CS339: Abstractions and Paradigms for Programming

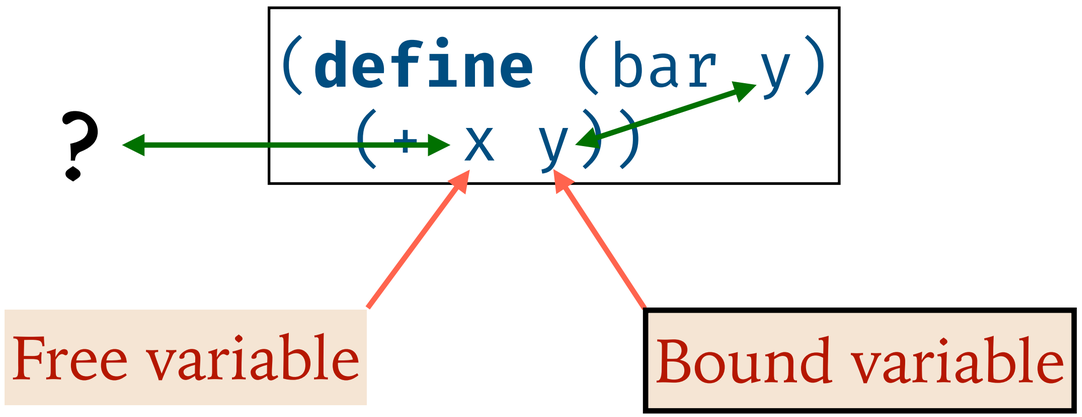
Higher Order Functions

Manas Thakur
CSE, IIT Bombay

Autumn 2024

The Blackbox [Procedural] Abstraction

- Allows procedures to be reused similar to primitive operators
- Enhances the vocabulary of the language
- Provides a namespace for variables



Scoping

- Determines which values are **free** variables bound to.
- Lexical/Static scoping: Look into the environment in which the procedure was defined.
- Dynamic scoping: Look into the environment in which the procedure was called.

Static Scoping

```
> (define x 20)
> (define (foo)
    (define x 30)
    (define (bar y)
      (+ x y))
    (bar 40))
> (foo)
```

70

Static Scoping

```
> (define x 20)
> (define (bar y)
      (+ x y))
> (define (foo)
    (define x 30)
    (bar 40))
> (foo)
```

60

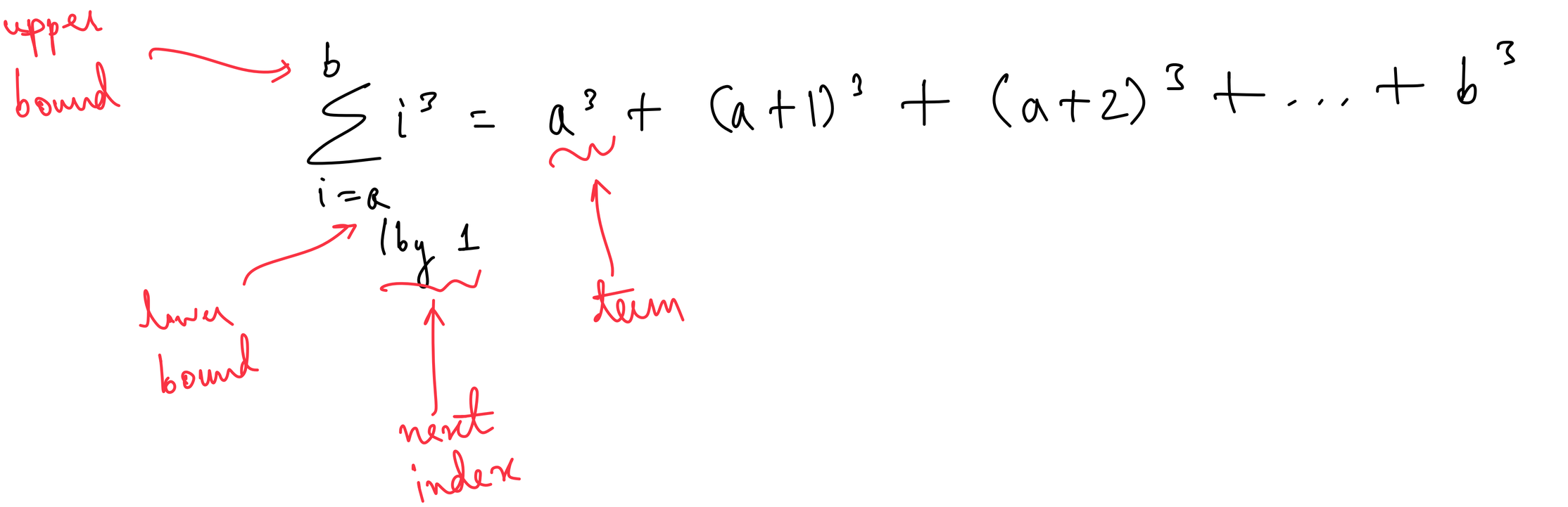
Dynamic Scoping

```
> (define x 20)
> (define (bar y)
      (+ x y))
> (define (foo)
    (define x 30)
    (bar 40))
> (foo)
```

70

Summation in Mathematics

$$\sum_{i=1}^{10} i = 1 + 2 + 3 + \ldots + 10$$



Summing a series of numbers

- Sum the integers from a to b:

Do they look similar?

```
(define (sum-ints a b)
  (if (> a b)
      0
      (+ a (sum-ints (+ a 1) b))))
```

- Sum the cubes of integers from a to b:

```
(define (sum-cubes a b)
  (if (> a b)
      0
      (+ (cube a) (sum-cubes (+ a 1) b))))
```

How about this one?

- Sum to obtain $\pi/8$:

$$\frac{1}{1.3} + \frac{1}{5.7} + \frac{1}{9.11} + \ldots \approx \frac{\pi}{8}$$

```
(define (sum-pi a b)
  (if (> a b)
      0
      (+ (/ 1.0 (* a (+ a 2)))
         (sum-pi (+ a 4) b))))
```

Now why do we have so many of them!

```
(define (sum-ints a b)
  (if (> a b)
      0
      (+ a
         (sum-ints (+ a 1) b))))
```

```
(define (sum-cubes a b)
  (if (> a b)
      0
      (+ (cube a)
         (sum-cubes (+ a 1) b))))
```

```
(define (sum-pi a b)
  (if (> a b)
      0
      (+ (/ 1.0 (* a (+ a 2)))
         (sum-pi (+ a 4) b))))
```

- What is the common structure?

```
(define (<name> a b)
  (if (> a b)
      0
      (+ <term(a)>
         (<name> <next(a)> b))))
```

The HOF Abstraction

```
(define (sum-series term a next b)
  (if (> a b)
      0
      (+ (term a)
         (sum-series (next a) b))))
```

- Sum the integers from a to b:

```
(define (id x) x)

(define (inc x) (+ x 1))

(define (sum-ints a b)
  (sum-series id a inc b))

(sum-ints 1 10)
```

- Sum the cubes of integers from a to b:

```
(define (cube x) (* x x x))

(define (sum-cubes a b)
  (sum-series cube a inc b))
```

- Sum to obtain $\pi/8$?

Another example

```
(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)))

(define (avg x y) (/ (+ x y) 2))

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

- How does it work?

```
<proc>:

  (lambda (y)
    (avg y (/ 2 y)))
```

```
(sqrt 2)

(fixed-point <proc> 1.0)

(iter 1.0 (<proc> 1.0))

(iter 1.0 (avg 1.0 (/ 2 1.0)))

(iter 1.0 1.5)

...
```

First-class values

Passing the `term` and the `next` functions as arguments allowed us to express summation as a *general* concept, and *abstracted* the specific logic for given series.

- 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
- Tomorrow we would return functions from higher order procedures, and see that real magic begins!