

# CS339: Abstractions and Paradigms for Programming

*Closures and Let Bindings*

**Manas Thakur**  
CSE, IIT Bombay



Autumn 2024

# Recall first-class values

---

- In a PL, a value is **first-class** if it can be:
  - named
  - taken as an argument by a procedure
  - returned back from a procedure
  - stored into data structures
- We had seen examples of passing procedures as arguments.



# Let's return functions

---

```
; Equivalent:  
> (define (make-inc)  
    (define (foo x) (+ x 1))  
    foo)
```

```
> (define (make-inc)  
    (lambda (x) (+ x 1)))
```

A function that returns a function.

```
> ((make-inc) 15)
```

Apply the returned function.

```
> (define inc (make-inc))
```

Name and use the returned function.

```
> (inc 15)
```



# Now let's see a MAGIC

---

```
> (define (make-addx x)
    (lambda (y) (+ x y)))

> (define magic (make-addx 3))

> (magic 4)
7
```

➤ What was the magic?

Where did the '3' come from?

You might for now say “Substitution”. But...



# MAGIC continues...

---

```
> (define x 4)
> (define (foo y)
  (define x 4)
  (define (bar z)
    (+ x y z))
  bar)
> ((foo 5) 40)
49
```

```
> (define x 100)
> ((foo 5) 40)
49
```

Now where is the '4' coming from?

Procedures in Scheme are not simply functions; they are **CLOSURES**.



# Closure = Lambda + Environment

---

- Along with the body of a lambda, closures encapsulate bindings from the environment in which the lambda was defined.
- When we apply a closure, after substituting arguments for parameters, we evaluate the body in the enclosed environment.
- This also happens when we pass a procedure:

```
> (define (foobar f x) (f x))  
  
> (define (foo y)  
  (define x 4)  
  (define (bar z)  
    (+ x y z)))  
  (foobar bar 40))  
  
> (foo 5)  
  
49
```



# Abstraction with Procedures as Return Values

- Recall how we took averages while computing square root to converge and reach the answer. *Average damping* is a general technique:

```
(define (avg-damp f)
  (lambda (z) (avg z (f z))))
```

```
(define (fixed-point f start)
  (define tolerance 0.001)
  (define (close-enough? u v)
    (< (abs (- u v)) tolerance))
  (define (iter old new)
    (if (close-enough? old new)
        new
        (iter new (f new))))
  (iter start (f start)))
```

; Square root again:

```
(define (sqrt x)
  (fixed-point (avg-damp (lambda (y) (/ x y)))
    1.0))
```

How does this work?

How is this different from last class's sqrt?





# Abstraction with Procedures as Return Values [Cont.]

; Last class

```
(define (sqrt x)
  (fixed-point (lambda (y) (avg y (/ x y)))
               1.0))
```

```
(define (avg-damp f)
  (lambda (z) (avg z (f z))))
```

; Today

```
(define (sqrt x)
  (fixed-point (avg-damp (lambda (y) (/ x y)))
               1.0))
```

Returning a function allowed us to express average-damping as a *general* concept, and *abstracted* away the specific logic for sqrt.





# Local Names

$$f(x, y) = x(1+xy)^2 + y(1-y) + (1+xy)(1-y)$$

Let  $a$  be  $(1+xy)$  and  $b$  be  $(1-y)$ . Thus:

$$f(x, y) = xa^2 + yb + ab$$

```
(define (f x y)
  (let ((a (+ 1 (* x y)))
        (b (- 1 y)))
    (+ (* x (square a))
      (* y b)
      (* a b))))
```

let bindings

let body

**let** bindings are created simultaneously, and **let** body is evaluated with respect to those bindings.



# Let's Practice

---

```
> (define x 2)
> (let ((x 3)
        (y (+ x 2)))
    (* x y))
```

➤ 15? 12? Error? 6?

➤ 12

➤ When will there be an error?

➤ Remove (define x 2)

`let` bindings are created *simultaneously*, and `let` body is evaluated with respect to those bindings.



# Which syntax are we sugar for?

**let** bindings are created *simultaneously*, and **let** body is evaluated with respect to those bindings.

```
> (define (f x y)
    (let ((a (+ 1 (* x y)))
          (b (- 1 y)))
      (+ (* x (square a))
         (* y b)
         (* a b))))
> (f 2 3)
```

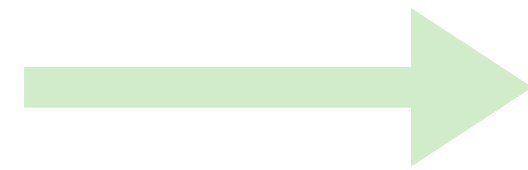


```
> (define (f x y)
    ((lambda (a b)
      (+ (* x (square a))
         (* y b)
         (* a b)))
      (+ 1 (* x y))
      (- 1 y)))
> (f 2 3)
```

# Lambda-Lambda Everywhere!

```
> (define (f x y)
  ((lambda (a b)
    (+ (* x (square a))
      (* y b)
      (* a b))))
  (+ 1 (* x y))
  (- 1 y)))

> (f 2 3)
```



```
> (define (f x y)
  ((lambda (a b)
    (+ (* x ((lambda (x) (* x x)) a))
      (* y b)
      (* a b))))
  (+ 1 (* x y))
  (- 1 y)))

> (f 2 3)
```

```
> ((lambda (x y)
  ((lambda (a b)
    (+ (* x ((lambda (x) (* x x)) a))
      (* y b)
      (* a b))))
  (+ 1 (* x y))
  (- 1 y)))
2 3)
```

With **Lambda Calculus**, we can even remove the numbers and operators :-)