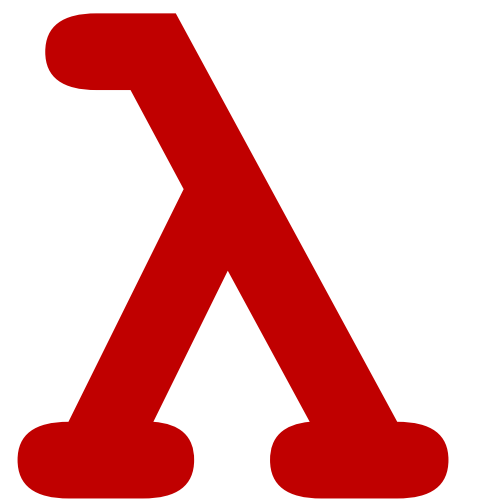


CSCI 275: Programming Abstractions

Lecture 34: The Y Combinator
Fall 2024

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

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

A. I like

expn])

B. It

Also a nice mix of the theory,

C. D

implementation & parameter

expn])

D. S

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)))])))
```

Currying!

- 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)))])))
  (error "too long"))

(define L0 (make-length 0))

(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
```

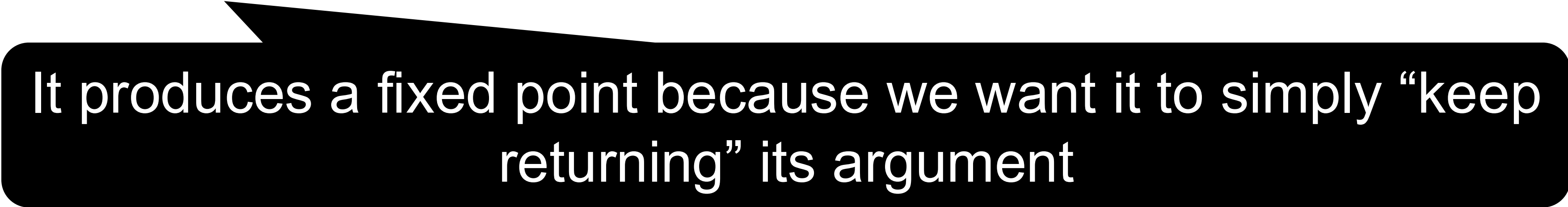
In all the L_N cases, `make-length`
and $L_{(N-1)}$ are both functions

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

$Y = (\lambda (t) (\lambda (f) t (f f)) (\lambda (f) t (f f)))$

So if we pass through some function `fact`, we get

$Y \text{ fact} = (\lambda (t) (\lambda (f) t (f f)) (\lambda (f) t (f f))) \text{ fact}$

$\rightarrow (\lambda (f) \text{ fact} (f f)) (\lambda (f) \text{ fact} (f f))$

$\rightarrow \text{fact} ((\lambda (f) \text{ fact} (f f)) (\lambda (f) \text{ fact} (f f)))$

$Y \text{ fact} = \text{fact} (Y \text{ fact})$

What is $Y (\lambda (x) x)$? i.e. Y applied to the identity.

$Y = (\lambda (t) (\lambda (f) t (f f)) (\lambda (f) t (f f)))$

A. $(\lambda (f) f f) (\lambda (f) f f)$

B. $(\lambda (f) f f)$

C. $(\lambda (f) f)$

D. $Y (\lambda (f) f)$

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 (t)
    ((lambda (f) (t (f f)))
     (lambda (f) (t (f f))))))
```

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

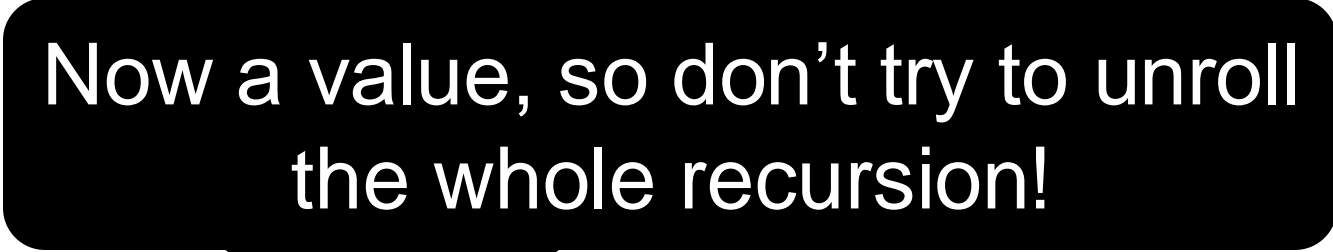
```
(Y foo) = ((lambda (f) (foo (f f)))
           (lambda (f) (foo (f f))))
        = (foo ((lambda (f) (foo (f f)))
                (lambda (f) (foo (f f)))))
        = (foo (Y foo))
```

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 (t)
    ( (lambda (f) (t (lambda (v) ((f f) v))))
      (lambda (f) (t (lambda (v) ((f f) 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))
```

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 (t)  
    ( (lambda (f) (t  
          (lambda args (apply (f f) args))))  
      (lambda (f) (t  
            (lambda args (apply (f f) args))))))
```

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

Example: combinator map

```
( (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)`