

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 (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
          (λ (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

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

```
(let ([len
      (λ (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 `λ`
- 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 λ s)

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?

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

Not a terrible attempt, we still have ???, but now we can compute lengths of the empty list and a single element list.

Maybe we can abstract out the function

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

This isn't a function that operates on lists!

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 using the passed in `len` function

make-length

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

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"))))
```

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

make-length

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

```
(define L0 (make-length (λ (lst) (error "too long"))))
(define L1 (make-length L0))
(define L2 (make-length L1))
(define L3 (make-length L2))
```

- L_n 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!

Enter the Y combinator

Y is a "fixed-point combinator"

▸ $Y = (S (K (S I I)) (S (S (K S) K) (K (S I I))))$

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

```
(Y make-length)
=> (make-length (Y make-length))
=> (λ (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 this length?

How is this length?

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

How is this 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)

How is this 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)))])
```

How is this 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)

How is this 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)))])))

How is this 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 (add1 (length '(3)))) ; lst is bound to '(3)

How is this 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 (add1 (length '(3)))) ; lst is bound to '(3)

=> (add1 (add1 (cond [...] [else (add1 ...)])))

How is this 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 (add1 (length '(3)))) ; lst is bound to '(3)

=> (add1 (add1 (cond [...] [else (add1 ...)])))

=> (add1 (add1 (add1 (length '())))) ; lst is bound to '()

How is this 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 (add1 (length '(3)))) ; lst is bound to '(3)

=> (add1 (add1 (cond [...] [else (add1 ...)])))

=> (add1 (add1 (add1 (length '())))) ; lst is bound to '()

=> (add1 (add1 (add1 (cond [(empty? lst) 0] [...]))))

How is this 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 (add1 (length '(3)))) ; lst is bound to '(3)

=> (add1 (add1 (cond [...] [else (add1 ...)])))

=> (add1 (add1 (add1 (length '())))) ; lst is bound to '()

=> (add1 (add1 (add1 (cond [(empty? lst) 0] [...]))))

=> (add1 (add1 (add1 0)))

How is this 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 (add1 (length '(3)))) ; lst is bound to '(3)

=> (add1 (add1 (cond [...] [else (add1 ...)])))

=> (add1 (add1 (add1 (length '())))) ; lst is bound to '()

=> (add1 (add1 (add1 (cond [(empty? lst) 0] [...]))))

=> (add1 (add1 (add1 0)))

=> 3

How is this 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 (add1 (length '(3)))) ; lst is bound to '(3)

=> (add1 (add1 (cond [...] [else (add1 ...)])))

=> (add1 (add1 (add1 (length '())))) ; lst is bound to '()

=> (add1 (add1 (add1 (cond [(empty? lst) 0] [...]))))

=> (add1 (add1 (add1 0)))

=> 3

But wait, how can that work?

Two problems:

- ▶ We defined Y in terms of Y! It's recursive and the whole point was to write recursive anonymous functions
 - Not quite, $Y = (S (K (S I I)) (S (S (K S) K) (K (S I I))))$, but we still need to write this in Racket
- ▶ $(Y\ f) = (f\ (Y\ f))$ but then
$$(f\ (Y\ f)) = (f\ (f\ (Y\ f))) = (f\ (f\ (f\ (Y\ f)))) = \dots$$
and this will never end

Defining Y

```
(define Y
  (λ (f)
    ( (λ (g) (f (g g)))
      (λ (g) (f (g g))) ) ) )
```

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

```
(Y foo) = ( (λ (g) (foo (g g)))           ; By applying Y to foo
            (λ (g) (foo (g g))) )
        = (foo ( (λ (g) (foo (g g)))      ; By applying orange fun
                (λ (g) (foo (g g))) ) ) ; to purple argument
        = (foo (Y foo))                  ; From definition of Y
```

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  
  (λ (f)  
    ( (λ (g) (f (λ (v) ((g g) v))))  
      (λ (g) (f (λ (v) ((g g) v)))))))
```

This is the argument to our recursive function

With this definition, we can create a length function

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

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 (λ (foo) (λ (x) ... (foo ...) ...)))
```

Factorial

```
(Z (λ (fact)  
  (λ (n)  
    (if (zero? n)  
        1  
        (* n (fact (sub1 n)))))))
```

What about multi-argument functions?

We can use apply!

```
(define z*  
  (λ (f)  
    ( (λ (g) (f (λ args (apply (g g) args))))  
      (λ (g) (f (λ args (apply (g g) args)))))))
```

This is the list of arguments to our recursive function

Example: map

```
( ( Z*  (λ (map)
          (λ (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)`

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

```
(letrec ([foo (λ (lst) (... (foo ...) ...))] )  
  (foo '(1 2 3)))
```

A.

```
(let ([foo (z (λ (lst)  
                (... (foo ...) ...))] )  
  (foo '(1 2 3)))
```

B.

```
(let ([foo (z (λ (foo)  
                (λ (lst)  
                  (... (foo ...) ...))))) ] )  
  (foo '(1 2 3)))
```

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