Mike's World-O-Programming

Rants and articles about programming languages

The Y Combinator (Slight Return)

or:

How to Succeed at Recursion Without Really Recursing

> Tiger got to hunt,
> Bird got to fly;
> Lisper got to sit and wonder, (Y (Y Y))?
>
> Tiger got to sleep,
> Bird got to land;
> Lisper got to tell himself he understand.
>
> — Kurt Vonnegut, modified by Darius Bacon

Introduction

I recently wrote a blog post about the Y combinator. Since then, I've received so many useful comments that I thought it was appropriate to expand the post into a more complete article. This article will go into greater depth on the subject, but I hope it'll be more comprehensible as well. You don't need to have read the previous post to understand this one (in fact, it's probably better if you haven't.) The only background knowledge I require is a tiny knowledge of the Scheme programming language including recursion and first-class functions, which I will review. Comments are (again) welcome.

Why Y?

Before I get into the details of what Y actually is, I'd like to address the question of why you, as a programmer, should bother to learn about it. To be honest, there aren't a lot of good nuts-and-bolts practical reasons for learning about Y. Even though it does have a few practical applications, for the most part it's mainly of interest to computer language theorists. Nevertheless, I do think it's worth your while to know something about Y for the following reasons:

1. It's one of the most beautiful ideas in all of programming. If you have any sense of programming aesthetics, you're sure to be delighted by Y.
2. It shows in a very stark way how amazingly powerful the simple ideas of functional programming are.

In 1959, the British scientist C. P. Snow gave a famous lecture called **The Two Cultures** where he bemoaned the fact that many intelligent and well-educated people of the time had almost no knowledge of science. He used knowledge of the Second Law of Thermodynamics as a kind of dividing line between those who were scientifically literate and those who weren't. I think we can similarly use knowledge of the Y combinator as a dividing line between programmers who are "functionally literate" (i.e. have a reasonably deep knowledge of functional programming) and those who aren't. There are other topics that could serve just as well as Y (notably monads), but Y will do nicely. So if you aspire to have the True Lambda-Nature, read on.

By the way, **Paul Graham** (the Lisp hacker, Lisp book author, essayist, and now venture capitalist) apparently thinks so highly of Y that he named his startup incubator company **Y Combinator**. Paul got rich from his knowledge of ideas like these; maybe someone else will too. Maybe even you.

A puzzle

Factorials

We'll start our exploration of the Y combinator by defining some functions to compute factorials. The factorial of a non-negative integer `n` is the product of all integers starting from `1` and going up to and including `n`. Thus we have:

```
factorial 1 = 1
factorial 2 = 2 * 1 = 2
factorial 3 = 3 * 2 * 1 = 6
factorial 4 = 4 * 3 * 2 * 1 = 24
```

and so on. (I'm using a function notation without parentheses here, so `factorial 3` is the same as what is usually written as `factorial(3)`. Humor me.) Factorials increase very rapidly with increasing `n`; the factorial of `20` is

Recent Entries
Archive
Friends
Profile

AUGUST 2010

S	M	T	W	T	F	S
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31				

`2432902008176640000`. The factorial of `0` is defined to be `1`; this turns out to be the appropriate definition for the kinds of things factorials are actually used for (like solving problems in combinatorics).

Recursive definitions of the factorial function

It's easy to write a function in a programming language to compute factorials using some kind of a looping control construct like a `while` or `for` loop (e.g. in C or Java). However, it's also easy to write a recursive function to compute factorials, because factorials have a very natural recursive definition:

```
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

where the second line applies for all `n` greater than zero. In fact, in the computer language **Haskell**, that's the way you actually define the factorial function. In **Scheme**, the language we'll be using here, this function would be written like this:

```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))))
```

Scheme uses a parenthesized prefix notation for everything, so something like `(- n 1)` represents what is usually written `n - 1` in most programming languages. The reasons for this are beyond the scope of this article, but getting used to this notation isn't very hard.

In fact, the above definition of the factorial function in Scheme could also be written in a slightly more explicit way as follows:

```
(define factorial
  (lambda (n)
    (if (= n 0)
        1
        (* n (factorial (- n 1))))))
```

The keyword `lambda` simply indicates that the thing we're defining (i.e. whatever is enclosed by the open parenthesis to the immediate left of the `lambda` and its corresponding close parenthesis) is a function. What comes immediately after the word `lambda`, in parentheses, are the formal arguments of the function; here there is just one argument, which is `n`. The body of the function comes after the formal arguments, and here consists of the expression `(if (= n 0) 1 (* n (factorial (- n 1))))`. This kind of function is an anonymous function. Here you do give the anonymous function the name `factorial` after you've defined it, but you don't have to, and often it's handy not to if you're only going to be using it once. In Scheme and some other languages, anonymous functions are also called lambda expressions. Many programming languages besides Scheme allow you to define anonymous functions, including Python, Ruby, Javascript, Ocaml, and Haskell (but not C, C++, or Java, unfortunately). We'll be using lambda expressions a lot below.

In the Scheme language, the definition of `factorial` just given is identical to the one before it; Scheme simply translates the first definition into the second one before evaluating it. So all functions in Scheme are really lambda expressions.

Note that the body of the function has a call to the `factorial` function (which we're in the process of defining) inside it, which makes this a recursive definition. I will call this kind of definition, where the name of the function being defined is used in the body of the function, an explicitly recursive definition. (You might wonder what an "implicitly recursive" function would be. I'm not going to use that expression, but the notion I have in mind is a recursive function which is generated through non-recursive means — keep reading!)

For the sake of argument, we're going to assume that our version of Scheme doesn't have the equivalent of `for` or `while` loops in C or Java (although in fact, real Scheme implementations do have such constructs, but under a different name), so that in order to define a function like `factorial`, we pretty much have to use recursion. Scheme is often used as a teaching language partly for this reason: it forces students to learn to think recursively.

Functions as data and higher-order functions

Scheme is a cool language for many reasons, but one that is relevant to us here is that it allows you to use functions as "first class" data objects (this is often expressed by saying that Scheme supports first-class functions). This means that in Scheme, we can pass a function to another function as an argument, we can return a function as the result of evaluating another function applied to its arguments, and we can create functions on-the-fly as we need them (using the `lambda` notation shown above). This is the essence of functional programming, and it will feature prominently in the ensuing discussion. Functions which take other functions as arguments, and/or which return other functions as their results, are usually referred to as higher-order functions.

Eliminating explicit recursion

Now, here's the puzzle: what if you were asked to define the `factorial` function in Scheme, but were told that you could not use recursive function calls in the definition (for instance, in the `factorial` function given above you

cannot use the word `factorial` anywhere in the body of the function). However, you are allowed to use first-class functions and higher-order functions any way you see fit. With this knowledge, can you define the `factorial` function?

The answer to this question is yes, and it will lead us directly to the Y combinator.

What the Y combinator is and what it does

The Y combinator is a higher-order function. It takes a single argument, which is a function that isn't recursive. It returns a version of the function which is recursive. We will walk through this process of generating recursive functions from non-recursive ones using Y in great detail below, but that's the basic idea.

More generally, Y gives us a way to get recursion in a programming language that supports first-class functions but that doesn't have recursion built in to it. So what Y shows us is that such a language already allows us to define recursive functions, even though the language definition itself says nothing about recursion. This is a Beautiful Thing: it shows us that functional programming alone can allow us to do things that we would never expect to be able to do (and it's not the only example of this).

Lazy or strict evaluation?

We will be looking at two broad classes of computer languages: those that use lazy evaluation and those that use strict evaluation. Lazy evaluation means that in order to evaluate an expression in the language, you only evaluate as much of the expression as is needed to get the final result. So (for instance) if there is a part of the expression that doesn't need to get evaluated (because the result will not depend on it) it won't be evaluated. In contrast, strict evaluation means that all parts of an evaluation will be evaluated completely before the value of the expression as a whole is determined (with some necessary exceptions, such as `if` expressions, which have to be lazy to work properly). In practice, lazy evaluation is more general, but strict evaluation is more predictable and often more efficient. Most programming languages use strict evaluation. The programming language **Haskell** uses lazy evaluation, and this is one of the most interesting things about that language. We will use both kinds of evaluation in what follows.

One Y combinator or many?

Even though we often refer to Y as "the" Y combinator, in actual fact there are an infinite number of Y combinators. We will only be concerned with two of these, one lazy and one strict. We need two Y combinators because the Y combinator we define for lazy languages will not work for strict languages. The lazy Y combinator is often referred to as the normal-order Y combinator and the strict one is referred to as the applicative-order Y combinator. Basically, normal-order is another way of saying "lazy" and applicative-order is another way of saying "strict".

Static or dynamic typing?

Another big dividing line in programming languages is between static typing and dynamic typing. A statically-typed language is one where the types of all expressions are determined at compile time, and any type errors cause the compilation to fail. A dynamically-typed language doesn't do any type checking until run time, and if a function is applied to arguments of invalid types (e.g. by trying to add together an integer and a string), then an error is reported. Among commonly-used programming languages, C, C++ and Java are statically typed, and Perl, Python and Ruby are dynamically typed. Scheme (the language we'll be using for our examples) is also dynamically typed. (There are also languages that straddle the border between statically-typed and dynamically-typed, but I won't discuss this further.)

One often hears static typing referred to as strong typing and dynamic typing referred to as weak typing, but this is an abuse of terminology. Strong typing simply means that every value in the language has one and only one type, whereas weak typing means that some values can have multiple types. So Scheme, which is dynamically typed, is also strongly typed, while C, which is statically typed, is weakly typed (because you can cast a pointer to one kind of object into a pointer to another type of object without altering the pointer's value). I will only be concerned with strongly typed languages here.

It turns out to be much simpler to define the Y combinator in dynamically typed languages, so that's what I'll do. It is possible to define a Y combinator in many statically typed languages, but (at least in the examples I've seen) such definitions usually require some non-obvious type hackery, because the Y combinator itself doesn't have a straightforward static type. That's beyond the scope of this article, so I won't mention it further.

What a "combinator" is

A combinator is just a lambda expression with no free variables. We saw above what lambda expressions are (they're just anonymous functions), but what's a free variable? It's a variable (i.e. a name or identifier in the language) which isn't a bound variable. Happy now? No? OK, let me explain.

A bound variable is simply a variable which is contained inside the body of a lambda expression that has that variable name as one of its arguments.

Let's look at some examples of lambda expressions and free and bound variables:

1. `(lambda (x) x)`

2. `(lambda (x) y)`
3. `(lambda (x) (lambda (y) x))`
4. `(lambda (x) (lambda (y) (x y)))`
5. `(x (lambda (y) y))`
6. `((lambda (x) x) y)`

Are the variables in the body of these lambda expressions free variables or bound variables? We'll ignore the formal arguments of the lambda expressions, because only variables in the body of the lambda expression can be considered free or bound. As for the other variables, here are the answers:

1. The `x` in the body of the lambda expression is a bound variable, because the formal argument of the lambda expression is also `x`. This lambda expression has no other variables, therefore it has no free variables, therefore it's a combinator.
2. The `y` in the lambda body is a free variable. This lambda expression is therefore not a combinator.
3. Aside from the formal arguments of the lambda expression, there is only one variable, the final `x`, which is a bound variable (it's bound by the formal argument of the outer lambda expression). Therefore, this lambda expression as a whole has no free variables, so this is a combinator.
4. Aside from the formal arguments of the lambda expression, there are two variables, the final `x` and `y`, both bound variables. This is a combinator.
5. The entire expression is not a lambda expression, so it's by definition not a combinator. Nevertheless, the `x` is a free variable and the final `y` is a bound variable.
6. Again, the entire expression isn't a lambda expression (it's a function application), so this isn't a combinator either. The second `x` is a bound variable while the `y` is a free variable.

When you're wondering if a recursive function like `factorial`:

```
(define factorial
  (lambda (n)
    (if (= n 0)
        1
        (* n (factorial (- n 1))))))
```

is a combinator, you don't consider the `define` part, so what you're really asking is if

```
(lambda (n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))))
```

is a combinator. Since in this lambda expression, the name `factorial` represents a free variable (the name `factorial` is not a formal argument of the lambda expression), this is not a combinator. This will be important below. In fact, the names `=`, `*`, and `-` are also free variables, so even without the name `factorial` this would not be a combinator (to say nothing of the numbers!).

Back to the puzzle

Abstracting out the recursive function call

Recall the factorial function we had previously:

```
(define factorial
  (lambda (n)
    (if (= n 0)
        1
        (* n (factorial (- n 1))))))
```

What we want to do is to come up with a version of this that does the same thing but doesn't have that pesky recursive call to `factorial` in the body of the function.

Where do we start? It would be nice if you could save all of the function except for the offending recursive call, and put something else there. That might look like this:

```
(define sort-of-factorial
  (lambda (n)
    (if (= n 0)
        1
        (* n (<???> (- n 1))))))
```

This still leaves us with the problem of what to put in the place marked `<???>`. It's a tried-and-true principle of functional programming that if you don't know exactly what you want to put somewhere in a piece of code, just abstract it out and make it a parameter of a function. The easiest way to do this is as follows:

```
(define almost-factorial
  (lambda (f)
    (lambda (n)
      (if (= n 0)
          1
          (* n (f (- n 1)))))))
```

What we've done here is to rename the recursive call to `factorial` to `f`, and to make `f` an argument to a function which we're calling `almost-factorial`. Notice that `almost-factorial` is not at all the factorial function. Instead, it's a higher-order function which takes a single argument `f`, which had better be a function (or else `(f (- n 1))` won't make sense), and returns another function (the `(lambda (n) ...)` part) which (hopefully) will be a factorial function if we choose the right value for `f`.

It's important to realize that this trick is not in any way specific to the `factorial` function. We can do exactly the same trick with any recursive function. For instance, consider a recursive function to compute fibonacci numbers. The recursive definition of fibonacci numbers is as follows:

```
fibonacci 0 = 0
fibonacci 1 = 1
fibonacci n = fibonacci (n - 1) + fibonacci (n - 2)
```

(In fact, that's the definition of the fibonacci function in Haskell.) In Scheme, we can write the function this way:

```
(define fibonacci
  (lambda (n)
    (cond ((= n 0) 0)
          ((= n 1) 1)
          (else (+ (fibonacci (- n 1)) (fibonacci (- n 2)))))))
```

(where `cond` is just a shorthand expression for nested `if` expressions). We can then remove the explicit recursion just like we did for `factorial`:

```
(define almost-fibonacci
  (lambda (f)
    (lambda (n)
      (cond ((= n 0) 0)
            ((= n 1) 1)
            (else (+ (f (- n 1)) (f (- n 2))))))))
```

As you can see, the transformation from a recursive function to a non-recursive `almost-` equivalent function is a purely mechanical one: you rename the name of the recursive function inside the body of the function to `f` and you wrap a `(lambda (f) ...)` around the body.

If you've followed what I just did (never mind why I did it; we'll see that later), then congratulations! As Yoda says, you've just taken the first step into a larger world.

Sneak preview

I probably shouldn't do this yet, but I'm going to give you a sneak preview of where we're going. Once we define the Y combinator, we'll be able to define the factorial function using `almost-factorial` as follows:

```
(define factorial (Y almost-factorial))
```

where `Y` is the Y combinator. Note that this definition of `factorial` doesn't have any explicit recursion in it. Similarly, we can define the `fibonacci` function using `almost-fibonacci` in the same way:

```
(define fibonacci (Y almost-fibonacci))
```

So the Y combinator will give us recursion wherever we need it as long as we have the appropriate `almost-` function available (i.e. the non-recursive function derived from the recursive one by abstracting out the recursive function calls).

Read on to see what's really going on here and why this will work.

Recovering `factorial` from `almost-factorial`

Let's assume, for the sake of argument, that we already had a working factorial function lying around (recursive or not, we don't care). We'll call that hypothetical factorial function `factorialA`. Now let's consider the following:

```
(define factorialB (almost-factorial factorialA))
```

Question: does `factorialB` actually compute factorials?

To answer this, it's helpful to expand out the definition of `almost-factorial`:

```
(define factorialB
  ((lambda (f)
```

```
      (lambda (n)
        (if (= n 0)
            1
            (* n (f (- n 1))))))
   factorialA))
```

Now, by substituting `factorialA` for `f` inside the body of the lambda expression we get:

```
(define factorialB
  (lambda (n)
    (if (= n 0)
        1
        (* n (factorialA (- n 1))))))
```

This looks a lot like the recursive factorial function, but it isn't: `factorialA` is not the same function as `factorialB`. So it's a non-recursive function that depends on a hypothetical `factorialA` function to work. Does it actually work? Well, it's pretty obvious that it should work for `n = 0`, since `(factorialB 0)` will just return `1` (the factorial of `0`). If `n > 0`, then the value of `(factorialB n)` will be `(* n (factorialA (- n 1)))`. Now, we assumed that `factorialA` would correctly compute factorials, so `(factorialA (- n 1))` is the factorial of `n - 1`, and therefore `(* n (factorialA (- n 1)))` is the factorial of `n` (by the definition of factorial), thus proving that `factorialB` computes the factorial function correctly as long as `factorialA` does. So this works. The only problem is that we don't actually have a `factorialA` lying around.

Now, if you're really clever, you might be asking yourself whether we can just do this:

```
(define factorialA (almost-factorial factorialA))
```

The idea is this: let's assume that `factorialA` is a valid factorial function. Then if we pass it as an argument to `almost-factorial`, the resulting function will have to be a valid factorial function, so why not just name that function `factorialA`?

It looks like you've created a perpetual-motion machine (or perhaps I should say a perpetual-calculation machine), and there must be something wrong with this definition... mustn't there?

In fact, this definition will work fine as long as the Scheme language you're using uses lazy evaluation! Standard Scheme uses strict evaluation, so it won't work (it'll go into an infinite loop). If you use **DrScheme** as your Scheme interpreter (which you should), then you can use the "lazy Scheme" language level, and the above code will actually work (huzzah!). We'll see why below, but for now I want to stick to standard (strict) Scheme and approach the problem in a slightly different way.

Let's define a couple of functions:

```
(define identity (lambda (x) x))
(define factorial0 (almost-factorial identity))
```

The `identity` function is pretty simple: it takes in a single argument and returns it unchanged (it's also a combinator, as I hope you can tell). We're basically going to use it as a placeholder when we need to pass a function as an argument and we don't know what function we should pass.

`factorial0` is more interesting. It's a function that can compute some, but not all factorials. Specifically, it can compute the factorials up to and including the factorial of zero (which means that it can only compute the factorial of zero, but you'll soon see why I describe it this way). Let's verify that:

```
(factorial0 0)

==> ((almost-factorial identity) 0)

==> (((lambda (f)
        (lambda (n)
           (if (= n 0)
               1
               (* n (f (- n 1))))))
      identity)
     0)

==> ((lambda (n)
       (if (= n 0)
           1
           (* n (identity (- n 1)))))
     0)

==> (if (= 0 0)
```

```
        1
        (* 0 (identity (- 0 1))))

==> (if #t
        1
        (* 0 (identity (- 0 1))))

==> 1
```

OK, so it works. Unfortunately, it won't work for `n > 0`. For instance, if `n = 1` then we'll have (skipping a few obvious steps):

```
(factorial0 1)

==> (* 1 (identity (- 1 1)))

==> (* 1 (identity 0))

==> (* 1 0)

==> 0
```

which is not the correct answer.

Now consider this spiffed-up version of `factorial0`:

```
(define factorial1
  (almost-factorial factorial0))
```

which is the same thing as:

```
(define factorial1
  (almost-factorial
    (almost-factorial identity)))
```

This will correctly compute the factorials of `0` and `1`, but it will be incorrect for any `n > 1`. Let's verify this as well, again skipping some obvious steps:

```
(factorial1 0)

==> ((almost-factorial factorial0) 0)

==> 1 (via essentially the same derivation we showed above)

(factorial1 1)

==> ((almost-factorial factorial0) 1)

==> (((lambda (f)
        (lambda (n)
          (if (= n 0)
              1
              (* n (f (- n 1))))))
      factorial0)
     1)

==> ((lambda (n)
       (if (= n 0)
           1
           (* n (factorial0 (- n 1)))))
     1)

==> (if (= 1 0)
        1
        (* 1 (factorial0 (- 1 1))))

==> (if #f
        1
        (* 1 (factorial0 (- 1 1))))
```

```
==> (* 1 (factorial0 (- 1 1)))

==> (* 1 (factorial0 0))

==> (* 1 1)

==> 1
```

which is the correct answer. So `factorial1` can compute factorials for `n = 0` and `n = 1`. You can verify, though, that it won't be correct for `n > 1`.

We can keep going, and define functions which can compute factorials up to any particular limit:

```
(define factorial2 (almost-factorial factorial1))
(define factorial3 (almost-factorial factorial2))
(define factorial4 (almost-factorial factorial3))
(define factorial5 (almost-factorial factorial4))
etc.
```

`factorial2` will compute correct factorials for inputs between `0` and `2`, `factorial3` will compute correct factorials for inputs between `0` and `3`, and so on. You should be able to verify this for yourself using the above derivations as models, though you probably won't be able to do it in your head (at least, I can't do it in my head).

One interesting way of looking at this is that `almost-factorial` takes in a crappy factorial function and outputs a factorial function that is slightly less crappy, in that it will handle exactly one extra value of the input correctly.

Note that you can again rewrite the definitions of the factorial functions like this:

```
(define factorial0 (almost-factorial identity))

(define factorial1
  (almost-factorial
    (almost-factorial identity)))

(define factorial2
  (almost-factorial
    (almost-factorial
      (almost-factorial identity))))

(define factorial3
  (almost-factorial
    (almost-factorial
      (almost-factorial
        (almost-factorial identity)))))

(define factorial4
  (almost-factorial
    (almost-factorial
      (almost-factorial
        (almost-factorial
          (almost-factorial identity))))))

(define factorial5
  (almost-factorial
    (almost-factorial
      (almost-factorial
        (almost-factorial
          (almost-factorial
            (almost-factorial identity)))))))
```

and so on. Again, if you're very clever you might wonder if you could do this:

```
(define factorial-infinity
  (almost-factorial
    (almost-factorial
      (almost-factorial
        ...))))
```

where the `...` means that you're repeating the chain of `almost-factorials` an infinite number of times. If you did wonder this, go to the head of the class! Unfortunately, we can't write this out directly, but we can define the

equivalent of this. Note also that `factorial-infinity` is just the `factorial` function we want: it works on all integers greater than or equal to zero.

What we have shown is that if we could define an infinite chain of `almost-factorials`, that would give us the factorial function. Another way of saying this is that the factorial function is the fixpoint of `almost-factorial`, which is what I will explain next.

Fixpoints of functions

The notion of a fixpoint should be familiar to anyone who has amused themselves playing with a pocket calculator. You start with `0` and hit the `cos` (cosine) key repeatedly. What you find is that the answer rapidly converges to a number which is (approximately) `0.73908513321516067`; hitting the `cos` key again doesn't change anything because `cos(0.73908513321516067) = 0.73908513321516067`. We say that the number `0.73908513321516067` is a fixpoint of the cosine function.

The cosine function takes a single input value (a real number) and produces a single output value (also a real number). The fact that the input and output of the function are the same type is what allows you to apply it repeatedly, so that if `x` is a real number, we can calculate what `cos(x)` is, and since that will also be a real number, we can calculate what `cos(cos(x))` is, and then what `cos(cos(cos(x)))` is, and so on. The fixpoint is the value `x` where `cos(x) = x`.

Fixpoints don't have to be real numbers. In fact, they can be any type of thing, as long as the function that generates them can take the same type of thing as input as it produces as output. Most importantly for our discussion, fixpoints can be functions. If you have a higher-order function like `almost-factorial` that takes in a function as its input and produces a function as its output (with both input and output functions taking a single integer argument as input and producing a single integer as output), then it should be possible to compute its fixpoint (which will, naturally, be a function which takes a single integer argument as input and produces a single integer as output). That fixpoint function will be the function for which

```
fixpoint-function = (almost-factorial fixpoint-function)
```

By repeatedly substituting the right-hand side of that equation into the `fixpoint-function` on the right, we get:

```
fixpoint-function =
  (almost-factorial
    (almost-factorial fixpoint-function))

  = (almost-factorial
      (almost-factorial
        (almost-factorial fixpoint-function)))

  = ...

  = (almost-factorial
      (almost-factorial
        (almost-factorial
          (almost-factorial
            (almost-factorial ...)))))
```

As we saw above, this will be the factorial function we want. Thus, the fixpoint of `almost-factorial` will be the `factorial` function:

```
factorial = (almost-factorial factorial)
  = (almost-factorial
      (almost-factorial
        (almost-factorial
          (almost-factorial
            (almost-factorial ...)))))
```

That's all well and good, but just knowing that `factorial` is the fixpoint of `almost-factorial` doesn't tell us how to compute it. Wouldn't it be nice if there was some magical higher-order function that would take as its input a function like `almost-factorial`, and would output its fixpoint function, which in that case would be `factorial`? Wouldn't that be really freakin' sweet?

That function exists, and it's the Y combinator. Y is also known as the fixpoint combinator: it takes in a function and returns its fixpoint.

Eliminating (most) explicit recursion (lazy version)

OK, it's time to derive Y.

Let's start by specifying what Y does:

```
(Y f) = fixpoint-of-f
```

What do we know about the fixpoint of `f`? We know that

```
(f fixpoint-of-f) = fixpoint-of-f
```

by the definition of what a fixpoint of a function is. Therefore, we have:

```
(Y f) = fixpoint-of-f = (f fixpoint-of-f)
```

and we can substitute `(Y f)` for `fixpoint-of-f` to get:

```
(Y f) = (f (Y f))
```

Voila! We've just defined Y. If we want it to be expressed as a Scheme function, we would have to write it like this:

```
(define (Y f) (f (Y f)))
```

or, using an explicit `lambda` expression, as:

```
(define Y
  (lambda (f)
    (f (Y f))))
```

However, there are two caveats regarding this definition of Y:

1. It will only work in a lazy language (see below).
2. It is not a combinator, because the `Y` in the body of the definition is a free variable which is only bound once the definition is complete. In other words, we couldn't just take the body of this version of `Y` and plop it in wherever we needed it, because it requires that the name `Y` be defined somewhere.

Nevertheless, if you're using lazy Scheme, you can indeed define factorials like this:

```
(define Y
  (lambda (f)
    (f (Y f))))

(define almost-factorial
  (lambda (f)
    (lambda (n)
      (if (= n 0)
          1
          (* n (f (- n 1)))))))

(define factorial (Y almost-factorial))
```

and it will work correctly.

What have we accomplished? We originally wanted to be able to define the factorial function without using any explicitly recursive functions at all. We've almost done that. Our definition of `Y` is still explicitly recursive. However, we've taken a giant step, because this is the only function in our language that needs to be explicitly recursive in order to define recursive functions. With this version of `Y` we can go ahead and define other recursive functions (for instance, defining `fibonacci` as `(Y almost-fibonacci)`).

Eliminating (most) explicit recursion (strict version)

I said above that the definition of Y that we derived wouldn't work in a strict language (like standard Scheme). In a strict language, we evaluate all the arguments to a function call before applying the function to its arguments, whether or not those arguments are needed. So if we have a function `f` and we try to evaluate `(Y f)` using the above definition, we get:

```
(Y f)
= (f (Y f))
= (f (f (Y f)))
= (f (f (f (Y f))))
etc.
```

and so on ad infinitum. The evaluation of `(Y f)` will never terminate, so we will never get a usable function out of it. This definition of Y doesn't work for strict languages.

However, there is a clever hack that we can use to save the day and define a version of Y that works in strict languages. The trick is to realize that `(Y f)` is going to become a function of one argument. Therefore, this equality will hold:

```
(Y f) = (lambda (x) ((Y f) x))
```

Whatever one-argument function `(Y f)` is, `(lambda (x) ((Y f) x))` has to be the same function. All you're doing is taking in a single input value `x` and giving it to the function defined by `(Y f)`. In a similar way, this will be true:

```
cos = (lambda (x) (cos x))
```

It doesn't matter whether you use `cos` or `(lambda (x) (cos x))` as your cosine function; they will both do the same thing.

However, it turns out that `(lambda (x) ((Y f) x))` has a big advantage when defining Y in a strict language. By the reasoning given above, we should be able to define Y as follows:

```
(define Y
  (lambda (f)
    (f (lambda (x) ((Y f) x)))))
```

Since we know that `(lambda (x) ((Y f) x))` is the same function as `(Y f)`, this is a valid version of Y which will work just as well as the previous version, even though it's a bit more complicated (and perhaps a tiny bit slower in practice). We could use this version of Y to define the `factorial` function in lazy Scheme, and it would work fine.

The cool thing about this version of Y is that it will also work in a strict language (like standard Scheme)! The reason for this is that when you give Y a particular `f` to find the fixpoint of, it will return

```
(Y f) = (f (lambda (x) ((Y f) x)))
```

This time, there is no infinite loop, because the inner `(Y f)` is kept inside a `lambda` expression, where it sits until it's needed (since the body of a lambda expression is never evaluated in Scheme until the lambda expression is applied to its arguments). Basically, you're using the lambda to delay the evaluation of `(Y f)`. So if `f` was `almost-factorial`, we would have this:

```
(define almost-factorial
  (lambda (f)
    (lambda (n)
      (if (= n 0)
          1
          (* n (f (- n 1)))))))

(define factorial (Y almost-factorial))
```

Expanding out the call to Y, we have:

```
(define factorial
  ((lambda (f) (f (lambda (x) ((Y f) x))))
   almost-factorial))

==>

(define factorial
  (almost-factorial (lambda (x) ((Y almost-factorial) x))))

==>

(define factorial
  (lambda (n)
    (if (= n 0)
        1
        (* n ((lambda (x) ((Y almost-factorial) x)) (- n 1))))))
```

Here again, `(lambda (x) ((Y almost-factorial) x))` is the same function as `(Y almost-factorial)`, which is the fixpoint of `almost-factorial`, which is just the factorial function. However, the `(Y almost-factorial)` in `(lambda (x) ((Y almost-factorial) x))` won't be evaluated until the entire lambda expression is applied to its argument, which won't happen until later (or not at all, for the factorial of zero). Therefore this factorial function will work in a strict language, and the version of Y used to define it will also work in a strict language.

I realize that the preceding discussion and derivation is nontrivial, so don't be discouraged if you don't get it right away. Just sleep on it, play with it in your mind and with your trusty DrScheme interpreter, and you'll eventually get it.

At this point, we've accomplished everything we've set out to accomplish, except for one tiny little detail: we haven't yet derived the Y combinator itself.

Deriving the Y combinator

The lazy (normal-order) Y combinator

At this point, we want to define not just Y, but a Y combinator. Note that the previous (lazy) definition of Y:

```
(define Y
  (lambda (f)
    (f (Y f))))
```

is a valid definition of Y but is not a Y combinator, since the definition of Y refers to Y itself. In other words, this definition is explicitly recursive. A combinator isn't allowed to be explicitly recursive; it has to be a lambda expression with no free variables (as I mentioned above), which means that it can't refer to its own name (if it even has a name) in its definition. If it did, the name would be a free variable in the definition, as we have in our definition of Y:

```
(lambda (f)
  (f (Y f)))
```

Note that Y in this definition is free; it isn't the bound variable of any lambda expression. So this is not a combinator.

Another way to think about this is that you should be able to replace the name of a combinator with its definition everywhere it's found and have everything still work. (Can you see why this wouldn't work with the explicitly recursive definition of Y? You would get into an infinite loop and you'd never be able to replace all the Ys with their definitions.) So whatever the Y combinator will be, it will not be explicitly recursive. From this non-recursive function we will be able to define whatever recursive functions we want.

I'm going to go back a bit to our original problem and derive a Y combinator from the bottom up. After I've done that I'll check to make sure that it is a fixpoint combinator, like the versions of Y we've already seen. In what follows I will borrow (steal) liberally from a very elegant derivation of the Y combinator sent to me by Eli Barzilay (thanks, Eli!), who is one of the DrScheme developers and an all-around Scheme uberstud.

Recall our original recursive `factorial` function:

```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))))
```

Recall that we want to define a version of this without the explicit recursion. One way we could do this is to pass the factorial function itself as an extra argument when you call the function:

```
;; This won't work yet:
(define (part-factorial self n)
  (if (= n 0)
      1
      (* n (self (- n 1)))))
```

Note that `part-factorial` is not the same as the `almost-factorial` function described above. We would have to call this `part-factorial` function in a different way to get it to compute factorials:

```
(part-factorial part-factorial 5) ==> 120
```

This is not explicitly recursive because we send along an extra copy of the `part-factorial` function as the `self` argument. However, it won't work unless the point of recursion calls the function the exact same way:

```
(define (part-factorial self n)
  (if (= n 0)
      1
      (* n (self self (- n 1)))))  ;; note the extra "self" here


(part-factorial part-factorial 5) ==> 120
```

This works, but now we have moved away from our original way of calling the factorial function. We can move back to something closer to our original version by rewriting it like this:

```
(define (part-factorial self)
  (lambda (n)
    (if (= n 0)


        1
        (* n ((self self) (- n 1))))))


((part-factorial part-factorial) 5) ==> 120
(define factorial (part-factorial part-factorial))
(factorial 5) ==> 120
```

Pause for a second here. Notice that we've already defined a version of the factorial function without using explicit recursion anywhere! This is the most crucial step. Everything else we do will be concerned with packaging what

we've already done so that we can easily re-use it with other functions.

Now let's try to get back something like our `almost-factorial` function by pulling out the `(self self)` call using a `let` expression outside of a `lambda`:

```
(define (part-factorial self)
  (let ((f (self self)))
    (lambda (n)
      (if (= n 0)
          1
          (* n (f (- n 1)))))))

(define factorial (part-factorial part-factorial))
(factorial 5) ==> 120
```

This will work fine in a lazy language. In a strict language, the `(self self)` call in the `let` statement will send us into an infinite loop, because in order to calculate `(part-factorial part-factorial)` (in the definition of `factorial`) you will first have to calculate `(part-factorial part-factorial)` (in the `let` expression). (For fun: figure out why this wasn't a problem with the previous definition.) I'll let this go for now, because I want to define the lazy Y combinator, but in the next section I'll solve this problem in the same way we solved it before (by wrapping a `lambda` around the `(self self)` call). Note that in a lazy language, the `(self self)` call in the `let` statement will never be evaluated unless `f` is actually needed (for instance, if `n = 0` then `f` isn't needed to compute the answer, so `(self self)` won't be evaluated). Understanding how lazy languages evaluate expressions is not trivial, so don't worry if you find this a little confusing. I recommend you experiment with the code using the lazy Scheme language level of DrScheme to get a better feel for what's going on.

It turns out that any `let` expression can be converted into an equivalent `lambda` expression using this equation:

```
(let ((x <expr1>)) <expr2>)
==> ((lambda (x) <expr2>) <expr1>)
```

where `<expr1>` and `<expr2>` are arbitrary Scheme expressions. (I'm only considering `let` expressions with a single binding and `lambda` expressions with a single argument, but the principle can easily be generalized to `lets` with multiple bindings and `lambdas` with multiple arguments.) This leads us to:

```
(define (part-factorial self)
  ((lambda (f)
     (lambda (n)
       (if (= n 0)
           1
           (* n (f (- n 1))))))
   (self self)))

(define factorial (part-factorial part-factorial))
(factorial 5) ==> 120
```

If you look closely, you'll see that we have our old friend the `almost-factorial` function embedded inside the `part-factorial` function. Let's pull it outside:

```
(define almost-factorial
  (lambda (f)
    (lambda (n)
      (if (= n 0)
          1
          (* n (f (- n 1)))))))

(define (part-factorial self)
  (almost-factorial
    (self self)))

(define factorial (part-factorial part-factorial))
(factorial 5) ==> 120
```

I don't know about you, but I'm getting pretty fed up with this whole `(part-factorial part-factorial)` thing, and I'm not going to take it anymore! Fortunately, I don't have to; I can first rewrite the `part-factorial` function like this:

```
(define part-factorial
  (lambda (self)
    (almost-factorial
      (self self))))
```

Then I can rewrite the `factorial` function like this:

```
(define almost-factorial
  (lambda (f)
    (lambda (n)
      (if (= n 0)
          1
          (* n (f (- n 1)))))))

(define factorial
  (let ((part-factorial (lambda (self)
                          (almost-factorial
                            (self self)))))
    (part-factorial part-factorial)))

(factorial 5) ==> 120
```

The `factorial` function can be written a little more concisely by changing the name of `part-factorial` to `x` (since we aren't using this name anywhere else now):

```
(define factorial
  (let ((x (lambda (self)
             (almost-factorial (self self)))))
    (x x)))
```

Now let's use the same `let ==> lambda` trick we used above to get:

```
(define almost-factorial
  (lambda (f)
    (lambda (n)
      (if (= n 0)
          1
          (* n (f (- n 1)))))))

(define factorial
  ((lambda (x) (x x))
   (lambda (self)
     (almost-factorial (self self)))))

(factorial 5) ==> 120
```

And again, to make this definition a little more concise, we can rename `self` to `x` to get:

```
(define almost-factorial
  (lambda (f)
    (lambda (n)
      (if (= n 0)
          1
          (* n (f (- n 1)))))))

(define factorial
  ((lambda (x) (x x))
   (lambda (x)
     (almost-factorial (x x)))))

(factorial 5) ==> 120
```

Note that the two `lambda` expressions in the definition of `factorial` both are functions of `x`, but the two `x`'s don't conflict with each other. In fact, we could have renamed `self` to `y` or almost any other name, but it'll be convenient to use `x` in what follows.

We're almost there! This works fine, but it's too specific to the `factorial` function. Let's change it to a generic `make-recursive` function that makes recursive functions from non-recursive ones (sound familiar?):

```
(define almost-factorial
  (lambda (f)
    (lambda (n)
      (if (= n 0)
          1
          (* n (f (- n 1)))))))
```

```
(define (make-recursive f)
  ((lambda (x) (x x))
   (lambda (x) (f (x x)))))

(define factorial (make-recursive almost-factorial))

(factorial 5) ==> 120
```

The `make-recursive` function is in fact the long-sought lazy Y combinator, also known as the normal-order Y combinator, so let's write it that way:

```
(define almost-factorial
  (lambda (f)
    (lambda (n)
      (if (= n 0)
          1
          (* n (f (- n 1)))))))

(define (Y f)
  ((lambda (x) (x x))
   (lambda (x) (f (x x)))))

(define factorial (Y almost-factorial))
```

I'm going to expand out the definition of Y a little bit:

```
(define Y
  (lambda (f)
    ((lambda (x) (x x))
     (lambda (x) (f (x x))))))
```

Note that we can apply the inner `lambda` expression to its argument to get an equivalent version of Y:

```
(define Y
  (lambda (f)
    ((lambda (x) (f (x x)))
     (lambda (x) (f (x x))))))
```

What this means is that, for a given function `f` (which is a non-recursive function like `almost-factorial`), the corresponding recursive function can be obtained first by computing `(lambda (x) (f (x x)))`, and then applying this `lambda` expression to itself. This is the usual definition of the normal-order Y combinator.

The only thing left to do is to check that this Y combinator is a fixpoint combinator (which it has to be in order to compute the right thing). To do this we have to demonstrate that this equation is correct:

```
(Y f) = (f (Y f))
```

From the definition of the normal-order Y combinator given above, we have:

```
(Y f)

= ((lambda (x) (f (x x)))
   (lambda (x) (f (x x))))
```

Now apply the first lambda expression to its argument, which is the second lambda expression, to get this:

```
= (f ((lambda (x) (f (x x)))
      (lambda (x) (f (x x)))))

= (f (Y f))
```

as desired. So, not only is the normal-order Y combinator also a fixpoint combinator, it's just about the most obvious fixpoint combinator there is, in that the proof that it's a fixpoint combinator is so trivial.

If you've made it through all of this derivation, you should pat yourself on the back and take a well-deserved break. When you come back, we'll finish off by deriving...

The strict (applicative-order) Y combinator

Let's pick up the previous derivation just before the point where it failed for strict languages:

```
(define (part-factorial self)
  (lambda (n)
    (if (= n 0)
        1
```

```
        (* n ((self self) (- n 1))))))

((part-factorial part-factorial) 5) ==> 120
(define factorial (part-factorial part-factorial))
(factorial 5) ==> 120
```

Up to this point, everything works in a strict language. Now if we pull the `(self self)` out into a `let` expression as before, we have:

```
(define (part-factorial self)
  (let ((f (self self)))
    (lambda (n)
      (if (= n 0)
          1
          (* n (f (- n 1)))))))

(define factorial (part-factorial part-factorial))
(factorial 5) ==> 120
```

As I said above, this will not work in a strict language, because whenever the `factorial` function is called it will evaluate the function call `(part-factorial part-factorial)`, and when that function call is evaluated it will first evaluate `(self self)` as part of the `let` expression, which in this case will be `(part-factorial part-factorial)`, leading to an infinite loop of `(part-factorial part-factorial)` calls.

We saw above that the way around problems like this is to realize that what we are trying to evaluate are functions of one argument. In this case, `(self self)` will be a function of one argument (it's going to be the same as `(part-factorial part-factorial)`, which is just the `factorial` function). We can wrap a lambda expression around this function to get an equivalent function:

```
(define (part-factorial self)
  (let ((f (lambda (y) ((self self) y))))
    (lambda (n)
      (if (= n 0)
          1
          (* n (f (- n 1)))))))
```

All we've done here is convert `(self self)`, a function of one argument, to `(lambda (y) ((self self) y))`, an equivalent function of one argument (we saw this trick earlier). I'm using `y` instead of `x` as the variable binding of the new lambda expression so as not to cause name conflicts later on in the derivation when `self` gets renamed to `x`, but I could have chosen another name as well.

After we've done this, the `part-factorial` function will now work even in a strict language. That's because once `(part-factorial part-factorial)` is evaluated, as part of evaluating the `let` expression the code `(lambda (x) ((self self) x))` will be evaluated. Unlike before, this will not send us into an infinite loop; the lambda expression won't be evaluated further until it's applied to its argument. This lambda wrapper doesn't change the value of the thing it wraps, but it does delay its evaluation, which is all we need to get the definition of `part-factorial` to work in a strict language.

And that's the trick. After that, we carry through every other step of the derivation in exactly the same way. We end up with this definition of the strict Y combinator:

```
(define Y
  (lambda (f)
    ((lambda (x) (f (lambda (y) ((x x) y))))
     (lambda (x) (f (lambda (y) ((x x) y)))))))
```

This can also be written in the equivalent form:

```
(define Y
  (lambda (f)
    ((lambda (x) (x x))
     (lambda (x) (f (lambda (y) ((x x) y)))))))
```

Hopefully, you can see why this is equivalent. Either of these are the strict Y combinator, or as it's called in the technical literature, the applicative-order Y combinator. In a strict language (like standard Scheme) you can use this to define the factorial function in the usual way:

```
(define factorial (Y almost-factorial))
```

I recommend you try this out with DrScheme, and lo! marvel at the awesome power of the applicative-order Y combinator, that which hath created recursion where no recursion hath previously existed.

Other matters

Practical applications

This article has (I hope) convinced you that you don't need to have explicit recursion built in to a language in order for that language to allow you to define recursive functions, as long as the language supports first-class functions so that you can define a Y combinator. However, I don't want to leave you with the notion that recursion in real computer languages is implemented this way. In practice, it's far more efficient to just implement recursion directly in a computer language than to use the Y combinator. There are lots of other interesting issues that come up when considering how to implement recursion efficiently, but those issues are beyond the scope of this article. The point is that implementing recursion using the Y combinator is mainly of theoretical interest.

That said, in the paper **Y in Practical Programs**, Bruce McAdams discusses a few ways in which Y can be used to define variants of recursive functions that e.g. print traces of their execution or automatically memoize their execution to give greater efficiency (as well as a few more esoteric applications), so Y isn't just a theoretical construct.

Mutual Recursion

Experienced functional programmers and/or particularly astute readers may have noticed that I didn't describe how to use the Y combinator to implement mutual recursion, which is where you have two or more functions which all call each other. The simplest example I can think of to illustrate mutual recursion are the following pair of functions which determine whether a non-negative integer is even or odd:

```
(define (even? n)
  (if (= n 0)
      #t
      (odd? (- n 1))))

(define (odd? n)
  (if (= n 0)
      #f
      (even? (- n 1))))
```

Before you start yelling at me, yes, I know that this isn't the most efficient way to compute evenness or oddness — it's just to illustrate what mutual recursion is. Any computer language that supports recursive function definitions has to support mutual recursion as well, but I haven't shown you how to use Y to define mutually-recursive functions. I'm going to cop out here because I think this article is long enough as it is, but rest assured that it is possible to define analogs of Y that can define mutually-recursive functions.

Further reading

- The Wikipedia article on the **Y combinator** is somewhat difficult reading, but it has some interesting material I didn't cover here.
- **The Little Schemer**, 4th. ed., by Dan Friedman and Matthias Felleisen. Chapter 9 has a derivation of the Y combinator which is what got me interested in this subject.
- The article **Y in Practical Programs**, by Bruce McAdams, which was referred to in the previous section.

Acknowledgments

I would like to thank the following people:

- Everyone who commented on my first blog post on the Y combinator, and also everyone who comments on this article.
- Eli Barzilay, for a very interesting email discussion on this subject. The derivation of the normal-order Y combinator is taken directly from Eli (with permission).
- My friend Darius Bacon for the poem. I'd also like to apologize to the estate of Kurt Vonnegut for abusing his work. The original poem appeared in Vonnegut's brilliant novel **Cat's Cradle**. If you haven't read it, you should do so as soon as possible.
- All the **DrScheme** implementors for giving me a terrific tool with which to explore this subject.
- The authors of the book **The Little Schemer**, Dan Friedman and Matthias Felleisen. This article is (in my mind at least) a massive expansion of chapter 9 of their book.
