Address book

More efficient update

```
;; change-number3:  symbol  number -> boolean
;; Purpose: changes an existing phone number in the address book
;; Effect: modifies entry's phone number in place
(define (change-number3 who phone)
   local [(define aloe (filter (lambda(x)(symbol=? who (entry-name x)))
                                   address-book))]
          (cond [(empty? aloe)  false]
                  [(cons?  aloe)
                   (begin
                      (set-entry-number! (first aloe)  phone)
                       true)] )))
```

Interface changed, too

- For no extra cost, we can return false on failure
- Does not add new entries

Administrative Notes

Extra credit homework

- Due Friday
- Counts as 10 point extra credit toward homework grade

Third exam

- Available Friday
- Due April 24 at 5 pm
- Closed notes, closed book
- Material since second exam

Back to the address book

More efficient update

```
;; change-number3:  symbol  number -> boolean
;; Purpose: changes an existing phone number in the address book
;; Effect: modifies entry's phone number in place
(define (change-number3 who phone)
   local [(define aloe (filter (lambda(x)(symbol=? who (entry-name x)))
                                    address-book))]
          (cond [(empty? aloe)  false]
                  [(cons?  aloe)
                   (begin
                     (set-entry-number! (first aloe)  phone)
                      true)] )))
```

Changes contents of (first aloe)

Deeper question: when are two structures the same

Interface changed, too

- For no extra cost, we can return false on failure
- Does not add new entries

Identity among structures

If we type

```
(define x (make-entry 'keith  7133486013))
(define y (make-entry 'keith 7133486013))
```

Are x and y the same?

- What does this question mean?
  - → Are the structures identical (same value, same behavior)?
  - → Are the structures implemented with the same object?
- They have the same shape
- They have the same values in the same places

```
(= (entry-number x) (entry-number y))  ⇒ true
(symbol=? (entry-name x) (entry-name y)) ⇒ true
```

What about x and y?

Identity among structures

Can we tell if x and y are the same structure?

- Scheme’s equal? predicate tests equality

```
(define x (make-entry ‘keith 7133486013))
(define y (make-entry ‘keith 7133486013))
(equal? x y)  ⇒ true
```

Now, try

```
(set-entry-number! y 12)
(equal? x y) ⇒ false
```

This raises a number of questions

- Does set-entry-number! change y?
- Does set-entry-number! change x?*
- How do we model (& understand) x and y?

Identity among structures

Equality operators

```
(equal?  x  y)
```

Tests whether two values have the same structure & values

- Checks value in each position in structure
- Performs check recursively

This provides an *extensional* notion of equality

- Starts from the structure of each argument
- Equality based on identical structure & identical value

⇒ DrScheme

Identity among structures

Equality operators

(eq? x y)

Tests whether two names refer to the same object

- Returns true if they refer to the same object
- Returns false if they refer to different objects
  - → Even if the objects are equivalent (equal?)

This provides an *intensional* notion of equality

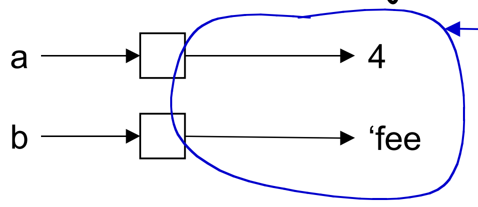
- Objects are identical if & only if they have the same implementation
- Equality based on where in memory the values reside

⇒ *DrScheme*

Identity

Can we make this more concrete?

- a and b are Scheme objects



Each Scheme object has a value

(define a 4)
(define b 'fee)

Gives a and b the appropriate values

Value must be different than the name for set! to work

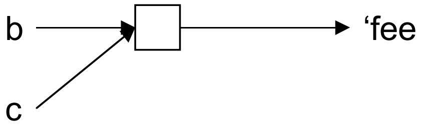
Identity

Can we make this more concrete?

- Some objects have unique implementations

```
(define b ‘fee)
(define c ‘fee)
(eq?  b c) ⇒ true
```

This tells us something about the implementation



What about numbers? ⇒ *DrScheme*

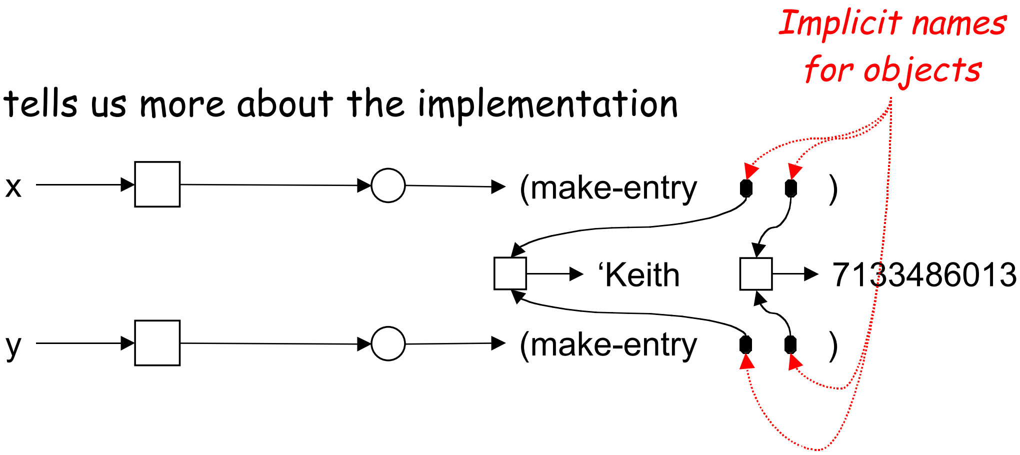
Identity

Can we make this more concrete?

- Structures are Scheme objects

```
(define x  (make-entry ‘Keith 7133486013))
(define y  (make-entry ‘Keith 7133486013))
(eq?  x y) ⇒ false
```

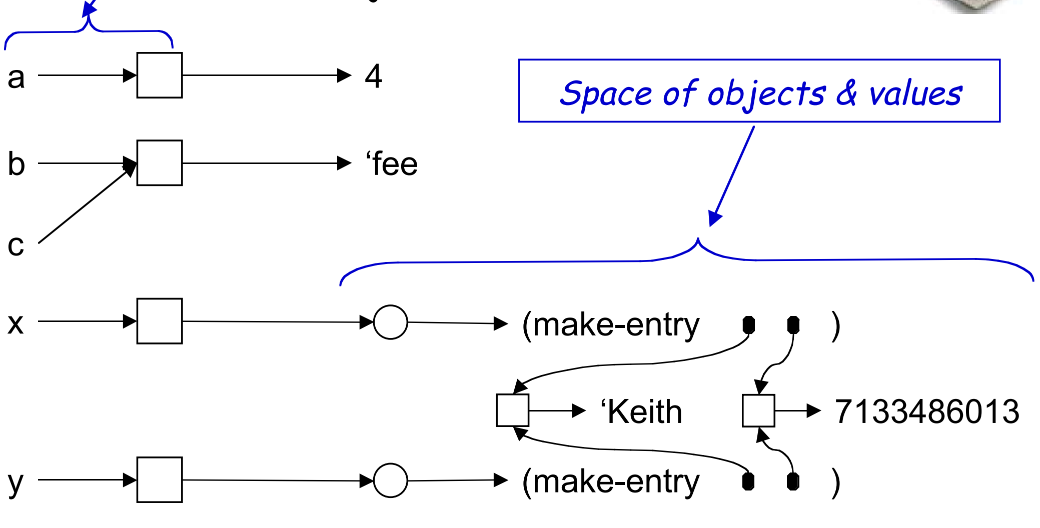
This tells us more about the implementation



We can show this in DrScheme

Identity

We've studied a few objects



Mapping from names to objects

Space of objects & values

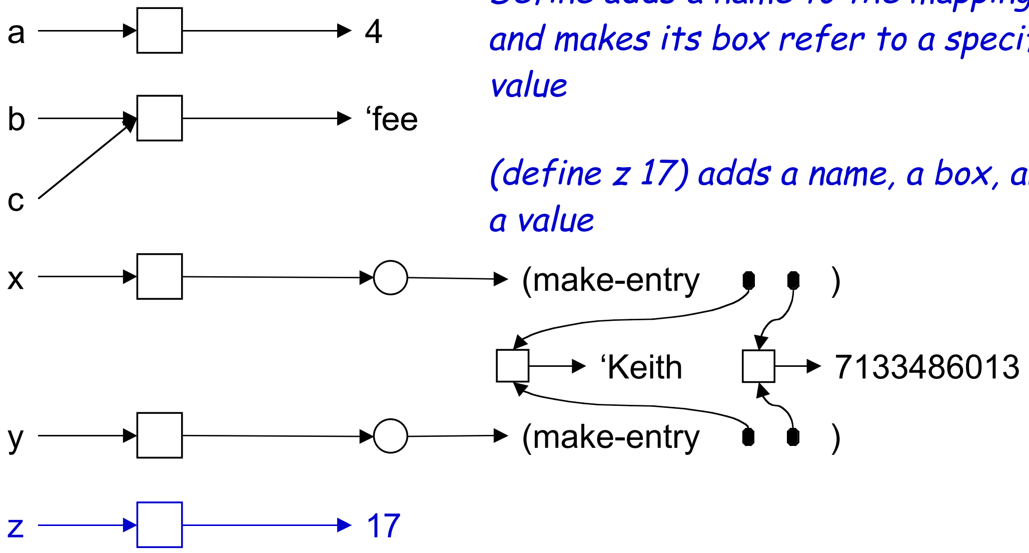
Can they teach us about <u>define</u>, <u>set!</u>, and <u>set-structure!</u> ?

Identity

We've studied a few objects

Define adds a name to the mapping and makes its box refer to a specific value

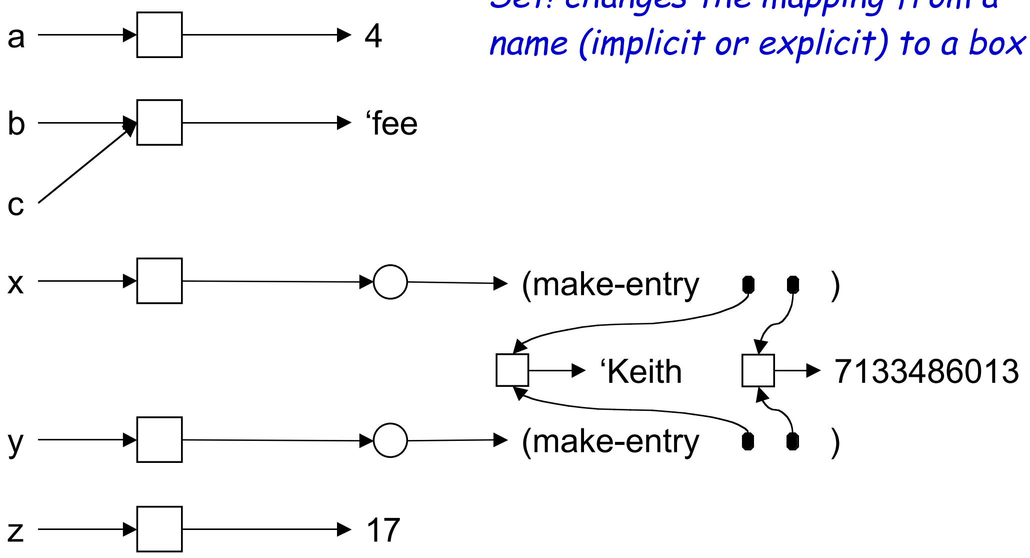
(define z 17) adds a name, a box, and a value



Identity

We've studied a few objects

Set! changes the mapping from a name (implicit or explicit) to a box

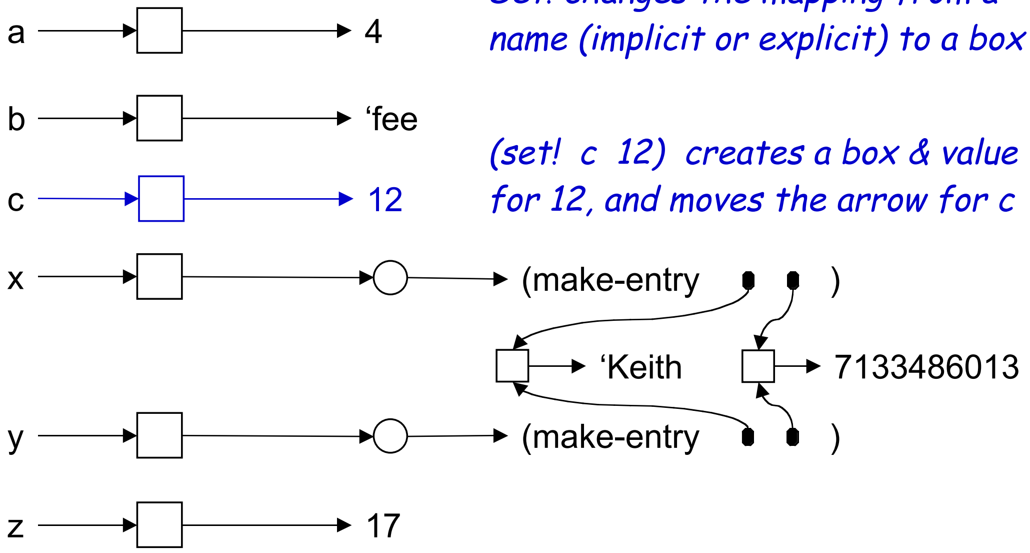


Identity

We've studied a few objects

Set! changes the mapping from a name (implicit or explicit) to a box

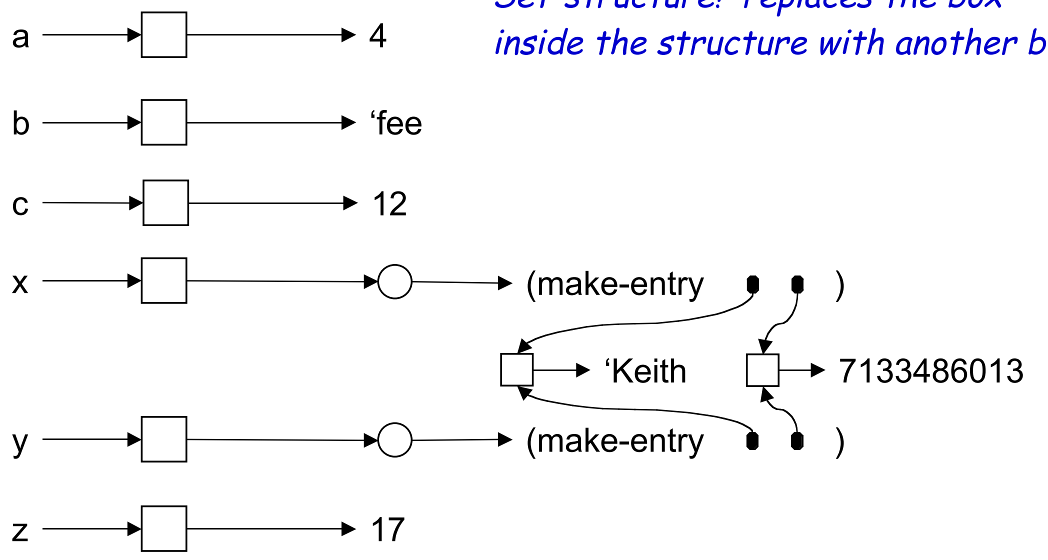
(set! c 12) creates a box & value for 12, and moves the arrow for c



Identity

We've studied a few objects

Set-structure! replaces the box inside the structure with another box

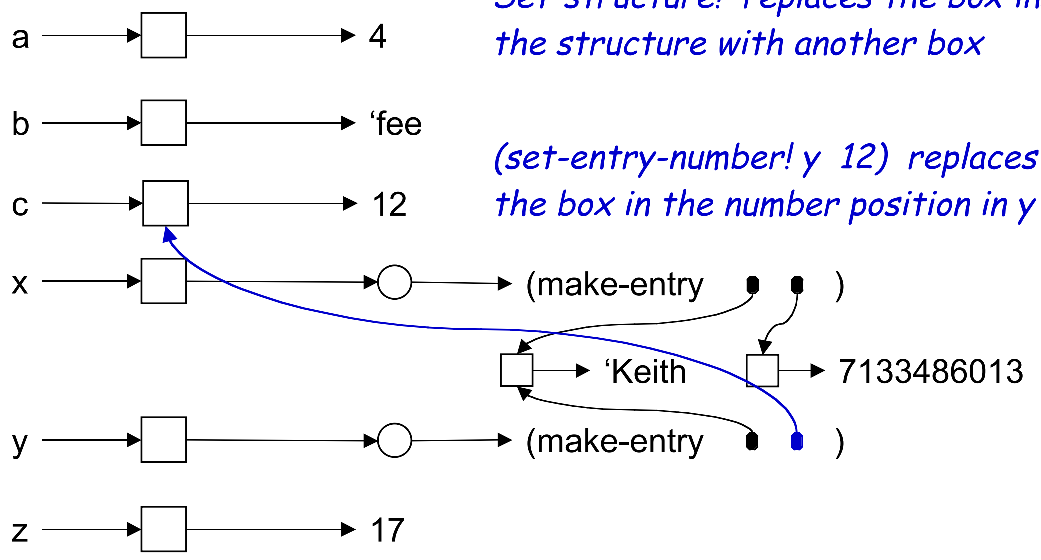


Identity

We've studied a few objects

Set-structure! replaces the box inside the structure with another box

(set-entry-number! y 12) replaces the box in the number position in y



Identity

Now, we can explain equal? and eq?

- equal? is a a recursive program
  - → Tests, at each level, for structure & value identity
  - → Must traverse entire structure
  - → Returns true if & only if all components are identical

- eq? checks if the arguments refer to the same box
  - → No notion of value or structure
  - → Simply looks at the box
  - → Returns false for different boxes, even if arguments are actually “equal?”