Variables, Environments and Closures

Overview

We will

- Touch on the notions of variable extent and scope

- Introduce the notions of lexical scope and dynamic scope for variables

- Provide a simple model for variable environments in Scheme

- Show examples of closures in Scheme

Variables, free and bound

- In this function, to what does the variable *GOOGOL* refer?

  (define (big-number? x)

   ;; returns true if x is a really big number

   (> x GOOGOL))

- The **scope** of the variable X is just the body of the function for which it's a parameter.

Here, **GOOGOL** is a global variable

```
> (define GOOGOL (expt 10 100))
> GOOGOL
10000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
> (define (big-number? x) (> x GOOGOL))
> (big-number? (add1 (expt 10 100)))
#t
```

Which X is accessed at the end?

```
> (define GOOGOL (expt 10 100))
> GOOGOL
10000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000
0000000000000
> (define x -1)
> (define (big-number? x) (> x GOOGOL))
> (big-number? (add1 (expt 10 100)))
#t
```

Variables, free and bound

- In the body of this function, we say that the variable (or symbol) X is **bound** and GOOGOL is **free**

  (define (big-number? x)

  ; returns true if X is a really big number

  (> X GOOGOL))

- If it has a value, it has to be bound somewhere else

The let form creates local variables

> (let [(pi 3.1415)

(e 2.7168)]

(big-number? (expt pi e)))

#f

- The general form is (let <varlist> <body>)
- It creates a local environment, binding the variables to their initial values, and evaluates the expressions in <body>

Let creates a block of expressions

```
(if (> a b)
    (let (  )
          (printf "a is bigger than b.~n")
          (printf "b is smaller than a.~n")
           #t)
    #f)
```

Let is just syntactic sugar for lambda

(let [(pi 3.1416) (e 2.7183)]
  (big-number? (expt pi e)))

((lambda (pi e) (big-number? (expt pi e)))
  3.1416
  2.7183)

and this is how we did it back before ~1973

Let is just syntactic sugar for lambda

What happens here?

```
(define x 2)
(let [ (x 10) (xx (* x 2)) ]
    (printf "x is ~s and xx is ~s.~n" x xx))
```

x is 10 and xx is 4.

Let is just syntactic sugar for lambda

What happens here?

```
(define x 2)

( (lambda (x xx) (printf "x is ~s and xx is ~s.~n" x xx))
  10
  (* 2 x))
```

x is 10 and xx is 4.

Let is just syntactic sugar for lambda

What happens here?

```
(define x 2)

(define (f000034 x xx)
  (printf "x is ~s and xx is ~s.~n" x xx))
(f000034 10 (* 2 x))
```

x is 10 and xx is 4.

let and let*

- The let special form evaluates all initial value expressions, and then creates a new environ-ment with local variables bound to them, "in parallel"

- The let* form does this sequentially

- let* expands to a series of nested lets, e.g.

   (let* [(x 100)(xx (* 2 x))] (foo x xx)) expands to

   (let [(x 100)]
       (let [(xx (* 2 x))] (foo x xx)))

What happens here?

```
> (define X 10)
> (let [(X (* X X))]
      (printf "X is ~s.~n" X)
      (set! X 1000)
      (printf "X is ~s.~n" X)
      -1 )
???

> X
???
```

What happens here?

> (define X 10)

➢ (let [(X (* X X))]
     (printf "X is ~s\n" X)
     (set! X 1000)
     (printf "X is ~s\n" X)
     -1)

X is 100

X is 1000

-1

> X

10

What happens here?

> (define GOOGOL (expt 10 100))

> (define (big-number? x) (> x GOOGOL))

> (let [(GOOGOL (expt 10 101))]

    (big-number? (add1 (expt 10 100))))

???

What happens here?

> (define GOOGOL (expt 10 100))

> (define (big-number? x) (> x GOOGOL))

> (let [(GOOGOL (expt 10 101))]
    (big-number? (add1 (expt 10 100))))

#t

- The free variable GOOGOL is looked up in the environment in which the big-number? Function was defined!

- Not in the environment in which it was called

functions

- Note that a simple notion of a function can give us the machinery for

  - Creating a block of code with a sequence of expressions to be evaluated in order

  - Creating a block of code with one or more local variables

- Big idea in functional programming is to use functions to provide other familiar constructs (e.g., objects)

- And also constructs that are unfamiliar

Dynamic vs. Static Scoping

- Programming languages either use dynamic or static (aka lexical) [scoping](#)

- In a statically scoped language, free variables in functions are looked up in the environment in which the function is defined

- In a dynamically scoped language, free variables are looked up in the environment in which the function is called – i.e. at execution time!

History

- Lisp started out as a dynamically scoped language and moved to static scoping with [Common Lisp](#) in ~1980

- Today, fewer languages use only dynamic scoping, [Logo](#) and [Emacs Lisp](#) among them

- Perl and Common Lisp let you define some variables as dynamically scoped

Dynamic scoping

Here's a model for dynamic binding:
- Variables have a global stack of bindings
- Creating a new variable X in a block pushes a binding onto the global X stack
- Exiting the block pops X's binding stack
- Accessing X always produces the top binding

Special variables in Lisp

- Common Lisp's dynamically scoped variables are called special variables

- Declare a variable special using defvar

```
> (set 'reg 5)
5
> (defun check-reg () reg)
CHECK-REG
> (check-reg)
5
> (let ((reg 6)) (check-reg))
5
```

```
> (defvar *spe* 5)
*SPEC*
> (defun check-spe () *spe*)
CHECK-SPEC
> (check-spec)
5
> (let ((*spe* 6)) (check-spe))
6
```

Advantages and disadvantages

- + Easy to implement
- + Easy to modify a function's behavior by dynamically rebinding free variables

  (let ((IO stderr)) (printf "warning…"))

- - Can unintentionally shadow a global variable
- - A compiler can never know what a free variable will refer to, making type checking impossible

Closures

- Lisp is a **lexically scoped** language
- Free variables referenced in a function are looked up in the environment in which the function is defined

    Free variables are those a function (or block) doesn't create scope for

- A **closure** is a function that "remembers" the environment in which it was created
- An **environment** is just a collection of variable names and their values, plus a parent environment

Example: make-counter

- make-counter creates an environment using let with a local variable *C* initially 0

- It defines and returns a new function, using lambda, that can access and modify *C*

```
> (define (make-counter)
   (let ((C 0))
     (lambda ()
       (set! C (+ 1 C))
       C)))
> (define c1 (make-counter))
> (define c2 (make-counter))
```

```
> (c1)
1
> (c1)
2
> (c1)
3
> (c2)
???
```

```
> (define C 100)
> (define (mc) (let ((C 0)) (lambda () (set! C (+ C 1)) C)))
> (define c1 (mc))
> (define c2 (mc))
> (c1)
1
> (c2)
1
```
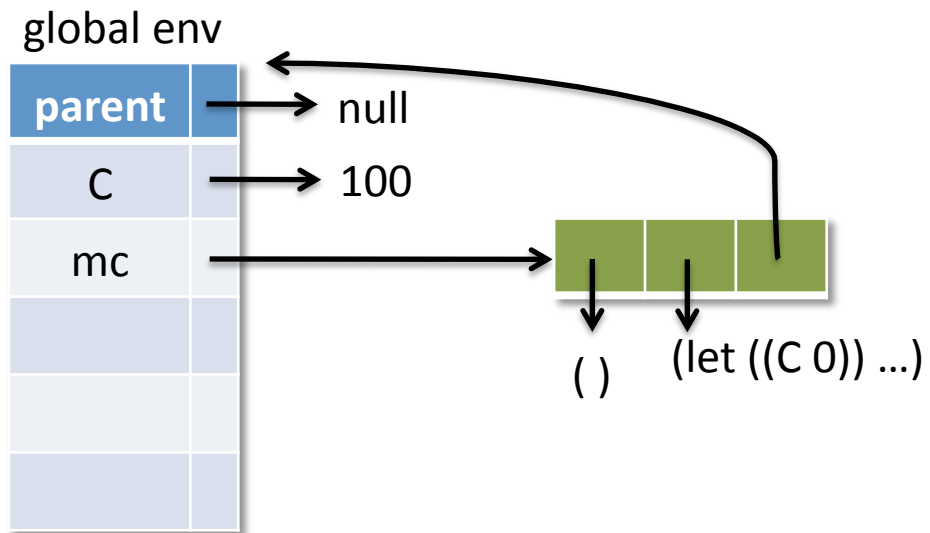
global env

parent	→ null

```
> (define C 100)
> (define (mc) (let ((C 0)) (lambda () (set! C (+ C 1)) C)))
> (define c1 (mc))
> (define c2 (mc))
> (c1)
1
> (c2)
1
```

global env

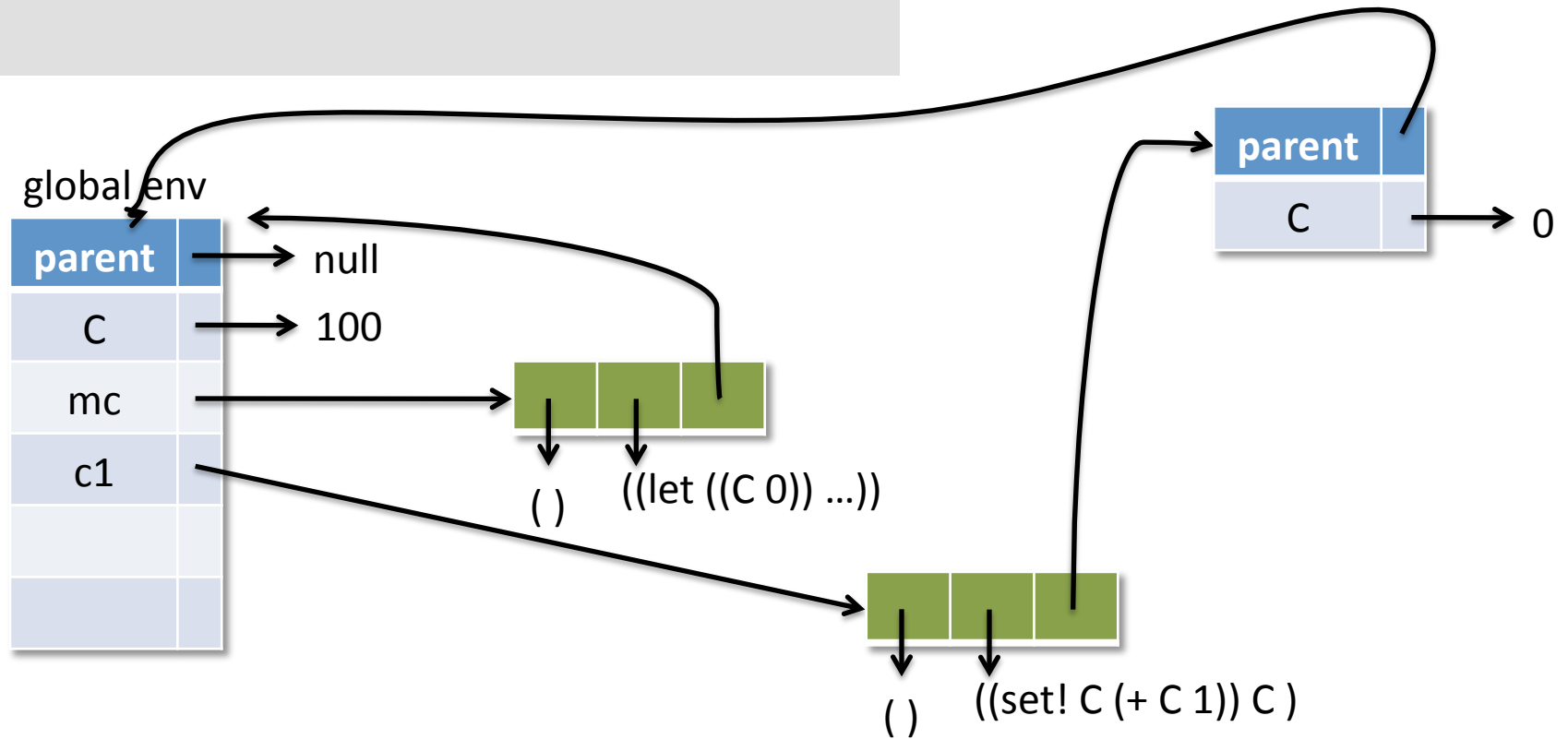| parent		→ null
C		→ 100

```
> (define C 100)
> (define (mc) (let ((C 0)) (lambda () (set! C (+ C 1)) C)))
> (define c1 (mc))
> (define c2 (mc))
> (c1)
1
> (c2)
1
```

global env

parent	→ null
C	→ 100
mc	

( )   (let ((C 0)) ...)

> (define C 100)
> (define (mc) (let ((C 0)) (lambda () (set! C (+ C 1)) C)))
> (define c1 (mc))
> (define c2 (mc))
> (c1)
1
> (c2)
1

global env

parent	
C	
mc	
c1	

null

100

( )
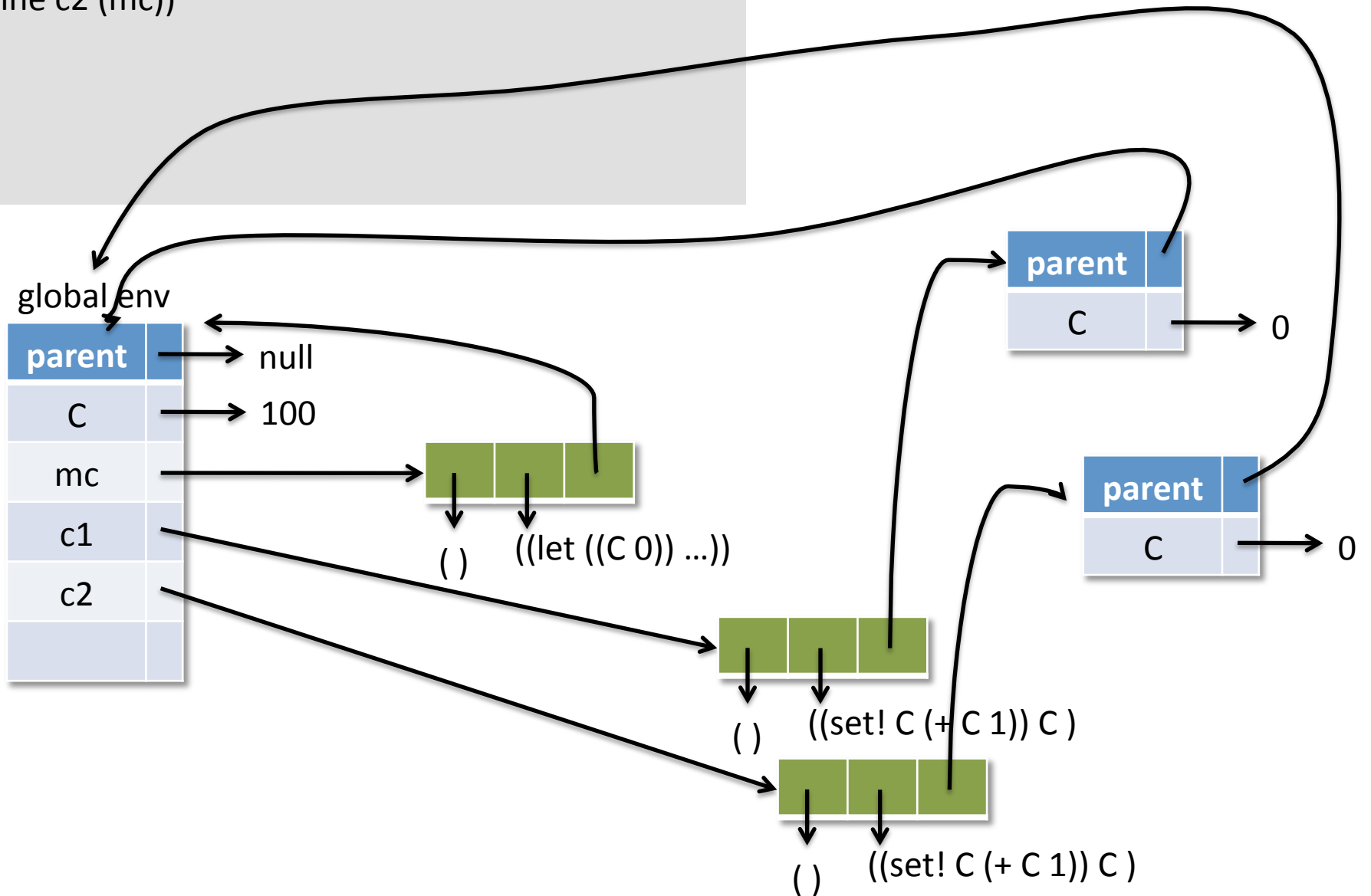
((let ((C 0)) ...))

parent

C

0

( )

((set! C (+ C 1)) C)

```
> (define C 100)
> (define (mc) (let ((C 0)) (lambda () (set! C (+ C 1)) C)))
> (define c1 (mc))
> (define c2 (mc))
> (c1)
1
> (c2)
1
```

global env

parent	
C	→ 100
mc	
c1	
c2	

parent → null

(() ((let ((C 0)) ...))

parent
| C | → 0 |

parent
| C | → 0 |

(() ((set! C (+ C 1)) C)

(() ((set! C (+ C 1)) C)

> (define C 100)
> (define (mc) (let ((C 0)) (lambda () (set! C (+ C 1)) C)))
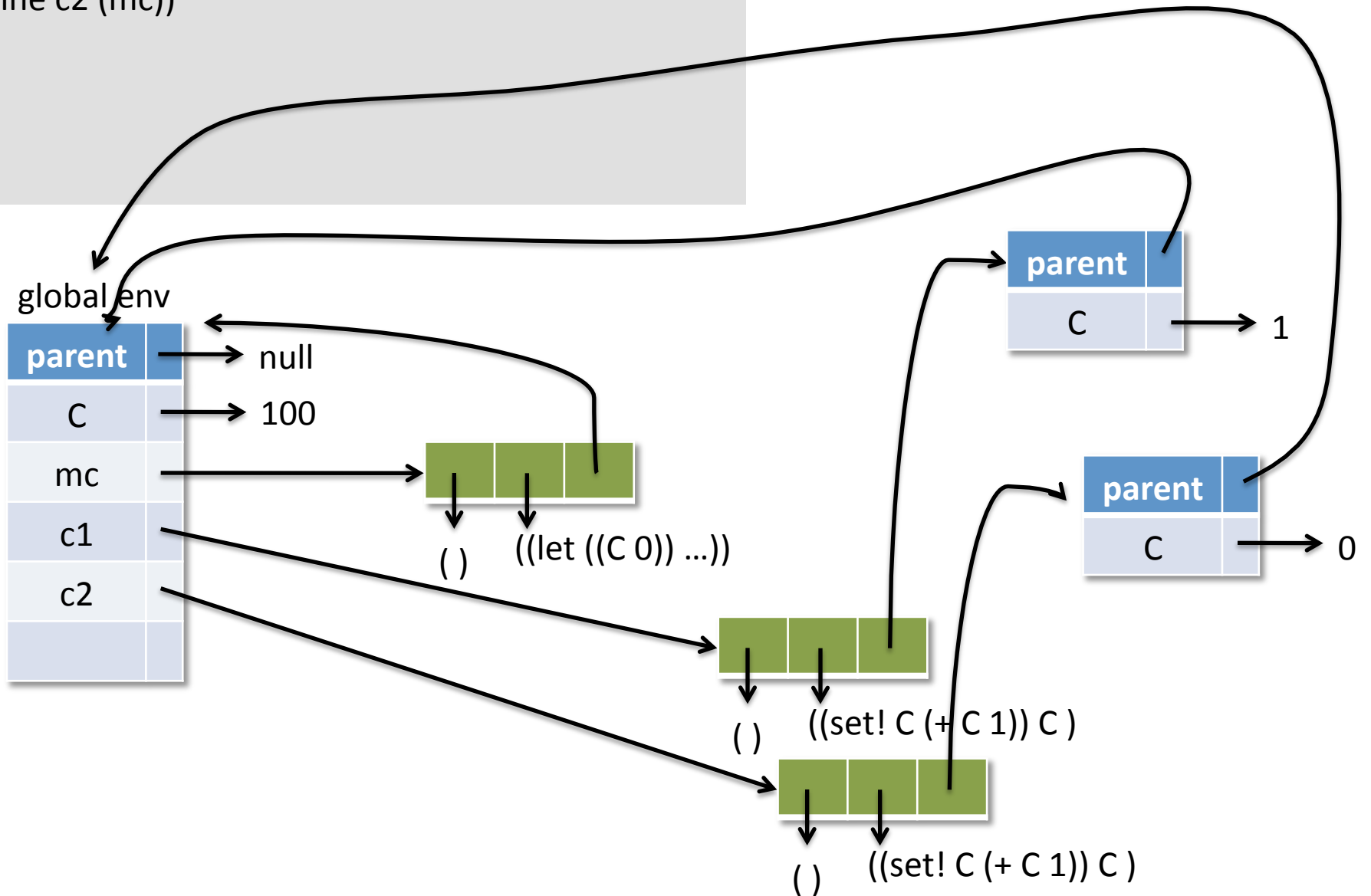> (define c1 (mc))
> (define c2 (mc))
> (c1)
1
> (c2)
1

global env

parent		→ null
C	→ 100
mc	
c1	
c2	

(()   ((let ((C 0)) ...)))

parent
| C | → 1 |

parent
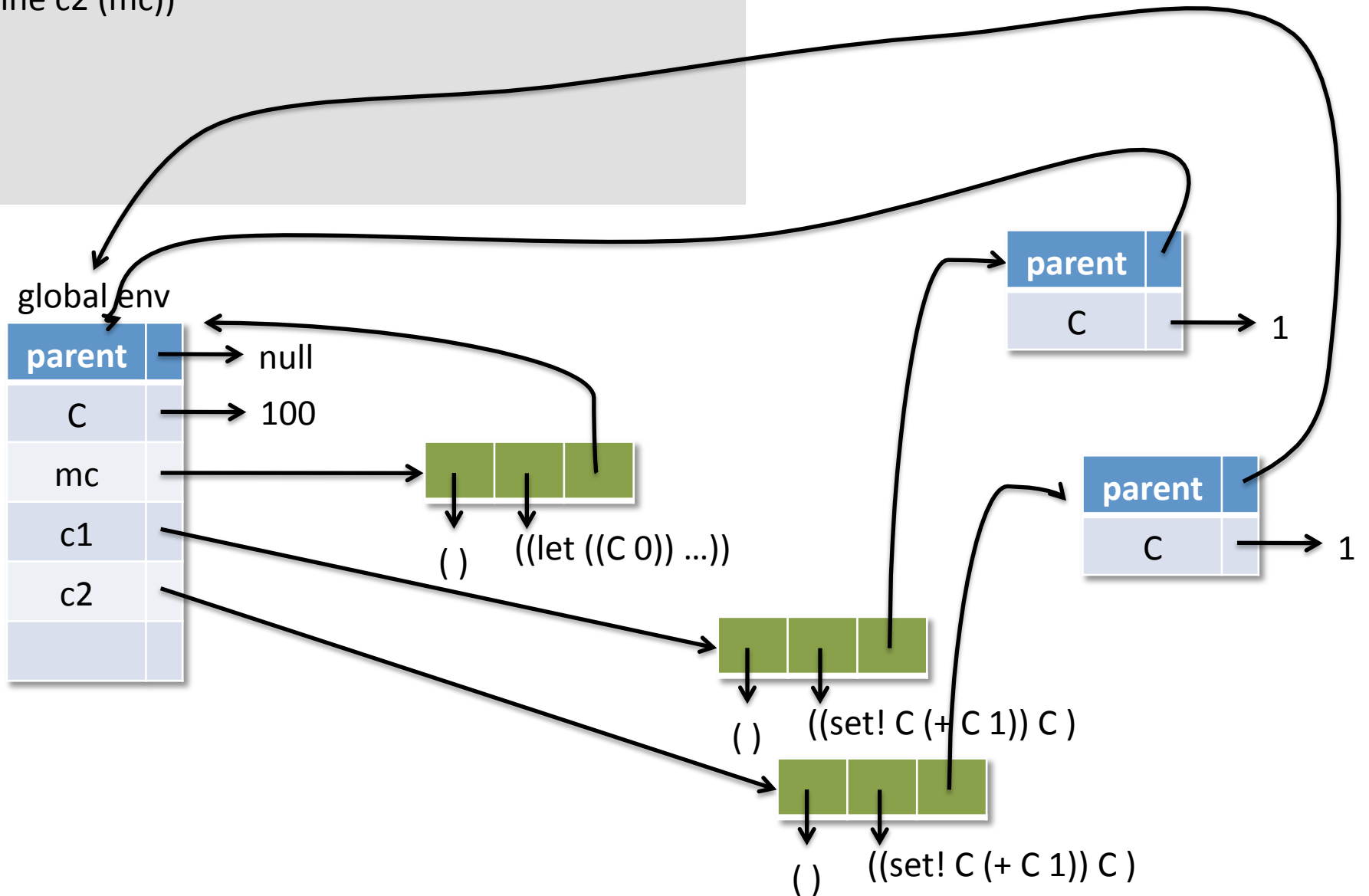| C | → 0 |

(()   ((set! C (+ C 1)) C))

(()   ((set! C (+ C 1)) C))

```
> (define C 100)
> (define (mc) (let ((C 0)) (lambda () (set! C (+ C 1)) C)))
> (define c1 (mc))
> (define c2 (mc))
> (c1)
1
> (c2)
1
```

global env

parent	
C	
mc	
c1	
c2	

null

100

(()) ((let ((C 0)) ...))

parent

C → 1

parent

C → 1

(()) ((set! C (+ C 1)) C)

(()) ((set! C (+ C 1)) C)

A fancier make-counter

Write a fancier make-counter function that takes an optional argument that specifies the increment

> (define by1 (make-counter))

> (define by2 (make-counter 2))

> (define decrement (make-counter -1))

> (by2)

2

(by2)

4

Optional arguments in Scheme

```
> (define (f (x 10) (y 20))
      (printf "x=~a and y=~a\n" x y))
> (f)
x=10 and y=20
> (f -1)
x=-1 and y=20
> (f -1 -2)
x=-1 and y=-2
```

Fancier make-counter

```
(define (make-counter (inc 1))
  (let ((C 0))
    (lambda ( ) (set! C (+ C inc)))))
```

Keyword arguments in Scheme

- Scheme, like Lisp, also has a way to define functions that take *keyword arguments*
  - (make-counter)
  - (make-counter :initial 100)
  - (make-counter :increment -1)
  - (make-counter :initial 10 :increment -2)
- Different Scheme dialects have introduced different ways to mix positional arguments, optional argu-ments, default values, keyword argument, etc.

Closure tricks

We can write several functions that are closed in the same environment, which can then provide a private communication channel

```
(define foo #f)
(define bar #f)

(let ((secret-msg "none"))
 (set! foo
   (lambda (msg)
     (set! secret-msg msg)))
 (set! bar
   (lambda () secret-msg)))

(display (bar)) ; prints "none"
(newline)
(foo "attack at dawn")
(display (bar)) ; prints "attack at dawn"
```

Summary

- Scheme, like most modern languages, is lexically scoped

- Common Lisp is by default, but still allows some variables to be declared to be dynamically scoped

- A few languages still use dynamic scoping

- Lexical scoping supports functional program-ming & powerful mechanisms (e.g., closures)

- More complex to implement, though