

Y-Combinator

Y-Combinator Background

- A Y-Combinator is considered a Recursion Maker
- It is complex code, designed to make people ooh-and-aahh about Racket/Scheme/Lisp
- Let's see whether we can figure out how it works.

Set-up

- Consider the following function.
 - What does it do when passed with `(lambda (x) x)`
- ```
(define foo
 (lambda (g)
 (lambda (n)
 (if (zero? n)
 1
 (* n (g (- n 1)))))))
```
- It returns a procedure of one arg, the one with parameter **n**.

## Set-up

- Follow-up question, what happens when we call:

```
((foo (lambda (x) x)) 5)
```

```
(define foo
 (lambda (g)
 (lambda (n)
 (if (zero? n)
 1
 (* n (g (- n 1)))))))
```

- It returns 20, which is  $5 * 4$
- Key observation, no recursion, just a procedure call on `(- n 1)`

## Scary Code

```
(define Y
 (lambda (f)
 ((lambda (x)
 (f (lambda (t)
 ((x x) t))))
 (lambda (z)
 (f (lambda (s)
 ((z z) s))))))))
```

- Observation:  $f$  has to be a procedure.
- Observation: No recursion here, moving on...

## Scary Code

```
(define Y
 (lambda (f)
 ((lambda (x)
 (f (lambda (t)
 ((x x) t))))
 (lambda (z)
 (f (lambda (s)
 ((z z) s))))))))
```

- The shaded code is equal, safe for different variable names.

## Scary Code

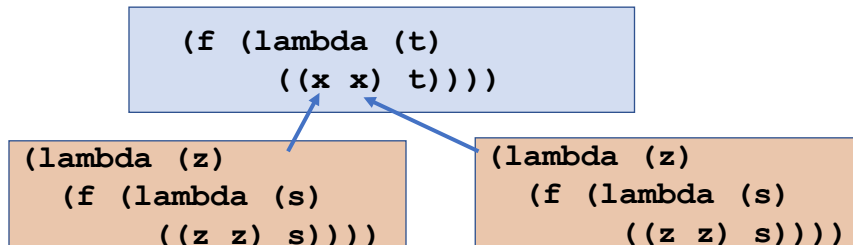
```
(define Y
 (lambda (f)
 ((lambda (x)
 (f (lambda (t)
 ((x x) t))))
 (lambda (z)
 (f (lambda (s)
 ((z z) s))))))))
```

- Both shaded portions are part of an app expression.

## Scary Code - Dissection

- Let's dissect the code.
- The result of the app-exp evaluation is that the red-shaded code gets assigned to **x** :

```
((lambda (x)
 (f (lambda (t)
 ((x x) t))))
 (lambda (z)
 (f (lambda (s)
 ((z z) s))))))
```



## Scary Code – One Step at a Time

- Suppose **f** is **foo**:

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

```
(define foo
 (lambda (g)
 (lambda (n)
 (if (zero? n)
 1
 (* n (g (- n 1)))))))
```

- Then `(lambda (t) ((x x) t))` will be assigned to **g**.
- And the interpreter returns:

```
(lambda (n)
 (if (zero? n)
 1
 (* n (g (- n 1)))))
```

## Scary Code

```
(lambda (n)
 (if (zero? n)
 1
 (* n (g (- n 1)))))
```

- Suppose we call the returned procedure with 5:

- We get:

```
(* 5 (g 4))
```

- So what again was **g**, you ask?

- Here it is:

```
(lambda (t) ((x x) t))
```

- Since **t** is 4, we get:

```
((x x) 4)
```

- Ok, and what was **x** again?

- The red shaded code:

```
(lambda (z)
 (f (lambda (s)
 ((z z) s))))
```

## Scary Code

- You've got to be kidding me!
- So, when we issue:  
`((x x) 4)`
- We get:

```
(((lambda (z)
 (f (lambda (s)
 ((z z) s))))
```

```
(lambda (z)
 (f (lambda (s)
 ((z z) s))))
```

```
) 4)
```

- Wait, isn't that what we started out with?

```
(lambda (z)
 (f (lambda (s)
 ((z z) s))))
```

## Yes, indeed!

```
(((lambda (z)
 (f (lambda (s)
 ((z z) s))))
```

```
(lambda (z)
 (f (lambda (s)
 ((z z) s))))
```

```
) 4)
```

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

```
(lambda (z)
 (f (lambda (s)
 (((z z) s)))))
```

- Notice that safe for different variable names in the original version, both versions are the same.
- Double notice: I changed the variable names in the original version, so as to make things just slightly less confusing.

## Wrap-up

- So how do you initially call this code?
- I am glad you asked.
- We already established that we should call **Y** with **foo**: **(Y foo)**
- This returned a procedure that needs an integer, hence the overall call is: **((Y foo) 5)** or some other integer of your choosing.

## Want to do that again?

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

(define foo
 (lambda (g)
 (lambda (n)
 (if (zero? n)
 1
 (* n (g (- n 1)))))))
```

- Call like so: **((Y foo) 5)**

## Handout

```
(define Y
 (lambda (f)
 ((lambda (x)
 (f (lambda (t)
 ((x x) t))))
 (lambda (z)
 (f (lambda (s)
 ((z z) s))))))))))

(define foo
 (lambda (g)
 (lambda (n)
 (if (zero? n)
 1
 (* n (g (- n 1))))))))
```

- Call like so: `((Y foo) 5)`