Closures

CSCI 3136: Principles of Programming Languages

Warning:
Jargon Ahead

Agenda

- Announcements
  - Assignment 7 is out and due July 12.
  - Scheme Tutorial on Monday, 15th, 2:30 – 4pm in CS 127
- Readings: Read Chapter 3.6
- Lecture Contents
  - Motivation
  - Definition of a closure
  - Linked Referencing Environments
  - Closures in Scheme

Halfway Check-in Survey Results

What's working

- Top Hat quizzes
- Lectures
- Slides
- Approachable

What needs work

- More examples needed
- Assignments are too hard
- Writing on whiteboard

A Lambda Aside

- In Scheme, the Lambda keyword defines a function
- Example:

```
( define add ( lambda ( a b )
                 ( + a b )
               )
)
```

- This code defines a function that
  - Takes two arguments
  - Adds them together
  - Returns the result
- The `define` keyword binds the name **add** to this function.
- So, how is a closure different from a function?

A Useful Short-form

In Scheme, this

```
( define add ( lambda ( a b )
                  ( + a b )
               )
)
```

Is equivalent to

```
( define (add a b)
                  ( + a b )
)
```

The latter is a short form for the former, only available for the `define` statement

Shallow or Deep Binding in Scheme?

```
(define increase_x
  (lambda ()
    (set! x (+ x 1))))

(define execute
  (lambda (f)
    (let ((x 20))
      (display (list "inner x before:" x))
      (f)
      (display (list "inner x after:  " x)))))

(define x 1)
(display (list "outer x before:" x))
(execute increase_x)
(display (list "outer x after:  " x))
```

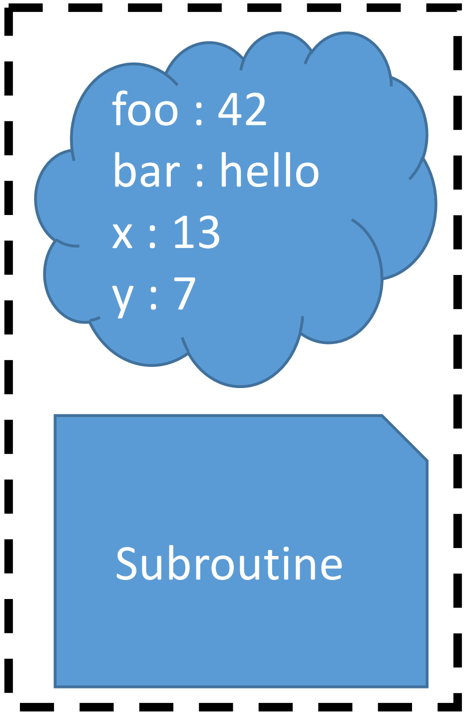
What is the output?

```
(outer x before: 1)
(inner x before: 20)
(inner x after:  20)
(outer x after:  2)
```

Motivation for Closures

- A subroutine is a general term for a procedure or a function
- Idea: Passing subroutines is allowed in many languages
  - Reference to a subroutine can be passed as a parameter
  - Subroutine has access to all active bindings in its scope
- Idea: *Referencing environment* of subroutine contains all the active bindings
  - If deep binding is used, referencing environment is created when subroutine is passed
  - If shallow binding is used, referencing environment is created when subroutine is called
- Closures are a construct found in languages with deep binding

Closure



- A *closure* consists of
  - A reference to a subroutine
  - A referencing environment
- Analogy: A program and its data.
- This is different from an object:
  - **object** = data + operations on the data
  - **closure** = subroutine (1 op) + data that it needs
- Idea: Closures can be used like objects
- Challenge: implementing closures when they can be returned by functions
  - When the subroutine is invoked, the scope it refers to may no longer exist
  - Scopes must be preserved for use in closures

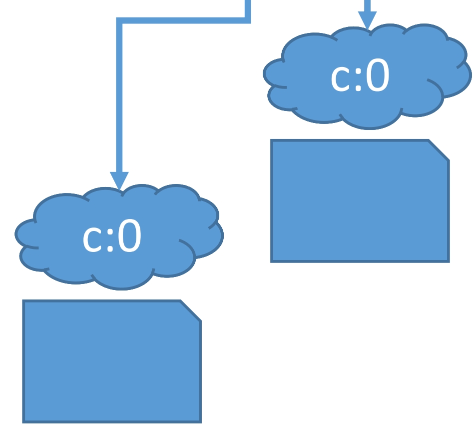
Closure Example

```
(define new-
counter
  (lambda ()
    (define c 0)
    (lambda ()
      (set! c (+ c
1))
      c
    )
  )
)
```

Closure Example

```
(define new-counter
  (lambda ()
    (define c 0)
    (lambda ()
      (set! c (+ c 1))
      c
    )
  )
)
```

```
> (define cnta (new-counter))
> (cnta)
1
> (cnta)
2
> (define cntb (new-counter))
> (cntb)
1
```

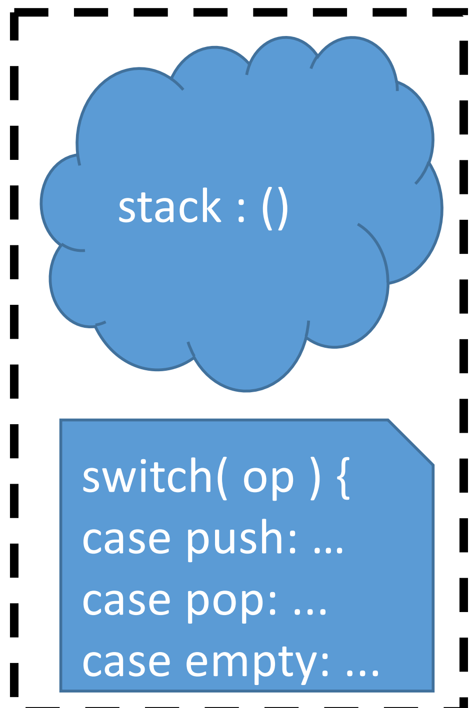


Example

```
(define new-stack (lambda ()
  (let ((stack ()))
    (lambda (op arg)
      (cond
        ((eq? op push)
          (set! stack (cons arg stack))
        )
        ((eq? op pop)
          (let ((top (car stack)))
            (set! stack (cdr stack))
            top
          )
        )
        ((eq? op empty)
          (null? stack)
        )
      )
    )
  )
)
```

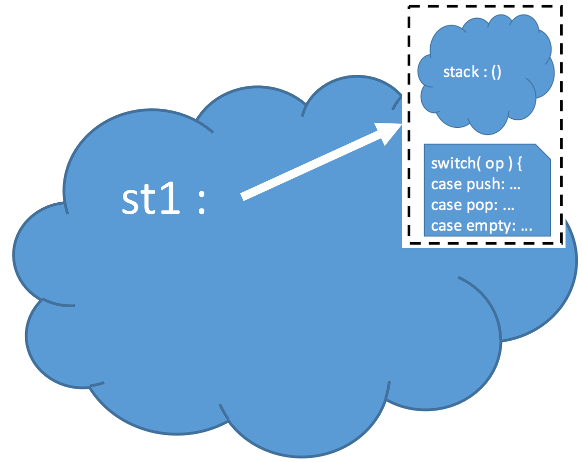
The evaluation of the **(let …)** returns the above lambda with the stack variable

A call to new-stack creates and returns the closure below.



```
(define new-stack (lambda ()
  (let ((stack ()))
    (lambda (op arg)
      (cond
        ((eq? op push)
          (set! stack (cons arg stack))
        )
        ((eq? op pop)
          (let ((top (car stack)))
            (set! stack (cdr stack))
            top
          )
        )
        ((eq? op empty)
          (null? stack)
        )
      )
    )
  )
)
```

Result

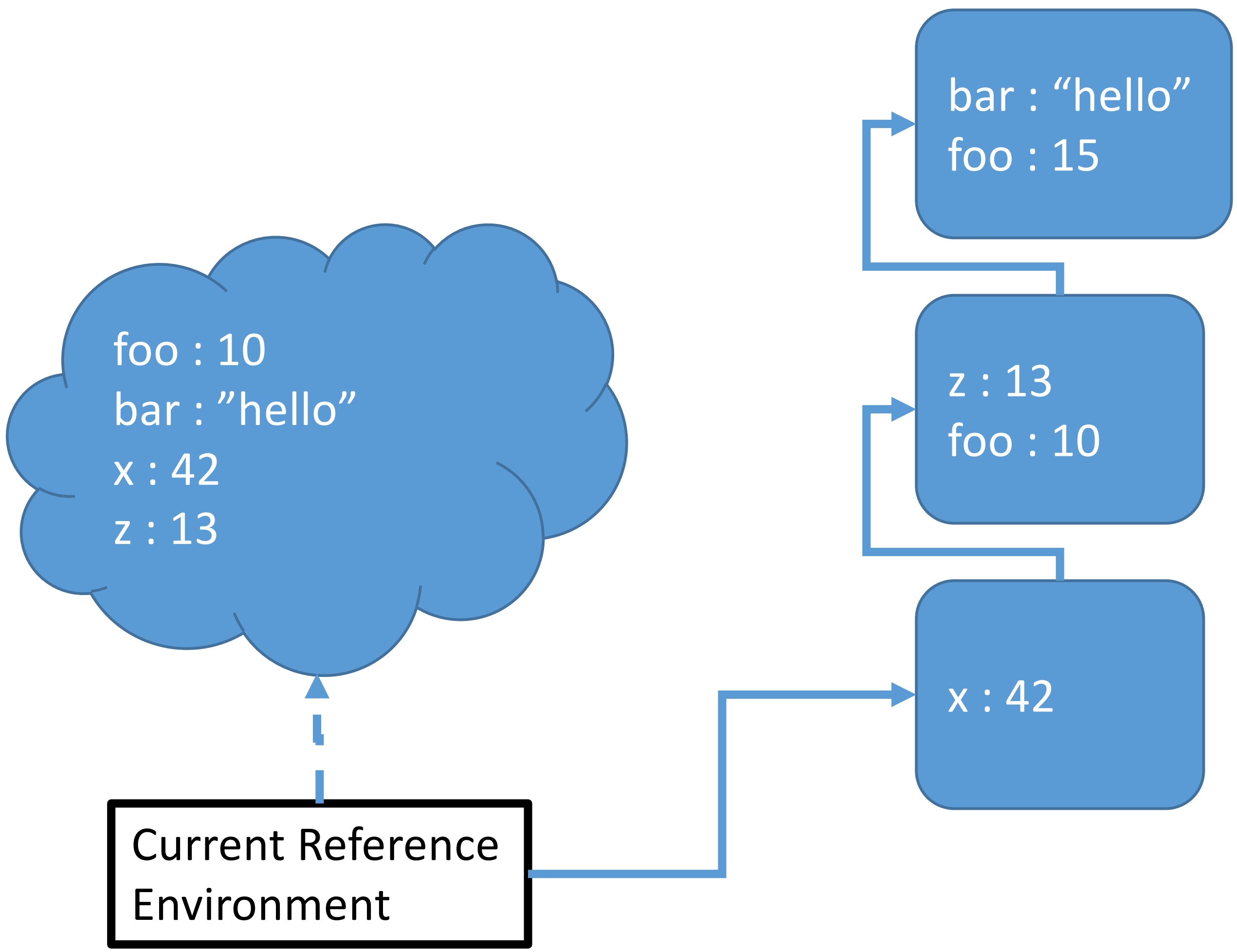



```
> (define st1 (new-stack))
> (st1 push 3)
> (st1 empty)
#f
> (st1 push 4)
> (st1 pop)
4
> (st1 pop)
3
> (st1 empty)
#t
```

Linked Referencing Environments

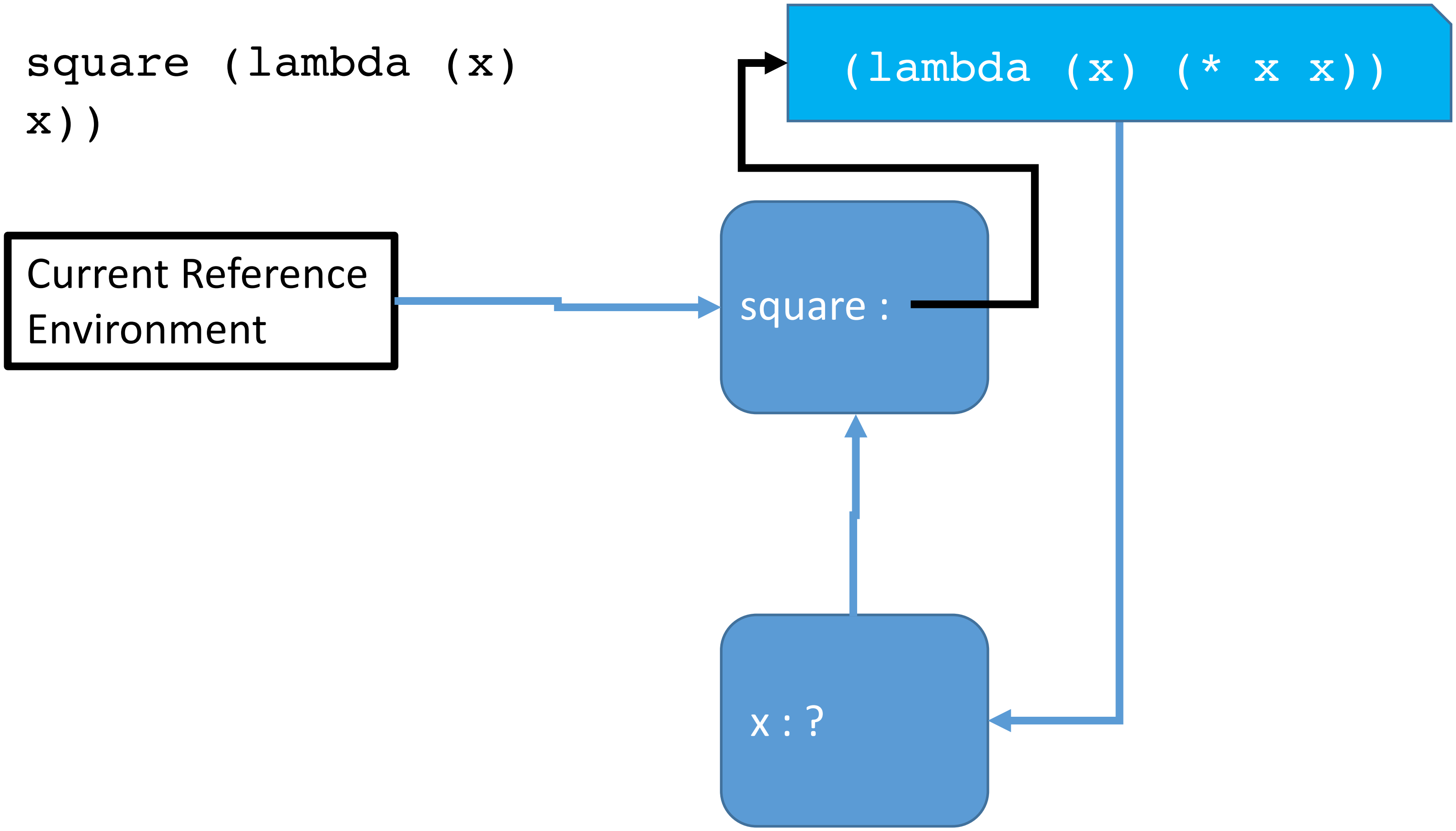
- A ***frame*** is a collection of variable-object bindings
- Frames can point to parent frames, resulting in a chain of frames
- A reference environment is a chain of frames starting with the most local frame
- Variable x in environment E is unbound if none of E’s frames binds x
- Otherwise x’s value is the value bound to it in the closest frame that contains a binding for x

Scheme Referencing Environments



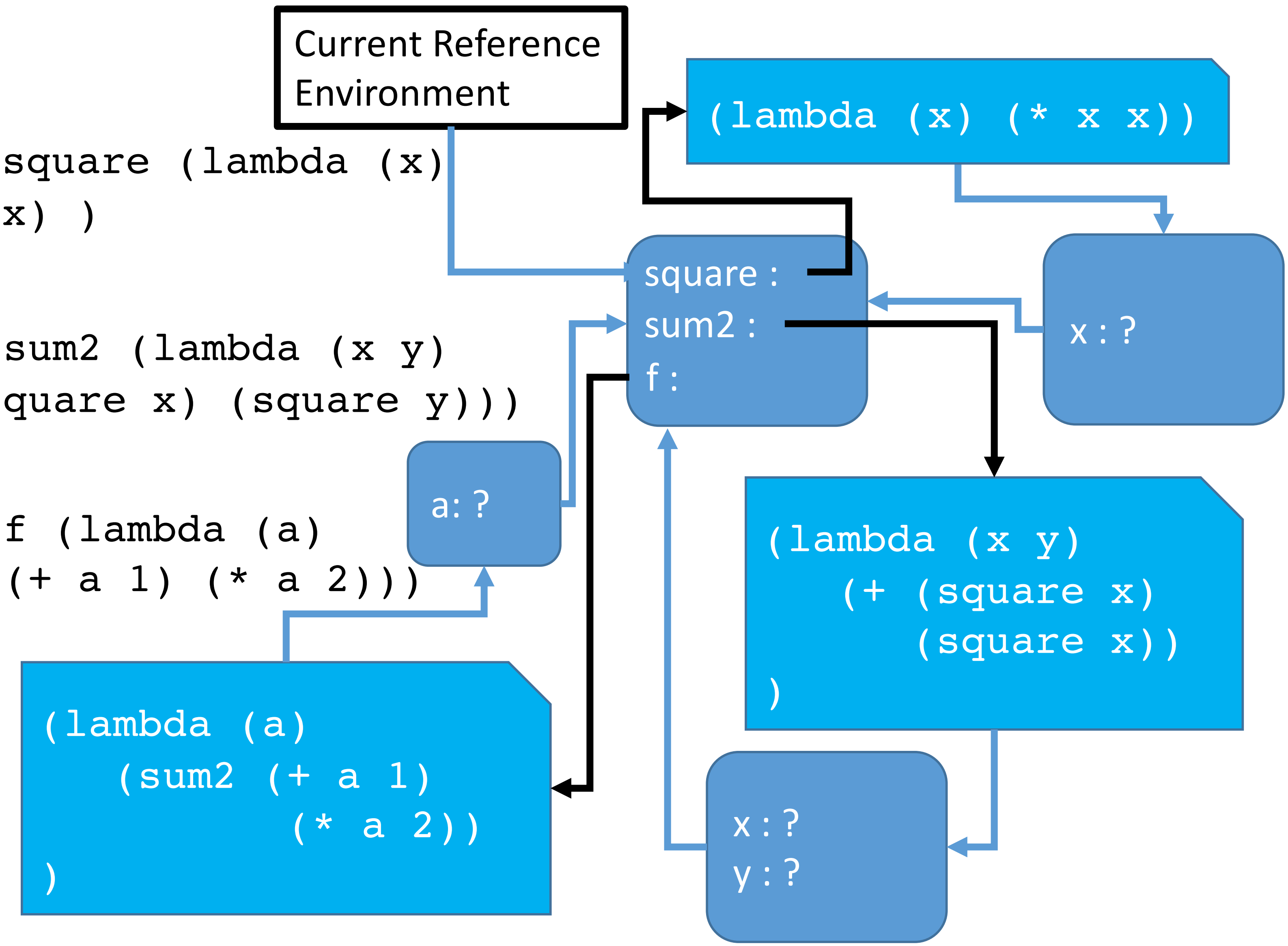
Referencing Environment Example

```
(define square (lambda (x)
   (* x x))
)
```



Referencing Environment Example

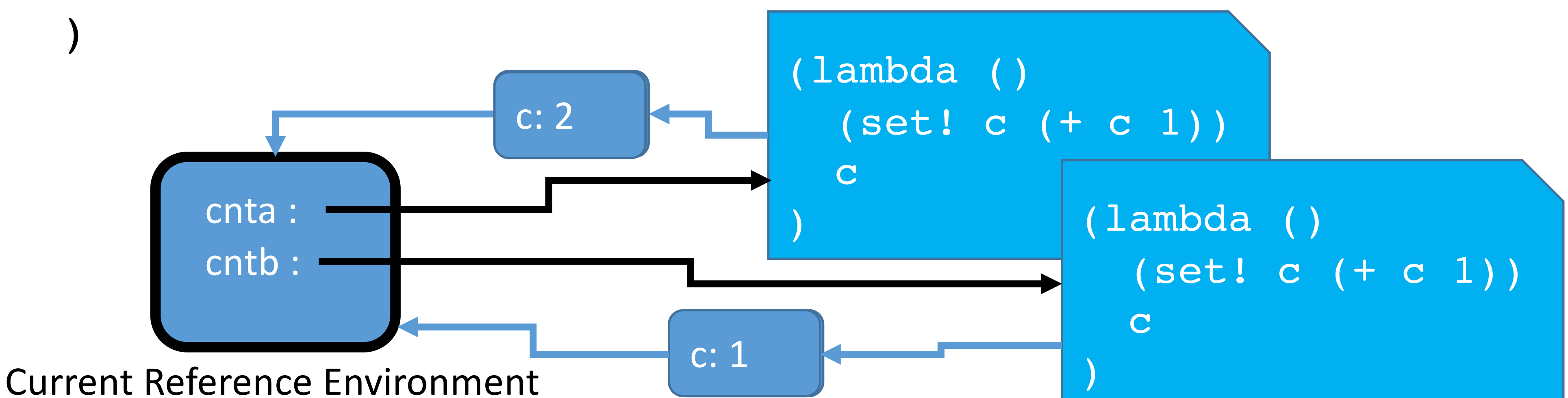
```
(define square (lambda (x)
   (* x x) )
)

(define sum2 (lambda (x y)
   (+ (square x) (square y)))
)

(define f (lambda (a)
  (sum2 (+ a 1) (* a 2)))
)
```



Simulating Objects with Closures

```
(define new-counter
  (lambda ()
    (define c 0)
    (lambda ()
      (set! c (+ c 1))
      c
    )
  )
)
```

```
> (define cnta (new-counter))
> (cnta)
1
> (cnta)
2
> (define cntb (new-counter))
> (cntb)
1
```



Discussion

- Closures seem esoteric, but they are very common in many programming langauges
  - Java (v8)
  - Scheme
  - Python
  - Most functional languages
  - Ruby
- Java used to use anonymous classes to create small listeners and callbacks, where closures are more appropriate
- Learn them, Use them!

Bindings in Scheme: `define` and `let`

- `(... (define x exp) fun1 fun2 ... funk )`
  All code after the define can see the binding of **x**
- `(let ((x exp1) (y exp2) (z exp3))`
  `fun1 fun2 ... funk )`
  Only the code following the bindings defined by `let` can see the bindings
- `(let* ((x exp1) (y exp2) (z exp3))`
  `fun1 fun2 ... funk )`
  Each binding becomes visible as soon as it is activated
- `(letrec ((x exp1) (y exp2) (z exp3))`
  `fun1 fun2 ... funk )`
  Bindings can become active in order immediately (used for declaring recursive functions)