Programming Abstractions

Lecture 4: Environments and Closures

Stephen Checkoway

Local variables

`(let ([id1 s-exp1] [id2 s-exp2]…) body)`

`let` enables us to create some new bindings that are visible only inside `body`

```
(let ([x 37]          ; binds 37 to x
      [y (foo 42)]) ; binds the result of (foo 42) to y
  (if (< x y)
      (bar x)
      (bar y)))
```

`x` and `y` are only bound inside the body of the let expression

That is, the *scope* of the identifiers bound by `let` is `body`

Example

```
(define (sum-of-odd lst)
  (if (empty? lst)
      0
      (let ([head (first lst)]
            [tail (rest lst)])
        (if (odd? head)
            (+ head (sum-of-odd tail))
            (sum-of-odd tail)))))
```

Using variables

Recall that when Racket evaluates a variable, the result is the value that the variable is bound to

- If we have `(define x 10)`, then evaluating `x` gives us the value `10`
- If we have `(define (foo x) (- x y))`, then evaluating `foo` gives us the procedure `(λ (x) (- x y))` along with a way to get the value of `y`

Racket needs a way to look up values that correspond to variables: an environment

Environments

Environments are mappings from identifiers to values

There's a top-level environment containing many default mappings

- `list ↦ #<procedure:list>`
  (↦ is read as "maps to", #<procedure:xxx> is how DrRacket displays procedures)
- `+ ↦ #<procedure:+>`

Each file in Racket (technically, a module) has an environment that extends the top-level environment that contains all of the defines in the file

Basic operations on environments

Lookup an identifier in an environment

Bind an identifier to a value in an environment

Extend an environment

- This creates a new environment with mappings from identifiers to values as well as a reference to the environment being extended
- The extended and original environment may both contain mappings for the same identifier

Modify the binding of an identifier in an environment (we will avoid doing this in this course)

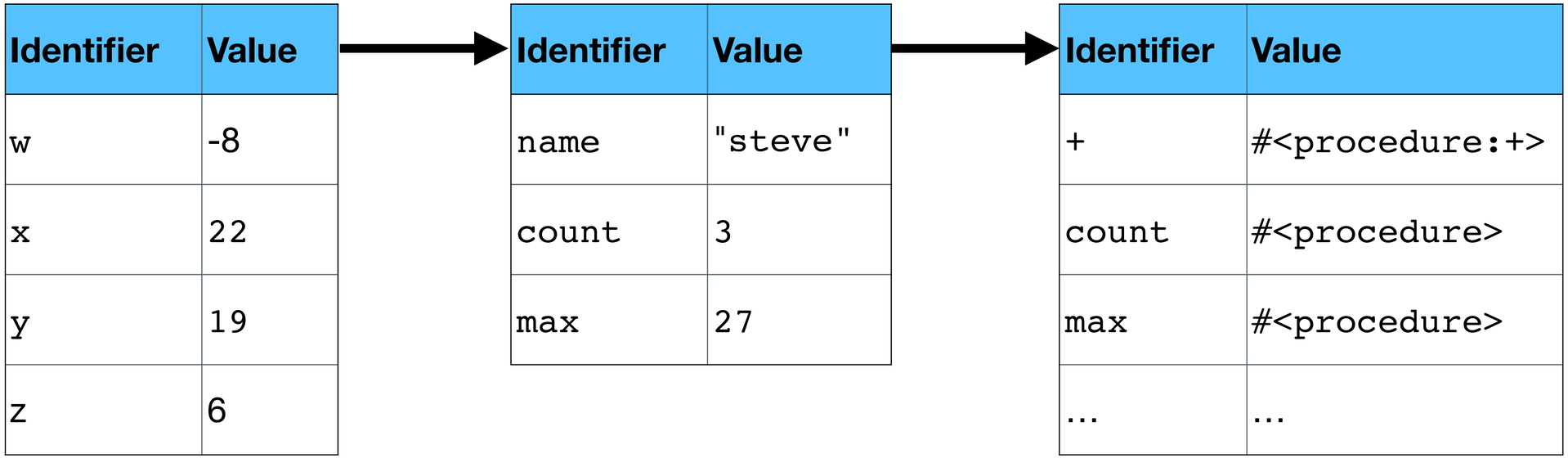
Looking up an identifier in an environment

If an identifier has been bound in the current environment, its value is returned

Otherwise, if the current environment extends another environment, the identifier is (recursively) looked up in the other environment.

Otherwise, there's no binding for the identifier and an error is reported

Consider the environments where (A → B means A extends B).



Identifier	Value
w	-8
x	22
y	19
z	6

Identifier	Value
name	"steve"
count	3
max	27

Identifier	Value
+	#<procedure:+>
count	#<procedure>
max	#<procedure>
...	...

What is the value of looking up `count` in the left-most environment?

A. Error: `count` is undefined in that environment

B. `3`

C. A procedure

Adding a new mapping to an environment

`(define identifier s-exp)`

`define` will add `identifier` to the current environment and bind the value that results from evaluating `s-exp` to it

In any environment, an identifier may only be defined once

- except in the interpreter which lets you redefine identifiers

Adding a new mapping to an environment

`(define (identifier params) body)`

Recall that `(define (foo x y) body)` is the same as
`(define foo (λ (x y) body))`
in that it binds the value of the λ-expression, namely a closure, to `foo`

A closure keeps a reference to the current environment in which the λ-expression was evaluated

Extending an environment

Calling a closure

Calling a closure extends the environment of the closure with the values of the arguments bound to the procedure's parameters

```
(define (sum lst)
  (cond [(empty? lst) 0]
        [else (+ (first lst) (sum (rest lst)))]))

(define (average lst)
  (/ (sum lst) (length lst)))
```

Calling `(average '(1 2 3))` extends the environment of average (namely the module's environment which contains mappings for `sum` and `average`) with the mapping `lst ↦ '(1 2 3)` and runs `average` with that environment

Example bindings

Shadowing a binding

```
(define (sum lst)
  (cond [(empty? lst) 0]
        [else (+ (first lst) (sum (rest lst)))]))

(define (foo sum x y)
  (average (list sum x y)))

(define (average lst)
  (/ (sum lst) (length lst)))
```

Inside the body of `foo`, `sum` refers to the parameter
Inside the body of `average`, `sum` refers to the procedure

Example bindings

Shadowing a binding

```
(define (sum lst)
  (cond [(empty? lst) 0]
        [else (+ (first lst) (sum (rest lst)))]))

(define (foo sum x y)
  (average (list sum x y)))

(define (average lst)
  (/ (sum lst) (length lst)))
```

Inside the body of `foo`, `sum` refers to the parameter

Inside the body of `average`, `sum` refers to the procedure

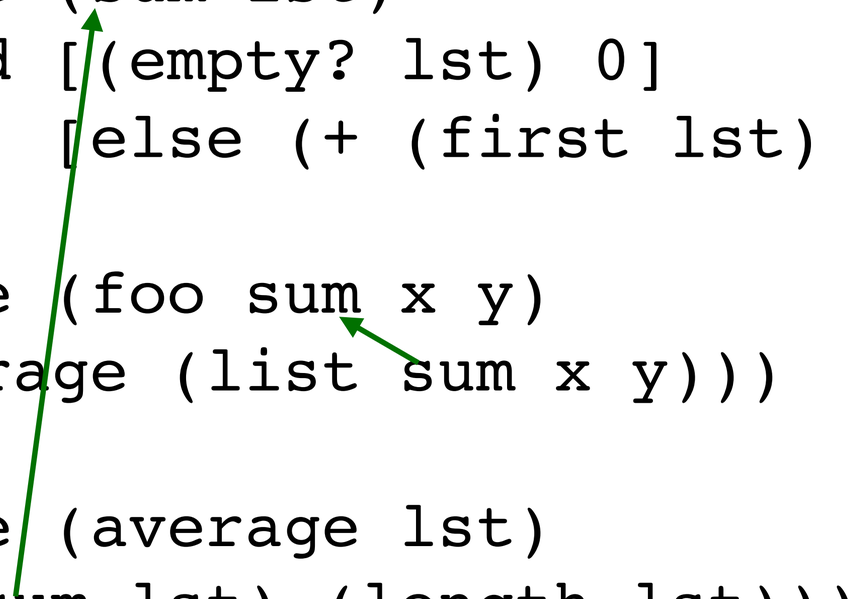
Example bindings

Shadowing a binding

```
(define (sum lst)
  (cond [(empty? lst) 0]
        [else (+ (first lst) (sum (rest lst)))]))

(define (foo sum x y)
  (average (list sum x y)))

(define (average lst)
  (/ (sum lst) (length lst)))
```



Inside the body of `foo`, `sum` refers to the parameter

Inside the body of `average`, `sum` refers to the procedure

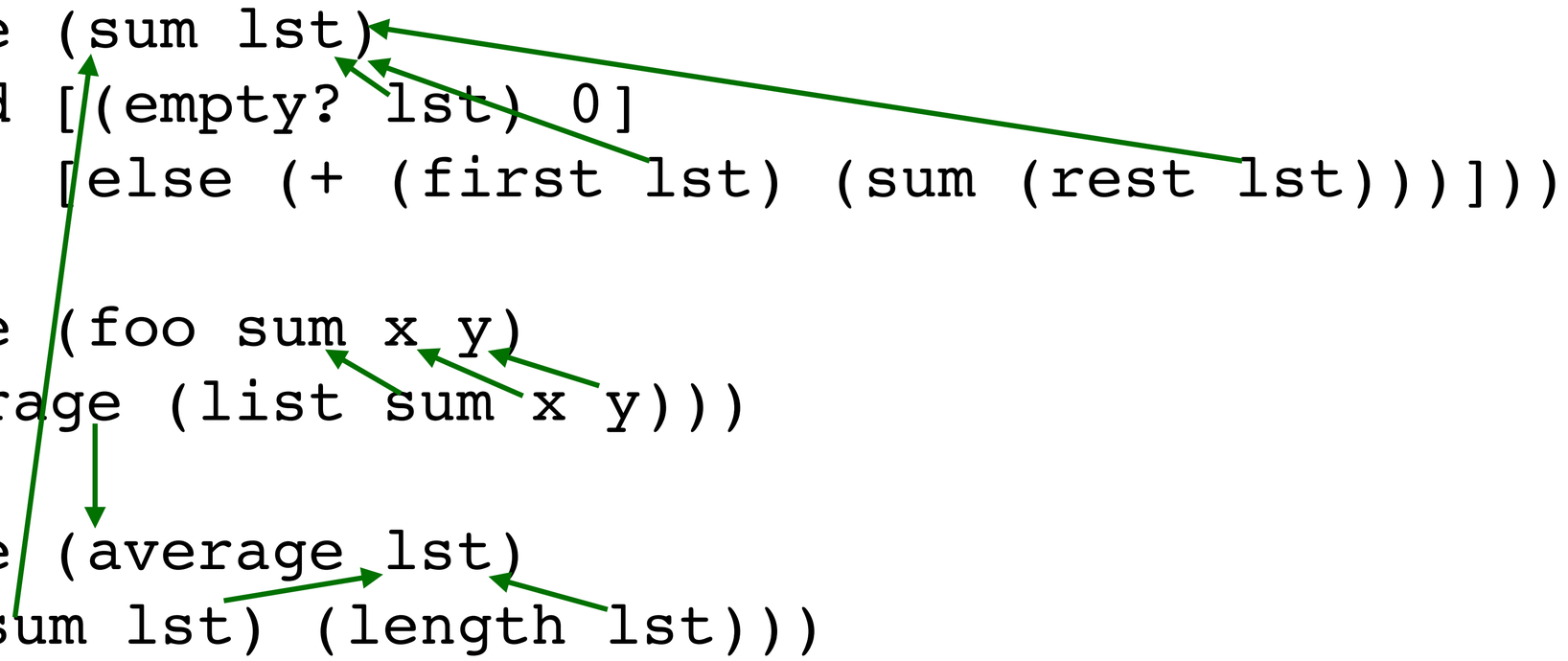
Example bindings

Shadowing a binding

```
(define (sum lst)
  (cond [(empty? lst) 0]
        [else (+ (first lst) (sum (rest lst)))]))

(define (foo sum x y)
  (average (list sum x y)))

(define (average lst)
  (/ (sum lst) (length lst)))
```



Inside the body of `foo`, `sum` refers to the parameter

Inside the body of `average`, `sum` refers to the procedure

Extending an environment

`(let ([id1 s-exp1] [id2 s-exp2]…) body)`

let extends its environment

```
(let ([x 37]          ; binds 37 to x
      [y (foo 42)]) ; binds the result of (foo 42) to y
  (if (< x y)
      (bar x)
      (bar y)))
```

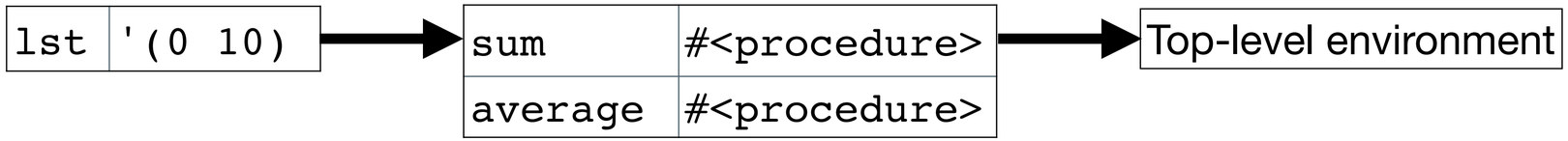
`x` and `y` are only bound inside the body of the let expression

That is, the *scope* of the identifiers bound by `let` is `body`

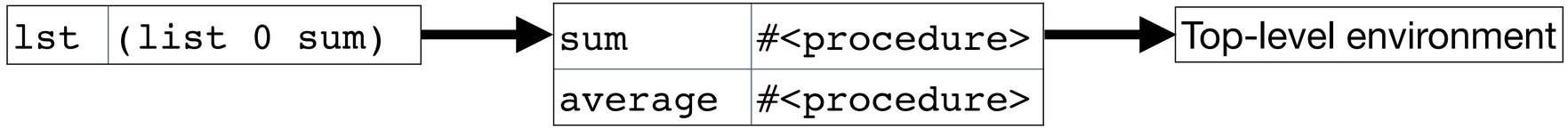
```
(define (sum lst)
  (if (empty? lst)
      0
      (+ (first lst) (sum (rest lst)))))
(define (average lst)
  (/ (sum lst) (length lst)))
(let ([sum 10])
  (average (list 0 sum)))
```

In the body of `average`, while computing `(average (list 0 sum))`, which of the following is `average`'s environment (an arrow points at the environment being extended)?

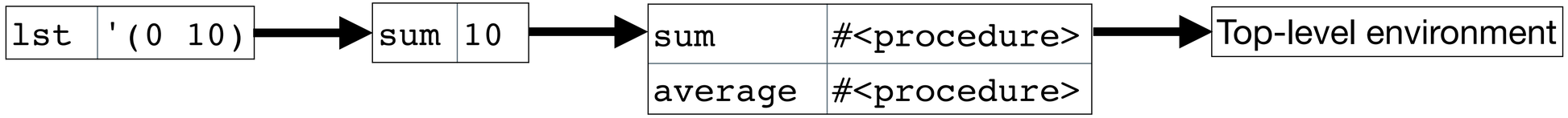
A.



B.



C.



Example: Filtering a list

`(filter pred lst)`

filter takes a predicate (a 1-argument function that returns `#t` or `#f`) and a list and returns a list as follows

- For each element `x` in `lst`, run `(pred x)`
- If `(pred x)` returns true (anything other than `#f`), add `x` to the list to return

Examples

- `(filter positive? '(2 -3 4 5 -1 0)) => '(2 4 5)`
- ```
  (filter (λ (s) (string-prefix? s "A"))
          '("Adam" "Janet" "Alice")) => '("Adam" "Alice")
  ```
  ```

Passing a closure to filter

```
(define (filter pred lst)
  (cond [(empty? lst) empty]
        [(pred (first lst)) (cons (first lst)
                                  (filter pred (rest lst)))]
        [else (filter pred (rest lst))]))

(define (foo prefix lst)
  (filter (λ (s) (string-prefix? s prefix)) lst))
```

Modifying a binding

Scheme lets us modify a binding, but we're not going to do that

This type of side-effect makes reasoning about code much harder