Programming Abstractions Week 4-2: Y Combinator

Stephen Checkoway

How do we write a recursive function?

Easy, use define

(define len (λ (lst) (cond [(empty? lst) 0] [else (add1 (len (

[else (add1 (len (rest lst)))])))
For the rest of this lecture, we're not going to use (define (fun args) ...)

How do we write a recursive function? (without using define) Easy, use letrec (letrec ([len $(\lambda \ (lst))$ (cond [(empty? lst) 0] [else (add1 (len (rest lst)))])) len)

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

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

```
letrec.)
```

```
(let ([len
        (\lambda \ (lst))
           (cond [(empty? lst) 0]
  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 λ

Why does this not work to create a length procedure? (Note let rather than

[else (add1 (len (rest lst)))]))])

- 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 do we write a recursive function? (just using anonymous functions created via λs)

Less easy, but let's give it a go!

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

We need to put something in the recursive case in place of the ??? but what?

If we replace the ??? with $(\lambda (lst) (error "List too long!"))$ we'll get a function that correctly computes the length of empty lists, but fails with nonempty lists

Put the function itself there?

Not a terrible attempt, we still have ?? empty list and a single element list.

- (cond [(empty? lst) 0]
 [else (add1 (??? (rest lst)))]))
 est lst)))]
- Not a terrible attempt, we still have ???, but now we can compute lengths of the

Maybe we can abstract out the function

This isn't a function that operates on lists!

using the passed in len function

- rest lst))))))
- It's a function that takes a function len as a parameter and returns a closure that takes a list lst as a parameter and computes a sort of length function

make-length

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

This is the same function as before 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 len)

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

[else (add1 (len (rest lst)))])))

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

make-length

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

- (define L0 (make-length (λ (lst) (error "too long"))))
- (define L1 (make-length L0))
- (define L2 (make-length L1))
- (define L3 (make-length L2))
- Ln correctly computes the length of lists of size at most n
- We need an L_{∞} in order to work for all lists
- (make-length length) would work correctly, but that's cheating!

[else (add1 (len (rest lst)))])))

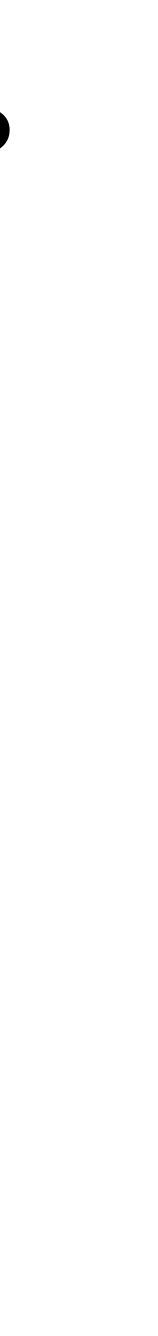
Enter the Y combinator

- Y is a "fixed-point combinator"
- Y = (S(K(S|I))(S(S(KS)K)(K(S|I))))If f is a function of one argument, then (Y f) = (f (Y f))

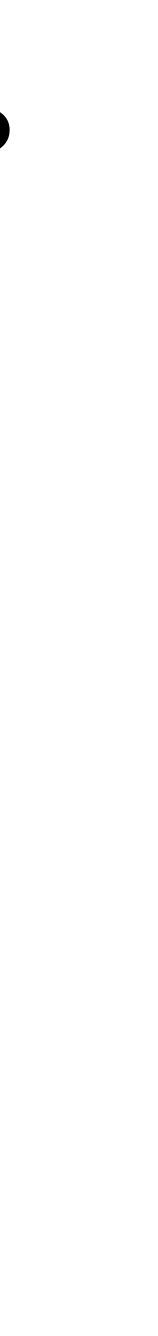
- (Y make-length)
- => (make-length (Y make-length)) $=> (\lambda (lst))$

(cond [(empty? lst) 0]

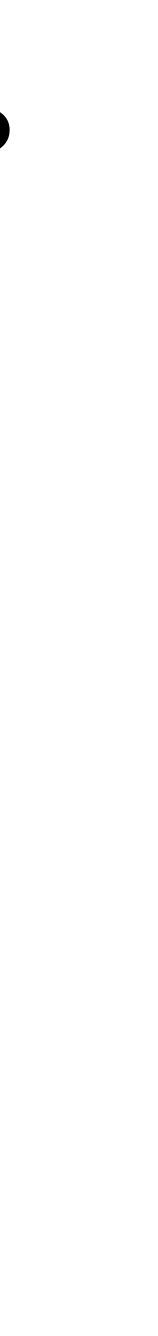
[else (add1 ((Y make-length) (rest lst)))])) This is precisely the length function: (define length (Y make-length))



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

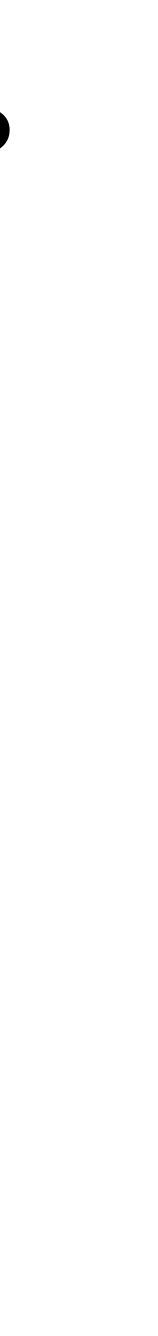


Let's step through applying our length function to '(1 2 3) (length '(1 2 3)); so lst is bound to '(1 2 3)

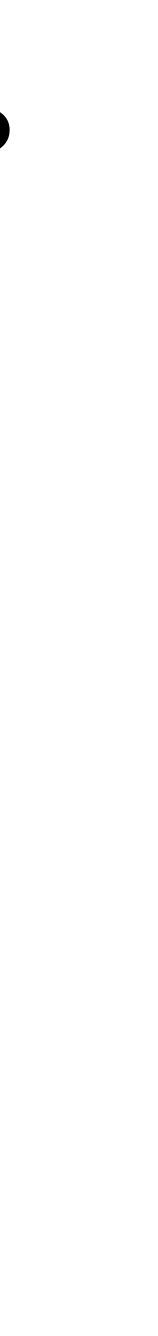


Let's step through applying our length function to '(1 2 3) (length '(1 2 3)); so lst is bound to '(1 2 3) => (cond [(empty? lst) 0]

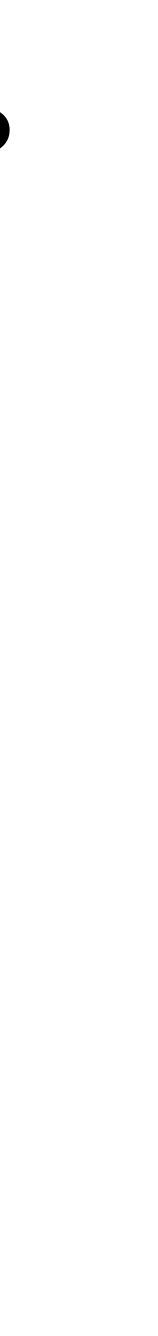
- [else (add1 ((Y make-length) (rest lst)))])



- 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))); lst is bound to '(2 3)



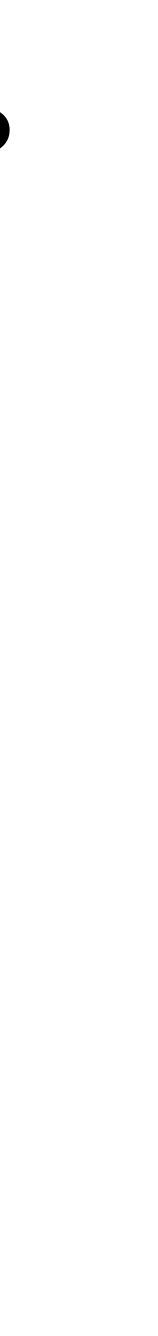
- Let's step through applying our length function to '(1 2 3) (length '(1 2 3)); so lst is bound to '(1 2 3) => (cond [(empty? lst) 0] [else (add1 ((Y make-length) (rest lst)))])
- => (add1 (length '(2 3))); lst is bound to '(2 3)
- => (add1 (cond [(empty? lst) 0]
 - [else (add1 ((Y make-length) (rest lst)))]))



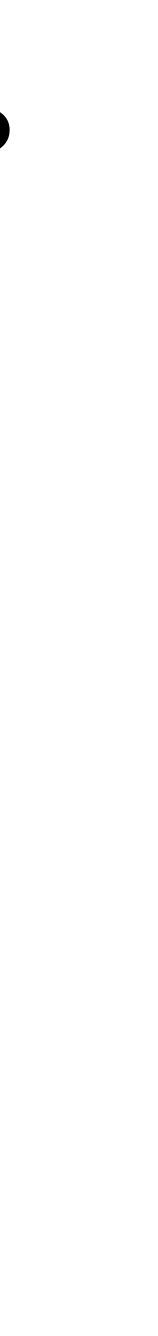
Let's step through applying our length function to '(1 2 3) (length '(1 2 3)); so lst is bound to '(1 2 3) => (cond [(empty? lst) 0]

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

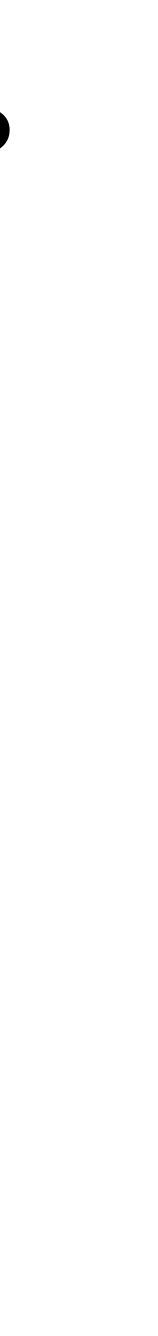
- [else (add1 ((Y make-length) (rest lst)))])



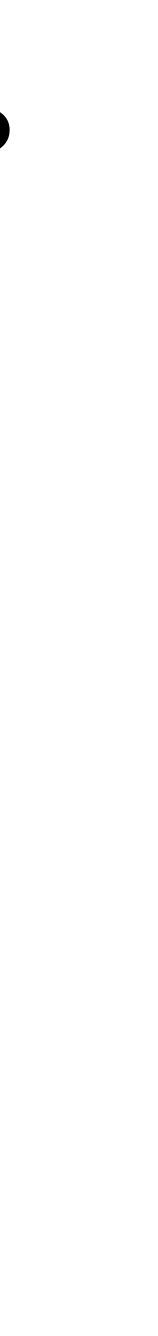
How is (Y make-length) the same as length? (define length (Y make-length)) Let's step through applying our length function to '(1 2 3) (length '(1 2 3)); so lst is bound to '(1 2 3) => (cond [(empty? lst) 0] [else (add1 ((Y make-length) (rest lst)))]) => (add1 (length '(2 3))); lst is bound to '(2 3) => (add1 (cond [(empty? lst) 0] [else (add1 ((Y make-length) (rest lst)))])) => (add1 (length '(3))); lst is bound to '(3) => (add1 (add1 (cond [...][else (add1 ...)])))



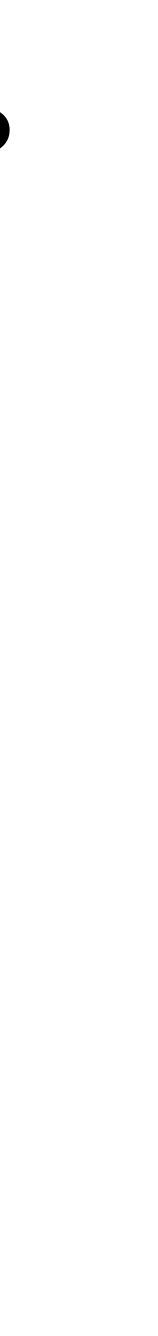
How is (Y make-length) the same as length? (define length (Y make-length)) Let's step through applying our length function to '(1 2 3) (length '(1 2 3)); so lst is bound to '(1 2 3) => (cond [(empty? lst) 0] [else (add1 ((Y make-length) (rest lst)))]) => (add1 (length '(2 3))); lst is bound to '(2 3) => (add1 (cond [(empty? lst) 0] [else (add1 ((Y make-length) (rest lst)))])) => (add1 (length '(3))); lst is bound to '(3) => (add1 (add1 (cond [...][else (add1 ...)]))) => (add1 (add1 (length '()))); lst is bound to '()



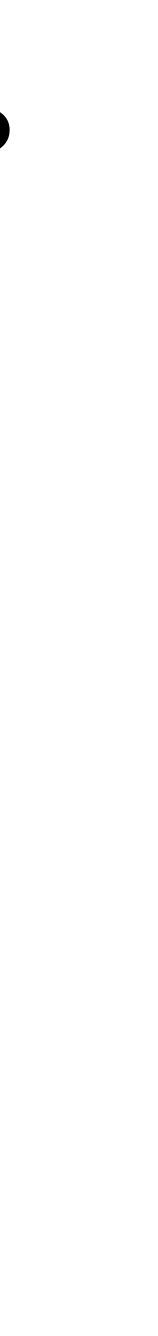
How is (Y make-length) the same as length? (define length (Y make-length)) Let's step through applying our length function to '(1 2 3) (length '(1 2 3)); so lst is bound to '(1 2 3) => (cond [(empty? lst) 0] [else (add1 ((Y make-length) (rest lst)))]) => (add1 (length '(2 3))); lst is bound to '(2 3) => (add1 (cond [(empty? lst) 0] [else (add1 ((Y make-length) (rest lst)))])) => (add1 (length '(3))); lst is bound to '(3) => (add1 (add1 (cond [...][else (add1 ...)]))) => (add1 (add1 (length '()))); lst is bound to '() => (add1 (add1 (cond [(empty? lst) 0][...])))



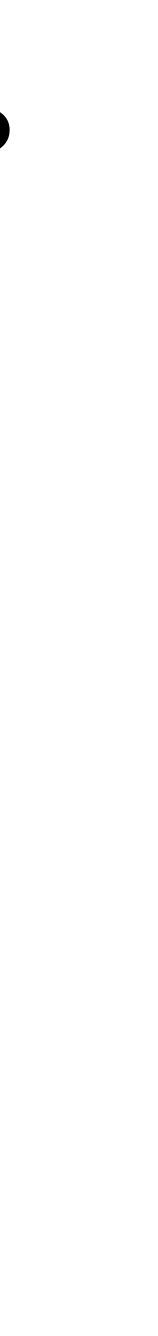
How is (Y make-length) the same as length? (define length (Y make-length)) Let's step through applying our length function to '(1 2 3) (length '(1 2 3)); so lst is bound to '(1 2 3) => (cond [(empty? lst) 0] [else (add1 ((Y make-length) (rest lst)))]) => (add1 (length '(2 3))); lst is bound to '(2 3) => (add1 (cond [(empty? lst) 0] [else (add1 ((Y make-length) (rest lst)))])) => (add1 (length '(3))); lst is bound to '(3) => (add1 (add1 (cond [...][else (add1 ...)]))) => (add1 (add1 (length '()))); lst is bound to '() => (add1 (add1 (cond [(empty? lst) 0][...]))) => (add1 (add1 (add1 0)))



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



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



But wait, how can that work?

Two problems:

- recursive anonymous functions
 - this in Racket
- (Y f) = (f (Y f)) but then (f (Y f)) = (f (f (Y f)) = (f (f (Y f))) = ...and this will never end

We defined Y in terms of Y! It's recursive and the whole point was to write

- Not quite, Y = (S(K(S|I))(S(KS)(K)(K(S|I)))), but we still need to write

Defining Y

(define Y $(\lambda (f))$ $((\lambda (g) (f (g g))))$ $(\lambda (q) (f (q q))))))$

It's tricky to see what's going on but Y is a function of f and its body is applying the anonymous function $(\lambda (g) (f (g g)))$ to the argument $(\lambda (q) (f (g q)))$ and returning the result.

 $(Y foo) = ((\lambda (g) (foo (g g)))$; By applying Y to foo $(\lambda (g) (foo (g g))))$ = (foo ($(\lambda (g) (foo (g g)))$; By applying orange fun $(\lambda (g) (foo (g g))));$ to purple argument From definition of Y = (foo (Y foo))





Never ending computation

- This form of the Y-combinator doesn't work in Scheme because the computation would never end
- We can fix this by using the related Z-combinator
- (define Z This is the argument to our recursive function $(\lambda (f))$ $(\lambda (g) (f (\lambda (v) ((g g))))$ $(\lambda (g) (f (\lambda (v) ((g g))))$
- With this definition, we can create a length function (define length (Z make-length))

What is length actually defined as here?

)))))))

```
(g g) v))))
(g g) v))))
(make-length (λ (v) ((g g) v))))
(make-length (λ (v) ((g g) v))))
```

Let's apply some equivalences

- (make-length (λ (v) (((λ (g) (make-length (λ (v) ((g g) v)))) V)))
- => (make-length (λ (v) ((Z make-length) v)))
- => (cond [(empty? lst) 0] [else (add1 ((λ (v) ((Z make-length) v)) (rest lst))]
- => (cond [(empty? lst) 0] [else (add1 ((λ (v) (length v)) (rest 1st)))])
- => (cond [(empty? lst) 0] [else (add1 (length (rest lst)))])

 $(\lambda (g) (make-length (\lambda (v) ((g g) v))))$

We can use Z to make recursive functions

Given a recursive function of one variable (define foo

(λ (x) ... (foo ...) ...)

we can construct this only using anonymous functions by way of Z $(Z (\lambda (foo) (\lambda (x) ... (foo ...))))$

```
Factorial
(Z (\lambda (fact)
     (λ (n)
        (if (zero? n)
            (* n (fact (sub1 n))))))
```

Step by step

- 1. Write your recursive function normally with recursive calls: (define foo $(\lambda (x) ...)$)
- the same name as your function: (define foo (λ (foo) (λ (x) ...))
- 3. Apply Z to that (define foo (Z (λ (foo) (λ (x) ...)))
- write recursive functions!

2. Wrap the lambda in another, single-argument lambda whose argument has

4. Be thankful that programming language designers give us easier ways to

 $(letrec ([foo (\lambda (lst) (... (foo ...) ...)]))$ (foo '(1 2 3)))

A. (let ([foo (Z (λ (lst)) (... (foo ...) ...))]) (foo '(1 2 3))) B. (let ([foo (Z (λ (foo))

 $(\lambda \ (lst))$ (foo '(1 2 3)))

C. It's not possible to write recursive functions without define or letrec in Scheme

Imagine a version of Scheme without define or letrec, how can we write a recursive function foo and call it on a list? In other words, how do we write

(... (foo ...) ...))))))

What about multi-argument functions?

We can use apply!

(define Z* $(\lambda (f))$ ((λ (g) (f (λ args (apply (g g) args))) $(\lambda (g) (f (\lambda args (apply (g g) args))))))$

This is the list of arguments to our recursive function

Example: map

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)

(empty? lst) empty]
[else (cons (proc (first lst))
 (map proc (rest lst)))]))))