding the denotation language, which is a serious disadvantage when that language is as large and ill-specified as Ruby, but it becomes an advantage when we have a small mathematical language for writing denotations.

Applications

Having done all this work, what does this denotational semantics achieve? Its main purpose is to show how to translate SIMPLE into Ruby, using the latter as a tool to explain what various language constructs mean. This happens to give us a

Copyrighted material

way to execute SIMPLE programs—because we've written the rules of the denotational semantics in executable Ruby, and because the rules' output is itself executable Ruby—but that's incidental, since we could have given the rules in plain English and used some mathematical language for the denotations. The important part is that we've taken an arbitrary language of our own devising and converted it into a language that someone or something else can understand.

To give this translation some explanatory power, it's helpful to bring parts of the language's meaning to the surface instead of allowing them to remain implicit. For example, this semantics makes the environment explicit by representing it as a tangible Ruby object—a hash that's passed in and out of procs—instead of denoting variables as real Ruby variables and relying on Ruby's own subtle scoping rules to specify how variable access works. In this respect the semantics is doing more than just offloading all the explanatory effort onto Ruby; it uses Ruby as a simple foundation, but does some extra work on top to show exactly how environments are used and changed by different program constructs.

We saw earlier that operational semantics is about explaining a language's meaning by designing an interpreter for it. By contrast, the language-to-language translation of denotational semantics is like a *compiler*: in this case, our implementations of `#to_ruby` effectively compile SIMPLE into Ruby. None of these styles of semantics necessarily says anything about how to *efficiently* implement an interpreter or compiler for a language, but they do provide an official baseline against which the correctness of any efficient implementation can be judged.

These denotational definitions also show up in the wild. Older versions of the Scheme standard use denotational semantics to specify the core language, unlike the current standard's small-step operational semantics, and the development of the XSLT document-transformation language was guided by Philip Wadler's denotational definitions of XSLT patterns and XPath expressions.

See Semantics for a practical example of using denotational semantics to specify the meaning of regular expressions.

Formal Semantics in Practice

Copyrighted material

This chapter has shown several different ways of approaching the problem of giving computer programs a meaning. In each case, we've avoided the mathematical details and tried to get a flavor of their intent by using Ruby, but formal semantics is usually done with mathematical tools.

Formality

Our tour of formal semantics hasn't been especially formal. We haven't paid any serious attention to mathematical notation, and using Ruby as a metalanguage has meant we've focused more on different ways of executing programs than on ways of understanding them. Proper denotational semantics is concerned with getting to the heart of programs' meanings by turning them into well-defined mathematical objects, with none of the evasiveness of representing a SIMPLE «`while`» loop with a Ruby `while` loop.

> **NOTE**
>
> The branch of mathematics called *domain theory* was developed specifically to provide definitions and objects that are useful for denotational semantics, allowing a model of computation based on fixed points of monotonic functions on partially ordered sets. Programs can be understood by "compiling" them into mathematical functions, and the techniques of domain theory can be used to prove interesting properties of these functions.

On the other hand, while we only vaguely sketched denotational semantics in Ruby, our approach to operational semantics is closer in spirit to its formal presentation: our definitions of `#reduce` and `#evaluate` methods are really just Ruby translations of mathematical inference rules.

Finding Meaning

An important application of formal semantics is to give an unambiguous specification of the meaning of a programming language, rather than relying on more informal approaches like natural-language specification documents and "specification by implementation." A formal specification has other uses too,

Copyrighted material

such as proving properties of the language in general and of specific programs in particular, proving equivalences between programs in the language, and investigating ways of safely transforming programs to make them more efficient without changing their behavior.

For example, since an operational semantics corresponds quite closely to the implementation of an interpreter, computer scientists can treat a suitable interpreter as an operational semantics for a language, and then prove its correctness with respect to a denotational semantics for that language—this means proving that there is a sensible connection between the meanings given by the interpreter and those given by the denotational semantics.

Denotational semantics has the advantage of being more abstract than operational semantics, by ignoring the detail of how a program executes and concentrating instead on how to convert it into a different representation. For example, this makes it possible to compare two programs written in different languages, if a denotational semantics exists to translate both languages into some shared representation.

This degree of abstraction can make denotational semantics seem circuitous. If the problem is how to explain the meaning of a programming language, how does translating one language into another get us any closer to a solution? A denotation is only as good as its meaning; in particular, a denotational semantics only gets us closer to being able to actually execute a program if the denotation language has some *operational* meaning, a semantics of its own that shows how it may be executed instead of how to translate it into yet another language.

Formal denotational semantics uses abstract mathematical objects, usually functions, to denote programming language constructs like expressions and statements, and because mathematical convention dictates how to do things like evaluate functions, this gives a direct way of thinking about the denotation in an operational sense. We've taken the less formal approach of thinking of a denotational semantics as a compiler from one language into another, and in reality, this is how most programming languages ultimately get executed: a Java program will get compiled into bytecode by `javac`, the bytecode will get just-in-time compiled into x86 instructions by the Java virtual machine, then a CPU will decode each x86 instruction into RISC-like microinstructions for execution on a core...where does it end? Is it compilers, or virtual machines, all the way down?

Of course programs do eventually execute, because the tower of semantics

Copyrighted material

finally bottoms out at an *actual* machine: electrons in semiconductors, obeying the laws of physics.[19] A computer is a device for maintaining this precarious structure, many complex layers of interpretation balanced on top of one another, allowing human-scale ideas like multitouch gestures and `while` loops to be gradually translated down into the physical universe of silicon and electricity.

Alternatives

The semantic styles seen in this chapter go by many different names. Small-step semantics is also known as *structural operational semantics* and *transition semantics*; big-step semantics is more often called *natural semantics* or *relational semantics*; and denotational semantics is also called *fixed-point semantics* or *mathematical semantics*.

Other styles of formal semantics are available. One alternative is *axiomatic semantics*, which describes the meaning of a statement by making assertions about the state of the abstract machine before and after that statement executes: if one assertion (the *precondition*) is initially true before the statement is executed, then the other assertion (the *postcondition*) will be true afterward. Axiomatic semantics is useful for verifying the correctness of programs: as statements are plugged together to make larger programs, their corresponding assertions can be plugged together to make larger assertions, with the goal of showing that an overall assertion about a program matches up with its intended specification.

Although the details are different, axiomatic semantics is the style that best characterizes the RubySpec project, an “executable specification for the Ruby programming language” that uses RSpec-style assertions to describe the behavior of Ruby’s built-in language constructs, as well as its core and standard libraries. For example, here’s a fragment of RubySpec’s description of the `Array#<<` method:

```
describe "Array#<<" do
  it "correctly resizes the Array" do
    a = []
    a.size.should == 0
    a << :foo
    a.size.should == 1
```

Copyrighted material

```
    a << :bar << :baz
    a.size.should == 3

    a = [1, 2, 3]
    a.shift
    a.shift
    a.shift
    a << :foo
    a.should == [:foo]
  end
end
```

Implementing Parsers

In this chapter, we've been building the abstract syntax trees of SIMPLE programs manually—writing longhand Ruby expressions like `Assign.new(:x, Add.new(Variable.new(:x), Number.new(1)))`—rather than beginning with raw SIMPLE source code like `'x = x + 1'` and using a parser to automatically turn it into a syntax tree.

Implementing a SIMPLE parser entirely from scratch would involve a lot of detail and take us on a long diversion from our discussion of formal semantics. Hacking on toy programming languages is fun, though, and thanks to the existence of parsing tools and libraries it's not especially difficult to construct a parser by relying on other people's work, so here's a brief outline of how to do it.

One of the best parsing tools available for Ruby is Treetop, a domain-specific language for describing syntax in a way that allows a parser to be automatically generated. A Treetop description of a language's syntax is written as a *parsing expression grammar* (PEG), a collection of simple, regular-expression-like rules that are easy to write and to understand. Best of all, these rules can be annotated with method definitions so that the Ruby objects generated by the parsing process can be given their own behavior. This ability to define both a syntactic structure and a collection of Ruby code that operates on that structure makes Treetop ideal for sketching out the syntax of a language and giving it an executable semantics.

To give us a taste of how this works, here's a cut-down version of the Treetop

Copyrighted material

grammar for SIMPLE, containing only the rules needed to parse the string `'while (x < 5) { x = x * 3 }'`:

```
grammar Simple
  rule statement
    while / assign
  end

  rule while
    'while (' condition:expression ') { ' body:statement ' }' {
      def to_ast
        While.new(condition.to_ast, body.to_ast)
      end
    }
  end

  rule assign
    name:[a-z]+ ' = ' expression {
      def to_ast
        Assign.new(name.text_value.to_sym, expression.to_ast)
      end
    }
  end

  rule expression
    less_than
  end

  rule less_than
    left:multiply ' < ' right:less_than {
      def to_ast
        LessThan.new(left.to_ast, right.to_ast)
      end
    }
    /
    multiply
  end
```

Copyrighted material

```
    rule multiply
      left:term ' * ' right:multiply {
        def to_ast
          Multiply.new(left.to_ast, right.to_ast)
        end
      }
      /
      term
    end

    rule term
      number / variable
    end

    rule number
      [0-9]+ {
        def to_ast
          Number.new(text_value.to_i)
        end
      }
    end

    rule variable
      [a-z]+ {
        def to_ast
          Variable.new(text_value.to_sym)
        end
      }
    end
  end
```

This language looks a little like Ruby, but the similarity is only superficial; grammars are written in the special Treetop language. The `rule` keyword introduces a new rule for parsing a particular kind of syntax, and the expressions inside each rule describe the structure of the strings it will recognize. Rules can recursively call other rules—the `while` rule calls the `expression` and `statement` rules, for instance—and parsing begins at the first rule, which is `statement` in this grammar.

Copyrighted material

The order in which the expression-syntax rules call each other reflects the precedence of SIMPLE's operators. The `expression` rule calls `less_than`, which then immediately calls `multiply` to give it a chance to match the `*` operator somewhere in the string before `less_than` gets a chance to match the lower-precedence `<` operator. This makes sure that `'1 * 2 < 3'` is parsed as «`(1 * 2) < 3`» and not «`1 * (2 < 3)`».

> **WARNING**
>
> To keep things simple, this grammar makes no attempt to constrain what kinds of expression can appear inside other expressions, which means the parser will accept some programs that are obviously wrong.
>
> For example, we have two rules for binary expressions—`less_than` and `multiply`—but the only reason for having separate rules is to enforce operator precedence, so each rule only requires that a higher precedence rule matches its left operand and a same-or-higher-precedence one matches its right. This creates the situation where a string like `'1 < 2 < 3'` will be parsed successfully, even though the semantics of SIMPLE won't be able to give the resulting expression a meaning.
>
> Some of these problems can be resolved by tweaking the grammar, but there will always be other incorrect cases that the parser can't spot. We'll separate the two concerns by keeping the parser as liberal as possible and using a different technique to detect invalid programs in Chapter 9.

Most of the rules in the grammar are annotated with Ruby code inside curly brackets. In each case, this code defines a method called `#to_ast`, which will be available on the corresponding syntax objects built by Treetop when it parses a SIMPLE program.

If we save this grammar into a file called *simple.treetop*, we can load it with Treetop to generate a `SimpleParser` class. This parser allows us to turn a string of SIMPLE source code into a representation built out of Treetop's `SyntaxNode` objects:

```
>> require 'treetop'
```

Copyrighted material

```
=> true
>> Treetop.load('simple')
=> SimpleParser
>> parse_tree = SimpleParser.new.parse('while (x < 5) { x = x * 3 }')
=> SyntaxNode+While1+While0 offset=0, "…5) { x = x * 3 }"
(to_ast,condition,body):
     SyntaxNode offset=0, "while ("
     SyntaxNode+LessThan1+LessThan0 offset=7, "x < 5" (to_ast,left,right):
       SyntaxNode+Variable0 offset=7, "x" (to_ast):
         SyntaxNode offset=7, "x"
       SyntaxNode offset=8, " < "
       SyntaxNode+Number0 offset=11, "5" (to_ast):
         SyntaxNode offset=11, "5"
     SyntaxNode offset=12, ") { "
     SyntaxNode+Assign1+Assign0 offset=16, "x = x * 3"
(to_ast,name,expression):
       SyntaxNode offset=16, "x":
         SyntaxNode offset=16, "x"
       SyntaxNode offset=17, " = "
       SyntaxNode+Multiply1+Multiply0 offset=20, "x * 3"
(to_ast,left,right):
         SyntaxNode+Variable0 offset=20, "x" (to_ast):
           SyntaxNode offset=20, "x"
         SyntaxNode offset=21, " * "
         SyntaxNode+Number0 offset=24, "3" (to_ast):
           SyntaxNode offset=24, "3"
     SyntaxNode offset=25, " }"
```

This SyntaxNode structure is a *concrete syntax tree*: it's designed specifically for manipulation by the Treetop parser and contains a lot of extraneous information about how its nodes are related to the raw source code that produced them. Here's what the Treetop documentation has to say about it:

> *Please don't try to walk down the syntax tree yourself, and please don't use the tree as your own convenient data structure. It contains many more nodes than your application needs, often even more than one for every character of input.*
>
> *Instead, add methods to the root rule that return the information you require in a sensible form. Each rule can call its sub-rules, and this method of walking the syntax tree is much preferable to attempting to walk it from the outside.*

Copyrighted material

And that's what we've done. Rather than manipulate this messy tree directly, we've used annotations in the grammar to define a `#to_ast` method on each of its nodes. If we call that method on the root node, it'll build an abstract syntax tree made from SIMPLE syntax objects:

```
>> statement = parse_tree.to_ast
=> «while (x < 5) { x = x * 3 }»
```

So we've automatically converted source code to an abstract syntax tree, and now we can use that tree to explore the meaning of the program in the usual ways:

```
>> statement.evaluate({ x: Number.new(1) })
=> {:x=>«9»}
>> statement.to_ruby
=> "-> e { while (-> e { (-> e { e[:x] }).call(e) < (-> e { 5 }).call(e)
}).call(e); e = (-> e { e.merge({ :x => (-> e { (-> e { e[:x] }).call(e) *
(-> e { 3 }).call(e) }).call(e) }) }).call(e); end; e }"
```

Copyrighted material

WARNING

Another drawback of this parser, and of Treetop in general, is that it generates a *right-associative* concrete syntax tree. This means that the string `'1 * 2 * 3 * 4'` is parsed as if it had been written `'1 * (2 * (3 * 4))'`:

```
>> expression = SimpleParser.new.parse('1 * 2 * 3 * 4', root:
:expression).to_ast
=> «1 * 2 * 3 * 4»
>> expression.left
=> «1»
>> expression.right
=> «2 * 3 * 4»
```

But multiplication is conventionally *left-associative*: when we write `'1 * 2 * 3 * 4'` we actually mean `'((1 * 2) * 3) * 4'`, with the numbers grouped together starting at the lefthand end of the expression, not the right. That doesn't matter much for multiplication—both ways produce the same result when evaluated—but for operations like subtraction and division, it creates a problem, because «`((1 - 2) - 3) - 4`» does *not* evaluate to the same value as «`1 - (2 - (3 - 4))`».

To fix this, we'd have to make the rules and `#to_ast` implementations more complicated. See Parsing for a Treetop grammar that builds a left-associative AST.

It's convenient to be able to parse Simple programs like this, but Treetop is doing all the hard work for us, so we haven't learned much about how a parser actually works. In Parsing with Pushdown Automata, we'll see how to implement a parser directly.

---

[2] In the context of programming language theory, the word *semantics* is usually treated as singular: we describe the meaning of a language by giving it *a semantics*.

[3] Although access to ISO/IEC 30170 costs money, an earlier draft of the same specification can be downloaded for free from http://www.ipa.go.jp/osc/english/ruby/.

[4] This can be an abbreviation for simple imperative language if you want it to be.

Copyrighted material

[5] For the sake of simplicity, we'll resist the urge to extract common code into superclasses or modules.

[6] Although this is pretty much exactly how we'd write `#reducible?` in a functional language like Haskell or ML.

[7] At the moment, it doesn't make any difference *which* order we choose, but we can't avoid making the decision.

[8] This conditional is not the same as Ruby's `if`. In Ruby, `if` is an expression that returns a value, but in SIMPLE, it's a statement for choosing which of two other statements to evaluate, and its only result is the effect it has on the current environment.

[9] For our purposes, it doesn't matter whether this statement has been constructed as «`(x = 1 + 1; y = x + 3); z = y + 5`» or «`x = 1 + 1; (y = x + 3; z = y + 5)`». This choice would affect the exact order of the reduction steps when we ran it, but the final result would be the same either way.

[10] We can already hardcode a fixed number of repetitions by using sequence statements, but that doesn't allow us to control the repetition behavior at runtime.

[11] There's a temptation to build the iterative behavior of «`while`» directly into its reduction rule instead of finding a way to get the abstract machine to handle it, but that's not how small-step semantics works. See Big-Step Semantics for a style of semantics that lets the rules do the work.

[12] Ruby's procs permit complex expressions to be assigned to variables in some sense, but a proc is still a value: it can't perform any more evaluation by itself, but can be reduced as part of a larger expression involving other values.

[13] Reducing an expression and an environment gives us a new expression, and we may reuse the old environment next time; reducing a statement and an environment gives us a new statement and a new environment.

[14] Our Ruby implementation of big-step semantics won't be ambiguous in this way, because Ruby itself already makes these ordering decisions, but when a big-step semantics is specified mathematically, it can avoid spelling out the exact evaluation strategy.

[15] Of course, there's nothing to prevent SIMPLE programmers from writing a «`while`» statement whose condition never becomes «`false`» anyway, but if that's what they ask for then that's what they're going to get.

[16] There is an alternative style of operational semantics, called *reduction semantics*, which explicitly separates these "what do we reduce next?" and "how do we reduce it?" phases by introducing so-called *reduction contexts*. These contexts are just patterns that concisely describe the places in a program where reduction can happen, meaning we only need to

Copyrighted material

write reduction rules that perform real computation, thereby eliminating some of the boilerplate from the semantic definitions of larger languages.

[17] This means we'll be writing Ruby code that generates Ruby code, but the choice of the same language as both the denotation language and the implementation metalanguage is only to keep things simple. We could just as easily write Ruby that generates strings containing JavaScript, for example.

[18] We can only do this because Ruby is doing double duty as both the implementation and denotation languages. If our denotations were JavaScript source code, we'd have to try them out in a JavaScript console.

[19] Or, in the case of a mechanical computer like the Analytical Engine designed by Charles Babbage in 1837, cogs and paper obeying the laws of physics.

Copyrighted material

Chapter 3. The Simplest Computers

In the space of a few short years, we've become surrounded by computers. They used to be safely hidden away in military research centers and university laboratories, but now they're everywhere: on our desks, in our pockets, under the hoods of our cars, implanted inside our bodies. As programmers, we work with sophisticated computing devices every day, but how well do we understand the way they work?

The power of modern computers comes with a lot of complexity. It's difficult to understand every detail of a computer's many subsystems, and more difficult still to understand how those subsystems interact to create the system as a whole. This complexity makes it impractical to reason directly about the capabilities and behavior of real computers, so it's useful to have simplified models of computers that share interesting features with real machines but that can still be understood in their entirety.

In this chapter, we'll strip back the idea of a computing machine to its barest essentials, see what it can be used for, and explore the limits of what such a simple computer can do.

Deterministic Finite Automata

Real computers typically have large amounts of volatile memory (RAM) and nonvolatile storage (hard drive or SSD), many input/output devices, and several processor cores capable of executing multiple instructions simultaneously. A *finite state machine*, also known as a *finite automaton*, is a drastically simplified model of a computer that throws out all of these features in exchange for being easy to understand, easy to reason about, and easy to implement in hardware or software.

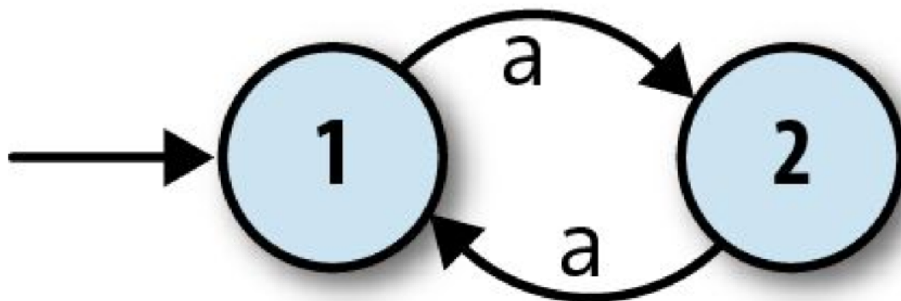
States, Rules, and Input

Copyrighted material

A finite automaton has no permanent storage and virtually no RAM. It's a little machine with a handful of possible *states* and the ability to keep track of which one of those states it's currently in—think of it as a computer with enough RAM to store a single value. Similarly, finite automata don't have a keyboard, mouse, or network interface for receiving input, just a single external stream of input characters that they can read one at a time.

Instead of a general-purpose CPU for executing arbitrary programs, each finite automaton has a hardcoded collection of *rules* that determine how it should move from one state to another in response to input. The automaton starts in one particular state and reads individual characters from its input stream, following a rule each time it reads a character.

Here's a way of visualizing the structure of one particular finite automaton:



The two circles represent the automaton's two states, 1 and 2, and the arrow coming from nowhere shows that the automaton always starts in state 1, its *start state*. The arrows between states represent the rules of the machine, which are:

- When in state 1 and the character a is read, move into state 2.
- When in state 2 and the character a is read, move into state 1.

This is enough information for us to investigate how the machine processes a stream of inputs:

- The machine starts in state 1.
- The machine only has rules for reading the character a from its input stream, so that's the only thing that can happen. When it reads an a, it moves from state 1 into state 2.

Copyrighted material