Assignment (`set!`) and Procedures with State

Lecturer:
Professor Albert R. Meyer

October 21, 1997

Use and Mis-Use of Set!

Functional programming style

```
(define (factorial n)
  (define (iter product counter)
    (if (> counter n)
        product
        (iter (* counter product)
              (+ counter 1))))
  (iter 1 1))
```

Imperative programming style:
"Do this; do that"

```
(define (factorial n)
 (let ((product 1)
       (count 1))
  (define (iter)
    (if (> count n)
        product
        (begin
         (set! product (* count product)) ;A
         (set! count (+ count 1))         ;B
         (iter))))
   (iter)))
```

Imperative style NOT CALLED FOR HERE.

Imperative programming style -- BUGGY!

```
(define (factorial n)
 (let ((product 1)
       (count 1))
  (define (iter)
    (if (> count n)
        product
        (begin
         (set! count (+ count 1))          ;B
         (set! product (* count product)) ;A
         (iter))))
   (iter)))
```

Implications of Mutation

- Must worry about **time** and **change**: order of evaluation matters!
- Variables no longer stand for values:
  - Become **places** whose contents may change.
- Must worry about **identity**: lose **referential transparency**

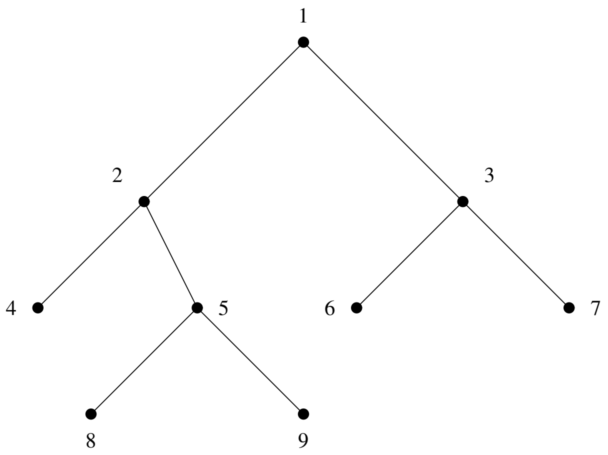
```
(= <expr> <expr>) ==> #f

   if <expr> is (count) (or (deposit 1) below).
```

- Great benefit: can **avoid copying** a big thing; make small change to big thing instead.
- Can help coordinate computations.
- Natural for modeling objects/systems with **state**.

Binary Tree Search

(purely functional)

```
(define (depth-first-tree-search label tree)
  (and (not (empty? tree))
       (or (same-label? label (label-of tree))
           (depth-first-tree-search
            label (left-branch tree))
           (depth-first-tree-search
            label (right-branch tree)))))
```




```
(define tree
  '(1
    (2
     (4 () ()) (5 (8 () ()) (9 () ())))
    (3
     (6 () ()) (7 () ()))))
```

Recursion works because:

SUBTREES ARE SMALLER

Cyclic Graph Search

Do tree-search but MARK! visited nodes to avoid cycling in place:

```
(define (depth-first-graph-search label graph)
  (and (not (empty? graph))
       (or (same-label? label (label-of graph))
           (begin
             (MARK! graph)
             (or
              (and
                (unmarked? (left-branch tree))
                (depth-first-graph-search
                 label
                 (left-branch tree)))
              (and
                (unmarked? (right-branch tree))
                (depth-first-graph-search
                 label
                 (right-branch tree)))))))))
```

Recursion works because:

NUMBER OF UNMARKED NODES GETS SMALLER

Imperative style IS CALLED FOR HERE. Marks coordinate different recursive calls. Doing cyclic graph search correctly in functional style is messy.

Deposit Bank Account

```
(define balance 100)

(define (deposit amount)
  (set! balance (+ balance amount))
  balance)

(deposit 0)  ==>  100

(deposit 5)  ==>  105

(deposit 5)  ==>  110

(deposit 0)  ==>  110
```

Hide the State Variable

```
(define new-deposit
  (let ((balance 100))
    (lambda (amount)
      (set! balance (+ balance amount))
     balance)))
```

balance is now PRIVATE to new-deposit.

Making Deposit-Accounts

```
(define (make-account balance)
  (lambda (amount)
    (set! balance (+ balance amount))
   balance))

(define deposit1 (make-account 100))

(define deposit2 (make-account 100))

(deposit1 5)  ==> 105

(deposit1 5)  ==> 110

(deposit2 5)  ==> 105
```

Multifunction Account

In message passing style:

```
(define (make-account balance)

  (define (withdraw amount)
    (let ((new-balance (- balance amount)))
       (if (< new-balance 0)
           "insufficient funds"
           (begin
             (set! balance new-balance)
             new-balance))))

  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)

  (define (show)
    balance)

  (define (dispatch m)
    (cond ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          ((eq? m 'show-balance) show)
          (else (error "MAKE-ACCT bad op" m))))

  dispatch)   ;the message-passing object
```

(This slide was not actually shown in lecture)

Mutable Data Structures

Can simulate mutable lists with a message passing object:

```
(define (cons x y)
  (define (set-x! v) (set! x v))
  (define (set-y! w) (set! y w))
  (define (dispatch m)
    (cond ((eq? m 'car) x)
          ((eq? m 'cdr) y)
          ((eq? m 'set-car!) set-x!)
          ((eq? m 'set-cdr!) set-y!)
          (else (error "CONS -- bad op" m))))
  dispatch)

(define (car pair)
 (pair 'car))

(define (set-car! pair val)
  ((pair 'set-car!) val))
```