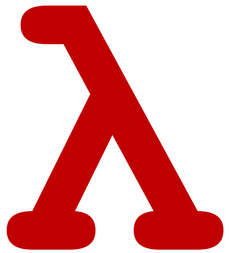
CSCI 275: Programming Abstractions

Lecture 34: The Y Combinator

Spring 2025

Stephen Checkoway
Slides from Molly Q Feldman



Motivation

How do we write a recursive function?

How do we write a recursive function?

Easy, use `define!`

```
(define len
  (lambda (lst)
    (cond [(empty? lst) 0]
          [else (add1 (len (rest lst)))])))
```

How do we write a recursive function?

Easy, use `letrec!`

```
(letrec ([len
          (lambda (lst)
            (cond [(empty? lst) 0]
                  [else (add1 (len (rest lst)))]))])
  len)
```

Recall, that this binds `len` to our function `(lambda (lst) ...)` in the body of the `letrec`

This expression returns the procedure bound to `len` which computes the length of its argument

Why does this not work to create a length procedure?

```
(let ([len
       (lambda (lst)
         (cond [(empty? lst) 0]
               [else (add1 (len (rest lst)))]))])
  len)
```

A. It would work but `letrec` more clearly conveys the programmer's intent to write a recursive procedure
B. `len` is not defined inside the `lambda`
C. `len` is not defined in the last line
D. `len` isn't being called in the last line, it's being returned and this is an error
E. None of the above

How did you feel about how we implemented `letrec` in MiniScheme?

A. I liked it!

B. It didn't feel satisfying

C. Definitely not a fan

D. Something else

Replace

```
(letrec ([f1 exp1] … [fn expn])
  body)
```

with

```
(let ([f1 0] … [fn 0])
  (let ([g1 exp1] … [gn expn])
    (begin
      (set! f1 g1)
      …
      (set! fn gn)
      body)))
```

How di
MiniSc

A. I like

B. It

C. D

D. S

Today: a *different* way to think about implementing recursion more generally!

Also a nice mix of the theory, implementation & parameter passing stuff we've been talking about!

How do we write a recursive function?

Let's just use lambdas, no Racket special forms

```
(lambda (lst)
  (cond [(empty? lst) 0]
        [else (add1 (??? (rest lst)))]))
```

How do we write a recursive function?

Let's just use lambdas, no Racket special forms

```
(lambda (lst)
  (cond [(empty? lst) 0]
        [else (add1 (??? (rest lst)))]))
```

Options for `???`:

How do we write a recursive function?

Let's just use lambdas, no Racket special forms

```
(lambda (lst)
  (cond [(empty? lst) 0]
        [else (add1 (??? (rest lst)))]))
```

Options for ???:

```
(lambda (lst) (error "List too long!"))
```

Issue: we get the right length for an empty list, but this does not work for non-empty lists

How do we write a recursive function?

Let's just use lambdas, no Racket special forms

```
(lambda (lst)
  (cond [(empty? lst) 0]
        [else (add1 (??? (rest lst)))]))
```

Options for ???:

Another copy of the function itself?

```
(lambda (lst)
  (cond [(empty? lst) 0]
        [else (add1 ((lambda (lst)
                       (cond [(empty? lst) 0]
                             [else (add1 (??? (rest lst)))]))
                     (rest lst)))]))
```

Issue: we get the right length for an empty **and single element** list, but still doesn't work in general

How do we write a recursive function?

Let's just use lambdas, no Racket special forms

```
(lambda (lst)
  (cond [(empty? lst) 0]
        [else (add1 (??? (rest lst)))]))
```

Options for ???:

Wrap the code above in a (lambda (len) …)

```
(lambda (len)
  (lambda (lst)
    (cond [(empty? lst) 0]
          [else (add1 (len (rest lst)))])))
```

Issue: This turns a function into an argument – not the functionality we really want

Progress towards what we want…

```
(define make-length
  (lambda (len)
    (lambda (lst)
      (cond [(empty? lst) 0]
            [else (add1 (len (rest lst)))]))))
```

Same function as last slide, **but bound to the identifier `make-length`**

- The orange text (together with purple text) is the body of `make-length`
- The purple text is the body of the closure returned by `(make-length x)`

Progress towards what we want…

```
(define make-length
  (lambda (len)
    (lambda (lst)
      (cond [(empty? lst) 0]
            [else (add1 (len (rest lst)))]))))
```

Same function as last slide, **but bound to the identifier `make-length`**

- The orange text (together with purple text) is the body of `make-length`
- The purple text is the body of the closure returned by `(make-length x)`

```
(define L0 (make-length (lambda (lst) (error "too long"))))
```

`L0` correctly computes the length of the empty list but fails on longer lists

Many make-length definitions

```
(define make-length
  (lambda (len)
    (lambda (lst)
      (cond [(empty? lst) 0]
            [else (add1 (len (rest lst)))]))))
```

```
(define L0 (make-length (lambda (lst) (error "too long"))))

(define L1 (make-length L0)) ;works for <= 1 element lists
```

If we have the definitions below, how can we define a new procedure `L3` that correctly calculates the length for lists of length 3 or less?

```
(define make-length
  (lambda (len)
    (lambda (lst)
      (cond [(empty? lst) 0]
            [else (add1 (len (rest lst)))]))))
```

```
(define L0
  (make-length (lambda (lst) (error "too long"))))
(define L1 (make-length L0))
```

A. `(define L3 (make-length L1))`

B. `(define L3 (make-length (make-length L1))`

C. `(define L3 (make-length 3))`

D. Something else

Many make-length definitions

```
(define make-length
  (lambda (len)
    (lambda (lst)
      (cond [(empty? lst) 0]
            [else (add1 (len (rest lst)))]))))
```

```
(define L0 (make-length (lambda (lst) (error "too long"))))

(define L1 (make-length L0)) ;works for <= 1 element lists
(define L2 (make-length L1)) ;works for <= 2 element lists
(define L3 (make-length L2)) ;works for <= 3 element lists
```

Insight: we'd need an L_∞ in order to work for all lists

We need a function on functions

```
(define L0 (make-length (lambda (lst) (error "too long"))))

(define L1 (make-length L0)) ;works for <= 1 element lists
(define L2 (make-length L1)) ;works for <= 2 element lists
(define L3 (make-length L2)) ;works for <= 3 element lists
```

In all the `LN` cases, make-length and `L(N-1)` are both functions

Some Definitions

Combinator: a function that operates on functions

Fixed-point (same as in math): a value that does not change under a given transformation

To solve our “pure” recursion problem we are going to use a term called a **fixed-point combinator**

It produces a fixed point because we want it to simply “keep returning” its argument

Enter: the Y Combinator

If `f` is a function of one argument, then `(Y f) = (f (Y f))`

```
(Y make-length)
=> (make-length (Y make-length))
=> (lambda (lst)
     (cond [(empty? lst) 0]
           [else (add1 ((Y make-length) (rest lst)))]))
```

This is precisely the length function!

```
(define length (Y make-length))
```

How is `(Y make-length)` the same as length?

```
(define length (Y make-length))
```

Let's step through applying our length function to `'(1 2 3)`

```
(length '(1 2 3))          ; so lst is bound to '(1 2 3)
=> (cond [(empty? lst) 0]
         [else (add1 ((Y make-length) (rest lst)))])
=> (add1 (length '(2 3)))
=> (add1 (cond [(empty? lst) 0]
               [else (add1 ((Y make-length) (rest lst)))]))
=> (add1 (add1 (length '(3))))
=> (add1 (add1 (cond […][else (add1 …)])))
=> (add1 (add1 (add1 (length '()))))
=> (add1 (add1 (add1 (cond [(empty? lst) 0][…]))))
=> (add1 (add1 (add1 0)))
=> 3
```

Um… what exactly is the definition of Y?

When we introduced Y, we said:
"If `f` is a function of one argument, then `(Y f) = (f (Y f))`"

Two issues:

1. We define Y in terms of Y – wasn't the whole point to write recursive anonymous functions?
2. If `(Y f) = (f (Y f))`, then
   `(f (Y f)) = (f (f (Y f)) = (f (f (f (Y f)))) =`
   …
   and this will never end

Definition of the Y Combinator in the λ-calculus

```
Y = λf. (λx. f (x x)) (λx. f (x x))
```

Let's apply it to some function `fun`

```
Y fun = (λf. (λx. f (x x)) (λx. f (x x))) fun
      → (λx. fun (x x)) (λx. fun (x x))
      → fun ((λx. fun (x x)) (λx. fun (x x)))
      = fun (Y fun)
```

Just as we wanted, `Y f = f (Y f)`

What is `Y (λ (x) x)`? i.e. `Y` applied to the identity.

```
Y = (λ (t) (λ (f) t (f f)) (λ (f) t (f f)))
```

A. `(λ (f) f f) (λ (f) f f)`

B. `(λ (f) f f)`

C. `(λ (f) f)`

D. `Y (λ (f) f)`

E. Something else

`id = λx. x` is the identity function and
`Y = λf. (λx. f (x x)) (λx. f (x x))` is the Y combinator

What is `Y id`? (I.e., apply Y to the identity function)

A. `(λx. x x) (λx. x x)`

B. `λx. x x`

C. `λx. x`

D. `Y (λf. f)`

E. Something else

Aside: Omega

$$\Omega = (\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$$

What is interesting about Ω is that, when we try to reduce it, we still just get Ω:

$$\begin{aligned}\Omega &= (\lambda x.\ x\ x)\ (\lambda x.\ x\ x) \\ &\rightarrow (\lambda x.\ x\ x)\ (\lambda x.\ x\ x) \\ &= \Omega\end{aligned}$$

Y in Racket

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

`Y` is a function of `f` and its body is applying the anonymous function `(lambda (x) (f (x x)))` to the argument `(lambda (x) (f (x x)))` and returning the result.

```
(Y foo) = ((lambda (x) (foo (x x)))
           (lambda (x) (foo (x x))))
        = (foo ((lambda (x) (foo (x x)))
                (lambda (x) (foo (x x)))))
        = (foo (Y foo))
```

Issue: The Y Combinator for Racket

This form of the Y-combinator doesn't work in Racket because the computation would never end (“CBV divergence problem”)

We can fix this by using the related Z-combinator

```
(define Z
  (lambda (f)
    ((lambda (x) (f (lambda (v) ((x x) v))))
     (lambda (x) (f (lambda (v) ((x x) v)))))))
```

Now a value, so don’t try to unroll the whole recursion!

With this definition, we can create a length function

```
(define length (Z make-length))
```

What just happened there?

We transformed `(x x)` into `(lambda (v) ((x x) v))`

`(x x)` is going to return a function in both cases and the second expression is just wrapping that function up into another function

Consider the functions `add1` and `(lambda (x) (add1 x))`

Just as these have the same behavior; `(x x)` and `(lambda (v) ((x x) v))` have the same behavior

The difference is the latter won't evaluate `(x x)` until it is called under call-by-value

Guide to Using Z Yourself!

1. Write your recursive function normally with recursive calls:
   `(define foo (lambda (x) …))`
2. Wrap the lambda in another, single-argument lambda whose argument has the same name as your function:
   `(define foo (lambda (foo) (lambda (x) …)))`
3. Apply Z to that
   `(define foo (Z (lambda (foo) (lambda (x) ...))))`
4. Recursion without special forms, achieved!

What about multi-argument functions?

We can use `apply`!

```
(define Z*
  (lambda (f)
    ((lambda (x)
       (f (lambda args (apply (x x) args))))
     (lambda (x)
       (f (lambda args (apply (x x) args)))))))
```

`args` here are the arguments to the recursive function that we are trying to write

Example: combinator `map` for one list

```
((Z* (lambda (map)
       (lambda (proc lst)
         (cond [(empty? lst) empty]
               [else (cons (proc (first lst))
                           (map proc (rest lst)))]))))
 add1
 '(1 2 3 4 5))
```

We're applying `Z*` to the orange function which returns a recursive map procedure

Then we're applying that procedure to the arguments `add1` and `'(1 2 3 4 5)`

Example: combinator `map` for two lists

```
(define map2
  (Z* (λ (map)
        (λ (f lst1 lst2)
          (cond [(empty? lst1) empty]
                [(empty? lst2) empty]
                [else (cons (f (first lst1) (first lst2))
                            (map f (rest lst1) (rest lst2)))])))))
```

We can then call `(map2 - '(1 5 3 2) '(0 2 1 6))` which returns `'(1 3 2 -4)`