

Latest: [Writing CEK-style interpreters in Haskell](#)

Next: [Non-termination without loops, iteration or recursion in Javascript](#)

Prev: [HOWTO: Fix mold and allergy problems](#)

Rand: [First-class macros from meta-circular evaluators](#)

Fixed-point combinators in JavaScript: Memoizing recursive functions

[\[article index\]](#) [\[email me\]](#) [\[@mattmight\]](#) [\[+mattmight\]](#) [\[rss\]](#)

It comes as a surprise to many programmers that it is possible to express a "recursive" function like factorial without using recursion or iteration.

The technique involved is subtle but powerful: the recursive function is computed as the "fixed point" of a non-recursive function. To compute the fixed point, we can use the **Y combinator**, which is itself a non-recursive function that computes fixed points.

That this manages to work is truly remarkable.

--Sussman and Steele on the Y Combinator

As a practical application of this theory, recursive functions expressed as fixed points allow the use of a memoizing fixed-point combinator. The combinator approach to recursion makes it possible to cache the internal calls to a recursive function automatically.

For example, this caching turns the naive, exponential implementation of Fibonacci into the optimized, linear-time version *for free*.

Read below to see how to do this all of this in JavaScript, courtesy of its anonymous function construct.



Get hassle-free redistribution rights
for ActivePython with OEM licensing.
Watch this webinar and find out how to save development
time and get to market faster.



If you like the article below, you might also enjoy:

- [My recommended reading](#) for programming languages.
- [What every CS major should know](#).
- [JavaScript: The Good Parts](#). Exactly that.
- [JavaScript: The Definitive Guide](#). *The* reference on JavaScript.

Recursion as fixed points

Students of algebra are already familiar with recursion and fixed points.

They just don't realize it.

Consider an equation like " $x = x^2 - 2$." (Programmers might recognize this as a recursive definition, in which x is being defined in terms of itself.)

If asked to solve for the value of x , a student might re-arrange the equation to use the quadratic formula. However, there is another way to express, and even find, the value(s) of x : fixed points.

A **fixed point** of a function f is an input that is equal to its output; that is x is a fixed point of the function f if $x = f(x)$. Some functions have no fixed points; some have many. The notation $\text{Fix}(f)$ denotes the set of fixed points of a function f .

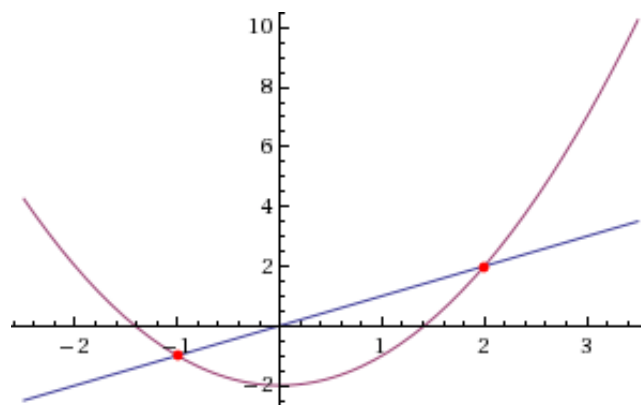
Define the function f such that $f(v) = v^2 - 2$. Then, observe that the original equation may now be re-written as " $x = f(x)$." In other words, the solutions to the equation are the fixed points of the function f ! That is, $\text{Fix}(f) = \{-1, 2\}$ - a fact we can verify by seeing that

$$f(-1) = (-1)^2 - 2 = 1 - 2 = -1,$$

and:

$$f(2) = (2)^2 - 2 = 4 - 2 = 2$$

or by graphing $y = x$ and $y = f(x)$:



These are exactly the [solutions to \$x = x^2 - 2\$](#) given by Wolfram Alpha.

The insight that powers the upcoming technique is the observation that any

time we have a recursive definition of the form " $x = f(x)$," the meaning of x is going to be defined in terms of fixed points.

The trick is to find a way to obtain fixed points when the equation has the form " $f = F(f)$," in which the value of f is not a number, but a function.

The Y combinator is that trick.

The Y combinator in theory

In his research on the [λ-calculus](#) and [combinatory logic](#), Haskell Curry discovered the "paradoxical" [fixed-point combinator](#) known as the Y combinator. The Y combinator takes a functional as input, and it returns the (unique) fixed point of that functional as its output. A **functional** is a function that takes a function for its input. Therefore, the fixed point of a functional is going to be a *function*.

Using the concepts of functionals and fixed points, we can eliminate explicit recursion for a function through two steps:

1. Find a functional whose fixed point is the recursive function we seek.
2. Find the fixed point of a functional *without recursion*.

A simple source transformation takes care of the first step. The Y combinator takes care of the second.

Deriving the Y combinator

The λ-calculus, the language in which the Y combinator is typically expressed, is a programming language which contains only anonymous functions, function applications and variable references. (Remarkably, this language is Turing-complete.) The notation $\lambda v.e$ stands for the function that maps the input v to the output e . JavaScript supports anonymous functions:

```
 $\lambda v.e$  == function (v) { return e ; }
```

So, if we can find a way to express the Y combinator in the λ-calculus, we can express it in JavaScript too.

To derive the Y combinator, start with the core property we seek. Namely, if we give the Y combinator a functional F , then $Y(F)$ needs to be a fixed point:

$$Y(F) = F(Y(F))$$

We could actually transliterate this definition into JavaScript as:

```
function Y(F) { return F(Y(F)) ; }
```

Of course, if we tried to use it, it would never work because the function Y

immediately calls itself, leading to infinite recursion.

Using a little λ -calculus, however, we can wrap the call to Y in a λ -term:

$$Y(F) = F(\lambda x.(Y(F))(x))$$

Now, when we invoke the the function Y , it immediately calls the function F , and passes it $\lambda x.(Y(F))(x)$, which is equivalent to the fixed point.

Or, in JavaScript:

```
function Y(F) { return F(function (x) { return (Y(F))(x) ; } ) ; }
```

This function will actually find the fixed point of a functional, and we could use it to eliminate recursion. Of course, as defined, the function Y calls itself recursively, so we haven't really eliminated recursion yet. We've just moved it all into the function Y .

Using another construct called the [U combinator](#), we can eliminate the recursive call inside the Y combinator, which, with a couple more transformations gets us to:

$$Y = (\lambda h.\lambda F.F(\lambda x.((h(h))(F))(x))) (\lambda h.\lambda F.F(\lambda x.((h(h))(F))(x)))$$

Note that the right-hand-side makes no reference to Y .

The Y combinator in JavaScript

Any untyped language which permits lexically scoped anonymous functions, such as JavaScript, can express the Y combinator without relying on recursion, iteration or side effects. Even without understanding *how* the Y combinator works, you still can see it in action and verify for yourself that no recursion or iteration is used. The following example expresses the factorial function without using recursion:

```
// A "functional" is just a function that takes
// another function as input.

// The Y combinator finds the fixed point
// of the "functional" passed in as an argument.

// Thus, the Y combinator satisfies the property:

//   Y(F) = F(Y(F))

// Note that Y does not reference itself:

var Y = function (F) {
  return function (x) {
    return F(function (y) { return (x(x))(y);});
  }
  (function (x) {
    return F(function (y) { return (x(x))(y);});
  });
};

// (In fact, all functions above are anonymous!)

// FactGen is the functional whose fixed point is
// factorial.

// That is, if you pass the factorial function to
// FactGen, you get back the factorial function.

// Since the Y combinator returns the fixed point
// of a functional, applying the Y combinator to
// FactGen returns the factorial function!

// Note that FactGen does not reference itself:

var FactGen = function (fact) {
  return function(n) {
    return ((n == 0) ? 1 : (n*fact(n-1))) ;
  };
};
```

Evaluate: [result1]

Take a close look at the definition of Y. It uses only three kinds of expression: anonymous functions, variable reference and function application. Each anonymous function has the form **function** (*argument*) { **return** *expression* ; }. The Y combinator is a closed expression--it makes no explicit reference to an outside variable or to itself. Clearly, there is no recursion, iteration or mutation.

The Y combinator allows a concise transformation from a recursive function to a non-recursive function. If we have a recursive function *f*:

```
function f (arg) {
  ... f ...
```

```
}
```

This definition can be transformed into a non-recursive form:

```
var f = Y(function(g) { return (function (arg) {  
  ... g ...  
}) ;}) ;
```

It is inspiring to see what you can achieve in just a few extra characters.

Exploiting the Y combinator

The Y combinator is a significant result in the theory of computation and the theory of programming languages. It offers another way to think about nontrivial functions in terms of fixed points, rather than the standard paradigms of recursion and iteration.

For instance, suppose we define a recursive function using the functional-fixed-point paradigm: can we then create a fixed-point combinator that automatically gives us better performance for the function? The answer is *yes*. We can create a **memoizing** fixed-point combinator: a Y-like combinator that caches the results of intermediate function calls.

For example, the naive way of defining Fibonacci using recursion makes two recursive calls, leading to exponential time complexity:

```
function fib(n) {  
  if (n == 0) return 0 ;  
  if (n == 1) return 1 ;  
  return fib(n-1) + fib(n-2) ;  
}
```

We could however, define Fibonacci using the Y combinator:

```
var fib = Y(function (g) { return (function (n) {  
  if (n == 0) return 0 ;  
  if (n == 1) return 1 ;  
  return g(n-1) + g(n-2) ;  
}) ; }) ;
```

This formulation still has exponential complexity, but we can change it to linear time *just by changing the fixed-point combinator*. The memoizing Y combinator, Y_{mem} , keeps a cache of computed results, and returns the pre-computed result if available:

```
// Ymem takes a functional and an (optional)
// cache of answers.

// It returns the fixed point of the functional
// that caches intermediate results.

function Ymem(F, cache) {
  if (!cache)
    cache = {}; // Create a new cache.
  return function(arg) {
    if (cache[arg])
      return cache[arg] ; // Answer in cache.
    var answer = (F(function(n){
      return (Ymem(F,cache))(n);
    }))(arg) ; // Compute the answer.
    cache[arg] = answer ; // Cache the answer.
    return answer ;
  } ;
}

var fib = Ymem(function (g) { return function (n) {
  if (n == 0) return 0 ;
  if (n == 1) return 1 ;
  return g(n-1) + g(n-2) ;
}}) ;
```

Evaluate: [result2]

There are a couple caveats with this particular Ymem:

1. Ymem only works on functions of one argument, but this could be remedied with Javascript's `apply` method and the use of a [trie](#)-like cache.
2. Ymem only works for indexable argument values like numbers and strings, but this can be circumvented by supplying a comparator on argument values, so that it can use a sorted tree for the cache.

The end result is that the 100th Fibonacci number is computed instantly, whereas the naive version (or the version using the ordinary Y combinator) would take well beyond the estimated lifetime of the universe.

External resources

- Benjamin Pierce's [Types and Programming Languages](#) is a great resource for the lambda calculus and programming language theory.
- [Doug Crockford's site](#) is a great reference on advanced JavaScript.
- [A cool paper](#) by [Daniel Brown](#) and [William Cook](#) on monads, mixins, inheritance and, yes, fixed points.

Related posts


- [Writing CEK-style interpreters in Haskell](#)
- [Closure conversion: How to compile lambda](#)
- [How to compile with continuations](#)
- [Understand exceptions by implementing them](#)
- [A-Normalization: Why and How](#)
- [Compiling up to the \$\lambda\$ -calculus](#)
- [Parsing with derivatives \(Yacc is dead: An update\)](#)
- [By example: Continuation-passing style in JavaScript](#)
- [7 lines of code, 3 minutes: Implement a programming language](#)
- [Architectures for interpreters](#)
- [First-class macros from meta-circular evaluators](#)
- [Programming with continuations by example](#)
- [Compiling Scheme to C](#)
- [Compiling to Java](#)
- [Church encodings in Scheme](#)
- [Non-termination without loops, iteration or recursion in Javascript](#)
- [Memoizing recursive functions in Javascript with the Y combinator](#)
- [Advanced programming languages](#)
- [Recommended books and papers for grad students](#)

[\[article index\]](#) [\[email me\]](#) [\[@mattmight\]](#) [\[+mattmight\]](#) [\[rss\]](#)

[Siebel Self Study CDs](#)

Study where and whenever you want Free online demo & latest offers

[StayAhead Java Courses](#)

Java & Web Development Courses by leading UK provider 020 7600 6116 

Latest: [Writing CEK-style interpreters in Haskell](#)

Next: [Non-termination without loops, iteration or recursion in Javascript](#)

Prev: [HOWTO: Fix mold and allergy problems](#)

Rand: [First-class macros from meta-circular evaluators](#)

matt.might.net is powered by **[linode](#)**.