

Y combinator - a short introduction

FliB lightning talk; July 22nd 2009

Thomas Holubar

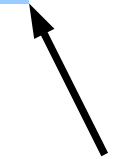
Y combinator

- getting started

Can we write recursive functions if the language does not directly support recursion?

Example: factorial

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

An arrow points from the 'factorial' argument in the recursive call '(factorial (- n 1))' to the 'factorial' identifier in the 'define factorial' line above it, illustrating a recursive call.

What if the bound identifier (“factorial”) could not be accessed in the body of the definition?

We would need to replace it with something else. What could that be?

Factorial without recursion - thoughts & ideas

What does “factorial” stand for?

It is bound to the body of the definition, something like:

```
factorial := (lambda (n)
              (if (= n 0)
                  1
                  (* n (<self> (- n 1))))))
```

But then again: What about “self”?

We could substitute it by the very same lambda expression. But that would yield an infinite series of substitutions (every lambda expression contains another reference to “self” to be substituted).

We want: write once – use everywhere ;-)

Factorial without recursion

- 1st attempt

Solution: parameterize factorial with itself!

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

;; usage:

```
((factorial factorial) 6)
```

```
; ==> 720
```

Drawback: can't be called as before – needs additional, redundant parameter.

Factorial without recursion

- 2nd attempt

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

```
(factorial 6) ; ==> 720 // But now the definition is clumsy...
```

Factorial without recursion - the magic

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

```
(define factorial (make-recursive factorial-core))
```

:: Usage:

```
(factorial 6)
```

```
; ==> 720
```

What is “make-recursive”?

Generating recursion: the Y combinator

“make-recursive” is just an alias for the famous Y combinator:

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

Generating recursion: the Y combinator

“make-recursive” is just an alias for the famous Y combinator:

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

Thank you for your time and attention!

To be continued... (derivation of Y)...