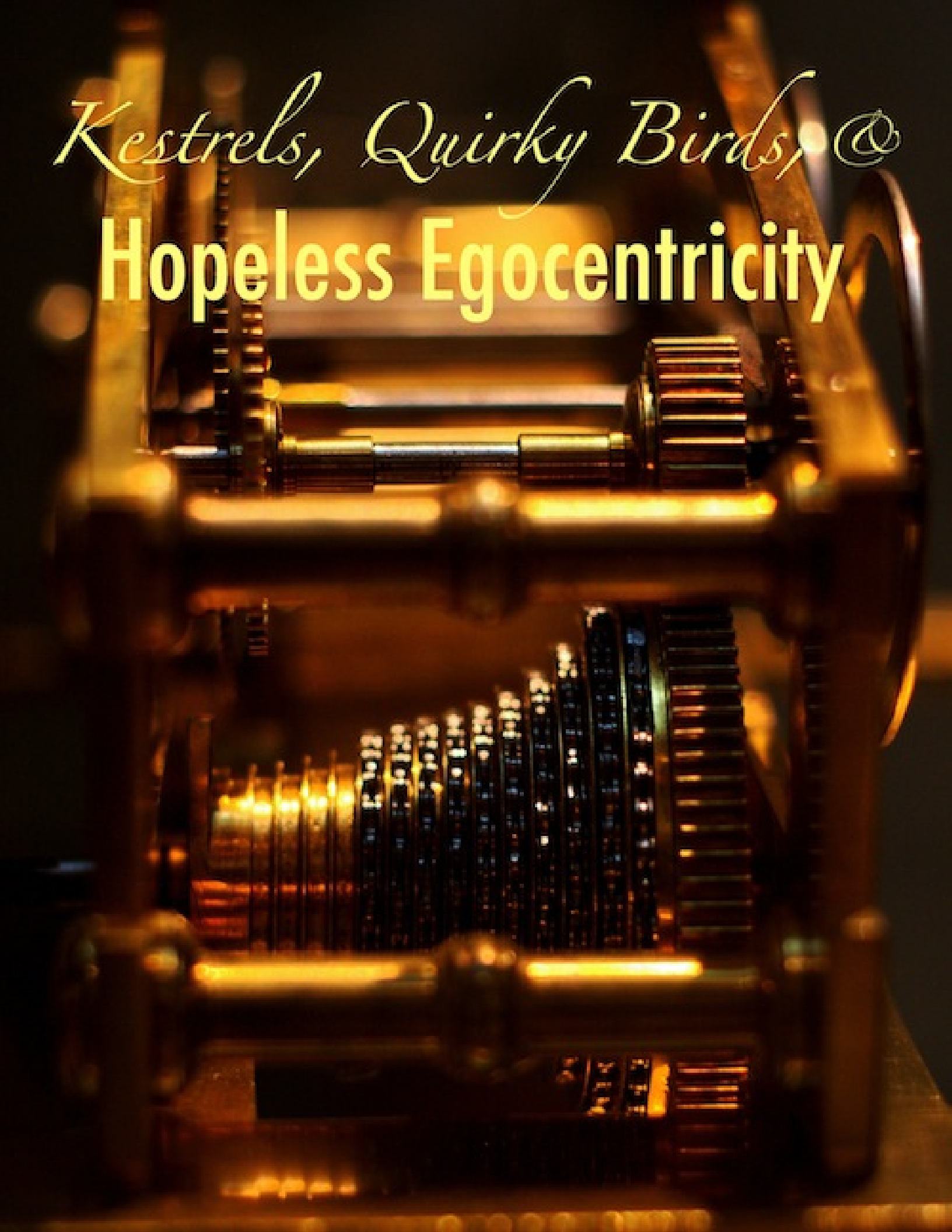
Kestrels, Quirky Birds, &
Hopeless Egocentricity

# Kestrels, Quirky Birds, and Hopeless Egocentricity

## Raganwald's collected adventures in Combinatory Logic and Ruby Meta-Programming

Reginald Braithwaite

This book is for sale at http://leanpub.com/combinators

This version was published on 2013-10-01

# Tweet This Book!

Please help Reginald Braithwaite by spreading the word about this book on Twitter!

The suggested hashtag for this book is #combinators.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search/#combinators

# Also By Reginald Braithwaite

What I've Learned From Failure

How to Do What You Love & Earn What You're Worth as a Programmer

CoffeeScript Ristretto

JavaScript Allongé

# Contents

## 0.1 The MIT License



## 0.2 Preface

The chapters of this book originally appeared as blog posts. You can still read them online, for free, at http://github.com/raganwald/homoiconic[1]. The original posts were released under the MIT license, you you can pass them around or incorporate them into your own works as you see fit. I decided to publish these essays as an e-book as well as online. This format doesn't replace the original online essays, it's a way to present these essays in a more coherent whole that's easier to read consecutively. I hope you like it.

–Reginald "Raganwald" Braithwaite[2], Toronto, November 2011

[1] http://github.com/raganwald/homoiconic
[2] http://braythwayt.com

# 1 Introduction

Like the Lambda Calculus, Combinatory Logic[1] is a mathematical notation that is powerful enough to handle set theory and issues in computability.

> Combinatory logic is a notation introduced by Moses Sch□□nfinkel[2] and Haskell Curry[3] to eliminate the need for variables in mathematical logic. It has more recently been used in computer science as a theoretical model of computation and also as a basis for the design of functional programming languages. It is based on combinators. A combinator is a higher-order function that uses only function application and earlier defined combinators to define a result from its arguments.

In this book, we're going to meet some of the standard combinators, and for each one we'll explore some of its ramifications when writing programs using the Ruby programming language. In Combinatory logic, combinators combine and alter each other, and our Ruby examples will focus on combining and altering Ruby code. From simple examples like the K Combinator and Ruby's `.tap` method, we'll work our way up to meta-programming with aspects and recursive combinators.

**about the bird names**

When Combinatory Logic was first invented by Haskell Curry, the standard combinators were given upper-case letters. For example, the two combinators needed to express everything in the Lambda Calculus and in Set Theory are the `S` and `K` combinators. In 1985, Raymond Smullyan published To Mock a Mockingbird[4], an exploration of combinatory logic for the recreational layman. Smullyan used a forest full of songbirds as a metaphor, with each of the combinators given the name of a songbird rather than a single letter. For example, the `S` and `K` combinators became the Starling and Kestrel, the `I` combinator became the Idiot bird, and so forth.

These ornithological nicknames have become part of the standard lexicon for combinatory logic.

**thanks**

There are too many people to name,but amongst the crowd, Alan Smith stands out.

[1] http://en.wikipedia.org/wiki/Combinatory_logic

[2] http://en.wikipedia.org/wiki/Moses_Sch□□nfinkel

[3] http://en.wikipedia.org/wiki/Haskell_Curry

[4] http://www.amazon.com/gp/product/B00A1P096Y/ref=as_li_ss_tl?ie=UTF8&camp=1789&creative=390957&creativeASIN=B00A1P096Y&linkCode=as2&tag=raganwald001-20

# 2 Kestrels

In Combinatory Logic, a Kestrel (or "K Combinator") is a function that returns a constant function, normally written `Kxy = x`. In Ruby, it might look like this:

```
# for *any* x,
kestrel.call(:foo).call(x)
  => :foo
```

Kestrels are to be found in Ruby. You may be familiar with their Ruby 1.9 name, `#tap`. Let's say you have a line like `address = Person.find(...).address` and you wish to log the person instance. With `tap`, you can inject some logging into the expression without messy temporary variables:

```
address = Person.find(...).tap { |p| logger.log "person #{p} found" }.address
```

`tap` is a method in all objects that passes `self` to a block and returns self, ignoring whatever the last item of the block happens to be. Ruby on Rails programmers will recognize the Kestrel in slightly different form:

```
address = returning Person.find(...) do |p|
  logger.log "person #{p} found"
end.address
```

Again, the result of the block is discarded, it is only there for side effects. This behaviour is the same as a Kestrel. Remember `kestrel.call(:foo).call(x)`? If I rewrite it like this, you can see the similarity:

```
Kestrel.call(:foo) do
  x
end
  => :foo
```

Both `returning` and `tap` are handy for grouping side effects together. Methods that look like this:

```
def registered_person(params = {})
  person = Person.new(params.merge(:registered => true))
  Registry.register(person)
  person.send_email_notification
  person
end
```

Can be rewritten using `returning`:

```
def registered_person(params = {})
  returning Person.new(params.merge(:registered => true)) do |person|
    Registry.register(person)
    person.send_email_notification
  end
end
```

It is obvious from the first line what will be returned and it eliminates an annoying error when the programmer neglects to make `person` the last line of the method.

## 2.1 Object initializer blocks

The Kestrel has also been sighted in the form of *object initializer blocks*. Consider this example using Struct[1]:

```
Contact = Struct.new(:first, :last, :email) do
  def to_hash
    Hash[*members.zip(values).flatten]
  end
end
```

The method `Struct#new` creates a new class. It also accepts an optional block, evaluating the block for side effects only. It returns the new class regardless of what happens to be in the block (it happens to evaluate the block in class scope, a small refinement).

You can use this technique when writing your own classes:

[1] http://blog.grayproductions.net/articles/all_about_struct

```
class Bird < Creature
  def initialize(*params)
    # do something with the params
    yield self if block_given?
  end
end

Forest.add(
        Bird.new(:name => 'Kestrel) { |k| combinators << k }
)
```

The pattern of wanting a Kestrel/returning/tap when you create a new object is so common that building it into object initialization is useful. And in fact, it's built into `ActiveRecord`. Methods like `new` and `create` take optional blocks, so you can write:

```
class Person < ActiveRecord::Base
  # ...
end

def registered_person(params = {})
  Person.new(params.merge(:registered => true)) do |person|
    Registry.register(person)
    person.send_email_notification
  end
end
```

In Rails, `returning` is not necessary when creating instances of your model classes, thanks to ActiveRecord's built-in object initializer blocks.

## 2.2 Inside, an idiomatic Ruby Kestrel

When we discussed `Struct` above, we noted that its initializer block has a slightly different behaviour than `tap` or `returning`. It takes an initializer block, but it doesn't pass the new class to the block as a parameter, it evaluates the block in the context of the new class.

Putting this into implementation terms, it evaluates the block with `self` set to the new class. This is not the same as `returning` or `tap`, both of which leave `self` untouched. We can write our own version of `returning` with the same semantics. We will call it `inside`:

```
module Kernel

  def inside(value, &block)
    value.instance_eval(&block)
    value
  end

end
```

You can use this variation on a Kestrel just like `returning`, only you do not need to specify a parameter:

```
inside [1, 2, 3] do
  uniq!
end
  => [1, 2, 3]
```

This isn't particularly noteworthy. Of more interest is your access to private methods and instance variables:

```
sna = Struct.new('Fubar') do
  attr_reader :fu
end.new

inside(sna) do
  @fu = 'bar'
end
  => #<struct Struct::Fubar >

sna.fu
  => 'bar'
```

`inside` is a Kestrel just like `returning`. No matter what value its block generates, it returns its primary argument. The only difference between the two is the evaluation environment of the block.

## 2.3 The Enchaining Kestrel

In **Kestrels**, we looked at `#tap` from Ruby 1.9 and `returning` from Ruby on Rails. No we'll going to look at another use for `tap`. As already explained, Ruby 1.9 includes the new method `Object#tap`. It passes the receiver to a block, then returns the receiver no matter what the block contains. The canonical example inserts some logging in the middle of a chain of method invocations:

```
address = Person.find(...).tap { |p| logger.log "person #{p} found" }.address
```

`Object#tap` is also useful when you want to execute several method on the same object without having to create a lot of temporary variables, a practice Martin Fowler calls [Method Chaining](http://martinfowler.com/dslwip/MethodChaining.html “”). Typically, you design such an object so that it returns itself in response to every modifier message. This allows you to write things like:

```
HardDrive.new.capacity(150).external.speed(7200)
```

Instead of:

```
hd = HardDrive.new
hd.capacity = 150
hd.external = true
hd.speed = 7200
```

And if you are a real fan of the Kestrel, you would design your class with an object initializer block so you could write:

```
hd = HardDrive.new do
        @capacity = 150
        @external = true
        @speed = 7200
end
```

But what do you do when handed a class that was not designed with method chaining in mind? For example, `Array#pop` returns the object being popped, not the array. Before you validate every criticism levelled against Ruby for allowing programmers to rewrite methods in core classes, consider using `#tap` with `Symbol#to_proc` or `String#to_proc` to chain methods without rewriting them.

So instead of

```
def fizz(arr)
        arr.pop
        arr.map! { |n| n * 2 }
end
```

We can write:

```
def fizz(arr)
  arr.tap(&:pop).map! { |n| n * 2 }
end
```

I often use `#tap` to enchain methods for those pesky array methods that sometimes do what you expect and sometimes don't. My most hated example is `Array#uniq!`[2]:

```
arr = [1,2,3,3,4,5]
arr.uniq, arr
        => [1,2,3,4,5], [1,2,3,3,4,5]
arr = [1,2,3,3,4,5]
arr.uniq!, arr
        => [1,2,3,4,5], [1,2,3,4,5]
arr = [1,2,3,4,5]
arr.uniq, arr
        => [1,2,3,4,5], [1,2,3,4,5]
arr = [1,2,3,4,5]
arr.uniq!, arr
        => nil, [1,2,3,4,5]
```

Let's replay that last one in slow motion:

```
[  1,  2,  3,  4,  5  ].uniq!
        => nil
```

That might be a problem. For example:

```
[1,2,3,4,5].uniq!.sort!
        => NoMethodError: undefined method `sort!' for nil:NilClass
```

`Object#tap` to the rescue: When using a method like `#uniq!` that modifies the array in place and sometimes returns the modified array but sometimes helpfully returns `nil`, I can use `#tap` to make sure I always get the array, which allows me to enchain methods:

```
[1,2,3,4,5].tap(&:uniq!).sort!
        => [1,2,3,4,5]
```

So there's another use for `#tap` (along with `Symbol#to_proc` for simple cases): We can use it when we want to enchain methods, but the methods do not return the receiver.

[2] http://ruby-doc.org/core/classes/Array.html#M002238

> In Ruby 1.9, `#tap` works exactly as described above. Ruby 1.8 does not have `#tap`, but you can obtain it by installing the andand gem. This version of `#tap` also works like a Quirky Bird, so you can write things like `HardDrive.new.tap.capacity(150)` for enchaining methods that take parameters and/or blocks. To get andand, `sudo gem install andand`. Rails users can also drop `andand.rb` in `config/initializers`.

## 2.4 The Obdurate Kestrel

The andand gem[3] includes `Object#tap` for Ruby 1.8. It also includes another kestrel called `#dont`. Which does what it says, or rather *doesn't* do what it says.

```
:foo.tap { p 'bar' }
bar
        => :foo # printed 'bar' before returning a value!

:foo.dont { p 'bar' }
        => :foo # without printing 'bar'!
```

`Object#dont` simply ignores the block passed to it. So what is it good for? Well, remember our logging example for `#tap`?

```
address = Person.find(...).tap { |p| logger.log "person #{p} found" }.address
```

Let's turn the logging off for a moment:

```
address = Person.find(...).dont { |p| logger.log "person #{p} found" }.address
```

And back on:

```
address = Person.find(...).tap { |p| logger.log "person #{p} found" }.address
```

I typically use it when doing certain kinds of primitive debugging. And it has another trick up its sleeve:

```
arr.dont.sort!
```

Look at that, it works with method calls like a quirky bird! So you can use it to `NOOP` methods. Now, you could have done that with `Symbol#to_proc`:

[3] http://github.com/raganwald/andand/tree

```
arr.dont(&:sort!)
```

But what about methods that take parameters and blocks?

```
JoinBetweenTwoModels.dont.create!(...) do |new_join|
        # ...
end
```

`Object#dont` is the Ruby-semantic equivalent of commenting out a method call, only it can be inserted inside of an existing expression. That's why it's called the *obdurate kestrel*. It refuses to do anything!

If you want to try `Object#dont`, or want to use `Object#tap` with Ruby 1.8, `sudo gem install andand`. Rails users can also drop `andand.rb` in `config/initializers` as mentioned above. Enjoy!

## 2.5 Kestrels on Rails

As mentioned, Ruby on Rails provides #returning, a method with K Combinator semantics:

```
returning(expression) do |name|
  # name is bound to the result of evaluating expression
  # this block is evaluated and the result is discarded
end
  => # the result of evaluating the expression is now returned
```

Rails also provides *object initializer blocks* for ActiveRecord models. Here's an example from one of my unit tests:

```
@board = Board.create(:dimension => 9) do |b|
  b['aa'] = 'black'
  b['bb'] = 'black'
  b['cb'] = 'black'
  b['da'] = 'black'
  b['ba'] = 'white'
  b['ca'] = 'white'
end
```

So, it looks like in Rails you can choose between an object initializer block and #returning:

```
@board = returning(Board.create(:dimension => 9)) do |b|
  b['aa'] = 'black'
  b['bb'] = 'black'
  b['cb'] = 'black'
  b['da'] = 'black'
  b['ba'] = 'white'
  b['ca'] = 'white'
end
```

In both cases the created object is returned regardless of what the block would otherwise return. But beyond that, the two Kestrels have very different semantics. "Returning" fully evaluates the expression, in this case creating the model instance in its entirety, including all of its callbacks. The object initializer block, on the other hand, is called as part of initializing the object *before* starting the lifecycle of the object including its callbacks.

"Returning" is what you want when you want to do stuff involving the fully created object and you are trying to logically group the other statements with the creation. In my case, that's what I want, I am trying to say that @board is a board with black stones on certain intersections and white stones on other intersections.

Object initialization is what you want when you want to initialize certain fields by hand and perform some calculations or logic before kicking off the object creation lifecycle. That wasn't what I wanted in this case because my `[]=` method depended on the object being initialized. So my code had a bug that was fixed when I changed from object initializers to #returning.

Summary: In Rails, object initializers are evaluated before the object's life cycle is started, #returning's block is evaluated afterwards. And that is today's *lingua obscura*.

## 2.6 Rewriting "Returning" in Rails

One of the most useful tools provided by Ruby on Rails is the #returning method, a simple but very useful implementation of the K Combinator or Kestrel. For example, this:

```
def registered_person(params = {})
  person = Person.new(params.merge(:registered => true))
  Registry.register(person)
  person.send_email_notification
  person
end
```

Can and should be expressed using #returning as this:

```
def registered_person(params = {})
  returning Person.new(params.merge(:registered => true)) do |person|
    Registry.register(person)
    person.send_email_notification
  end
end
```

Why? Firstly, you avoid the common bug of forgetting to return the object you are creating:

```
def broken_registered_person(params = {})
  person = Person.new(params.merge(:registered => true))
  Registry.register(person)
  person.send_email_notification
end
```

This creates the person object and does the initialization you want, but doesn't actually return it from the method, it returns whatever #send_email_notification happens to return. If you've worked hard to create fluent interfaces you might be correct by accident, but #send_email_notification could just as easily return the email it creates. Who knows?

Second, in methods like this as you read from top to bottom you are declaring what the method returns right up front:

```
def registered_person(params = {})
  returning Person.new(params.merge(:registered => true)) do # ...
    # ...
  end
end
```

It takes some optional params and returns a new person. Very clear. And the third reason I like #returning is that it logically clusters the related statements together:

```
returning Person.new(params.merge(:registered => true)) do |person|
  Registry.register(person)
  person.send_email_notification
end
```

It is very clear that these statements are all part of one logical block. As a bonus, my IDE respects that and it's easy to fold them or drag them around as a single unit. All in all, I think #returning is a big win and I even look for opportunities to refactor existing code to use it whenever I'm making changes.

### DWIM

All that being said, I have observed a certain bug or misapplication of #returning from time to time. It's usually pretty subtle in production code, but I'll make it obvious with a trivial example. What does this snippet evaluate to?

```
returning [1] do |numbers|
  numbers << 2
  numbers += [3]
end
```

This is the kind of thing that sadistic interviewers use in coding quizzes. The answer is **[1, 2]**, not [1, 2, 3]. The << operator mutates the value assigned to the numbers variable, but the += statement overwrites the reference assigned to the numbers variable without changing the original value. #returning remembers the *value* originally assigned to numbers and returns it. If you have some side-effects on that value, those count. But assignment does nothing to the value.

This may seem obvious, but in my experience it is a subtle point that causes difficulty. Languages with referential transparency escape the confusion entirely, but OO languages like Ruby have this weird thing where we have to keep track of references and labels on references in our head.

Here's something contrived to look a lot more like production code. First, without #returning:

```
def working_registered_person(params = {})
  person = Person.new(params.merge(:registered => true))
  if Registry.register(person)
    person.send_email_notification
  else
    person = Person.new(:default => true)
  end
  person
end
```

And here we've refactored it to use #returning:

```
def broken_registered_person(params = {})
  returning Person.new(params.merge(:registered => true)) do |person|
    if Registry.register(person)
      person.send_email_notification
    else
      person = Person.new(:default => true)
    end
  end
end
```

Oops! This no longer works as we intended. Overwriting the `person` variable is irrelevant, #returning returns the unregistered new person no matter what. So what's going on here?

One answer is to "blame the victim." Ruby has a certain well-documented behaviour around variables and references. #returning has a certain well-documented behaviour. Any programmer who makes the above mistake is–well–mistaken. Fix the code and set the bug ticket status to Problem Between Keyboard And Chair ("PBKAC").

Another answer is to suggest that the implementation of #returning is at fault. If you write:

```
returning ... do |var|
  # ...
  var = something_else
  # ...
end
```

You intended to change what you are returning from #returning. So #returning should be changed to do what you meant. I'm on the fence about this. When folks argue that designs should cater to programmers who do not understand the ramifactions of the programming language or of the framework, I usually retort that you cannot have progress and innovation while clinging to familiarity, an argument I first heard from Jef Raskin[4]. The real meaning of "The Principle of Least Surprise" is that a design should be *internally consistent*, which is not the same thing as *familiar*.

Ruby's existing use of variables and references is certainly consistent. And once you know what #returning does, it remains consistent. However, this design decision isn't really about being consistent with Ruby's implementation, we are debating how an idiom should be designed. I think we have a blank canvas and it's reasonable to at least *consider* a version of #returning that handles assignment to the parameter.

### Rewriting #returning

[4] http://weblog.raganwald.com/2008/01/programming-language-cannot-be-better.html

The RewriteRails[5] plug-in adds syntactic abstractions like Andand[6] to Rails projects without monkey-patching[7]. RewriteRails now includes its own version of #returning that overrides the #returning shipping with Rails.

When RewriteRails is processing source code, it turns code like this:

```
def registered_person(params = {})
  returning Person.new(params.merge(:registered => true)) do |person|
    if Registry.register(person)
      person.send_email_notification
    else
      person = Person.new(:default => true)
    end
  end
end
```

Into this:

```
def registered_person(params = {})
  lambda do |person|
    if Registry.register(person)
      person.send_email_notification
    else
      person = Person.new(:default => true)
    end
    person
  end.call(Person.new(params.merge(:registered => true)))
end
```

Note that in addition to turning the #returning "call" into a lambda that is invoked immediately, it also makes sure the new lambda returns the `person` variable's contents. So assignment to the variable does change what #returning appears to return.

Like all processors in RewriteRails, #returning is only rewritten in `.rr` files that you write in your project. Existing `.rb` files are not affected, including all code in the Rails framework: RewriteRails will never monkey with other people's expectations. RewriteRails doesn't physically modify the .rr files you write: The rewritten code is put in another file that the Ruby interpreter sees. So you see the code you write and RewriteRails figures out what to show the interpreter. This is a little like a Lisp macro.

[5] http://github.com/raganwald-deprecated/rewrite_rails/tree/master

[6] http://github.com/raganwald-deprecated/rewrite_rails/tree/master/doc/andand.textile

[7] http://avdi.org/devblog/2008/02/23/why-monkeypatching-is-destroying-ruby/

# 3 The Thrush

In Combinatory Logic, the thrush is an extremely simple *permuting* combinator; it reverses the normal order of evaluation. The thrush is written `Txy = yx`. It *reverses* evaluation. In Ruby terms,

```
thrush.call(a_value).call(a_proc)
  => a_proc.call(a_value)
```

In No Detail Too Small[1], I defined `Object#into`, an implementation of the thrush as a Ruby method:

```
class Object
  def into expr = nil
    expr.nil? ? yield(self) : expr.to_proc.call(self)
  end
end
```

If you are in the habit of violating the Law of Demeter[2], you can use `#into` to make an expression read consistently from left to right. For example, this code:

```
lambda { |x| x * x }.call((1..100).select(&:odd?).inject(&:+))
```

Reads "Square (take the numbers from 1 to 100, select the odd ones, and take the sum of those)." Confusing. Whereas with `#into`, you can write:

```
(1..100).select(&:odd?).inject(&:+).into { |x| x * x }
```

Which reads "Take the numbers from 1 to 100, keep the odd ones, take the sum of those, and then answer the square of that number."

A permuting combinator like `#into` is not strictly necessary when you have parentheses or local variables. Which is kind of interesting, because it shows that if you have permuting combinators, you can model parentheses and local variables.

But we are not interested in theory. `#into` may be equivalent to what we can accomplish with other means, but it is useful to us if we feel it makes the code clearer and easier to understand. Sometimes a longer expression should be broken into multiple small expressions to make it easier to understand. Sometimes it can be reordered using tools like `#into`.

[1] http://weblog.raganwald.com/2008/01/no-detail-too-small.html

[2] http://en.wikipedia.org/wiki/Law_of_Demeter

## 3.1 Let

`Object#into` defines the thrush as a method that takes a block, lambda, or anything that can become a block or lambda as its argument. There is another way to formulate a Thrush:

```
module Kernel
  def let it
    yield it
  end
end
```

It's remarkably simple, so simple that it appears to be less useful than `#into`. The example above would look like this if we used `let`:

```
let (1..100).select(&:odd?).inject(&:+) do |x|
        x * x
end
```

How does that help? I'll let you in on a secret: Ruby 1.9 changes the game. In Ruby 1.8, `x` is local to the surrounding method, so it doesn't help. But in Ruby 1.9, `x` is a *block local variable*, meaning that it does not clobber an existing variable. So in Ruby 1.8:

```
def say_the_square_of_the_sum_of_the_odd_numbers(x)
        sotos = let (1..x).select(&:odd?).inject(&:+) do |x|
                x * x
        end
        "The square of the sum of the odd numbers from 1..#{x} is #{sotos}"
end

say_the_square_of_the_sum_of_the_odd_numbers(10)
 => "The square of the sum of the odd numbers from 1..25 is 625"
```

`1..25`!? What happened here is that the `x` inside the block clobbered the value of the `x` parameter. Not good. In Ruby 1.9:

```
say_the_square_of_the_sum_of_the_odd_numbers(10)
 => "The square of the sum of the odd numbers from 1..10 is 625"
```

Much better, Ruby 1.9 creates a new scope inside the block and `x` is local to that block, *shadowing* the `x` parameter. Now we see a use for `let`:

```
let(some_expression) do |my_block_local_variable|
        # ...
end
```

`let` creates a new scope and defines your block local variable inside the block. This signals[3] that the block local variable is not used elsewhere. Imperative methods can be easier to understand when they are composed of smaller blocks with well-defined dependencies between them. A variable local to the entire method creates a dependency across the entire method. A variable local to a block only creates dependencies within that block.

Although Ruby 1.8 does not enforce this behaviour, it can be useful to write code in this style as a signal to make the code easier to read.

**Summary**

We have seen two formulations of the thrush combinator, `#into` and `let`. One is useful for making expressions more consistent and easier to read, the other for signaling the scope of block-local variables.

[3] http://weblog.raganwald.com/2007/11/programming-conventions-as-signals.html

# 4 Songs of the Cardinal

In Combinatory Logic, the cardinal is one of the most basic *permuting* combinators; it reverses and parenthesizes the normal order of evaluation. The cardinal is written `Cxyz = xzy`. In Ruby:

```
cardinal.call(proc_over_proc).call(a_value).call(a_proc)
  => proc_over_proc.call(a_proc).call(a_value)
```

What does this mean? Let's compare it to the Thrush. The thrush is written `Txy = yx`. In Ruby terms,

```
thrush.call(a_value).call(a_proc)
  => a_proc.call(a_value)
```

The salient difference is that a cardinal doesn't just pass `a_value` to `a_proc`. What it does is first passes `a_proc` to `proc_over_proc` and then passes `a_value` to the result. This implies that `proc_over_proc` is a function that takes a function as its argument and returns a function.

Or in plainer terms, you want a cardinal when you would like to modify what a function or a block does. Now you can see why we can derive a thrush from a cardinal. If we write:

```
identity = lambda { |f| f }
```

Then we can write:

```
thrush = cardinal.call(identity)
```

What we have done is say a thrush is what you get when you use a cardinal and a function that doesn't modify its function but answers it right back.

> *Note to ornithologists and ontologists*:
>
> This is not object orientation: a thrush is not a kind of cardinal. The correct relationship between them in Ruby is that a cardinal creates a thrush. Or in Smullyan's songbird metaphor, if you call out the name of an identity bird to a cardinal, it will call out the name of a thrush back to you.

Now, this bizarre syntactic convention of writing `foo.call(bar).call(bash)` is not very helpful for actually writing software. It is great for explaining what's going on, but if we are going to use Ruby for the examples, we need to lift our game up a level and make some idiomatic Ruby.

## 4.1 Building a Cardinal in Ruby

The next chunk of code works around the fact that Ruby 1.8 can't define a proc that takes a block and also doesn't allow `define_method` to define a method that takes a block. So for Ruby 1.8, we will start by making a utility method that defines methods that can take a block, based on an idea from coderr[1]. For Ruby 1.9 this is not necessary: you can use `define_method` to define methods that take blocks as arguments.

```
def define_method_taking_block(name, method_body_proc)
  self.class.send :define_method, "__cardinal_helper_#{name}__", &method_body_proc
  eval <<-EOM
    def #{name}(a_value, &a_proc)
      __cardinal_helper_#{name}__(a_value, a_proc)
    end
  EOM
end
```

> Now we can see what the expression "accidental complexity" means. Do you see how we need a long paragraph and a chunk of code to explain how we are working around a limitation in our tool? And how the digression to explain the workaround is longer than the actual code we want to write? Ugh!

With *that* out of the way, we can write our cardinal:

```
def cardinal_define(name, &proc_over_proc)
  define_method_taking_block(name) do |a_value, a_proc|
      proc_over_proc.call(a_proc).call(a_value)
  end
end
```

Ready to try it? Here's a familiar example. We'll need a `proc_over_proc`, our proc that modifies another proc. Because we're trying to be Ruby-ish, we'll write it out as a block:

```
do |a_proc|
```

lambda { |a_value|

[1] http://coderrr.wordpress.com/2008/10/29/using-define_method-with-blocks-in-ruby-18/

```
    a_proc.call(a_value) unless a_value.nil?
  }
end
```

This takes a `a_proc` and returns a brand new proc that only calls `a_proc` if the value you pass it is not nil. Now let's use our cardinal to define a new method:

```
cardinal_define(:maybe) do |a_proc|
  lambda { |a_value|
    a_proc.call(a_value) unless a_value.nil?
  }
end
```

Let's try it out:

```
maybe(1) { |x| x + 1 }
  => 2
maybe(nil) { |x| x + 1 }
  => nil
```

If we're using Rails, we can make a slightly different version of `maybe`:

```
cardinal_define(:unless_blank) do |a_proc|
```

lambda { |a_value|

```
    a_proc.call(a_value) unless a_value.blank?
  }
end

unless_blank(Person.find(...).name) do |name|
  register_name_on_title(name)
end
```

Remember we said the cardinal can be used to define a thrush? Let's try our Ruby cardinal out to do the same thing. Recall that expressing the identity bird as a block is:

```
do |a_proc|
  a_proc
end
```

Therefore we can define a thrush with:

```
cardinal_define(:let) do |a_proc|
  a_proc
end

let((1..10).select { |n| n % 2 == 1 }.inject { |mem, var| mem + var }) do |x|
  x * x
end
  => 625
```

As you can see, once you have a defined a cardinal, you can create an infinite variety of methods that have thrush-like syntax–a method that applies a value to a block–but you can modify or augment the *semantics* of the block in any way you want.

In Ruby terms, you are meta-programming. In Smullyan's terms, you are *Listening to the Songs of the Cardinal.*

# 5 Quirky Birds and Meta-Syntactic Programming

In Combinatory Logic, the Queer Birds are a family of combinators which both parenthesize and permute. One member of the family, the *Quirky Bird*, has interesting implications for Ruby. The quirky bird is written `Qxyz = z(xy)`. In Ruby:

```
quirky.call(value_proc).call(a_value).call(a_proc)
  => a_proc.call(value_proc.call(a_value))
```

Like the Cardinal, the quirky bird reverses the order of application. But where the cardinal modifies the function that is applied to a value, the quirky bird modifies the value itself. Let's compare how cardinals and quirky birds work.

**a cardinals refresher**

The cardinal is defined in its simplest Ruby form as:

```
cardinal.call(proc_over_proc).call(a_value).call(a_proc)
  => proc_over_proc.call(a_proc).call(a_value)
```

From that definition, we wrote a method called `cardinal_define` that writes methods in idiomatic Ruby. For example, here's how we used `cardinal_define` to generate the `maybe` method:

```
cardinal_define(:maybe) do |a_proc|
  lambda { |a_value|
    a_proc.call(a_value) unless a_value.nil?
  }
end

maybe(1) { |x| x + 1 }
  => 2
maybe(nil) { |x| x + 1 }
  => nil
```

Now we are not looking at the source code for `maybe`, but from the definition of a cardinal above we know that any method defined by `cardinal_define` will look roughly like:

```
def defined_by_a_cardinal(a_value, &a_proc)
        proc_over_proc.call(a_proc).call(a_value)
end
```

Or in our case:

```
def maybe(a_value, &a_proc)
        lambda do |a_proc|
          lambda { |a_value|
            a_proc.call(a_value) unless a_value.nil?
          }
        end.call(a_proc).call(a_value)
end
```

### and now to the quirky bird

From the definition for the quirky bird, we expect that if we write `quirky_bird_define`, the methods it generates will look roughly like:

```
def defined_by_a_quirky_bird(a_value, &a_proc)
        a_proc.call(value_proc.call(a_value))
end
```

So, are we ready to write `quirky_bird_define`? This seems too easy. Just copy the `cardinal_define` code, make a few changes, and we're done:

```
def quirky_bird_define(name, &value_proc)
  define_method_taking_block(name) do |a_value, a_proc|
    a_proc.call(value_proc.call(a_value))
  end
end

# method_body_proc should expect (a_value, a_proc)
# see http://coderrr.wordpress.com/2008/10/29/using-define_method-with-blocks-in-\
ruby-18/
def define_method_taking_block(name, &method_body_proc)
  self.class.send :define_method, "__quirky_bird_helper_#{name}__", method_body_p\
roc
  eval <<-EOM
    def #{name}(a_value, &a_proc)
      __quirky_bird_helper_#{name}__(a_value, a_proc)
    end
  EOM
end
```

Ok, let's try it out on something really trivial:

```
quirky_bird_define(:square_first) do |a_value|
        a value * a_value
end

square_first(1) { |n| n + 1 }
        => 2

square_first(2) { |n| n + 1 }
        => 5
```

It works, good. Now let's define `maybe` using the quirky bird we just wrote. Just so we're clear, I want to write:

```
quirky_bird_define(:maybe) do |a_value|
        # ... something goes here ...
end

maybe(1) { |n| n + 1 }
        => 2

maybe(nil) { |n| n + 1 }
        => nil
```

Scheisse! Figuring out what to put in the block to make `maybe` work is indeed queer and quirky!!

Now, the simple truth is, I know of no way to use a quirky bird to cover all of the possible blocks you could use with `maybe` so that it works exactly like the version of `maybe` we built with a cardinal. However, I have found that sometimes it is interesting to push an incomplete idea along if it is incomplete in interesting ways. "Maybe" we can learn something in the process.

## 5.1 A limited interpretation of the Quirky Bird in Ruby

Let's solve `maybe` any-which-way-we-can and see how it goes. When we used a cardinal, we wanted a proc that would modify another proc to such that if it was passed `nil`, it would answer `nil` without evaluating its contents.

Now we want to modify a value such that if it is `nil`, it responds `nil` to the method `+`. This is doable, with the help of the `BlankSlate` class, also called a `BasicObject`. You'll find `BlankSlate` and `BasicObject` classes in various frameworks and Ruby 1.9, and there's one at blank_slate.rb[1] you can use.

[1] http://github.com/raganwald/homoiconic/tree/master/2008-11-04/blank_slate.rb

`BlankSlate` is a class with no methods, which is very different from the base class `Object`. That's because `Object` in Ruby is *heavyweight*, it has lots of useful stuff. But we don't want useful stuff, because our mission is to answer a value that responds `nil` to any method you send it.

The Ruby way to handle any method is with `method_missing`. Here's a really simple expression that answers an object that responds `nil` to any method:

```
returning(BlankSlate.new) do |it|
  def it.method_missing(*args)
    nil
  end
end
```

Hmmm. What about:

```
quirky_bird_define(:maybe) do |value|
  if value.nil?
    returning(BlankSlate.new) do |it|
      def it.method_missing(*args)
        nil
      end
    end
  else
    value
  end
end
```

This is saying, "Let's define a quirky bird method based on a `value_proc` as usual. Our `value_proc` will take a value, and if the value is `nil` we will return an object that responds with `nil` to any method. But if the value is not nil, our `value_proc` will respond with the object."

Let's try it:

```
maybe(1) { |n| n + 1 }
        => 2

maybe(nil) { |n| n + 1 }
        => nil
```

Now, I admit this is *very* flawed:

```
maybe(nil) { |n| n + 1 + 1 }
```

⇒ NoMethodError: undefined method '+' for nil:NilClass

```
maybe(nil) { |n| 1 + n }
```

⇒ TypeError: coerce must return [x, y]

The basic problem here is that we only control the value we pass in. We can't modify how other objects respond to it, nor can we control what happens to any objects we return from methods called on it. So, the quirky bird turns out to be useful in the case where (a) the value is the receiver of a method, and (b) there is only one method being called, not a chain of methods.

Hmmm again.

## 5.2 Embracing the Quirky Bird

Maybe we shouldn't be generating methods that deal with arbitrary blocks and procedures. One way to scale this down is to deal only with single method invocations. For example, what if instead of designing our new version of `maybe` so that we invoke it by writing `maybe(nil) { |n| n + 1 }` or `maybe(1) { |n| n + 1 }`, we design it so that we write `nil.maybe + 1` or `1.maybe + 1` instead?

In that case, `maybe` becomes a method on the object class that applies `value_proc` to its receiver rather than being a method that takes a value and a block. Getting down to business, we are going to open the core `Object` class and add a new method to it. The body of that method will be our `value_proc`:

```
def quirky_bird_extend(name, &value_proc)
  Object.send(:define_method, name) do
    value_proc.call(self)
  end
end
```

Just as we said, we are defining a new method in the `Object` class.

> We are using `define_method` and a block rather than the `def` keyword. The reason is that when we use `define_method` and a block, the body of the method executes in the context of the block, not the context of the object itself. Blocks are closures in Ruby, which means that the block has access to `value_proc`, the parameter from our `quirky_bird_extend` method.

> Had we used `def`, Ruby would try to evaluate `value_proc` in the context of the object itself. So our parameter would be lost forever. Performance wonks and compiler junkies will be interested in this behaviour, as it has very serious implications for garbage collection and memory leaks.

Now let's use it with exactly the same block we used with `quirky_bird_define`:

```
require 'quirky_bird'
require 'blank_slate'
require 'returning'

quirky_bird_extend(:maybe) do |value|
  if value.nil?
    returning(BlankSlate.new) do |it|
      def it.method_missing(*args)
        nil
      end
    end
  else
    value
  end
end

nil.maybe + 1
  => nil

1.maybe + 1
  => 2
```

It works. And it looks familiar! We have defined our own version of andand[2], only this is much more **interesting**. Instead of a one-off handy-dandy, we have created a method that creates similar methods.

Let's try it again, this time emulating Chris Wanstrath's try[3]:

[2] http://github.com/raganwald/andand/tree
[3] http://ozmm.org/posts/try.html

```
quirky_bird_extend(:try) do |value|
  returning(BlankSlate.new) do |it|
    def it.__value__=(arg)
       @value = arg
    end
    def it.method_missing(name, *args)
      if @value.respond_to?(name)
        @value.send(name, *args)
      end
    end
    it.__value__ = value
  end
end

nil.try + 1
  => nil

1.try + 1
  => 2

1.try.ordinalize
  => nil
```

As you can see, we can used the quirky bird to create a whole family of methods that modify the receiver in some way to produce new semantics. I can't show you the source code, but here is something from a proprietary Rails application:

```
Account.without_requiring_authorization.create!(...)
```

In this case, `without_requiring_authorization` follows the quirky bird pattern, only instead of taking an instance and producing a version that handles certain methods specially, this one takes a class and produces a version that doesn't enforce authorization for use in test cases.

**so what have we learned?**

The quirky bird is superficially similar to the cardinal, however it can be used to generate syntax that is a little more method-oriented rather than function-oriented. And what's better than a handy method like andand? A method for defining such methods, of course.

## 5.3 Andand even more

As we've discovered, "andand" is a Quirky Bird. Here's a little tip for using it effectively: You already know that you can use it to conditionally invoke a method on an object:

```
"foo".andand + "bar"
    # => "foobar"
nil.andand + "bar"
    # => nil
```

In other words, it's a Quirky Bird. But did you know that you can also use it to conditionally invoke a block?

```
(1..10).andand do |numbers|
  doubles = numbers.map { |n| n * 2 }
  double_doubles = doubles.map { |n| n * 2 }
end
    # => [4, 8, 12, 16, 20, 24, 28, 32, 36, 40]

nil.andand do |numbers|
  doubles = numbers.map { |n| n * 2 }
  double_doubles = doubles.map { |n| n * 2 }
end
    # => nil
```

It's not just a Quirky Bird, it's also a Cardinal!

Consider this conditional code:

```
if my_var = something_or_other()
  3.times do
    yada(my_var)
  end
end
```

I'm not a big fan. The obvious sin is the pathetic 90s cultural reference. But I'm even more annoyed by having side-effects in the predicate of an `if` clause, in this case assigning something to the variable `my_var`. Although I'm not switching to a purely functional language any time soon, I strongly prefer that when you write `if something()`, then "something()" should not cause any side effects, ever.

Another problem is that we are obviously creating `my_var` just to use inside the block, but we're declaring it in top-level scope. We could fool around with a Thrush like `let`, but instead let's use `Object#andand`:

```
something_or_other().andand do |my_var|
  3.times do
    yada(my_var)
  end
end
```

Now we are making it clear that we wish to execute this block only if `something_or_other()` is not nil. Furthermore, we are assigning the result of `something_or_other()` to `my_var` and using it within the block. Crisp and clean, no caffeine[4].

Note that if we don't actually *need* `my_var` in the block, we don't really need `andand` either:

```
something_or_other() and begin
  3.times do
    yada()
  end
end
```

Like anything else, `andand do ... end` is a tool to be used in specialized situations. Use it whenever you want to do something more complicated than a simple message send, but only when the subject is not `nil`.

[4] http://www.youtube.com/watch?v=ryXsn7fLV-M

# 6 Aspect-Oriented Programming in Ruby using Combinator Birds

In Combinatory Logic, the bluebird is one of the most important and fundamental combinators, because the bluebird *composes* two other combinators. Although this is usually discussed as part of functional programming style[1], it is just as valuable when writing object-oriented programs. In this post, we will develop an aspect-oriented programming[2] (or "AOP") module that adds before methods and after methods to Ruby programs, with the implementation inspired by the bluebird. The bluebird is written `Bxyz = x(yz)`. In Ruby, we can express the bluebird like this:

```
bluebird.call(proc1).call(proc2).call(value)
        => proc1.call(proc2.call(value))
```

If this seems a little arcane, consider a simple Ruby expression `(x * 2) + 1`: This expression *composes* multiplication and addition. Composition is so pervasive in programming languages that it becomes part of the syntax, something we take for granted. We don't have to think about it until someone like Oliver Steele writes a library like functional javascript[3] that introduces a `compose` function, then we have to ask what it does.

Before we start using bluebirds, let's be clear about something. We wrote that `bluebird.call(proc1).call(proc2).` is equivalent to `proc1.call(proc2.call(value))`. We want to be very careful that we understand what is special about `proc1.call(proc2.call(value))`. How is it different from `proc1.call(proc2).call(value)`?

The answer is:

```
proc1.call(proc2.call(value))
        => puts value into proc2, then puts the result of that into proc1

proc1.call(proc2).call(value)
        => puts proc2 into proc1, getting a function out, then puts value into the new f\
unction
```

So with a bluebird you can chain functions together in series, while if you didn't have a bluebird all you could do is write functions that transform other functions. Not that there's anything wrong with that, we used that to great effect with cardinals[4] and quirky birds[5].

---

[1] http://weblog.raganwald.com/2007/03/why-why-functional-programming-matters.html

[2] http://en.wikipedia.org/wiki/Aspect-oriented_programming

[3] http://osteele.com/sources/javascript/functional/

[4] http://github.com/raganwald/homoiconic/tree/master/2008-10-31/songs_of_the_cardinal.markdown#readme

[5] http://github.com/raganwald/homoiconic/tree/master/2008-11-04/quirky_birds_and_meta_syntactic_programming.markdown#readme

## 6.1 Giving methods advice

We're not actually going to Greenspun[6] an entire aspect-oriented layer on top of Ruby, but we will add a simple feature, we are going to add *before and after methods*. You already know what a normal method is. A before method simply specifies some behaviour you want executed before the method is called, while an after method specifies some behaviour you want executed after the method is called. In AOP, before and after methods are called "advice."

> There is an unwritten rule that says every Ruby programmer must, at some point, write his or her own AOP implementation –Avdi Grimm

Ruby on Rails programmers are familiar with method advice. If you have ever written any of the following, you were using Rails' built-in aspect-oriented programming support:

```
after_save
validates_each
alias_method_chain
before_filter
```

These and other features of Rails implement method advice, albeit in a very specific way tuned to portions of the Rails framework. We're going to implement method advice in a module that you can use in any of your classes, on any method or methods you choose. We'll start with before methods. Here's the syntax we want:

```
def something(parameter)
        # do stuff...
end

before :something do |parameter|
        # stuff to do BEFORE we do stuff...
end

before :something do |parameter|
        # stuff to do BEFORE stuff to do BEFORE we do stuff...
end
```

As we can see, the before methods get chained together before the method. To keep this nice and clean, we are going to make them work just like composable functions: whatever our before method's block returns will be passed as a parameter up the chain. We also won't fool around with altering the order of before methods, we'll just take them as they come.

This is really simple, we are composing methods. To compare to the bluebird above, we are writing `before`, then the name of a method, then a function. I'll rewrite it like this:

[6] http://en.wikipedia.org/wiki/Greenspun%27s_Tenth_Rule

```
bluebird.call(something).call(stuff_to_do_before_we_do_stuff).call(value)
        => something.call(stuff_to_do_before_we_do_stuff.call(value))
```

Now we can see that this newfangled aspect-oriented programming stuff was figured out nearly a century ago by people like Alonzo Church[7].

Okay, enough history, let's get started. First, we are not going to write any C, so there is no way to actually force the Ruby VM to call our before methods. So instead, we are going to have to rewrite our method. We'll use a trick[8] I found on Jay Fields' blog:

```
module NaiveBeforeMethods

  module ClassMethods

    def before(method_sym, &block)
      old_method = self.instance_method(method_sym)
      if old_method.arity == 0
        define_method(method_sym) do
          block.call
          old_method.bind(self).call
        end
      else
        define_method(method_sym) do |*params|
          old_method.bind(self).call(*block.call(*params))
        end
      end
    end

  end

  def self.included(receiver)
    receiver.extend         ClassMethods
  end

end
```

As you can see, we have a special case for methods with no parameters, and when we have a method with multiple parameters, our before method must answer an array of parameters. And the implementation relies on a "flock of bluebirds:" Our before methods and the underlying base method are composed with each other to define the method that is actually executed at run time.

Using it is very easy:

[7] http://en.wikipedia.org/wiki/Alonzo_Church

[8] http://blog.jayfields.com/2006/12/ruby-alias-method-alternative.html

```
class SuperFoo

  def one_parameter(x)
    x + 1
  end

  def two_parameters(x, y)
    x * y
  end

end

class Foo < SuperFoo

  include NaiveBeforeMethods

  before :one_parameter do |x|
    x * 2
  end

  before :two_parameters do |x, y|
    [x + y, x - y]
  end

end

Foo.new.one_parameter(5)
        => 11

Foo.new.two_parameters(3,1)
        => 8
```

> This could be even more useful if it supported methods with blocks. Adventurous readers may want to combine this code with the tricks in `cardinal.rb` and see if they can build a version of `before` that supports methods that take blocks.

## 6.2 The super keyword, perhaps you've heard of it?

Of course, Ruby provides a means of 'decorating' methods like this by overriding a method and calling `super` within it. So we might have written:

```
class Foo < SuperFoo

  def one_parameter(x)
    super(x * 2)
  end

  def two_parameters(x, y)
    super(x + y, x - y)
  end

end
```

On a trivial example, the two techniques seem equivalent, so why bother with the extra baggage? The answer is that using `super` is a little low level. When you see a method definition in a language like Ruby, you don't know whether you are defining a new method, overriding an existing method with entirely new functionality, or "decorating" a method with before advice. Using advice can be useful when you want to signal exactly what you are trying to accomplish.

Another reason to prefer method advice is when you want to share some functionality:

```
class LoggingFoo < SuperFoo

        def one_parameter(x)
                log_entry
                returning(super) do
                        log_exit
                end
        end

        def two_parameters(x, y)
                log_entry
                returning(super) do
                        log_exit
                end
        end

end
```

This could be written as:

```
class LoggingFoo < SuperFoo

  include NaiveBeforeMethods

  before :one_parameter, :two_parameters do # see below
    log_entry
  end

  after :one_parameter, :two_parameters do
    log_exit
  end

end
```

This cleanly separates the concern of logging from the mechanism of what the methods actually do

> Although this is not the main benefit, method advice also works with methods defined in modules and the current class, not just superclasses. So in some ways it is even more flexible than Ruby's `super` keyword.

## 6.3 The Queer Bird

That looks handy. But we also want an *after method*, a way to compose methods in the other order. Good news, the Queer Bird combinator is exactly what we want. Written `Qxyz = y(xz)`, the Ruby equivalent is:

```
queer_bird.call(something).call(stuff_to_do_after_we_do_stuff).call(value)
        => stuff_to_do_after_we_do_stuff.call(something.call(value))
```

Which is, of course:

```
def something(parameter)
        # do stuff...
end

after :something do |return_value|
        # stuff to do AFTER we do stuff...
end
```

The difference between before and after advice is that after advice is consumes and transforms whatever the method returns, while before advice consumes and transforms the parameters to the method.

We *could* copy, paste and modify our bluebird code for the before methods to create after methods. But before you rush off to implement that, you might want to think about a few interesting "real world" requirements:

1. If you define before and after methods in any order, the final result should be that all of the before methods are run before the main method, then all of the after methods. This is not part of combinatory logic, but it's the standard behaviour people expect from before and after methods.
2. You should be able to apply the same advice to more than one method, for example by writing `after :foo, :bar do ... end`
3. If you declare parameters for before advice, whatever it returns will be used by the next method, just like the example above. If you do not declare parameters for before advice, whatever it returns should be ignored. The same goes for after advice.
4. If you override the main method, the before and after methods should still work.
5. The blocks provided should execute in the receiver's scope, like method bodies.

One implementation meeting these requirements is in the appendix. Embedded in a lot of extra moving parts, the basic pattern of composing methods is still evident:

**naive_before_advice.rb**

```ruby
module NaiveBeforeAdvice

  module ClassMethods

    def before(method_sym, &block)
      old_method = self.instance_method(method_sym)
      if old_method.arity == 0
        define_method(method_sym) do
          block.call
          old_method.bind(self).call
        end
      else
        define_method(method_sym) do |*params|
          old_method.bind(self).call(*block.call(*params))
        end
      end
    end
```

```
  end

  def self.included(receiver)
    receiver.extend         ClassMethods
  end

end
```

That is why we looked at supporting just before methods first. If you are comfortable with the naïve implementation of before advice discussed above, the mechanism is easy to understand. The complete version is considerably more powerful. As mentioned, it supports before and after advice. It also uses `instance_exec` to evaluate the blocks in the receiver's scope, providing access to private methods and instance variables. And it works properly even when you override the method being advised.

# 7 Mockingbirds

In this chapter, we will meet a combinator that duplicates its arguments, and see how to use it to achieve recursion. Such combinators are called *recursive combinators*, and are an important foundation for separating the concrete implementation of an algorithm from its definition.

## 7.1 Duplicative Combinators

Almost all of the combinators we've seen in previous essays about combinators "conserve" their arguments. For example, if you pass `xyz` to a *Bluebird*, you get one `x`, one `y`, and one `z` back, exactly what you passed in. You get `x(yz)` back, so they have been grouped for you. But nothing has been added and nothing has been taken away. Likewise the *Thrush* reverses its arguments, but again it answers back the same number arguments you passed to it. The Kestrel, on the other hand, does not conserve its arguments. It *erases* one. If you pass `xy` to a Kestrel, you only get `x` back. The `y` is erased. Kestrels do not conserve their arguments.

Today we are going to meet another combinator that does not conserve its arguments, the Mockingbird. Where a Kestrel erases one of its arguments, the Mockingbird *duplicates* its argument. In logic notation, `Mx = xx`. Or in Ruby:

```
mockingbird.call(x)
        #=> x.call(x)
```

The Mockingbird is not the only combinator that duplicates one or more of its arguments. Logicians have also found important uses for many other duplicating combinators like the Starling (`Sxyz = xz(yz)`), which is one half of the SK combinator calculus[1], and the Turing Bird (`Uxy = y(xxy)`), which is named after its discoverer[2].

The great benefit of duplicative combinators from a *theoretical* perspective is that combinators that duplicate an argument can be used to introduce recursion without names, scopes, bindings, and other things that clutter things up. Being able to introduce anonymous recursion is very elegant, and there are times when it is useful in its own right[3].

[1] http://en.wikipedia.org/wiki/SKI_combinator_calculus

[2] http://www.alanturing.net/turing_archive/index.html

[3] http://www.eecs.harvard.edu/~cduan/technical/ruby/ycombinator.shtml

## 7.2 Recursive Lambdas in Ruby

Let's write a simple recursive combinator in Ruby from first principles. To start with, let's pick a recursive algorithm to implement: We'll *sum the numbers of a nested list.* In other words, we're going to traverse a tree of numbers and generate the sum of the leaves, a recursive problem.

> This is a trivial problem in Ruby, `[1, [[2,3], [[[4]]]]].flatten.inject(&:+)` will do the trick. Of course, it does so by calling `.flatten`, a built-in method that is itself recursive. However, by picking a really simple example, it's easy to focus on the recursion rather than by the domain-specific parts of our problem. That will make things look a little over-engineered here, but when you're interested in the engineering, that's a good thing.

So what is our algorithm?

1. If we are given a number, return it.
2. If we are given a list, call ourself for each item of the list and sum the numbers that are returned.

In Ruby:

```
sum_of_nested_list = lambda do |arg|
  arg.kind_of?(Numeric) ? arg : arg.map { |item| sum_of_nested_list.call(item) }.\
inject(&:+)
end
```

One reason we don't like this is that it breaks badly if we ever modify the variable `sum_of_nested_list`. Although you may think that's unlikely, it can happen when writing the method combinators you've seen in previous chapters. For example, imagine you wanted to write to the log when calling this function, but only once, you don't want to write to the log when it calls itself.

```
old_sum = sum_of_nested_list
sum_of_nested_list = lambda do |arg|
  puts "sum_of_nested_list(#{arg.inspect})"
  old_sum.call(arg)
end

sum_of_nested_list.call([[[[[6]]]]])

  sum_of_nested_list([[[[[6]]]]])
```

```
  sum_of_nested_list([[[[6]]]])
  sum_of_nested_list([[[6]]])
  sum_of_nested_list([[6]])
  sum_of_nested_list([6])
  sum_of_nested_list(6)
    #=> 6
```

This doesn't work because inside our original `sum_of_nested_list`, we call `sum_of_nested_list` by name. If that gets redefined by a method combinator or anything else, we're calling the new thing and not the old one.

Another reason to eschew having lambdas call themselves by name is that we won't be able to create anonymous recursive lambdas. Although naming things is an important part of writing readable software, being able to make anonymous things like object literals opens up a world where everything is truly first class and can be created on the fly or passed around like parameters. So by figuring out how to have lambdas call themselves without using their names, we're figuring out how to make all kinds of lambdas anonymous and flexible, not just the non-recursive ones.

## 7.3 Recursive Combinatorics

The combinator way around this is to find a way to pass a function to itself as a parameter. If a lambda only ever calls its own parameters, it doesn't depend on anything being bound to a name in its environment. Let's start by rewriting our function to take itself as an argument:

sum_of_nested_list = lambda do |myself, arg|

```
  arg.kind_of?(Numeric) ? arg : arg.map { |item| myself.call(myself, item) }.inje\
ct(&:+)
```

end

One little problem: How are we going to pass our function to itself? Let's start by *currying* it into a function that takes one argument, itself, and returns a function that takes an item:

```
sum_of_nested_list = lambda do |myself|
  lambda do |arg|
    arg.kind_of?(Numeric) ? arg : arg.map { |item| myself.call(myself).call(item)\
 }.inject(&:+)
  end
end
```

Notice that we now have `myself` call itself and have the result call an item. To use it, we have to have it call itself:

sum_of_nested_list.call(sum_of_nested_list).call([1, [[2,3], [[[4]]]]]) #⇒ 10

This works, but is annoying. Writing our function to take itself as an argument and return a function is one thing, we can fix that, but having our function call itself by name defeats the very purpose of the exercise. Let's fix it. First thing we'll do, let's get rid of `myself.call(myself).call(item)`. We'll use a new parameter, `recurse` (it's the *last* parameter in an homage to callback-oriented programming style). We'll pass it `myself.call(myself)`, thus removing `myself.call(myself)` from our inner lambda:

```
sum_of_nested_list = lambda do |myself|
  lambda do |arg|
    lambda do |arg, recurse|
      arg.kind_of?(Numeric) ? arg : arg.map { |item| recurse.call(item) }.inject(\
&:+)
    end.call(arg, myself.call(myself))
  end
end

sum_of_nested_list.call(sum_of_nested_list).call([1, [[2,3], [[[4]]]]])
```

#⇒ 10

Next, we hoist our code out of the middle and make it a parameter. This allows us to get rid of the ‘ sum_of_nested_list.call(sum_of_nested_list)‘ by moving it into our lambda:

```
sum_of_nested_list = lambda do |fn|
  lambda { |x| x.call(x) }.call(
    lambda do |myself|
      lambda do |arg|
        fn.call(arg, myself.call(myself))
      end
    end
  )
end.call(
  lambda do |arg, recurse|
    arg.kind_of?(Numeric) ? arg : arg.map { |item| recurse.call(item) }.inject(&:\
+)
  end
)

sum_of_nested_list.call([1, [[2,3], [[[4]]]]])
```

```
#⇒ 10
```

Lots of code there, but let's check and see that it works as an anonymous lambda:

```
lambda do |fn|
  lambda { |x| x.call(x) }.call(
    lambda do |myself|
      lambda do |arg|
        fn.call(arg, myself.call(myself))
      end
    end
  )
end.call(
  lambda do |arg, recurse|
    arg.kind_of?(Numeric) ? arg : arg.map { |item| recurse.call(item) }.inject(&:\
+)
  end
).call([1, [[2,3], [[[4]]]]])
```

```
#⇒ 10
```

Looking at this final example, we can see it has two cleanly separated parts:

```
# The recursive combinator

lambda do |fn|
  lambda { |x| x.call(x) }.call(
    lambda do |myself|
      lambda do |arg|
        fn.call(arg, myself.call(myself))
      end
    end
  )
end.call(

  # The lambda we wish to make recursive

  lambda do |arg, recurse|
    arg.kind_of?(Numeric) ? arg : arg.map { |item| recurse.call(item) }.inject(&:\
+)
  end

)
```

## 7.4 Recursive Combinators in Idiomatic Ruby

We've now managed to separate the mechanism of recursing (the combinator) from what we want to do while recursing. Let's formalize this and make it idiomatic Ruby. We'll make it a method for creating recursive lambdas and call it with a block instead of a lambda:

```
def lambda_with_recursive_callback
  lambda { |x| x.call(x) }.call(
    lambda do |myself|
      lambda do |arg|
        yield(arg, myself.call(myself))
      end
    end
  )
end

sum_of_nested_list = lambda_with_recursive_callback do |arg, recurse|
  arg.kind_of?(Numeric) ? arg : arg.map { |item| recurse.call(item) }.inject(&:+)
end

sum_of_nested_list.call([1, [[2,3], [[[4]]]]])
```

#⇒ 10

Not bad. But hey, let's DRY things up. Aren't `x.call(x)` and `myself.call(myself)` the same thing?

## 7.5 The Mockingbird

Yes, `x.call(x)` and `myself.call(myself)` *are* the same thing:

```
def mockingbird &x
  x.call(x)
end

def lambda_with_recursive_callback
  mockingbird do |myself|
    lambda do |arg|
      yield(arg, mockingbird(&myself))
    end
  end
end
```

```
sum_of_nested_list = lambda_with_recursive_callback do |arg, recurse|
  arg.kind_of?(Numeric) ? arg : arg.map { |item| recurse.call(item) }.inject(&:+)
end

sum_of_nested_list.call([1, [[2,3], [[[4]]]]])
```

#⇒ 10

But does it blend?

```
lambda_with_recursive_callback { |arg, recurse|
  arg.kind_of?(Numeric) ? arg : arg.map { |item| recurse.call(item) }.inject(&:+)
}.call([1, [[2,3], [[[4]]]]])
```

#⇒ 10

Yes!

And now we have our finished recursive combinator. We are able to create recursive lambdas in Ruby without relying on environment variables, just on parameters passed to blocks or lambdas. Our recursive combinator is built on the simplest and most basic of duplicating combinators, the Mockingbird.

In the next chapter, we'll build more sophisticated (and practical) recursive combinators. And while doing so, we'll take an aggressive approach to separating interfaces from implementations in algorithms.

# 8 Refactoring Methods with Recursive Combinators

In previous chapters, we have met some of Combinatory Logic's most interesting combinators like the Kestrel, Thrush, Cardinal, Quirky Bird, and Bluebird. Today we are going to learn how combinators can help us separate the general form of an algorithm like "divide and conquer" from its specific concrete steps. Consider the method `#sum_squares`: It sums the squares of a tree of numbers, represented as a nested list.

```
def sum_squares(value)
  if value.kind_of?(Enumerable)
    value.map do |sub_value|
      sum_squares(sub_value)
    end.inject() { |x,y| x + y }
  else
    value ** 2
  end
end

p sum_squares([1, 2, 3, [[4,5], 6], [[[7]]]])
        => 140
```

And the method `#rotate`: It rotates a square matrix, provided the length of each side is a power of two:

```
def rotate(square)
  if square.kind_of?(Enumerable) && square.size > 1
          half_sz = square.size / 2
          sub_square = lambda do |row, col|
            square.slice(row, half_sz).map { |a_row| a_row.slice(col, half_sz) }
          end
          upper_left = rotate(sub_square.call(0,0))
          lower_left = rotate(sub_square.call(half_sz,0))
          upper_right = rotate(sub_square.call(0,half_sz))
          lower_right = rotate(sub_square.call(half_sz,half_sz))
          upper_right.zip(lower_right).map { |l,r| l + r } +
                  upper_left.zip(lower_left).map { |l,r| l + r }
```

```
  else
    square
  end
end

p rotate([[1,2,3,4], [5,6,7,8], [9,10,11,12], [13,14,15,16]])
        => [[4, 8, 12, 16], [3, 7, 11, 15], [2, 6, 10, 14], [1, 5, 9, 13]]
```

Our challenge is to refactor them. You could change `sub_square` from a closure to a private method (and in languages like Java, you have to do that in the first place). What else? Is there any common behaviour we can extract from these two methods?

Looking at the two methods, there are no lines of code that are so obviously identical that we could mechanically extract them into a private helper. Automatic refactoring tools fall down given these two methods. And yet, there is a really, really important refactoring that should be performed here.

## 8.1 Divide and Conquer

Both of these methods use the Divide and Conquer[1] strategy.

As described, there are two parts to each divide and conquer algorithm. We'll start with conquer: you need a way to decide if the problem is simple enough to solve in a trivial manner, and a trivial solution. You'll also need a way to divide the problem into sub-problems if it's too complex for the trivial solution, and a way to recombine the pieces back into the solution. The entire process is carried our recursively.

For example, here's how `#rotate` rotated the square. We started with a square matrix of size 4:

```
[
        [  1,  2,  3,  4],
        [  5,  6,  7,  8],
        [  9, 10, 11, 12],
        [ 13, 14, 15, 16]
]
```

That cannot be rotated trivially, so we divided it into four smaller sub-squares:

[1] http://www.cs.berkeley.edu/~vazirani/algorithms/chap2.pdf

```
[           [
       [  1,  2],   [  3,  4],
       [  5,  6]    [  7,  8]
]           ]

[           [
       [  9, 10],   [ 11, 12],
       [ 13, 14]    [ 15, 16]
]           ]
```

Those couldn't be rotated trivially either, so our algorithm divide each of them into four smaller squares again, giving us sixteen squares of one number each. Those are small enough to rotate trivially (they do not change), so the algorithm could stop subdividing.

We said there was a recombination step. For `#rotate`, four sub-squares are recombined into one square by moving them counter-clockwise 90 degrees. The sixteen smallest squares were recombined into four sub-squares like this:

```
[           [
       [  2,  6],   [  4,  8],
       [  1,  5]    [  3,  7]
]           ]

[           [
       [ 10, 14],   [ 12, 16],
       [  9, 13]    [ 11, 15]
]           ]
```

Then those four squares were recombined into the final result like this:

```
[           [
       [  4,  8],   [ 12, 16],
       [  3,  7]    [ 11, 15]
]           ]

[           [
       [  2,  6],   [ 10, 14],
       [  1,  5]    [  9, 13]
]
```

And smooshed (that is the technical term) back together:

```
[
        [  4,  8,  12, 16],
        [  3,  7,  11, 15],
        [  2,  6,  10, 14],
        [  1,  5,   9, 13]
]
```

And Voila! There is your rotated square matrix.

Both rotation and summing the squares of a tree combine the four steps of a divide and conquer strategy:

1. Deciding whether the problem is divisible into smaller pieces or can be solved trivially,
2. A solution fro the trivial case,
3. A way to divide a non-trivial problem up,
4. And a way to piece it back together.

Here are the two methods re-written to highlight the common strategy. First, `#sum_squares_2`:

```
public

def sum_squares_2(value)
  if sum_squares_divisible?(value)
    sum_squares_recombine(
      sum_squares_divide(value).map { |sub_value| sum_squares_2(sub_value) }
    )
  else
    sum_squares_conquer(value)
  end
end

private

def sum_squares_divisible?(value)
  value.kind_of?(Enumerable)
end

def sum_squares_conquer(value)
  value ** 2
end

def sum_squares_divide(value)
```

```
  value
end

def sum_squares_recombine(values)
  values.inject() { |x,y| x + y }
end
```

And `#rotate_2`:

```
public

def rotate_2(value)
  if rotate_divisible?(value)
    rotate_recombine(
      rotate_divide(value).map { |sub_value| rotate_2(sub_value) }
    )
  else
    rotate_conquer(value)
  end
end

private

def rotate_divisible?(value)
  value.kind_of?(Enumerable) && value.size > 1
end

def rotate_conquer(value)
  value
end

def rotate_divide(value)
  half_sz = value.size / 2
  sub_square = lambda do |row, col|
    value.slice(row, half_sz).map { |a_row| a_row.slice(col, half_sz) }
  end
  upper_left = sub_square.call(0,0)
  lower_left = sub_square.call(half_sz,0)
  upper_right = sub_square.call(0,half_sz)
  lower_right = sub_square.call(half_sz,half_sz)
  [upper_left, lower_left, upper_right, lower_right]
end
```

```
def rotate_recombine(values)
  upper_left, lower_left, upper_right, lower_right = values
  upper_right.zip(lower_right).map { |l,r| l + r } +
  upper_left.zip(lower_left).map { |l,r| l + r }
end
```

Now the common code is glaringly obvious. The main challenge in factoring it into a helper is deciding whether you want to represent methods like `#rotate_divide` as lambdas or want to fool around specifying method names as symbols. Let's go with lambdas for the sake of writing a clear example:

```
public

def sum_squares_3(list)
  divide_and_conquer(
    list,
    :divisible? => lambda { |value| value.kind_of?(Enumerable) },
    :conquer    => lambda { |value| value ** 2 },
    :divide     => lambda { |value| value },
    :recombine  => lambda { |list| list.inject() { |x,y| x + y } }
  )
end

def rotate_3(square)
  divide_and_conquer(
    square,
    :divisible? => lambda { |value| value.kind_of?(Enumerable) && value.size > 1 \
},
    :conquer => lambda { |value| value },
    :divide => lambda do |square|
            half_sz = square.size / 2
            sub_square = lambda do |row, col|
              square.slice(row, half_sz).map { |a_row| a_row.slice(col, half_sz) }
            end
            upper_left = sub_square.call(0,0)
            lower_left = sub_square.call(half_sz,0)
            upper_right = sub_square.call(0,half_sz)
            lower_right = sub_square.call(half_sz,half_sz)
            [upper_left, lower_left, upper_right, lower_right]
    end,
    :recombine => lambda do |list|
```

```
            upper_left, lower_left, upper_right, lower_right = list
            upper_right.zip(lower_right).map { |l,r| l + r } +
            upper_left.zip(lower_left).map { |l,r| l + r }
    end
  )
end

private

def divide_and_conquer(value, steps)
  if steps[:divisible?].call(value)
    steps[:recombine].call(
      steps[:divide].call(value).map { |sub_value| divide_and_conquer(sub_value, \
steps) }
    )
  else
    steps[:conquer].call(value)
  end
end
```

Now we have refactored the common algorithm out. Typically, something like divide and conquer is treated as a "pattern," a recipe for writing methods. We have changed it into an *abstraction* by writing a `#divide_and_conquer` method and passing it our own functions which it combines to form the final algorithm. That ought to sound familiar: `#divide_and_conquer` is a *combinator* that creates recursive methods for us.

You can also find recursive combinators in other languages like Joy, Factor, and even Javascript (the recursive combinator presented here as `#divide_and_conquer` is normally called `multirec`). Eugene Lazutkin's article on [Using recursion combinators in JavaScript](http://lazutkin.com/blog/2008/jun/30/using-recursion-combinators-javascript/ "") shows how to use combinators to build divide and conquer algorithms in Javascript with the Dojo libraries. This example uses `binrec`, a recursive combinator for algorithms that always divide their problems in two:

```
var fib0 = function(n){
    return n <= 1 ? 1 :
        arguments.callee.call(this, n - 1) +
            arguments.callee.call(this, n - 2);
};

var fib1 = binrec("<= 1", "1", "[[n - 1], [n - 2]]", "+");
```

## 8.2 The Merge Sort

Let's look at another example, implementing a merge sort[2]. This algorithm has a distinguished pedigree: It was invented by John Von Neumann in 1945.

> Von Neumann was a brilliant and fascinating individual. he is most famous amongst Computer Scientists for formalizing the computer architecture which now bears his name. he also worked on game theory, and it was no game to him: He hoped to use math to advise the United States whether an when to launch a thermonuclear war on the USSR. If you are interested in reading more, Prisoner's Dilemma[3] is a very fine book about both game theory and one of the great minds of modern times.

Conceptually, a merge sort works as follows:

- If the list is of length 0 or 1, then it is already sorted.
- Otherwise:
  1. Divide the unsorted list into two sublists of about half the size.
  2. Sort each sublist recursively by re-applying merge sort.
  3. Merge the two sublists back into one sorted list.

The merge sort part will be old hat given our `#divide_and_conquer` helper:

```
def merge_sort(list)
  divide_and_conquer(
    list,
    :divisible? => lambda { |list| list.length > 1 },
    :conquer    => lambda { |list| list },
    :divide     => lambda do |list|
      half_index = (list.length / 2) - 1
      [ list[0..half_index], list[(half_index + 1)..-1] ]
    end,
    :recombine  => lambda { |pair| merge_two_sorted_lists(pair.first, pair.last) }
  )
end
```

The interesting part is our `#merge_two_sorted_lists` method. Given two sorted lists, our merge algorithm works like this:

[2] http://en.wikipedia.org/wiki/Merge_sort

[3] http://www.amazon.com/gp/product/038541580X?ie=UTF8&tag=raganwald001-20&linkCode=as2&camp=1789&creative=390957&creativeASIN=038541580X

- If either list is of length zero, return the other list.
- Otherwise:
  1. Compare the first item of each list using <=>. Let's call the list which has the "preceding" first item the preceding list and the list which has the "following" first item the following list.
  2. Create a pair of lists consisting of the preceding item and an empty list, and another pair of lists consisting of the remainder of the preceding list and the entire following list.
  3. Merge each pair of lists recursively by applying merge two sorted lists.
  4. Catenate the results together.

As you can tell from the description, this is another divide and conquer algorithm:

```
def merge_two_sorted_lists(*pair)
  divide_and_conquer(
    pair,
    :divisible? => lambda { |pair| !pair.first.empty? && !pair.last.empty? },
    :conquer => lambda do |pair|
      if pair.first.empty? && pair.last.empty?
        []
      elsif pair.first.empty?
        pair.last
      else
        pair.first
      end
    end,
    :divide => lambda do |pair|
      preceding, following = case pair.first.first <=> pair.last.first
        when -1: [pair.first, pair.last]
        when 0:  [pair.first, pair.last]
        when 1:  [pair.last, pair.first]
      end
      [
        [[preceding.first], []],
        [preceding[1..-1], following]
      ]
    end,
    :recombine => lambda { |pair| pair.first + pair.last }
  )
end
```

That's great. Well, that's barely ok, actually. The problem is that when doing our merge sort, when we decide which item is the preceding item (least most, front most, whatever you want to call it), we

already know that it is a trivial item and that it doesn't need any further merging. The only reason we bundle it up in `[[preceding.first], []]` is because our `#divide_and_conquer` method expects to recursively attempt to solve all of the sub-problems we generate.

In this case, `#merge_two_sorted_lists` does not really divide a problem into a list of one or more sub-problems, some of which may or may not be trivially solvable. Instead, it divides a problem into a part of the solution and a single sub-problem which may or may not be trivially solvable. This common strategy also has a name, linear recursion[4].

Let's write another version of `#merge_two_sorted_lists`, but his time instead of using `#divide_and_conquer`, we'll write a linear recursion combinator:

```
def merge_two_sorted_lists(*pair)
  linear_recursion(
    pair,
    :divisible? => lambda { |pair| !pair.first.empty? && !pair.last.empty? },
    :conquer => lambda do |pair|
      if pair.first.empty? && pair.last.empty?
        []
      elsif pair.first.empty?
        pair.last
      else
        pair.first
      end
    end,
    :divide => lambda do |pair|
      preceding, following = case pair.first.first <=> pair.last.first
        when -1: [pair.first, pair.last]
        when 0:  [pair.first, pair.last]
        when 1:  [pair.last, pair.first]
      end
      [ preceding.first, [preceding[1..-1], following] ]
    end,
    :recombine => lambda { |trivial_bit, divisible_bit| [trivial_bit] + divisible\
_bit }
  )
end

def linear_recursion(value, steps)
  if steps[:divisible?].call(value)
    trivial_part, sub_problem = steps[:divide].call(value)
    steps[:recombine].call(
```

[4] http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Recn/Linear/

```
      trivial_part, linear_recursion(sub_problem, steps)
    )
  else
    steps[:conquer].call(value)
  end
end
```

You may think this is even better, and it is.

## 8.3 Separating Declaration from Implementation

Using recursive combinators like `#divide_and_conquer` and `#linear_recursion` are abstraction wins. They make recursive code much easier to read, because you know the general form of the algorithm and don't need to pick through it to discover the individual steps. But there's another benefit we should consider: *Recursive combinators separate declaration from implementation.*

Consider `#linear_recursion` again. This is *not* the fastest possible implementation. There is a long and tedious argument that arises when one programmer argues it should be implemented with iteration for performance, and the other argues it should be implemented with recursion for clarity, and a third programmer who never uses recursion claims the iterative solution is easier to understand…

Imagine a huge code base full of `#linear_recursion` and `#divide_and_conquer` calls. What happens if you decide that each one of these algorithms should be implemented with iteration? Hmmm… How about we modify `#linear_recursion` and `#divide_and_conquer`, and all of the methods that call them switch from recursion to iteration for free?

Or perhaps we decide that we really should take advantage of multiple threads… Do you see where this is going? You can write a new implementation and again, all of the existing methods are upgraded.

Even if you do not plan to change the implementation, let's face a simple fact: when writing a brand new recursive or iterative method, you really have two possible sources of bugs: you may not have declared the solution correctly, and you may not implement it correctly.

Using combinators like `#divide_and_conquer` simplifies things: You only need to get your declaration of the solution correct, the implementation is taken care of for you. This is a tremendous win when writing recursive functions.

For these reasons, I strongly encourage the use of recursion combinators, either those supplied here or ones you write for yourself.

## 8.4 Practical Recursive Combinators

We've seen how recursive combinators like `#divide_and_conquer` and `#linear_recursion` are abstraction wins. They make recursive code much easier to read, because you know the general

form of the algorithm and don't need to pick through it to discover the individual steps.

We also saw that by separating the recursion implementation from the declaration of how to perform the steps of an algorithm like `#rotate`, we leave ourselves the opportunity to improve the performance of our implementation without the risk of adding bugs to our declaration. And today we're going to do just that, along with a few tweaks for usability.

In this section, we're going to optimize our combinators' performance and make them a little easier to use with goodies like `string_to_proc`. To do that, we're going to work with closures, defining methods with `define_method`, and implement functional programming's partial application. We'll wrap up by converting `linrec` from a recursive to an iterative implementation.

First, a little organization. Here are the original examples. I've placed them in a module and named the combinators `multirec` and `linrec` in conformance with common practice:

```
module RecursiveCombinators

  def multirec(value, steps)
    if steps[:divisible?].call(value)
      steps[:recombine].call(
        steps[:divide].call(value).map { |sub_value| multirec(sub_value, steps) }
      )
    else
      steps[:conquer].call(value)
    end
  end

  def linrec(value, steps)
    if steps[:divisible?].call(value)
      trivial_part, sub_problem = steps[:divide].call(value)
      steps[:recombine].call(
        trivial_part, linrec(sub_problem, steps)
      )
    else
      steps[:conquer].call(value)
    end
  end

  module_function :multirec, :linrec

end
```

Since they are also module functions, call them by sending a message to the module:

```
def merge_sort(list)
  RecursiveCombinators.multirec(
    list,
    :divisible? => lambda { |list| list.length > 1 },
    :conquer    => lambda { |list| list },
    :divide     => lambda do |list|
      half_index = (list.length / 2) - 1
      [ list[0..half_index], list[(half_index + 1)..-1] ]
    end,
    :recombine  => lambda { |pair| merge_two_sorted_lists(pair.first, pair.last) }
  )
end
```

Or you can include the `RecursiveCombinators` module and call either method directly:

```
include RecursiveCombinators

def merge_two_sorted_lists(*pair)
  linrec(
    pair,
    :divisible? => lambda { |pair| !pair.first.empty? && !pair.last.empty? },
    :conquer => lambda do |pair|
      if pair.first.empty? && pair.last.empty?
        []
      elsif pair.first.empty?
        pair.last
      else
        pair.first
      end
    end,
    :divide => lambda do |pair|
      preceding, following = case pair.first.first <=> pair.last.first
        when -1: [pair.first, pair.last]
        when 0:  [pair.first, pair.last]
        when 1:  [pair.last, pair.first]
      end
      [ preceding.first, [preceding[1..-1], following] ]
    end,
    :recombine => lambda { |trivial_bit, divisible_bit| [trivial_bit] + divisible\
_bit }
  )
end
```

```
merge_sort([8, 3, 10, 1, 9, 5, 7, 4, 6, 2])
        => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Ok, we're ready for some slightly more substantial work. These methods were fine for illustration, but I have a few questions for the author(!)

## 8.5 Spicing things up

First, note that every single time we call a method like `merge_sort`, we create four new lambdas from scratch. This seems wasteful, especially since the lambdas never change. Why create some objects just to throw them away?

On the other hand, it's nice to be able to use create algorithms without having to define a method by name. Although I probably wouldn't do a merge sort anonymously, when I need a one-off quickie, I might like to write something like:

```
RecursiveCombinators.multirec(
  [1, 2, 3, [[4,5], 6], [[[7]]]],
  :divisible? => lambda { |value| value.kind_of?(Enumerable) },
  :conquer    => lambda { |value| value ** 2 },
  :divide     => lambda { |value| value },
  :recombine  => lambda { |list| list.inject() { |x,y| x + y } }
)
        => 140
```

But when I want a permanent sum of the squares method, I **don't** want to write:

```
def sum_squares(list)
  RecursiveCombinators.multirec(
    list,
    :divisible? => lambda { |value| value.kind_of?(Enumerable) },
    :conquer    => lambda { |value| value ** 2 },
    :divide     => lambda { |value| value },
    :recombine  => lambda { |list| list.inject() { |x,y| x + y } }
  )
end
```

...because that would create four lambdas every time I call the function. There are a couple of ways around this problem. First, our "recipe" for summing squares is a simple hash. We could extract that from the method into a constant:

```
SUM_SQUARES_RECIPE = {
   :divisible? => lambda { |value| value.kind_of?(Enumerable) },
   :conquer    => lambda { |value| value ** 2 },
   :divide     => lambda { |value| value },
   :recombine  => lambda { |list| list.inject() { |x,y| x + y } }
}

def sum_squares(list)
  RecursiveCombinators.multirec(list, SUM_SQUARES_RECIPE)
end
```

That (and the isomorphic solution where the constant `SUM_SQUARES_RECIPE` is instead a private helper method `#sum_squares_recipe`) is nice if you have some reason you wish to re-use the recipe elsewhere. But we don't, so this merely clutters our class up and separates the method definition from its logic.

I have something in mind. To see what it is, let's start by transforming our method definition from using the `def` keyword to using the `define_method` private class method. This obviously needs a module or class to work:

```
class Practicum

  include RecursiveCombinators

  define_method :sum_squares do |list|
    multirec(
      list,
           :divisible? => lambda { |value| value.kind_of?(Enumerable) },
           :conquer    => lambda { |value| value ** 2 },
           :divide     => lambda { |value| value },
           :recombine  => lambda { |list| list.inject() { |x,y| x + y } }
         )
  end

end

Practicum.new.sum_squares([1, 2, 3, [[4,5], 6], [[[7]]]])
```

As you probably know, any method taking a block can take a lambda using the `&` operator, so:

```
define_method :sum_squares, &(lambda do |list|
  multirec(
    list,
  :divisible? => lambda { |value| value.kind_of?(Enumerable) },
  :conquer    => lambda { |value| value ** 2 },
  :divide     => lambda { |value| value },
  :recombine  => lambda { |list| list.inject() { |x,y| x + y } }
 )
end)
```

This is useful, because now we can express what we want: a lambda taking one argument that in turn calls `multirec` with the other arguments filled in. Functional programmers call this Partial Application[5]. The idea is that if you have a function or method taking two arguments, if you only give it one argument you get a function back that takes the other. So:

```
multirec(x).call(y)
        => multirec(x,y)
```

Now the drawback with this "standard" implementation of partial application is that we would pass a list to `multirec` and get back a function taking a hash of declarations. That isn't what we want. We could partially apply things *backwards* so that `multirec(x).call(y) => multirec(y,x)` (if Ruby was a concatenative language, we would be concatenating the multirec combinator with a thrush). The trouble with that is it is the reverse of how partial application works in every other programming language[6] and functional programming library[7].

Instead, we will switch the arguments to `multirec` ourselves, so it now works like this:

```
multirec(
        {
                :divisible? => lambda { |value| value.kind_of?(Enumerable) },
                :conquer    => lambda { |value| value ** 2 },
                :divide     => lambda { |value| value },
                :recombine  => lambda { |list| list.inject() { |x,y| x + y } }
        },
        list
)
```

The drawback with this approach is that we lose a little of Ruby's syntactic sugar, the ability to fake named parameters by passing hash arguments without `{}` if they are the last parameter. And

[5] http://ejohn.org/blog/partial-functions-in-javascript/
[6] http://www.haskell.org/
[7] https://github.com/osteele/functional-javascript/tree

now, let's give it the ability to partially apply itself. You can do some stuff with allowing multiple arguments and counting the number of arguments, but we're going to make the wild assumption that you never attempt a recursive combinator on `nil`. Here's `multirec`, you can infer the implementation for `linrec`:

```
def multirec(steps, optional_value = nil)
  worker_proc = lambda do |value|
    if steps[:divisible?].call(value)
      steps[:recombine].call(
        steps[:divide].call(value).map { |sub_value| worker_proc.call(sub_value) }
      )
    else
      steps[:conquer].call(value)
    end
  end
  if optional_value.nil?
    worker_proc
  else
    worker_proc.call(optional_value)
  end
end
```

Notice that you get the same correct result whether you write:

```
RecursiveCombinators.multirec(
  :divisible? => lambda { |value| value.kind_of?(Enumerable) },
  :conquer    => lambda { |value| value ** 2 },
  :divide     => lambda { |value| value },
  :recombine  => lambda { |list| list.inject() { |x,y| x + y } }
).call([1, 2, 3, [[4,5], 6], [[[7]]]])
        => 140
```

Or:

```
RecursiveCombinators.multirec(
        {
           :divisible? => lambda { |value| value.kind_of?(Enumerable) },
           :conquer    => lambda { |value| value ** 2 },
           :divide     => lambda { |value| value },
           :recombine  => lambda { |list| list.inject() { |x,y| x + y } }
        },
        [1, 2, 3, [[4,5], 6], [[[7]]]]
)
        => 140
```

Let's go back to what we were trying to do with `&`:

```
define_method :sum_squares, &(lambda do |list|
  multirec(
    list,
  :divisible? => lambda { |value| value.kind_of?(Enumerable) },
  :conquer    => lambda { |value| value ** 2 },
  :divide     => lambda { |value| value },
  :recombine  => lambda { |list| list.inject() { |x,y| x + y } }
 )
end)
```

Now we know how to build our lambda:

```
require 'partial_application_recursive_combinators'

class Practicum

  extend PartialApplicationRecursiveCombinators   # so we can call multirec in cl\
ass scope

  define_method :sum_squares, &multirec(
   :divisible? => lambda { |value| value.kind_of?(Enumerable) },
   :conquer    => lambda { |value| value ** 2 },
   :divide     => lambda { |value| value },
   :recombine  => lambda { |list| list.inject() { |x,y| x + y } }
  )

end

Practicum.new.sum_squares([1, 2, 3, [[4,5], 6], [[[7]]]])
        => 140
```

You can verify for yourself that no matter how many times you call `sum_squares`, you do not build those lambdas again. What we have just done is added partial application to `multirec` and `linrec`, which in turn allows us to ensure that he cost of constructing lambdas for our methods is only done when the method is defined, not every time it is called.

## 8.6 Building on a legacy

We have already renamed `divide_and_conquer` and `linear_recursion` to bring them into line with standard practice and other programming languages. Now it's time for us to bring the parameters–the declarative lambdas–into line with standard practice.

The four arguments to both methods are normally called `cond`, `then`, `before`, and `after`:

- `cond` is the logical inverse of `divisible?` So if `cond(value)` evaluates to true, then we do not need to subdivide the problem.
- `then` is exactly the same as `conquer`, *if* cond *then* then. That's the way I think of it.
- `before` is the same as `divide`.
- `after` is the same as `recombine`.

Things look very similar with the new scheme for now:

```
require 'legacy_recursive_combinators'

class Practicum

  extend LegacyRecursiveCombinators   # so we can call multirec in class scope

  define_method :sum_squares, &multirec(
    :cond   => lambda { |value| value.kind_of?(Numeric) }, # the only change right\
 now
    :then   => lambda { |value| value ** 2 },
    :before => lambda { |value| value },
    :after  => lambda { |list| list.inject() { |x,y| x + y } }
  )

end
```

All right, now our combinators will look familiar to functional programmers, and even better when we look at functional programs using recursive combinators we will understand them at a glance. Okay, let's get serious and work on making our combinators easy to use and our code easy to read.

## 8.7 Seriously

As long as you're writing these lambdas out, writing `:cond => ` isn't a hardship. And in an explanatory article like this, it can help at first. However, what if you find a way to abbreviate things? For example, you might alias `lambda` to `L`[8]. Or you might want to use `string\_to\_proc`:

**string_to_proc.rb**

```
class String
  unless ''.respond_to?(:to_proc)
    def to_proc &block
      params = []
      expr = self
      sections = expr.split(/\s*->\s*/m)
      if sections.length > 1 then
          eval sections.reverse!.inject { |e, p| "(Proc.new { |#{p.split(/\s/).jo\
in(', ')}| #{e} })" }, block && block.binding
      elsif expr.match(/\b_\b/)
          eval "Proc.new { |_| #{expr} }", block && block.binding
      else
          leftSection = expr.match(/^\s*(?:[+*\/%&|\^\.=<>\[]|!=)/m)
          rightSection = expr.match(/[+\-*\/%&|\^\.=<>!]\s*$/m)
          if leftSection || rightSection then
              if (leftSection) then
                  params.push('$left')
                  expr = '$left' + expr
              end
              if (rightSection) then
                  params.push('$right')
                  expr = expr + '$right'
              end
          else
              self.gsub(
                  /(?:\b[A-Z]|\.[a-zA-Z_$])[a-zA-Z_$\d]*|[a-zA-Z_$][a-zA-Z_$\d]*:\
|self|arguments|'(?:[^'\\]|\\.)*'|"(?:[^"\\]|\\.)*"/, ''
              ).scan(
                /([a-z_$][a-z_$\d]*)/i
              ) do |v|
                params.push(v) unless params.include?(v)
              end
```

[8] http://github.com/gilesbowkett/archaeopteryx/tree/master

```
          end
          eval "Proc.new { |#{params.join(', ')}| #{expr} }", block && block.bind\
ing
      end
    end
  end
end
```

So we should support passing the declarative arguments by position as well as by 'name.' And with a final twist, if any of the declarative arguments aren't already lambdas, we'll try to create lambdas by sending them the message `to_proc`. So our goal is to write what we wrote above or either of the following and have it "just work:"

```
define_method :sum_squares, &multirec(
        lambda { |value| value.kind_of?(Numeric) }, # the only change right now
        lambda { |value| value ** 2 },
        lambda { |value| value },
        lambda { |list| list.inject() { |x,y| x + y } }
)

include 'string-to_proc'

define_method :sum_squares, &multirec("value.kind_of?(Numeric)", "value ** 2","va\
lue","value.inject(&'+')")
```

And here is the code that makes it work:

**recursive_combinators.rb**

```
module RecursiveCombinators

  separate_args = lambda do |args|
    if ![1,2,4,5].include?(args.length)
      raise ArgumentError
    elsif args.length <= 2
      steps = [:cond, :then, :before, :after].map { |k| args.first[k].to_proc }
      steps.push(args[1]) unless args[1].nil?
      steps
    else
      steps = args[0..3].map { |arg| arg.to_proc }
      steps.push(args[4]) unless args[4].nil?
```

```
      steps
    end
  end

  define_method :multirec do |*args|
    cond_proc, then_proc, before_proc, after_proc, optional_value = separate_args\
.call(args)
    worker_proc = lambda do |value|
      if cond_proc.call(value)
        then_proc.call(value)
      else
        after_proc.call(
          before_proc.call(value).map { |sub_value| worker_proc.call(sub_value) }
        )
      end
    end
    if optional_value.nil?
      worker_proc
    else
      worker_proc.call(optional_value)
    end
  end

  define_method :linrec do |*args|
    cond_proc, then_proc, before_proc, after_proc, optional_value = separate_args\
.call(args)
    worker_proc = lambda do |value|
      trivial_parts, sub_problem = [], value
      while !cond_proc.call(sub_problem)
        trivial_part, sub_problem = before_proc.call(sub_problem)
        trivial_parts.unshift(trivial_part)
      end
      trivial_parts.unshift(then_proc.call(sub_problem))
      trivial_parts.inject do |recombined, trivial_part|
        after_proc.call(trivial_part, recombined)
      end
    end
    if optional_value.nil?
      worker_proc
    else
      worker_proc.call(optional_value)
    end
```

```
  end

  module_function :multirec, :linrec

end
```

Now when we have trivial lambdas, we can use nice syntactic sugar to express them. `string_to_proc` is **not** part of our recursive combinators, but making recursive combinators flexible, we make it "play well with others," which is a win for our code.

## 8.8 Separating Implementation from Declaration

In Refactoring Methods with Recursive Combinators, we read the claim that by separating the recursion implementation from the declaration of how to perform the steps of an algorithm like `#rotate`, we leave ourselves the opportunity to improve the performance of our implementation without the risk of adding bugs to our declaration.

In other words, we can optimize `linrec` if we want to. Well, we want to. So what we're going to do is optimize its performance by trading time for space. Let's have a quick look at the `worker_proc` lambda inside of `linrec`:

```
worker_proc = lambda do |value|
  if cond_proc.call(value)
    then_proc.call(value)
  else
    trivial_part, sub_problem = before_proc.call(value)
    after_proc.call(
      trivial_part, worker_proc.call(sub_problem)
    )
  end
end
```

As you can see, it is recursive, it calls itself to solve each sub-problem. And here is an iterative replacement:

```
worker_proc = lambda do |value|
  trivial_parts, sub_problem = [], value
  while !cond_proc.call(sub_problem)
    trivial_part, sub_problem = before_proc.call(sub_problem)
    trivial_parts.unshift(trivial_part)
  end
  trivial_parts.unshift(then_proc.call(sub_problem))
  trivial_parts.inject do |recombined, trivial_part|
    after_proc.call(trivial_part, recombined)
  end
end
```

This version doesn't call itself. Instead, it uses an old-fashioned loop, accumulating the results in an array. In a certain sense, this uses more explicit memory than the recursive implementation. However, we both know that the recursive version uses memory for its stack, so that's a bit of a wash. However, the Ruby stack is limited while arrays can be much larger, so this version can handle much larger data sets.

If you drop the new version of `worker_proc` into the `linrec` definition, each and every method you define using `linrec` gets the new implementation, for free. This works because we separated the implementation of recursive divide and conquer algorithms from the declaration of the steps each particular algorithm. Here's our new version of `linrec`:

```
define_method :linrec do |*args|
  cond_proc, then_proc, before_proc, after_proc, optional_value = separate_args.c\
all(args)
  worker_proc = lambda do |value|
    trivial_parts, sub_problem = [], value
    while !cond_proc.call(sub_problem)
      trivial_part, sub_problem = before_proc.call(sub_problem)
      trivial_parts.unshift(trivial_part)
    end
    trivial_parts.unshift(then_proc.call(sub_problem))
    trivial_parts.inject do |recombined, trivial_part|
      after_proc.call(trivial_part, recombined)
    end
  end
  if optional_value.nil?
    worker_proc
  else
    worker_proc.call(optional_value)
  end
end
```

## 8.9 A Really Simple Recursive Combinator

In [Recursive Lambdas in Ruby using Object#tap](http://ciaranm.wordpress.com/2008/11/30/recursive-lambdas-in-ruby-using-objecttap/ “”), Ciaran McCreesh explained how he used `#tap` to write a recursive function without cluttering the scope up with an unneeded variable. (If you would like a refresher, `Object#tap` is explained in Kestrels[9]).

Ciaran’s final solution was:

```
lambda do | recurse, spec |
  case spec
    when AllDepSpec, ConditionalDepSpec
      spec.each { | child | recurse.call(recurse, child) }
    when SimpleURIDepSpec
      puts spec
  end
end.tap { | r | r.call(r, id.homepage_key.value) } if id.homepage_key
```

There are two great things about this solution. First, Ciaran doesn’t need to calculate a result, he is just performing this computation for its side-effect, `puts`. Therefore, using a kestrel like `#tap` signals that he is not interested in the result. Second, he is using an off-the-shelf component and not writing a “horrid untyped lambda calculus construct” to get the job done. Fewer moving parts is a laudable goal.

That being said, when solving other problems, this solution may not meet our needs:

- Since it doesn’t return a result, we cannot use it for functions that compute values and not just generate side effects;
- Within the lambda, our `recurse` function must be called with itself as a parameter. This mixes the mechanics of our recursive implementation up with the semantics of what we’re trying to accomplish.

If we find ourselves needing to work around these limitations, we’ll need to go a bit further. Let’s use a brutally trivial example, factorial. (The naive implementation of factorial is a *terrible* piece of programming, but it’s simple enough that we can focus on how we’re implementing recursion and not what we are computing).

We could use one of our existing recursive combinators like `linrec`:

[9] http://github.com/raganwald/homoiconic/tree/master/2008-10-29/kestrel.markdown#readme

```
include 'string-to_proc'

linrec('< 2', '1', 'n -> [n, n - 1]', '*').call(5)
        => 120

# or perhaps you prefer...

linrec(
        lambda { |n| n < 2 },
        lambda { |n| 1 },
        lambda { |n| [n, n - 1] },
        lambda { |n, m| n * m }
).call(5)
        => 120
```

That gets us what we want without using a untyped lambda calculus construct, because it uses a combinatorial logic construct instead. But let's work something out that is closer to the spirit of Ciaran's approach. For starters, we can't use `#tap` because we need the result of the computation, so we'll imagine we have a new method, `#rcall`. Our first cut will look like this:

```
class Proc

  def rcall(*args)
    call(self, *args)
  end

end

lambda { |r, n| n < 2 ? 1 : n * r.call(r, n-1) }.rcall(5)
```

That solves our first problem very nicely: we can call a lambda with a value and it knows to pass itself to itself. Now what about our second problem? We are still cluttering up the inside of our function with passing itself to itself. Instead of calling `r.call(r, n-1)`, can we just call `r.call(n-1)`?

That would make our function look a lot simpler.

Well, we start with `lambda { |r, *args| ... }`. But if we are to call `r.call(n)`, we need to pass in a function like `lambda { |*args| ... }`. What does that function do? Send the message `#rcall` to our original function, of course. So we can write:

```
class Proc

  def rcall(*args)
    call(lambda { |*args| self.rcall(*args) }, *args)
  end

end

lambda { |r, n| n < 2 ? 1 : n * r.call(n-1) }.rcall(5)
        => 120
```

And that's it, we've accomplished recursion without using any untyped lambda calculus constructs. It may look at first glance like we're using an anonymous recursive combinator like Y[10], but we aren't. We're actually taking advantage of Ruby's `self` variable, so `#rcall` does not really implement anonymous recursion, it just lets us write recursive lambdas without explicitly binding them to a variable.

And our new method, `#rcall`, returns a value from our recursion and doesn't force us to remember to pass our lambda to itself when making a recursive call.

Cheers!

```
class Proc

  def rcall(*args)
    call(lambda { |*args| self.rcall(*args) }, *args)
  end

end
```

[10] http://www.ece.uc.edu/~franco/C511/html/Scheme/ycomb.html

# 9 You can’t be serious!?

In Practical Recursive Combinators, we enhanced `multirec` (a/k/a “Divide and Conquer”) and `linrec` (“Linear Recursion”) to accept as arguments any object that supports the `#to_proc` method. Today we’re going demonstrate why: We will look at how removing the ceremony around lambdas makes using combinators like `multirec` more valuable for code we share with others.

Using recursive_combinators.rb[1] to define how to sum the squares of a nested list of numbers, we can write:

```
require 'recursive_combinators'

include RecursiveCombinators

multirec(
  lambda { |x| x.kind_of?(Numeric) },
  lambda { |x| x ** 2 },
  lambda { |x| x },
  lambda { |x| x.inject { |sum, n| sum + n } }
)
```

The trouble with this–to quote a seasonally appropriate character[2]–is the noise, noise, Noise, NOISE! All those lambdas and parameter declarations outweigh the actual logic we are declaring, so much so that declaring this function using our abstraction is longer and may seem more obscure than declaring it without the abstraction.

This whole thing reminds me of languages where the keywords must be in UPPER CASE. Reading code in such languages is like listening to a poetry reading where the author shouts the punctuation:

> Two roads diverged in a yellow wood COMMA!
> And sorry I could not travel both
> And be one traveler COMMA! long I stood
> And looked down one as far as I could
> To where it bent in the undergrowth SEMI-COMMA!!

[1] http://github.com/raganwald/homoiconic/tree/master/2008-11-26/recursive_combinators.rb

[2] http://www.amazon.com/gp/product/B000HA4WDY?ie=UTF8&tag=raganwald001-20&linkCode=as2&camp=1789&creative=390957&creativeASIN=B000HA4WDY

Finding ways to abbreviate our declaration is more than just a little "syntactic sugar:" It's a way of emphasizing what is important, our algorithms, and de-emphasizing what is not important, the scaffolding and ceremony of instantiating `Proc` objects in Ruby. One of those ways is to use `String#to_proc`[3].

# 9.1 String to Proc

`String#to_proc` adds the `#to_proc` method to the `String` class in Ruby. This allows you to write certain simple lambdas as strings instead of using the `lambda` keyword, the `proc` keyword, or `Proc.new`. The reason why you'd bother is that `String#to_proc` provides some shortcuts that get rid of the noise.

**gives**

`String#to_proc` provides several key abbreviations: First, `->` syntax for lambdas in Ruby 1.8. So instead of `lambda { |x,y| x + y }`, you can write `'x,y -> x + y'`. I read this out loud as "*x and y gives x plus y*."

This syntax gets rid of the noisy `lambda` keyword and is much closer to Ruby 1.9 syntax. And frankly, reading it out loud makes much more sense than reading lambdas aloud. Our example above could be written:

```
require 'string_to_proc'

multirec(
  'x -> x.kind_of?(Numeric)',
  'x -> x ** 2',
  'x -> x',
  'x -> x.inject { |sum, n| sum + n }'
)
```

This is a lot better than the version with lambdas, and if the `->` seems foreign, it is only because `->` is in keeping with modern functional languages and mathematical notation, while `lambda` is in keeping with Lisp and lambda calculus notation without the ability to use a single lambda character unicode.

**inferred parameters**

Second, `String#to_proc` adds inferred parameters: If you do not use `->`, `String#to_proc` attempts to infer the parameters. So if you write `'x + y'`, `String#to_proc` treats it as `x,y -> x + y`. There are certain expressions where this doesn't work, and you have to use `->`, but for really simple cases it works just fine. And frankly, for really simple cases you don't need the extra scaffolding. Here's our example with the first three lambdas using inferred parameters:

[3] http:string_to_proc.rb

```
multirec(
  'x.kind_of?(Numeric)',
  'x ** 2',
  'x',
  'z -> z.inject { |sum, n| sum + n }'
)
```

> I have good news and bad news about inferred parameters and `String#to_proc` in general. It uses regular expressions to do its thing, which means that complicated things often don't work. For example, nesting `->` only works when writing functions that return functions. So `'x -> y -> x + y'` is a function that takes an `x` and returns a function that takes a `y` and returns `x + y`. That works. But `'z -> z.inject(&"sum, n -> sum + n")'` does NOT work.
>
> I considered fixing this with more sophisticated parsing, however the simple truth is this: `String#to_proc` is not a replacement for `lambda`, it's a tool to be used when what you're doing is so simple that `lambda` is overkill. If `String#to_proc` doesn't work for something, it probably isn't ridiculously simple any more.

### it

The third abbreviation is a special case. If there is only one parameter, you can use _ (the underscore) without naming it. This is often called the "hole" or pronounced "it." If you use "it," then `String#to_proc` doesn't try to infer any more parameters, so this can help you write things like:

```
multirec(
  '_.kind_of?(Numeric)',
  '_ ** 2',
  '_',
  '_.inject { |sum, n| sum + n }'
)
```

Admittedly, use of "it"/the hole is very much a matter of taste.

### point-free

`String#to_proc` has a fourth and even more extreme abbreviation up its sleeve, point-free style[4]: "Function points" are what functional programmers usually call parameters. Point-free style consists of describing how functions are composed together rather than describing what happens with their arguments. So, let's say that I want a function that combines `.inject` with `+`. One way to say that is to say that I want a new function that takes its argument and applies an `inject` to it, and the inject takes another function with two arguments and applies a `+` to them:

[4] http://blog.plover.com/prog/haskell/

```
lambda { |z| z.inject { |sum, n| sum + n } }
```

The other way is to say that I want to compose `.inject` and `+` together. Without getting into a `compose` function like Haskell's `.` operator, `String#to_proc` has enough magic to let us write the above as:

```
".inject(&'+')"
```

Meaning "*I want a new lambda that does an inject using plus.*" Point-free style does require a new way of thinking about some things, but it is a clear win for simple cases. Proof positive of this is the fact that Ruby on Rails and Ruby 1.9 have both embraced point-free style with `Symbol#to_proc`. That's exactly how `(1..100).inject(&:+)`[5] works!

`String#to_proc` supports fairly simple cases where you are sending a message or using a binary operator. So if we wanted to go all out, we could write our example as:

```
multirec('.kind_of?(Numeric)', '** 2', 'x', ".inject(&'+')")
```

> There's no point-free magic for the identity function, although this example tempts me to special case the empty string!

### When should we use all these tricks?

`String#to_proc` provides these options so that you as a programmer can choose your level of ceremony around writing functions. But of course, you have to use the tool wisely. My *personal* rules of thumb are:

1. Embrace inferred parameters for well-known mathematical or logical operations. For these operations, descriptive parameter names are usually superfluous. Follow the well-known standard and use `x`, `y`, `z`, and `w`; or `a`, `b` and `c`; or `n`, `i`, `j`, and `k` for the parameters. If whatever it is makes no sense using those variable names, don't used inferred parameters.
2. Embrace the hole for extremely simple one-parameter lambdas that aren't intrinsically mathematical or logical such as methods that use `.method_name` and for the identity function.
3. Embrace point-free style for methods that look like operators.
4. Embrace `->` notation for extremely simple cases where I want to give the parameters a descriptive name.
5. Use lambdas for everything else.

So I would write:

[5] http://weblog.raganwald.com/2008/02/1100inject.html

```
multirec( '_.kind_of?(Numeric)', '** 2', '_', "_.inject(&'+')")
```

I read the parameters out loud as:

- *it kind_of? Numeric;*
- *raise to the power of two;*
- *it;*
- *it inject plus.*

And yes, I consider `multirec( '_.kind_of?(Numeric)', '** 2', '_', "_.inject(&'+')")` more succinct and easier to read than:

```
def sum_squares(value)
  if value.kind_of?(Numeric)
    value ** 2
  else
    value.map do |sub_value|
      sum_squares(sub_value)
    end.inject { |x,y| x + y }
  end
end
```

If all this is new too you, `String#to_proc` may seem like gibberish and `def sum_squares` may seem reassuringly sensible. But try to remember that combinators like `multirec` are built to disentangle the question of what we are doing from how we are doing it. This is the third straight post about recursive combinators using one of three different examples. So of course we know what `sum_squares` does and how it does it.

But try to imagine you are looking at a piece of code that isn't so simple, that isn't so obvious. Maybe it was written by someone else, maybe you wrote it a while ago. If you see:

```
def rotate(square)
  return square unless square.kind_of?(Enumerable) && square.size > 1
  half_sz = square.size / 2
  sub_square = lambda do |row, col|
    square.slice(row, half_sz).map { |a_row| a_row.slice(col, half_sz) }
  end
  upper_left = rotate(sub_square.call(0,0))
  lower_left = rotate(sub_square.call(half_sz,0))
  upper_right = rotate(sub_square.call(0,half_sz))
  lower_right = rotate(sub_square.call(half_sz,half_sz))
  upper_right.zip(lower_right).map { |l,r| l + r } +
          upper_left.zip(lower_left).map { |l,r| l + r }
end
```

Do you see at once how it works? Do you see at a glance whether the recursive strategy was implemented properly? Can you tell whether there's something buggy about it? For example, this code only works rotating square matrices that have sides which are powers of two. What needs to be changed to fix that? Are you sure you can fix it without breaking the divide and conquer strategy?

For a method like this, I would write:

```
multirec(
  :cond => "!(_.kind_of?(Enumerable) && _.size > 1)",
  :then => "_",
  :before => lambda do |square|
          half_sz = square.size / 2
          sub_square = lambda do |row, col|
            square.slice(row, half_sz).map { |a_row| a_row.slice(col, half_sz) }
          end
          upper_left = sub_square.call(0,0)
          lower_left = sub_square.call(half_sz,0)
          upper_right = sub_square.call(0,half_sz)
          lower_right = sub_square.call(half_sz,half_sz)
          [upper_left, lower_left, upper_right, lower_right]
  end,
  :after => lambda do |list|
          upper_left, lower_left, upper_right, lower_right = list
          upper_right.zip(lower_right).map(&'+') + upper_left.zip(lower_left).map(&'+')
  end
end
```

And be assured that months from now if I wanted to support rotating rectangular matrices of arbitrary size, I could modify `:cond`, `:before`, and `:after` with confidence that the basic method was not being broken.

## 9.2 The Message

The message here is that taken by themselves, tools like recursive combinators or `String#to_proc` just look strange. But when we use them together, they reinforce each other and the sum becomes much greater than the sum of the parts. In the case of `String#to_proc`, it looks like frivolity to most Ruby programmers, because they don't use that many lambdas: Why should they when the existing syntax makes writing combinators hard to use? But when we have combinators in our hand, we see how `String#to_proc` can make them a win. So two things that look weird on their own are a useful tool when used in conjunction.

Our final example ended up being slightly longer than a naive version, however it is longer in ways that matter rather than longer in a mindless ceremonial way like some languages.

And that's the point of languages like Ruby: **You** have the tools to decide which portions of you code matter more than others, and to make the parts that matter stand out and the parts that don't go away. You may disagree with my choice of what matters for a recursive divide and conquer algorithm, but I hope we can agree that it's valuable to be able to make that choice for yourself or your team.

Seriously.

# 10 The Hopelessly Egocentric Book Chapter

In Raymond Smullyan's delightful book on Combinatory logic, To Mock a Mockingbird[1], Smullyan explains combinatory logic and derives a number of important results by presenting the various combinators as songbirds in a forest. One of his concepts is the Hopelessly Egocentric Bird:

> We call a bird B *hopelessly egocentric* if for *every* bird `x`, `Bx = B`. This means that whatever bird `x` you call out to `B` is irrelevant; it only calls `B` back to you! Imagine that the bird's name is Bertrand. When you call out "Arthur," you get the response "Bertrand"; when you call out "Raymond," you get the response "Bertrand"; when you call out "Ann," you get the response "Bertrand." All this bird can ever think about is itself!

Some folks have proposed that by making Ruby's `nil` hopelessly egocentric, we can avoid the need for monadic idioms like `#andand`. Let's examine the idea and see what consequences this has.

## 10.1 Object-oriented egocentricity

One of the tenets of OO programming is that programs consist of *objects* that respond to *messages* they send each other. A hopelessly egocentric object is easy to imagine: No matter what message you send it, the hopelessly egocentric object responds with itself:

```
class HopelesslyEgocentric < BlankSlate

  def method_missing(*arguments); self; end

end
```

Now you can create a hopelessly egocentric object with `HopelesslyEgocentric.new` and no matter what message you send it, you will get it back in response. And? What good is this? What can it do? Why should we put it in our Zoo?

In Objective C, nil is hopelessly egocentric. As Learn Objective-C[2] puts it, *You usually don't need to check for nil before calling a method on an object. If you call a method on nil that returns an object,*

[1] http://www.amazon.com/gp/product/0192801422?ie=UTF8&tag=raganwald001-20&linkCode=as2&camp=1789&creative=9325&creativeASIN=0192801422

[2] http://cocoadevcentral.com/d/learn_objectivec/

*you will get nil as a return value.* The idea here is that instead of getting a `NoMethodError` when we send a message to nil, we get nil back.

Some people like this so much they've composed the same semantics for Ruby[3]:

```
class NilClass

  def method_missing(*args); nil; end

end
```

Now instead of writing `person && person.name && person.name.upcase` or `person.andand.name.andand.upcase`, you write `person.name.upcase` and either get the person's name in upper case or nil. Wonderful! Or is it? Let's take a look at what we're trying to accomplish and the limitations of this approach.

**queries**

Hopelessly egocentric nil works reasonably for querying *properties*, in other words sub-entities when an entity is constructed by composition, things like `.name`. I'm quite happy if `person.name` returns nil whether we don't have a person or if the person doesn't have a name. And we can extend this to what I would call *purely functional transformations* like `.upcase`. Just as `''.upcase` is `''`, it is reasonable to think of `nil.upcase` as nil.

Now let's look at some things that aren't properties and aren't purely functional transformations. What do we do with methods that are intended to *update* their receiver? Consider a bank account object. Do we really want to write things like:

```
person.account = nil
person.account.increment_balance(100)
  => nil
```

This makes no sense. If we want to give them a hundred dollars, we had better have their actual account on hand! Clearly there is a huge difference between methods that are *queries* and methods that are *updates*. (Note that `andand` doesn't save us either, except by virtue of being explicit rather than magical so we can eschew it for update methods like `#increment_balance`.)

**updates**

Now that we are talking about methods with side-effects, let's be more specific. Our hopelessly egocentric nil does return nil to any method. But it has another property, it has no side-effects. This is sometimes what we want! Let's look at our nil account again. What about this code:

[3] http://rubyenrails.nl/articles/2008/02/29/our-daily-method-18-nilclass-method_missing

```
person.account.update_attribute(:primary_email, 'reg@braythwayt.com')
```

To decide what we think of this, we need to be specific about the meaning of nil. Generally, nil means one of two things:

1. NONE, meaning "There isn't one of these," or;
2. UNKNOWN, meaning "There *is* one of these, but we don't know what it is."

`person.account.update_attribute(:primary_email, 'reg@braythwayt.com')` is an example of why this difference matters. If `person.account` is an account, we want to update its primary email address, of course. And if `person.account` is NONE, we might be very happy not updating its primary email address. Perhaps our code looks like this:

```
class Person < ActiveRecord::Base
  belongs_to :account

  def update_email(new_email)
    self.class.transaction do
      update_attribute(:primary_email, new_email)
      account.update_attribute(:primary_email, new_email)
    end
  end

  # ...

end

Person.find(:first, :conditions => {...}).update_email('reg@braythwayt.com')
```

Meaning, update our person's primary email address, and if they have an account, update it too. If nil means NONE, this works. But what if nil really means UNKNOWN rather than NONE? **Now it is wrong to silently fail**. Let me give you a very specific way this can happen. When performing a database query, we can specify the exact columns we want returned. In Active Record, we might write something like this:

```
person = Person.find(:first, :conditions => {...}, :select => 'id, name')
```

What this means is that there is an `account_id` column in the `people` table, however we are deliberately not loading it into `person`. ActiveRecord will still supply us with a `#account` method, however it will return nil. This absolutely, positively means that `person.account` is UNKNOWN, not NONE. There could well be an account in our database for this person, and now if we write:

```
person.update_email('reg@braythwayt.com')
```

We do not want it to silently ignore the account email update, because we haven't loaded the `account` associated model. So for UNKNOWN, our two rules are:

1. Querying UNKNOWN returns UNKNOWN;
2. All attempts to update UNKNOWN are errors.

What about NONE? We gave two examples of updates, one of which was a really bad idea,`#increment_balance`, and the other of which was fine `update_attribute(:primary_email, new_email)`. Thus we have three rules for NONE:

1. Querying NONE returns NONE;
2. Some updates to NONE may return NONE and have no side effects;
3. Some updates to NONE may be errors.

With a little forethought and design, you may be able to construct one or more classes if your application for which all updates to NONE return NONE and have no side effects. But for all others, methods like `#increment_balance` represent a semantic problem with using a hopelessly egocentric nil to represent NONE. We also see a problem with writing a hopelessly egocentric nil to handle UNKNOWN: How does it know which methods are queries and which methods are updates?

If we work really hard and eliminate all possibility of an update to NONE being an error, are there any other issues with using a hopelessly egocentric nil? Let's return to our initial case:

```
person.name
  => nil

person.name.upcase
  => nil
```

Makes sense. And then we write:

```
person.name + ", esq."
  => nil
```

Dubious, but let's go with it. If this makes sense, we ought to be able to write this as well:

```
"Mister " + person.name
  => TypeError: can't convert nil into String
```

Why is this an error?[4] Things don't get any better using a hopelessly egocentric nil to handle UNKNOWN. Even if we can get past the issue of update methods, we have another problem that is much more difficult to resolve. UNKNOWN introduces tri-value logic:

```
UNKNOWN == Object.new
  => UNKNOWN
UNKNOWN != Object.new
  => UNKNOWN
UNKNOWN == UNKNOWN
  => UNKNOWN
UNKNOWN != UNKNOWN
  => UNKNOWN
Object.new == UNKNOWN
  => UNKNOWN
Object.new != UNKNOWN
  => UNKNOWN
```

When you don't know something's value, it is neither equal to nor not equal to any other value, including another unknown value. And our fifth and sixth examples suffer from the same problem as `nil + ", esq."` vs. `"Mister " + nil`. We would need to patch all sorts of other objects to make equality testing many many other methods work. (What is `42 < UNKNOWN`?) But things get worse:

How does *truthiness* work? In Ruby, you cannot override the way `and`, `or`, `if`, `unless`, `&&`, and `||` work. What are the semantics of `if UNKNOWN`? What do `true && UNKNOWN` or `UNKNOWN or true` return? Before implementing a true UNKNOWN in any language, I would want those questions answered.

Finally, there is actually a fifth and sixth rule that we are ignoring because these examples are in Ruby rather than a language with an expressive type system. Consider:

```
'Reg Braithwaite'.wealthy?
  => NoMethodError: undefined method `wealthy?' for "Reg Braithwaite":String
```

And now we write:

```
person.name.wealthy?          # or...
person.name.andand.wealthy?
```

[4] http://weblog.raganwald.com/2007/10/too-much-of-good-thing-not-all.html

What happens if `person.name` is NONE? What happens if `person.name` is UNKNOWN? Our problem here is that `#wealthy?` is never a valid message to send to something returned by `person.name`. Our behaviour ought to be:

- Sending an invalid message to NONE raises a `NoMethodError`;
- Sending an invalid message to UNKNOWN raises a `NoMethodError`.

There is no easy way to do this in Ruby, of course. Not only do we have trouble disambiguating queries from updates, we have trouble disambiguating valid from invalid messages.

For all of these reasons, I am loathe to implement a hopelessly egocentric nil and prefer to use an explicit idiom like `#andand` or `#try`. With explicit idioms, I can deal with the ambiguity between nil meaning NONE and nil meaning UNKNOWN and make sure my code does not violate the rules given here. But what I like about the *idea* of a hopelessly egocentric nil is that thinking the consequences provokes me to really think about the semantics of my data schemas.

Representing NONE and UNKNOWN values is a subtle problem requiring a deep and pervasive approach to typing similar to C++'s `const` keyword and/or writing custom null objects[5] that understand which methods are safe to respond egocentrically and which are errors.

[5] http://en.wikipedia.org/wiki/Null_Object_pattern

# 11 Bonus Chapter: Separating Concerns in Coffeescript using Aspect-Oriented Programming

*This chapter isn't strictly about combinatory logic and it especially isn't about Ruby programming. However, once you grasp the underlying fundamental principles, you can apply them in other environments using other programming languages.*

*You shouldn't find it too difficult to relate the content to previous chapters, the title alone provides a massive hint.*

Modern object-oriented software design favours composition over inheritance[1] and celebrates code that is DRY[2]. The idea is to separate each object's concerns and responsibility into separate units of code, each of which have a single responsibility. When two different types of objects share the same functionality, they do not repeat their implementation, instead they share their implementation.

When composing functionality at the method level of granularity, techniques such as mixins and delegation are effective design tools. But at a finer level of granularity, we sometimes wish to share functionality within methods. In a traditional design, we have to extract the shared functionality into a separate method that is called by other methods.

**decomposing methods**

You might think of extracting smaller methods from bigger methods as *decomposing* methods. You break them into smaller pieces, and thus you can share functionality or rearrange the pieces so that your code is organized by responsibility.

For example, let's say that we are writing a game for the nostalgia market, and we wish to use partially constructed objects to save resources. When we go to actually use the object, we *hydrate* it, loading the complete object from persistent storage. This is a coarse kind of *lazy evaluation.*

Here's some bogus code:

[1] https://en.wikipedia.org/wiki/Composition_over_inheritance
[2] http://en.wikipedia.org/wiki/Don't_repeat_yourself

```
class Wumpus
        roar: ->
                # code that hydrates a Wumpus
                # ...
                # code that roars
                # ...
        run: ->
                # code that hydrates a Wumpus
                # ...
                # code that runs
                # ...

class Hunter
        draw: (bow) ->
                # code that hydrates a Hunter
                # ...
                # code that draws a bow
                # ...
        run: ->
                # code that hydrates a Hunter
                # ...
                # code that runs
                # ...
```

We can decompose it into this:

```
class Wumpus
        roar: ->
                hydrate(this)
                # code that roars
                # ...
        run: ->
                hydrate(this)
                # code that runs
                # ...

class Hunter
        draw: (bow) ->
                hydrate(this)
                # code that draws a bow
                # ...
        run: ->
```

```
                hydrate(this)
                # code that runs
                # ...

hydrate = (object) ->
        # code that hydrates the object from storage
```

### composing methods

On an ad hoc basis, decomposing methods is fine. But there is a subtle problem. Implementation tricks like hydrating objects, memoizing return values, or other performance tweaks are orthogonal to the mechanics of what methods like `roar` or `run` are supposed to do. So why is `hydrate(this)` in every method?

Now the obvious answer is, "Ok, it might be orthogonal to the main business of each method, but it's just one line." The trouble with this answer is that method decomposition doesn't scale. We need a line for hydration, a line or two for logging, a few lines for error handling, another for wrapping certain things in a transaction…

Even when each orthogonal concern is boiled down to just one line, you can end up having the orthogonal concerns take up more space than the main business. And that makes the code hard to read in practice. You don't believe me? take a look at just about every programming tutorial ever written. They almost always say "Hand waving over error handling and this and that" in their code examples, because they want to make the main business of the code clearer and easier to read.

We ought to do the same thing, move hydration, error handling, logging, transactions, and anything else orthogonal to the main business of a method out of the method. And we can.

### method combinations

Here's our code again, this time using the YouAreDaChef[3] library to provide *before combinations*:

```
YouAreDaChef = require('YouAreDaChef.coffee').YouAreDaChef

class Wumpus
        roar: ->
                # ...
        run: ->
                #...

class Hunter
        draw: (bow) ->
                # ...
        run: ->
```

[3] https://github.com/raganwald/YouAreDaChef

```
                #...

hydrate = (object) ->
        # code that hydrates the object from storage

YouAreDaChef(Wumpus, Hunter)
        .before 'roar', 'draw', 'run', () ->
                hydrate(this)
```

Whenever the `roar`, `draw`, or `run` methods are called, YouAreDaChef calls `hydrate(this)` first. And the two concerns–How a Wumpus works and when it ought to be hydrated–are totally separated. This isn't a new idea, it's called aspect-oriented programming[4], and practitioners will describe what we're doing in terms of method advice and point cuts.

Ruby on Rails programmers are familiar with this idea. If you have ever written any of the following, you were using Rails' built-in aspect-oriented programming support:

```
after_save
validates_each
alias_method_chain
before_filter
```

These and other features of Rails implement method advice, albeit in a very specific way tuned to portions of the Rails framework.

**the unwritten rule**

> There is an unwritten rule that says every Ruby programmer must, at some point, write his or her own AOP implementation –Avdi Grimm

Let's look at how YouAreTheChef works. Here's a simplified version of the code for the `before` combination:

[4] http://en.wikipedia.org/wiki/Aspect-oriented_programming

```
YouAreDaChef: (clazzes...) ->
    before: (method_names..., advice) ->
      _.each method_names, (name) ->
        _.each clazzes, (clazz) ->
          if _.isFunction(clazz.prototype[name])
            pointcut = clazz.prototype[name]
            clazz.prototype[name] = (args...) ->
              advice.apply(this, args)
              pointcut.call(this, args)
```

This is really simple, we are composing a method with a function. The method already defined in the class is called the *pointcut*, and the function we are supplying is called the *advice*. Unlike a purely functional combinator, we are only executing the advice for side-effects, not for its result. But in object-oriented imperative programming, that's usually what we want.

**other method combinations**

That looks handy. But we also want an *after method*, a way to compose methods in the other order. Good news, the after combination is exactly what we want. After combinations are very handy for things like logging method calls or cleaning things up.

But there's another great use for after combinators, triggering events. Event triggering code is often very decoupled from method logic: The whole point of events is to invert control so that an object like a `Wumpus` doesn't need to know which objects want to do something after it moves. For example, a Backbone.js view might be observing the Wumpus and wish to update itself when the Wumpus moves:

```
YouAreDaChef(Wumpus, Hunter)
        .after 'run', () ->
                this.trigger 'move', this

CaveView = Backbone.View.extend
        initialize: ->
                # ...
                @model.bind 'move', @wumpusMoved
        wumpusMoved: (wumpus) ->
                # ...
```

The code coupling the view to the model has now been separated from the code defining the model itself.

YouAreDaChef also provides other mechanisms for separating concerns. *Around combinations* (also called around advice) are a very general-purpose combinator. With an around combination, the original method (the pointcut) is passed to the advice function as a parameter, allowing it to be called at any time.

Around advice is useful for wrapping methods. Using an around combinator, you could bake error handling and transactions into methods without encumbering their code with implementation details. In this example, we define the methods to be matched using a regular expression, and YouAreDaChef passes the result of the match to the advice function, which wraps them in a transaction and adds some logging:

```
class EnterpriseyLegume
  setId:         (@id)         ->
  setName:       (@name)       ->
  setDepartment: (@department) ->
  setCostCentre: (@costCentre) ->

YouAreDaChef(EnterpriseyLegume)
  .around /set(.*)/, (pointcut, match, value) ->
    performTransaction () ->
      writeToLog "#{match[1]}: #{value}"
      pointcut(value)
```

**summary**

Method combinations are a technique for separating concerns when the level of granularity is smaller than a method. This makes the code DRY and removes the clutter of orthogonal responsibilities.

# 12 Appendix: Finding Joy in Combinators

In this book, we have looked at a few interesting combinators and some Ruby code inspired by them. Today we'll review the definition of a combinator, and from there we'll learn something intriguing about an entire family of programming languages, the Concatenative Languages[1].

Let's start at the beginning: What is a combinator?

One definition of a combinator is *a function with no free variables*. Another way to put it is that a combinator is a function that takes one or more arguments and produces a result without introducing anything new. In Ruby terms, we are talking about blocks, lambdas or methods that do not call anything except what has been passed in.

So if I tell you that:

```
finch.call(a).call(b).call(c)
        => c.call(b).call(a)
```

Then you know that `finch` is a combinator because the effect it produces is made up solely of combining the effects of the things it takes as parameters. That's easy, but yet… Where is our vaunted simplicity? Working with Ruby's lambdas and braces and calls gets in our way. We can learn a lot from combinatorial logic to help our Ruby programming, but Ruby is a terrible language for actually learning about combinatorial logic.

## 12.1 Languages for combinatorial logic

Combinatorial logicians use a much simpler, direct syntax for writing expressions:

```
Fabc => cba
```

Whenever a logician writes `abc`, he means the same thing as when a Rubyist writes `a.call(b).call(c)`. Note that like Ruby, the precedence in combinatorial logic is to the left, so `abc` is equivalent to `(ab)c` just as in Ruby `a.call(b).call(c)` is equivalent to `(a.call(b)).call(c)`.

I think you'll agree that `abc` is much simpler than `a.call(b).call(c)`. Here's another look at the combinators we've met in this series, using the simpler syntax:

[1] http://en.wikipedia.org/wiki/Concatenative_programming_language

```
Kxy => x
Txy => yx
Cxyz => xzy
Q3xyz => z(xy) # Q3 is shorthand for the Quirky bird
Bxyz = x(yz)
Qxyz = y(xz) # Q is shorthand for the Queer bird
```

There are many, many more combinators, of course. Infinitely more, in fact. We only have names for some of the most useful. For example, the Warbler Twice Removed, or `W**` is written:

```
W**xyzw => xyzww
```

(Warblers are actually in a whole 'nother family of birds that introduce *duplication*. Other members of that family include the Mockingbird and Starling. They're incredibly useful for introducing ideas like iteration and recursion.)

You could say that combinators take a string of symbols (like x, y, z, w, and so forth), then they introduce some erasing, some duplication, some permutation, and add some parentheses. That they work to rearrange our string of symbols.

We have seen that parentheses are allowed, and that some combinators introduce parentheses. Before you say that the combinators introduce new symbols, remember that parentheses are *punctuation*. If you think of the symbols as words and the parentheses as punctuation, you see that the combinators simply rearrange the words and change the punctuation without introducing new words.

Now I said that combinators work with strings of symbols. This was a terrible analogy, because it made us talk about punctuation and why parentheses are not symbols. Another thing you could say is that combinator work with *lists* of symbols, then they re-arrange the symbols, including removing symbols, introducing sub-lists, and duplicating symbols.

This is more interesting! Now we can see that in our notation, adding parentheses is a way of introducing a sub list. Let's revisit the bluebird:

```
Bxyz = x(yz)
```

Now what we can say is this: The bluebird takes a list of three symbols and answers a list of one symbol and a sublist of two symbols. In Ruby:

```
bluebird = lambda { |*args|
        x, y, z = args
        [x, [y, z]]
}

bluebird.call(:x, :y, :z)
        => [:x, [:y, :z]]
```

This is easy. What about the Thrush?

```
thrush = lambda { |*args|
        x, y = args
        [y, x]
}

thrush.call(:x, :y)
        => [:y, :x]
```

Now let's pause for a moment. Imagine we had an entire programming language devoted to this style of programming. The primary thing it does is define combinators that take a list of symbols and recombine them. Since it works with lists and we are thinking about combinatory logic, we will represent our expressions as lists:

```
idiot :x
        => :x

mockingbird :x
        => :x :x

bluebird :x :y :z
        => :x [:y :z]

thrush :x :y
        => :y :x
```

Wait! Do not shout Lisp! Just because we have lists of things does not mean we are programming in Lisp!! Let's keep going, and you will see in the next example that I do not mean Lisp:

```
bluebird thrush :x :y :z
        => thrush [:x :y] :z
        => :z [:x :y]
```

And therefore in our fictitious language we can write:

```
quirky = bluebird thrush
```

And thus:

```
quirky :x :y :z
        => :z [:x :y]
```

This looks familiar. Have you ever written a program in Postscript[2]? Or [Forth](http://en.wikipedia.org/wiki/Forth_(programming_language)? What if instead of using a thrush we used a word called `swap`? Or instead of a mockingbird we used a word called `dup`?

## 12.2 Concatenative languages

Concatenative (or stack-based) programming languages–like Postscript, Forth, Factor[3], and Joy[4]–are almost direct representations of combinatorial logic. There is a list of things, words or combinators permute the list of things, and the things can be anything: data, other combinators, or even programs. These languages are called concatenative languages because the primary way to compose programs and combinators with each other is to concatenate them together, like we did with the bluebird and thrush above.

> For me the purpose of life is partly to have joy. Programmers often feel joy when they can concentrate on the creative side of programming, So Ruby is designed to make programmers happy. –Yukihiro Matsumoto

You have probably heard that it is a good idea to learn a new programming language every year. Is a concatenative language on your list of languages to learn? No? Well, here is the reason to learn a concatenative language: *You will learn to think using combinatorial logic.* For example, the Y Combinator is expressed in Joy as:

[2] http://en.wikipedia.org/wiki/PostScript

[3] http://www.factorcode.org/

[4] http://www.latrobe.edu.au/philosophy/phimvt/joy/j00ovr.htmll

```
[dup cons] swap concat dup cons i
```

Where `dup` is a mockingbird, `swap` is a thrush, `i` is an idiot bird, and `cons` and `concat` are likewise two other combinators. Writing in Joy is writing directly in combinators.

In other programming languages, combinatorial logic is an underpinning. It helps us explain and prove certain things, It inspires us to invent certain things. It is behind everything we do. That's good. But in a concatenative language, it is not an underpinning or behind a curtain. It is right out there in front of you. And learning to program in a concatenative language means learning to think in combinators.

The combinators we've discussed in depth so far are all fascinating, however as a basis for writing programs they are incomplete. You cannot represent every possible program using kestrels, thrushes, cardinals, quirky birds, bluebirds, and queer birds. To represent all possible programs, we need to have at least one combinator that duplicates symbols, like a mockingbird or another from its family.

# 13 Appendix: Source Code

All source code is published under the following license:

```
# The MIT License
#
# All contents Copyright (c) 2004-2008 Reginald Braithwaite
# <http://braythwayt.com>  except as otherwise noted.
#
# Permission is hereby granted, free of charge, to any person obtaining a copy
# of this software and associated documentation files (the "Software"), to deal
# in the Software without restriction, including without limitation the rights
# to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
# copies of the Software, and to permit persons to whom the Software is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in
# all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
# AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
# THE SOFTWARE.
#
# http://www.opensource.org/licenses/mit-license.php
```

## 13.1 kestrels

**returning.rb**

```ruby
# The MIT License
#
# All contents Copyright (c) 2004-2008 Reginald Braithwaite
# <http://braythwayt.com>  except as otherwise noted.
#
# Permission is hereby granted, free of charge, to any person obtaining a copy
# of this software and associated documentation files (the "Software"), to deal
# in the Software without restriction, including without limitation the rights
# to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
# copies of the Software, and to permit persons to whom the Software is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in
# all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
# AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
# THE SOFTWARE.
#
# http://www.opensource.org/licenses/mit-license.php

unless respond_to?(:returning)
  def returning(it)
    yield it
    it
  end
end
```

## 13.2 thrushes

**into.rb**

```
class Object
  def into expr = nil
    expr.nil? ? yield(self) : expr.to_proc.call(self)
  end
end
```

**let.rb**

```
module Kernel
  def let it
    yield it
  end
end
```

# 13.3 the cardinal

**cardinal.rb**

```
def cardinal_define(name, &proc_over_proc)
  define_method_taking_block(name) do |a_value, a_proc|
    proc_over_proc.call(a_proc).call(a_value)
  end
end

# method_body_proc should expect (a_value, a_proc)
# see http://coderrr.wordpress.com/2008/10/29/using-define_method-with-blocks-in-\
ruby-18/
def define_method_taking_block(name, &method_body_proc)
  self.class.send :define_method, "__cardinal_helper_#{name}__", method_body_proc
  eval <<-EOM
    def #{name}(a_value, &a_proc)
      __cardinal_helper_#{name}__(a_value, a_proc)
    end
  EOM
end
```

# 13.4 quirky birds

**andand.rb**

```
module AndAnd

  # This module is included in Object, so each of these methods are added
  # to Object when you require 'andand'. Each method is an *adverb*: they are
  # intended to be enchained with another method, such as receiver.adverb.method
  #
  # The purpose of an adverb is to modify what the primary method returns.
  #
  # Adverbs also take blocks or procs, passing the receiver as an argument to the
  # block or proc. They retain the same semantics with a block or proc as they
  # do with a method. This behaviour weakly resembles a monad.
  module ObjectGoodies

    # Returns nil if its receiver is nil, regardless of whether nil actually hand\
les the
    # actual method ot what it might return.
    #
    #    'foo'.andand.size => 3
    #    nil.andand.size => nil
    #    'foo'.andand { |s| s << 'bar' } => 'foobar'
    #    nil.andand { |s| s << 'bar' } => nil
    def andand (p = nil)
      if self
        if block_given?
          yield(self)
        elsif p
          p.to_proc.call(self)
        else
          self
        end
      else
        if block_given? or p
          self
        else
          MockReturningMe.new(self)
        end
      end
```

```
end

# Invokes the method and returns the receiver if nothing is raised. Therefore,
# the purpose of calling the method is strictly for side effects. In the block
# form, it resembles #tap from Ruby 1.9, and is useful for debugging. It also
# resembles #returning from Rails, with slightly different syntax.
#
#    Object.new.me do |o|
#      def o.foo
#        'foo'
#      end
#    end
#      => your new object
#
# In the method form, it is handy for chaining methods that don't ordinarily
# return the receiver:
#
#    [1, 2, 3, 4, 5].me.pop.reverse
#      => [4, 3, 2, 1]
def me (p = nil)
  if block_given?
    yield(self)
    self
  elsif p
    p.to_proc.call(self)
    self
  else
    ProxyReturningMe.new(self)
  end
end

unless Object.instance_methods.include?('tap')
  alias :tap :me
end

# Does not invoke the method or block and returns the receiver.
# Useful for comemnting stuff out, especially if you are using #me for
# debugging purposes: change the .me to .dont and the semantics of your
# program are unchanged.
#
#    [1, 2, 3, 4, 5].me { |x| p x }
#      => prints and returns the array
```

```
    #    [1, 2, 3, 4, 5].dont { |x| p x }
    #      => returns the array without printing it
    def dont (p = nil)
      if block_given?
        self
      elsif p
        self
      else
        MockReturningMe.new(self)
      end
    end

  end

end

class Object
  include AndAnd::ObjectGoodies
end

unless Module.constants.include?('BlankSlate')
  if Module.constants.include?('BasicObject')
    module AndAnd
      class BlankSlate < BasicObject
      end
    end
  else
    module AndAnd
      class BlankSlate
        def self.wipe
          instance_methods.reject { |m| m =~ /^__/ }.each { |m| undef_method m }
        end
        def initialize
          BlankSlate.wipe
        end
      end
    end
  end
end

module AndAnd
```

```
  # A proxy that returns its target without invoking the method you
  # invoke. Useful for nil.andand and #dont
  class MockReturningMe < BlankSlate
    def initialize(me)
      super()
      @me = me
    end
    def method_missing(*args)
      @me
    end
  end

  # A proxy that returns its target after invoking the method you
  # invoke. Useful for #me
  class ProxyReturningMe < BlankSlate
    def initialize(me)
      super()
      @me = me
    end
    def method_missing(sym, *args, &block)
      @me.__send__(sym, *args, &block)
      @me
    end
  end

end
```

**blank_slate.rb**

```
# The MIT License
#
# All contents Copyright (c) 2004-2008 Reginald Braithwaite
# <http://braythwayt.com>  except as otherwise noted.
#
# Permission is hereby granted, free of charge, to any person obtaining a copy
# of this software and associated documentation files (the "Software"), to deal
# in the Software without restriction, including without limitation the rights
# to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
# copies of the Software, and to permit persons to whom the Software is
# furnished to do so, subject to the following conditions:
#
```

```
# The above copyright notice and this permission notice shall be included in
# all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
# AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
# THE SOFTWARE.
#
# http://www.opensource.org/licenses/mit-license.php

unless Module.constants.include?('BlankSlate')
  if Module.constants.include?('BasicObject')
    class BlankSlate < BasicObject
    end
  else
    class BlankSlate
      def self.wipe
        instance_methods.reject { |m| m =~ /^__/ }.each { |m| undef_method m }
      end
      def initialize
        BlankSlate.wipe
      end
    end
  end
end
```

**quirky_bird.rb**

```
# The MIT License
#
# All contents Copyright (c) 2004-2008 Reginald Braithwaite
# <http://braythwayt.com>  except as otherwise noted.
#
# Permission is hereby granted, free of charge, to any person obtaining a copy
# of this software and associated documentation files (the "Software"), to deal
# in the Software without restriction, including without limitation the rights
# to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
# copies of the Software, and to permit persons to whom the Software is
```

```
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in
# all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
# AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
# THE SOFTWARE.
#
# http://www.opensource.org/licenses/mit-license.php

def quirky_bird_define(name, &value_proc)
  define_method_taking_block(name) do |a_value, a_proc|
    a_proc.call(value_proc.call(a_value))
  end
end

# method_body_proc should expect (a_value, a_proc)
# see http://coderrr.wordpress.com/2008/10/29/using-define_method-with-blocks-in-\
ruby-18/
def define_method_taking_block(name, &method_body_proc)
  self.class.send :define_method, "__quirky_bird_helper_#{name}__", method_body_p\
roc
  eval <<-EOM
    def #{name}(a_value, &a_proc)
      __quirky_bird_helper_#{name}__(a_value, a_proc)
    end
  EOM
end

def quirky_bird_extend(name, &value_proc)
  Object.send(:define_method, name) do
    value_proc.call(self)
  end
end
```

**quirky_songs.rb**

```
# The MIT License
#
# All contents Copyright (c) 2004-2008 Reginald Braithwaite
# <http://braythwayt.com>  except as otherwise noted.
#
# Permission is hereby granted, free of charge, to any person obtaining a copy
# of this software and associated documentation files (the "Software"), to deal
# in the Software without restriction, including without limitation the rights
# to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
# copies of the Software, and to permit persons to whom the Software is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in
# all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
# AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
# THE SOFTWARE.
#
# http://www.opensource.org/licenses/mit-license.php

require 'quirky_bird'
require 'blank_slate'
require 'returning'

quirky_bird_extend(:maybe) do |value|
  if value.nil?
    returning(BlankSlate.new) do |it|
      def it.method_missing(*args)
        nil
      end
    end
  else
    value
  end
end
```

```
quirky_bird_extend(:try) do |value|
  returning(BlankSlate.new) do |it|
    def it.__value__=(arg)
       @value = arg
    end
    def it.method_missing(name, *args)
      if @value.respond_to?(name)
        @value.send(name, *args)
      end
    end
    it.__value__ = value
  end
end
```

## 13.5 bluebirds

**before_and_after_advice.rb**

```
# This code contains ideas snarfed from:
#
#   http://github.com/up_the_irons/immutable/tree/master
#   http://blog.jayfields.com/2006/12/ruby-alias-method-alternative.html
#   http://eigenclass.org/hiki.rb?bounded+space+instance_exec
#
# And a heaping side of http://blog.grayproductions.net/articles/all_about_struct

module BeforeAndAfterAdvice

  # Random ID changed at each interpreter load
  UNIQ = "_#{object_id}"

  Compositions = Struct.new(:before, :between, :after)

  module MethodAdvice; end

  module ClassMethods

    # Example:
    #
```

```
    #   before :foo, :bar do
    #     # ...
    #   end
    #
    # This executes the body of the block before the #foo and #bar instance metho\
ds
    # for side effects without modifying the parameters (if any) passed to #foo
    # and #bar
    #
    #   before :fizz, :buzz do |p1, p2|
    #     # ...
    #     [p1, p2]
    #   end
    #
    # This executes the body of the block before the #foo and #bar instance metho\
ds
    # for side efects, AND determines what is passed along as parameters. If the \
block
    # takes parameters, it acts as a filter, transforming the parameters.
    #
    # It is possible to chain #before advice, and you can add more advice in subc\
lasses:
    #
    #   class Foo
    #     def foo(bar); end
    #   end
    #
    #   class Bar < Foo
    #     include MethodAdvice
    #
    #     before :foo do
    #       # ...
    #     end
    #
    #   end
    #
    #   class Blitz < Bar
    #     include MethodAdvice
    #
    #     before :foo do |bar|
    #       # ...
    #       bar
```

```
    #     end
    #
    #   end
    #
    def before(*method_symbols, &block)
      options = method_symbols[-1].kind_of?(Hash) ? method_symbols.pop : {}
      method_symbols.each do |method_sym|
        __composed_methods__[method_sym].before.unshift(__unbound_method__(block,\
 options[:name]))
        __rebuild_method__(method_sym)
      end
    end

    # Example:
    #
    #   after :foo, :bar do
    #     # ...
    #   end
    #
    # This executes the body of the block after the #foo and #bar instance methods
    # for side effects without modifying the return values of the #foo and #bar m\
ethods
    #
    #   after :fizz, :buzz do |r|
    #     # ...
    #     r
    #   end
    #
    # This executes the body of the block after the #foo and #bar instance methods
    # for side efects, AND determines what is returned from the call. If the block
    # takes parameters, it acts as a filter, transforming the return value.
    #
    # It is possible to chain #after advice, and you can add more advice in subcl\
asses:
    #
    #   class Foo
    #     def foo(bar); end
    #   end
    #
    #   class Bar < Foo
    #     include MethodAdvice
    #
```

```
    #     after :foo do
    #       # ...
    #     end
    #
    #   end
    #
    #   class Blitz < Bar
    #     include MethodAdvice
    #
    #     after :foo do |r|
    #       # ...
    #       r
    #     end
    #
    #   end
    #
    def after(*method_symbols, &block)
      options = method_symbols[-1].kind_of?(Hash) ? method_symbols.pop : {}
      method_symbols.each do |method_sym|
        __composed_methods__[method_sym].after.push(__unbound_method__(block, opt\
ions[:name]))
        __rebuild_method__(method_sym)
      end
    end

    # Removes all advice from the named methods. Intended for testing.
    #
    def reset_befores_and_afters(*method_symbols)
      method_symbols.each do |method_sym|
        __composed_methods__[method_sym].before = []
        __composed_methods__[method_sym].after = []
        __rebuild_method__(method_sym)
      end
    end

    # Modified to re-apply advice when a method is overridden. So:
    #
    #   class Foo
    #     def foo(bar); end
    #   end
    #
    #   class Bar < Foo
```

```
    #     include MethodAdvice
    #
    #     after :foo do
    #       # ...
    #     end
    #
    #   end
    #
    #   class Blitz < Bar
    #     include MethodAdvice
    #
    #     def foo(bar)
    #       # ...
    #     end
    #
    #   end
    #
    # In this case the class Blitz overrides the method #foo, but the advice in
    # class Bar is still applied, the override happens ONLY on the inner method,
    # not the advice.
    #
    # Note well that super has undefined behaviour in this situation.
    #
    def method_added(method_sym)
      unless instance_variable_get("@#{UNIQ}_in_method_added")
        __safely__ do
          __composed_methods__[method_sym].between = self.instance_method(method_\
sym)
          @old_method_added and @old_method_added.call(method_sym)
          __rebuild_method__(method_sym)
        end
      end
    end

    def __composed_methods__
      ancestral_composer = ancestors.detect { |ancestor| ancestor.instance_variab\
le_defined?(:@__composed_methods__) }
      if ancestral_composer
        ancestral_composer.instance_variable_get(:@__composed_methods__)
      else
        @__composed_methods__ ||= Hash.new { |hash, method_sym| hash[method_sym] \
= BeforeAndAfterAdvice::Compositions.new([], self.instance_method(method_sym), []\
```

```
) }
      end
    end

    def __rebuild_without_advice__(method_sym, old_method)
      if old_method.arity == 0
        define_method(method_sym) { old_method.bind(self).call }
      else
        define_method(method_sym) { |*params| old_method.bind(self).call(*params)\
 }
      end
    end

    def __rebuild_advising_no_parameters__(method_sym, old_method, befores, after\
s)
      define_method(method_sym) do
        befores.each do |before_advice_method|
          before_advice_method.bind(self).call
        end
        afters.inject(old_method.bind(self).call) do |ret_val, after_advice_metho\
d|
          after_advice_method.bind(self).call
        end
      end
    end

    def __rebuild_advising_with_parameters__(method_sym, old_method, befores, aft\
ers)
      define_method(method_sym) do |*params|
        afters.inject(
          old_method.bind(self).call(
            *befores.inject(params) do |acc_params, before_advice_method|
              if before_advice_method.arity == 0
                before_advice_method.bind(self).call
                acc_params
              else
                before_advice_method.bind(self).call(*acc_params)
              end
            end
          )
        ) do |ret_val, after_advice_method|
          if after_advice_method.arity == 0
```

```
            after_advice_method.bind(self).call
            ret_val
          else
            after_advice_method.bind(self).call(ret_val)
          end
        end
      end
    end

    def __rebuild_method__(method_sym)
      __safely__ do
        composition = __composed_methods__[method_sym]
        old_method = composition.between
        if composition.before.empty? and composition.after.empty?
          __rebuild_without_advice__(method_sym, old_method)
        else
          arity = old_method.arity
          if old_method.arity == 0
            __rebuild_advising_no_parameters__(method_sym, old_method, compositio\
n.before, composition.after)
          else
            __rebuild_advising_with_parameters__(method_sym, old_method, composit\
ion.before, composition.after)
          end
        end
      end
    end

    def __safely__
      was = instance_variable_get("@#{UNIQ}_in_method_added")
      begin
        instance_variable_set("@#{UNIQ}_in_method_added", true)
        yield
      ensure
        instance_variable_set("@#{UNIQ}_in_method_added", was)
      end
    end

    def __unbound_method__(a_proc, name_prefx = nil)
      begin
        old_critical, Thread.critical = Thread.critical, true
        n = 0
```

```
        n += 1 while respond_to?(mname="#{name_prefx || '__method_advice'}_#{n}")
        MethodAdvice.module_eval{ define_method(mname, &a_proc) }
      ensure
        Thread.critical = old_critical
      end
      begin
        MethodAdvice.instance_method(mname)
      ensure
        MethodAdvice.module_eval{ remove_method(mname) } unless name_prefx rescue\
 nil
      end
    end

  end

  def self.included(receiver)
    receiver.extend         ClassMethods
    receiver.send :include, MethodAdvice
    receiver.instance_variable_set("@#{UNIQ}_in_method_added", false)
    receiver.instance_variable_set(:@old_method_added, receiver.public_method_def\
ined?(:method_added) && receiver.instance_method(:method_added))
  end

end
```

# 14 About The Author

When he's not shipping Ruby, Javascript and Java applications scaling out to millions of users, Reg "Raganwald" Braithwaite has authored libraries[1] for Javascript and Ruby programming such as Katy, JQuery Combinators, YouAreDaChef, andand, and others.

He writes about programming on his "Homoiconic[2]" un-blog as well as general-purpose ruminations on his posterous space[3]. He is also known for authoring the popular raganwald programming blog[4] from 2005-2008.

## 14.1 contact

Twitter: @raganwald
Email: raganwald@gmail.com

[1] http://github.com/raganwald
[2] http://github.com/raganwald/homoiconic
[3] http://raganwald.posterous.com
[4] http://weblog.raganwald.com

**Reginald Braithwaite**



[5] http://creativecommons.org/licenses/by-sa/2.0/deed.en